ECE 252: Systems Programming and Concurrency Fall 2019

Lecture 23 — Condition Variables, Monitors, Atomic Types

Jeff Zarnett *2019-08-10*

Condition Variables

Condition variables are another way to achieve synchronization. Rather than designating critical areas and enforcing rules about how many threads may be in the critical area, 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 (like interrupts from hardware when there is new data to read rather than polling to check periodically or constantly).

An event is similar to, but slightly different from, a counting semaphore. 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 [HZMG15].

Consider the condition variable functions:

```
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 );
```

To initialize a `pthread_cond_t` (condition variable type), the function is `pthread_cond_init` and to destroy them, `pthread_cond_destroy`. As with threads and a mutex, we can initialize them with attributes, and there are functions to create and destroy the attribute structures, too. But the default attributes will be fine, at least in this course.

Condition variables are always used in conjunction 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. 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 [Bar14]. Obviously, failing to lock and unlock the mutex before and after using the condition variable, respectively, can result in problems.

In addition to the expected `pthread_cond_signal` function that signals a provided condition variable, there is also `pthread_cond_broadcast` that 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.

Let us now examine a code example from [Bar14] which has been cut down a bit to make it more readable:

```
#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 );
}

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 );
}
```

It should be noted that if a thread signals a condition variable that an event has occurred, but no thread is waiting for that event, the event is "lost". Because an event that takes place when no thread is listening is simply lost, it is (almost always) a logical error to signal or broadcast on a condition variable before some thread is waiting on it. This is sometimes called the "lost wakeup problem", because threads don't get woken up if they weren't waiting for this. Maybe it's not as dire as that, however, because sometimes an event is broadcast but nobody is interested and that's fine.

The condition variable with broadcast can be used to replace some of the synchronization constructs we've seen already. Consider the barrier pattern from earlier. There are n threads and we wait for the last one to arrive. Then the last thread signals to unlock the barrier and then each thread calls post to unblock the next thread until all of them are through. This is a lot of calls and maybe it would be better to make it a broadcast instead. Remember the simple barrier (one phase rather than two), on the left, and then the condition variable on the right:

```
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? What's important to remember is

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

Okay, let's think about how to put that to use in an actual code example. Assume that `count`, `lock` and `cv` are initialized correctly as they should be.

```
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 );
}
```

If every thread calls `barrier()` before going on to whatever is next, they will wait until the last thread arrives – as they should. Broadcast wakes up all the other threads. It's possible to use a for loop to signal on the condition variable n times but that mostly defeats the purpose of using the condition variable instead of a regular semaphore, doesn't it?

Monitors

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

The objective of the monitor is to make it so that programmers do not need to code the synchronization part directly, making it less likely a programmer makes an error. There are numerous steps where you can get it wrong in a typical program. Look at:

```
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 );
}
```

In this case, there is control flow that could lead to exiting this function `foo` without unlocking the mutex `l`. Sure, when I cut the remaining code out and just replace it with some comments so the control flow that causes the problem here is obvious, but in a real code example the function in question is likely to be in the hundreds of lines and it's easy to overlook something.

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`, adding it after the access modifier keyword (public, private, etc) in the function definition, and 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: a set of threads waiting for the lock to become available [SGG13].

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

```
}
```

Note that in Java we can make a method `synchronized` or 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 and abstract away some of the details of mutual exclusion.

Monitors don't have to be written in Java (or a similar language) but they very commonly appear in object-oriented programming languages because of the goal of packaging up your data with the associated synchronization constructs looks a lot like Object-Oriented Programming: package up the data with its associated functions. Moreover, in an object-oriented language thanks to things like encapsulation (i.e., `private` variables), it's somewhat easier to "force" all access to be through the appropriate monitor.

Atomic Types

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

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

While is is fully correct, if this happens frequently there is a lot of locking and unlocking on the same mutex, just to do the increment. So there's a fair amount of overhead on this. 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? We can!

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.

The kernel itself contains an atomic type, `atomic_t`, but this is not intended for use outside of the kernel.

The following function listings are an overview of the atomic operations, from [FSF19]. These are, however, `glibc` specific, and not necessarily available in a general system. In the C11 (2011) standard, atomic types were finally introduced as part of the language specification itself. But before this we just had implementation-specific options. In the specification, we see `type` as the type, but that's of course not a real 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, if it's an integral type or a pointer.

The following function is used to assign a new value, and returns the old value. However, the name is unfortunate, because it conflicts with the hardware instruction test-and-set which we discussed earlier. But this atomically sets the value of the variable as expected:

```
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 );
```

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 );
```

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 have mostly tried to avoid relying on anything that is implementation-specific behaviour... 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. And that would be a bit more portable as well. But for true portability you will need to use C11 or a semaphore/mutex.

Atomic operations are helpful for scenarios like a single variable being modified and read. But atomic operations are not always ideal. Consider this:

```
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 );
```

Although the set of each of `x` and `y` is atomic, the operation as a whole is not. The write of `x` could succeed and then a read of both in a different thread could take place before the write of `y`, meaning that a reader would see (25, 0) when that's probably not valid. Similarly, the state could be totally corrupted if another thread did atomic writes of (10, 15) in between the two, leading to a final state of (10, 30).

When a number of writes need to take place as a "package", then a mutex type is the appropriate choice.

Spinlocks. 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. The implementation is an integer that is checked by a thread; if the value is 0, the thread can lock it (set the value to 1) and continue; if it is nonzero, it constantly checks the value until the value becomes 0. As you know, this is very inefficient; it would be better to let another thread execute, except in the circumstances where the amount of time waiting on the lock might be less than it would take to block the process, switch to another, and unblock it when the value changes [Sta14].

```
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. This is implemented as a 24-bit reader counter and an unlock flag, with the meaning defined as follows [Sta14].

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.

There are further additional details related to use of spinlocks, which can of course be explored by reading the Linux kernel documentation.

References

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: `https://computing.llnl.gov/tutorials/pthreads/`.

[FSF19] Inc. Free Software Foundation. Built-in functions for atomic memory access, 2019. Online; accessed 3-July-2019. URL: `https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`.

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.