ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 18 — Condition Variables, Monitors, Atomic Types

Jeff Zarnett *2018-10-06*

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]. 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".

Condition variables are also supported in pthreads. To create a `pthread_cond_t` (condition variable type), the function is `pthread_cond_init` and 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. But the default attributes will be fine.

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. 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. It's this "broadcast" idea that makes the condition variable more interesting than the simple "signalling"

Condition Variable Example. Let us now examine a code example from [Bar14]:

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t) {
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /* Check the value of count and signal waiting thread when condition is
```

```
    reached.  Note that this occurs while mutex is locked. */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
       my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t) {
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /* Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning. */
  pthread_mutex_lock(&count_mutex);

  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count);
    printf("watch_count(): thread %ld Updating the value of count...\n", my_id,count);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
          NUM_THREADS, count);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);
```

```
}
```

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.

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*, where P waits until Q leaves the monitor (or another condition); or *signal-and-continue* where P continues and Q waits for P to leave the monitor or another condition. 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 [SGG13].

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 the details of mutual exclusion.

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 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. 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 [Sta14].

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

The following table gives an overview of the atomic integer and bitmap operations, from [Lov05]:

Function	**Description**
`atomic_t 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`

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 {
  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 );
```

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/`.

[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.

[Lov05] Robert Love. *Linux Kernel Development, 2nd Edition*. Sams Publishing, 2005.

[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.