14-2 — Monitors

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

What's Wrong With You?

As we have seen so far, using semaphores correctly is not straightforward.

For example, changing the order of `wait` and `post` operations can easily lead to deadlock.

Semaphores serve a dual purpose: they are used for both mutual exclusion and scheduling constraints.

Who Said What?

Dijkstra himself noted this:

> *During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.*

Monitors

Uh... wait... this isn't the right kind.

What's a Monitor?

A monitor consists of a `mutex` (lock) and at least one `condition variable` (CV).

Multiple CVs can be associated with the same mutex, but not vice versa.

The mutex is used for mutual exclusion, and CVs are used for scheduling constraints.

We will next look into pthread mutex and CVs.

Mutex Syntax

While it is possible, of course, to use a semaphore as a mutex, a more specialized tool for this task is the pthread mutex.

In fact, it’s generally good practice to use the more specialized tool.

Mutex Syntax

The structure representing the mutex is of type `pthread_mutex_t`.

```
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
```

`mutex`: the mutex to intiialize.

`attributes`: the attributes; `NULL` is fine for defaults.

Shortcut if you do not want to set attributes:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

By default, the mutex is created as unlocked.

```
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
```

Unlock is self-explanatory.

`pthread_mutex_lock` is blocking.

`pthread_mutex_trylock` is nonblocking.

Trylock will come up again soon when we look at another classical synchronization problem.

Destroy the Mutex

```
pthread_mutex_destroy( pthread_mutex_t *mutex )
```

Destroy is also self-explanatory.

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

Attempting to destroy a locked one results in undefined behaviour.

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.

```
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.

```
#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[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 );
}
```

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.

```
wait( mutex )
count++
if count == n
    post( barrier )
end if
post( mutex )
wait( barrier )
post( barrier )
```

```
wait( mutex )
count++
if count < n
    cond_wait( barrier, mutex )
else
    cond_broadcast( barrier )
end if
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?