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.*

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.

Lock and Unlock

```
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 soon when we look at classical synchronization problems.

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

Condition variables are another way to achieve synchronization.

Condition Variables

Conceptually, a CV is a queue on which a thread may wait inside a critical section for a condition to become true.

While a thread is waiting on a CV, other threads may enter the critical section to modify the shared data.

These other threads can then signal the CV to indicate that the condition has become true after the changes have been made.

Condition Variable

Consider a monitor with mutex `m` and a CV `c`.

There are three main operations on `c`:

- `cond_wait( c, m )`
- `cond_signal( c )`
- `cond_broadcast( c )`

CV: Wait

`cond_wait` takes the following steps.

Step (1): *atomically*:

1. unlock the mutex `m`;
2. put the calling thread on c's wait queue; and
3. block the calling thread (i.e., make it sleep) and yield the processor to another thread.

Step (2): Once a waiting thread is subsequently notified/signaled (see below) and resumed, it will automatically try to re-lock the mutex `m`.

Step (1)

The atomicity of operations within Step (1) is crucial.

One failure mode that could occur if these operations were not atomic is a missed wakeup.

CV: Signal

`cond_signal` is called by a thread to indicate that a condition associated with `c` has become true.

A thread that calls `cond_signal` should do so while it is still inside the critical section, before unlocking the mutex `m`.

If there are waiting threads on `c`'s wait queue, then `cond_signal` will move one of them to the ready queue.

If there are no waiting threads, then the signal is ignored.

In other words, unlike semaphores, CVs are *memory-less*.

HEY GUYS!!! GUESS WHAT!!!

`cond_broadcast` is similar to `cond_signal` but wakes up all waiting threads in c's wait queue if there are any.

What Should I Do With This?

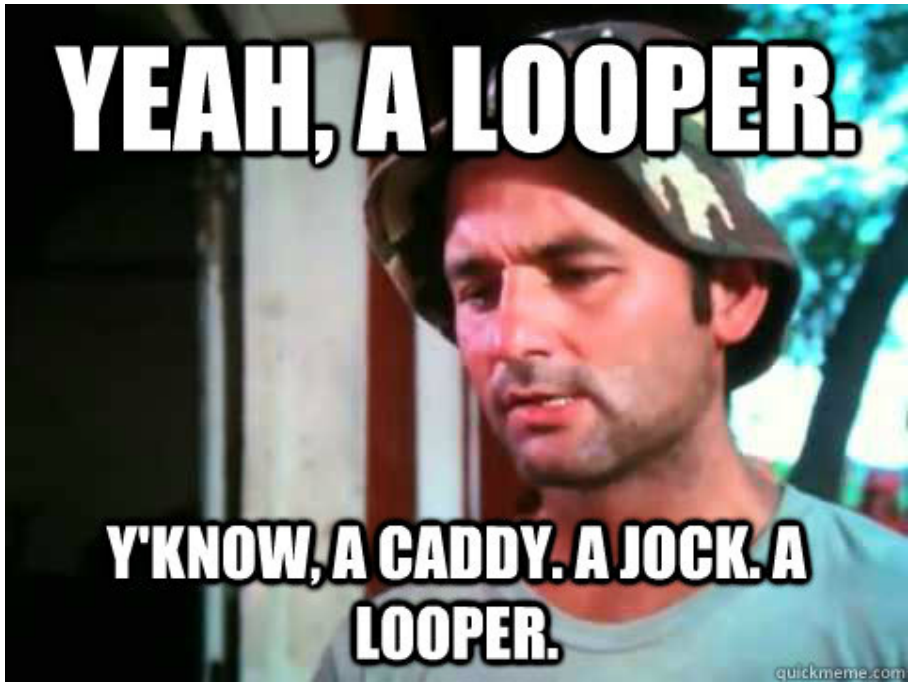
Waiting Thread

```
lock( m )
/* critical section */
while( !is_condition_true() ) {
    cond_wait( m, c)
}
/* rest of critical section */
unlock( m )
```

Signaling Thread

```
lock( m )
/* critical section */
/* if is_condition_true() is now true */
cond_signal( c )
/* rest of critical section */
unlock( m )
```

Why Should We Loop?



Ice Cream

Suppose that thread A locks the mutex and enters the critical section.

It then checks if there is ice cream in the freezer and calls `cond_wait` because there is none.

Inside `cond_wait`, the mutex is unlocked before thread A is put to sleep.

Ice Cream

Next, thread B arrives, locks the mutex, and enters the critical section.

Thread B buys ice cream and puts it in the freezer.

Subsequently, thread B signals the condition variable.

Since thread A is the only waiting thread, it wakes up and moves to the ready queue.

Ice Cream

Thread B continues to run and eventually exits the critical section.

Before thread B finishes its execution, thread C arrives and is put in the ready queue alongside thread A.

Thread B finishes its execution, and the OS picks thread C as the next thread to run.

Thread C locks the mutex, enters the critical section, and eats the ice cream.

Thread C then finishes up and exits the critical section.

Ice Cream

When A runs, there is no ice cream in the freezer.

Therefore, thread A must check the condition and call `cond_wait` again.

That's Not All!

Java User Manual:

> *When waiting upon a Condition, a spurious wakeup is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.*

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. lock( mutex )
2. count++
3. if count == n
4.     cond_broadcast( barrier )
5. else while count < n
6.     cond_wait( barrier, mutex )
7. end if
8. unlock( mutex )
```

The `cond_wait` takes place before `unlock( 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.

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.

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_broadcast( &cv );
  } else {
    while ( count < NUM_THREADS ) {
      pthread_cond_wait( &cv, &lock );
    }
  }
  pthread_mutex_unlock( &lock );
}
```

Will this work?