ECE 252: Systems Programming and Concurrency Spring 2024

Lecture 14-2 — Monitors

By Jeff Zarnett and Seyed Majid Zahedi

Monitors

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. Similarly, analyzing code that uses semaphores is complex. The main reason for this complexity is that semaphores serve a dual purpose (i.e., they are used for both mutual exclusion and scheduling constraints). Dijkstra himself noted this [Dij68]:

> *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.*

To solve this issue, we use *monitors*. 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 (Lock)

While it is possible, of course, to use a semaphore as a mutex, a more specialized tool for this task is the pthread mutex. The structure representing the mutex is of type `pthread_mutex_t`. We don't care about the internals or what the struct is made of; it is either locked or unlocked and that's all that matters to us.

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

The first function of note is `pthread_mutex_init` which is used to create a new mutex variable and returns it, with type `pthread_mutex_t`. It takes an optional parameter, the attributes (the details of which are not important at the moment, but relate mostly to priorities). We can initialize it using `NULL` and that is sufficient. There is also a syntactic shortcut to do static initialization if you do not want to set attributes [Bar14]:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

When created, by default, the mutex is unlocked. There are three methods related to using the mutex; two to lock it and one to unlock it, all of which take as a parameter the mutex to (un)lock. The unlock method, `pthread_mutex_unlock` is self-explanatory. As expected, attempting to unlock a mutex that is not currently locked is an error, but it is also an error if one thread attempts to unlock a mutex owned by another thread [Bar14].

The two kinds of lock are `pthread_mutex_lock`, which is blocking, and `pthread_mutex_trylock`, which is nonblocking. The lock function works as you would expect: if the mutex is currently locked, the calling function is blocked until its turn to enter the critical section; if the mutex is unlocked then it changes to being locked and the current thread enters the critical section. Trylock is more complicated and will come up again soon when we look at classical synchronization problems.

To destroy a mutex, there is a method `pthread_mutex_destroy`. As expected, it cleans up a mutex and should be used when finished with it. If attributes were created with `pthread_mutexattr_init` they should be destroyed with `pthread_mutexattr_destroy`.

An attempt to destroy the mutex may fail if the mutex is currently locked. The specification says that destroying an unlocked mutex is okay, but attempting to destroy a locked one results in undefined behaviour. Undefined behaviour is, in the words of the internet, the worst thing ever: it means code might work some of the time or on some systems, but not others, or could work fine for a while and then break suddenly later when something else is changed[1].

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.

Considering a monitor with mutex `m` and a CV `c`, there are three main operations on `c`:

First, we have `cond_wait( c, m )`. This operation is called by a thread that needs to wait until a condition is true before proceeding. It is called inside the critical section, which means that the calling thread has already locked the mutex `m`. The operation 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`.

While the thread is sleeping and in `c`'s wait queue, the next program counter to be executed is at Step (2), in the middle of `cond_wait`. Therefore, the thread sleeps and later wakes up in the middle of the `cond_wait` function.

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*. In this scenario, the thread could be on `c`'s wait queue and have unlocked the mutex, but a preemptive thread-switch occurs before the thread goes to sleep. If another thread then calls a `cond_signal` operation (explained below) on `c`, moving the first thread out of `c`'s wait queue, the signal will be lost. This is because when the first thread is switched back, its program counter will be at Step 1.3, causing it to sleep and making it impossible to wake up again.

Second, we have `cond_signal( c )`. This operation 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*.

Third, we have `cond_broadcast( c )`. This operation is similar to `cond_signal` but wakes up all waiting threads in `c`'s wait queue if there are any.

Given these three operations, a proper usage of a CV is:

[1] Sadly, the specifications for C and POSIX and many other things are riddled with these "undefined behaviour" situations and it causes programmers everywhere a great deal of stress and difficulty. Another example: reading from an uninitialized variable in C produces undefined behaviour too.

Waiting Thread

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

Signaling Thread

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

Note that `is_condition_true()` is checked in a `while` loop. To understand why, consider the following scenario. Suppose that thread A locks the mutex and enters the critical section. It then checks condition p (e.g., is there ice cream in the freezer) and calls `cond_wait` because p is false. Inside `cond_wait`, the mutex is unlocked before thread A is put to sleep.

Next, thread B arrives, locks the mutex, and enters the critical section. Thread B makes some changes to the shared data that make condition p true (e.g., 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.

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 and enters the critical section. Thread C makes further changes to the shared data that make condition p false again (e.g., eats the ice cream). Thread C then finishes up and exits the critical section.

Finally, the OS picks thread A to run. Thread A re-locks the mutex. Since thread A was signaled, the condition p has changed (there is again no ice cream in the freezer). Therefore, thread A must check the condition and call `cond_wait` again.

Apart from this, some implementations of CV allow for a *spurious wakeup*! For example, Java User Manual reads:

> *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.*

The CV 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 CV on the right:

With Semaphores

```
 1. wait( mutex )
 2. count++
 3. if count == n
 4.    post( barrier )
 5. end if
 6. post( mutex )
 7. wait( barrier )
 8. post( barrier )
 9. /* critical section */
10. wait( mutex )
11. counter--
12. if counter == 0
13.    wait( barrier )
14. end if
15. post( mutex )
```

With Monitor

```
 1. lock( mutex )
 2. count++
 3. if count == n
 4.    cond_broadcast( barrier )
 5. end if
 6  while count < n
 7.    cond_wait( barrier, mutex )
 8. end while
 9. unlock( mutex )
10. /* critical section */
11. lock( mutex )
12. counter--
13. unlock( mutex )
```

Note again that the `cond_wait` takes place before `unlock( mutex )`. What's important to remember is that we unlock the mutex when we wait on the CV, 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.

We are now ready to look at the syntax of the pthread CV:

```
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` (CV 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.

CVs are always used in conjunction with a mutex. To wait on a CV, the function `pthread_cond_wait` takes two parameters: the CV 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 CV, respectively, can result in problems.

In addition to the expected `pthread_cond_signal` function that signals a provided CV, there is also `pthread_cond_broadcast` that signals all threads waiting on that CV. It's this "broadcast" idea that makes the CV more interesting than the simple "signalling" pattern we covered much earlier on.

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 enter_barrier( ) {
  pthread_mutex_lock( &lock );
  count++;
  if ( count == NUM_THREADS ) {
    pthread_cond_broadcast( &cv );
  }
  while ( count < NUM_THREADS ) {
    pthread_cond_wait( &cv, &lock );
  }
  pthread_mutex_unlock( &lock );
}

void exit_barrier( ) {
  pthread_mutex_lock( &lock );
  count--;
  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 CV n times but that mostly defeats the purpose of using the CV instead of a regular semaphore, doesn't it?

References

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

[Dij68] Edsger Dijkstra. The Structure of the 'THE' Multiprogramming System. In *Communications of the ACM*, 1968.