ECE459: Programming for Performance Winter 2018

Lecture 7 — Use of Locks, Lock Convoys

Jeff Zarnett

Appropriate Use of Locking

In previous courses you learned about locking and how it all works, then we did a quick recap of what you need to know about it. And perhaps you were given some guidance in the use of locks, but probably in earlier scenarios it was sufficient to just avoid all the bad stuff (data races, deadlock, starvation). That's important but is no longer enough. Now we need to use locking and other synchronization techniques appropriately.

I like to say that critical sections should be as large as they need to be but no larger. That is to say, if we have some shared data that needs to be protected by some mutual exclusion constructs, we need to consider carefully where to place the statements. They should be placed such that the critical section contains all of the shared accesses, both reads *and* writes, but also does contain any statements. The ones that don't need to be there are those that don't operate on shared data.

This can mean that a block of code or contents of a function need to be re-arranged to move some statements up or down so they are no longer in the critical section. Sometimes control flow or other very short statements might get swept into the critical section being created to make sure all goes as planned but those should be the exception rather than the rule.

Let's consider a short code example from the producer-consumer problem. We have some global variables below that will be initialized as appropriate. There is also a definition of the function that will consume the data.

```
sem_t spaces;
sem_t items;
int counter;
int* buffer;
int pindex = 0;
int cindex = 0;
int ctotal = 0;
pthread_mutex_t prod_mutex;
pthread_mutex_t con_mutex;

void consume( int to_consume );
```

And then here is our single-threaded code for the consumer:

```
void* consumer( void* arg ) {
  while( ctotal < MAX_ITEMS_CONSUMED ) {
    sem_wait( &items );
    consume( buffer[cindex] );
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    ++ctotal;
    sem_post( &spaces );
  }
}
```

To this we need to add some mutual exclusion if we want to allow multiple consumers at the same time. I'll leave aside the case of only allowing one consumer by putting the lock and unlock statements outside the while loop since that defeats the purpose of having multiple threads altogether. One approach we could take is that which allows exactly one consumer to run at a time, as below. But what's wrong with this?

```
void* consumer( void* arg ) {
  while( ctotal < MAX_ITEMS_CONSUMED ) {
    pthread_mutex_lock( &con_mutex );
    sem_wait( &items );
```

```
    consume( buffer[cindex] );
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    ++ctotal;
    sem_post( &spaces );
    pthread_mutex_lock( &con_mutex );
  }
}
```

What I recommend is of course to analyze this function one statement at a time and look into which of these access global variables. We're not worried about statements like locking, wait, or post, but let's look at the rest and decide if they really belong. Can any statements be removed from the critical section?

At first glance it is probably not very obvious but the `consume` function takes a regular integer, any old integer, not a pointer of some sort. So we could. inside the critical section, read the value of the buffer at the current index into a temp variable. That temp variable then can be given to the `consume` function at any time... outside of the critical section. Everything else inside our lock and unlock statements seems to be shared data: operates on `cindex` or `ctotal`.

```
void* consumer( void* arg ) {
  while( ctotal < MAX_ITEMS_CONSUMED ) {
    pthread_mutex_lock( &con_mutex );
    sem_wait( &items );
    int temp = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    ++ctotal;
    sem_post( &spaces );
    pthread_mutex_lock( &con_mutex );
    consume( temp );
  }
}
```

Next question then. With nothing left to take away, is there something left to add? Yes! The condition of the while loop checks the value of `ctotal` and that is a read of shared data. Now we maybe have a problem. How do we get that inside the critical section? One idea we might have is to read the value of ctotal into a temporary variable and use that, but it might cause some headaches with the timing (the end of the loop might be mispredicted...). Instead what I'd recommend is to make the loop a while true loop and then have a test of the value to determine when we should break out of the loop. See the example below, remembering of course there is the potential pitfall of forgetting to unlock the mutex if we are going to the break statement:

```
void* consumer( void* arg ) {
  while( 1 ) {
    pthread_mutex_lock( &con_mutex );
    if ( ctotal == MAX_ITEMS_CONSUMED ) {
      pthread_mutex_unlock( &con_mutex );
      break;
    }
    sem_wait( &items );
    int temp = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    ++ctotal;
    pthread_mutex_unlock( &con_mutex );
    sem_post( &spaces );
    consume( temp );
  }
  pthread_exit( NULL );
}
```

At this stage we should (mostly) be happy with the conversion of the function to support multithreaded operation. This conversion isn't the only way, but there are others. Remember, though, that keeping the critical section as small as possible is important because it speeds up performance (reduces the serial portion of your program). But that's not the only reason. The lock is a resource, and contention for that resource is itself expensive.

Lock Convoys

We'd like to avoid, if at all possible, a situation called a *lock convoy*. This happens when we have at least two threads that are contending for a lock of some sort. And it's sort of like a lock traffic jam. A more full and complex description from [Loh05]:

> A lock convoy is a situation which occurs when two or more threads at the same priority frequently (several times per quantum) acquire a synchronization object, even if they only hold that object for a very short amount of time. It happens most often with critical sections, but can occur with mutexes, etc as well. For a while the threads may go along happily without contending over the object. But eventually some thread's quantum will expire while it holds the object, and then the problem begins. The expired thread (let's call it Thread A) stops running, and the next thread at that priority level begins. Soon that thread (let's call it Thread B) gets to a point where it needs to acquire the object. It blocks on the object. The kernel chooses the next thread in the priority-queue. If there are more threads at that priority which end up trying to acquire the object, they block on the object too. This continues until the kernel returns to Thread A which owns the object. That thread begins running again, and soon releases the object. Here are the two important points. First, once Thread A releases the object, the kernel chooses a thread that's blocked waiting for the object (probably Thread B), makes that thread the next owner of the object, and marks it as "runnable." Second, Thread A hasn't expired its quantum yet, so it continues running rather than switching to Thread B. Since the threads in this scenario acquire the synchronization object frequently, Thread A soon comes back to a point where it needs to acquire the object again. This time, however, Thread B owns it. So Thread A blocks on the object, and the kernel again chooses the next thread in the priority-queue to run. It eventually gets to Thread B, who does its work while owning the object, then releases the object. The next thread blocked on the object receives ownership, and this cycle continues endlessly until eventually the threads stop acquiring so often.

References

[Loh05] Sue Loh. Lock Convoys and How to Recognize Them, 2005. Online; accessed 3-December-2017. URL: `https://blogs.msdn.microsoft.com/sloh/2005/05/27/lock-convoys-and-how-to-recognize-them/`.