Lecture 7 — Use of Locks, Lock Convoys

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 21, 2017

Use of Locks

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.

In earlier scenarios it was sufficient to just avoid all the bad stuff.

That’s important but is no longer enough.

Critical Section Size

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.

The critical section must contain all of the shared accesses, both reads *and* writes.

But no extraneous statements (those that don't operate on shared data).

Rewrite This Block

This can mean that a block of code or contents of a function need to be re-arranged.

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.

Those should be the exception rather than the rule.

Remember the Producer-Consumer Problem?

Let's consider a short code example from the producer-consumer problem.

```
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 );
```

Remember the Producer-Consumer Problem?

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 );
  }
}
```

Multiple Consumers

To this we need to add some mutual exclusion if we want to allow multiple consumers at the same time.

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 );
  }
}
```

Anything to Kick Out?

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`.

Updated Code

```
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 );
  }
}
```

Did We Get Everything?

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...

Not Forgetting The Loop Condition

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.

```
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 );
}
```

Are we done?

Locks Have Costs

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.

Locking Granularity

Alright, we already know that locks prevent data races.

So we need to use them to prevent data races, but it's not as simple as it sounds.

We have choices about the granularity of locking, and it is a trade-off.

Coarse-Grained vs Fine-Grained Locking

Coarse-grained locking is easier to write and harder to mess up, but it can significantly reduce opportunities for parallelism.

Fine-grained locking requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc).

Locks' extents constitute their granularity.

Lock Concerns

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e., remaining race conditions).

Lock Overhead

Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention

Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks

Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Okay, in a formal sense, the four conditions for deadlock are:

1. **Mutual Exclusion**
2. **Hold-and-Wait**
3. **No Preemption**
4. **Circular-Wait**

Simple Deadlock Example

Consider, for instance, two processors trying to get two locks.

Thread 1

```
Get Lock 1
Get Lock 2
Release Lock 2
Release Lock 1
```

Thread 2

```
Get Lock 2
Get Lock 1
Release Lock 1
Release Lock 2
```

Avoiding Deadlocks

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**.

You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
void f1() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

```
void f2() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

Why Ordering Works

If the set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$, we assign to each resource R_k a unique integer value.

Let us define this function as $f(R_i)$, that maps a resource to an integer value.

This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$.

Note that this is a strictly greater-than relationship; if the process needs more than one of R_i then the request for all of these must be made at once.

To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$.

If these two protocols are followed, then a circular-wait condition cannot hold.

Trylock

Alternately, you can use trylock.

Recall that Pthreads' `trylock` returns 0 if it gets the lock.

But if it doesn't, your thread doesn't get blocked.

```
void f1() {
    lock(&l1);
    while (trylock(&l2) != 0) {
        unlock(&l1);
        // wait
        lock(&l1);
    }
    // protected code
    unlock(&l2);
    unlock(&ll);
}
```

This prevents the hold and wait condition.

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (1?).

This is *coarse-grained locking*; it does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

Your parallel program will quickly become sequential.

Python

Python puts a lock around the whole interpreter (known as the *global interpreter lock*).

This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for IO.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*.

The big advantage: it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Lock Scope Examples

Databases may lock fields / records / tables. (fine-grained $\rightarrow$ coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)