ECE459: Programming for Performance Winter 2019

Lecture 10 — Use of Locks)

Jeff Zarnett *2019-01-03*

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 extraneous 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. We want our consumer threads to consume exactly `MAX_ITEMS_CONSUMED` items and then cleanly exit; we don't want anything to stay stuck at a mutex for instance.

```
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_unlock( &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?

We cannot move the `sem_wait` before the lock acquisition if we want to consume exactly `MAX_ITEMS_CONSUMED` items. If we did, then we could have threads that make it past the `sem_wait` but get stuck at the lock acquisition. They won't do anything bad, but they also never terminate. Such threads would be tricky to terminate: `pthread_join` would block the main thread, and thread cancellation (danger! danger!) is not only dangerous, but also difficult, because it's not clear which thread should be cancelled and which one has finished.

We can move the `sem_post` after the unlock if we're trying to reduce critical sections.

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;
    pthread_mutex_unlock( &con_mutex );
    sem_post( &spaces );
    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. You then do have to move the lock before the sem_wait. 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.

Locking Granularity

Alright, we already know that locks prevent data races. If this is news to you, how did you pass the operating systems course?! 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 (like always).

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*. In coarse-grained locking, you lock large sections of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections with multiple smaller locks.

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). We'll assume there are adequate locks and that data accesses are protected.

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**: A resource belongs to, at most, one process at a time.
2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be “taken” from the process that holds it; only the process currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

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

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

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);
}
```

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it’s harder to ensure a consistent deadlock when lock identity is not statically visible.

Or another example, with threads P and Q attempting to acquire `a` and `b`. Thread Q requests `b` first and then `a`, while P does the reverse. The deadlock would not take place if both threads requested these two resources in the same order, whether `a` then `b` or `b` then `a`. Of course, when they have names like this, a natural ordering (alphabetical, or perhaps reverse alphabetical) is obvious.

To generalize and formalize this principle, 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 (in a single request). 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 [SGG13].

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. Checking the return value is important, but at the very least, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

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

(Incidentaly, using trylocks can also help you measure lock contention.)

This prevents the hold and wait condition, which was one of the four conditions. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources.

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). 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.

It also, however, has one big disadvantage in terms of programming for performance: your parallel program will quickly become sequential.

Example: Python (and other interpreters). 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 I/O.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

You might think "this is stupid, who would ever do this?" Yet a lot of OS kernels do in fact have (or at least had) a "big kernel lock", including Linux and the Mach Microkernel. This lasted in Linux for quite a while, from the advent of SMP support up until sometime in 2011. As much as this ruins performance, correctness is more important. We don't have a class "programming for correctness" (software testing? Hah!) because correctness is kind of assumed. What we want to do here is speed up our program as much as we can while maintaining correctness...

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!)

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.)

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.