

# Lecture 7 — Use of Locks, Lock Granularity

Jeff Zarnett & Patrick Lam

{jzarnett, patrick.lam}@uwaterloo.ca

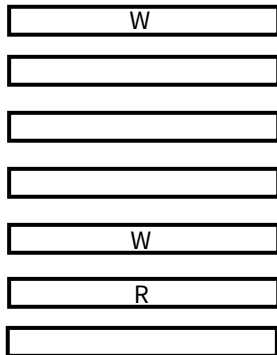
Department of Electrical and Computer Engineering  
University of Waterloo

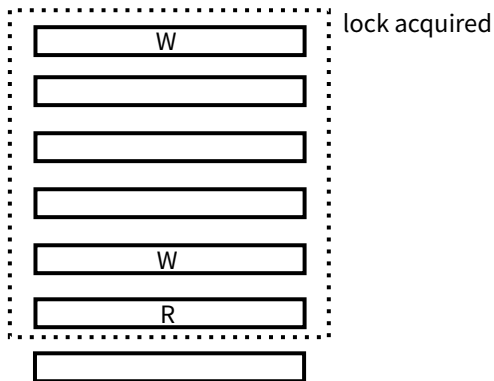
January 25, 2018

In previous courses: learned about locking and how it works.

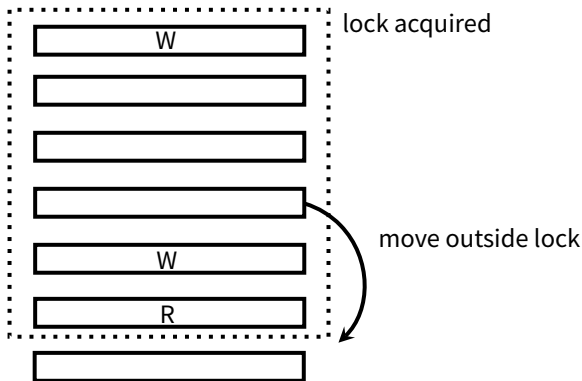
Before: was enough to avoid bad stuff.

Now: Still important, but not enough.





Critical sections should be as large as they need to be but no larger.



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?

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 ctotat = 0;  
pthread_mutex_t prod_mutex;  
pthread_mutex_t con_mutex;  
  
void consume( int to_consume );
```

---

# Remember the Producer-Consumer Problem?

Single-threaded consumer code:

---

```
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 );  
    }  
}
```

---

Must add mutual exclusion to allow multiple simultaneous consumers.

Could allow exactly one consumer to run at a time. We can do better...

---

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

---



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]; // store for now  
        buffer[cindex] = -1;  
        cindex = (cindex + 1) % BUFFER_SIZE;  
        ++ctotal;  
        sem_post( &spaces );  
        pthread_mutex_unlock( &con_mutex );  
        consume( temp ); // consume outside mutex  
    }  
}
```

---

The while condition 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[index];  
        buffer[index] = -1;  
        index = (index + 1) % BUFFER_SIZE;  
        ++ctotal;  
        pthread_mutex_unlock( &con_mutex );  
        sem_post( &spaces );  
        consume( temp );  
    }  
    pthread_exit( NULL );  
}
```

---

Are we done?

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.

Alright, we already know that locks prevent data races.

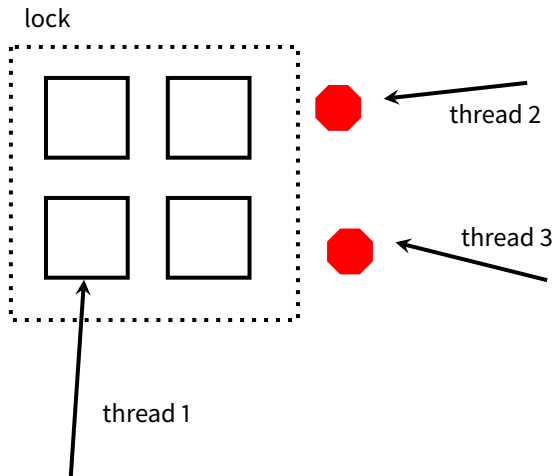
So we need to use them to prevent data races, but it's not as simple as it sounds.

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

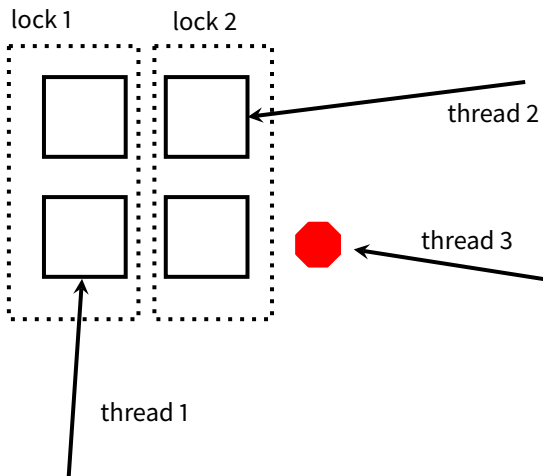
**LOCK ALL THE THINGS**



# Coarse-Grained Lock







# 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.

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).

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.

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.

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

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);  
}
```

---



If the set of all resources in the system is  $R = \{R_0, R_1, R_2, \dots, R_m\}$ , 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.

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(&l1);  
}
```

---

This prevents the hold and wait condition.

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 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).

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!)

Databases may lock fields / records / tables. (fine-grained → coarse-grained).

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