Lecture 4 — Review of Concurrency Control

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 26, 2023

Recall two important concepts: the semaphore and the mutex.

We'll review the basic theory and usage functions for:

1. Mutex
2. Semaphore
3. Readers-Writers Lock
4. Condition Variable

Stay Out of Trouble

Concurrency control is about ensuring the correctness of the program.

Race conditions are “hazards”:

	Second Access	
First Access	**Read**	**Write**
Read	No Dependency Read After Read (RAR)	Anti-Dependency Write After Read (WAR)
Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

No Such Thing

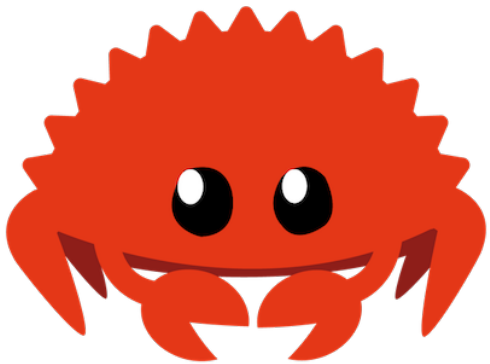
RAR is not real – no hazard!

Other situations easy to imagine.

Advisory Locks Only

Concurrency control mechanisms only work if used correctly.

Unless...?



But this is C so the onus is on the application developer.

Mutex

We spent a lot of time talking about semaphores, but the mutex is often enough.

Usually we talk about locking and unlocking.

The mutex has two states, and only two states, locked and unlocked. There's no need to specify an initial value or anything to that effect.

Mutex Functions

```
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 )
```

When created, by default, the mutex is unlocked.

The intended behaviour is that the thread that locked the mutex is the same the one to unlock it later on.

Semaphore

In UNIX the kind of semaphore is known as a *counting* or *general* semaphore.

```
sem_init( sem_t* semaphore, int shared, int initial_value)
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

Of these functions, the only one where the parameters are not obvious is the initialization routine.

UNIX semaphores do not give you the option of a maximum value.

Increment and Decrement

A thread that waits on that semaphore will decrement the integer by 1.

A thread that signals on the semaphore will increment the integer by 1.

New value negative? Calling thread is blocked.

Cannot Check

No option to check the value of the semaphore.

Trylock sort of helps?

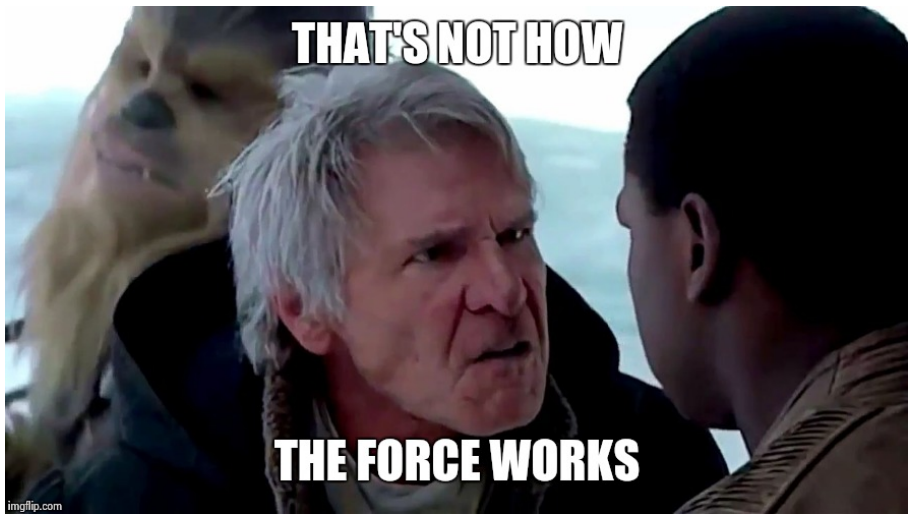
But the answer could change.

After You...

When a thread signals a semaphore, it likewise does not know if any other thread(s) are waiting on that semaphore.

What thread continues? Scheduling decision.

Misconception?



If the semaphore's value is currently -3 and there are 3 threads waiting, a call to `post` will both increment the value to -2 and unblock one of the threads waiting.

Semaphore Destruction

Unlike the mutex, there's no requirement for the semaphore to be in a certain state before it is destroyed.

Reusable barrier was more desirable only for efficiency?

Semaphores Are Flexible

We used the semaphore in all sorts of synchronization problems.

We won’t be examining all the ways we could use it any further.

Readers-Writers Locks

Unlike the producer-consumer problem, some concurrency is allowed, recognizing the fact that there is no read-after-read dependency:

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers are allowed).

RW Lock Functionality

The type for the lock is `pthread_rwlock_t`.

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock )
pthread_rwlock_trywrlock( pthread_rwlock_t  *  rwlock )
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

In theory, the same thread may lock the same rwlock *n* times; just remember to unlock it *n* times as well.

Priority?

As for whether readers or writers get priority, the specification says this is implementation defined.

If possible, for threads of equal priority, a writer takes precedence over a reader.

We also examined situations where we really did need to enforce writer priority would require us to build our own complicated solution with semaphores and a mutex.

RW Example

```
pthread_rwlock_t rwlock;

void init( ) {
  pthread_rwlock_init( &rwlock, NULL );
}

void cleanup( ) {
  pthread_rwlock_destroy( &rwlock );
}
```

```
void* writer( void* arg ) {
  pthread_rwlock_wrlock( &rwlock );
  write_data( arg );
  pthread_rwlock_unlock( &rwlock );
}

void* reader( void* read ) {
  pthread_rwlock_rdlock( &rwlock );
  read_data( arg );
  pthread_rwlock_unlock( &rwlock );
}
```

Condition Variable

The condition variable looks a lot like a semaphore, but it has a few interesting differences.

We have the option, when an event occurs, to signal either one thread waiting for that event to occur…

… or to broadcast (signal) to all threads waiting for the event.

Condition Variable Functions

```
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 );
```

Condition variables are always used in conjunction with a mutex.

Why Use It?

A thread is partway through some thing and wants to wait until a condition is fulfilled before proceeding?

Also if you need broadcast functionality?

Lost Wakeup

If a thread signals/broadcasts and nobody's listening, the event is "lost".

Sometimes that's fine.

More Efficient?

The broadcast does wake up many different threads, but it does not mean they all start running immediately.

They each have to wait their turn for the mutex to continue for real.

Consider the barrier pattern from earlier: broadcast may be better!

Implementation

But how do they really work?

The answer is that it's both simpler and more complicated than it might seem.