ECE 350: Real-Time Operating Systems Spring 2023

Lecture 4 — Review of Concurrency Control

Jeff Zarnett *2023-09-09*

Concurrency Control

When discussing concurrency and synchronization in previous courses we talked about two important concepts: the semaphore and the mutex. We used them for multiple scenarios including mutual exclusion (preventing race conditions), making one thread wait for another, or managing space in a buffer with the producer-consumer problem. We'll review the basic theory and usage functions for our four favourite mutual exclusion constructs: the mutex, the semaphore, the readers-writers lock, and the condition variable.

Yes, you can, to some extent, get the program behaviour that you want with *just* the semaphore. The other sorts of concurrency control constructs will make it easier to accomplish the goals and harder to make a mistake.

The purpose of the concurrency control constructs is to ensure the correctness of the program. We use them to prevent race conditions, but also to ensure that the program executes in the order that we intend. We said it's a race condition where you have unsynchronized concurrent accesses to the same memory location, where at least one of those accesses is a write. Those situations will fall into one of the following categories, which you may also see referred to sometimes as "hazards":

		Second Access	
		Read	**Write**
First Access	**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)

One of them is not real, though: there's no read-after-read dependency, because it doesn't meet the definition we just covered: in read-after-read, none of the accesses are a write so there's no issue. And indeed, if a global variable is never changed then concurrent accesses to it are not a problem. It appears in the table for completeness; otherwise there might be a question about why that box is missing or blank.

The other situations are all easy to imagine. The true dependency is likely the first one we think of when imagining a dependency: writing the result of a previous stage of the calculation must go first because the value will be read as the input to the next stage. The output dependency is also common: we don't want the older version of the weather forecast to overwrite the more up-to-date one. The anti-dependency is perhaps the least likely one to imagine when we talk about hazards, but it is what happens when we cannot overwrite thee contents of some message buffer until that data has been consumed. You can frame other scenarios of program coordination as one or another of these dependencies.

It's important to remember that all of the forms of concurrency control that we talked about in the previous course are what is called *advisory*. As I like to say in the analogy, they are like seatbelts in a car: they only work if you use them correctly. For example, nothing prevents the program author from accessing shared memory without holding the appropriate lock, and nothing compels them to unlock it when they're finished. They are supposed to. The program doesn't work correctly if they don't. But nothing forces them to. Other programming languages (like Rust!) make it harder to forget to unlock things and forbid access to things protected by a Mutex. But this is C, so, the onus is on the application developer.

The car analogy works here too: the OS is like the manufacturer: it provides seatbelts in the vehicle (concurrency control constructs) and even explains in the owner's manual how they work, but the manufacturer can't force everyone to put them on. The operating system does not second guess the application developers' choices, even if it leads to wrong outcomes or concurrency issues.

The Mutex

Although we spent a lot of time talking about the semaphore, in many scenarios, all we really need is the mutex functionality to prevent unwanted concurrent access to the same memory location(s). Whereas semaphores talk about wait and signal/post, most commonly in discussions we talk about locking and unlocking the mutex.

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.

```
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 )
```

The first function of note is `pthread_mutex_init` which is used to create a new mutex variable and returns it, with type `pthread_mutex_t`. It takes an optional parameter, the attributes. We can initialize it using `NULL` and that is sufficient. There is also a syntactic shortcut to do static initialization if you do not want to set attributes [Bar14]:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

When created, by default, the mutex is unlocked. There are three methods related to using the mutex; two to lock it and one to unlock it, all of which take as a parameter the mutex to (un)lock. The unlock method, `pthread_mutex_unlock` is self-explanatory. As expected, attempting to unlock a mutex that is not currently locked is an error [Bar14]. Why? This is a binary sort of construct (locked or unlocked), so call to unlock does not mean that two threads that try to lock it will proceed; in this sense it is not like a semaphore.

The intended behaviour is that the thread that locked the mutex is the same the one to unlock it later on. There are scenarios where it's okay to send a locked lock over to another thread and have it take some action and then unlock it, but that's rare and harder to get right.

The two kinds of lock are `pthread_mutex_lock`, which is blocking, and `pthread_mutex_trylock`, which is nonblocking. The lock function works as you would expect: if the mutex is currently locked, the calling function is blocked until its turn to enter the critical section; if the mutex is unlocked then it changes to being locked and the current thread enters the critical section.

To destroy a mutex, use `pthread_mutex_destroy`. As expected, it cleans up a mutex and should be used when finished with it. If attributes were created with `pthread_mutexattr_init` they should be destroyed with `pthread_mutexattr_destroy`. An attempt to destroy the mutex may fail if the mutex is currently locked. The specification says that destroying an unlocked mutex is okay, but attempting to destroy a locked one results in undefined behaviour.

The Semaphore

You remember, I'm sure, that in UNIX the kind of semaphore is known as a *counting* or *general* semaphore. Instead of having only the values 0 and 1, the setup routine for the counting semaphore allows the choice of an integer value as the initial value. So, the functions are:

```
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. The parameter `shared` will be set to either 0 or 1: 1 if the semaphore is to be shared between processes (e.g., using shared memory), 0 otherwise. I'll also take a moment also to point out the importance of getting the initial value correct. If we choose the wrong initial value then our program might get stuck or we might not get the mutual exclusion behaviour we are supposed to have.

In the previous course, the introduction to the semaphore talked about a hypothetical implementation that allowed for the possibility that the mutex could have a maximum value. UNIX semaphores do not give you this option, so if you call `post` 300 times on the semaphore whose initial value was 5, it doesn't matter if those 300 post calls result in a final value of 305.

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. If a thread attempts to wait on a semaphore and the decrement operation makes the integer value negative, the calling thread is blocked. If, however, the semaphore is, for example, initialized with 5 and the current value is 2, a thread that waits on that general semaphore will not be blocked. Remember that the increment and decrement of the counter are unconditional. Things are a bit different in the mutex, but we didn't need to know so much about that to use it.

Note also that the semaphore does not provide any facility to "check" the current value. Thus a thread does not know in advance if it will block when it waits on the semaphore. It can only wait and may be blocked or may proceed directly into the critical section if there is no other thread in there at the moment. The closest we get to that is the trylock behaviour, which will acquire the lock if it can, but it still doesn't really tell you the current value. We learned previously that the reason is that the attempt to "check" the value might return you the current value, sure, but by the time you get that answer back and decide to do something with it, the value could have changed, making the answer useless.

When a thread signals a semaphore, it likewise does not know if any other thread(s) are waiting on that semaphore. There is no facility to check this, either. When thread A signals a semaphore, if another thread B is waiting, B will be unblocked and either thread A or thread B may continue execution (or both, if it is a multiprocessor system), or another unrelated thread may be the one to continue execution. That's an operating system (scheduling) decision.

Another possible misconception around semaphores relates to posting on the value when the value is negative. 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. Sometimes there is an expectation that the counter must get back to 0 before any waiting thread is allowed to proceed, but it's easy to show that that kind of semantics would result in a deadlock in situations where multiple threads must wait their turn before proceeding.

Unlike the mutex, there's no requirement for the semaphore to be in a certain state before it is destroyed. The previous course talked about the reusable barrier as generally desirable over the non-resuable versions since it meant there was less overhead of destroying and recreating the semaphores. But efficiency (real or imagined!) was the only motivation there; there was no logical error in choosing the other route.

We used the semaphore in all sorts of synchronization problems. It could be used in place of a mutex, sure, but it can also be used to arrange a rendezvous, allow limited concurrency with multiplex, and we could even use its internal value indirectly in the producer-consumer problem. As fun as it was to imagine scenarios involving making pizza and the Rebellion's attempt to escape Lord Vader on Hoth, we won't be examining all the ways we could use it any further.

The Readers-Writers Lock

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).

Or, to sum that up, a writer cannot enter the critical section while any other thread (whether reader or writer) is there. While a writer is in the critical section, neither readers nor writers may enter the critical section [Dow08]. This is very often how file systems work: a file may be read concurrently by any number of threads, but only one thread may write to it at a time (and to prevent reading of inconsistent data, no thread may read during the write).

This is similar to, but distinct from, the general mutual exclusion problem and the producer-consumer problem. In the readers-writers problem, readers do not modify the data (consumers do take things out of the buffer, modifying it). If any thread could read or write the shared data structure, we would have to use the general mutual exclusion solution. Although the general mutual exclusion routine would work in that it would prevent errors, it is a serious performance reduction versus allowing multiple readers concurrently [Sta18].

The type for the lock is `pthread_rwlock_t`. It is analogous, obviously, to the mutex type `pthread_mutex_t`. Let's consider the functions that we have:

```
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 general our syntax very much resembles that of the mutex (attribute initialization and destruction not shown but they do exist). There are some small noteworthy differences, other than obviously the different type of the structure passed. Whereas before we had functions for lock and trylock, we now have those split into readlock and writelock (each of which has its own trylock function).

In theory, the same thread may lock the same rwlock n times; just remember to unlock it n times as well. And speaking of unlock, there's no specifying whether you are releasing a read or write lock. This is because it is unnecessary; the implementation unlocks whatever type the calling thread was holding. Much like `close()`, if we can figure out what we're closing we don't need the caller of the function to specify what to do.

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. But your system may vary.

And we also saw an example of it:

```
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 );
}
```

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. There's nothing wrong with that, but it doesn't need to be re-examined here since it's not a system-offered concurrency control mechanism. Or in other words, we won't be examining the details of that in the next topic, since it's just built upon the semaphore and mutex types.

The 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 [HZMG20].

Consider the 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 );
```

To initialize a `pthread_cond_t` (condition variable type), the function is `pthread_cond_init` and to destroy them, `pthread_cond_destroy`. As with threads and a mutex, we can initialize them with attributes, and there are functions to create and destroy the attribute structures, too.

Condition variables are always used in conjunction with a mutex. To wait on a condition variable, the function `pthread_cond_wait` takes two parameters: the condition variable and the mutex. This routine should be called only while the mutex is locked. It will automatically release the mutex while it waits for the condition; when the condition is true then the mutex will be automatically locked again so the thread may proceed. The programmer then unlocks the mutex when the thread is finished with it [Bar14]. Obviously, failing to lock and unlock the mutex before and after using the condition variable, respectively, can result in problems.

One possible scenario where it makes sense to use the condition variable is one in which a thread is partway through some thing and wants to wait until a condition is fulfilled before proceeding. It requires unlocking the mutex and letting other threads proceed, but the condition variable implementation means there's no need exit from this function and restart it and try again – we can just stop now and continue when it's time.

In addition to the expected `pthread_cond_signal` function that signals a provided condition variable, there is also `pthread_cond_broadcast` that signals all threads waiting on that condition variable. It's this "broadcast" idea that makes the condition variable more interesting than the simple "signalling" pattern we covered much earlier on.

It should be noted that if a thread signals a condition variable that an event has occurred, but no thread is waiting for that event, the event is "lost". Because an event that takes place when no thread is listening is simply lost, it is (almost always) a logical error to signal or broadcast on a condition variable before some thread is waiting on it. This is sometimes called the "lost wakeup problem", because threads don't get woken up if they weren't waiting for this. Maybe it's not as dire as that makes it sound, however, because sometimes an event is broadcast but nobody is interested and that's fine.

The broadcast does wake up many different threads, in the sense that they are no longer waiting for the condition variable, but it does not mean they all start running immediately. They each have to wait their turn for the mutex to continue for real. So if you're worried that a broadcast results in many threads immediately running... no need.

The condition variable with broadcast can be used to replace some of the synchronization constructs we've seen already. Consider the barrier pattern from the previous course. There are n threads and we wait for the last one to arrive. Then the last thread signals to unlock the barrier and then each thread calls post to unblock the next thread until all of them are through. This is a lot of calls and maybe it would be better to make it a broadcast instead.

The Implementation

Having seen and worked with these before, we should have a pretty good understanding of what it's like to be a user program making use of these constructs. But how do they really work? The answer is that it's both simpler and more complicated than it might seem.

References

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: `https://computing.llnl.gov/tutorials/pthreads/`.

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.

[HZMG20] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2020. Online; version 0.2020.07.19.

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.