Lecture 33 — Concurrency Control Implementation

Jeff Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 7, 2024

We talked a lot about why we need concurrency control mechanisms.

We even spent a little time discussing how they might work.

But...

One possible solution that works in an embedded system or very simple OS is disabling interrupts.

Crude and permits bad behaviour...

Does not work for multiple processors.

Where we landed was the use of test-and-set!

A special machine instruction that is performed in a single instruction cycle and is therefore not interruptible.

```
boolean test_and_set( int* i ) {
  if ( *i == 0 ) {
    *i = 1;
    return true;
  } else {
    return false;
  }
}
```

An example of the code that uses the test_and_set routine:

```
while ( !test_and_set( busy ) ) {
    /* Wait for my turn */
}
/* critical section */
busy = 0;
```

No matter how many threads are executing the code above concurrently, only one will succeed in actually setting the value to 1.

You will notice here that the assignment of `busy = 0` doesn't use something like a test-and-clear instruction or similar.

Is that assignment okay? Grudgingly!

Test-and-Set does not work for the general, counting semaphore.

Alternative: compare-and-swap (or compare-and-exchange).

```c
int compare_and_swap( int * value, int old_value, int new_value ) {
  if ( *value == old_value ) {
    *value = new_value;
    return old_value;
  }
  return *value;
}
```

And to make use of it in trying to decrement a semaphore:

```
int old = 1;
while (true) {
  int actual = compare_and_swap( sem, old, old − 1 );
  if ( actual == old ) {
    old = old − 1;
    break;
  } else {
    old = actual;
  }
}
/* WAIT FOR OUR TURN */
/* critical section */
while (true) {
  int actual = compare_and_swap( sem, old, old + 1 );
  if ( actual == old ) {
    break;
  } else {
    old = actual;
  }
}
```

We will eventually succeed, even if it takes an arbitrary amount of time.

It might be possible for a thread to be so unlucky it never gets a turn, but let's just say that this does not happen.

The initial guess for `old` can certainly be wrong; the initial attempt to set it will fail and we'll get the correct value.

Need to use CAS for decrementing also!

Alternative: if there is appropriate hardware support, do an atomic increment or atomic decrement.

That prevents a scenario where multiple attempts are necessary to get the value.

There is the big `WAIT FOR OUR TURN` comment there...

Uh... how *do* we wait for it?

Earlier: if we don't get the result we want, the OS blocks the thread.

No details were provided as to how that happens; just hand-waving.

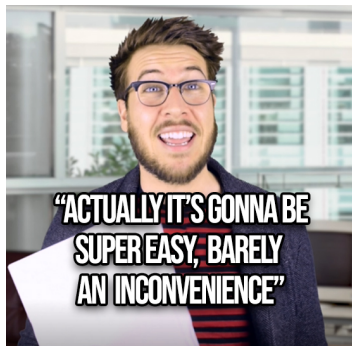When you leave things for tomorrow
and tomorrow arrives

So you want to lock a mutex or wait on a semaphore...

There was a system call; it had a TAS, CAS, or atomic operation.

If this was a trylock call, they don't get blocked; skip this section.

Blocking a thread is easy for the operating system.

Change the status to be blocked and choose another thread to run.

Marking a thread as blocked or moving it to a "blocked" queue is insufficient.

We will need some way of knowing that this thread is blocked on the particular semaphore or mutex it was accessing.

Maybe the mutex/semaphore has its own queue?

Whichever thread did lock the mutex will want to unlock it or some thread will post on a semaphore.

Counting semaphore: always increase counter, unblock a thread waiting.

Mutex: change the counter only if no thread waiting, otherwise unblock.

This prompts immediately the question of what thread should be unblocked when an unlock or post event occurs.

Again, that's a scheduling decision, but you could choose a simple and "fair" approach of just taking the first thread in the queue.

This might not be optimal. Why?

The first-come-first-served approach does prevent the possibility of starvation, as each thread will eventually get a turn.

But just basing it on priority may not be enough?

Could we try to detect deadlock risk?

The thread that's unblocked is marked as ready to run again.
   Moved to ready queue?

And at this point it is a scheduling decision: which thread continues execution.

But we knew that from doing all this from the application developer point of view.

When a thread is unblocked it resumes its execution at the return of the system call and will proceed as expected.

That was easier than expected!

Previously: build up the RW lock using combo of semaphores/mutex.

It's likely more efficient to have a self-contained construct that meets the goal.

A simple counter is probably insufficient. Why?

If the counter is currently 1, is that a reader or a writer thread?

So let's have two counters, shall we? One for readers and one for writers.

But things get interesting depending on whether you care about writer priority.

How should this work if we don't give priority to either side?

At first we might consider two counters; one for readers and one for writers.

If the readers counters is 7, is that seven readers currently in the room or seven readers waiting to enter?

It's hard to know and we'll need some more accounting.

We talked about the light bulb analogy for the way of understanding the reader-writer lock...

Idea: a boolean variable that says whether the lights are on.

Then a counter could be used to keep track of the number of readers currently in the room.

The reader-writers lock can have associated reader and writer queues.

Reader: try to set light switch from 0 to 1 with TAS.

If they succeeded, they're the first reader: proceed.

If they failed, why? Readers or writers?

When the reader is done, decrement.

If it falls to zero, unblock a writer if one waiting...
    Otherwise set light switch to 0.

If a writer wants to enter the room, try to change the light switch from 0 to 1.

If the writer succeeds, it can proceed.

If it fails, block the writer.

When the writer is done, unblock either all waiting readers or a writer.

Note that it's *all* the readers!

Can we prioritize writers here?

Yes, writers always choose writers at the end.

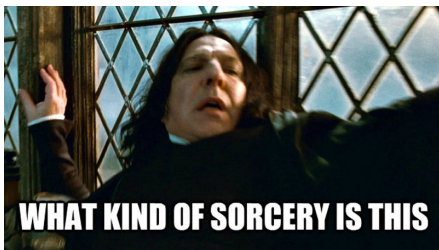There is a risk of starvation – is that okay?

That's not quite what we wanted.

Goal: when a writer is waiting, no new readers are allowed to enter the room.

How would we go about that?

When a reader wants to enter the room, take a look at the state of the waiting queue for writers.

If there is one, even if we would normally let the reader proceed, block the reader thread.

The logic requires us to try to change the light switch and also to maybe modify the counter and then decide based on that.

That's not atomic!

We may need a mutex internally here.

We discussed the implementation of a RW spinlock in Linux:

Counter	Flag	Interpretation
0	1	The spinlock is released and available.
0	0	The spinlock has been acquired for writing.
n ($n > 0$)	0	The spin lock has been acquired for reading by n threads.
n ($n > 0$)	1	Invalid state.

I'd say this validates our idea for how to implement RW Locks...