ECE 350: Real-Time Operating Systems Spring 2023

Lecture 4 — Concurrency Control Implementation

Jeff Zarnett *2023-03-06*

Concurrency Control Construct Implementation

In the concurrency course and the content just reviewed, we discussed at length the need for concurrency-control constructs and their purpose in ensuring the correct execution of the user program. Still, the introduction to the subject covered ideas about how concurrency control might actually be achieved. There were some solutions that did not work about flags and such.

One possible solution that works in an embedded system or very simple OS is disabling interrupts. This is pretty crude, but it does get the job done – if there are no interrupts, then we don't have process switches as a result of hardware operations completing, or the timer interrupt for switching between threads. That does permit certain bad behaviour, though, where one thread can disable interrupts and then just never re-enable them, guaranteeing that it can monopolize the CPU time. Or maybe it just accidentally exits before re-enabling interrupts and the whole system is stuck. And, of course, if there's an emergency situation, the system can't respond to that either. So disabling interrupts is dangerous even if it might sometimes be effective.

But as we saw, even if every process behaves in a very thoughtful and considerate way, if we have multiple processors, then disabling interrupts will not be sufficient [Sta18]. Where we landed was that for the mutex (binary semaphore) we could use the test-and-set instruction. Remember this?

Test-and-Set. The Test-and-Set instruction is a special machine instruction that is performed in a single instruction cycle and is therefore not interruptible. It is therefore an atomic read and write. The idea is that the Test-and-Set instruction returns a boolean value. When run, it will examine the flag variable (in this example, `i`) and if it is zero, it will set it to 1 and return true. If `i` is currently set to 1, it will return false. The meaning of the return value is clear: if it is true, it is the current thread's turn to enter the critical section. The Test-and-Set instruction is not actually implemented like this, but a description of its functionality in C is:

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

Because it is done in the single machine instruction, this works no matter how many CPUs are executing the same code in different threads concurrently (and there's some hardware magic around caches and cache coherence, but that's for another course). 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;
```

You will notice here that the assignment of `busy = 0` doesn't use something like a test-and-clear instruction or similar. If you wanted to use one it would make sense, but you don't really need it: if only one thread is ever in that critical section at a time, only that thread will be modifying the variable `busy`. Because the other access are atomic they will either succeed or fail in one step – and either the value is 0 or 1 (it won't be somewhere in between) – so this is grudgingly okay. That doesn't work for all scenarios, of course, and an analysis tool like Helgrind might call this out as being a possible race condition. So probably we shouldn't do that in reality.

Compare-and-Swap. That works okay for the mutex scenario, but doesn't work for the general, counting, semaphore where the values can be things other than 0 or 1 (locked and unlocked). For that we have the compare-and-swap, or sometimes it's called compare-and-exchange, instruction. Like test-and-set, it is implemented as a hardware instruction that is completed in one cycle and is uninterruptible. As before, it is not implemented like this, but a precise C-language definition looks like this:

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

The CAS routine will change the value of the integer `value` from `old_value` to `new_value` if it succeeds and make no change if it did not succeed. Why might it fail? It might fail if some other thread has modified the value in the meantime. That's why we get the actual value back as the return statement of the function: so we can update the old value in the current thread. If we wanted to change it from 1 to 0 and we find it's already 0, then we just need to try again changing it from 0 to -1. That might also fail, but 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. And we could prevent that risk entirely if we use an alternative approach below.

The initial guess for `old` can certainly be wrong; the initial attempt to set it will fail and we'll get the correct value. Take note also the need to do the compare-and-swap operation to increment the value as well – otherwise we'll have race conditions there. That wasn't a concern for the test-and-set approach because in that case because the only thread trying to change it from 1 to 0 was the one that got in, but this still relies on busy-waiting.

Alternative: if there is appropriate hardware support, you could use a simpler version of this where you just attempt to do an atomic increment or atomic decrement. That prevents the scenario where multiple attempts are necessary to get the value. Just make sure to choose the correct atomic primitive that returns the *new* value, since that will determine whether the calling thread should get blocked or not.

Regardless of whether we use the atomic decrement approach or the compare-and-swap, there is the big `WAIT FOR OUR TURN` comment there. That's covering up the fact that unlike the test-and-set approach, succeeding at the compare-and-swap routine execution doesn't mean it's actually the current thread's turn – just that it successfully updated the counter value. We actually need to wait until we know for sure that this thread is permitted to proceed.

How do we wait for it? That part's the job of the operating system. Where we left things off in ECE 252 was that sometimes we don't get the result that we want and if it's not our turn, then the operating system blocks the

process. No real details were provided as to how that happens. That sort of hand-waving of OS magic really just put off handling this scenario to later, except later is now.

Blocking and Unblocking

So the caller has tried to lock the mutex or wait on the semaphore; each of those was a system call. Inside the system call it uses one of the methods above – test-and-set, compare-and-swap, atomic operation – to do the assessment of whether the thread can enter. But it happens in the system call and most likely inside the kernel code. It doesn't have to be in the kernel, but it does have to report a negative result to the kernel, at the very least.

If the caller used a try-lock or try-wait sort of call, then they don't get blocked no matter the outcome; if they would have been blocked, simply send back a response that says "no" (usually, a non-zero return value) and maybe increment the counter again (depending on how the call to try-wait worked). Then the implementation is done and you can skip the remainder of this section.

If the thread is supposed to be blocked, that's not a very complex operation from the point of view of the operating system. Change the status of the calling thread to be blocked (so that it is not ready to run), and then choose another thread to continue execution. Which one? Scheduling decision, as you know, except scheduling is *also* something that we hand-waved away in the past and are about to find ourselves forced to revisit. Anyway, back to blocked threads.

Marking a thread as blocked or moving it to a "blocked" queue is insufficient, though. We will need some way of knowing that this thread is blocked on the particular semaphore or mutex it was accessing – hence, an implementation of blocking and unblocking will require, perhaps, that the concurrency control construct has an associated queue of the threads waiting for it. Or maybe when a thread is blocked there's somewhere in its PCB that says what it's waiting for. At that point the thread can just wait until its turn arrives

Eventually, whichever thread did lock the mutex will want to unlock it or some thread will post on a semaphore. If it's a counting semaphore, we always increase its internal counter by 1 and will unblock a waiting thread. For the mutex, we'll only change the internal counter from 0 to 1 if there is no thread waiting; otherwise the only thing to do is unblock a waiting thread. That particular implementation is not the only way to go about it, but it's important not to just set the mutex back to unlocked and forget about whatever threads are waiting – or worse, set it back to unlocked AND unblock one of the threads and allow it to proceed. Either one of those outcomes would ruin the mutual exclusion behaviour we want.

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, because it ignores things like thread priorities, but it does work. A more advanced system could consider thread priorities, to be sure, but could also even consider things like whether a deadlock is possible or more likely. Just keep in mind that even commercial operating systems don't do this, so the simple approach to dealing with deadlock of ignoring it is the most popular one.

The thread that's unblocked is marked as ready to run again (moved to the ready queue, perhaps?). And at this point it is again a scheduling decision as to which thread continues execution. But we knew that from doing all this from the application developer point of view.

When a thread is unblocked after waiting for the mutex or semaphore, it resumes its execution at the return of the system call and will proceed as expected. So it turns out, the actual implementation of blocking and unblocking of the various threads to have concurrency control is rather straightforward for both the semaphore and the mutex scenarios.

Condition Variables

Condition variables have a lot of similarities, but their semantics are different. Remember that we can have the lost-wakeup problem, but also that the condition variable is always paired with a mutex.

When a thread waits on a condition variable, it must be holding the mutex it has previously acquired to call the wait function. This releases the mutex, or unblocks some thread that was waiting for it. But the current thread always will get blocked, unconditionally, and will be waiting in the condition variable's queue.

If a thread signals on the condition variable, if any one thread is waiting, it is removed from the blocked queue of the condition variable, but it goes into the blocked queue for the mutex. If a thread broadcasts on the condition variable, then all the threads waiting in the condition variable queue are moved to be waiting for the mutex. Whether it's one thread or many, waiting on the condition to be fulfilled doesn't mean that they execute immediately. That would actually be bad.

Notice that no manipulation of the internal counter takes place as there is no internal counter to speak of: the behaviour of the conditional variable as described has no need for one. In a condition variable scenario, if no thread was waiting when a signal or broadcast takes place then no thread gets woken up. Hence, the lost-wakeup problem: if things happen in the wrong order somehow, then the signal is not received by any thread and a thread could be waiting forever. That might not be what you want. This isn't to say that condition variables are useless, but they fulfill a different purpose than just semaphore with broadcast.

So, the functionality as described here does not require a counter, and in fact, it would be incorrect to have one. In a semaphore, if the call to post takes place first it increments the counter and the thread that calls wait doesn't get blocked. So we cannot just use the semaphore implementation as the basis for the condition variable implementation. But that's okay, this is simpler anyway.

References

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