CS162 Discussion #04: Monitors and Deadlock

Announcements

- Project 1 initial design + individual portion due Tuesday, 09/27 @11:59pm
  - Sign up for design review slots 09/28 and 09/29 at http://goo.gl/KmRYu
  - Give Angela a hardcopy of your design doc!

Monitors Make Multithreading More Manageable

A monitor is a lock combined with zero or more condition variables. Using this combination of a lock and condition variables gives you flexibility to separate mutual exclusion from conditional/scheduling constraints. A lock is used to guard access to shared data; a condition variable is used while a thread already has access to the shared data (i.e., has possession of the lock) and allows the thread to wait for some condition to be fulfilled. You can think of a monitor as a general pattern for managing concurrent access to resources:

```
acquire lock
while condition not met:
  wait on condition variable
do stuff with shared data
release lock
```

From the standpoint of the thread executing this code, it looks very simple. All the thread has to do is lock the resource, wait around until its needs are met, do its job, and unlock. Under the abstraction, though, there's actually a lot more going on. The condition variable's `wait()` method makes the thread release its possession of the lock and go to sleep until a `signal()` wakes it back up, at which point it has to attempt to reacquire the lock again – and this could happen any number of times before the thread actually gets to go on and execute the rest of its critical section. (Why is it important that we use `while` instead of `if` when checking if the condition is met?) This allows the thread to sleep safely while inside the critical section.

Note that you could mimic this behavior by using semaphores. A semaphore initialized to 1 can act as a lock, and additional semaphores can be used to let threads wait on conditions. However, you have to be careful here; the order in which `p()` is called on each of the semaphores is important in ensuring the correct behavior. With this implementation, sleeping on your condition semaphore inside the critical section (i.e., calling `p()` on your condition semaphore after calling `p()` on your lock semaphore) will result in deadlock. Not only is this fact not readily apparent, it's also easier to make this kind of mistake, especially when the web of semaphores gets even more complicated.

What about just using a lock on its own, without any condition variables? You can certainly protect your critical section with just a lock, and it'll be correct in terms of synchronization. However, in the case that some condition you need isn't met, without using condition variables your thread will be forced to busy-wait. Busy-waiting is bad[1]…so don't do it!

[1] Well, busy-waiting is bad most of the time. As I briefly mentioned in section last week, there actually are some scenarios in which busy-waiting can be more advantageous than sleeping. For example, if the scheduler overhead of sleeping exceeds your expected wait time, busy-waiting may actually end up being cheaper.

Disallow Deadlocks by Denying Demands

We've already seen that threads can cause all sorts problems for each other just by not being synchronized. However, they can still get into trouble even if they are synchronized…by getting into a deadlock. A deadlock happens when threads are blocked in such a way that they are unable to continue running. More specifically, we have the following four conditions necessary for deadlock:

- *Mutual exclusion.* Only one thread at a time is allowed to use each resource.
- *Hold and wait.* There must be a thread holding at least one resource and also waiting to acquire a resource held by another thread.
- *No preemption.* Resources must be released voluntarily by the threads holding them; resources can't be forcibly taken away from a thread.
- *Circular wait.* There must be a set of threads waiting in a cycle.

Deadlock can happen if and only if all of these conditions are satisfied. (Can you prove it's not possible to have deadlock if any one of the conditions is false?) One of the ways we can avoid deadlock is by simulating resource allocation using the Banker's algorithm. To run the Banker's algorithm, we need to know:

- How many of each resource is available
- How many of each resource each thread might request (maximum)
- How many of each resource each thread still needs
- How many of each resource each thread is currently holding

We only allow threads to acquire resources if the number requested does not exceed the amount it needs and is currently available. If we are able to grant the request, we simulate the allocation, updating all resource counts as necessary, and then check to see if the resulting state is safe. A state is considered safe if there is some possible sequence of requests from that state that will allow all threads to finish; if there is no such sequence, then the state is unsafe and we know that the request shouldn't be granted because it will result in a deadlock. Otherwise, we go ahead and grant the thread's request. We can continue allocating resources in this way, ensuring that each request we grant results in a safe state, until each thread is finished.

Note that when checking that a state is safe, Banker's algorithm only comes up with a hypothetical sequence of resource requests, acquires, and releases that will not result in deadlock…these requests aren't actually granted, and just because a safe sequence exists doesn't mean that requests will necessarily be made this way when the threads run.

Another possible problem we might have is starvation. Starvation happens when a thread has a possible way to terminate, but doesn't get a chance to run for some reason (unfair scheduling, etc.). Even though this thread isn't getting to run, starvation does not count as deadlock and won't be detected by the Banker's algorithm.