CS162 Discussion #5: How Many Bankers Does it Take To Fix a Deadlock?

Announcements

- Project 1 code due Tuesday, 10/05 @ 11:59pm – use `submit proj1-code`
  - Submit to test autograder with `submit proj1-test`
  - Group evaluations and final designs due 10/06 @ 11:59pm
- Project 2 starts on Tuesday too
- Come to Nachos Night on Monday, 10/04 @ 7pm in 711 Soda
  - There will be food! And you can work on your project! And I'll be there :)

If you kill a livelock, do you get a deadlock?

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.

Nachos Feedback

As you're working on your design for phase 2 this coming week, please consider the following points, which I've put together after design reviews and reading initial designs for phase 1. You might find them helpful for future design docs and design reviews:

Design Docs:

- No need to repeat the spec for each task – you can assume the TAs already know what the tasks are, and it'll save you some space on your doc too.
- Break up pseudocode into small chunks with some brief explanation. Also, make sure lines of pseudocode don't wrap around to the next line without being indented correctly…these can get pretty hard to read!
- Remember to fully explain what your methods are doing; don't rely on your pseudocode to do all the explanation. All the functionality you're planning to implement should be described in adequate detail – don't assume your TA is going to fill in ambiguities for you.
- Try to proofread before you submit.

Design Reviews:

- Make sure each member knows at least what all the important methods involved in the design are supposed to do at a high level, even if they don't know all the implementation details.
- Don't just read off of your design doc when you're explaining your design.
- Make sure to bring a copy of your doc that you can leave with your TA.
- Each member should contribute to the discussion in some significant way.