CS162 Discussion #3: Time to Get Synchronized

Announcements

- Project 1 has started
    - Project 1 initial design is due Monday, 09/27
    - Design reviews Tuesday, 09/28 and Wednesday, 09/29…keep an eye out for times

Semaphores, Locks, and Condition Variables…Oh My!

Last week we saw how context switching between threads could cause unexpected, nondeterministic bugs – these are especially frustrating! We can prevent (or at least mostly prevent) these bugs from happening by using some tools for synchronization. Our goal is to protect shared resources from being accessed at the same time by multiple threads if these accesses may conflict with each other. Any chunk of code that should only be executed by one thread at a time is called a *critical section*. We could protect critical sections by disabling interrupts completely, but this is usually not a preferred solution. Therefore, we have some other useful tools to use for synchronization:

- A *semaphore* is initialized with an integer and has two operations, `p()` and `v()`. The integer represents how many resources or "open spots" are currently remaining. A call to `p()` will check if the integer's value is positive. If the value is positive, it will decrement and allow the calling thread to access the requested resource. If the value is 0, the thread will be put to sleep until the resource becomes available. This is achieved when a thread finished with the resource makes a call to `v()`, which will re-increment the value.
- A *lock* provides mutual exclusion by preventing a thread from doing something. It has the operations `acquire()` and `release()`. Locks are associated with some kind of resource; threads should `acquire()` before entering a critical section and `release()` afterward. If a thread attempts to acquire the lock while another thread still has possession of it, that thread will be put to sleep until the lock is released.
- A *condition variable* allows threads to wait on some condition. Condition variables have two operations, `wait()` and `signal()`. `wait()` puts a thread to sleep on a queue, and `signal()` wakes a thread up from the queue.
- A *monitor* combines a lock with zero or more condition variables. Threads attempt to acquire the lock, but may also choose to sleep on a condition variable while inside the critical section until some condition has been met. With Mesa-style monitors (which Nachos uses), threads are marked as "ready" after waking from condition variables and must reattempt to acquire the lock.

Note that any form of synchronization must involve some kind of waiting, whether it is sleeping or busy waiting. (Usually sleeping seems to be the better option…but when might busy waiting be advantageous?)

Catching Bugs

As you are working on your design and implementation for the project, you should definitely think about how you will test your code. Testing nachos code is not a trivial task, so it will pay off to come up with some testing strategies before you start. Because there are a lot of dependencies, writing isolation/unit tests can be somewhat challenging. Here are some tips that might be helpful:

- *Mock out dependencies.* If you want to test an object or class in isolation that depends on another object or class, mock it out. This often means creating a skeleton of the dependency and giving it function stubs and other properties you can easily control. This way, you don't depend on the correctness of another unpredictable piece of code when you're testing something.
- *Check code coverage.* A good (but rough) rule of thumb is to always make sure your tests have run through every, or at least almost every, line of code you're testing. It's important to check path coverage too – make sure all possible different paths of execution have been covered.
- *Test repeatedly with randomized parameters.* Especially for programs that can have nondeterministic behavior, running the same test many times may help you find bugs that only happen some of the time. You can also give your tests some element of randomness to try to catch edge cases.
- *Use testing tools.* There are a number of tools for testing out there, such as JUnit and EasyMock. These can make part of the testing process easier for you, but might also take some time to set up and learn if you don't already know how to use them.

Nachos also has some built in features that can help you debug:

- *Debug flags.* In your program arguments, you can use the flag `-d` with any combination of letters representing debug flags, which you can define inside classes yourself. These are very useful for selectively printing debug statements in your code using `Lib.debug()`.
- *Assert statements.* You can use `Lib.assertTrue()` to put assert statements into your code and make sure your invariants hold when they should.
- *Nachos configurations.* You can edit the `nachos.conf` file to change some settings, including what kind of scheduler is used. The kinds of settings you can change differs for each project.
- *Randomize context switches.* You can use the `-s` flag in program arguments to change where `yield()` gets inserted in the code when you run it.