CS162 Discussion #03: Synchronization

About your TA

Name:	Angela Juang	
Email:	ajuang@berkeley.edu (Put [CS 162] in the subject line)	
Website:	http://inst.eecs.berkeley.edu/~cs162-ta	
Sections:	Tuesday	5-6pm, room TBA
		6-7pm, room TBA
Office Hours:	Thursday	10-11am, 651 Soda
	Friday	2-3pm, 651 Soda

Announcements

- Project 1 starts THIS Thursday, 09/15
  - Specs aren't up yet – keep an eye out for updates
  - Familiarize yourself with Java Threads (`java.lang.Thread`)
  - You will NOT be allowed to use anything out of `java.util.concurrent` except what we explicitly state you can use in the spec.

I'll be watching…and waiting

Last week we talked about how threads can context switch to achieve the illusion that they are running at the same time. However, context switches may happen at any time during a thread's execution. What happens if the thread is in the middle of accessing some resource when it is forced to yield to another thread that also wants to access the same resource?

For example, consider a simple case where two threads each want to: 1) check how many items are in a list, and 2) if there are still items in the list, remove one. Now suppose the list has one item and Thread A checks, sees that there is one item, and is about to remove that item when a context switch gives execution over to Thread B. Thread B does the same, goes ahead and removes the item, and then yields back to A. Thread A thinks it has already passed the check and resumes trying to remove an item from the list…but there's none left! We can see that if more than one thread is trying to access the same resource, an unlucky choice in scheduling could cause a lot of problems – unexpected, nondeterministic bugs are especially frustrating. Therefore, some kind of synchronization is needed to ensure that context switches won't cause threads to jeopardize each other.

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:

- *Atomic instructions* such as `test&set` and `compare&swap` read and write values to/from memory atomically (can't be interrupted), and are implemented in hardware. `test&set(addr)` takes whatever is stored at `addr`, replaces it with a 1, and returns what was originally stored

there. `compare&swap(addr, reg1, reg2)` stores what's in `reg2` into `addr` if `addr` contains what's in `reg1` and returns success, otherwise it returns failure. Both `test&set` and `compare&swap` can be used to implement locks (how?).

- 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, threads are marked as "ready" after waking from condition variables and must reattempt to acquire the lock.

When dealing with synchronization, be careful of deadlock! Deadlock happens when all threads are stuck waiting for a resource and none of them can progress, causing the system to just stay stuck there forever until some intervention is made (i.e., killing the processes). For example, if thread A is waiting for a lock thread B is holding, and thread B is waiting for a lock thread A is holding, neither of them will ever get anywhere because they're both waiting for each other. We'll go into more detail later in the course how to deal with deadlock and try to prevent it. For now, try to take precautions when dealing with locking (in your first project!) to avoid allowing situations like this to arise.

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?)

A Note on Groups

Make sure you meet with your group early and often, and keep communication open so each member knows what's going on and what he/she should be doing. Keep in mind that it's not necessary for each member of your group to contribute in the same way – play to each member's strengths! (No, this does not mean that one of your members can say that he has no strengths and just slack off.)

If you do begin to have concerns with any part of your group, please talk to Angela about it as soon as possible before it becomes a problem. You are welcome to arrange for either private or group meetings at any point in the semester to talk about concerns with group organization or projects – check Angela's website for details on how to request a meeting!