# CS162 Discussion #4: More Synchronization Issues?

## Announcements

- Project 1 initial design is due Monday, 09/27 – submit your pdf online with glookup
- Design reviews Tuesday, 09/28 and Wednesday, 09/29 – make sure you sign up on the CS 162 website! (Tuesday reviews in 310 Soda, Wednesday reviews in 380 Soda)

## More On Monitors

Last week we introduced the concept of using a monitor for synchronization – 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. Remember from last time that 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! (Especially in Nachos! You'll incur serious penalties on your project if we find any busy-waiting in there.)

**Threads Can Have Friends Too**

Up until now, we've only really talked about synchronization with respect to one thread having access to some resource at a time. But what if we want to let more than one thread access the same resource at the same time? Or what if we want to let all the threads from a certain group have access to the resource at the same time, but exclude all other threads? In this case, threads can't just acquire the lock on the resource before using it and release it after, because they'd be blocking other threads that should also have a right to access the resource simultaneously. Instead, we can use a check-in/check-out system in which the critical sections consist of updating variables that indicate which threads are currently accessing the shared data. This way, the actual access to the data can happen outside the critical section, so that multiple threads can do it at once. This is the basis for a solution to the "readers-writers problem," which will be discussed in upcoming lectures.

**Design Review Madness**

Since your design reviews are coming up next week, it'll probably be useful for you to know what exactly will be happening at the review and how to prepare for it. Here's a basic overview of how design reviews will be held.

Each review will last about 20 minutes. You'll be expected to present your design to your TA – make sure each group member participates! Not every member has to know every part of the design equally well, but make sure each person knows the general concept of your design for the entire project and also knows at least one substantial part of the design well enough to explain it clearly and answer questions about it. The point of the review is to make sure your group has a good understanding of the project and a reasonable solution, so expect some questions and feedback from your TA. Make sure every member of your team shows up!

You'll also want to take some time preparing your design doc so that it's easy to read and understand. Here are some quick tips for enhancing the readability of your design doc:

- Break your doc down into sections, one for each task.
- List out added fields at the top of each section, along with short descriptions for each one.
- Write pseudocode when appropriate. Emphasize parts of your pseudocode that are important (for example, where you change interrupt status, acquire/release locks, etc.)
- Take some time to format with indenting, headings, etc.
- Make sure the names of all your team members are on the front.

[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.