

## Section: Sample Problems With Synchronization

### 1 Announcements

1. New slower pace for section since we don't have to race to get through project material
2. Course Survey's
3. Video Taping FOR ME!
4. Design Reviews coming up (BE ON TIME! ... the times are not Berkeley times)
5. Once you have my design review, you're welcome to start coding the project

### 2 Errata from last time

Last time I mentioned that you guys have to nest your locks. This is true, but when i tried to do the example in class i tried to show that it breaks when i reversed the lock release. This of course doesn't work. So the key idea here is that Lock acquisition must be nested consistently. If you don't nest lock release you might get into performance problems but the solution will still be correct. **however, remember to nest your lock acquisition to avoid deadlock** We'll be talking much more about deadlock next week.

### 3 Clarification on Monitors

So far I think the most abstract concept in the class has been the idea of a Monitor. The book gave one definition and Professor Kubi gave another definition in the class. Professor Kubi said that a Monitor is a lock and zero or more condition variables that protect access to shared data. The book however says "a monitor presents a set of programmer-defined operations that are provide mutual exclusion within the monitor." Notice that they are almost the same. The higher-level take away from these confusing and partly conflicting definitions is that monitors are high level programming and synchronization constructs that abstract away lock management away from the user level to further prevent programmers from doing stupid things with parallel code. Since writing parallel code is hard we need as many mechanisms as we can to prevent this type of behavior. When I think "monitor," I take it to mean a special tool or object "monitoring" access to shared data structures. If you feel the concept is very abstract, don't worry you aren't alone. For example if you create a java class and for manipulation of a shared queue that handles the synchronization internally in such away that no synchronization problems are exposed to the user, then you've just created a monitor. Take a look at the profs lecture notes for more examples.

## 4 Clarification on Condition Variables

The main aim of a condition variable is to have a much finer lock control than `lock.acquire` and `lock.release` give us. With condition variables its easier to build the idea of “i want to give this lock to this thread or this type of thread next.” With `lock.acquire` its a much more heavy-weight process since everyone will be contending for the lock. Think back to the readers/writers example in lecture. If we have 2 producers and 100 consumers that are fighting for the same lock, the consumers will win most of the time. Even if we set up priorities right, we still have to worry about the problem of priority inversion and chaining priority properly, starvation, etc, etc. However condition variables provide a clean way out. On a particular lock, we can tell the only the producers to wake up and fight for the lock in which they’ll get a 1/2 chance to run instead of 2/100 . Thus this finer grain control is essential to getting systems in which we have many threads to make progress. Solution with `Lock.Acquisition` can lead to eventually the right answer, but it might take a lot longer.

**Questions?**

## 5 Sample Problems

Since we are finishing up synchronization i wanted to spend some time in this discussion section talking about sample problems about synchronization and just go through the solutions. They will be a little more involved than the ones you’ve seen in class.

### 5.1 Hardware Primitives

**Exercise:** *lets say that `swap` atomically swaps a register and memory location, implement `test-and-set` with `swap`. implement `swap` with `test and set`*

---

**Algorithm 1** Lock Acquire

---

```
while test-and-set(guard) do
  do nothing
end while
if value == BUSY then
  put thread on wait queue
  atomically go to sleep and set the guard to 0
else
  set value to BUSY
  set guard to 0
end if
```

---

**Exercise:** *Professor Kubi mentioned this bug in class and i want to make sure I touch upon it because it exposes a new class of bugs that not many of you have seen before. There is still a very small timing problem with the `lock acquire` and `release` methods in `Algorithm 1` and `Algorithm 2` that can cause us to busy wait for a long time. What is it? (this type of bug is called a performance bug)*

---

**Algorithm 2** Lock Release

---

```
while test-and-set(guard) do
  do nothing
end while
if anybody on the wait queue then
  take the thread of the wait queue
  put the thread on the ready queue
else
  set value to FREE
end if
set guard to 0
```

---

**Exercise:** *In the last exercise and class we keep alluding to the fact that busy waiting is bad. Then why on most large multi-processors are all locks and thread synchronization tools implemented with busy waiting rather than yielding? Why is busy waiting on SMP style (multiprocessors, hyperthreaded, and multi-core) machines ok? What is the key hardware primitive you need to get it right? (notice that the problem became significantly harder*

## 5.2 Condition Variables and Semaphores

In this section we'll go through a couple of sample exercises with condition variables and semaphores to make sure that you understand what they are doing and how they can be implemented.

**Exercise:** *Lets say that you have a perfect semaphore implementation, how would you implement a condition variable with **one** semaphore. Condition.java in the nachos code uses multiple semaphores and I think it is slightly overkill*

**Exercise:** *How would you build semaphores by disabling and re-enabling interrupts?*

**Exercise:** *Lets say that you have a perfect implementation of condition variables and don't worry about how it works under the covers. How would you build a semaphore with condition variables?*

The main aim of all these exercises is to show that it is possible to build one synchronization primitive from the others. By the time the midterm rolls around you should be able to do these types of conversions because they really *test* you're understanding of what the different synchronization primitives do.