

Computer Science 161: Operating Systems

Section Handout: Week 3

CS161 Course Staff / Andrew McCollum
{cs161, mccollum}@fas.harvard.edu

February 25, 2007

1 Administrivia

- (Teams): Hopefully you have your teams set by now, and are already hard at work on some wicked ASCII art to represent your chosen animal. If not, email cs161@fas as soon as possible so we can get you straightened out.
- (Code): Once you have ASST1 all wrapped up, start thinking about whose code you are going to use going forward. This also means that you and your partner need to start agreeing on things such as style (variable names, comments, etc.) and CVS usage. Since you are now working on the same source tree, you need to be careful about not stepping on each other's toes or blowing away a partner's changes with a bad commit (although you can always check out previous versions for this very reason).

2 Quick API Review

2.1 Semaphores

- P() - decrement semaphore, or if 0, wait until semaphore is available, then decrement. Used to acquire a resource.
- V() - increment semaphore. Used to signal the freeing of a resource.

2.2 Locks

- Acquire - claim ownership of a lock, or if unavailable, wait until current owner releases and then claim it.
- Release - release ownership of a lock so that another thread can claim it.

2.3 Condition Variables

- Wait - wait for another thread to signal/broadcast on this condition variable
- Signal - signal exactly one waiting thread.
- Broadcast - signal all waiting threads.

3 Sample Synchronization Problems

3.1 The Surreal Nightclub

It is a little known fact that in Boston there is a small, disheveled nightclub that goes by the name of “The Kinky Monkey.” This nightclub has very particular rules regarding admission, and these are as follows: only monkeys and ponies are allowed entrance, but in particular, a monkey must courteously gesture a pony into the nightclub before it can enter. As soon as the pony is gestured in it may enter, but the monkey must wait to be signalled by the pony. Similarly, a pony cannot enter on its own: it must be gestured in by a monkey. Your task is to devise a synchronization protocol to implement the monkey and pony threads. Use semaphores.

- How many semaphores do we need?
- What steps should the monkey take?
 -
 -
- What steps should the pony take?
 -
 -
- Is it guaranteed that the pony, given a signalled monkey, will signal the same monkey in return?
- Who enters the nightclub first?

3.2 Fast Food Drive-Thru

Finally, we’re going to teach you some skills that you will be able to put to use in your life after Harvard!

You’re in charge of running a fast food restaurant and you have a limited number of cooks making food. Customers can either make orders from the counter or at the drive-thru. You would like to prioritize the drive-thru customers over the walkins because people waiting in cars are more impatient.

Also, there is some chance that a customer (either walkin or drive-thru) may have an error in their order, in which case they will loudly complain which prevent the placement of any new walkin (but not drive-thru) orders until they leave. Complainers will politely wait for other complainers to finish before they start complaining. Hard life, huh?

How can you solve this problem using the synchronization primitives we’ve learned class?

3.2.1 First Try

Walkin Customer:

```
acquire(drive_thru_mutex);
/* Check for cars at the drive-thru */
while (drive_thru_cars) {
    release(drivethru_mutex);
    sleep(1);
    acquire(drivethru_mutex);
}
release(drive_thru_mutex);
```

```

/* Try to order */
acquire(complaint_lock);
P(cooks_sem);
release(complaint_lock);

/* Order food */

if (not_what_i_ordered(food)) {
    acquire(complaint_lock);
    /* Complain */
    release(complaint_lock);
}

V(cooks_sem);

Drive_thru :

/* Show that we have arrived at the drive-thru */
drive_thru_cars++;

/* Since we're in the car, we aren't bothered by complaints */
P(cooks_sem);

/* Order food */

if (not_what_i_ordered(food)) {
    acquire(complaint_lock);
    /* Complain */
    release(complaint_lock);
}

V(cooks_sem);

/* Show that we are leaving the drive-thru */
drive_thru_cars--;

```

For this problem, ironically, ignore the possibility of starvation. However, there are at least three errors in the above code. What are they? How can we improve this code?

4 Solving Synchronization Problems

We want to avoid race conditions, deadlocks and starvation. How can we do this?

4.1 Avoiding race conditions and starvation

- Model your problem:
 - What does your stoplight look like?
 - What are its states?

- Write down rules for each component (thread)
 - How does your stoplight change its states?
 - How does a car know that it should enter the intersection?
 - What should the car check for before entering the intersection?

4.2 Avoiding deadlocks

These methods were discussed in class in relation to the Dining Philosopher's problem.

- All or nothing
 - Either we get all the resources we need, or none of them
 - Problem: This can become very inefficient
- Break Conflicts
 - Somehow revoke rights to a resource to resolve conflicts
 - Problem: Difficult to handle failure states
- Locking Order
 - Acquire resources in a pre-specified order
 - Problem: Sometimes it isn't obvious how to do this when you have different access patterns.
 - However, this is generally your best choice

5 General Synchronization Advice

Synchronization is hard. Here are some tips to make it a little easier.

- When to synchronize?
 - Modifying global variable in different threads
 - Protecting state during forced sleep (i.e. I/O)
- Pick the right primitives
 - Locks - Used to make critical sections, especially modifying shared state
 - Semaphores - Used when two threads interact with each other cooperatively
 - CVs - Used when a thread needs to wait for a condition to be true before proceeding
 - These are just common examples – be ready when things don't match any one category, or you need to mix primitives.
- Organize and limit conflicts
 - Try to modularize your code to minimize critical sections which require synchronization
 - Keep related synchronization close together, not spread across different files
 - Build things iteratively when possible to cut down on the possible causes of the problem
- When in doubt, draw pictures
 - Draw graphs of resources and their consumers
 - List the order in which things are acquired
 - Look for inconsistent orders of acquisition and circular dependencies