

# Assignment 2

## Synchronization

**Written Answers Due:** 11:00 PM, Friday, February 6, 2004

**Final Patches Due:** 11:00 PM, Friday, February 6, 2004

In this assignment, which is the first group programming assignment, you will implement synchronization primitives for OS/161 and learn how to use them to solve several synchronization problems. Once you have completed the written and programming exercises you should have a fairly solid grasp of the pitfalls of concurrent programming and, more importantly, how to avoid those pitfalls in the code you will write later this semester.

To complete this assignment you will need to be familiar with the OS/161 thread code. The thread system provides interrupts, control functions, and semaphores. You will implement locks and condition variables.

## Preliminaries

Recall that you *must* have your path correctly set to undertake all CS 182 assignments. If you haven't done so already, put the appropriate line in your `.login` or `.bashrc`. Do it.

If disk space is tight (and it may be) you can remove your files from Assignment 1—you should no longer need them. To remove these files, run

```
cd ~/cs182
rm -rf assign1
```

Then, check out the code for this assignment by running the following commands:

```
cd ~/cs182
cs182checkout assign2
cd assign2
./setup
```

As in the first assignment, when `./setup` finishes, it will have built all of the user-level commands (such as `/sbin/reboot`) and libraries for OS/161, but *not* the kernel itself.

Note that from this assignment onwards, you and your partner will be using the same, *shared*, repository. You can both check out copies of the assignment, but in order for your partner to see your files, you must use the submit system. To see any changes checked in by your partner, run `cs182resync`.<sup>1</sup> In addition to merging any changes your partner has checked in (indicated by the letter U next to a filename), `cs182resync`

---

1. If you have files open in an editor, *save them* and quit the editor before running `cs182resync`.

also reveals which files you have modified but not yet checked in (marked with an M), and which local files aren't in the repository at all (marked with a ?). You should *always* run `cs182resync` before running one of the commands to check in changes if there is the slightest chance your partner could have changed any files. The fastest way to check in changes you've made so that your partner can see them is to use `cs182checkin`, but remember that changes checked in with `cs182checkin` don't count as submitted until you run `cs182submit`.<sup>2</sup>

Both `cs182resync` and `cs182checkin` only work on the files in the current directory and its subdirectories. If you are deep down in the directory structure and run `cs182resync`, you won't see changes your partner has made to files in directories higher up (until you `cd` into a high enough directory and run `cs182resync` again).

## Building the Kernel

The build process is identical to the one you used for Assignment 1, except that you will configure the build using the HW2 configuration file. Make sure your current directory is `~/cs182/assign1`, and then run

```
cd src/kern/conf
bash config HW2
```

Once the kernel is configured, you can build it by typing

```
cd ../compile/HW1
make depend && make && make install
```

Once it's built, you can test it by changing your directory to your virtual root directory as follows

```
cd ~/cs182/assign1/root
sys161 kernel
```

(The differences in configuration between this assignment and the previous assignment are that the timer interrupt happens more frequently and that `sys161.conf` in the root directory gives the machine more RAM (2 MB).)

## Code Quality

In this and subsequent programming assignments you will be developing a *patch* to OS/161 to implement a particular feature. This patch will be voted on by the other

---

2. As you'll have found in the first assignment, the `cs182submit` command is a little slow because the process of submission tags every file in the repository—even ones you haven't changed.

members of the class, and assessed according to the following criteria:

- Your code should work correctly
- Your code should be cleanly written and easy to follow (for someone else in the class)
- Your code should be sufficiently commented
- Your code should match the coding style of OS/161

In other words, the skills you learned in CS 70 are still needed.

## Wiki Component

To implement synchronization primitives, you will have to understand the operation of the threading system in OS/161. It may also help you to look at the provided implementation of semaphores. Finally, your understanding of the threading subsystem will not be complete without understanding the operation of the scheduler.

This component is not graded in the conventional sense—completion of this part is covered under the course's *participation* requirement. Every member of the class should should post the answer to one of the questions below on the course's Wiki and understand the answers to *all* of the questions. There are fewer questions than people in the class because you may also correct or amplify someone else's answer. If all of the questions have been answered, you should think of a closely related question and both ask and answer it.

In class I may ask you about the answers to these questions, *or* ask you closely related questions whose answers you should know from answering the questions below.

### Thread Questions

- W1. What happens to a thread when it exits (i.e., calls `thread_exit()`)? What about when it sleeps?
- W2. What function(s) handle(s) a context switch?
- W3. How many thread states are there? What are they?
- W4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?
- W5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

### Scheduler Questions

- W6. What function is responsible for choosing the next thread to run?

- W7. How does that function pick the next thread?
- W8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

### Synchronization Questions

- W9. Describe how `thread_sleep()` and `thread_wakeup()` are used to implement semaphores. What is the purpose of the argument passed to `thread_sleep()`?
- W10. Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

## Written Problems—Deadlock Identification

The following problems are designed to familiarize you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

Place the answers to the following questions in `answers.txt` in the `handin` directory. In addition, provide information on how you split up the task of answering these questions in `whodidwhat.txt`.

- P1. If the code below is executed by two threads, with the first running the `me` function and the second running the `you` function, it can deadlock. Give a sequence of execution and context switches in which these two threads can deadlock.

You should assume that `mutex` and `data` are binary semaphores that have been initialized in the usual way.

Lay out your answer using two columns, one for the first thread (running `me`) and one for the second thread (running `you`), much like the way that kernel debugging was explained in the first assignment.

```

void me()
{
    P(mutex);
    /* do something */
    P(data);
    /* do something else */
    V(mutex);
    /* clean up */
    V(data);
}

void you()
{
    P(data);
    P(mutex);
    /* do something */
    V(data);
    V(mutex);
}

```

- P2. Propose a change to one or both of the functions from the previous question to make deadlock impossible for `you` and `me`. What general principle do the original threads violate that causes them to deadlock?

- P3. Below are more functions that will each be executed by a separate thread. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free. (Assume that mutex, file1 and file2 are OS/161 locks.)

```

void laurel()
{
    lock_acquire(mutex);
    /* do something */
    lock_acquire(file1);
    /* write to first file */
    lock_acquire(file2);
    /* write to second file */
    lock_release(file1);
    lock_release(mutex);
    /* do something */
    lock_acquire(file1);
    /* read from first file */
    /* write to second file */
    lock_release(file2);
    lock_release(file1);
}

void hardy()
{
    /* do stuff */
    lock_acquire(file1);
    /* read from first file */
    lock_acquire(file2);
    /* write to second file */
    lock_release(file1);
    lock_release(file2);
    lock_acquire(mutex);
    /* do something */
    lock_acquire(file1);
    /* write to first file */
    lock_release(file1);
    lock_release(mutex);
}

```

- P4. The thread subsystem in OS/161 uses a queue structure to manage some of its state. This queue structure does not contain any synchronization primitives. Why not? Under what circumstances should you use a synchronized queue structure?

## Coding Component

In this portion of the assignment, you will flesh out one part of OS/161 and implement some code that will serve as a test of both your implementation of locks.

### Implementing Mutual Exclusion Mechanisms

- C1. Implement locks for OS/161. The interface for the lock structure is defined in kern/include/synch.h. Stub code is provided in kern/threads/synch.c.
- C2. Implement condition variables for OS/161. The interface for the cv structure is also defined in synch.h and stub code is provided in synch.c.

### Solving Synchronization Problems

The following problem will give you the opportunity to write some fairly straightforward concurrent programs and get a more detailed understanding of how to use

threads to solve problems. We have provided you with basic driver code that starts a predefined number of threads. You are responsible for what those threads do.

When you configure your kernel for HW2, the driver code and extra menu options for executing your solutions are automatically compiled in.

- C3. Professor Thom has a number of cats and (unfortunately) a number of mice that inhabit her home on Mt. Baldy. When Professor Thom first moved in, the mice were no match for the cats and many died. The two that remain are the sneakiest—they realize that they must *never* be seen. The mice survive by stealing pieces of the cats' food. Of course this is a dangerous maneuver, but the smell of the food masks the smell of the mice and the bowls are in a dark corner, so they'll only be discovered if the cats try to eat from one of the food bowls while the mice are there stealing food.

There are two food dishes, six cats (not really, but it makes the problem more interesting), and two house mice.

Your job is to synchronize the cats and mice so that no mice get eaten. This will require you to sometimes make cats or mice wait a moment for their food. You can assume that if a cat is eating at either food dish, any mouse attempting to eat from the other dish will be seen and eaten and that when cats aren't eating, they will not see mice eating. Only one mouse or cat may eat at a given dish at any one time. You should also assume the bowls always have plenty of food.

Develop *two* solutions to this problem:

- (a) Using semaphores, fleshing out the skeleton code provided in `catsem.c`
- (b) Using condition variables and locks, fleshing out the skeleton code provided in `catlock.c`

You can run the functions defined in these files from the kernel's main menu.

The provided code (which is located in `kern/hw2/`) only forks the required number of cat and mouse threads—you will need to do the rest. Each cat and mouse thread should identify itself (by printing its name) and which bowl it is eating at when it begins and ends eating. Simulate a cat or mouse eating by calling `clocksleep()`.

Your solution must not be prone to starvation, either of cats or mice.

### Patch Submission

- C4. Make a patch for your implementation of locks and place it in the `handin` directory. Full instructions for making a patch will be mailed to the class.

## Notes & Tips

### Concurrent Programming with OS/161

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all the constraints applied to them and that they do not deadlock.

### Built-In Thread Tests

When you booted OS/161 in the last assignment, you may have seen the options to run the thread tests. The thread test code uses the semaphore synchronization primitive. You should trace the execution of one of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should be able to step through a call to `mi_switch()` and see exactly where the current thread changes.

Thread test 1 (`tt1` at the prompt or `tt1` on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 (`tt2`) prints only when each thread starts and exits. The latter is intended to show that the scheduler doesn't cause starvation—the threads should all start together, spin for awhile, and then end together.

### Debugging Concurrent Programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, it complicates the process of debugging your concurrent programs.

The random-number generator used to vary the time between these `thread_yield()` calls uses the same seed as the random device in System/161. Thus you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into random device by editing the “random” line in your `sys161.conf` file. For example, to set the seed to 1, you would comment out the existing line that configures the random device and add a replacement line that reads as follows:

```
28 random seed=1
```

We recommend that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are confident that your threads do what they are supposed to do, set the random device back to the `autoseed` setting. Using different random seeds should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.