

Assignment 2

Synchronization

Wiki Answers Due: 1:15 PM, Thursday, September 14, 2006

Final Patches Due: 11:59 PM, Thursday, September 21, 2006

In this 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 (which will should be a useful reference when implementing locks and condition variables).

Preliminaries

Recall that you *must* have your path correctly set to undertake all CS 134 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.

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

```
mkdir -p ~/courses/cs134
chmod go-rx ~/courses/cs134
cd ~/courses/cs134
svn checkout svn://inuyasha.cs.hmc.edu/cs134/fall06/pairs/<group>/assign2
                                     (i.e., "<group>" is the name of your group)
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.

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 that your current directory is `~/cs134/assign2`, and then run

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

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

```
cd ../compile/HW2
make install
```

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

```
cd ~/courses/cs134/assign2/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).)

If you run the kernel and choose menu option 1a, you should see output roughly as follows:

```
OS/161 kernel [? for menu]: 1a
There are 6 cats, 2 mice, and 2 bowls.
Cath created (Cath is a cat)!
All done!
sleep: Dropping thread Cath
scheduler: Dropping thread Clay.
scheduler: Dropping thread Cody.
scheduler: Dropping thread Cher.
scheduler: Dropping thread Carl.
scheduler: Dropping thread Cole.
scheduler: Dropping thread Mike.
scheduler: Dropping thread Mimi.
panic: Assertion failed: active_animals == 0, at ../../hw2/catsem.c:211 (catmousesem)
sys161: 24570380 cycles (17332652k, 0u, 7237728i)
sys161: 9510 irqs 0 exns 0r/0w disk 3r/972w console 0r/0w/1m emufs 0r/0w net
sys161: Elapsed real time: 2.654635 seconds (9.25565 mhz)
sys161: Elapsed virtual time: 0.982815200 seconds (25 mhz)
```

(Some of the output lines may be interleaved with each other. This phenomenon is normal at this stage, and results from multiple threads printing at the same time.)

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

- Your code should work correctly;

- Your code should be cleanly written and easy (for someone else in the class) to follow;
- You should supply documentation with your patch;
- Your code should be sufficiently commented;
- Your code should match the coding style of OS/161 (not just formatting, it should feel like it fits in with “the OS/161 way” as best you understand it).

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 pair should post the answer to two 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.

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()`?

W11. 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, *locks* (a.k.a. *mutexes*) and *condition variables*. You will also write some code to test of both your implementation of these features.

Read the entire coding component before you begin. Doing so will help you work out how much work you have to do, and give you things to be mulling in the back of your mind while you work on other parts. Also, remember that each member of your team can work independently, provided that you synchronize suitably with each other, contribute equally, and understand each others' work at the end of the assignment. Thus, if you have read all of the coding component before you begin, you may realize that some parts are independent of each other (e.g., C1 and C3(a)).

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` with stub code provided in `synch.c`.

You can use the provided implementation of semaphores for inspiration and as a reminder of OS/161 coding style. You should not implement locks and condition variables in terms of semaphores.

Note that you may modify *any* of the source files of OS/161 to provide your implementation. Your criteria for change should be sanity and *invasiveness*. For example, redefining `spl0` as `disable_interrupts` because you think that your name is clearer is likely not to be viewed very kindly by your peer reviewers because such a change modifies a large number of files and offers marginal benefits.

The OS/161 tests menu has two tests (`sy2` and `sy3`) that are useful in checking your implementation (although you should not assume that these tests are exhaustive).

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. A Harvey Mudd E4 team has developed a complex automated animal feeding bowl that can feed a variety of different animals, including cats, mice, and dogs. So far, two prototypes for the automatic feeding bowl design have been built and need to be tested. The bowls do work, but the team has yet to complete a proper formal test due to some unforeseen “animal interactions”, such as the cats preferring to eat mice rather than food from the food bowls. After various attempts to arbitrate access to the bowls (including schemes where only one bowl ever seemed to be used, and another where the mice got fat and the cats starved), they have come to you asking for help in developing a synchronization algorithm.

As the E4 team discovered, we cannot allow a free-for-all for the food bowls. Cats will attack mice, and dogs will chase both cats and mice. You need to devise a synchronization scheme that will control access to the bowls in such a way that different species can share the bowls without ever actually seeing each other (i.e., only one species may be using the bowls at a time).

This assignment requires you to handle a simulation of two food dishes, six cats, and two mice, where each animal eats ten times. Thus, you can choose to ignore the issue of dogs entirely, but the provided skeleton code includes commented-out code for dogs and can easily be extended to other species and adjusted to vary the number of bowls, cats, and mice. Although it is *not* required, your solution should ideally be able to handle these variants of the problem.

All the animals are represented by independent threads that become hungry and want to go to the food-bowl area, eat at a particular food bowl, and then spend a random period satisfied by the food they have eaten before becoming hungry again. In the case of working with just cats and mice, your job is to develop a synchronization scheme where the cat and mouse threads are synchronized such that no mice could be eaten. For example, if a cat is eating at either food dish, any mouse attempting to eat from the other dish would be seen by the cat and could be eaten, so this situation must be avoided. Your synchronization scheme will require you to sometimes make cats or mice wait a moment for their food. Only one mouse or cat may eat at a given dish at any one time, but you should try to ensure that both bowls get used when doing so is practical (i.e., you can have a cat at each of the two bowls, or two mice eating at the bowls). You can assume that 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 (located in `kern/hw2/`) does little more than fork the required number of cat and mouse threads—you will need to do the rest.

Your solutions

- Must never allow two animals of different species to be at the bowls at the same time;
- Must not be prone to starvation, of either cats or mice (or dogs);
- Must protect the `kprintf` statements such that a status message from one thread cannot be interwoven into a status message from another thread;
- Must not change the format of the thread status messages from the format given in the skeleton code (although you may output additional messages to provide more information or to assist you in debugging);
- Must have each animal eat exactly ten times (i.e., `NUMLOOPS` times);
- Must not exit the main `catmouselock` or `catmousesem` until all the animal threads have terminated;¹
- Should provide good utilization of the bowls.

PATCH SUBMISSION

- C4. You should also create a kernel patch to add lock and condition variable support to OS/161. You should create your patch by running

```
cd ~/courses/cs134/assign2/handin
./makepatch > synch.patch
```

Once you have created the basic patch file, you should edit the file and to add any additional documentation you think appropriate (as plain text, *before* the patch proper). You *may* assume that the person reading your patch understands what locks and condition variables are.

Notes & Tips

KERNEL-MODE CODE

The code you write in part C3 is very unusual code. Normally, programs such as this would be written in user-mode code. We have instead included the code as a part of

1. OS/161 doesn't yet have a way to check whether a thread has terminated, so you'll need to keep enough housekeeping information around to be able to wait until they have all finished.

the kernel itself. We have done so for two reasons: First, your lock code is intended to be used by kernel functions that you will write in future assignments, not (directly) by user code, so that code does belong in the kernel—the cats-and-mice code is essentially just test code for that lock implementation, and it is easiest to write kernel code to test kernel code. Second, OS/161 is not yet able to run meaningful user-mode programs, so we have little other choice at present.

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.

SHARE TIPS

Although you may not share implementation code, you *should* feel free to share useful tips with other class members on topics such as using OS/161, designing a lock implementation, debugging with gdb, and so forth. The class Wiki is a good place to post such tips. Assignments are *not* competitive. If you all turn in great work, you will all get As.