Assignment 1: Synchronization

Assignment 1: Synchronization

Due: by 11:59pm on Wednesday, February 18, 2009.

Objectives

- Understand how OS/161 implements semaphores.
- Be comfortable developing/implementing synchronization primitives.
- Be able to select an appropriate synchronization primitive for a given problem.
- Be able to properly synchronize different types of problems.

Introduction

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. You will implement locks, condition variables, and read/write locks.

Write readable code!

In your programming assignments, you are expected to write well-documented, readable code. There are a variety of reasons to strive for clear and readable code. Since in the future you will be working in pairs, it will be important for you to be able to read your partner's code. Also, since you will be working on OS/161 for the entire semester, you may need to read and understand code that you wrote several months earlier. Finally, clear, well-commented code makes your TFs happy!

There is no single right way to organize and document your code. It is not our intent to dictate a particular coding style for this class. The best way to learn about writing readable code is to read other people's code. Read the OS/161 code, read your partner's code, read the source code of some freely available operating system. When you read someone else's code, note what you like and what you don't like. Pay close attention to the lines of comments which most clearly and efficiently explain what is going on. When you write code yourself, keep these observations in mind.

Here are some general tips for writing better code:

- Group related items together, whether they are variable declarations, lines of code, or functions.
- Use descriptive names for variables and procedures. Be consistent with this throughout the program.
- Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."

You and your partner will probably find it useful to agree on a coding style -- for instance, you might want to agree on how variables and functions will be named (my_function, myFunction, MyFunction, mYfUnCtIoN, ymayUnctionFay, etc.), since your code will have to interoperate.

Begin Your Assignment

The very first thing you need to do is apply the latest bug-fix patch to OS/161. You can get the patch here. Follow the directions in the Applying Patches guide. Make sure that you do not have any outstanding updates in your Subversion tree. Use `svn update` and `svn commit` to get your tree commited in the state from which you want to begin patching, and then commit the patched kernel (just as you should any other major change).

Now, "tag" your Subversion repository. The purpose of tagging your repository is to make sure that you have something against which to compare your final tree. Again, make sure do not have any outstanding updates in your tree (such as the

patch).

Now, tag your repository exactly as shown below.

```
% svn copy http://minitrue/repos/<username>/trunk http://minitrue/repos/<username>/tags/asst1-start
```

Configure OS/161 for ASST1

We have provided you with a framework to run your solutions for ASST1. This framework consists of driver code (found in `kern/synchprobs`) and menu items you can use to execute your solutions from the OS/161 kernel boot menu.

You have to reconfigure your kernel before you can use this framework. The procedure for configuring a kernel is the same as in ASST0, except you will use the ASST1 configuration file:

```
% cd kern/conf
% ./config ASST1
```

You should now see an ASST1 directory in the `compile` directory.

Building for ASST1

When you built OS/161 for ASST0, you ran `bmake` from `compile/ASST0`. In ASST1, you run `bmake` from (you guessed it) `compile/ASST1`.

```
% cd compile/ASST1
% bmake depend
% bmake
% bmake install
```

If you are told that the compile/ASST1 directory does not exist, make sure you ran config for ASST1.

Command Line Arguments to OS/161

Your solutions to ASST1 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu.

IMPORTANT: Please DO NOT change these menu option strings! We use them to automate testing.

"Physical" Memory

In order to execute the tests in this assignment, you will need more than the 1MB of memory configured into System/161 by default. We suggest that you allocate at least 2MB of RAM to System/161. This configuration option is passed to the `mainboard` device with the `ramsize` parameter in your `sys161.conf` file. Make sure the `mainboard` device line looks like the following:

```
31    mainboard  ramsize=2097152  cpus=2
```

Note: 2097152 bytes is 2MB.

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 of the constraints applied to them and that they do not deadlock.

Built-in thread tests

When you booted OS/161 in ASST0, 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 `thread_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. This means that 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 edit the line to look like:

```
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 to `autoseed`. This should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

(Note that for assignments 3 and 4, the disk device has a random delay that means that even if you use the same seed, you may not get reproducible results.)

Code Reading (10 points)

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. When you are writing solution code for the synchronization problems it will help if you also understand exactly what the OS/161 scheduler does when it dispatches among threads.

Please answer the following questions and submit them with your assignment. Place the answers in a file called `codereading.txt`

Thread Questions

1. What happens to a thread when it exits (i.e., calls `thread_exit()`)? What about when it sleeps?
2. What function(s) handle(s) a context switch?
3. What does it mean for a thread to be in each of the possible thread states?
4. 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?
5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

Scheduler Questions

1. What function(s) choose(s) the next thread to run?
2. How does it (do they) pick the next thread?
3. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

Synchronization Questions

1. Describe how `wchan_sleep()` and `wchan_wakeone()` are used to implement semaphores.
2. Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

Coding Exercises (70 points)

We know: you've been itching to get to the coding. Well, you've finally arrived! You can implement these primitives in terms of any other primitive(s), but you should put some thought into choosing the right implementation.

Synchronization Primitives (30 Points)

1. Implement locks

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`.

2. Implement condition variables

Implement condition variables with Mesa semantics for OS/161. The interface for the cv structure is also defined in `synch.h` and stub code is provided in `synch.c`.

3. Implement read/write locks

Implement read/write locks for OS/161. As described in class, a read/write lock is a lock that threads can acquire in one of two ways: read mode or write mode. Read mode does not conflict with read mode, but read mode conflicts with write mode and write mode conflicts with write mode. The result is that many threads can acquire the lock in read mode, **or** one thread can acquire the lock in write mode. Your solution must also ensure that no threads waits to acquire the lock indefinitely (starvation).

Unlike regular locks and condition variables, we have not provided you with an interface, stub code, or iron-clad semantics for read/write locks. You must develop your own. Your implementation must solve "many readers, one writer" problems. Make sure you choose something you will be comfortable using later.

Implement your interface in `synch.h` and your code in `synch.c`. Also, make a simple test program showing your implementation solves a "many readers, one writer" problem. Edit `/kern/startup/menu.c` and add a command `sy4` that runs your test program. Please use "sy4" verbatim - we'll use it to automate testing.

Solving Synchronization Problems (40 Points)

The following problems 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 in `/kern/synchprobs` that starts a predefined number of threads. You are responsible for what those threads do.

When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in. Type `?` at the menu to get a list of commands. Remember to specify a seed to use in the random number generator by editing your `sys161.conf` file. It is much easier to debug initial problems when the sequence of execution and context switches is reproducible.

There are two synchronization problems posed for you. You can solve these problems using semaphores, locks, and/or condition variables, but one way may be more straightforward than another. You should put some thought into choosing the correct synchronization primitive(s).

1. The Classic CS161 Whale Mating Problem

You have been hired by the New England Aquarium's research division to help find a way to increase the whale population. Because there are not enough of them, the whales are having synchronization problems in finding a mate. The trick is that in order to have children, three whales are needed; one male, one female, and one to play matchmaker - literally, to push the other two whales together (We're not making this up!).

Your job is to write the three procedures `male()`, `female()`, and `matchmaker()`. Each whale is represented by a separate thread. A male whale calls `male()`, which waits until there is a waiting female and matchmaker; similarly, a female whale must wait until a male whale and matchmaker are present. Once all three are present, all three return.

The test driver for this program is in the file `/kern/synchprobs/whalemating.c`. It forks thirty threads, and has ten of them invoke `male()`, ten of them invoke `female()`, and ten of them invoke `matchmaker()`. Stub routines, which do nothing but print diagnostic messages, are provided for these three functions. Your job will be to re-implement these functions so that they solve the synchronization problem described above. Each whale (thread) should print out a message when it begins, identifying itself, and then again when it has successfully mated (or assisted a couple in mating).

2. Bathroom Synchronization

NOTE: The Bathroom Synchronization problem does not ship with OS/161. This is a good opportunity for you to practice adding new files to your build, and new tests to your menu.

1. Get `bathroom.c` from here and add it to `/kern/synchprobs/`.
2. Put a function declaration for `bathroom()` in `test.h`.

3. Edit `menu.c` and add a command called `sp2` that runs `bathroom()`. Please use "sp2" verbatim - we'll use it to automate testing.
4. Add `bathroom.c` to your `conf.kern` and reconfigure your build.
5. Don't forget to add `bathroom.c` to your Subversion tree as well!

Harvard University has recently constructed a new dormitory. Because of budget cuts, this dormitory has only one small co-ed bathroom that can accommodate only three people. Furthermore, prudish members of the faculty insist that boys and girls should not be allowed in the bathroom at the same time. This leads to considerable synchronization problems and long lines in the morning.

Your job is to write the two procedures `boy()` and `girl()`. Each person who needs to go to the bathroom is represented by a seperate thread. A boy calls `boy()` to get in line and a girl calls `girl()`. To actually enter and use the bathroom, a person calls `shower()`. Your solution must ensure that no more than three people are in the bathroom at once, that every person in the bathroom is the same gender, and that no thread waits indefinitely to use the bathroom (starves). Your solution should also try to be "fair" - "boys before girls" may solve the synchronization problem, but it is not a fair solution. Make it clear in your comments what fairness policy you have implemented.

The test driver for the program is in the file `kern/synchprobs/bathroom.c`. It forks twenty threads, and has threads alternately invoke `boy()` and `girl()`. Stub routines, which do nothing but unsychronized `shower()`, are provided for these two functions. Your job will be to re-implement these functions so that they solve the synchronization problem described above. Each person (thread) should print out a message when it begins (identifying itself), when it enters the bathroom, and then again when it has successfully used the bathroom.

Online Submission

When you are finished with Assignment 1, create a directory called `asst1`. In this directory, you should place the following:

- `asst1-diff`: a diff file listing all the changes you made for this assignment.
- `username-asst1.tar.gz`: a tarball containing the latest release of your source tree (`username` should be replaced by your fas login name).
- your `sys161.conf` file.
- `codereading.txt`: your answers to the code reading questions.

Then add and commit the `asst1` directory to `minitrue`.

See the handout "Preparing your Assignments for Submission" for instructions on generating a diff listing and a tarball containing your latest source tree release, and submitting your assignment.

You do not need to print out anything for this assignment.