CS350 Operating Systems Winter 2009

Assignment One

This assignment has three parts. The first part of the assignment, which is described in Section 1, requires you to read some parts of the OS/161 code and to answer questions based on your reading. The second part of the assignment requires you to implement two synchronization primitives (locks and condition variables) in the OS/161 kernel. It is described in Section 2. The third and final part of the assignment, described in Section 3, asks you to implement a solution to a synchronization problem in the OS/161 kernel using the synchronization primitives.

1 Code Reading

The OS/161 implementation is a substantial body of code. You will need to understand this code to do the assignments. The best way to get started is to spend time browsing though the code to see what's there and how it fits together.

The rest of this section of the assignment consists of an overview of the structure of the OS/161 code. Mixed in with this overview are some numbered short-answer questions. These questions are intended to provide you with some specific targets for your code browsing. Look through the code to find answers.

You are expected to prepare a brief document, in PDF format, containing your answers to these questions. In your document, please number each of your answers and ensure that your answer numbers correspond to the question numbers. Your PDF document should be placed in a file called `codeanswers1.pdf`.

Top Level Directory

If you followed the standard instructions for setting up OS/161 in your course account, the top of the OS/161 source code tree is located in the directory `$HOME/cs350-os161/os161-1.11`. In this top-level source code directory you should find the following files:

`Makefile`: top-level makefile; builds the OS/161 distribution, including all the provided utilities, but does not build the operating system kernel.

`configure`: this is an autoconf-like script. It tries to customize the OS/161 build process for the machine on which the build is occurring.[1]

`defs.mk`: this file is generated when you run `./configure`. You needn't do anything to this file.

`defs.mk.sample`: this is a sample `defs.mk` file. Ideally, you won't be needing it either, but if configure fails, use the comments in this file to fix `defs.mk`.

`bin`: this directory is where the source code lives for all the utilities that are typically found in `/bin` on UNIX systems, e.g., `cat, cp, ls`. The things in bin are considered "fundamental" utilities that the system needs to run.

`include`: these are include files that you would typically find in `/usr/include` on UNIX systems. These are user level include files; not kernel include files.

`kern`: this is where the kernel source code lives.

`lib`: library code lives here. We have only two libraries: `libc`, the C standard library, and `hostcompat`, which is for recompiling OS/161 programs for the host UNIX system. There is also a `crt0` directory, which contains the startup code for user programs.

[1] In particular, if you work at home and then copy your code to the university environment for testing and submission, you will have to rerun `configure` after you move your code.

`man`: the OS/161 manual (man) pages appear here. The man pages document (or specify) every program, every function in the C library, and every OS/161 system call. The man pages are HTML and can be read with any browser.

`mk`: this directory contains pieces of makefile that are used for building the system. You don't need to worry about these.

`sbin`: this is the source code for the utilities typically found in `/sbin` on a typical UNIX installation. In our case, there are some utilities that let you halt the machine, power it off and reboot it, among other things.

`testbin`: these are pieces of test code.

You needn't understand the files in bin, sbin, and testbin now, but you certainly will later on. Similarly, you need not read and understand everything in lib and include, but you should know enough about what's there to be able to get around the source tree easily. The rest of this code walk-through is going to concern itself with the `kern` subtree.

The Kern Subdirectory

Once again, there is a Makefile. This Makefile installs header files but does not build anything. In addition, we have subdirectories for each component of the kernel as well as some utility directories.

`kern/arch`: This is where architecture-specific code goes. By architecture-specific, we mean the code that differs depending on the hardware platform on which you're running. At present, OS/161 supports only the MIPS architecture, so there is only one subdirectory: `kern/arch/mips`. It, in turn, contains three subdirectories, `conf`, `include`, and `mips`, which are described next.

`kern/arch/mips/conf`:

`conf.arch`: This tells the kernel config script where to find the machine-specific, low-level functions it needs (see `kern/arch/mips/mips`).

`Makefile.mips`: Kernel Makefile; this is copied when you "config a kernel".

`kern/arch/mips/include`: These files are include files for the machine-specific constants and functions.

Question 1. Which register number is used for the stack pointer (sp) in OS/161?

Question 2. What bus/busses does OS/161 support?

Question 3. What is the difference between splhigh and spl0?

Question 4. What are some of the details which would make a function "machine dependent"? Why might it be important to maintain this separation, instead of just putting all of the code in one function?

`kern/arch/mips/mips`: These are the source files containing the machine-dependent code that the kernel needs to run. Most of this code is quite low-level.

Question 5. What does splx return?

Question 6. What is the highest interrupt level?

`kern/asst1`: This is the directory that contains the framework code that you will need to complete assignment 1.

`kern/compile`: This is where you build kernels. In the compile directory, you will find one subdirectory for each kernel you want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, etc. In our world, each build directory will correspond to a programming assignment, e.g., ASST1 and ASST2. These directories are created when you configure a kernel.

`kern/conf`: `config` is the script that takes a config file, like ASST1, and creates the corresponding build directory. So, in order to build a kernel for assignment 1, you should:

```
% cd kern/conf
% ./config ASST1
% cd ../compile/ASST1
% make depend
% make
```

This will create the ASST1 build directory and then actually build a kernel in it. Note that you should specify the complete pathname `./config` when you configure OS/161. If you omit the `./`, you may end up running the configuration command for the system on which you are building OS/161, and that is almost certainly not what you want to do!

`kern/dev:` This is where all the low level device management code is stored. Unless you are really interested, you can safely ignore most of this directory.

`kern/include:` These are the include files that the kernel needs. The `kern` subdirectory contains include files that are visible not only to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up when installed.)

Question 7. How frequently are hardclock interrupts generated?

`kern/lib:` These are library routines used throughout the kernel, e.g., managing sleep queues, run queues, kernel malloc.

`kern/main:` This is where the kernel is initialized and where the kernel main function is implemented.

`kern/thread:` Threads are the fundamental abstraction on which the kernel is built.

Question 8. What are the possible states that a thread can be in? When do "zombie" threads finally get cleaned up?

Question 9. What function puts a thread to sleep? When might you want to use this function?

`kern/userprog:` This is where you will add code to create and manage user level processes. As it stands now, OS/161 runs only kernel threads; there is no support for user level code.

`kern/vm:` This directory is for the virtual memory implementation. Currently, it is mostly vacant.

`kern/fs:` The file system implementation has two subdirectories. `kern/fs/vfs` is the file-system independent layer (`vfs` stands for "Virtual File System"). It establishes a framework into which you can add new file systems easily. You will want to go look at `vfs.h` and `vnode.h` before looking at this directory. `kern/fs/sfs:` is the simple file system that OS/161 contains by default.

2 Implement Kernel Synchronization Mechanisms

Note that all code changes for Assignment 1 should be enclosed in `#if OPT_A1` statements. For this to work you must also be sure to add `#include "opt-A1.h"` to any file in which you will make edits for assignment 1. Put the `#include` at the top of the file, with the other `#include`s. Then use `#if OPT_A1` anywhere you make code changes in that file for Assignment 1. This will help you to track your code changes. It will also make it possible for us to identify your code changes, should we need to do so.

For example:

```
#include "opt-A1.h"

#if OPT_A1
   // Code executed when compiling ASST1 kernel.
   kprintf("OPT_A1 is turned on\n");
#else
   // Code executed when OPT_A1 is not defined
   kprintf("OPT_A1 is not turned on\n");
#endif /* OPT_A1 */
```

2.1 Implement Locks

Your first coding assignment is to 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`. You can use the implementation of semaphores as a model, but do not build your lock implementation on top of semaphores or you will be penalized.

The file `kern/test/synchtest.c` implements one simple test case for locks (called `locktest`) that can be called from `kern/main/menu.c`. This can be run by typing `sy2` from the OS/161 prompt, e.g.,:

```
OS/161 kernel [? for menu]: sy2
```

or from the command line as:

```
% sys161 kernel sy2
```

The following command (assuming you use the csh) can be useful for both seeing the output of your command on the screen and capturing the output into the file named `OUTPUT`. This can be especially useful when debugging is turned on and while looking for bugs because you can run the command, capture the output and then search and navigate through the output using an editor.

```
% sys161 kernel sy2 |& tee OUTPUT
```

You should feel free to add your own tests to `kern/test/synchtest.c` and to add the ability to execute those tests from the OS/161 menu by modifying `kern/main/menu.c`. *However, you should not modify any of the existing tests, and you should not make any changes to the way that the existing tests are invoked, e.g., do not change "sy2" to "sy2a".* If you do create new tests, note that the main OS/161 kernel thread will not block waiting for forked kernel threads to complete. Thus, if you create your own test and it involves forking additional kernel threads, you may get a prompt from the main kernel thread before the threads you've forked have completed. To avoid this, you can use the same technique that the examples in `kern/test/synchtest.c` use to ensure that the main kernel thread waits/blocks until the forked threads complete before it prints a new prompt.

Note that the existing OS/161 code operating already makes calls to `lock_create`, `lock_acquire`, `lock_release`, `lock_destroy`. Currently these are just empty functions that do nothing. You will need properly functioning locks for this and future assignments. Make sure that you get locks working before moving on to the other parts of the assignment.

2.2 Implement Condition Variables

Implement condition variables for OS/161. The interface for the `cv` structure is defined in `kern/include/synch.h` and stub code is provided in `kern/thread/synch.c`.

The file `kern/test/synchtest.c` implements one simple test case for condition variables (called `cvtest`) that can be called from `kern/main/menu.c`. This can be done by typing `sy3` from the OS/161 prompt, e.g.,:

```
OS/161 kernel [? for menu]: sy3
```

or from the command line as:

```
% sys161 kernel sy3
```

Again, you are free to devise your own `cv` tests, but *do not change the built-in tests in any way.*

3 Solve a Synchronization Problem

In this section of the assignment, you are asked to design and implement a solution to a synchronization problem using the synchronization primitives available in the OS/161 kernel: semaphores, locks, and condition variables. You are free to use whichever synchronization primitives you choose. The synchronization problem is called the "cats and mice" problem.

The Cats and Mice Problem

A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats' food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice (or else lose face with all of their cat friends). There are `NFOODBOWLS` catfood dishes, `NCATS` cats, and `NMICE` mice.

Your job is to synchronize the cats and mice so that the following requirements are satisfied:

No mouse should ever get eaten.

You should assume that if a cat is eating at a food dish, any mouse attempting to eat from that dish or any other food dish will be seen and eaten. When cats aren't eating, they will not see mice eating.

Only one mouse or one cat may eat from a given dish at any one time.

Neither cats nor mice should starve.

A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable.

There are many ways to synchronize the cats and mice that will satisfy the requirements above. From among the possible solutions that satisfy the requirements, we *prefer* solutions that (a) avoid unnecessary synchronization delays, and (b) treat cats and mice evenly. Avoiding unnecessary synchronization delays means not blocking cats and mice unnecessarily. For example, a solution that uses a single lock to ensure that only one creature eats at a time (regardless of the number of bowls) satisfies the three problem requirements, but it may delay creatures unnecessarily. (For example, if there are 5 cats and 5 bowls and no mice, then 5 cats should should be able to eat simultaneously.) Treating cats and mice fairly means avoiding systematic bias against cats or mice. For example, a solution that makes all mice wait until all cats have finished eating as many times as they want satisfies the three problem requirements, but it is biased in favour of the cats.

Note that there may be a tradeoff between performance (avoiding delays) and fairness. For example, consider a situation in which there are two bowls, at cat is eating at one of the bowls, and a mouse is waiting to eat so that it does not get seen by the cat that is eating. Now, suppose that a second cat shows up, wanting to eat. Should you allow it to eat immediately from the unused bowl? Or should you make it wait until the mouse has had a chance to eat? After all, the mouse was there first. You should be aware of such tradeoffs in your design, so that you can discuss them in your design document (described below).

In the OS/161 kernel, each cat and each mouse is implemented by a thread. The cat threads run the function `cat_simulation`, and the mouse threads run the function `mouse_simulation`. Both of these functions are found in `kern/asst1/catmouse.c`. There is driver code in `catmouse.c` which forks the appropriate numbers of cat and mouse threads, and there are simple implementations of `cat_simulation` and `mouse_simulation` already in place. However, the existing cat and mouse simulations *do not provide any synchronization* - the cats and mice simply eat without regard to which other creatures are eating. Your job is to add appropriate synchronization to `cat_simulation` and `mouse_simulation` so that the synchronization rules described above are enforced.

When you look at the existing implementations of `cat_simulation` and `mouse_simulation`, you will see that eating is simulated by calls to functions `cat_eat` and `mouse_eat`. These functions are implemented in the file `kern/asst1/bowls.c`. You are free to inspect the code in `bowls.c`, **but you may not change it in any way**. The code in `bowls.c` keeps track of which creatures are eating at which bowls, and checks to make sure that the synchronization requirements are not violated. If `bowls.c` detects a violation of the synchronization rules, it will shut down OS/161 and display a message to indicate the problem. You can try this out with the given implementations of `cat_simulation` and `mouse_simulation` - since they don't make any attempt to enforce the rules, it is very likely that the rules will be violated and OS/161 will shut down.

You can launch the cat and mouse simulation from the OS/161 kernel menu or from the command line. For example, from the kernel menu the command

```
OS/161 kernel [? for menu]: 1a 6 3 5 10
```

will launch a cats and mice simulation with 6 food bowls, 3 cats, and 5 mice. The last parameter, 10, indicates that each cat and mouse will eat 10 times before terminating. You can run different tests by varying these parameters to change the numbers of bowls, cats, and mice or the number of iterations.

You can also start OS/161 and launch a simulation right from the UNIX command line:

```
% sys161 kernel ‘‘1a 6 3 5 10; q’’
```

This will start OS/161, run the cat/mouse simulation with the same parameters as described above and, finally, shut down OS/161. (That is the purpose of the "q" command following the "1a" command.)

Note that when you modify `cat_simulation` `mouse_simulation`, you should preserve the meaning of the iteration parameter, which indicates how many times each cat and mouse should eat before terminating.

Finally, note that we expect your solution to work properly regardless of the numbers of cats, mice, bowls, and interactions that are specified when the test is launched. We will test your implementation by running simulations with a variety of different input parameter values.

Cats and Mice Design Document

In addition to implementing your solution, problem, **you are expected to prepare and submit a short *design document* that describes how you solved the cats and mice problem**. Your design document should indicate which synchronization primitives you used to solve the problem and how many instances of each primitive you used. In addition, for each instance of a synchronization primitive, you should clearly describe its use and purpose, i.e., you should indicate which synchronization requirement(s) that primitive is intended to enforce and how it does so. Finally, your design document should explain how your design attempts to avoid unnecessary synchronization delays, and it should identify any unfairness (bias against cats or mice) that you can identify in your design.

We (course personnel) should be able to understand how you solved the synchronization problem by reading your design document, and your document should be self-contained That is, it should not assume that the reader has access to your source code.

Your design document must be prepared in PDF format and saved in a file called `design1.pdf`. The document must be *at most* 2 pages long - 1 page is fine if you can clearly explain your design in one page. Also, please ensure that your PDF documents are laid out in portrait mode, not landscape mode.

4 What to Submit

You should submit your kernel source code, your code question answers, and your design document using the `cs350_submit` command, as you did for Assignment 0. It is important that you use the `cs350_submit` command - do not use the regular `submit` command directly.

Assuming that you followed the standard OS/161 setup instructions, your OS/161 source code will be located in `$HOME/cs350-os161/os161-1.11`. To submit your work, you should

1. place `codeanswers1.pdf` in the directory `$HOME/cs350-os161/`
2. place `design1.pdf` in the directory `$HOME/cs350-os161/`
3. run `cs350_submit 1` in the directory `$HOME/cs350-os161/`. The parameter "1" to `cs350_submit` indicates that you are submitting Assignment 1.

This will package up your OS/161 kernel code and submit it, along with the two PDF files, to the course account.

Important: The `cs350_submit` script packages and submits everything under the `os161-1.11/kern` directory, except for the subtree `os161-1.11/kern/compile`. This means that any changes that you make to the OS/161 source should be confined to the kernel code under `os161-1.11/kern`. Any changes that you make elsewhere will not be submitted. This should not be a problem since everything that you need to do for this assignment is done in the kernel source.