CS350 **Operating Systems** **Fall 2007**

Assignment One (OS/161)

Version 1.2 September 21 (previous was Version 1.1 September 19)

1 Key Requirements

This section outlines what is absolutely essential for you implement correctly for this and future assignments because they rely on correct working versions. These are bare minimum requirements.

- Locks: Be sure to meet the requirements in Section 3.1. If you aren't able to get locks implemented and debugged properly this and future assignments will not work. The base OS/161 kernel relies on correctly working locks.
- Semaphores: if you modify the semaphore implementation in any way, be sure that it still functions correctly. The base OS/161 kernel relies on correctly working semaphores.

2 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 `codeanswers.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`. (If you installed on your own machine, this pathname may be different.) 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 occuring. **NOTE: that if you first work at home you will have to rerun configure when you move your code to the university environment**

`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.

`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. Eventually, you will want to modify these and/or write your own utilities and these are good models.

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?

Question 8. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?

Question 9. How large are OS/161 pids? How many processes do you think OS/161 could support as you have it now (ASST1)? A sentence or two of justification is fine.

Question 10. A vnode is a kernel abstraction that represents a file. What operations can you do on a vnode? If two different processes open the same file, do we need to create two vnodes?

Question 11. What is the system call number for a reboot? Is this value available to userspace programs? Why or why not.

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

Question 12. What is the purpose of functions like copyin and copyout in `copyinout.c`? What do they protect against? Where might you want to use these functions?

Question 13. Operating systems usually have two separate implementations of `malloc`. In OS/161 one is in kern/lib/kheap.c, the other one is missing but it belongs in lib/libc/malloc.c. Explain why there needs to be two separate implementations and describe the implications of the missing implementation.

`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 14 Is it OK to initialize the thread system before the scheduler? Why (not)?

Question 15. What are the possible states that a thread can be in? When do "zombie" threads finally get cleaned up?

Question 16. 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.

Question 17. What does a device pathname in OS/161 look like?

Question 18. What does a raw device name in OS/161 look like?

Question 19. What lock protects the vnode reference count?

3 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"` in your file. This will help you to track your code changes and for the markers to easily find where your code has been changed, but you do not need to ensure that the code compiles or runs when `OPT_A1` is not defined.

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 */
```

Also note that as provided the main OS/161 kernel thread will not block waiting for forked kernel threads to complete. As a result if you create your own tests you may get a prompt from the main kernel thread before the test you've implemented completes. You may wish to 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.

3.1 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`. 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.

Your implementation of locks should ensure fairness. That is, threads should be granted locks in the order that they are blocked when unsuccessfully trying to acquire a lock.

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 test from the OS/161 menu by modifying `kern/main/menu.c`. Be sure to test and document your changes in your design document.

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. So please take care to ensure that you get locks implemented and working.

3.2 Implement Condition Variables

Implement condition variables for OS/161. The interface for the `cv` structure is also 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
```

Please do not change the existing built-in synchronization tests in any way.

4 Solve Synchronization Problems

In this section of the assignment, you are asked to implement solutions to two synchronization problems using the synchronization primitives available in the OS/161 kernel.

You must solve the Cats and Mice problem using only locks and condition variables and the Traffic Control problem using only semaphores.

4.1 Cats and Mice

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 to all their cat friends). There are `NFOODBOWLS` (two) catfood dishes, `NCATS` (six) cats, and `NMICE` (two) mice. Be sure that your solution does not depend on the current values of `NFOODBOWLS`, `NCATS` or `NMICE`.

Your job is to synchronize the cats and mice. No mouse should ever get eaten. 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. When cats aren't eating, they will not see mice eating. Also, you may not starve either the cats or the mice. Only one mouse or cat may eat at a given dish at any one time.

The driver code for the Cats and Mice problem is found in the driver file `kern/asst1/catlock.c`. Each cat and mouse is represented by a kernel thread. Right now the driver code only forks the required number of cat and mouse threads. Your job is to implement a solution to the Cats and Mice problem.

Each cat and mouse thread should identify itself and which bowl it is eating at when it begins and ends eating as well as printing how many cats and how many mice are eating at the bowls. (Note, that when one or more cats are eating there should be no mice eating and when one or more mice are eating there should be no cats eating.) Simulate a cat or mouse eating by calling `clocksleep()`. In order to ensure that your solution works properly, each cat and mouse thread should execute some number of loops before exiting.

```
#define NLOOPS   (3)

for (i=0; i<NLOOPS; i++)
{
   // try to eat and other stuff that needs to be done
}
```

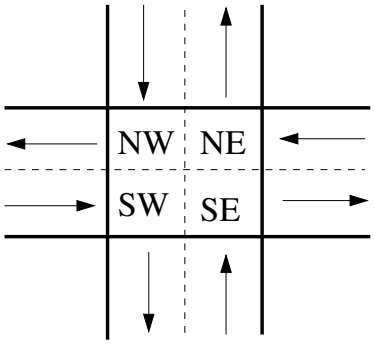

Figure 1: Modeling the Intersection Using Four Quadrants

That way at the end each cat and each mouse will have eaten `NLOOPS` times.

You will also need to modify the `catmouselock` function so that after forking the cats and mice threads, it waits for all of them to complete. Hint: use a semaphore for this as is done in some of the tests like `locktest` in `kern/test/synchtest.c`. This is not considered part of synchronizing the cats and mice so you still need to use only locks and condition variables for that part of the solution.

4.2 Traffic Control

Your job is to design and implement a mechanism for controlling traffic through an intersection. Your solution may use only semaphores for thread synchronization.

For the purposes of this problem we will model the intersection as shown in Figure 1, dividing it into quadrants and identifying each quadrant with which lane enters the intersection through that portion. Turns are represented by a progression through one, two, or three quadrants (for simplicity assume that U-turns do not occur in the intersection). So if a car approaches from the North, depending on where it is going, it proceeds through the intersection as follows:

Right: NW

Straight: NW-SW

Left: NW-SW-SE

The driver for the traffic control problem is in `kern/asst1/stoplight.c` It consists of `createcars()` which creates 20 cars and passes them to `approachintersection()`, which assigns each a random direction. We forgot to assign them a random turn direction; please do this in `approachintersection()` as well. The file `kern/asst1/stoplight.c` also includes routines `gostraight()`, `turnright()` and `turnleft()` that may or may not be helpful in implementing your solution. Use them or discard them as you like.

Your solution should satisfy the following requirements:

- No two cars can be in the same portion of the intersection at the same time.
- Cars going the same way do not pass each other. If two cars both approach from the same direction and head in the same direction, the first car to reach the intersection should be the first to reach the destination.
- Your solution should maximize traffic flow without allowing traffic from any direction to starve traffic from any other direction. In other words, you should permit as many cars as possible into the intersection, provided all other requirements for the problem can be met.
- Each car should print a message as it approaches, enters, and leaves the intersection indicating the car number, approach direction and destination direction.

- All cars must complete their journey

You will also need to modify the `createcars` function so that after forking all of the car threads, it waits for all of them to complete. Hint: use a semaphore for this as is done in some of the tests like `locktest` in `kern/test/synchtest.c`.

In addition to coding implementations of your solutions to these two problems, you should also prepare a short design document describing your solutions. For each problem, your design document should describe what synchronization primitives you used, and how you used those primitives to solve the problem. For the Traffic Control problem, your document should explicitly describe how each of the solution requirements is achieved. Please prepare your design document as a PDF file called `design.pdf`. Your design document should be *at most* three pages long.

5 What to Submit

You should submit the following items using the `submit` command:

- A copy of your OS/161 source code, including your implementations of the synchronization primitives and your implemented synchronization problem solutions. Please submit the top of the OS/161 source tree (e.g., `$HOME/cs350-os161/os161-1.11`) and everything below it.
- Your file `codeanswers.pdf`, described in Section 2.
- Your file `design.pdf`, described in Section 4.