CS350 Operating Systems Spring 2012

Assignment One

This assignment has three parts. The first part, which is described in Section 1, requires you to read the OS/161 code and to answer a few questions based on your reading. The second part of the assignment, described in Section 2, requires you to implement two synchronization primitives (locks and condition variables) in the OS/161 kernel. The third 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 some time browsing through the code to see what's there and how it fits together.

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

You are expected to prepare a brief document, in PDF format, containing your answers to these questions. Your PDF document should be placed in a file called `codeanswers1.pdf`, which should be at most 1 page long, using a 10 point (or larger) font and 3/4-inch (or larger) top, bottom, and side margins. In your document, please number each of your answers and ensure that your answer numbers correspond to the question numbers.

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 and directories:

`Makefile`: This top-level Makefile is used to build the OS/161 distribution, including all of the provided utilities and test programs; however, it does not build the operating system kernel.

`configure`: This is an autoconf-like script which you used when you were installing OS/161. It tries to customize the OS/161 build process for the host 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. You should not need to do anything to this file either.

`bin`: This directory holds the source code for a variety of utility programs. These utilities are similar to those that are typically found in `/bin` on UNIX systems, e.g., `cat, cp, ls`. These are fundamental, basic utilities that are important to the operation of the system.

`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`: User-level library code lives here. We have only two libraries: `libc`, the standard C 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 standard C library, and every OS/161 system call. The man pages are in 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 that are found in `/sbin` on a typical UNIX installation. There are some utilities that let you halt the machine, power it off and reboot it, among other things.

`testbin`: These are application programs that are used to test the OS/161 kernel. You will not need these for Assignment 1, but you will use them for the subsequent assignments.

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, which is where the kernel code lives.

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 kernel code goes. By architecture-specific, we mean the code that differs depending on the hardware platform on which the kernel will run. 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`: This directory contains information that is used when the kernel is being configured.

`kern/arch/mips/include`: These are kernel include files that define machine-specific constants and functions.

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

Question 2. What is the name of the kernel function that disables interrupts?

`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 3. What does splx return?

Question 4. What is the highest interrupt level?

`kern/asst1`: This is the directory that contains the framework code that you will use for the third part of 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 the kernel for a particular assignment.

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

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

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 for now.

`kern/include:` These are the include files that the kernel needs. The subdirectory `kern/include/kern` contains include files that are used by both the kernel and by user-level application programs. They typically define constants that are used in the system call interface between the applications and the kernel.

Question 5. How many times per second is the kernel's `hardclock()` function invoked? Which hardware device causes these invocations to occur?

`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. The kernel's command menu is also implemented by code in this directory.

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

Question 6. What are the possible states that a thread can be in?

Question 7. When do "zombie" threads finally get cleaned up?

Question 8. Which function is used to put a thread to sleep?

Question 9. What is the purpose of the kernel's `curthread` global variable?

`kern/userprog:` This is where you will add code to create and manage user level processes. You will not need this directory until Assignment 2.

`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. `kern/fs/sfs:` is the very simple OS/161 file system. Again, you will not need this directory for Assignment 1.

2 Implement Kernel Synchronization Mechanisms

The OS/161 kernel includes three types of synchronization primitives: semaphores, locks, and condition variables. Semaphores are already implemented. Locks and condition variables are not – it is your task to implement them.

All of the code changes that you make 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
   // Your modified code, which will be used
   // when you compile the ASST1 kernel.
```

```
   kprintf("brand new messge used for Asst 1");
#else
   // The original code, NOT included
   // when you compile the ASST1 kernel.
   kprintf("some kind of message");
#endif /* OPT_A1 */
```

2.1 Implement Locks

Your first task 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. In other words, your lock implementation should not use `sem_create(), P(), V()` or any of the other functions from the semaphore interface.

The file `kern/test/synchtest.c` implements one simple test case for locks, called `locktest`. It 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".[2]

Locks are used throughout the OS/161 kernel. You will need properly functioning locks for this and future assignments to ensure that the kernel's threads are properly synchronized. Because of this, implementing locks correctly - though not difficult - is the most important part of this assignment. **Make sure that you get locks working before moving on to the other parts of the assignment.**

2.2 Implement Condition Variables

The second task is to 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`. **Note:** Each condition variable is intended to work with a lock: condition variables are only used *from within the critical section that is protected by the lock*

The file `kern/test/synchtest.c` implements one simple test case for condition variables, called `cvtest`. It can be run 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
```

[2]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.

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

Testing synchronization primitives like locks and condition variables is difficult. The `sy3` condition variable test is subject to false positives. In other words, an incorrect condition variable implementation may pass the `sy3` test. However, if your implementation *fails* the `sy3` test, there is definitely a problem.

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 `NumBowls` catfood dishes, `NumCats` cats, and `NumMice` 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. In other words, this requirement states that if a cat is eating from any bowl, then no mouse should be eating from any bowl.

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. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve.

Your solution must not rely on knowledge of the numbers of cats and mice in the system. In particular, you should not make direct or indirect use of the variables `NumCats` and `NumMice` in your solution. (Those parameters should be used only by the `catmouse()` function to create the correct numbers of cats and mice to run a particular test.) It *is* OK to make use of the `NumBowls` parameter in your solution.

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 are (a) efficient and (b) fair.

An efficient solution avoids allowing food bowls to go unused when there are creatures waiting to eat. Sometimes it is necessary to make creatures wait, even when there are unused food bowls, in order to satisfy the synchronization requirements, so an efficient solution should avoid unnecessary delays. 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 problem requirements, but it is not efficient because it may delay creatures unnecessarily. To see this, consider a scenario in which there are 5 cats, 5 bowls and no mice. In that scenario, the 5 cats should should be able to eat simultaneously.

Fairness means avoiding favouring some creatures over others, and in particular avoiding 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.

There is a tradeoff between efficiency and fairness. For example, consider a situation in which there are two bowls, a 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. If your solution allows the cat to eat, it has sacrificed some fairness for efficiency by allowing the cat to jump in front of the waiting mouse. On the other hand, if your solution makes the second cat wait then it sacrifices some efficiency since a bowl is left unused though there is an eligible cat waiting.

Both approaches are reasonable (provided that there is no possibility that the waiting mouse will starve), but one leans in favour of performance, the other in favour of fairness. You should be aware of such tradeoffs in your design, as we will ask you to explain them.

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. If you inspect these simulation functions, you will see that each cat and mouse simply alternates between sleeping and eating. The existing simulation functions *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. Don't synchronize sleeping - any number of cats and mice, in any combination, should be able to sleep at the same time.

When you look at the existing implementations of `cat_simulation` and `mouse_simulation`, you will see that eating and sleeping are simulated by calls to functions `cat_eat`, `mouse_eat`, `cat_sleep` and `mouse_sleep`. 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 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 the simulation functions 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 Questions

Please provide short answers to the following questions about your solution to the cat and mouse synchronization problem

Question 1.

List all of the synchronization primitives and shared variables that you have used to synchronize the cats and mice and identify the purpose of each one.

Question 2.

Briefly explain how the listed variables and synchronization primitives are used to synchronize the cats and mice.

Question 3.
Briefly explain why it is not possible for two creatures to eat from the same bowl at the same time under your synchronization technique.

Question 4.
Briefly explain why it is not possible for mice to be eaten by cats under your synchronization technique.

Question 5.
Briefly explain why it is not possible for cats or mice to starve under your synchronization technique.

Question 6.
Briefly discuss the fairness and efficiency of your synchronization technique, and describe any unfairness (bias against cats or mice) that you can identify in your design. Did you have to balance fairness and efficiency in some aspect of your design? If so, explain how you chose to do so.

Please include the answers to these questions in a PDF document called `design1.pdf`, which should be at most **2 pages** long, using a **10 point** (or larger) font and 3/4-inch (or larger) top, bottom, and side margins. One page should be sufficient if your answers are brief and clearly written – do not feel obligated to submit a two-page document. Also, make sure that your PDF documents are laid out in portrait mode, not landscape mode.

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

4 What to Submit

You should submit your kernel source code, your code question answers, and your design question answers 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 code.