COMP 3500 Introduction to Operating Systems
Project 3 – Synchronization

Points Possible: 100
Submission via Canvas.

No collaboration between groups. Students in one group should NOT share any project code with any other group. Collaborations among groups in any form will be treated as a serious violation of the University's academic integrity code.

Requirements

1. Each student should independently accomplish the written exercises and submit `codereading.txt` and `exercises.txt`.
2. The coding exercise consists of 3 subtasks.
3. This lab assignment has to be done individually by each group. You are allowed to discuss with other students to solve the coding problems.
4. Each group is required to demonstrate the code to the TA. Your source code will be graded by the TA. Importantly, the TA will ask each student an array of questions regarding the source code.

Objectives:

- To implement the lock mechanism
- To implement condition variables
- To solve a synchronization problem using different mechanisms
- To improve your source code reading skills
- To strengthen your debugging skill

1. Introduction

This is a project that requires you to work in teams of two. It may become difficult if you are working alone, because the assignment may be too complex to be done single-handedly. You will gain valuable real-world experience from learning to effectively work in a team.

1.1 Project Goals

This assignment provides you with the opportunity to explore the synchronization topics in more details than can be covered in our class. Specifically, you will design and

implement synchronization primitives for the OS/161. You also will apply your newly implemented mechanisms to solve a synchronization problem. On completion of this programming project, you are expected to demonstrate an ability to develop concurrent programs.

You have to be familiar with the OS/161 thread code in order to accomplish this project. Note that the thread system is comprised of interrupts, control functions, and semaphores. Please keep in mind that in this project you must implement locks and condition variables.

1.2 Make Your Code Readable

It is very important for you to write well-documented and readable code in this programming assignment. The reason for making your code clear and readable is three-fold. First, you will be working in a group of two, you should strive to enable your group members to read and understand your code. Second, there is a likelihood that you will read and understand code written by yourselves in the future. Last, but not least, it will be a whole lot easier for the TA to grade your programming projects if you provide well-commented code.

Since there are a variety of ways to organize and document your code, you are allowed to make use of any particular coding style for this programming assignment. It is believed that reading other people's code is a way of learning how to writing readable code. In particular, reading the OS/161 code and the source code of some freely available operating system provides a capability for you to learn good coding styles. Importantly, when you write code, please pay attention to comments which are used to explain what is going on in your system.

Some general tips for writing good code are summarized as below:

- A little time spent thinking up better names for variables can make debugging a lot easier. Use descriptive names for variables and procedures.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Watch out for uninitialized variables.
- Split large functions that span multiple pages. Break large functions down! Keep functions simple.
- Always prefer legibility over elegance or conciseness. Note that brevity is often the enemy of legibility.
- Code that is sparsely commented is hard to maintain. 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."

- Backing up your code as you work is difficult to remember to do sometimes. As soon as your code works, back it up. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

2. Getting Started

2.1 Setup $PATH and $CVSROOT

Please ensure that your account is properly set up for use with OS161. You need to have the correct directories in your `$PATH`, and your `$CVSROOT` has to be set correctly. If you are using bash as your shell you should add the following line near the end of the `~/.bashrc` file.

```
export PATH=~/cs161/bin:$PATH
```

Set your CVSROOT environment variable as follows. This will keep you from having to specify the -d argument every time you use CVS.

```
%export CVSROOT=~/cs161/cvsroot
```

2.2 Create a New CVS Repository

This project does not rely on Project 2 and; therefore, you will start with a new CVS repository. In case you are willing to keep your old repository, you can move it to a new location (the "mv" commands in steps 2 and 3 below allow you to do this). You should be able to keep the `root` directory, as the necessary files are automatically overwritten.

Important! This step is very important; please follow the instructions carefully.

```
% cd ~/cs161
% mkdir archive
% mv cvsroot archive/cvsroot-asst0
% mv src archive/src-asst0
% mkdir cvsroot
% cvs init
% tar xvfz os161-1.10.gz
% cd os161-1.10
% cvs import -m "Initial import" src os161 os161-1_10
% cd ..
% rm -rf os161-1.10
% cvs co src
```

Prior to working on this project, please tag your CVS repository (See the following command below). The purpose of tagging your repository is to ensure that you have something against which to compare your final tree.

```
% cvs tag asst1-start src
```

2.3 Project Configuration

You are provided with a framework to run your solutions for this project. This framework consists of driver code (found in `kern/asst1`) 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. Please note that if you intend to work in a directory that's not `$HOME/cs161` (which you will be doing when you test your later submissions), you will have to use the `–ostree` option to specify a directory in which you are working. Please note that the command `./configure –help` explains the other options.

```
%cd ~/cs161/src
%./configure
%cd ~/cs161/src/kern/conf
%./config ASST1
```

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

2.4 Building for ASST1

Recall that when you built OS/161 for project 2, you ran `make` from this directory `kern/compile/ASST0`. In this project, you need to run make from another directory, namely, `kern/compile/ASST1`. You can following the following three commands to build OS/161 for this project.

```
% cd ../compile/ASST1
% make depend
% make
% make install
```

Important! In case that your `compile/ASST1` directory does not exist, please ran the aforementioned `config` step (see Section 2.3) for `ASST1`.

Please place `sys161.conf` in your OS/161 root directory (`~/cs161/root`).

2.5 Command Line Arguments to OS/161

Your solutions to this project (a.k.a., 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 the OS/161 root menu option strings!

2.6 Physical Memory

To execute the tests in this programming project, you need more than 512 KB of memory configured into System/161 by default. You are advised to allocate at least 2MB of RAM to System/161. This configuration option is passed to the busctl device with the ramsize parameter in your `sys161.conf` file. The busctl device line looks like the following:

```
31 busctl ramsize=2097152
```

It is worth noting that 2097152 bytes is 2MB.

3. Concurrent Programming with OS/161

If your code is properly synchronized, it is guaranteed that the timing of context switches and the order in which threads run will not change the behavior of your solutions. Although your threads may print messages in different orders, you can easily verify that they follow all of constraints applied to them; you can also verify that there is no deadlock.

3.1 Built-in Thread Tests

Important! When you booted OS/161 in project 2 (a.k.a., ASST0), you may have seen the options to run the thread tests. The thread test code makes use of the semaphore synchronization primitive. You should be able to 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 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 any starvation (e.g., the threads should all start together, run for a while,

and then end together).

3.2 Debugging Concurrent Programs

`thread_yield()` is automatically called for you at intervals that vary randomly. This randomness is fairly close to reality, but 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. The implication is 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
```

Important! It is strongly recommended 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 allows you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

4. Code-Reading Exercises

Please answer the following questions related to OS/161 threads. Please place answers to the following questions in a file called `codereading.txt`.

To implement synchronization primitives, you 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.

4.1 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) How many thread states are there? What are they?
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?

4.2 Scheduler Questions

6) What function is responsible for choosing the next thread to run?
7) How does that function pick the next thread?
8) What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

4.3 Synchronization Questions

9) Describe how `thread_sleep()` and `thread_wakeup()` are used to implement semaphores. What is the purpose of the argument passed to `thread_sleep()`?
10) Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

5. Programming Exercises

Now we are in a position to focus on kernel code writing.

5.1 Synchronization Primitives: Implementing Locks

In the first programming exercise of this project, you will implement locks for OS/161. The interface for the lock structure is defined in `kern/include/synch.h`. Stub code is provided in `kern/thread/synch.c`.

Important! You may use the implementation of semaphores as a model, but do NOT build your lock implementation on top of semaphores.

5.2 Synchronization Primitives: Implementing Condition Variables (cv)

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

5.3 Solving a Synchronization Problem

The following problem offers you an opportunity to write some fairly straightforward concurrent programs and to 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.

Again, remember to specify a seed to use in the random number generator by editing your `sys161.conf` file. It is a lot easier to debug initial problems when the sequence of execution and context switches is reproducible.

When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in.

We have posed **one** synchronization problem for you. You must solve this problem TWICE: once using only semaphores (`catsem.c`), and once using locks and possibly condition variables (`catlock.c`).

5.4 The Synchronization Problem: Cats and Mice

A professor has several cats and (unfortunately) also some mice that like to hang out in her house. The cats and mice operate need to work out a deal where the mice are allowed to steal bits of cat food, so long as the cats never actually see the mice actually doing so. If a cat sees a mouse eating from a cat dish, then the cat is obligated to eat the mouse (or else lose face before their cat buddies).

Your task is to synchronize the eating habits of cats and mice, such that:

- no mouse ever gets eaten, and
- neither the cats or the mice starve.

To solve this synchronization problem, we make the following assumptions:

- there are two cat food dishes, 6 cats, and two house mice to be coordinated,
- only one mouse or cat may eat at a given dish at any one time,
- if a cat is eating at either dish, a mouse attempting to eat from the other dish will be seen and therefore eaten,
- when cats aren't eating, they will not see mice eating.

The driver code for the Cats and Mice Synchronization problem is found in two separate files: `kern/asst1/catsem.c` and `kern/asst1/catlock.c`. Right now the driver code in both only forks the required number of cat and mouse threads.

Your job is to implement two seperate solutions to this synchronization problem:

- a semaphore-based solution in `catsem.c` and,
- a locks based solution possibly with condition-variables in `catlock.c`

Each cat and mouse thread should identify itself and which dish it is eating from at the point when it begins and when it finishes eating. Simulate a cat or mouse eating by calling `clocksleep()`.

Important! Once you have completed your solution, answer these questions in your exercises.txt file.

1) Explain how each of your solutions avoid starvation.
2) Comment on the relative ease of implementation under the two schemes. Can you derive any principles about the use of these different synchronization primitives?

6. Deliverables

Make sure the final versions of all your changes are added and committed to the CVS repository before proceeding. If you have already used `asst1-end` to tag your repository, you will need to use a unique tag in this part. Just replace all instances of `asst1-end` with your chosen unique tag below.

```
%cd ~/cs161
%cvs commit
%cvs tag asst1-end src
%cvs -q rdiff -rasst1-start -rasst1-end -c src > asst1.diff
```

It is a very good idea to inspect your `asst1.diff` file carefully to make sure that all your changes are present before submitting. It is your responsibility to know how to correctly use CVS.

You can check out a clean copy of the repository and test your patch. You can delete the temporary copy when you are done. Assuming your diff file is named

```
~/cs161/asst1.diff:
%cd ~/cs161
%mkdir temp
%cd temp
%cvs co -rasst1-start src
%cd src
%patch -p1 < ~/cs161/asst1.diff
%./configure --ostree=`echo ~/cs161/temp/root`
%cd kern/conf
%./config ASST1
%cd ../compile/ASST1
%make depend
%make
%make install
%cd ../../..
%make
%cd ../root
```

Place `sys161.conf` in your OS/161 root

```
%sys161 kernel
%gzip asst1.diff
```

Important! Before creating a tarred and compressed file for your project 3, please place the following three files in your `project3` directory:

```
1) codereading.txt
2) exercises.txt
3) asst1.diff.gz
```

Let us assume that your `project3` directory is under the `cs161` directory. You can create a tarball using the following command:

```
%cd ~/cs161
%tar vfcz <group_ID>_project3.tgz project3
```

Important! Note that the above commands will not work if you do not place your project 3 directory inside the `cs161` directory.

Now, submit your tarred and compressed file named <group_ID>_project3.tgz through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted. For example, suppose that I am a member of group6, my submitted file would read "group6_project3.tgz".

7. Grading Criteria

1) Implementing Locks: 15%
2) Implementing Condition Variables (cv): 15%
3) A semaphore-based solution in `catsem.c`: 15%
4) A locks based solution possibly with condition-variables in `catlock.c`: 15%
5) Adhering to coding style and documentation guidelines: 10%.
6) Written exercises (`codereading.txt` and `exercises.txt`): 30%.

8. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

9. Rebuttal period

- You will be given a period of 72 hours to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.