COMP 3500 Introduction to Operating Systems
Project 3 – Synchronization Mechanisms

Long Version 1.2
Revised: June 3, 2022

There should be no collaboration among students. A student shouldn't share any project code or solution with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Objectives:

- To implement the lock mechanism
- To implement condition variables
- To improve your source code reading skills
- To strengthen your debugging skill

1. Project Goals

This project assignment provides you with the opportunity to learn how to implement basic synchronization mechanisms covered in our lectures. 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 the goal of this project is to implement locks and condition variables.

2. Getting Started

2.1 Setup `$PATH`

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

2.2 Create a New Git Repository

This project does not rely on Project 2 and; therefore, you will start with a new Git repository. 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 src archive/src-project2
tar vfzx os161-1.10.tar.gz
mv os161-1.10 src
cd src
git init
git add .
git commit -m "ASST1a initial commit"
```

Prior to working on this project, please tag your Git 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.

```
git tag asst1a-start
```

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

2.4 Building for ASST1

You can follow the three commands to build OS/161 for this project.

```
cd ~/cs161/src/kern/compile/ASST1
make depend
make
make install
cd ~/cs161/src
make
```

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.

2.6 Physical Memory

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 line, where 2097152 bytes is 2MB.

```
31 busctl ramsize=2097152
```

2.7 One-line Command to Rebuild and Test

You will have to run the make command four times before testing any code changes you make. You can simply use the following single command:

```
cd ~/cs161/src/kern/compile/ASST1 && make depend && make && make
install && cd ~/cs161/root && ./sys161 kernel
```

The "&&" between each make command runs only after the successful completion of the previous command.

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.

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 cs161-gdb to see how the scheduler acts, how threads are created, and what exactly happens in a context switch.

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. 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`. You should store `codereading.txt` in directory "`~/cs161/project3`". If you haven't yet created this directory, please run the following three commands. The `touch` command creates an empty text file named `codereading.txt`. You may use any text editor (e.g., `vi`) to modify this file.

```
cd ~/cs161
mkdir project3
touch codereading.txt
```

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

You need to implement a total of five functions to support the lock mechanism; you should implement the other five functions for the CV mechanism (see Sections 5.1 and 5.2). You must also modify `thread.h` and `thread.c` (see Section 5.3).

5.1 Synchronization Primitives: Implementing Locks

In the first programming exercise of project 3, 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`. In addition, you must modify synch.h to update the data structure for the lock mechanism.

Important! You may use the implementation of semaphores as a model, but you should **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 thread.h and thread.c: Implement `thread_wakeone()`

You should modify `threat.h` to add the following new function prototype for waking up a single thread that is sleeping on a specified address (e.g., the addresses of semaphores, locks, or CVs). You must, of course, modify thread.c to implement the function `thread_wakeone()`.

```
void thread_wakeone(const void *addr);
```

6. Deliverables

Make sure the final versions of all your changes are added and committed to the `Git` repository before proceeding. We assume that you haven't used `asst1a-end` to tag your repository. In case you have used `asst1a-end` as a tag, then you will need to use a unique tag in this part.

```
cd ~/cs161/src
git add .
git commit -m " ASST1a final commit"
git tag asst1a-end
git diff asst1a-start..asst1a-end > ../project3/asst1a.diff
```

`asst1a.diff` should be in the `~/cs161/project3` directory. It is prudent to carefully inspect your `asst1a.diff` file to make sure that all your changes are present before compressing and submitting this file through `Canvas`. It is your responsibility to know how to correctly use `Git` as a version control system.

Important! Before creating a tarball for your project 3, please ensure that you have the following two files in the `~/cs161/project3` directory.

```
codereading.txt and
asst1a.diff
```

You can create a tarball using the commands below:

```
cd ~/cs161
tar vfzc project3.tgz project3
```

Now, submit your tarred and compressed file named `project3.tgz` through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted).

7. Grading Criteria

1) Implementing Locks: 35%
2) Implementing Condition Variables (cv): 35%
3) Adhering to coding and comment styles and documentation guidelines: 10%.
4) Written exercises (`codereading.txt`): 20%.