# COMP 3500 Introduction to Operating Systems
# Project 3 – Synchronization Mechanisms

Long Version 5.4
Revised: October 11, 2023

Points Possible: 100.
Submission via Canvas.

**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.

**Requirements**
- This project assignment has to be done individually by each student.
- You should accomplish the written exercises and submit `codereading.txt`.
- The coding exercise consists of two subtasks (i.e., the implementations of *lock* and *CV*).
- You are allowed to discuss with other students to solve the coding problems at the design level (e.g., algorithms) rather than the programming level.

**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. 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 in the next project (i.e., Project 4). On completion of this programming project, you are expected to demonstrate an ability to develop *lock* and *CV* mechanisms supporting 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 the goal of this project is to implement locks and condition variables.

## 2. Getting Started
### 2.1 Setup `$PATH`
You need to have the correct directories in your `$PATH`. 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
```

### 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. 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 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 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 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 follow the three commands below to build OS/161 for this project.

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

```
        make
        make install
        cd ~/cs161/src
        make
```

**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`). If you have placed this file in the `root` directory when you carried out project 2, you don't have to overwrite the old `sys161.conf` file.

**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, which is located in `~/cs161/root`. The busctl device line looks like the following line, where 2097152 bytes is 2MB.

```
        31 busctl ramsize=2097152
```

You may edit sys161.conf using the following command

```
        gedit sys161.conf
```

**2.7 One-line Command to Rebuild and Test**

Recall that to rebuild the entire os161 kernel, you will have to run the make command four times before testing any code changes you make. Instead of typing each command one after the other, you can just 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.

**CAUTION!** You may use the above one-line command if your source code is compilation-error free. You should not use this approach to rebuilding the kernel in case there are any potential compilation errors. Rather, you should follow the instructions from Project 2 Section 6 on page 11 to rebuild the kernel.

## 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 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.

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`. 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
```

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. You need to fill out the ten functions in synch.c located in the `~/cs161/src/kern/thread` directory. More specifically, you must 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 slightly modify `thread.h` and `thread.c` (see Section 5.3 for the details).

**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()`

**Function Declaration:** 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). It is worth noting that a function prototype is a function declaration that declares the function's name, return type, and parameters without providing the actual implementation.

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

**Function Implementation:** You must, of course, implement function `thread_wakeone()` in source code file thread.c. The implementation of `thread_wakeone()` is almost identical to that of except `thread_wakeone()` has a break statement to alter the flow of loop finding threads to be waked up.

## 6. How to test your lock and CV implementations?

There is no need to write a driver to test the correctness of your lock and CV implementations because the two test drivers were fully built in os161. Please following the commands below to test your Lock/CV implementations.

**Step 1:** Run the os161 kernel

```
cd ~/cs161/root
./sys161 kernel
```

**Step 2:** On the os161's command line, type the following command to list the menu options of the test drivers

```
?t
```

**Step 3:** On the os161's command line, type the following command to test the lock implementation.

```
sy2
```

If you correctly implement the lock mechanism, then your output will be similar to the following sample.

```
Operation took 0.163270280 seconds
OS/161 kernel [? for menu]: sy2
Starting lock test...
Lock test done.
Operation took 3.433237400 seconds
```

**Step 4:** On the os161's command line, type the following command to test your CV implementation.

```
sy3
```

If your CV implement is correct, then the output of should be similar to the sample below.

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

```
Starting CV test...
Threads should print out in reverse order.
Thread 31
Thread 30
...
Thread 1
Thread 0
Thread 31
...
Thread 1
Thread 0
Thread 31
...
Thread 1
Thread 0
Thread 31
...
Thread 1
Thread 0
Thread 31
...
Thread 1
Thread 0
CV test done
Operation took 5.203599240 seconds
```

**7. 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).

## 8. 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%.

## 9. 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 two-fold. First, there is a likelihood that you will read and understand code written by yourselves in the future. Second, 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 write 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 using `Git`. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

## 10. 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.

## 11. Rebuttal Period

- You will be given a period of one week (7 days) 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.