CS 471 Operating Systems
Fall 2021

Programming Assignment 1 (PA1)
Due date for Part 1: Thursday, October 14, 11:59 pm
Due date for Part 2: Thursday, October 28, 11:59 pm

Table of contents

1. Introduction

We finished the discussion of threads and general synchronization in class, and you became familiar with the basic OS/161 environment in the first assignment (PA0). In this assignment you will implement synchronization primitives for OS/161 and learn how to use them to solve a synchronization problem. Once you have completed the written and programming exercises you should have a fairly solid grasp of the pitfalls of concurrent programming and synchronization.

This project has two parts with two distinct deadlines: Part 1 is due within two weeks and it consists of your answers to code reading questions, declaration of your project partner (if any), and a short progress report. Part 2 is due within four weeks (i.e., two weeks after the deadline for Part 1) and it consists of your coding solutions – the programming part.

To complete this assignment you will need to be familiar with the OS/161 thread code. The thread system provides interrupts, control functions, and semaphores.

Note that in addition to developing non-trivial synchronization code in C, you will need to again do some code reading and directory exploring in OS/161. This is normal. Considering that debugging synchronization programs requires also more time, make sure to allocate sufficient time from the beginning – if you start just a few days before the due date, you are likely to experience significant difficulties.

Working with partner

In this assignment you can - and you are ENCOURAGED to - work with a partner: another CS 471 student from your own section. No team can have more than two members. If you have worked alone in Project 0, you can form a team with another CS 471 student who worked alone in Project 0. You can also continue to work with the same partner you worked with in Project 0 (however, you cannot switch to a new partner). If believe you can finish it without a partner, you are welcome to work alone; but there will not be any bonus points for those working alone, and you should keep in mind that the assignment specification has been developed assuming that two students will actively interact and cooperate in exploring OS/161 and developing a synchronization program.

2. Begin your assignment

Download necessary files

For this assignment, you need to replace 3 of your existing files as follows:

- `menu.c` in 'kern/main/'
- `test.h` in 'kern/include/'
- `stoplight.c` in 'kern/asst1/'

The updated versions of these files can be downloaded from Blackboard. You must replace these three files accurately before you start coding the solution.

Tag your Git Repository

Before you do any real work on this assignment, tag your Git repository. The purpose of tagging your repository is to make sure that you have something against which to compare your final tree. Make sure that you do not have any outstanding updates in your tree. Use `git commit` to get your tree committed in the state from which you want to begin this assignment.

```
% git commit -am "End of project-0"
% git push
```

Now, tag your repository as shown below.

```
% cd ~/os161/os161-1.11
% git tag asst1-begin
```

NOTE: If you are working with a partner, you may want to set up the GitLab repository so that it may be accessed by both members of your group. In case you have not setup GitLab, the instructions for the initial setup can be found here:
https://mason.gmu.edu/~zan2/gitlab_setup.html

Configure OS/161 for ASST1

We have provided you with a framework to run your solutions for ASST1. This framework consists of driver code (found in kern/asst1/stoplight.c) and menu items that can be used to execute the 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:

```
% cd ~/os161/os161-1.11/kern/conf
% ./config ASST1
```

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

Building for ASST1

As in PA0, in order to build kernel, you have two options. You can use the php file provided by the TA or use the alternative method. Before starting any of the methods, make sure to issue the module load command

```
% module load sys161/1.14
```

IMPORTANT: You should remember to issue the module load command above every time you logon to zeus-2 when you intend to work on os161! (the command will not work on zeus-1).

In the first method, you must enter the following:

```
% wget http://mason.gmu.edu/~zan2/build-asst1.php
% php build-asst1.php
```

Or as the second method, you can run make from compile/ASST1.

```
% cd os161-1.11
% ./configure --ostree=$HOME/os161/root
% cd kern/conf
% ./config ASST1
% cd ~/os161/os161-1.11/kern/compile/ASST1
% make depend
% make
% make install
```

Command Line Arguments to OS/161

Your solutions to 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: DO NOT change the menu option strings!

"Physical" Memory

In order to execute the tests in this assignment, you will need more than the 512 KB of memory configured into System/161 by default. We suggest that you 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 (located under ~/os161/root/) . Make sure the `busctl` device line looks like the following:

`31 busctl ramsize=2097152` (Note: 2097152 bytes is 2MB).

Helpful hint: The php script will replace this sys161.conf file every time it is executed. Commenting out line 38 in the php script will fix that.

3. Concurrent Programming with OS/161

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you (and we!) should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

Built--in thread tests

When you booted OS/161 in ASST0, you may have seen the options to run the thread tests. The thread test code uses the semaphore synchronization primitive. You should 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 be able to 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 starvation -- the threads should all start together, spin for a while, and then end together.

Debugging concurrent programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, 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. This means 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
```

We recommend 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 should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

4. Part 1 (25 points)

Code reading

Please answer the following questions and save your answers in a file named "code-reading.txt".

To implement synchronization primitives, you will 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. Place the answers to the following questions in `code-reading.txt`.

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. What does it mean for a thread to be in each of the possible thread states?
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?

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?

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()`?

Once you finish the code reading questions, create a second file named "progress-report.txt"

In that file, first give the full names and G numbers of the team members who decided to collaborate in project PA1 (see the "Working with a partner" paragraph in Section 1 of the hand-out for the rules governing partner selection). Then, in the same file, give a short (around one-page) summary of the progress you made on the design and coding part of your project. Do not provide any code; just summarize your efforts. Finally, briefly explain the responsibilities of the team members in Part 2 of the project (the coding part). Failure to submit the progress report may result in deduction of points.

You can work with a partner in Part 2 (Coding Part) below only if that student's name is explicitly stated in Part 1 submission in a timely manner.

Submission of Part 1:

Create a directory called 'asst1-part1', inside the '~/os161/' directory.

```
% cd ~/os161/
% mkdir asst1-part1
```

When both code-reading.txt and progress-report.txt files are ready, put these two files in it.

Next, tar and compress your asst1-part1 directory.

```
% cd ~/os161
% tar -czf uid1_uid2-asst1-part1.tar.gz asst1-part1
```

You should replace `uid1` and `uid2` with you and your partner's GMU email IDs. If you are working alone, the last line should read `tar -czf uid-asst1-part1.tar.gz`. Submit the compressed Part 1 file on Blackboard**. All members of a group must submit separately the same compressed file, and before the Part 1 deadline 10/14 11:59 PM . Make sure to coordinate.**
Make sure to double check that you are submitting the correct file -- if you submit wrong, corrupt, or empty file you are likely to get zero.

Late penalty for Part 1 will be 15% for each day. Submissions that are late by 3 or more days will get zero. Please plan in advance.

You can make multiple submissions; we will consider ONLY the last submission that you make: so in case you need to re-submit, make sure the compressed file contains all the components listed above.

Note: The contents of Part 1 should be submitted separately from Part 2, and in a timely manner.

5. Part 2 - Coding (75 points)

Synchronization Primitives (20/75 Points)

Implement (mutual exclusion) 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`. You can use the implementation of semaphores as a model, but do NOT build your lock implementation on top of semaphores; your code should not use semaphores (or condition variables) in any way. The stub code also contains code templates for condition variables; you should ignore those parts.

Solving the Synchronization Problem (55/75 Points)

The following problem will give you the opportunity to write some fairly straightforward concurrent programs and 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. Remember to specify a seed to use in the random number generator by editing your `sys161.conf` file. It is much easier to debug initial problems when the sequence of execution and context switches is reproducible.

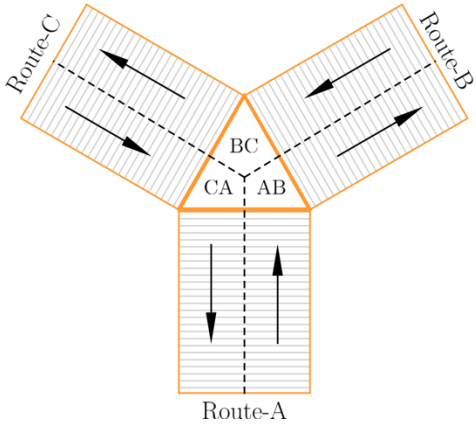
When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in. For example, you will see an option for "lock variables"; you can choose that option to see if your lock implementation is correct or not. Similarly by invoking the "stoplight.c" option (typing "1c") in the menu, you can run an initial test for your driver code `stoplight.c` implementation (The grader is likely to run additional tests; but you can consider these as initial tests). Note: typing '?t' on the kernel command line will show the full test list in os161 and then use 'sy2' on the kernel command line to test your lock implementation. You can ignore the 'elapsed time' information displayed by those tests.

Synchronization Problem: Traffic Management at Podunk

You must solve this problem using the locks that you implemented above. Other solutions (condition variables, semaphores, etc.) are not acceptable.

Traffic through the main intersection in the town of Podunk, KS has increased over the past few years. Until now the intersection has been a three-way stop but now the impending gridlock has forced the residents of Podunk to admit that they need a more efficient way for traffic to pass through the intersection. Your job is to design and implement a solution using the synchronization primitives (locks) that you have developed in the previous part.

Modeling the intersection



For the purpose of this problem we will model the intersection as shown above, dividing it into three portions (AB, BC, CA) and identifying each portion with the names of the lanes entering/leaving the intersection through that section. (Just to clarify: Podunk is in the US, so we're driving on the right side of the road.) Turns are represented by a progression through one or two portions of the intersection (for simplicity assume that U-turns do not occur in the intersection). So if a car approaches from Route-A, depending on where it is going, it proceeds through the intersection as follows:

- Right: AB
- Left: AB-BC

In addition to "Cars", there are also "Trucks" in Podunk. Trucks have a lower priority than cars when entering the intersection. If a lane in a given route has both cars and trucks in it, then the cars should cross the intersection before the trucks in that route. When no more cars are waiting, then the trucks on that lane can cross the intersection. It is fine if some trucks suffer starvation (i.e., cross last). Note that, cars have higher priority than trucks approaching the intersection through the same route (Route-A, Route-B, or Route-C). For example, if there are six vehicles (four cars and two trucks) approaching the intersection through Route-A, the two trucks should not start to cross before any of the four cars.

Before you begin coding, answer the follow questions in a file named exercises.txt (to be submitted in Part 2, along with your code):

1. Assume that the residents of Podunk are exceptional and follow the old (and widely ignored) convention that whoever arrives at the intersection first proceeds first. Using the language of synchronization primitives describe the way this intersection is controlled. In what ways is this method suboptimal?

2. Now, assume that the residents of Podunk are like most people and do not follow the convention described above. In what one instance can this three-way-stop intersection produce a deadlock? (It will be helpful to think of this in terms of the model we are using instead of trying to visualize an actual intersection).

Implementing your solution

The file you are to work with is in `~/os161/os161-1.11/kern/asst1`. Ignore `catsem.c` and `catlock.c`. You only need to work with `stoplight.c`
We have given you the model for the intersection. The following are the requirements for your solution:

- Each vehicle "approaches" the intersection when the corresponding thread is created, it "enters" the intersection when the conditions are right for it to proceed (see below), and after crossing the intersection, it "leaves". No two vehicles can be in the same portion of the intersection at the same time. (In Podunk they call this an accident).
- Cars have a higher priority than trucks. Before proceeding to (entering) the intersection, the trucks in a given lane must wait until all cars (which have already approached) on that lane have entered the intersection.
- Your solution must improve traffic flow without allowing traffic from any direction to starve traffic from any other direction. However, as per the spec, the trucks are supposed to proceed last in a given lane and that is how you should implement (i.e., extended waiting experienced by the trucks in the presence of cars is not a problem).
- For full credit, your solution should maximize concurrency across threads and should not impose unnecessary delays not imposed by the constraints given above. For example, while a vehicle X is crossing the intersection, it should not unnecessarily delay the entry of another vehicle (approaching from the same route or different route) to the intersection if the other constraints of the problem are not violated. Similarly, having the vehicles cross the intersection one by one is not acceptable.
- A truck must not wait for cars which have not approached yet (because this will incur unnecessary delay.) A truck can (and should) enter the intersection if its path does not cause a car to get delayed. A truck should not wait for any "future" cars which have not yet approached the intersection. In other words, generating all the vehicles approaching a given lane first, putting them in priority order (cars first), and only then allowing the vehicles to proceed to the intersection is not acceptable.
- Each vehicle should print a message as it approaches, enters, and leaves the intersection indicating the vehicle number, vehicle type (car or truck), approach direction and destination direction. You should also print messages to clearly show when a vehicle is in a specific portion (AB, BC, CA) of the intersection. The messages should unambiguously describe all the events in the intersection with proper ordering. This message is very important and it is your task to make sure that the message is displayed with clarity to allow the grader to follow all the events! (If necessary use synchronization primitives to enforce clear printing). Projects whose vehicle thread execution order cannot be clearly followed will not receive substantial scores.

Your code should be based on the locks that you implemented. Using other synchronization primitives or resorting to busy waiting is not allowed. This last constraint implies that a vehicle thread which cannot make progress in the intersection should not keep continuously checking the condition; instead it should suspend its execution to enable other threads to execute on the processor. It is your task to figure out how to achieve that objective efficiently.

The driver for the Podunk Traffic problem is in (the version that you updated from Blackboard) `~/os161/os161-1.11/kern/asst1/stoplight.c` (a not so subtle hint about one possible solution). It consists of `createvehicles()` which creates 20 vehicles and passes them to `approachintersection()` which assigns each a random direction. It also assigns each vehicle a random type as 'car' or 'truck'. We assigned them a random turn direction as well. The file `stoplight.c` also includes routines `turnright()` and `turnleft()` that may or may not be helpful in implementing your solution. Use them or discard them as you like.

Please note: The OS-161 menu should not appear on the screen until all the car (and truck) threads have completed; this is part of the synchronization problem you are solving!

6. Recommendations

If this is your first multi-threaded programming project (likely to be the case for most students) make sure to allocate sufficient time, in particular for testing and debugging. It's not uncommon to spend very significant time on debugging first multi-threaded programs, because it takes time to adjust to thinking with multiple threads executing the same program. That is why you are given several weeks to complete this project.

It is our strong recommendation that you develop the solution to the traffic management problem incrementally to make debugging a tractable task; for example, by developing and testing the locks, then modeling the intersection by allowing the vehicles to cross in any order, ignoring the "accidents", and printing the events properly. While that is not an acceptable solution to the project, it will give you good know-how about creating threads and displaying their progress at run-time, so that you can proceed with confidence.
In the next phase, you can update your solution to prevent accidents, and when successful, you can add high priority to cars. Finally, you can make sure that your solution maximizes concurrency across threads and doesn't impose any unnecessary delay.

All these phases require good planning and working according to a timetable through multiple weeks. Last-minute scrambles are known to be not very helpful in multithreaded projects.

7. Coding style, submission and grading

In your programming assignments, you are expected to write well-documented, readable code. There are a variety of reasons to strive for clear and readable code. Since you will be working in pairs, it will be important for you to be able to read your partner's code. Also, since you will be working on OS/161 in a future assignment, you may need to read and understand code that you wrote several weeks earlier. Finally, clear, well-commented code makes your TA happy!

Here are some general tips for writing better code:

- Group related items together, whether they are variable declarations, lines of code, or functions.

- Use descriptive names for variables and procedures. Be consistent with this throughout the program.
- Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find an eligible car" is much more informative than one that says "Find the first non-zero element of array."

You and your partner will probably find it useful to agree on a coding style - for instance, you might want to agree on how variables and functions will be named since your code will have to interoperate.

Working Environment

You will need to develop your code on zeus-2 server CEC. Your programs will be tested on zeus-2 (no exceptions!).

Submission for Part 2:

When you are finished create a directory called 'asst1-part2', inside the '~/os161/' directory.

```
% cd ~/os161/
% mkdir asst1-part2
```

Make sure you commit your latest working copy first:

```
% cd ~/os161/os161-1.11/
% git commit -am "Completed project-1"
```

Tag your latest working copy and run a diff:

```
% git tag asst1-end
% git diff asst1-begin asst1-end > ../asst1-part2/asst1.diff
```

In the 'asst1-part2' directory, you should place the following:

`asst1.diff:` a diff file listing all the changes you made for this assignment.
your `sys161.conf` file.
`exercises.txt:` your answers to the written synchronization exercises (do NOT include `code-reading.txt`)

Next, tar and compress your asst1-part2 directory AND your entire source tree.

```
% cd ~/os161
% tar -czf uid1_uid2-asst1-part2.tar.gz os161-1.11 asst1-part2
```

You should replace `uid1` and `uid2` with you and your partner's GMU email IDs. If you are working alone, the last line should read `tar -czf uid-asst1-part2.tar.gz`. Submit the compressed Part 2 file on Blackboard**. <u>All members of a group must submit separately the same compressed file, and before the Part 2 deadline 10/28 11:59 PM . Make sure to coordinate.</u>**
<u>Make sure to double check that you are submitting the correct file -- if you submit wrong, corrupt, or empty file you are likely to get zero.</u>

You can make multiple submissions; we will consider ONLY the last submission that you make: so in case you need to re-submit, make sure the compressed file contains all the components listed above.

Late penalty for Part 2 will be again 15% for each day. Projects that are late by 3 or more days will not be accepted. Please plan in advance.

Questions

Programming – and system-related questions about the project should be directed Fall 2021 CS 471 OS/161 Project Piazza Page, which is monitored by the GTAs from 10 AM to 6 PM on weekdays. Your posts to the Piazza are by default *private*, meaning that it is received only by the instructors and GTAs. Occasionally, the GTAs can decide to make some questions and answers *public,* if they think they are relevant for many students. Also: the GTAs are not allowed to reveal/code the solution or debug your programs; however, they can help with clarification and guidance.

The code-reading and exploring the OS/161 kernel files is a component of the assignment, so you should not expect detailed explanations of how the OS/161 kernel functions work (including the thread functions). You can direct conceptual questions about the Podunk Traffic Synchronization problem specification to the Piazza.

Similarly the Piazza page has specific threads you can use to post your "partner-search" ads, classified by CS471 sections. Post your ad to the appropriate thread, if you are looking for a partner.

Grading

This programming assignment (Part 1 and Part 2 combined) accounts for **12%** of your final grade.
Good luck!