# CS350 Operating Systems Winter 2016

## Assignment One

This assignment has two parts. The first part, described in Section 1, requires you to implement several kernel synchronization primitives. The second part (Section 2) requires you to solve a thread synchronization problem.

**Important:** before you start working on this assignment, you should reconfigure and rebuild your OS/161 kernel:

```
cd kern/conf
./config ASST1
cd ../compile/ASST1
bmake depend
bmake
bmake install
```

All kernel builds for this assignment should occur in the `kern/compile/ASST1` directory.

# 1 Implement Kernel Synchronization Primitives

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

## 1.1 Implement Locks

Your first task is to implement locks. 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.

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.**

## 1.2 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`. 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*. **Do not build your condition variable implementation on top of semaphores** or you will be penalized.

## 1.3 Testing Locks and Condition Variables

The file `kern/test/synchtest.c` implements a simple test case for locks, and another for condition variables. You can run the lock test from the kernel menu by issuing the `sy2` command, e.g,:

```
% sys161 kernel "sy2;q"
```

Similarly, the `sy3` command will run the condition variable test. If the lock test reports "Lock test done" without reporting any failure messages, it has succeeded. The output from the condition variable test should be self-explanatory.

Testing synchronization primitives like locks and condition variables is difficult. Both `sy2` and `sy3` are subject to false positives. In other words, an incorrect lock or condition variable implementation may pass these tests. However, if your implementation *fails* a test, there is definitely a problem. Since the synchronization tests are not perfect, we may use code inspection - in addition to testing - to evaluate your lock and condition variable implementations.

Although you are free to implement locks however you want, **you should not modify any of the kernel's test programs**, i.e., do not modify any of the files in `kern/test`. Furthermore, you should not make any changes to the way that the tests are invoked, e.g., do not change "`sy2`" to "`sy2a`".

# 2 Solve a Synchronization Problem

For this part of the assignment, you are expected to implement a solution to a synchronization problem called the "traffic intersection" problem. The synchronization primitives that you may use in your solution are **semaphores, locks, and condition variables**. You are free to use whichever of these synchronization primitives you choose, however you like. However, **you must not directly use any "lower-level" methods of synchronization**, such as wait channels or spinlocks.

**The Traffic Intersection Problem**

In the traffic intersection problem, vehicles are trying to pass through an intersection of two roads, one north/south, the other east/west, without colliding. Each vehicle arrives at the intersection from one of four directions (north, south, east, or west), called it's *origin*. It is trying to pass through the intersection and exit in some direction other than its origin, called it's *destination*.

In OS/161, each arriving vehicle is simulated by a thread, and the intersection is a critical section. Each vehicle (thread) enters the intersection (critical section) and eventually leaves. The vehicle simulation is implemented in the file `kern/synchprobs/traffic.c`. You are free to inspect the code in this file to understand the simulation. However, **you may not change this file in any way.**

The vehicle simulation works by creating a fixed number of concurrent threads. Each thread simulates a sequence of vehicles attempting to pass through the intersection. In a loop, each thread generates a random vehicle (randomly choosing an origin and destination for the vehicle) and then enters and leaves the critical section to simulate the vehicle passing through the intersection. Threads then sleep for a configurable period of time before generating another random vehicle and repeating the process. Each thread simulates one vehicle at a time. However, since there are multiple concurrent threads, there may be multiple vehicles attempting to enter the critical section (intersection) concurrently.

Your job is to synchronize the vehicles so that they do not collide in the intersection. Informally, it is OK for more than one vehicle to be in the intersection at the same time as long as their paths will not result in a collision. For example, it is OK for a vehicle that is traveling from north to south to share the intersection with another vehicle that is traveling from south to north, but not with one that is traveling from east to west. For the purposes of this assignment, we've codified a specific set of conditions under which vehicles can safely share the intersection (critical section). If two vehicles, $V_a$ and $V_b$, are in the intersection simultaneously, then *at least one* of the following must be true:

- $V_a$ and $V_b$ entered the intersection from the same direction, i.e., $V_a.origin = V_b.origin$, or
- $V_a$ and $V_b$ are going in opposite directions, i.e., $V_a.origin = V_b.destination$ and $V_a.destination = V_b.origin$, or
- $V_a$ and $V_b$ have different destinations, and at least one of them is making a right turn, e.g., $V_a$ is right-turning from north to west, and $V_b$ is going from south to north.

Note that it is possible for more than two vehicles to be in the intersection at the same time, as long as all pairs of vehicles satisfy one of the above conditions.

These conditions are less flexible than what is allowed at some real road intersections. For example, the conditions above do not allow cars traveling in opposite directions to make left turns concurrently. Nonetheless, for the purposes of this assignment, these somewhat simplified conditions will serve as our definition of correctness.

It is up to you to determine exactly how to synchronize the vehicles - there are many ways to satisfy the synchronization requirements. OS/161 includes a simple default solution (in the file `traffic_synch.c`, see Section 2.1) which uses a single semaphore to ensure that only one vehicle at a time can enter the intersection. This is a correct solution - it trivially satisfies all of the conditions above. It is also a fair solution, with no risk that arriving vehicles will "starve". However, it is not an *efficient* solution because it misses opportunities to let more than one vehicle use the intersection at the same time when the synchronization rules permit it. For this assignment, we are looking for solutions that are correct, fair, and efficient.

## 2.1 Implementing Your Solution

In the directory `kern/synchprobs`, there is a file called `traffic_synch.c`. Your solution to the "traffic intersection" problem should be implemented entirely in this file.

The `traffic_synch.c` file contains four functions, which are invoked by the traffic simulation program:

- `intersection_before_entry`: called by a vehicle simulation before a vehicle enters the intersection.
- `intersection_after_entry`: called by a vehicle simulation after a vehicle leaves the intersection
- `intersection_sync_init`: called only once, before any vehicles try to enter the intersection
- `intersection_sync_cleanup`: called only once, at the end of the simulation.

`traffic_synch.c` includes a simple default implementation of these functions. It uses a single semaphore to ensure that only one vehicle at a time will enter the intersection. For this assignment, you should *re-implement* these functions to produce a solution that is correct, fair, and efficient. In particular, you should use the `intersection_before_entry` function to make vehicles wait before entering the intersection, so that the synchronization requirements are not violated. To do this, your implementation of `intersection_before_entry` should cause that vehicle (thread) to *block* until it is OK for it to proceed into the intersection without violating the synchronization rules. In other words, `intersection_before_entry` should not return until it is OK for the vehicle to enter the intersection, since the vehicle will enter the intersection once `intersection_before_entry` returns.

## 2.2 Testing

You can launch the traffic simulation from the OS/161 kernel menu using the `sp3` command. It requires 4 parameters, like this:

```
sys161 kernel "sp3 5 10 1 2;q"
```

These parameters specify the number of threads, the number of iterations (vehicles simulated) per thread, the vehicle inter-arrival time per thread (the time that the thread waits "between" vehicles), and the service time (the amount of time that a vehicle spends in the intersection once it enters. Thus, the command above causes the test to use 5 threads, each simulating a sequence of 10 vehicles, i.e., the total number of simulated vehicles is 50. Each thread waits 1 second between vehicles, and remains in the critical section (intersection) for 2 second before leaving.

When the simulation program terminates, it prints several simulation statistics, including the total time required to simulate all of the vehicles, as well as the average vehicle waiting time, broken down by arrival direction. We will use the total simulation time as a measure of the efficiency of your solution - lower times are better. We will use the average vehicle waiting times as an indication of the fairness of your solution. Since cars are equally likely to arrive from any direction, waiting times for cars arriving from all directions should be similar. (We expect some variations due to the randomness of the simulations, but your solution should not introduce systematic bias against cars arriving from certain directions, e.g. do not always favor cars from the north over cars from the south.)

# 3 What to Submit

You should submit your kernel source code 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 `$HOME/cs350-os161/os161-1.99`. To submit your work, you should run `cs350_submit 1` in the directory `$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 to the course account.