Assignment 2: Synchronization

Due Feb 15 by 10pm **Points** 8

Overview

For this assignment, we're going back to the realm of user mode programming. In this assignment, you will work with synchronization primitives to implement a guidance system for traffic simulation.

Autonomous (Self-driving) cars are increasingly becoming more reliable, as development of smart technologies help better guide the vehicle through its environment safely. Autonomous cars depend on several sensors and complex logic to coordinate with other vehicles in a safe manner. At an intersection for example, cars must have a policy to coordinate crossing the intersection in order to avoid crashes and to ensure that everyone goes through the intersection eventually.

Your job is to implement synchronization in a traffic system that coordinates cars moving through two types of intersections: stop signs and traffic lights, by enforcing ordering between car arrivals in the same lane, avoiding potential crashes, and avoiding deadlocks.

Details

Each car is implemented as a separate thread, so as it passes through an intersection it must use synchronization to prevent collisions and data structure corruption. A car is defined by two parameters:

- **position** or the direction from which is starts (NORTH, SOUTH, EAST, WEST)
- **action** or how it passes through the intersection (STRAIGHT, RIGHT_TURN, LEFT_TURN)

Because both the stop sign algorithm and the traffic light algorithm share many common functions, we have implemented both in a single program. Your task is to complete the implementation of `safeStopSign.c` and `safeTrafficLight.c` by adding the appropriate synchronization. You should used locks and condition variables (not semaphores).

Each vehicle in the system runs logic from its own thread and actions taken by each thread must be synchronized to prevent collisions and data structure corruption. Each car has two parameters: the direction it starts from and the action it will take.

Stop Sign

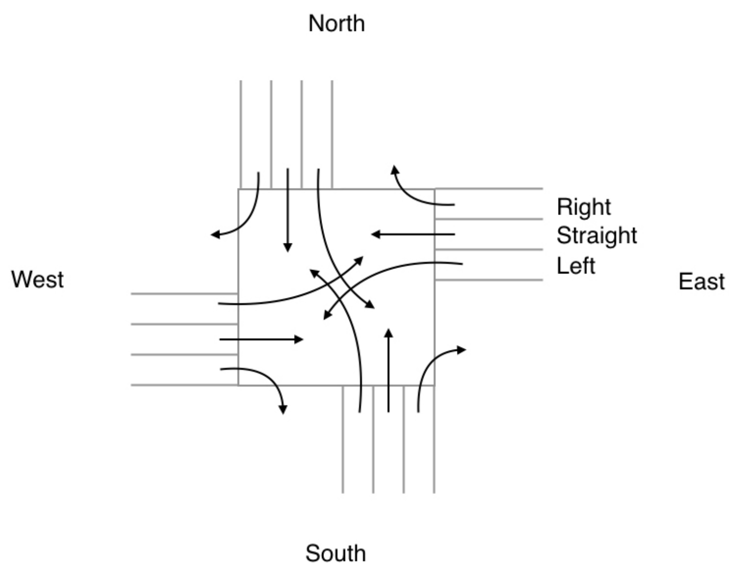
There are four entrance lanes and four exits to the intersection, each corresponding to a cardinal direction: North (N), South (S), East (E) and West (W). Each entrance into the intersection is referred to as a *lane* and the intersection itself is broken into four *quadrants*. Depending on the action, a car may pass through between one (right turn) and three (left turn) quadrants. A car may enter and perform its action provided that the quadrants it would pass through are completely clear of other cars, including any from the same direction.

In the stop sign simulation, each car thread follows 4 steps: (see `runStopSignCar()` in `safeStopSign.c`)

1. Enter the proper entry lane. (See `enterLane` in `intersection.c`)
2. Move through the intersection. (See `goThroughStopSign` in `stopSign.c`)
3. Formally exit the intersection; ensure that cars exit the intersection in the same order they entered their entry lane.(See `exitIntersection` in intersection.c)

Traffic Light

In the traffic light scenario, there are twelve entrance and exit lanes: one for each possible combination of direction and action. For example, depending on the action, a car entering from East may enter a left-turn lane, straight lane or right-turn lane. Unlike in stop sign, the only requirements for a car to enter the intersection is that it be in a lane and that the traffic light is green for its direction. Likewise, due to the arrangement of the lanes, the only danger of collision comes from left-turns. A vehicle wanting to make a left-turn first enters the intersection and then when there are no cars starting from the opposite direction going straight in the intersection, makes its left turn.



To simplify things, the traffic light has just three states: green for north-south, green for east-west and red. (There is only one traffic light.) After a variable number of cars enters the intersection, the light turns red. Once red, and once all the cars entered while it was green have exited, the light turns green for either N-S or E-W. Note that you are not allowed to assume that the light will turn green for the opposite orientation as before (you could see green N-S -> Red -> green N-S).

In the traffic light simulation, each car thread follows 4 steps: (see `runTrafficLightCar()` in `safeTrafficLight.c`)

1. Enter the proper lane.(See `enterLane` in `intersection.c`)
2. Enter the intersection when safe.(See `enterTrafficLight()` in `trafficLight.c`)
3. In the traffic light intersection: confirm it is safe to make the desired move. (See `actTrafficLight` in `trafficLight.c`)
4. Move through the intersection.(See `actTrafficLight` in `trafficLight.c`)
5. Formally exit the intersection; ensure that cars exit the intersection in the same order they entered their entry lane.(See `exitIntersection` in `intersection.c`)

Implementation

Your task is to add synchronization to the functions, `runStopSignCar()` and `runTrafficLightCar()` in `SafeStopSign.c` and `SafeTrafficLight.c` respectively. You should also add members to the `SafeStopSign` and `SafeTrafficLight` structs defined in `SafeStopSign.h` and `SafeTrafficLight.h` respectively as needed. You should not change any other code.

The code you write is part of a larger simulator framework. Starting in `carsim.c`, the program runs test cases defined in `testing.c`. These in turn will run your logic from `SafeStopSign.c` and `SafeTrafficLight.c.` The simulator is designed in a way that will help detect synchronization problems. When a car performs actions such as moving through intersections, its thread will sleep to simulate the time taken by the action. This also provides an opportunity to detect collisions, etc. Also, when a car performs some actions, the framework will provide it with a token. These are validated later to detect corruption or incorrect usage.

As mentioned above, you must also ensure proper lane ordering. If two cars that enter in the same lane, A and B, call `enterLane()` in that order, even if B wakes up and finishes going through the intersection before A, you must ensure that A and B call `exitIntersection()` in the same order.

In the traffic light scenario, entering (`enterTrafficLight()`) and actually moving through it (`actTrafficLight()`) are two separate operations. Also, `actTrafficLight()` accepts two callbacks (function pointers) that you can use to specify your own custom logic before and after the car physically moves through the intersection.

- carsim.c: Defines main() for the program. You can specify which simulations to run.
- car: Defines a car and a few related enumerations and helper functions.
- common.h: Defines a few helper functions.
- intersection.c: Defines logic common to the two scenarios, such as lanes.
- testing.c: Defines the simulation test cases. You can add your own test cases for better coverage.
- mutexAccessValidator: Defines a tool used to check for collisions in the stop sign scenario.
- SafeStopSign, SafeTrafficLight: Defines the functions each thread runs for each scenario.
- StopSign, TrafficLight: Defines the two scenarios. Check the headers for useful helper functions.

Notes:

- Unlike in real life, there are no limits on the number of cars allowed to enter the traffic light intersection and waiting to make a left turn.

- To simplify things in the traffic light intersection, you only need to check that there are no cars in the intersection from the opposite direction coming straight immediately before making a left-turn. You do not have to check that there are no left-turners when going straight.
- Rules:
  - Your code is run as part of a simulator framework with files that may be overwritten when we run your submission. Don't put any custom logic you may need in any of the other files!
  - You may not directly access members in the following structs (all interactions with them must be done through the functions given):
    - TrafficLight
    - StopSign
    - Lane
    - CarToken, IntersectionPart, MutexAccessValidator
- To earn full marks:
  - Your code should not be overly restrictive. E.g., if two cars can pass through an intersection at the same time, your code should not force one of them to wait.
  - Use condition variables to notify other threads when an event occurs – don't use a while loop with polling.
  - Properly destruct and free any objects and allocations you make.
  - Perform error checking.

Testing

The purpose of the mutexAccessValidator is to print messages to help you debug your code when the synchronization is not quite right. You can test your programs on relatively small numbers of cars (e.g. 16) and still expect to see some problems. As you become more confident that your synchronization is correct, you can try it out on larger numbers of cars and a higher number of experiments (say 80). Don't go crazy though, especially on wolf.teach.cs.toronto.edu. You will eventually hit a limit of the number of threads you can create.

Testing programs that involve synchronization primitives is difficult. Data races, deadlocks, and other synchronization problems may occur during one run but not the next, which makes detection of synchronization-related bugs a tedious (and frustrating) task. As computer scientists, solving frustrating bugs is your bread and butter though, right? Well, it doesn't have to be. Luckily, many share the same frustration of having to deal with complicated synchronization bugs for days at a time. As a result, automated tools for detecting possible synchronization problems are being developed and perfected, in order to assist programmers with developing parallel code.

One such tool is `valgrind`, which you may have used before in checking other problems in your code (e.g., memory leaks). Valgrind's instrumentation framework contains a set of tools which performs various debugging, profiling, and other tasks to help developers improve their code.

One of `valgrind`'s tools is called `helgrind`, a synchronization error checker (for the lack of a better word), which is tasked with helping developers identify synchronization bugs in programs that use POSIX pthreads primitives. You are encouraged to read through the **Documentation (http://valgrind.org/docs/manual/hg-manual.html)** for `helgrind`, in order to understand the kinds of synchronization errors it can detect and how each of these are reported.

The simplest example of how to use `helgrind` is as follows:

```
valgrind --tool=helgrind ./carsim stop 1 16
```

However, before you start testing your assignment code, you might want to consider trying it out on much simpler code, like the samples provided in lecture. For example, try testing `helgrind` on the producer-consumer solution with condition variables: **pc_cond.c**. As you can see, no synchronization errors are reported, and you should get a report that ends similarly to the one below:

```
==9395==
==9395== For counts of detected and suppressed errors, rerun with: -v
==9395== Use --history-level=approx or =none to gain increased speed, at
==9395== the cost of reduced accuracy of conflicting-access information
==9395== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 94 from 25)
```

For the most part, you can ignore most suppressed errors, but you might find it useful to enable the verbose flag (-v) to enable more detailed reports. Please consider reading the documentation for more information on this.

Although the documentation examples should be reasonably easy to follow, in order to practice your understanding of what is being reported for various synchronization bugs, you might want to consider trying to intentionally introduce various synchronization errors in the producer-consumer code and understanding the resulting `helgrind` report. For example, comment out in the `producer` function the line which acquires the `region_mutex`. What does the `helgrind` report tell you?

Warning: do not rely solely on this tool for writing correct code! Assistance from tools like `helgrind` should not be a substitute for carefully thinking through what your code is doing and what problems may arise. Although in this assignment the `helgrind` tool is probably sufficient for detecting most mistakes you could run into, it is not a "catch-all" solution for synchronization bugs and it is constantly being enhanced with more features. Once again, for the types of errors detected by `helgrind`, please read the **Documentation (http://valgrind.org/docs/manual/hg-manual.html)** carefully.

Submission

First of all, please do not manually create an a2 directory in your repo. An empty directory called 'a2' should be created for you automatically when you log into MarkUs and go on your a2 link. Next, you should see the newly-created 'a2' directory in your repository. It will include all of the starter code. Please make sure in advance that you can access your a2 directory, to avoid last-minute surprises.

All of your work should be in 4 files:

- safeStopSign.c
- safeStopSign.h
- safeTrafficLight.c
- safeTrafficLight.h

All submitted files should be in the a2 directory, not a subdirectory. Do not submit executables or object files!

Additionally, you may submit an `INFO.txt` file which contains a discussion of your implementation and any problems you encountered. If your program is partially working, this is a great place to let the TAs know what is and isn't working. This file is not required.

Finally, whether you work individually or in pairs with a partner, you **must** submit a `plagiarism.txt` file, with the following statement:
"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

Any missing code files or Makefile will result in a 0 on this assignment! Please reserve enough time before the deadline to ensure correct submission of your files. No remark requests will be addressed due to an incomplete or incorrect submission!

Again, make sure your code compiles without any errors or warnings. Your code will be tested on the teach.cs lab machines.
Code that does not compile will receive zero marks!

Marking scheme

We will be marking based on correctness (90%), and coding style (10%). Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly!
Once again: code that does not compile will receive 0 marks! More details on the marking scheme:

- stop sign: 30% (correctness and efficiency of the synchronization)
- traffic light: 60% (correctness and efficiency of the synchronization)
- Code style and organization: 10% - code design/organization (modularity, code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)
- **Negative deductions (please be careful about these!):**
    - Code does not compile -100% for *any* mistake, for example: missing source file necessary for building your code (including Makefile, provided source files, etc.), typos, any compilation error, etc.

- No `plagiarism.txt` file: -100% (we will assume that your code is plagiarized, if this file is missing) Please contact the instructor in this case.
- Warnings: -10%
- Extra output (other than the printf statement indicated in the comments): -20%
- Code placed in subdirectories: -20% (only place your code directly under your a2 directory)