Assignment 2: Synchronization

Due Oct 19 by 10pm **Points** 8

Overview

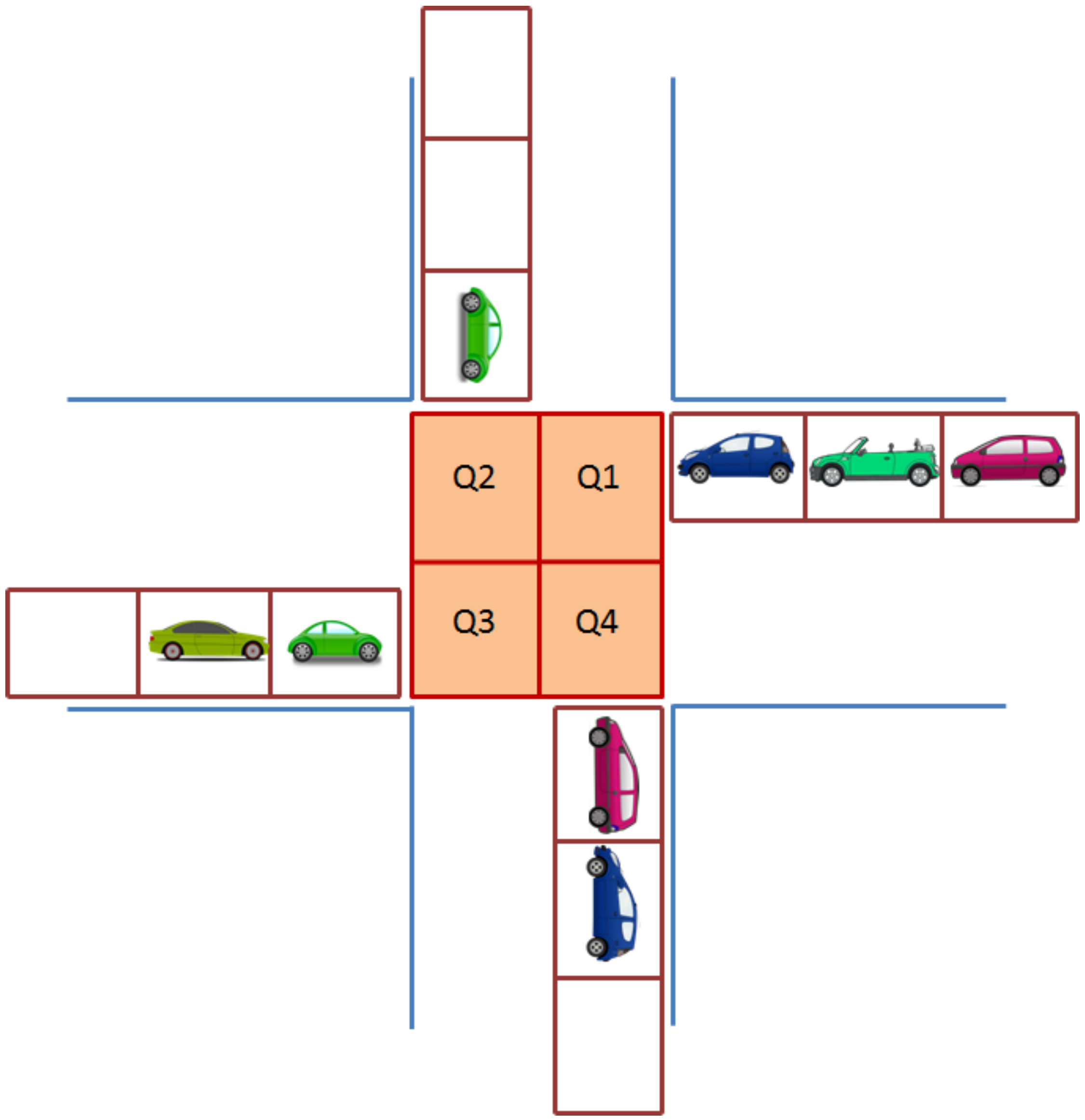
For this assignment, we're going back to the realm of user mode programming. In this assignment, you will work with synchronization primitives to implement a traffic monitoring and guidance system.

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 a traffic system that coordinates cars going through an intersection, by enforcing ordering between car arrivals in the same direction, avoiding potential crashes, and avoiding deadlocks.

Details

Consider the structure of the intersection, as shown in the figure below. There are four entrances/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 is represented by a fixed sized buffer. Exits from the intersection are represented by a simple linked list for each exit direction. The intersection itself is broken into quadrants, each represented by a lock.



Cars are represented by a 3-tuple (**id**, **in_dir**, **out_dir**) where **id** is a car's unique id, **in_dir** is the direction from which the car enters the intersection and **out_dir** is the direction the car leaves the intersection.

The traffic system enforces the policy for cars to pass through the intersection safely and in an orderly fashion. You will implement the traffic system using a monitor.

Policies:

1. Once a car approaches an intersection, it is placed in the corresponding lane. The car can only cross the intersection when all other cars already in the lane have crossed.

2. Once a car is the first in the lane, it can begin to cross the intersection. Before the car can enter the intersection it must guarantee that it has a completely clear path to its destination.

Implementation

The starter code can be found in /u/csc369h/fall/pub/a2/ on the teaching lab machines. This directory contains 3 items:

1. `a2-starter.tgz`- a tar file containing all files from a2-starter
2. `a2-starter` - a directory containing all the source code for the assignment
3. `student-tester` - an executable file that will only run on the Linux lab machines. You can either copy it to your local directory on the teach.cs machines or set your PATH variable to run it directly. Please do not commit this file to your repo.

Your task is to implement the methods for the monitor, given to you in the starter code. The monitor has 3 methods that you must fill:

```
/* Initialize all the monitor attributes */
void init_intersection();

/* Car approaches the intersection, must be added to the lane */
void *car_arrive(void *arg);

/* Car enters the intersection, must check for clear path */
void *car_cross(void *arg);
```

Additionally, the monitor contains the helper function below (which you must also fill in), and which will assist in making decisions about how cars cross the intersection.

```
/* given two directions returns the path a car would take through the intersection */
int *compute_path(enum direction in_dir, enum direction out_dir);
```

There are two threads per cardinal direction, one which executes

```
car_arrive()
```

and another which executes

```
car_cross()
```

You must use the starter code provided, which gives you detailed instructions on what you need to implement. Please make sure to implement all the parts indicated using detailed TODO comments.

The starter code contains a program (traffic.c) that serves as the entry point for the assignment. The traffic program takes the configuration of the approaching cars from a file, which contains a series of rows each corresponding to a car. Each row is formatted as follows:

```
id, in_dir, out_dir

ex.
1 0 2
2 1 2
3 1 3
4 0 1
```

For additional clarification check the definitions and comments in traffic.h, as well as the provided sample input file (schedule.txt).

You must ensure that there are no memory leaks in your code. **DO NOT free the out_cars array** because we will be inspecting its contents in the autotester, to verify some integrity constraints regarding the correctness of your implementation.

Testing

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 on how to use `helgrind` is as follows:

```
valgrind --tool=helgrind ./traffic schedule.txt
```

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.

Basic tester

We are also providing a self-tester that checks some basic properties of your code. The tester will check that all cars from a given lane enter the intersection in the right order (cars don't skip over others), by inspecting your output (see the comments from the starter code on what output format is expected). The tester will also check that all cars make it out of the intersection in the right quadrant, according to their out_dir. This is why you should not free the out_cars, and you should not modify the traffic code.

An example of how to run the student tester: $ ./student_tester schedule_file1 schedule_file2 etc.. The result of your run will be placed in a file called results.txt. If there are no errors in the result file, then your code passes some basic sanity checks. You should use more complex schedules than the one we gave you, and carefully-tailored schedules that capture various corner cases.

Once again, this tester is just a basic checker and should not preclude you from using valgrind to track down concurrency bugs, as well as thoroughly testing your code!

Submission

First of all, DO NOT MANUALLY CREATE A NEW DIRECTORY 'A2' IN YOUR REPOSITORY! 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. Please make sure in advance that you can access your A2 directory, to avoid last-minute surprises.

You will submit the `cars.c` file that contains your implementation, along with the files required to build your program (including the provided `traffic.h`, `Makefile`, and `traffic.c`, which you should not modify). All submitted files should be in the A2 directory, not a subdirectory. Do not submit executables or object files!

Additionally, you must submit an `INFO.txt` file, which contains as the first 3 lines the following:

- your name
- your UtorID(s)

- Optional: the **git commit hash** for your last submission. As a general rule, we will always take the last revision before the deadline (or after, if you decide to use grace tokens), so this is simply a sanity check for us that we did not miss a revision when we retrieve your code via MarkUs.
- **A paragraph titled "Discussion", which discusses whether starvation can happen when using this monitor. Describe your rationale.**

Aside from this, please feel free to describe in a separate paragraph(s), any problems you've encountered, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken (as long as they conform to the assignment specs), etc.

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."

A missing INFO.txt file will result in a 10% deduction (on top of an inherent penalty if we do not end up grading the revision you expect). **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:

- init_intersection: 10% (initializing the intersection structures correctly and completely.)
- car_arrive: 30% (correctly and *efficiently* implementing car_arrive - no big locks, cars arriving in waves, etc.)
- car_cross + compute_path: 40% (correctly and *efficiently* implementing car_cross)
- Starvation discussion: 10% (correct discussion and supporting your argument clearly, in detail)
- 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 plagiarised, if this file is missing)
- Missing or incorrect INFO.txt: -10%
- 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)