Project 3 (P3): Traffic Control

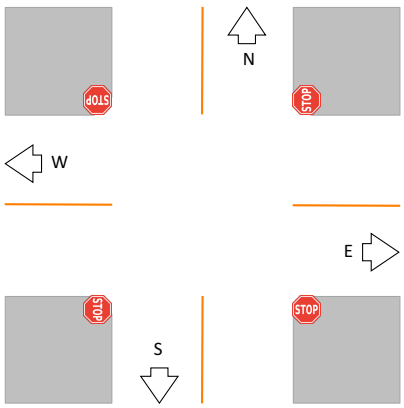
Instructor: Dr. **Probir Roy**

Due Time: 12PM, 11/20/2022

In this project, you will use `pthread lock` plus `semaphore` to implement a `traffic-control` system with stop signs at an intersection.

1 Stop Signs

We consider a typical intersection with stop signs as shown in the following picture: It has four directions



(`W(<)`, `E(>)`, `S(v)`, and `N(^)`). The road on each direction has only one lane with one stop sign.

In the following, we describe the traffic control policy for cars. In this system, each car is represented by one thread, which executes the subroutine `Car` when it arrives at the intersection:

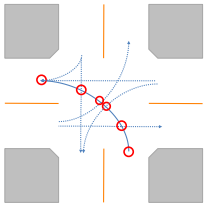
```
Car(directions dir) {
    ArriveIntersection(dir);
    CrossIntersection(dir);
    ExitIntersection(dir);
}
```

Data type `directions` is defined as

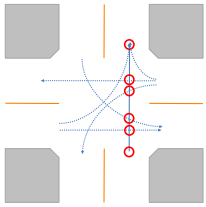
```
typedef struct _directions {
    char dir_original;
    char dir_target;
} directions;
```

where `dir_original` and `dir_target` show the original and targeted directions, respectively.

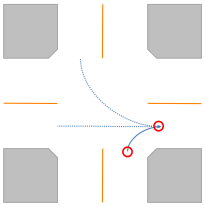
`ArriveIntersection` When a car arrives at the intersection, it will stop for 2 second. You could use `usleep` for the arrival time and the stop time.[1] If it finds another car in front, it has to wait until it is in front. Then it check if there is any car at other stop signs arriving earlier[2], it should wait until none of them is stuck at the intersection, but it doesn't need to wait until all of them finish crossing. Then it can either drive through, or turn left, or turn right (U-turn is not allowed) depending on the condition as shown in the following figures:



Turning left



Driving through



Turning right

In the above figure, the solid curve represents the path for this car, the dotted curves represent the paths with potential collision, and the circles represent potential collision locations.

`CrossIntersection` We assume that it takes fixed time periods (Δ_L, Δ_S, and Δ_R for turning left, going straight, and turning right, respectively) to cross the intersection and print out a debug message. You could use the `Spin` function [3] to simulate the crossing.

`ExitIntersection` It is called to indicate that the caller has finished crossing the intersection and should take steps to let additional cars cross the intersection.

2 Testing

Fix all the time periods described above as follows: ($\Delta_L = 5$s, $\Delta_S = 4$s, $\Delta_R = 3$s). The simulation will run until all cars finish crossing. In the main thread, you need to create threads, one for each car. The arrival pattern in the simulator is described as follows:

```
cid arrival_time dir_original dir_target
1   1.1          ^            ^
2   2.2          ^            ^
3   3.3          ^            <
4   4.4          v            v
5   5.5          v            >
6   6.6          ^            ^
7   7.7          >            ^
8   8.8          <            ^
```

Associated with every car is a unique id (`cid`) as the car number, the arrival time (`arrival_time`) for the given car, and the original `dir_original` and targeted `dir_target` directions. For example, the first line shows Car 0 arriving at Time 1.1 which originally goes northward and continue to go northward. Make sure that your simulation outputs information that clearly shows that your solution works. The

[1] `http://linux.die.net/man/3/usleep`

[2] Each car in the front is only aware of the order of car arrival at other stop signs, not the exact time.

[3] Download this: `http://pages.cs.wisc.edu/~remzi/OSTEP/Code/code.intro.tgz`. You can find the `Spin` function in `common.h`.

message should indicate car number and both original and targeted directions. In particular, messages should be printed at the following times:

- Whenever a car arrives at an intersection;
- Whenever a car is crossing the intersection;
- Whenever a car exits from the intersection.

This is the right output for the arrival pattern given above:

```
Time  1.1: Car 1 (^ ^) arriving
Time  2.2: Car 2 (^ ^) arriving
Time  3.1: Car 1 (^ ^)          crossing
Time  3.3: Car 3 (^ <) arriving
Time  4.2: Car 2 (^ ^)          crossing
Time  4.4: Car 4 (v v) arriving
Time  5.3: Car 3 (^ <)          crossing
Time  5.5: Car 5 (v >) arriving
Time  6.6: Car 6 (^ ^) arriving
Time  7.1: Car 1 (^ ^)                   exiting
Time  7.7: Car 7 (> ^) arriving
Time  8.2: Car 2 (^ ^)                   exiting
Time  8.8: Car 8 (< ^) arriving
Time 10.3: Car 3 (^ <)                   exiting
Time 10.3: Car 4 (v v)          crossing
Time 10.3: Car 6 (^ ^)          crossing
Time 14.3: Car 4 (v v)                   exiting
Time 14.3: Car 6 (^ ^)                   exiting
Time 14.3: Car 7 (> ^)          crossing
Time 19.3: Car 7 (> ^)                   exiting
Time 19.3: Car 8 (< ^)          crossing
Time 19.3: Car 5 (v >)          crossing
Time 22.3: Car 8 (< ^)                   exiting
Time 24.3: Car 5 (v >)                   exiting
```

3 Coding and Hints

You are recommended to use threading programming in C/C++. The programming should be tested in the Ubuntu environment in your P1 and P2, not in the `xv6`. Your executable file should be named as `tc`.

Use this sample code: `https://gist.github.com/tausen/4261887` as the basic framework. Make sure to identify the key resources each car needs to acquire, which you need to model with mutex or semaphore. For each red circle in previous figures, you need to create a mutex or semaphore.

For the head-of-line lock, when a car starts crossing, it should release the head-of-line lock so that the other cars behind can start crossing ASAP. For the other mutexes or semaphores, the car should release them after finishing crossing, which is simplified. If a flow of cars are with the same original direction, they should be allowed to go back-to-back if all other conditions are met. In this case, you could follow the example of readers' operations in the readers-writers problem to synchronize them.

4 Submission

You are going to report the following things:

(a) Describe in details how you implemented it in the report.

(b) Write a detailed report with the screenshots of the testing results. Each screenshot should include your username and the current time, which show that you did it by yourself. If your output is different from the expected one, provide a reason for the cause.

(c) Specify the contribution made by each member if you work as a group.

The report should be written in a ".docx", ".doc", or ".pdf" format. All source codes should be well formatted with good comments. Submit the report and the source code to Project P3 on Canvas. Any compression file format such as `.zip` is not permitted.