SCHOOL OF COMPUTING, QUEEN'S UNIVERSITY

CISC324: Operating Systems

Lab 4 [due Feb 28]

February 9, 2020

In this Lab you are asked to create a simulation of car traffic. As a starting point, you are given an initial code. This code was created by making the following modifications to the ReaderWriter code from Lab 3. (You should easily be able to make such changes yourself. The changes have been made for you to save you time.)

- Rename “Reader” to be “Car”. Get rid of “Writer”.
- Remove all synchronization. (No semaphores are used in the given code.)
- Change the output messages to refer to cars and intersections, instead of referring to reading and writing.

The given code contains no synchronization between cars. Each car does the following repeatedly (you can limit this into a certain number of rounds): starting in Barriefield, cross the causeway westbound into Kingston, then enters a petrol station to fill the car with petrol, then leaves the petrol station and cross the causeway eastbound back into Barriefield.

A file named `TimeSim.java` is provided because Java’s built-in sleep() method results in inexact timing. For example, if two threads execute a sleep instruction at the same time, the thread that executes `sleep(20)` may wake up before the thread that executes `sleep(10)`. File `TimeSim.java` contains an accurate time simulation written by Professor Blostein. The simulated time advances after all threads have reached a `sleep()` or `acquire()`. The implementation of timeSim needs an exact count of the number of threads. Therefore, every thread you create has to begin by calling Synch.timeSim.threadStart() and has to end by calling Synch.timeSim.threadEnd(). The car thread code in `Car.java` shows how to do this.

Task to do

Add synchronization (using semaphores) to simulate the following situation. The causeway is under construction, so only one lane is open. A traffic light system has been installed. This traffic light system repeatedly goes through the following cycle:

- The light is green for eastbound traffic for some time T.
- Then the light is red in both directions for some time Q.
- Then the light is green for westbound traffic for time T.
- Then the light is red in both directions for time Q.

You may choose values for T and Q. As you can see from Car.java, the simulated time to cross the bridge is 100, so it would make sense to choose Q=100. This allows the bridge to clear of eastbound traffic before westbound traffic starts, and vice versa. The value of T can be chosen pretty freely. In real life, T would probably be longer than Q. You need to decide the following things before you start coding.

- How are you going to represent the traffic light? Probably you need some variable(s) to keep track of the different states of the traffic light. In addition, do you need another thread to help set these variables? Do you need mutex protection to access these variables?

- How are you going to make the Car threads wait, when the traffic light is red? You should not use busy waiting code. An example of busy waiting would be "while (eastBoundLight = red) <do nothing>". This kind of coding is bad to use because it wastes a lot of CPU time. The car thread keeps testing and testing the eastBoundLight variable. Instead, you should use semaphores. If the eastbound car finds that eastBoundLight is red, then it should acquire some semaphore. Whatever thread is responsible for eventually setting the eastBoundLight to green should Release that semaphore. If 10 eastbound cars are waiting, then the semaphore Release should be sent 10 times. To achieve this, it might be necessary to have an EastBoundCarCount (an integer). Remember that you have to use a mutex-type semaphore whenever several threads access a shared variable. For example, you do not want to allow two Car threads to simultaneously try to increment EastBoundCarCount.

What should be submitted

- Place your codes (`MainMethod, Car, ...`) in one folder.

- Your codes should be well commented. You must add comments to explain why you have used a particular variable and what each piece of code performs.

- Save the output of one execution scenario (e.g., for 10 cars and 3 rounds) in a file called ReadMe.txt and mention on the top of the file, which configuration you have used. Also give an explanation why you think that your synchronization is working properly.

- Place everything inside one folder and compress the file using zip.

- Submit your lab over OnQ.

- A well annotated (or commented) code with a correct output will get 100% of the mark. A correct output with no annotation gets 75% of the mark. For outputs that contain errors, but the annotation claims that there are no errors (or there is no annotation) gets 50%.