CS 475 Fall 2017
Assignment 2 PART B
Concurrent Programming
DUE 11:59 PM Wednesday, October 18

Instructions:

- You can work on PART B in a teams of two students.
- The course late submission policy applies to PART B, i.e., you can use up to two "slip" days for this assignment, as long as you have not used up all 5 slip days available to you for the semester.
- Please upload your solutions to PART B to Blackboard. If you are working in a team, only one member of the team should submit the solution to PART B.

The goal of this exercise to give you some experience writing concurrent programs using the **Pthreads** multi-threaded programming library.

You have been hired by MetroTrans to synchronize traffic over a one-lane bridge on a public highway. Traffic may only cross the bridge in one direction at a time. Moreover, the bridge is really old and hasn't been repaired for lack of funds, so MetroTrans has decided to impose the restriction that there should never be more than 5 vehicles on the bridge at any given time. (If this limit is exceeded the bridge will collapse).

In this system, each vehicle is represented by one thread, which executes the procedure `OneVehicle` when it arrives at the bridge:

```
OneVehicle(int direc, int time_to_cross) {
     ArriveBridge(vehicle_id, direc);
     CrossBridge(vehicle_id,direc,time_to_cross);
     ExitBridge(vehicle_id,direc);
}
```

In the code above, `vehicle_id` is an integer which uniquely identifies each vehicle. (The vehicles arriving at the bridge should be assigned `vehicle_ids` 1,2,3... and so on.) `direc` is either 0 corresponding to east or 1 corresponding to west; it gives the direction in which the vehicle will cross the bridge. `time_to_cross` is the time it takes a vehicle to cross the bridge - assume that every vehicle takes 4 seconds to cross the bridge.

For this assignment, you have to implement **two different versions** of `OneVehicle` as discussed below. Both versions need to ensure that the following constraint is met:

- There must never be more than 5 vehicles on the bridge at any time. If the bridge already has 5 vehicles on it, additional vehicles arriving at the bridge must wait to enter the bridge in **two (FIFO) queues** on opposite sides of the bridge.

The two versions of `OneVehicle` differ in the traffic control policy that is implemented:

1. In the first (default) version of the policy, when a vehicle exits the bridge, a waiting vehicle that is traveling in the same direction as the exiting vehicle is permitted to enter the bridge in preference to a vehicle traveling in the opposite direction. In other words, a waiting vehicle traveling in the opposite direction may enter the bridge only if there is no waiting vehicle traveling in the same direction as the exiting vehicle.

2. While the policy above is acceptable when the traffic volume is low, it can lead to unfairness at rush hour. Implement a **fair** traffic control policy suitable for rush hour that imposes a limit (say 6) on the number of vehicles that can cross the bridge in one direction while vehicles traveling in the opposite direction are waiting to get on the bridge. NOTE: if there are no vehicles waiting to get on the bridge in the opposite direction, your policy should not switch directions. In other words, your policy should be smart enough to provide fairness if there is traffic in both directions but should never force a vehicle to wait to get on the bridge if there is no traffic in the opposite direction.

Write the procedures `ArriveBridge`, `Cross_Bridge` and `ExitBridge`, using mutex locks and condition variables for synchronization. (Note: you will need two different versions of `ArriveBridge` and `ExitBridge` corresponding to the two versions of `OneVehicle`.)

1. The `ArriveBridge` procedure must not return until it is safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses).

2. The `CrossBridge` procedure simply sleeps for `time_to_cross` seconds and print out a status message.

3. `ExitBridge` is called to indicate that the caller has finished crossing the bridge; `ExitBridge` should take steps to let additional cars cross the bridge.

4. In addition, `ExitBridge` should update a shared variable `departure_index`, which keeps track of the order in which the cars leave the bridge, i.e., the first car to leave the bridge has `departure_index` 1, the second car has `departure_index` 2, and so on. The `ExitBridge` procedure should also print out the `departure_index` for that vehicle.

NOTES:

- Your solution must not employ busy waiting.
- All accesses to shared variables must occur inside a critical section
- Occasionally, a vehicle (thread) may overtake another vehicle traveling in the same direction on the bridge. Do not worry about this. You do not have to ensure that vehicles leave the bridge in the same order as they entered it.
- The debug message printed out in `CrossBridge` should provide a snapshot of the bridge and the queues on the two ends of the bridge. It is probably also a good idea to print out debug messages in `ArriveBridge` and `ExitBridge`.

- Condition variables and locks are discussed in Chapters 25-30 of the online textbook OS: Three Easy Pieces. See `http://pages.cs.wisc.edu/%7Eremzi/OSTEP/`
- The Pthreads API and reference pages for the Pthreads library routines can be found at `https://computing.llnl.gov/tutorials/pthreads/`

In this assignment, you have to run your programs for the three vehicle arrival schedules given below:

1. 5 : DELAY(10) : 5 : DELAY(10) : 5 : DELAY(10) : 5
2. 10 : DELAY(10) : 10
3. 20

Here the numbers indicate the number of vehicles arriving simultaneously at the bridge, while the numbers in parentheses indicate the delay before the next arrival(s). For example, under schedule (1) 5 vehicles arrive simultaneously at the bridge at the start of the experiment, five more vehicles arrive simultaneously 10 seconds after the arrival of the first five vehicles and so on. In each of the three schedules, twenty vehicles arrive at the bridge during the course of the experiment. Note that vehicles arriving simultaneously does not imply that they are all traveling in the same direction. For each vehicle the direction it is travellng in is specified by the input arguments to the driver program discussed below.

To help you get started, we have provided you with a driver program that creates threads corresponding to vehicles arriving at the bridge and invokes the `OneVehicle` function for each thread. The command line arguments for the driver program are used to specify the vehicle arrival schedule as well as the direction each vehicle is traveling.

Start by copying the file `assign2.tar` to the protected directory in which you plan to do your work. Then do the following:

- Type the command `tar xvf assign2.tar` to expand the tarfile.
- The file `assign2.c` contains the driver routines we have provided as well as skeletons for the routines that you have to implement. We have also provided you with a makefile. Feel free to modify any of the code in `assign2.c`, although there should be no need to modify the driver routine which creates the threads corresponding to the vehicles being simulated based on command line arguments.
- Before you start the assignment, make a copy of `assign2.c` and rename it as `assign2-rush.c`. As indicated by the name of the file, `assign2-rush.c` should contain the code for the version of `OneVehicle` and associated functions corresponding to the traffic control policy used during rush hour, whereas `assign2.c` should contain the implementation of the default traffic control policy used at other (non-rush hour) times.
- Type your team member names in the header comment at the top of `assign2-rush.c` and `assign2.c`.

Evaluation

Your score will be computed out of a maximum of 100 points based on the following distribution:

80 Correctness: Your programs will be evaluated on correctness by both executing it on `zeus.vse.gmu.edu` and a thorough examination of the code. Please be aware that concurrent programs have non-deterministic behavior and that seemingly correct output for some tests is no guarantee of program correctness. Nevertheless, it is important that you verify that your code appears to be executing correctly at least for the three arrival schedules specified earlier in this handout. The output from your program will be useful in determining the order in which vehicles are able to cross the bridge while adhering to the policies specified in this handout.

Each policy (default and rush hour) is worth 40 points each. Note that grade for program correctness will be marked down by 50% if your program does not meet the specified requirements (**no busy waiting, FIFO queues on both sides of bridge**) even if the program enforces the traffic control policy correctly. Your grade will be marked down by 25% if any shared variables are accessed **outside a critical section.**

20 Design Document: Please submit a design document that describes your solutions for both the traffic control policies. You should describe (i) the data structures being used in your code. This should include a discussion of the synchronization mechanisms you are using (ii) a high-level overview of the the overall logic underlying your solutions including a discussion of why you believe your solutions are correct. The design document should not exceed 3 pages in length.

Handin Instructions

- Make sure you have included your name and login ID in the header comment of `assign2-rush.c` and `assign2.c`. If you are working in a group of two, please include both names. Only one member of the group should submit the assignment.
- Make sure that all the code you have written is included in files `assign2.c` and `assign2-rush.c`
- Create a `zip` or `tar` archive containing the files `assign2.c`, `assign2-rush.c` and your design document file. Name the archive file `loginId-assign2.tar` or `loginId-assign2.zip` as appropriate and upload it to Blackboard.
- If you discover a mistake and want to resubmit your assignment, Blackboard will permit you to do so until the submission deadline.