

**GEORGE MASON UNIVERSITY**  
**Computer Science Department**

**CS 475 – Concurrent and Distributed Software Systems**

Spring 2003

Assignment 2 – Multithreaded Programming

DUE DATE – March 10

**Process Synchronization** The goal of this exercise is to give you some experience writing concurrent programs.

You have been hired by MetroTrans to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each vehicle is represented by one thread, which executes the procedure `OneVehicle` when it arrives at the bridge:

```
OneVehicle(int vehicle_id,int direc, int time_to_cross)
{
    ArriveBridge(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 or 1; 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.

Implement a “fair” traffic control policy that imposes a limit (say 5) on the number of vehicles that can cross the bridge in one direction, while vehicles travelling 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`, `CrossBridge` and `ExitBridge`, using **mutex locks** and **condition variables** for synchronization<sup>1</sup>. 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). The `CrossBridge` procedure should just delay for `time_to_cross` seconds and print out a debug message. `ExitBridge` is called to indicate that the caller has finished crossing the bridge; `ExitBridge` should take steps to let additional cars cross the bridge. 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.

---

<sup>1</sup>If you use Java for the assignment, use the equivalent thread synchronization facilities

## NOTES:

- **Your solution must not employ busy waiting.**
- Your solution must use two queues on either end of the bridge where vehicles wait until it is safe for them to enter the bridge. In terms of threads, this means that you need two condition variables (or Java objects) where a thread is suspended if it cannot enter the `CrossBridge` procedure.
- For full credit, your solution **must not use** the `pthread_cond_broadcast()` function (or if you're using Java, the `notify_all()` method), i.e. your solution must only use `pthread_cond_signal()` or `notify()` for signalling any threads blocked on a condition variable. NOTE: invoking `pthread_cond_signal()` or `notify()` repeatedly in a loop is equivalent to `pthread_cond_broadcast()` and `notify_all()`
- Occasionally, a vehicle (thread) may overtake another vehicle travelling 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`.

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

- (i) 5 : DELAY(10) : 5 : DELAY(10) : 5 : DELAY(10) : 5 : DELAY(10) : 5 : DELAY(10) : 5
- (ii) 10 : DELAY(10) : 10 : DELAY(10): 10
- (iii) 30

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 (i) 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, thirty 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. Assume that the probability that a vehicle is traveling in direction "0" is 0.7

---

## NOTES:

1. You can use either Java or any multithreaded programming library for doing this assignment. Both Java and the Solaris thread library are available on machines in the IT&E lab.
2. On Solaris, information on how a thread library call is to be invoked is provided by the corresponding man page, e.g., to see how `pthread_create` is invoked type `man pthread_create`. Additional documentation on multi-threaded programming using the Solaris thread library is also available on the Solaris documentation web site (see link on class web page).
3. If you are using the Solaris library, use the system call `drand48()` to determine the direction of the vehicle (according to the probability distribution specified above). The random number stream should be initialized by calling `srand48()` with the desired `seed`. If you are using Java for this assignment, use the methods in class `java.util.Random` to create and use the random number stream that determines the direction of a vehicle.

4. You have to submit the following (i) a write up containing a description of the data structures used by your solution (ii) pseudo-code for your solution (iii) source code for your implementation and the output of runs for the 3 schedules with `seed = 9`. If you cannot make it to class to submit a hard copy, submit a softcopy via email and a hard copy in the subsequent class.
  5. Your code should be well documented and structured, i.e., it should have meaningful comments, meaningful variable names, etc.
-