**CSC 8980**

**Distributed Systems Fall 2022**

Homework #2

1. Imagine a very long bridge across the Mississippi River. A car will take upwards of 15 minutes to cross this bridge. Due to construction, the bridge has been reduced to a single lane that has to be shared by traffic in both the west-east and the east-west direction. It is obviously not possible to allow traffic in both directions simultaneously, so a special traffic control mechanism is installed with the following rules:
   * An arriving car will have to wait if the bridge is currently occupied by one or more cars moving in the opposite direction.
   * Multiple cars are allowed to cross the bridge in the same direction (otherwise there would be little bridge utilization and arriving cars would have to wait a long time).
   * In order to avoid starvation, the entry of cars onto the bridge in one direction must be stopped after a batch of k cars has entered the bridge to allow traffic in the opposite direction if there are any cars waiting.
   * If there are no cars, the bridge is open in both directions and the first arriving car will determine the direction of traffic.

Viewing each car as a process that is traveling in West-East (WE) or East-West (EW) direction, develop a MONITOR that implements the rules listed above. You MONITOR must use the monitor procedures **Enter\_WE(), Enter\_EW(), Exit\_WE(), Exit\_EW(),** to be executed when a car enters and exits the bridge.

Your solution must show all the necessary MONITOR variables and the condition variables and must not unnecessarily restrict vehicles to cross the bridge, must be deadlock free, and must not starve traffic in any direction.

**ANS:** *(Assumption – time considered in minutes.)*

The monitor goes as follows –

1. Initialize the following –

* QueueEW: *cars running from East to West*
* QueueWE: *cars running from West to East*
* BridgeCars: *an array* *containing #cars currently on the bridge*
* k: *maximum #cars that can be on the bridge at a time*
* trafficDirection: *traffic direction set by the first car arriving in either of the queues.*

1. While (True):
   1. whichQueue = Listen(QueueEW, QueueWE)
   2. !isEmpty(QueueEW)?trafficDirection = “EW”:trafficDirection = “WE”
   3. If !isEmpty(whichQueue):
      1. While(BridgeCars.length<=k):
         1. currentCar = whichQueue.pop()
         2. BridgeCars.add(currentCar)
      2. For car in range(BridgeCars):
         1. travelStartTime.append(startTimer())
         2. trafficDirection==”EW” ? **Enter\_EW(car)** : **Enter\_WE(car)**
         3. If ((currentTime()–travelStartTime[BridgeCars.indexOf(car)]) == 15):
            1. trafficDirection==”EW” ? **Exit\_EW(car)** : **Exit\_WE(car)**
            2. BridgeCars.remove(BridgeCars.indexOf(car))
            3. travelStartTime[BridgeCars.indexOf(car)]
   4. During steps a.) and c.), if the other Queue not under process gets a car, the cars in the other queue are put on hold till the BridgeCars get empty. Means till the current traffic on the bridge doesn’t clear, the cars wait in the queue.
2. For a semaphore *s*, we define the following:

*init[s]::=* initial value of *s*

*start\_P[s] ::=* the number of times P(s) has been started

*end\_P[s] ::=* the number of times P(s) has been completed

*end\_V[s] ::=* the number of times V(s) has been completed

A useful invariant is: *end\_P[s] = min(start\_P[s], init[s] + end\_V[s])*

Now consider the abstract version of the bounded buffer problem: empty, full, mutex: semaphore = (N, 0, 1);

Producer:

while(1) {

P(empty);

get buffer from freelist;

…. produce

put buffer on fullist;

V(full)

}

Consumer: while(1) {

P(full);

get buffer from fullist;

…. produce

put buffer on freelist; V(empty)

}

Prove that 0 ≤ end\_P[empty] – end\_P[full] ≤ N

1. Proof the correctness or give a counter-example for each of the following statements. You must state whether the statement is true or false and then show your arguments.
2. deadlock  cycle
3. cycle  deadlock
4. expedient & knot  deadlock
5. deadlock  knot

**ANS:**

* *deadlock -> cycle*
* *cycle -> deadlock*
* *expedient & knot -> deadlock*
* *deadlock -> knot*

1. Consider a system with N blocks of storage, each of which holds one unit of information (e.g. an integer, character, or employee record). Initially, these blocks are empty and are linked onto a list called *freelist*. Three processes communicate using shared memory in the following manner:

Shared Variables: freelist, list-1, list-2: block (where block is some data type to hold items)

PROCESS #1

var b: pointer to type block; while (1)

{

b:= unlink(freelist); produce\_information\_in\_block(b); link(b, list1);

}

*to do: understand the job of this*

PROCESS #2

var x,y: pointer to type block; while (1)

{

x:=unlink(list-1); y:=unlink(freelist);

use\_block\_x\_to\_produce info\_in\_y(x, y); link(x, freelist);

link(y, list-2);

}

*to do: understand the job of this*

PROCESS #3

var c: pointer to type block; while(1)

{

c:=unlink(list-2); consume\_information\_in\_block(c); link(c, freelist);

}

*To do: understand the job of this*

Use simple semaphores to implement the necessary mutual exclusion and synchronization. **The solution must be deadlock-free and concurrency should not be unnecessarily restricted.**