CSC 8980

Distributed Systems

Fall 2022

Homework #2 (Exam Preparation)

Due date: September 22th

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 and 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.

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:                          Consumer:
    while(1) {                         while(1) {
      P(empty);                          P(full);
      get buffer from freelist;          get buffer from fullist;
      …. produce                         …. produce
      put buffer on fullist;             put buffer on freelist;
      V(full)                            V(empty)
    }                                  }
```

   Prove that 0 ≤ end_P[empty] – end_P[full] ≤ N

3. 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.

   a. deadlock ➔ cycle
   b. cycle ➔ deadlock
   c. expedient & knot ➔ deadlock
   d. deadlock ➔ knot

4. 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);
}
```

```
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);
}
```

```
PROCESS #3
var c: pointer to type block;
while(1)
{
  c:=unlink(list-2);
  consume_information_in_block(c);
  link(c, freelist);
}
```

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