EE 361C Multicore Computing Spring 2024

Assignment 3

Due: February 27

Instructor: Professor Vijay K. Garg (email: garg@ece.utexas.edu)

1. **(20 points)** For each of the histories below, state whether it is (a) sequentially consistent, (b) linearizable. Justify your answer. All variables are initially zero.

```
Concurrent History H1

P1           [ read(x) returns   1]
P2       [ write(x,1)                   ]    [ read(x)  returns 2]
P3             [write(x,2)      ]
```

This is linearizable as shown in the linearization points, thus it's also sequentially consistent

```
Concurrent History H2

P1           [ read(x) returns   1]
P2       [ write(x,1)                   ]    [ read(x)  returns 1]
P3             [write(x,2)      ]
```

This is also linearizable as shown in the linearization points, thus it's also sequentially consistent

```
Concurrent History H3

P1           [ read(x) returns 1]
P2       [ write(x,1) ]                        [ read(x)  returns 1]
P3                        [write(x,2)      ]
```

this is sequentially consistent because an equivalent history that preserves process order can be legal if the write(2) comes after the last read(1), but it is not linearizable because doing so violates <_H

2. **(10 points)** Consider the following concurrent program.

```
Initially a, b and c are 0.
P1:  a:=1 ; print(b) ; print(c);
P2:  b:=1 ; print(a) ; print(c);
P3:  c:=1 ; print(a) ; print(b);
```

Which of the outputs are sequentially consistent. Justify your answer.

(a) P1 outputs 11, P2 outputs 01 and P3 outputs 11.

as long as p2 prints(a) before p1 sets it to one, and then all 3 processes do their first operation before any others print, this is possible wo violating process order

(b) P1 outputs 00, P2 outputs 11 and P3 outputs 01.

for p1 to output 00, it must complete before p3 does its first task. yet p3 print(a) prints 0, so p1 could not have set a to 1 prior to p3 setting c to 1. thus this can't occur wo violating process order

3. **(70 points, programming)** (a, 40 points) Implement Lock-based and Lock-Free unbounded queues of `Integers`. For the lock based implementation, use different locks for `enq` and `deq` operations. For the variable `count` use `AtomicInteger`. For the lock-free implementation, use Michael and Scott's algorithm as explained in the class. The *deq* operation should return *null* if the queue is empty.

(b, 30 points) Implement Lock-Free stack of `Integer`. You should provide `push(Integer x)` and `Integer pop()`. The `pop` operation should throw an exception called `EmptyStack` if the stack is empty.
For both the data structures use a list based implementation (rather than an array based implementation).