EE 382C Multicore Computing Homework 3

Mitchell, Olivia ozm59

Molter, Matthew mm58286

November 9, 2020

1 Problem 1

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

1.1 H1

```
Concurrent History H1

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

1.1.1 Equivalent order

This can be written as

H1 = $P_2.write(x,1), P_1.read(x), P_3.write(x,2), P_3.ok(), P_1.ok(1), P_2.ok(), P_2.read(x), P_2.ok(2)$

which is equivalent to

H1 = $P_2.write(x,1), P_2.ok(), P_1.read(x), P1.ok(1), P_3.write(x,2), P_3.ok(), P_2.read(x), P_2.ok(2)$

1.1.2 Solution part a - sequentially consistent?

Yes, this is sequentially consistent, as the above equivalency is legal (consistent with sequential history of each object), and satisfies process order (no invocations on same process before response of previous operation).

1.1.3 Solution part b - linearizable?

Yes, this is linearizable, as the previously shown equivalent history also satisfies the preserving $<_H$ requirement.

1.2 H2

```
Concurrent History H2

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

```
P3              [write(x,2)      ]
```

1.2.1 Equivalent order

This can be written as

H1 = $P_2.write(x,1), P_1.read(x), P_3.write(x,2), P_3.ok(), P_1.ok(1), P_2.ok(), P_2.read(x), P_2.ok(1)$

which is equivalent to

H1 = $P_2.write(x,1), P_2.ok(), P_1.read(x), P_1.ok(1), P_2.read(x), P_2.ok(1), P_3.write(x,2), P_3.ok()$

1.2.2 Solution part a - sequentially consistent?

Yes, this is sequentially consistent. Again, the above equivalency is legal and satisfies process order.

1.2.3 Solution part b - linearizable?

This is not linearizable. In order to have a sequential history that is legal, we must violate the $<_H$ requirement by reading on P_2 before P_3 has finished writing. However, per the original history, P_3 receives the $ok()$ for its write before P_2 invokes its own read, so the above equivalency violates $<_H$.

1.3 H3

```
Concurrent History H3

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

1.3.1 Equivalent order

This can be written as

H1 = $P_2.write(x,1), P_1.read(x), P_2.ok(), P_3.write(x,2), P_1.ok(1), P_3.ok(), P_2.read(x), P_2.ok(1)$

which is equivalent to

H1 = $P_2.write(x,1), P_2.ok(), P_1.read(x), P_1.ok(1), P_2.read(x), P_2.ok(1), P_3.write(x,2), P_3.ok()$

1.3.2 Solution part a - sequentially consistent?

Yes, this is sequentially consistent. Again, the above equivalency is legal and satisfies process order.

1.3.3 Solution part b - linearizable?

This is not linearizable. In order to have a sequential history that is legal, we must violate the $<_H$ requirement by reading on P_2 before P_3 has finished writing. However, per the original history, P_3 receives the $ok()$ for its write before P_2 invokes its own read, so the above equivalency violates $<_H$.

2 Problem 2

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

2.1 Solution 2a

This is sequential consistent since this order is legal and satisfies the process order:
$P_2.write(b,1), P_2.ok(), P_2.print(a), P_2.ok(0), P_1.write(a,1), P_1.ok(), P_3.write(c,1), P_3.ok(), etc$ (any order of the remaining prints past here would return 1)

2.2 Solution 2b

This is not sequentially constant since there is no order that and keeps process order.

For P_1 to print cleanly, these orderings of events must occur:
$P_1.write(a,1) -> P_1.ok() -> P_1.print(b)$
$P_1.print(b) -> P_2.write(b,1)$
$P_1.print(b) -> P_1.ok() -> P_1.print(c)$
$P_1.print(c) -> P_3.write(c,1)$

For P_2 to print cleanly, these orderings of events must occur:
$P_2.write(b,1) -> P_2.ok() -> P_2.print(a)$
$P_1.write(a,1) -> P_2.print(a)$
$P_2.print(a) -> P_2.ok(1) -> P_2.print(c)$
$P_3.write(c,1) -> P_2.print(c)$

For P_3 to print cleanly, these orderings of events must occur:
$P_3.write(c,1) -> P_3.ok() -> P_3.print(a)$
$P_3.print(a), P_3 -> P_1.write(a,1)$
$P_3.print(a) -> P_3.ok(0) -> P_3.print(b)$
$P_2.write(b,1) -> P_3.print(b)$

However, there is no combination of all of these events that maintains these ordering (P_3 and P_1 come into conflict) thus this is not sequentially consistent.

3 Problem 3

3.1 part a - queues

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.

3.2 part b - stacks

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).