ECE/CS 5510 Multiprocessor Programming
Homework 3

Mincheol Sung

October 5, 2018

Problem 1.
No.
The possible case is the writer is invoked, the line #5 and #6 can be reordered by compiler unless v is volatile. However, v is defined as volatile, this can not happen. In java, volatile variables cannot be reordered. Volatile variable is also synchronized in each cpu's memory cache that all threads will see same value of it.

Problem 2.
No. To be a sequential history, the method calls of different threads do not interleave.

Problem 3.
3-1:
p1: $x = 1$;
p1: print(y,z);
p2: $y = 1$;
p2: print(x,z);
p3: $z = 1$;
p3: print(x,y);

3-2:
P1: $x = 1$;
P1: print(y,z);
P2: $y = 1$;
P3: $z = 1$;
P2: print(x,z);
P3: print(x,y);

3-3:
This is illegal. The last two character printed must be 11 since the last print method is called after all variables are 1 based on sequential consistency.

Problem 4.
No. Even though memory is quiescently consistent, no necessary the individual registers are quiescently consistent.
Suppose x and y are the registers.
Time $\longrightarrow$
P1:$< \qquad\qquad write(x,1) \qquad\qquad >$
P2:$< write(y,2) > -- < read(y,3) >$
H is quiescently consistent because there is no quiescent period.
However, $H|y$ is not quiescently consistent.

Problem 5.
(a)
Suppose x and y are the registers.
Time $\longrightarrow$
P1:$< \qquad\qquad write(x,1) \qquad\qquad >$
P2:$< write(y,2) > -- < read(y,3) >$
This is quiescently consistent but not sequentially consistent.
(b)
Time $\longrightarrow$
T1:$-- < enq(a) >$
T2:$- - - - - - - - < enq(b) > - < deq() \to b >$
This is sequentially consistent but not quiescent consistent.
Quiescent consistency requires 0 enqueued before 1, but sequentially consistency doesn't

Problem 6.
Yes.
Time $\longrightarrow$
A:$< - - | - s.push(x) - - - - >< -| - s.pop() - - >$
B:$-- < - - s.push(y) - |- >< - - - - - - - - - s.pop() - - - - - - - - - - - - - -$

The legal sequential history can be
B: s.push(y)
B: s:void
A: s.push(x)
A: s:void
A: s.pop()
A: s:y

Problem 7.
Linearizable: yes
A: $< - - r.enq(x) - |- > - - - - < - - -r.deq(y) - - - |- >$
B: $- - - - - - - - - - - - < - - | - -r.enq(y) - - >$
Sequential consistent: Yes

Problem 8.
No. The history is not linearizable. element y is enqueued once that y can be popped once.

Problem 9.
In every sequential history S equivalent to H, if $S|x$ is not a legal sequential history, S itself is not a legal sequential history.

Problem 10.
1. Yes. If $Q.enq(b)$ happens before $Q.enq(a)$, $H|Q$ is a legal history.
2. Yes. If $Q'.enq(b')$ happens before $Q'.enq(a')$ $H|Q'$ is a legal history.
3. No. H is not a legal history. As the entire history, $Q.deq() \to b$ is illegal or $Q'.deq() \to b'$ is illegal.

Problem 11.
No. Based on the processor 1's observation, W(a,1) happens before W(b,1)
However, based on the processor 3's observation, W(b,1) must happens before W(a,1).

Problem 12.
Yes. This is a linearizable execution.
q.enq(x) takes effect between time 0 - 3;
q.enq(y) takes effect between time 4 - 7;
q.deq(x) takes effect between time 8 - 9;

Problem 13.
Yes. This is a linearizable execution.
Task A:$<----q.enq(x)--|-->$
Task B:$--<--|-q.enq(y)-->--<--q.deq()-|->$

Problem 14.
No. This is not a linearizable execution.
Task A's q.enq(x) always happens before Task B's q.enq(y). So Task B's q.deq(y) never happens.

Problem 15.
Thread 1 is enqueueing an item A. After thread 1 increment the tail, it is scheduled out before storing an item to array. Then thread 2 is enqueueing an item B. It can store B into tail. Finally, thread 2 is dequeueing. Dequeueing fails because the item at head is null. Even though the item B is successfully enqueued, thread 2 can't dequeue anything.

Problem 16.
First example:
1. P calls getAndIncrement() and returns 0.
2. Q calls getAndIncrement() and returns 1.
3. Q stores item q at array index 1.
4. R finds array index 0 empty.
5. R finds array index 1 full and dequeues q.
6. P stores item p at array index 0.
7. R finds array index 0 full and dequeues p.

Second example:
1. P calls getAndIncrement() and returns 0.
2. Q calls getAndIncrement() and returns 1.
3. Q stores item q at array index 1.
4. P stores item p at array index 0.
5. R findes array index 0 is full and dequeues p.
6. R findes array index 1 is full and dequeues q.

These doesn't mean enq() is not linearizable buy saying a single linearization point can be defined for all method calls.