1 Java memory model [7 points]

According to what the textbook says about the Java memory model, there is nothing preventing the `reader()` method from dividing by zero if the update to volatile variable v takes place before the update to non-volatile variable x or otherwise becomes visible before x.

However, in actuality this methodology is safe for Java 1.5+ due to how `volatile` now works; see http://stackoverflow.com/questions/8064150 for a detailed explanation (particularly the comments on the accepted answer). In short, the write and read of v acts as a memory fence of sorts that means non-volatile writes that are listed before the write to a volatile variable are guaranteed to be visible to another thread if the write to the volatile variable is visible (so once the reader thread sees that v is `true` it will only see x as `42`).

2 Sequential Consistency (1) [7 points]

For sequentially-consistent ones, listing a valid sequential order.

1. `P2: R(x,0); P1: W(x,1); P2: R(x,1);`
2. Not sequentially consistent: it is not possible to read 0 from `x` once the value 1 has been written to it.
3. `P1: W(x,1); P3: R(x,1); P2: W(x,2); P3: R(x,2);`
4. `P2: W(x,2); P3: R(x,2); P1: W(x,1); P3: R(x,1);`
5. Not sequentially consistent: `P3` and `P4` observe the writes performed by `P1` and `P2` in different order.
6. `P4: R(y,0); R(x,0); P1: W(x,1); R(x,1); R(y,0);`
   `P3: R(x,1); R(y,0); P2: W(y,1); R(y,1); R(x,1);`
7. Not sequentially consistent because, based on `P1`'s observations, `W(x,1)` happens before `W(y,1)`, but based on `P3`'s observations, `W(y,1)` must happen before `W(x,1)`.

3 Sequential Consistency (2) [6 points]

1. ```
   x = 1;
   print(y,z);
   y = 1;
   print(x,z);
   z = 1;
   print(x,y);
   ```

2. ```
   x = 1;
   print(y,z);
   y = 1;
   z = 1;
   print(x,z);
   print(x,y);
   ```

3. There is no interleaving that outputs `001110`. The last statement of any execution prints two different variables because sequential consistency requires program order to be preserved. All three variables are set to `1` before the last print statement, so independent of which variables the last statement prints, both of them are already `1`. In conclusion, the last two characters printed cannot be `10`.

4 Consistency (1) [8 points]

Counterexample using registers x and y with history H:

```
      P x.write(0)
      Q y.write(0)
      Q y:void
      Q y.read()
H =   Q y:1
      Q y.write(1)
      Q y:void
      Q y.read()
      Q y:0
```

H itself is quiescently consistent (it has no quiescent periods), but $H|y$ is not.

5 Consistency (2) [7 points]

The history H from the previous exercise is quiescently consistent but not sequentially consistent. In contrast, the following history is sequentially consistent but not quiescently consistent. With q as a (FIFO) queue:

```
      P q.enq(0)
      P q:void
      Q q.enq(1)
G =   Q q:void
      Q q.deq()
      Q q:1
      Q q.deq()
      Q q:0
```

Quiescent consistency would require that 0 be enqueued before 1, but sequential consistency does not.

6 Linearizability (1) [7 points]

1. H|B =
```
B r.write(1)
B r:void
B r.read()
B r:1
```

2. H|r =
```
B r.write(1)
A r.read()
C r.write(2)
A r:1
B r:void
C r:void
B r.read()
B r:1
C r.read()
```

3. $H' =$
```
B r.write(1)
A r.read()
C r.write(2)
A r:1
B r:void
C r:void
B r.read()
B r:1
A q.write(3)
A q:void
```

4. A history H is sequential (not the same as sequentially consistent) if:

   (a) The first event of H is an invocation.

   (b) Each invocation, except possibly the last, is followed by a matching response. Each response is immediately followed by an invocation.

   The given history is not sequential because, while the first event is an invocation, it is followed by another invocation, not by a matching response. This means the history is concurrent.

5. A history is well-formed if each thread subhistory $H|P$ of H is sequential. For the given H, the thread subhistories are:

$H|A =$
```
A r.read()
A r:1
A q.write(3)
A q:void
```

$H|B =$
```
B r.write(1)
B r:void
B r.read()
B r:1
```

$H|C =$
```
C r.write(2)
C r:void
C r.read()
```

   All of them are sequential; therefore, H is well-formed.

6. A history is linearizable if it can be reordered into a sequential history, which is correct according to the sequential defnition of the object. If

a response preceded an invocation in the original history, it must also precede it in the sequential reordering. We can reorder the history as shown in a sequential fashion that observes the semantics of a register (last written value is stored), so H is linearizable.

```
      C r.write(2)
      C r:void
      B r.write(1)
      B r:void
      A r.read()
S =   A r:1
      B r.read()
      B r:1
      A q.write(3)
      A q:void
      C r.read()
```

7. Two histories H and H' are equivalent if, for every process P, $H|P = H'|P$, which is the case if we swap the first two events in the given history.

7 Linearizability (2) [7 points]

This history is not linearizable as it violates queue semantics. The enqueueing of x precedes the enqueueing of y, thus x should be dequeued first. However, the first dequeue operation retrieves y.

Despite this, it is *sequentially* consistent; sequential consistency does not require the original event precedence to be preserved.

8 Linearizability (3) [7 points]

The given history is not linearizable because the element y is enqueued only once but dequeued twice. That violates queue semantics.

9 Compositional Linearizability [7 points]

Proof. If, in every sequential history S equivalent to H, $S|x$ is not a legal sequential history, then S itself is not a legal sequential history. □

10 More Histories (1) [7 points]

Yes; equivalent to `q.enq(x); q.enq(y); q.deq():x`.

11 More Histories (2) [7 points]

Yes; equivalent to `q.enq(x); q.enq(y); q.deq():x`.

12 More Histories (3) [7 points]

No; `q.enq(x)` precedes `q.enq(y)`, so the `q.deq()` operation cannot return `y`.

13 AtomicInteger [7 points]

Suppose A increments the tail to 1, but suspends before storing its value a in the array. Then B enqueues another value b, incrementing the tail to 2, and storing b in the array. B then does a dequeue. It sees the head slot is `null` and throws an empty exception, when, in fact, the queue cannot be empty because the enqueue of b precedes the dequeue, but no thread has removed b.

14 Herlihy/Wing queue [9 points]

Here is an execution where two `enq()` calls are not linearized in the order they execute line 14.

1. At line 14, P calls `getAndIncrement()`, returns 0.
2. At line 14, Q calls `getAndIncrement()`, 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, dequeues q.
6. P stores item p at array index 0.
7. R finds array index 0 full, dequeues p.

Here is an execution where two `enq()` calls are not linearized in the order they execute line 15.

1. P calls `getAndIncrement()`, returns 0.
2. Q calls `getAndIncrement()`, returns 1.
3. At line 15, Q stores item q at array index 1.
4. At line 15, P stores item q at array index 0.
5. R finds array index 0 full, dequeues p.
6. R finds array index 1 full, dequeues q.

These examples do *not* mean the method is not linearizable, it just means that we cannot define a single linearization point that works for all method calls. In fact, this was proven by Herlihy and Wing in their seminal paper on linearization [1].

References

[1] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.