

Notation We use $R(1)$ or $W(1)$ as shorthand for a method call that writes 1, and so on.

Exercise 21. Explain why quiescent consistency is compositional.

Solution Consider a history H involving multiple quiescently-consistent objects x_0, \dots, x_k . Write H as the concatenation of subhistories $H_0 \dots H_\ell$ where each H_i is separated from its neighbors by a quiescent period. Notice that each quiescent period in H is also a quiescent period in each $H|x_j$. Because each object x_j is quiescently consistent, each subhistory $H_i|x_j$ is equivalent to a sequential G_{ij} , and $G_{0j} \dots G_{kj}$ is equivalent to $H|x_j$. Moreover, H is equivalent to the sequential history

$$G_{00} \dots G_{0\ell} \dots G_{k0} \dots G_{k\ell}$$

in which every method call in H_i is ordered before any method call in H_{i+1} .

Exercise 22. Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory. Does the converse hold? If the memory is quiescently consistent, are the individual registers quiescently consistent? Outline a proof, or give a counterexample.

Solution Here is a counterexample. Suppose x and y are the registers. Consider the following history H :

$$\langle x \text{ write}(0) P \rangle \langle y \text{ write}(0) Q \rangle \langle y \text{ OK}() Q \rangle \langle y \text{ read}() Q \rangle \langle y \text{ OK}(1) Q \rangle \langle y \text{ write}(1) Q \rangle \langle y \text{ OK}() Q \rangle \langle y \text{ read}() Q \rangle \langle y \text{ OK}(0) Q \rangle$$

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

Exercise 23. Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

Solution The history H from the previous exercise is quiescently consistent but not sequentially consistent.

The following history is sequentially consistent but not quiescently consistent. Here, q is a FIFO queue.

1. P enqueues 0.
2. Q enqueues 1.
3. Q dequeues 1.
4. Q dequeues 0.

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

Exercise 24. For each of the histories shown in Figs. 1 and 2, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer.

Solution For the history shown in Fig. 1:

- This history is linearizable: $W(1)$, $R(1)$, and $W(2)$ overlap, so order $W(1)$ first, $R(1)$ second, and $W(2)$ third. These calls all precede $R(2)$, so order it last. The result is a legal sequential register history.

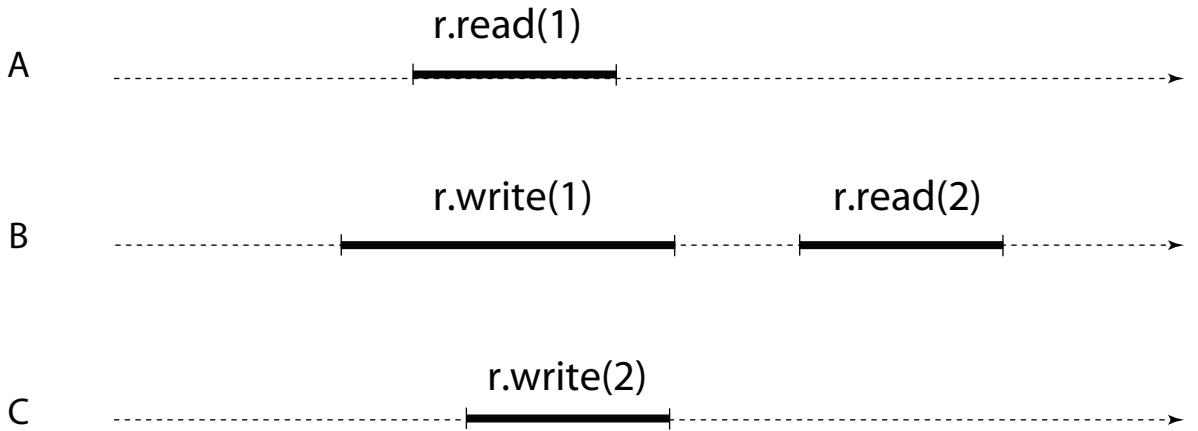


Figure 1: First history for Exercise 24.

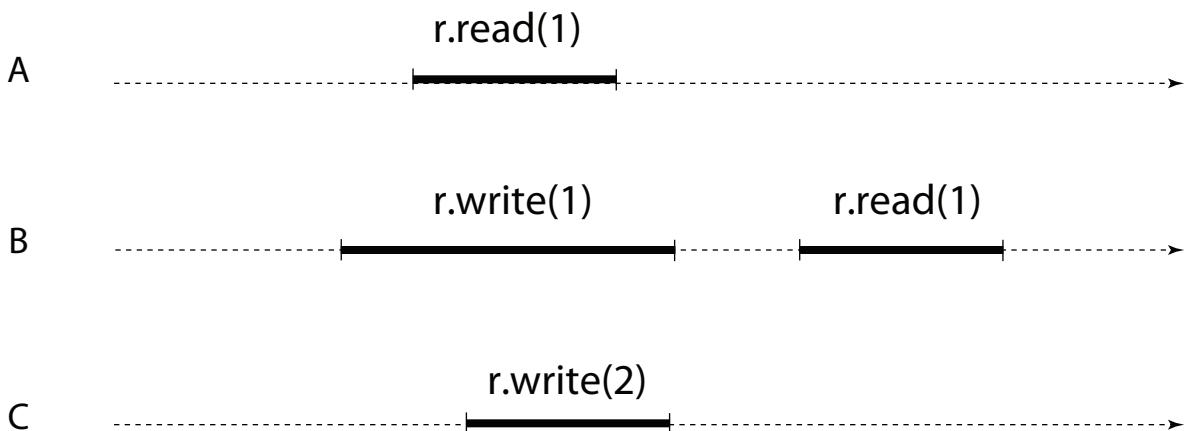


Figure 2: Second history for Exercise 24.

- It follows this history is also sequentially consistent.
- This history is quiescently consistent: $W(1)$, $R(1)$, and $W(2)$ are separated from $R(2)$ by a quiescent interval, so order $W(1)$ first, $R(1)$ second, and $W(2)$ third, and $R(2)$ last.

For the history shown in Fig. 1:

- This history is linearizable: $W(1)$, $R(1)$, and $W(2)$ overlap, so order $W(2)$ first, $W(1)$ second, and the first $R(1)$ third. These calls all precede the second $R(1)$, so order it last. The result is a legal sequential register history.
- It follows this history is also sequentially consistent.
- This history is quiescently consistent: $W(1)$, $R(1)$, and $W(2)$ are separated from $R(2)$ by a quiescent interval, so order $W(2)$ first, $W(1)$ second, the first $R(1)$ third. and the second $R(1)$ last.

Exercise 25. If we drop condition L2 from the linearizability definition, is the resulting property the same as sequential consistency? Explain.

```

1  class IQueue<T> {
2      AtomicInteger head = new AtomicInteger(0);
3      AtomicInteger tail = new AtomicInteger(0);
4      T[] items = (T[]) new Object[Integer.MAX_VALUE];
5      public void enq(T x) {
6          int slot;
7          do {
8              slot = tail.get();
9              } while (!tail.compareAndSet(slot, slot+1));
10             items[slot] = x;
11         }
12     public T deq() throws EmptyException {
13         T value;
14         int slot;
15         do {
16             slot = head.get();
17             value = items[slot];
18             if (value == null)
19                 throw new EmptyException();
20             } while (!head.compareAndSet(slot, slot+1));
21             return value;
22         }
23     }
```

Figure 3: IQueue implementation.

Solution *No.* If we drop L2, there is nothing to require that calls issued by a single thread appear to take effect in that order, thus *program order* is not enforced.

Exercise 26. Prove the “only if” part of Theorem ??

Solution

Theorem 0.0.1. H is linearizable only if, for each object x , $H|x$ is linearizable.

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. \square

Exercise 27. The `AtomicInteger` class (in the `java.util.concurrent.atomic` package) is a container for an integer value. One of its methods is

`boolean compareAndSet(int expect, int update).`

This method compares the object’s current value to `expect`. If the values are equal, then it atomically replaces the object’s value with `update` and returns `true`. Otherwise, it leaves the object’s value unchanged, and returns `false`. This class also provides

`int get()`

which returns the object’s actual value.

Consider the FIFO queue implementation shown in Fig. 3. It stores its items in an array `items`, which, for simplicity, we will assume has unbounded size. It has two `AtomicInteger` fields: `tail` is the index of the next slot from which to remove an item, and `head` is the index of the next slot in which to place an item. Give an example showing that this implementation is *not* linearizable.

Solution 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 .

Exercise 28. Consider the class shown in Fig. 4. According to what you have been told about the Java memory model, will the `reader` method ever divide by zero?

Solution The `reader()` method will divide by zero if an update to volatile variable v takes place before an update to non-volatile variable x . Nothing we have said rules out a possible division by zero. (If x were volatile, division by zero would be impossible.)

```

1  class VolatileExample {
2      int x = 0;
3      volatile boolean v = false;
4      public void writer() {
5          x = 42;
6          v = true;
7      }
8      public void reader() {
9          if (v == true) {
10              int y = 100/x;
11          }
12      }
13 }
```

Figure 4: Volatile field example from Exercise 28.

Exercise 29. Is the following property equivalent to saying that object x is wait-free?

For every infinite history H of x , every thread that takes an infinite number of steps in H completes an infinite number of method calls.

Solution Yes, this property is the same as being wait-free. The question is not well-phrased, however, since it is not clear about which steps appear in the history. The intent is that the history includes steps of x 's implementation, not just top-level method calls.

Exercise 30. Is the following property equivalent to saying that object x is lock-free?

For every infinite history H of x , an infinite number of method calls are completed.

Solution Yes, with the same caveats as the previous exercise.

Exercise 31. Consider the following rather unusual implementation of a method m . In every history, the i^{th} time a thread calls m , the call returns after 2^i steps. Is this method wait-free, bounded wait-free, or neither?

Solution This method is wait-free (every call that keeps running eventually returns) but not bounded wait-free.

```

1  public class HWQueue<T> {
2      AtomicReference<T>[] items;
3      AtomicInteger tail ;
4      static final int CAPACITY = 1024;
5
6      public HWQueue() {
7          items = (AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8              CAPACITY);
9          for (int i = 0; i < items.length; i++) {
10             items[i] = new AtomicReference<T>(null);
11         }
12         tail = new AtomicInteger(0);
13     }
14     public void enq(T x) {
15         int i = tail.getAndIncrement();
16         items[i].set(x);
17     }
18     public T deq() {
19         while (true) {
20             int range = tail.get();
21             for (int i = 0; i < range; i++) {
22                 T value = items[i].getAndSet(null);
23                 if (value != null) {
24                     return value;
25                 }
26             }
27         }
28     }
29 }
```

Figure 5: Herlihy/Wing queue.

Exercise 32. This exercise examines a queue implementation (Fig. 5) whose `enq()` method does not have a linearization point.

The queue stores its items in an `items` array, which for simplicity we will assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail`, and then stores the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.

The `deq()` method reads the value of `tail`, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps `null` with the current contents, returning the first non-`null` item it finds. If all slots are `null`, the procedure is restarted.

Give an example execution showing that the linearization point for `enq()`

cannot occur at Line 15.

Hint: give an execution where two `enq()` calls are not linearized in the order they execute Line 15.

Give another example execution showing that the linearization point for `enq()` cannot occur at Line 16.

Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

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

1. At Line 15, P calls `getAndIncrement()`, returns 0.
2. At Line 15, 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 16.

1. P calls `getAndIncrement()`, returns 0.
2. Q calls `getAndIncrement()`, returns 1.
3. At Line 16, Q stores item q at array index 1.
4. At Line 16, 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.

Exercise 33. Prove that sequential consistency is nonblocking.

Solution The proof is exactly the same as the proof for linearizability.