What is sequential consistency (SC)?

When thinking about concurrent programs, we implicitly assume what an execution can look like. E.g. in the following program, it is unintuitive for us to see "0" printed.

```
        A = B = 0
   T1                  T2
B = 42          if (A == 1) {
A = 1               print(B)
                }
```

If A was written, we expect B to have been written. This intuitive notion that we have of what executions are possible is exactly captured with SC. The exact assumptions we make under SC are:

① Any execution behaves as if each operation had been executed sequentially (i.e. one after the other) by a single thread.

This assumption seems obvious (what else should happen?), but nevertheless needs to be stated. For example, this then directly implies that the only possible results of the concurrent writes T_1 |B=1| / T_2 |B=2| are 1 or 2, and not something in between.

② This equivalent sequential execution obeys program order (PO).

This then eliminates the unexpected execution above. If the program execution was SC and T_2 printed "0", then T_2 saw that A=1 was written, so the equivalent sequential execution must have had

```
A = 1
if (A == 1) {            ?      B = 42
    print(B)
}
```

Since T_2 read

Now since $B = 42 \xrightarrow{PO} A = 1$, the sequential execution must have been

```
B = 42
A = 1
if (A == 1) {
    print(B)
}
```

But if we execute these operations in isolation on a single thread, we obviously print "42", a contradiction.

This is the same kind of reasoning we use when thinking about bad interleavings or when we proved that the Peterson lock provides mutual exclusion.

Correctness

Now when implementing concurrent objects ourselves (e.g. a queue) the same question comes up of what should happen when multiple threads execute these operations concurrently. And again since it matches our intuition, we could use SC as our condition for whether an object behaves correctly. This means that if we implement a concurrent queue q that allows an execution of

T_1	T_2
q.enq(x)	q.enq(y)
a = q.deq()	

where deq returns null or raises an exception because the queue is empty, then we would consider this an incorrect implementation.

We would call our object "sequentially consistent", if in any possible execution, the operations on a single instance this object are SC (and we ignore all other operations).

Ensuring SC

Already with the Java memory model we saw that sequential consistency is not always guaranteed. In the example from the beginning, it would be possible for the program to print "0" in Java. Memory models however give us synchronization tools to enforce sequential consistency (volatile, locks, etc.) when interacting with memory.

Assuming we know how to implement SC objects, the next question is, how can we be sure that executions using multiple SC objects will be SC?

Will any execution where the behavior of each individual object is SC automatically be SC as a whole? No! This property would be called "composability" and SC does not satisfy this property.

Processors are a real example for this. Despite having caches, (modern) processors guarantee that all accesses to a single memory location will be sequentially consistent (this is called "cache coherency" and is enforced using a cache coherency protocol). So the following execution is not possible

T_1	T_2
$A = 1$	$A = 2$
print(A) // "2"	print(A) // "1"

(because if T_1 prints "2" we must have $A=1 \rightarrow A=2 \rightarrow$ print(A) (T_1) in the equivalent sequential execution and since $A=2 \xrightarrow{po}$ print(A) (T_2), T_2 cannot read A as "1".)

The following, with 2 memory locations can however happen

$A = B = 0$

T_1	T_2
print(A) // "2"	$B = 1$
print(B) // "0"	$A = 2$

The executions of the individual objects (shared variables) are SC

T_1	T_2	T_1	T_2
print(B) // "0"	$B = 1$	print(A) // "2"	$A = 2$

Equivalent sequential executions:

print(B) // "0"
$B = 1$

$A = 2$
print(A) // "2"

but the execution as a whole is not. This could happen if T_2 reordered the two writes.

Another example using queues:

T_1	T_2
q.enq(x)	p.enq(y)
p.enq(x)	q.enq(y)
print(q.deq()) // "y"	print(p.deq()) // "x"

Again, the executions of the individual objects (shared variables) are SC

T_1	T_2	T_1	T_2
q.enq(x)	q.enq(y)	p.enq(x)	p.enq(y)
print(q.deq()) // "y"			print(p.deq()) // "x"

Equivalent sequential executions:

q.enq(y)
q.enq(x)
print(q.deq()) // "y"

p.enq(x)
p.enq(y)
print(p.deq()) // "x"

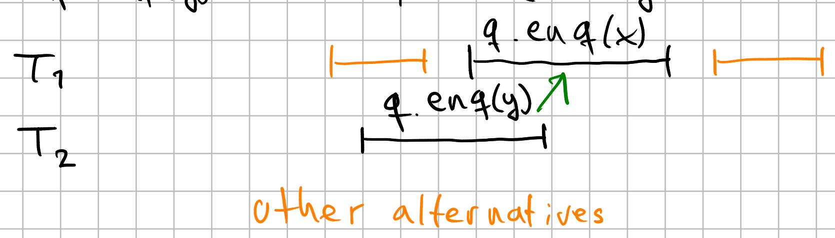
while the entire execution is not. If q.deq() gives "y", then we must have

$q.enq(y) \to q.enq(x)$ in the equiv. seq. exec., but with PO this gives us $p.enq(y) \xrightarrow{PO} q.enq(y) \to q.enq(x) \xrightarrow{PO} p.enq(x)$, so dequeuing "x" from p would not be possible under SC.

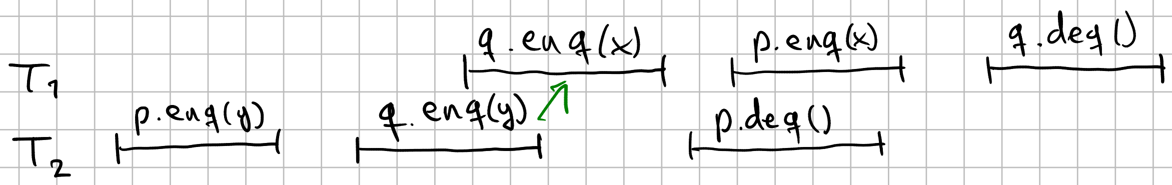
This is undesirable. In the end we want all our executions to be SC in total, but so far SC object implementations alone are not enough to guarantee us this and it is unclear how we can achieve our goal.

Linearizability

As often in math, when our definitions allow for undesirable cases, we try strengthening our assumptions. What went wrong in the previous example? Let's try imagining how the operations may have happened in time. Since $q.enq(y) \to q.enq(x)$, we would intuitively expect that "q.enq(x)" could not have completed before "q.enq(y)" started, so we imagine something like this:



Now if we consider when the other operations should have happened in PO:



Then we would naturally expect that p.deq could only return "y", since p.enq(y) must precede p.enq(x) in time. SC alone says nothing about time, we never mentioned it in our definition. This means that under SC alone, even if p.enq(y) precedes p.enq(x), nothing in our assumptions imply that in the equivalent sequential execution, the operation p.enq(y) would still precede p.enq(x). Important: Normally, we don't care or think about time, because we don't know how our program will be scheduled. But here we see that if we also add this constraint using the real-time ordering of operations, it eliminates the undesired behavior ☹

So let us define linearizability, by adding this new constraint to SC:

We call an execution linearizable, if:

1. It is SC. (see above)
2. The equivalent sequential execution from SC also satisfies all real-time orderings between operations, i.e. for any two operations σ_1, σ_2, if σ_1 precedes σ_2 in the real-time execution, then σ_1 should precede σ_2 in the equivalent sequential execution.

or just one of them if there are multiple.

Another way to say this is, as in the book:

Principle 3.5.1. Each method call should appear to take effect instantaneously at some moment between its invocation and response.

real-time order preserved

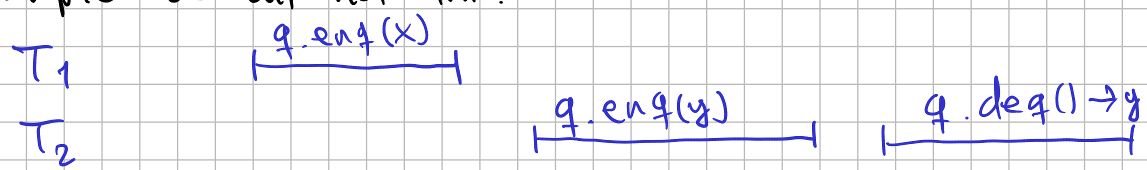
the moment it appears to "happen" and that determines where it is placed in the equivalent sequential execution.

We then call an object linearizable, if in an possible execution, the operations on a single instance this object are linearizable.

Linearizability is in fact composable (proof in the book on page 57), so if we can be sure that we use only linearizable objects, any execution will be linearizable and thus SC; and we are happy ☺ ☹

Linearizability vs. SC

To be able to contrast lin. and SC, we need to include the time component of our executions. The following execution is for example SC but not lin.:



Although it is unintuitive and harder to think of a queue implementation that would work this way, this execution is SC, since we can choose the equivalent sequential execution

q.enq(y)
q.enq(x)
q.deq() // → y

one could imagine each thread having a local buffer for enqueues and threads first checking their local buffer when dequeueing.

obeying program order. We are allowed to "reorder" q.enq(x) and q.enq(y) because SC says nothing about time.

Do NOT remember SC by "allowed to reorder operations in time". The picture of SC to keep in mind are the intuition/assumptions mentioned in the first section. Whenever you see SC in other places (it is an important term!), they will be referring to these ideas and not histories where inter-thread order can be ignored.

The real-time order between threads would only allow the sequential execution

```
q.enq(x)
q.enq(y)
q.deq() // -> y
```

which is not possible, hence the execution is not linearizable.

<u>Linearization points:</u>

The last open question is how to show that an object is linearizable. The way to do this is by identifying "linearization points" in our program. These are points in our program that occur atomically/instantaneously and determine when the operation has "completed" or "taken effect" (but not yet returned). There is no precise definition of what "completed" or "taken effect" means. What the linearization points need to fulfil is that if we order all operations by their linearization points in time, this gives us the (an) equivalent sequential execution, called the (a) "linearization" of the execution.