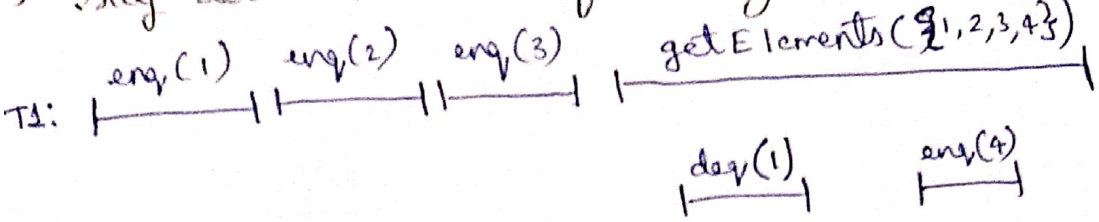
i) Say we have the following execution:



To be more precise:
getElements() executes #8 only once and then goes to sleep. which results in rv[.] storing 1 as the first elmt.
Meanwhile the other thread dequeues and the resultant state is {2,3}. followed by an enqueue which results in state {2,3,4}.

Now T1 resumes, upon which it reads {2,3,4} and stores it in rv. and so it returns {1,2,3,4} but this state is never possible according to the execution.
Hence, it is non-linearizable.

We could also show a cycle.

since, getElements() has '4' in it, it should have happened after enq(4).

⇒ "enq(4)" ⟶ "getElements({1,2,3,4})"

getElements() has '1' in it, so it should have happened before deq(1)

⇒ "getElements({1,2,3,4})" ⟶ "deq(1)"

But "deq(1)" ⟶ "enq(4)"

Hence, we have a cycle, so it is non-linearizable

2) No, they are not equivalent.

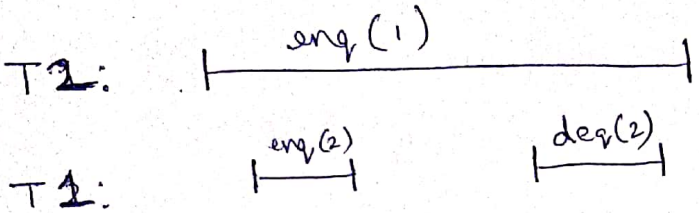
Let us say x has two methods.
1st method is lock free and in 2nd method deadlock is possible

Let us say there are 3 threads a, b, c.

a executes method 1 always.
b & c execute method 2.

Now b & c might have ended up in a deadlock but since a executes lock free method it could keep executing.
clearly object is not lock free as it has deadlock method but infinite history H of x has infinite number of method calls completed.

3) Assume the following execution: (tail = 0 initially)



For this to be linearizable:

enq(2) → enq(1), so that deq() returns 2.

Now we show the following to prove its not: (lets say we did enq(1) → enq(2))

T1: i = tail.getAndIncrement(); (i = 0)

T2: i = tail.getAndIncrement(); (i = 1)

a) Linearizable point for enqueue cannot be #15.

Assume it is so.

For above execution let following be the order:

T2: i = 0 (by executing #15)

T1: i = 1 (by executing #15)

T1: items[1] = 2

T1: deq(2)

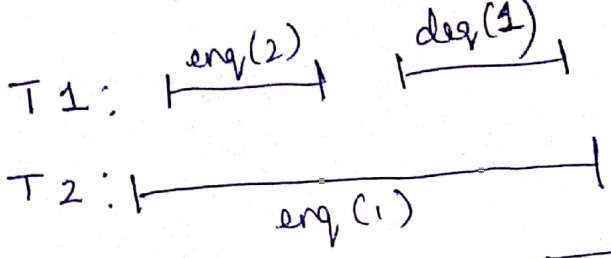
T2: items[0] = 1.

Now even though T2 executed #15 before T1.

T1 "enqueued" 2 before T2 got to "enqueue" 1.

Hence, #15 is not the linearization point

3b) Linearization point is #16.



similar to earlier, enq(1) → enq(2).

~~as shown earlier enq(2) → enq(1).~~

Now let us show following order:

T2: i = 0 (via #15)

T1: i = 1 (via #15)

T1: items[1] = 2

T2: items[0] = 1

T1: deq(1).

Here, even though T1 executed line #16 before T2, that is enq(2) before enq(1) if #16 was linearization point in reality enq(1) → enq(2).

so, #16 is not the linearization point

4) True.

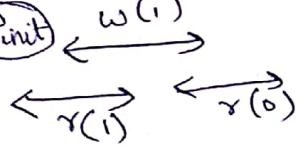
since individual registers are regular read() call will return new or old value & on overlap with write() call which is perfectly reasonable.

we dont read future as regular makes sure its only currently written value or old ones.

we dont read distant past as after write call is completed, regular makes sure that value is only returned on no overlaps.

5) It still works.

The anomaly in case of regular register is that if read overlaps with write call it can return the value being written or the old value.

say for example 

it reads 1 and then 0. which is acceptable. (i.e) order might change here. it enters as expected.

If only one thread wants to enter CS

If two threads want to enter:

If thread A tries to enter while thread B is at #10.

If A hasnt updated flag yet, then B enters CS as usual while A is busy in while loop.

If A updates flag & B reads true as normal execution it waits until A sets itself to victim & enters CS.

If A updates flag & B reads false, then B would enter CS as it should have anyway when A sets itself to victim.

So, it runs as expected for lock in parallel.

If B was at unlock & A is stuck busy spinning it can read either true or false.

If it reads true it would just spin once more until write call ends or B eventually setting itself to intim on next lock()

If it reads false, then it works as intended by default

Hence, we proved that Peterson lock algorithm still works.