Properties of concurrent computation

Essential Properties

- Correctness (Safety): bad things never happens
- Progress (Liveness): a good thing eventually happens

Essential Properties

- Correctness (Safety): bad things never happens
  - Quiescent Consistency
  - Linearizability
  - Sequential Consistency
- Progress (Liveness): a good thing eventually happens

Essential Properties

- Correctness (Safety): bad things never happens
  - Quiescent Consistency Weak
  - Sequential Consistency weak
  - Linearizability Standard
- Progress (Liveness): a good thing eventually happens

Sequential Correctness

- Formally: provide all correct executions (problem: infinite)
- Alternative: describe in words.
- Typical properties defined inductively:
  - Initial state
  - Applying an operation:
    If the object is in state S satisfies something,
    then after applying an operation O,
    the next state S' satisfies something (may relate to S)

precondition

postcondition

Sequential Object

- Example: a queue
- State is an ordered sequence of values

Op	Precondition	Postcondition
enq(z)	q	$q \cdot z$
deq()	$x \cdot q$	q and return value = x
deq()	[ ]	EmptyException

Sequential Correctness

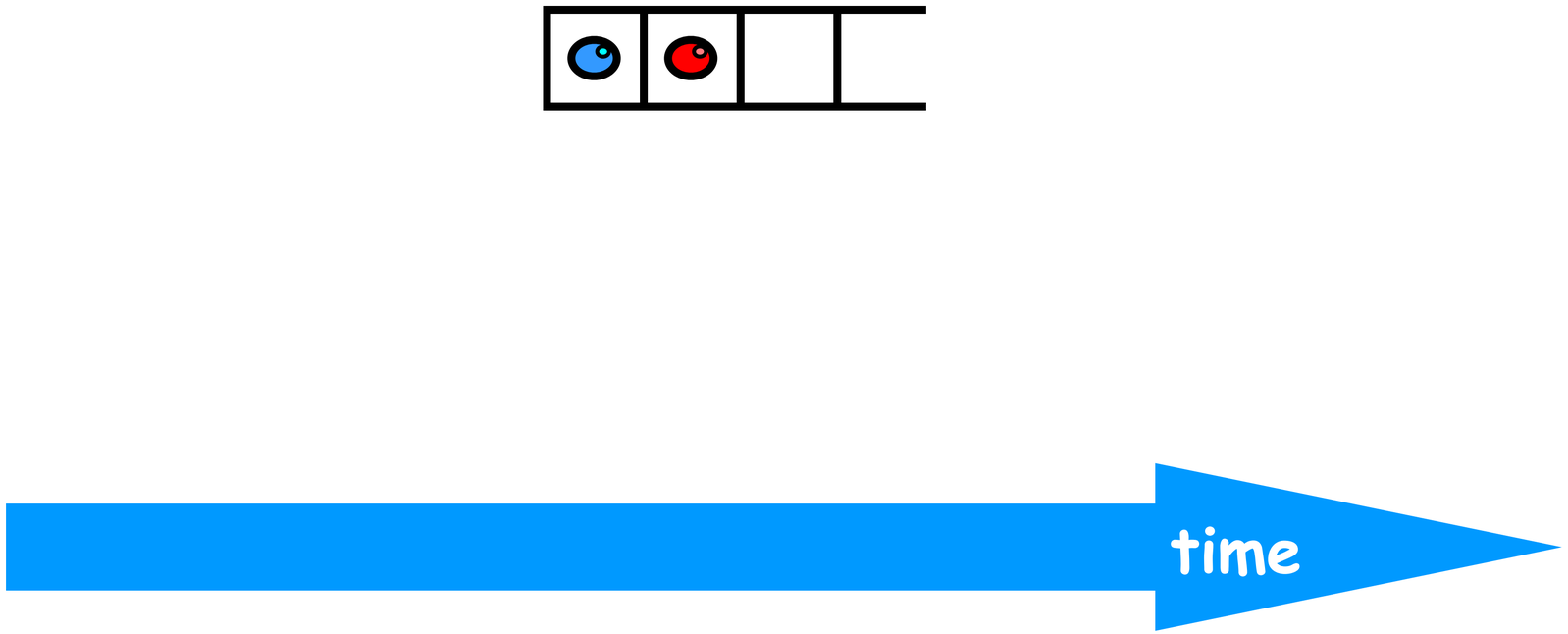
- Typical properties defined inductively:
  - Initial state
  - Preconditions and postconditions per operation

- Very powerful: short elegant formal definitions
- Main tool: the state.
- State is not really needed. We only care about results of operations.

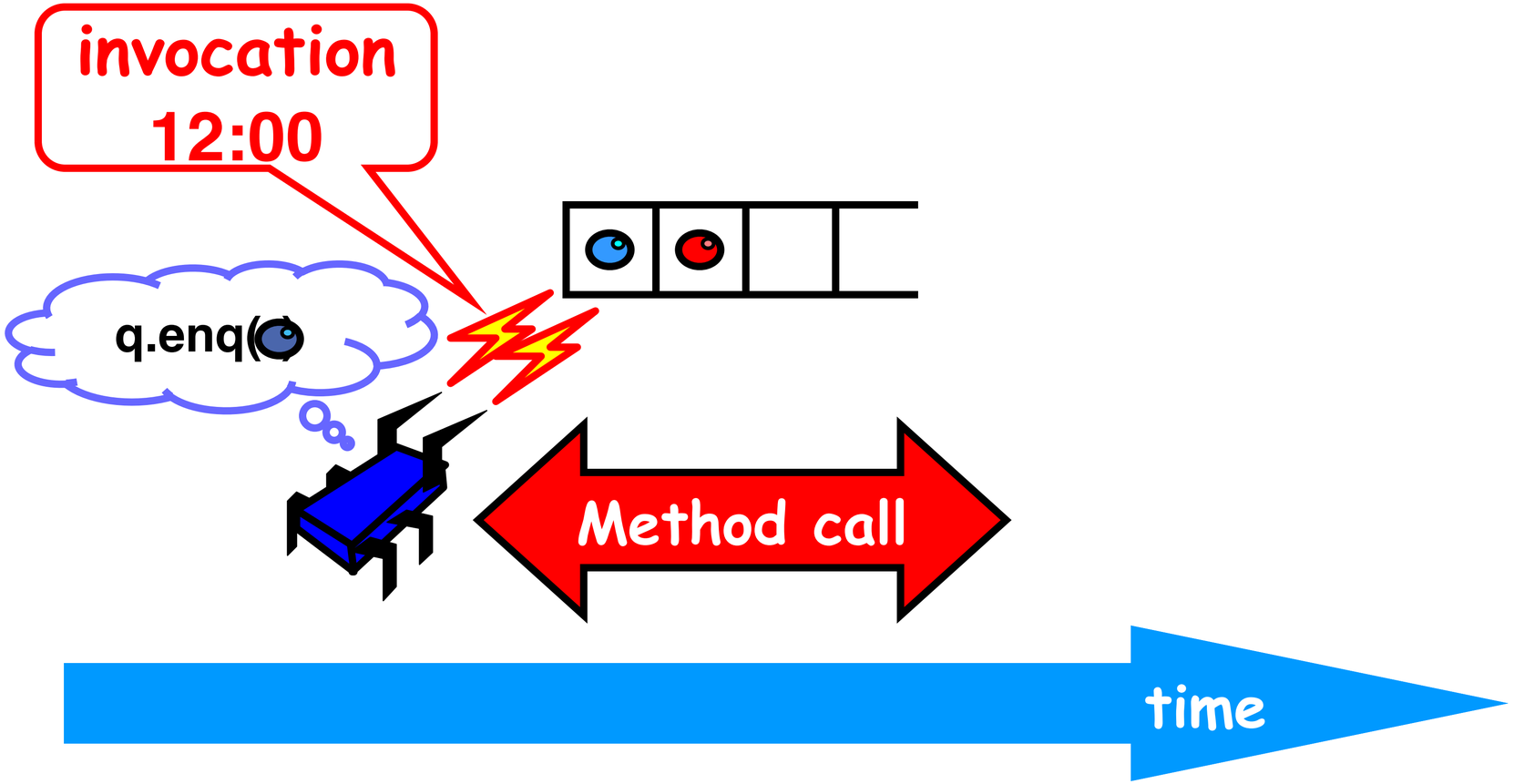
Concurrent Specification:

The same?

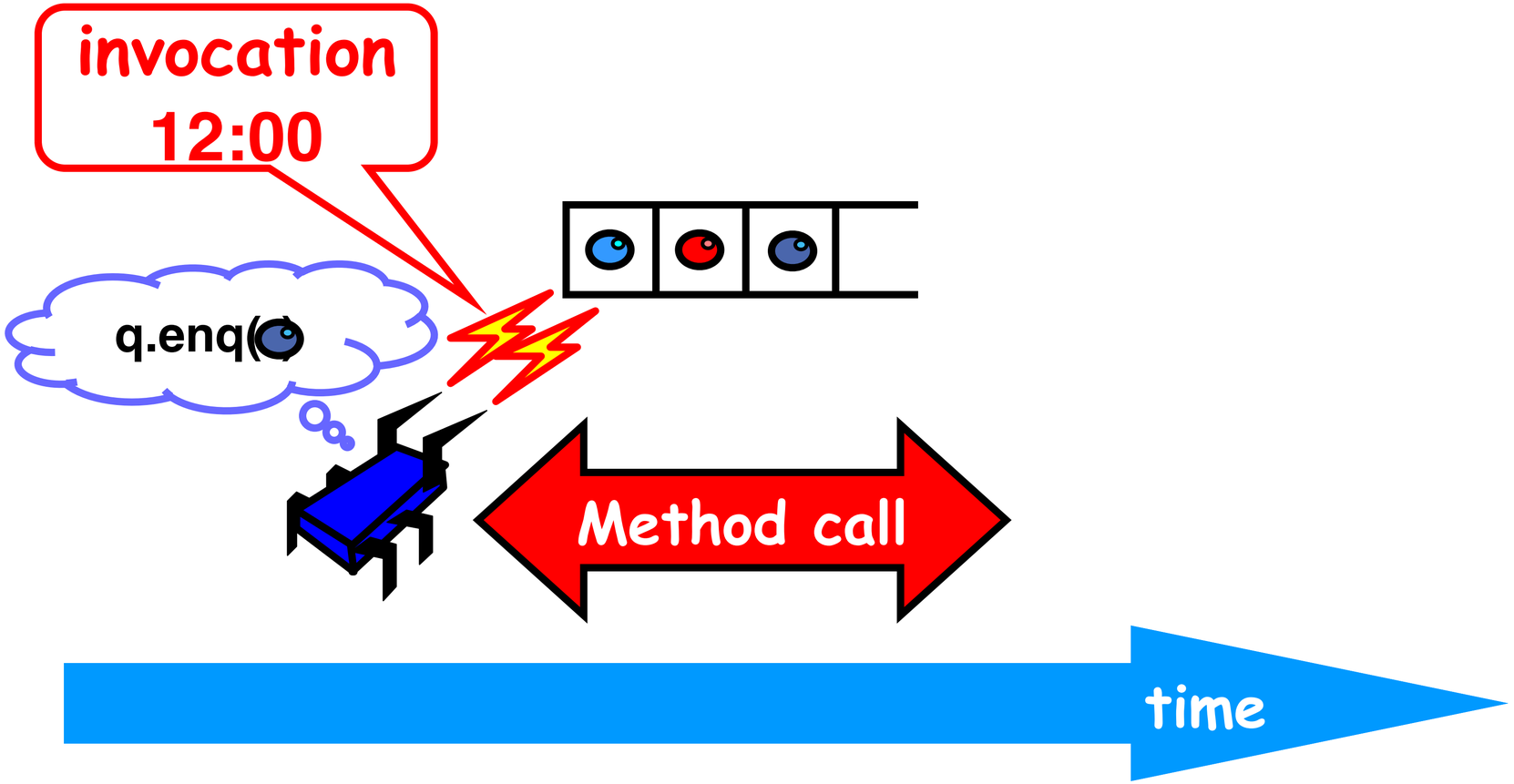
Methods Take Time



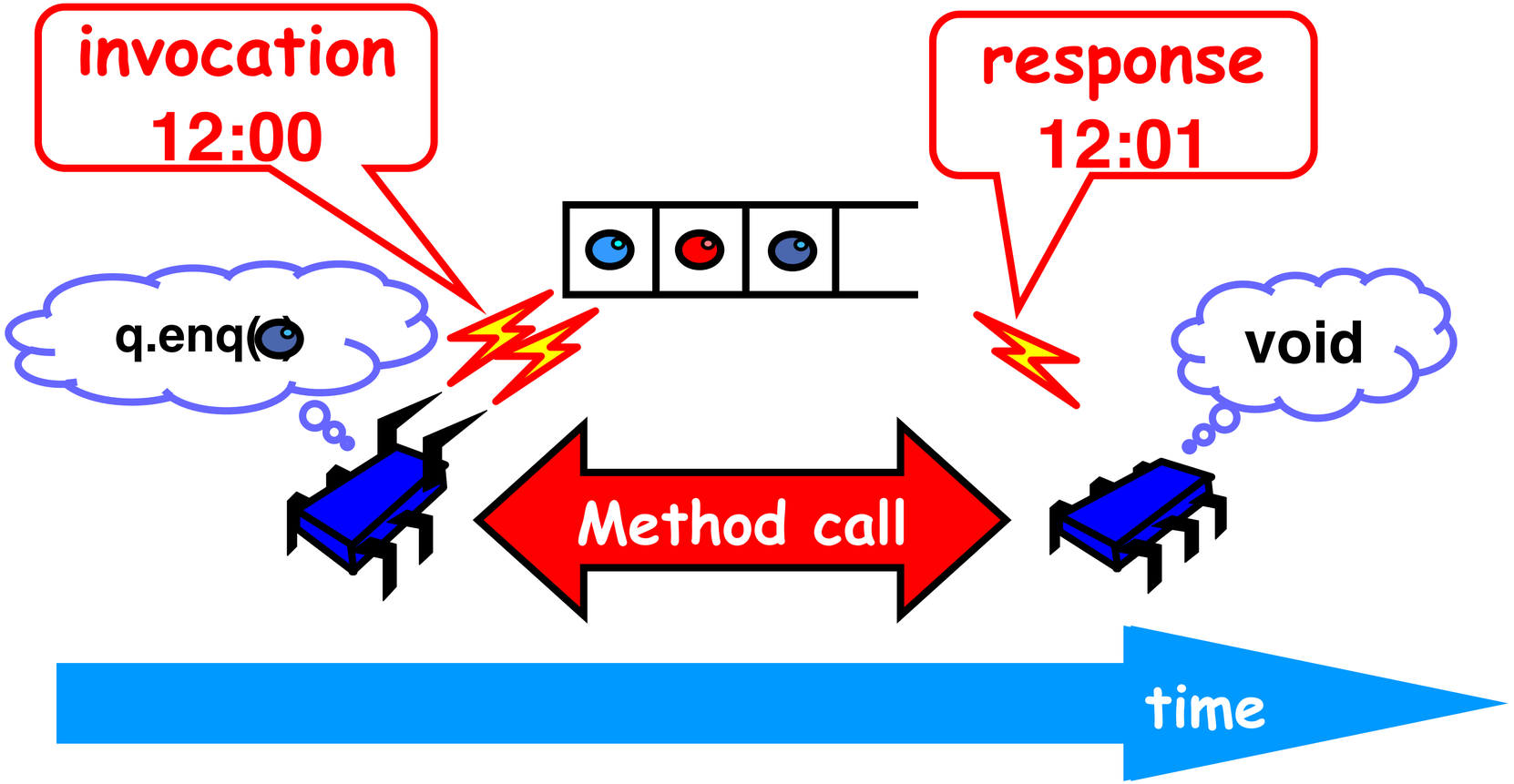
Methods Take Time



Methods Take Time

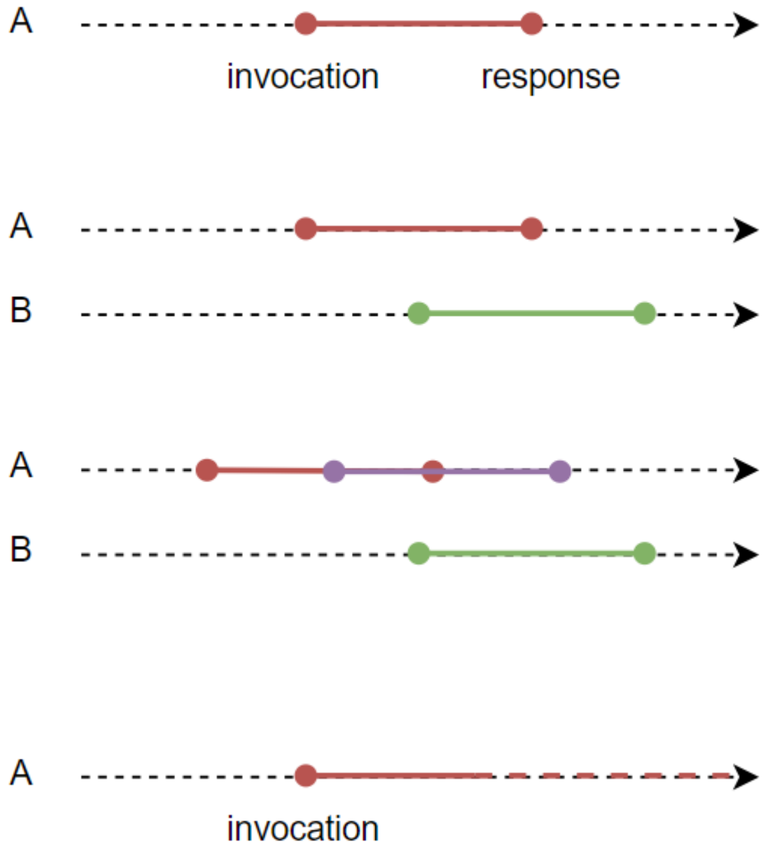


Methods Take Time



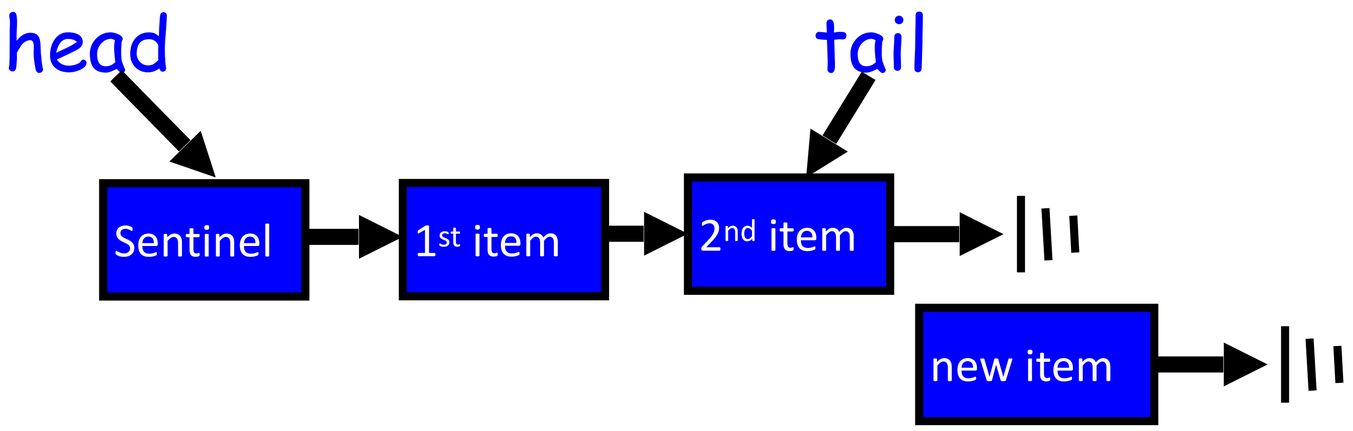
Invocation and Response

- A method call is the interval that starts with an *invocation* event and ends with a *response* event
- Method calls by concurrent threads may overlap
- Method calls by a single thread are always sequential
- A method call is pending if its call event has occurred, but not its response event



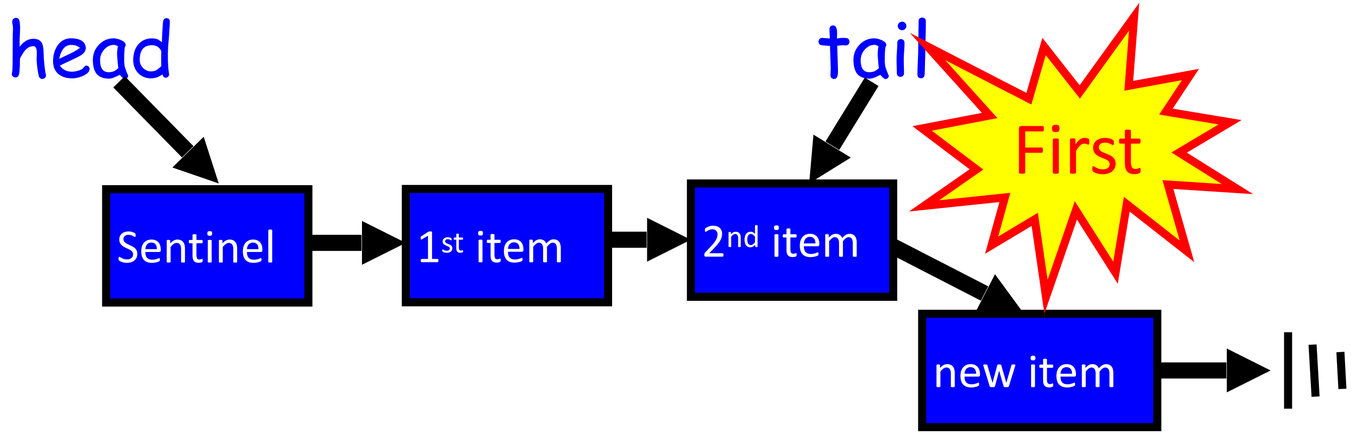
Example: Concurrent Queue

- Think of enqueue.
- Two modifications
  - Add node after tail,
  - Modify tail.



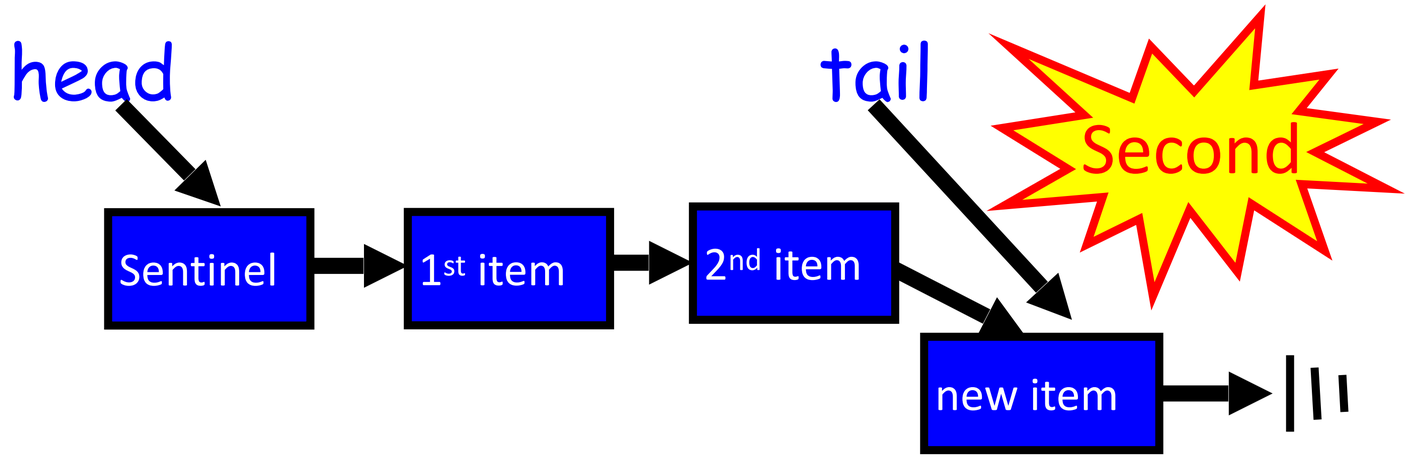
Concurrent Queue

- Think of enqueue.
- Two modifications
  - Add node after tail,
  - Modify tail.



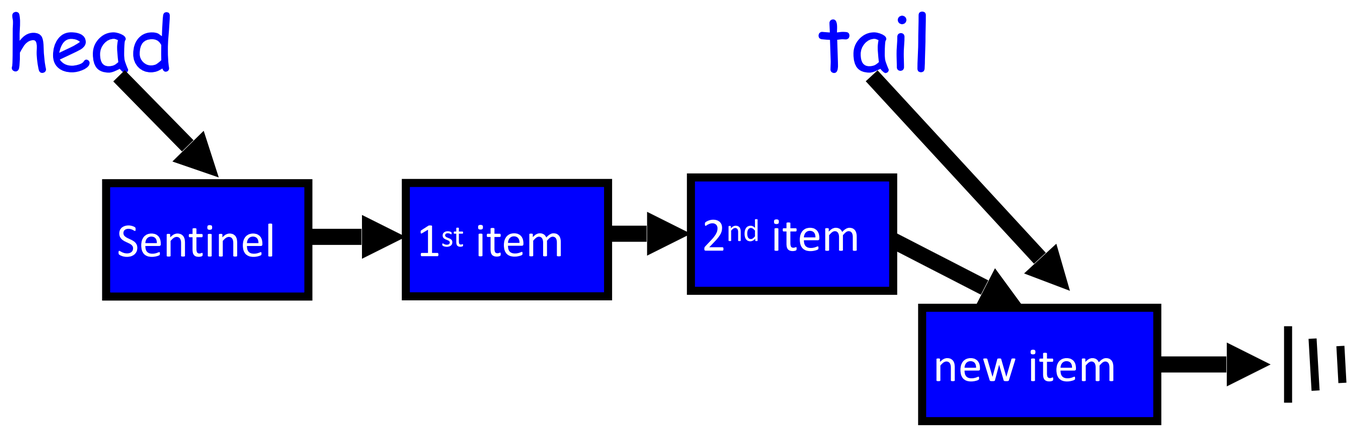
Concurrent Queue

- Think of enqueue.
- Two modifications
  - Add node after tail,
  - Modify tail.



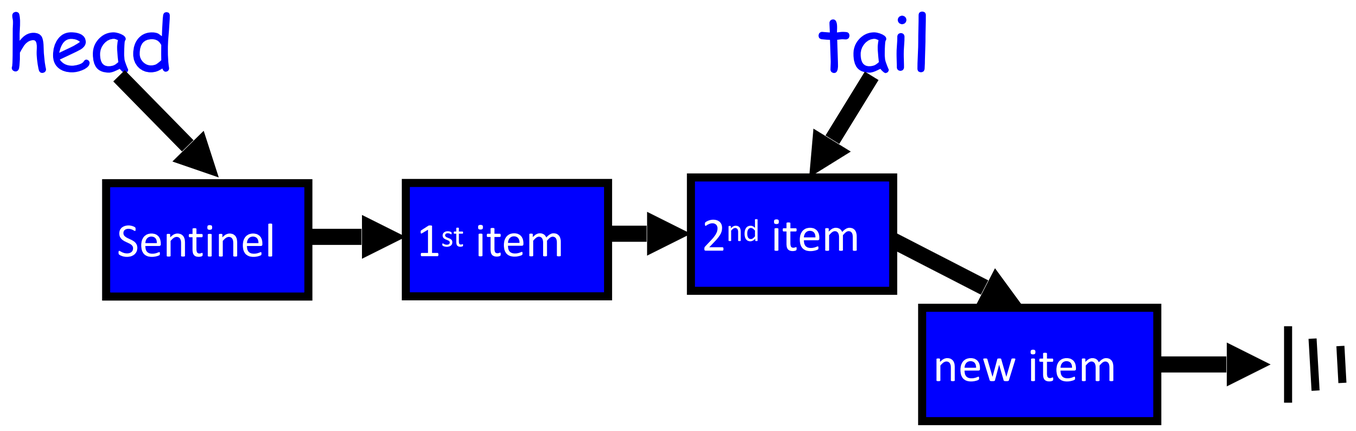
Concurrent Queue

- Think of enqueue.
- Two modifications
  - Add node after tail,
  - Modify tail.



An Intermediate State

- After the first modification, structure is “inconsistent”
- Must make sure that other operations are OK with this view



Complication

- **Sequential**: specify state before and after an operation
- **Concurrent**: Object might never be between operation.

- **Sequential**: think of each operation separately
- **Concurrent**: must consider interaction between any multiple concurrent operations
  - What if 2 enqueues overlap? What if 5 enqueue and 7 dequeues overlap?

- **Sequential**: add new operations without affecting existing
- **Concurrent**: old operations must be OK with new ones

Define Correctness (Simplification)

- Use the **sequential** correctness!
- This **concurrent** execution “looks like” a **sequential** execution, and the **sequential** execution is correct
- “Looks like”: sees the same results

Correctness

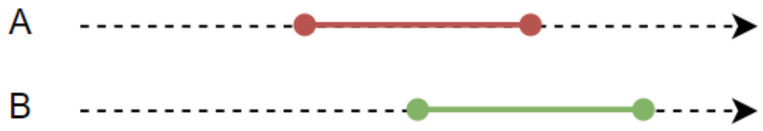
- What does it mean for a **concurrent** object to be correct?

- What *is* a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Concurrent means ambiguous temporal order

- If two enqueues are concurrent, which of them should be first?

- Answer: we don't care

- Basic non-rule: if two operations happen concurrently, any order between them is fine.



Correctness

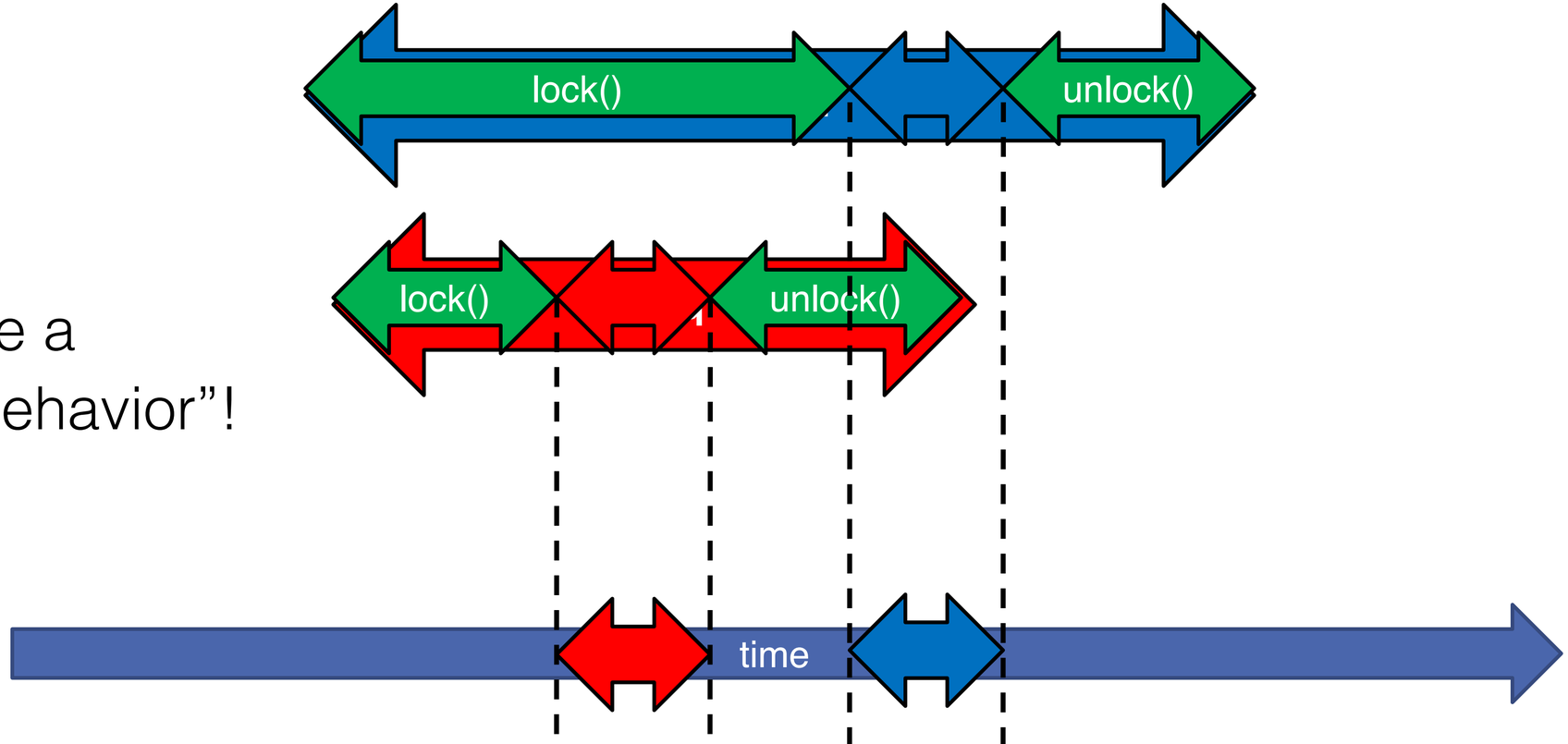
- Let's consider locks.
- We believe a queue with locking is fine, but why?

```
public void enq(T x) throws FullException {
  lock.lock();
  try {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
     tail++;
  }  finally {
    lock.unlock();
  }
}
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Why is the lock-based implementation correct?

Let's explain the notion "concurrent" via "sequential":

Locks provide a "sequential behavior"!



The Problem with Locks

- If all methods hold exclusive locks, then we get no concurrency, and no scalability.
- Is the following non-locking queue correct?
- It doesn't look good. But what is "correct"?

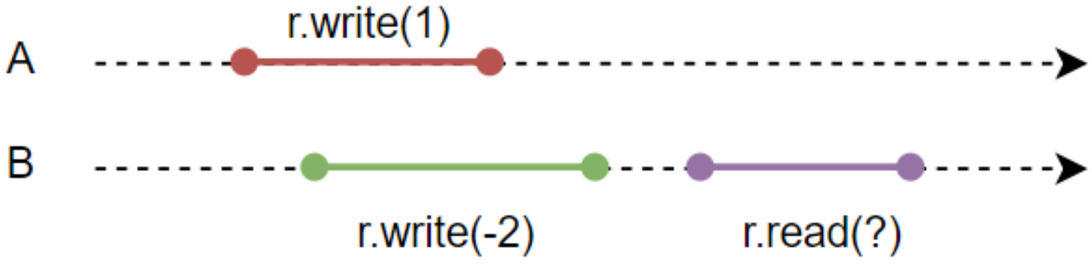
```
public void enq(T x) throws FullException {
  if (tail - head == items.length)
    throw new FullException();
  items[tail % items.length] = x;
  tail++;
}
public T deq() throws EmptyException {
  if (tail - head == 0)
    throw new EmptyException();
  T x = items[head % items.length];
  head++;
  return x;
}
```

Correctness Conditions

- Quiescent consistency
  - Allows high performance at the cost of “weak correctness”
- Sequential consistency
- Linearizability
  - Strong correctness and composability
  - De-facto standard correctness for concurrent systems

First Principle

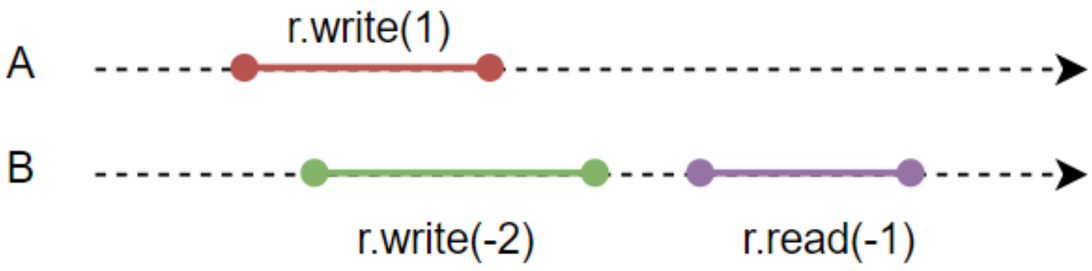
- Intuition:



- What do we except B to read?

First Principle

- Intuition:



- What do we except B to read? Surely not a mixture of the writes!

First Principle

- This leads us to the first principle:

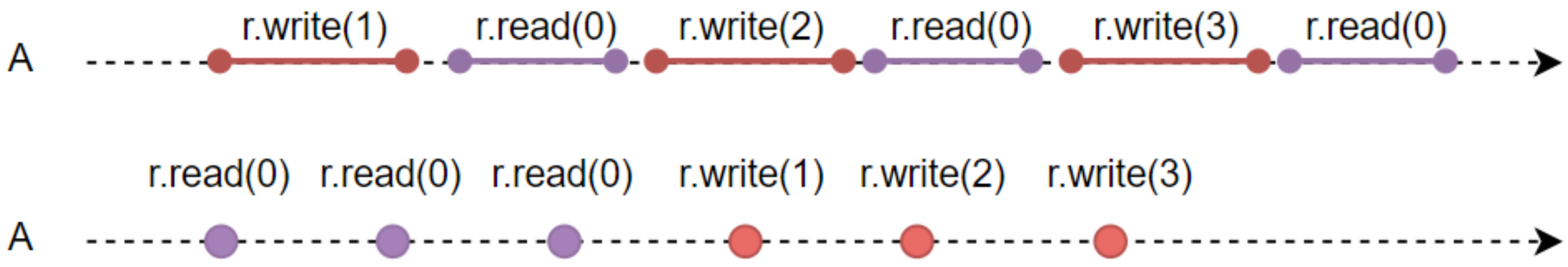
1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

First Principle

- This leads us to the first principle:

1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

- By itself, this principle is too weak to be useful.



Quiescent Consistency

לא פעיל

- An object is quiescent if it has no pending method calls

2. Method calls separated by a period of quiescence should appear to take effect in their real-time order

Quiescent Consistency

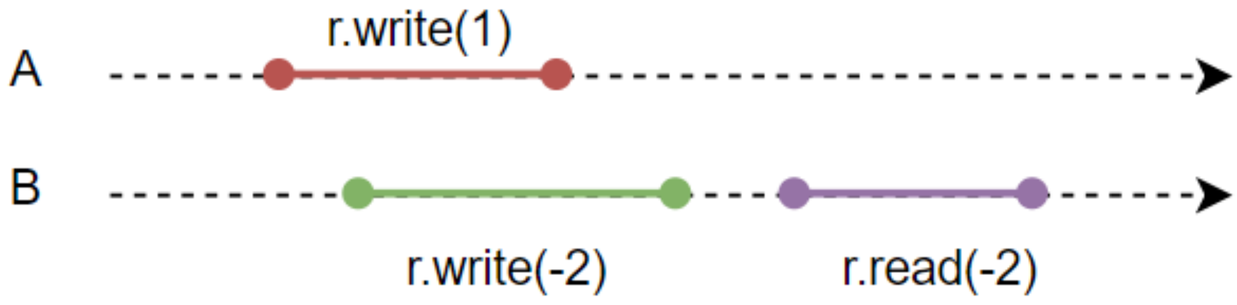
1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

2. Method calls separated by a period of quiescence should appear to take effect in their real-time order

- Informally - non-overlapping operations take effect in their real-time order, overlapping operations might be reordered

Quiescent Consistency

- Check out the following example:

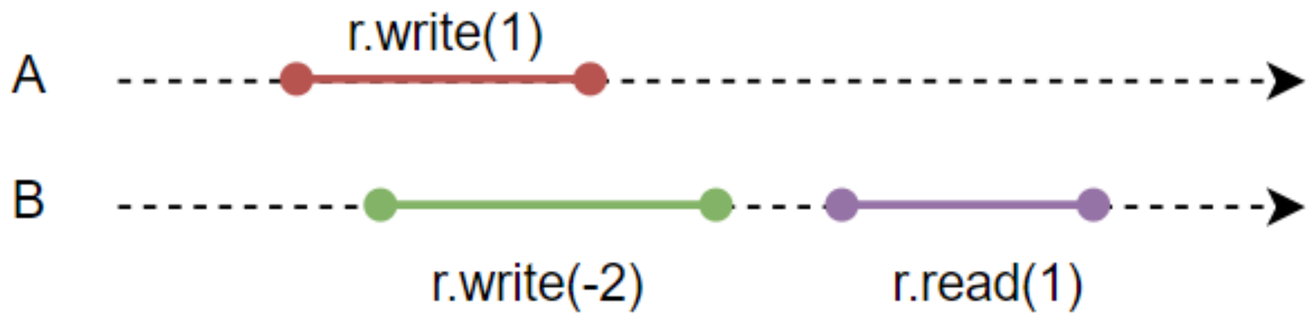


- Is it quiescent consistent? **Yes!**

Quiescent Consistency

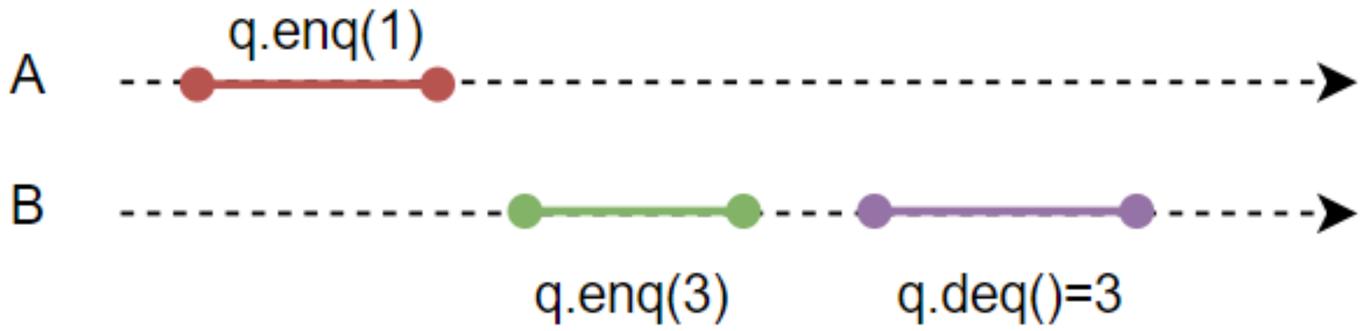
- How about this one?

Yes!



Quiescent Consistency

- And this?

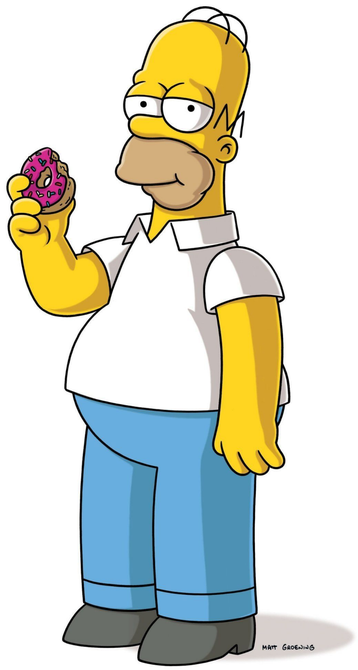


Quiescent Consistency

- Sounds like a property of an execution..?🤔
- An object (data structure) has this property if any possible execution has this property
- Quiescent consistency is compositional

Compositional Correctness

- A correctness property P is **compositional** if, whenever each object in the system satisfies P, the system as a whole satisfies P



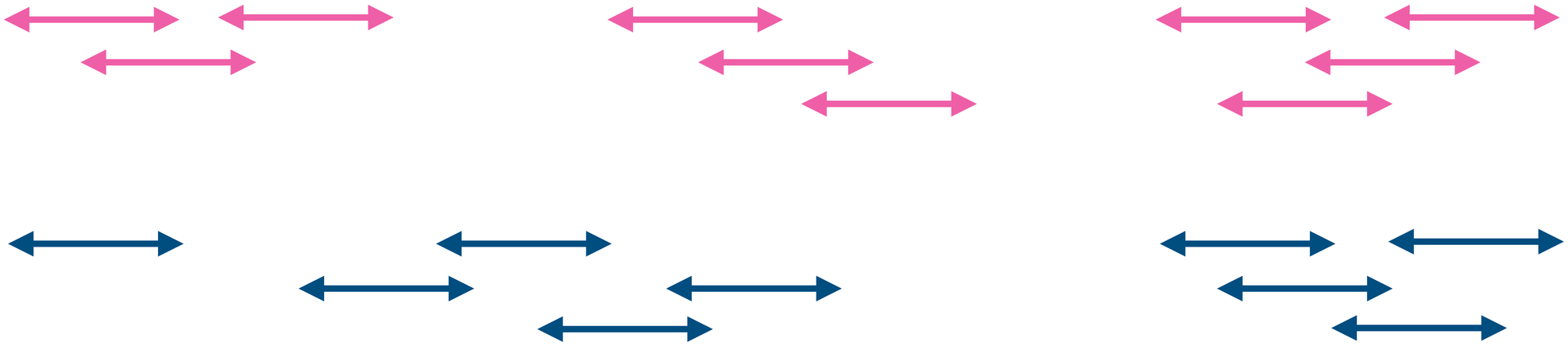
Quiescent Consistency

- An object (data structure) is quiescently consistent if any possible execution of the object is quiescently consistent

- Quiescent consistency is compositional
- If all objects in the system are quiescently consistent then any execution of these objects is quiescently consistent.

Quiescent Consistency is Compositional

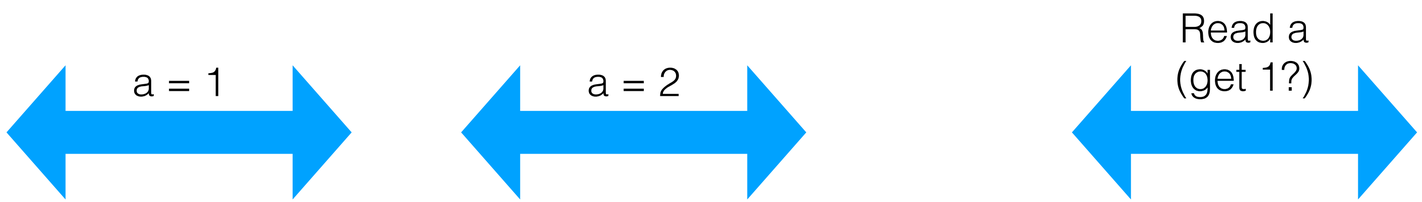
- Informally… Queue in pink, Stack in blue.



Program Order

- The order in which a single thread issues method calls is called *program order*
- We propose a new principle:

3. Method calls should appear to take effect in program order



Sequential Consistency

- Together with principle 1 we define sequential consistency:

1. Method calls should appear to happen in a one-at-a-time, sequential order

3. Method calls should appear to take effect in program order

Sequential Consistency

- Together with principle 1 we define sequential consistency:

1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

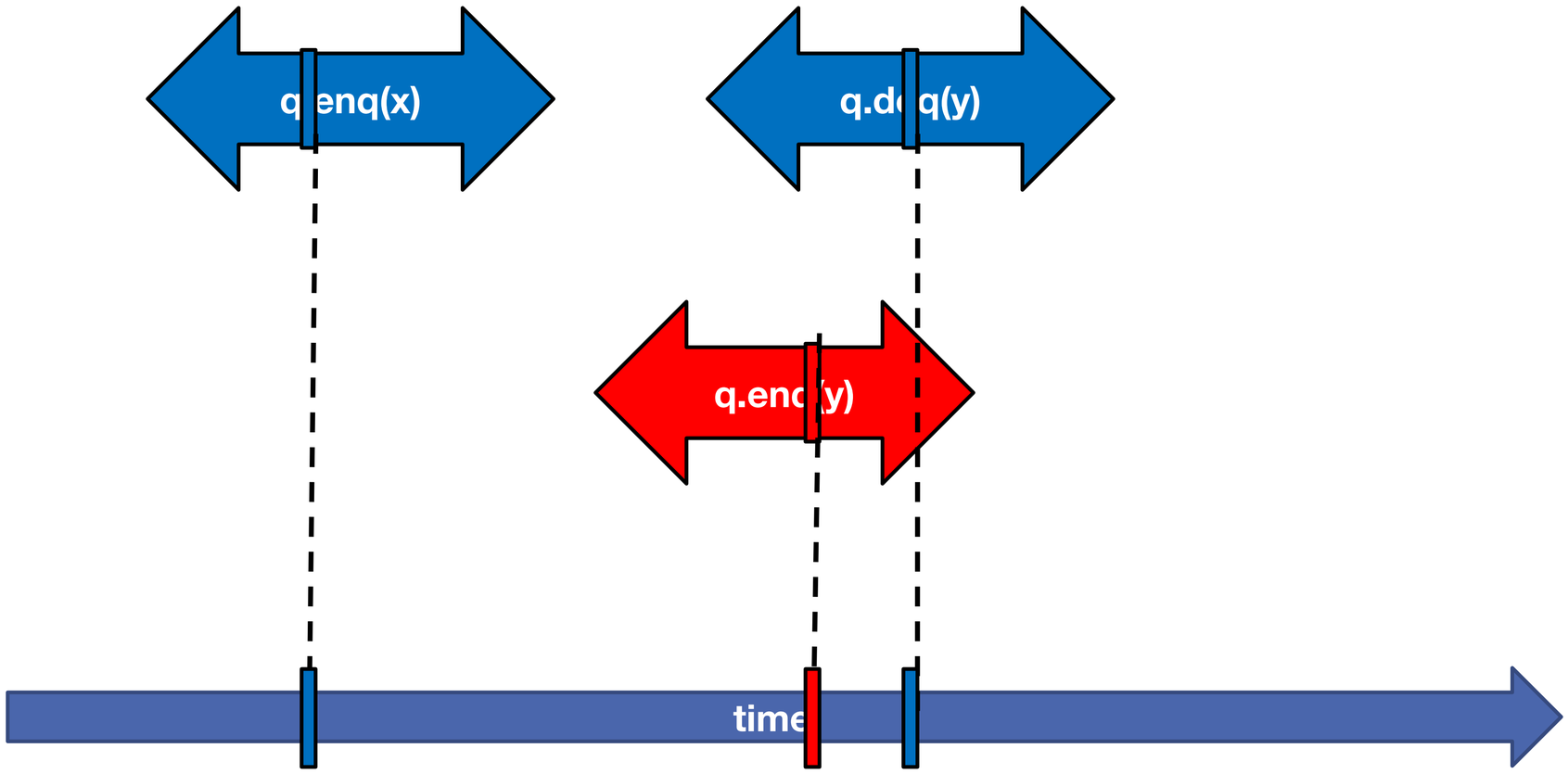
3. Method calls should appear to take effect in program order

- We say that an execution is sequentially consistent if it satisfies 1 and 3.
- We say that an object (data structure) implementation is sequentially consistent if any execution of its ,methods satisfy 1 and 3.

Example

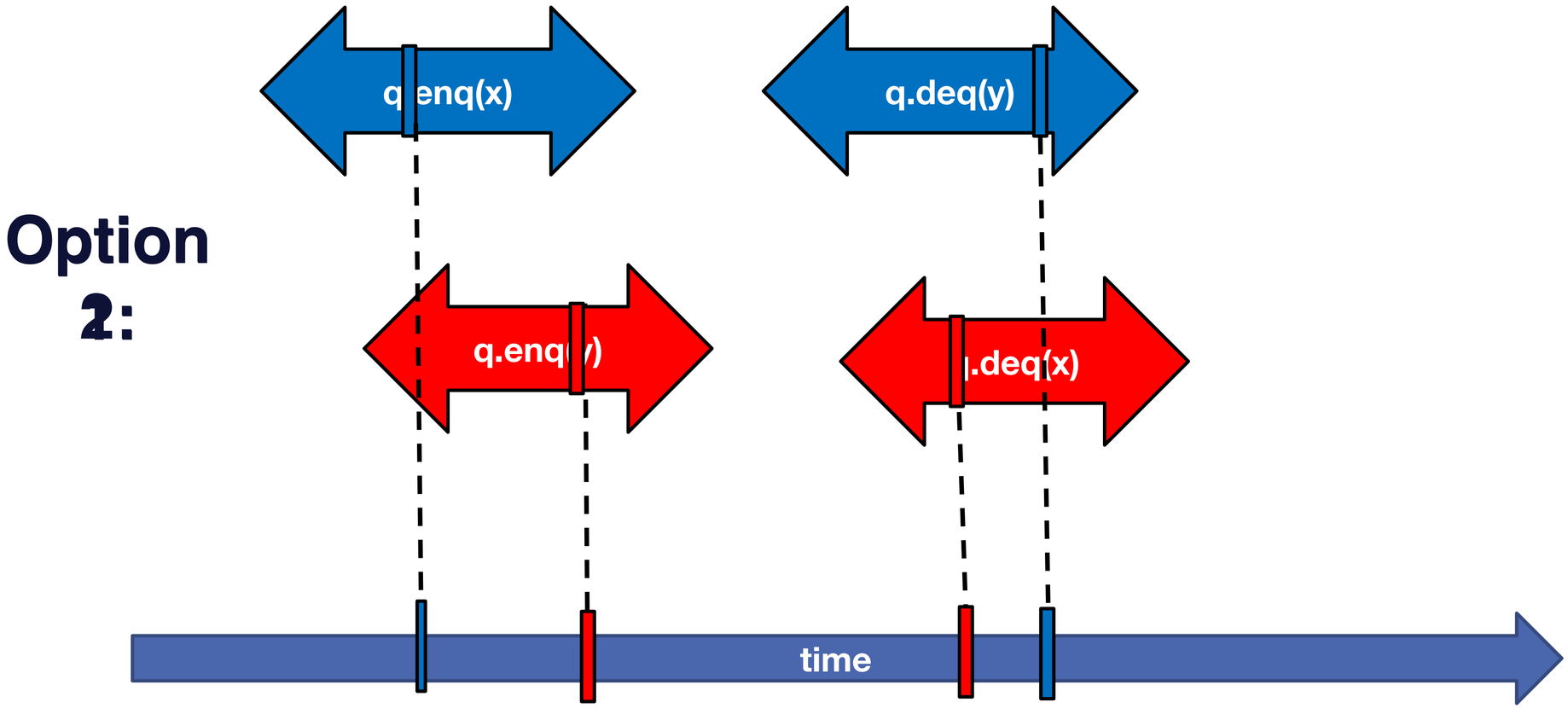
- Is this sequentially consistent?

Yes!



Ordering

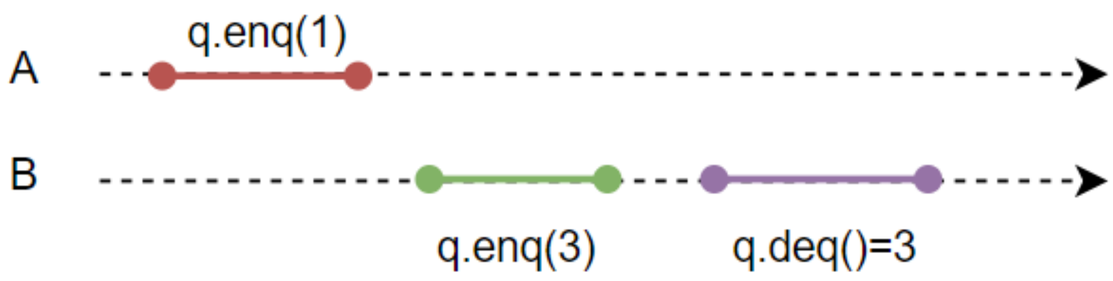
- Is the sequential order that fulfills the requirements unique?
- No, for example:



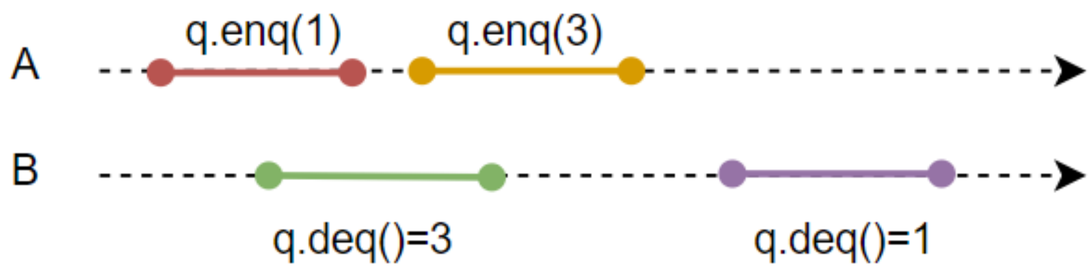
Sequential vs. Quiescent

- What is the relation between quiescent and sequential consistency?
- They're incomparable

Sequential but not quiescent consistent

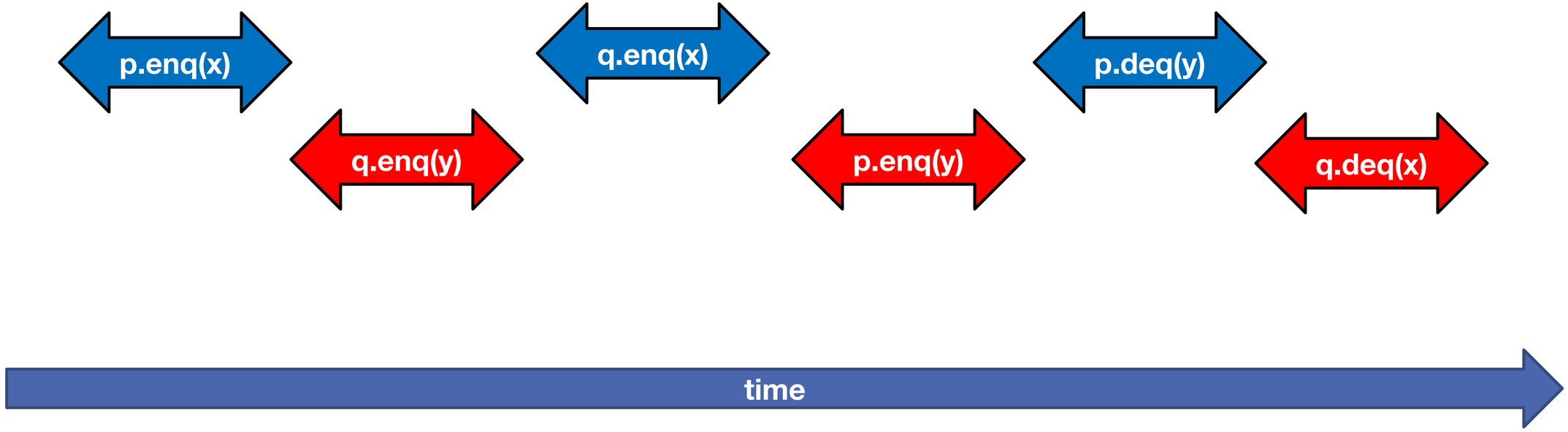


Quiescent but not sequential consistent



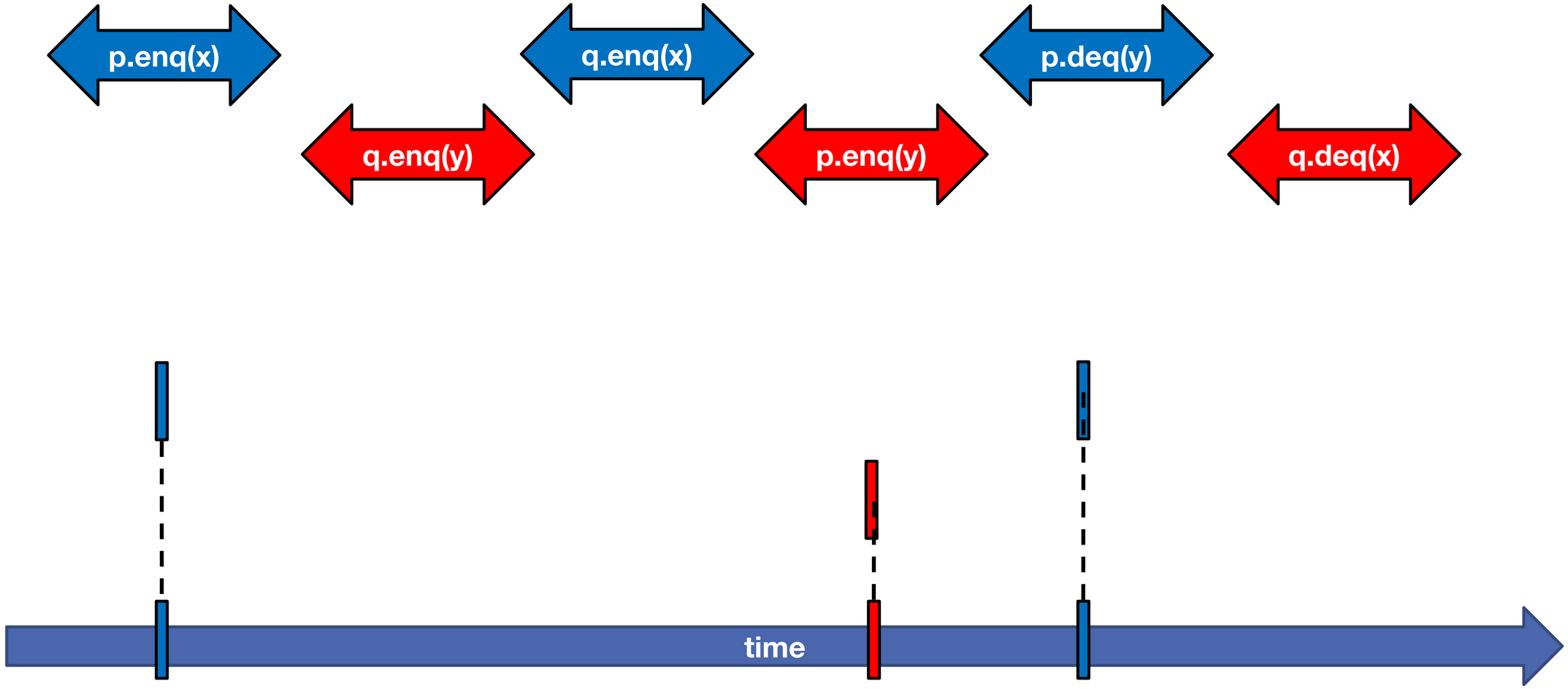
Sequential Consistency Compositional?

- Is sequential consistency compositional?
- No 😭 Let's see a FIFO queue example:



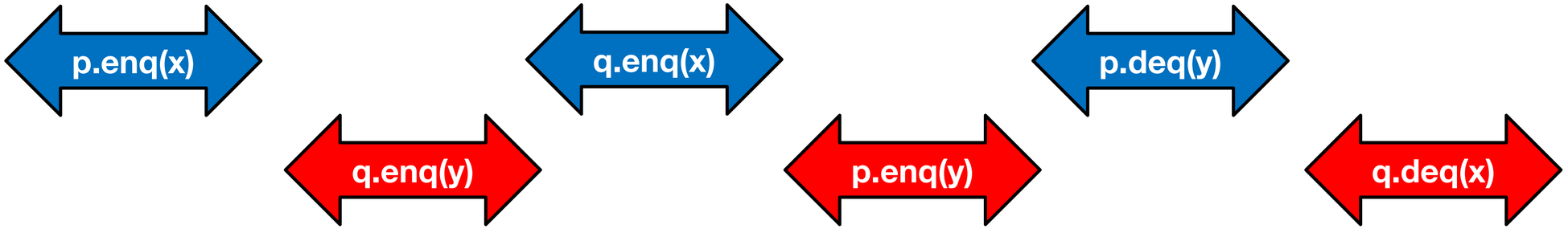
Is p sequentially consistent?

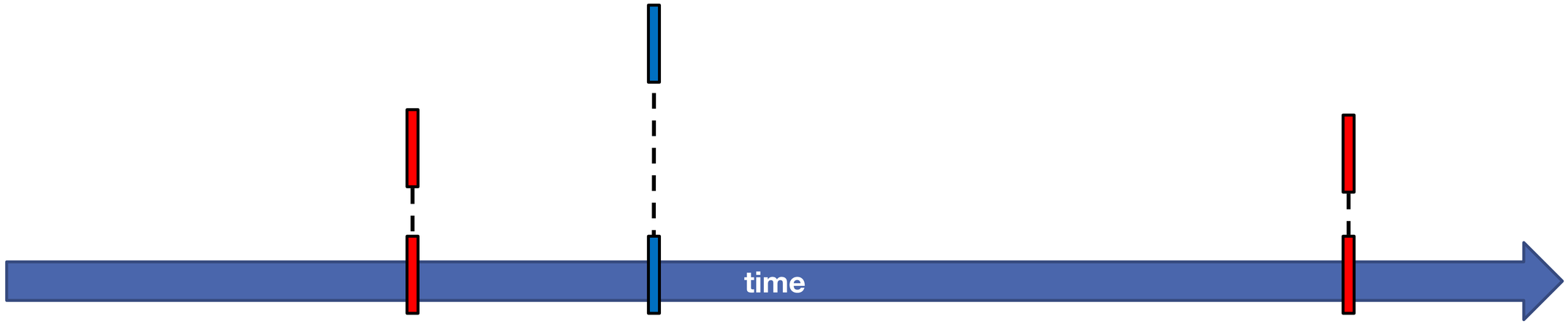
Yes!



Is q sequentially consistent?

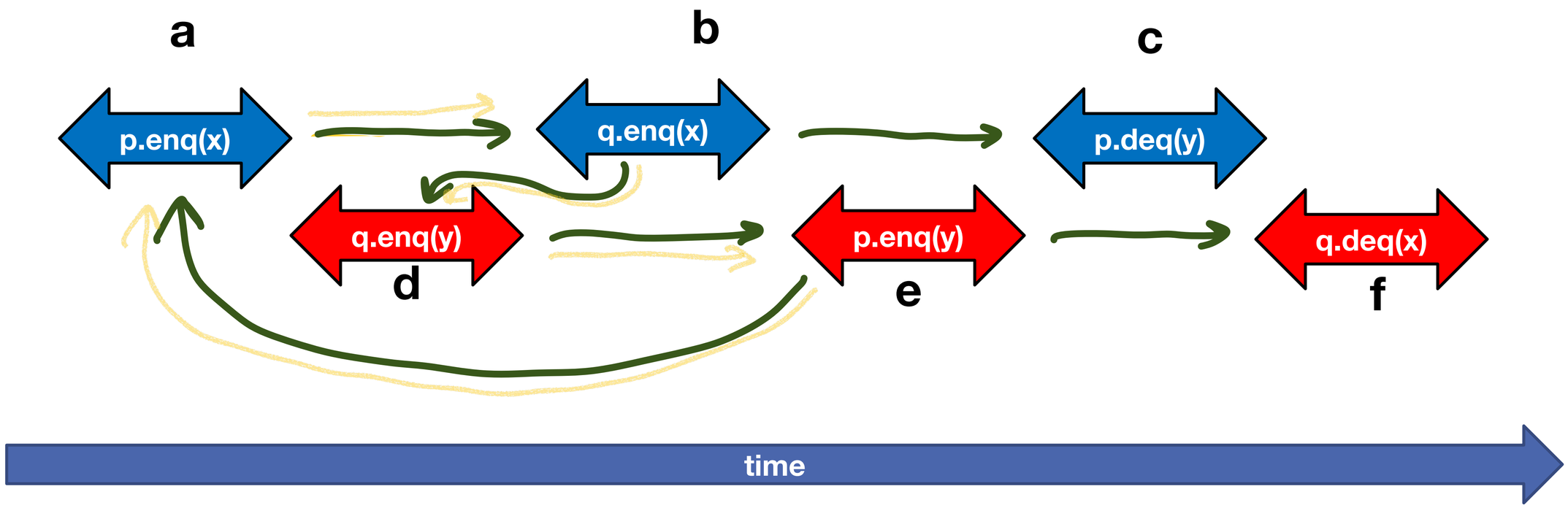
Yes!





Sequential Consistency

- Assume, by way of contradiction, these method can be reordered to form a correct execution, but in program order



Sequential Consistency

- Non-compositionality is a disadvantage
- Need to show for each set of objects that they work well together.

- Therefore, we strengthen the definition: linearizability

Recall Sequential Consistency

- We strengthen 3 —> 4

1. Method calls should appear to happen in a one-at-a-time, sequential order

3. Method calls should appear to take effect in program order

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

Program order:

Linearizability

- Replacing principle 3 with 4, we define linearizability:

1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

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

Linearizability $\Longrightarrow$ **Sequential Consistency**

Program order: $\longleftrightarrow$ $\longleftrightarrow$ $\longleftrightarrow$ $\longleftrightarrow$

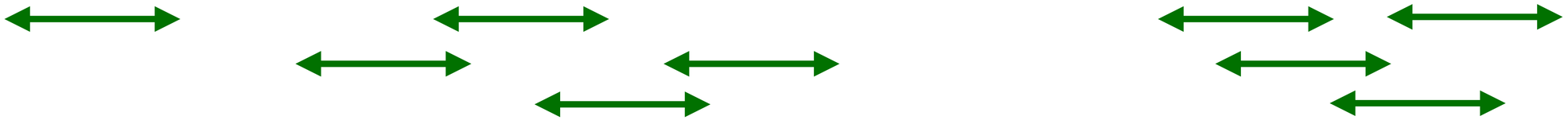
Linearizability

- Replacing principle 3 with 4, we define linearizability:

1. Method calls should appear to happen in a one-at-a-time, (correct) sequential order

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

Linearizability $\Longrightarrow$ Quiescent Consistency



Linearization Points

- Typically, to show linearizability, identify for each method a **linearization point** where the method takes effect.

Increment Counter:

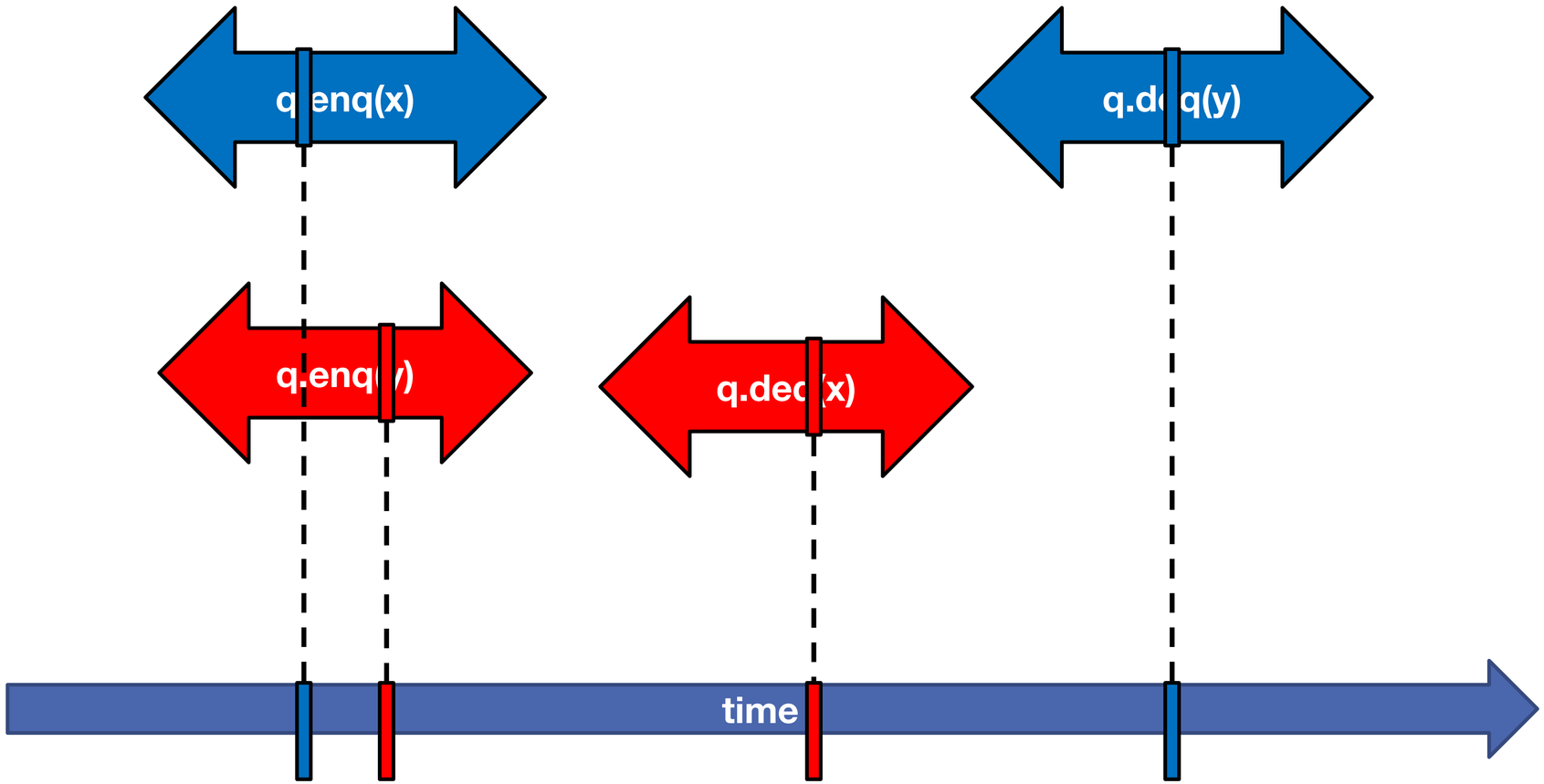
```
While(true){
  old = counter
  new = old++
  flag = CAS(&counter, old, new)
  if (flag) break;
}
```

Linearization point: a successful CAS here

Linearizability Examples

- Is this linearizable?

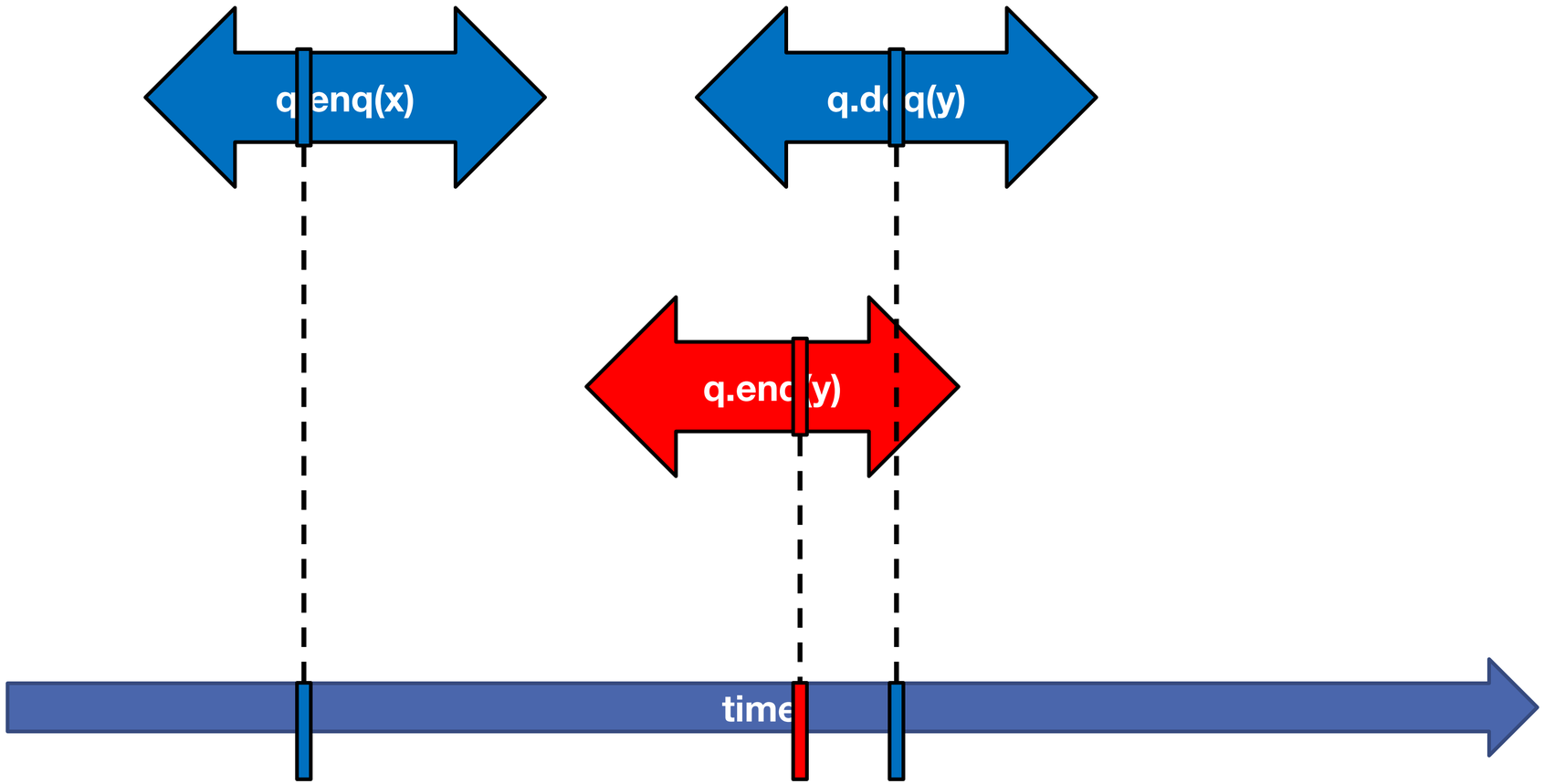
Yes!



Linearizability Examples

- Is this linearizable?

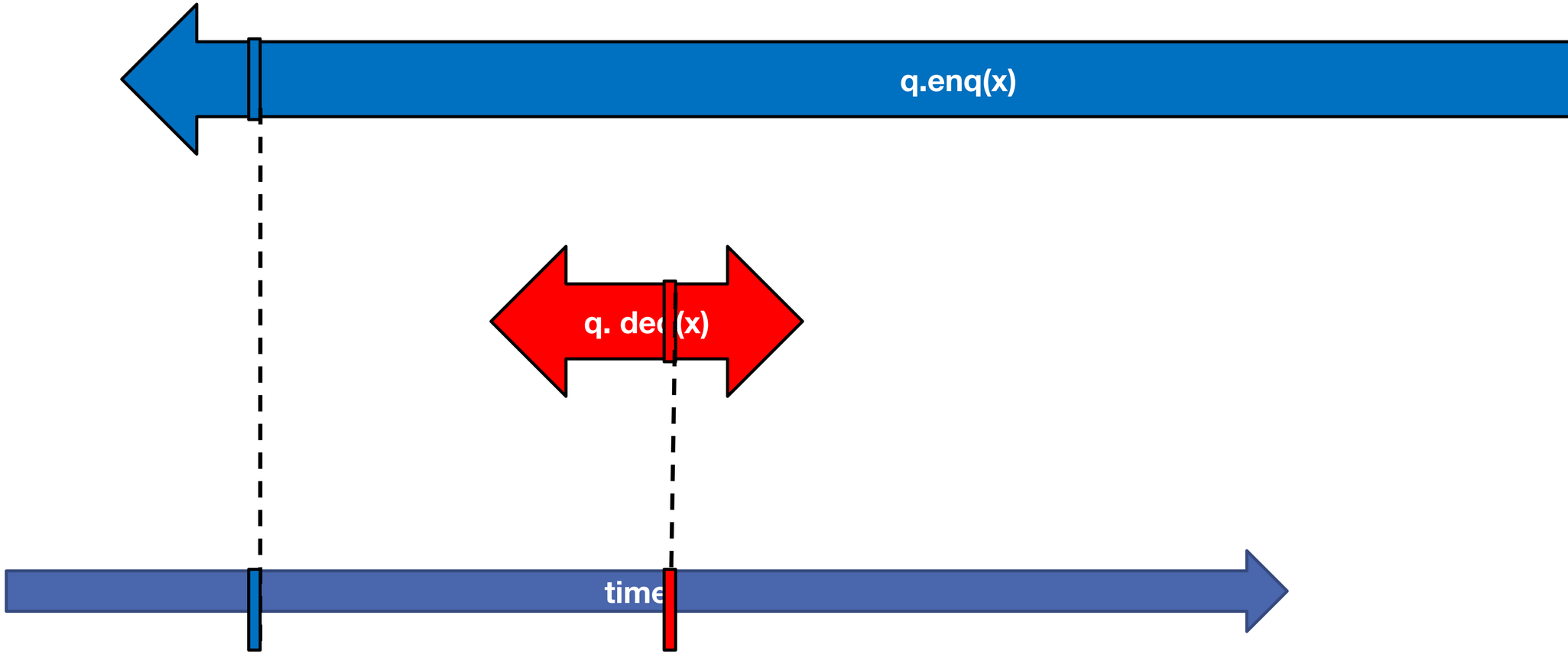
No!



Linearizability Examples

- Is this linearizable?

Yes!



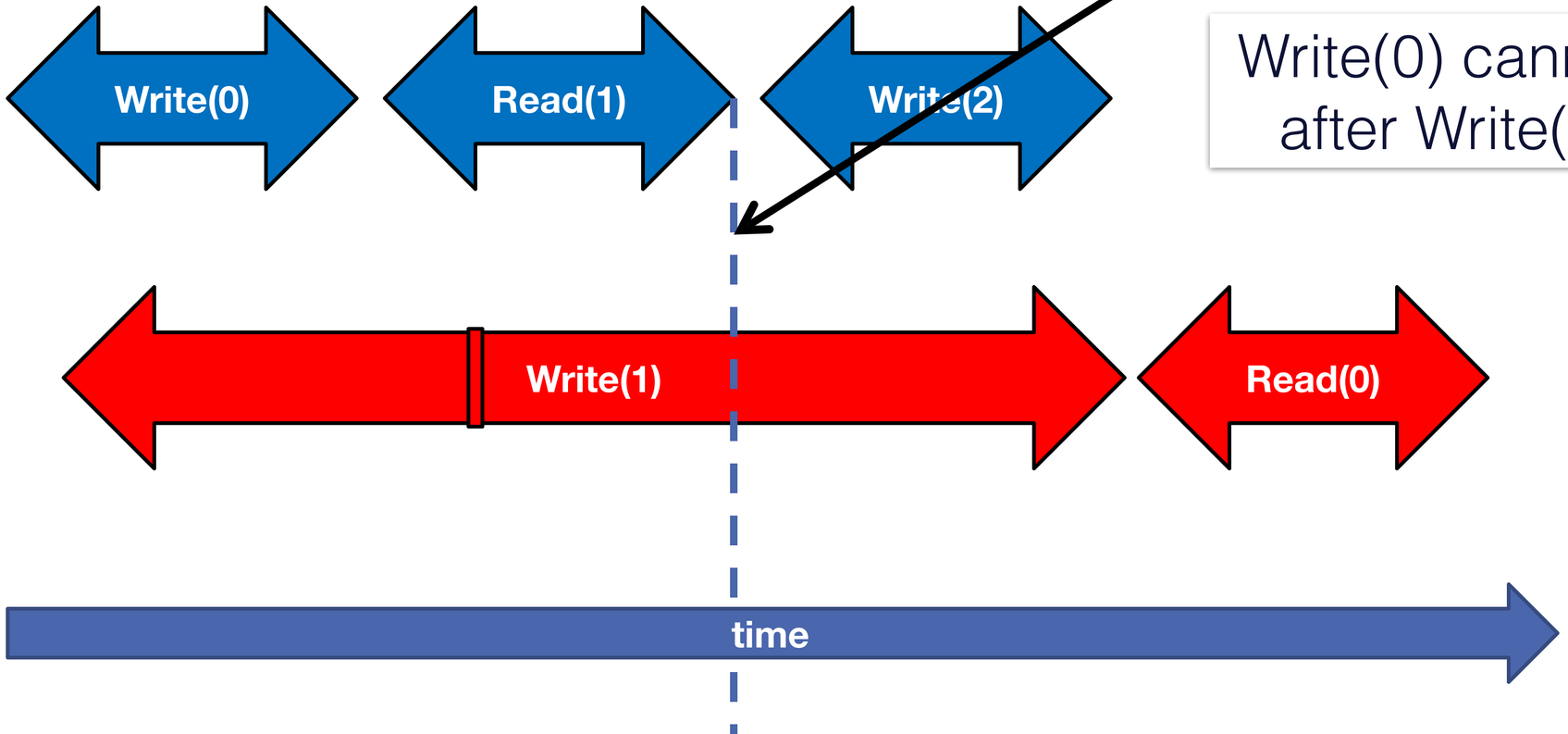
Linearizability Examples

- Is this linearizable?

No!

At this point, we know that write(1) has previously occurred

Write(0) cannot occur after Write(1) does

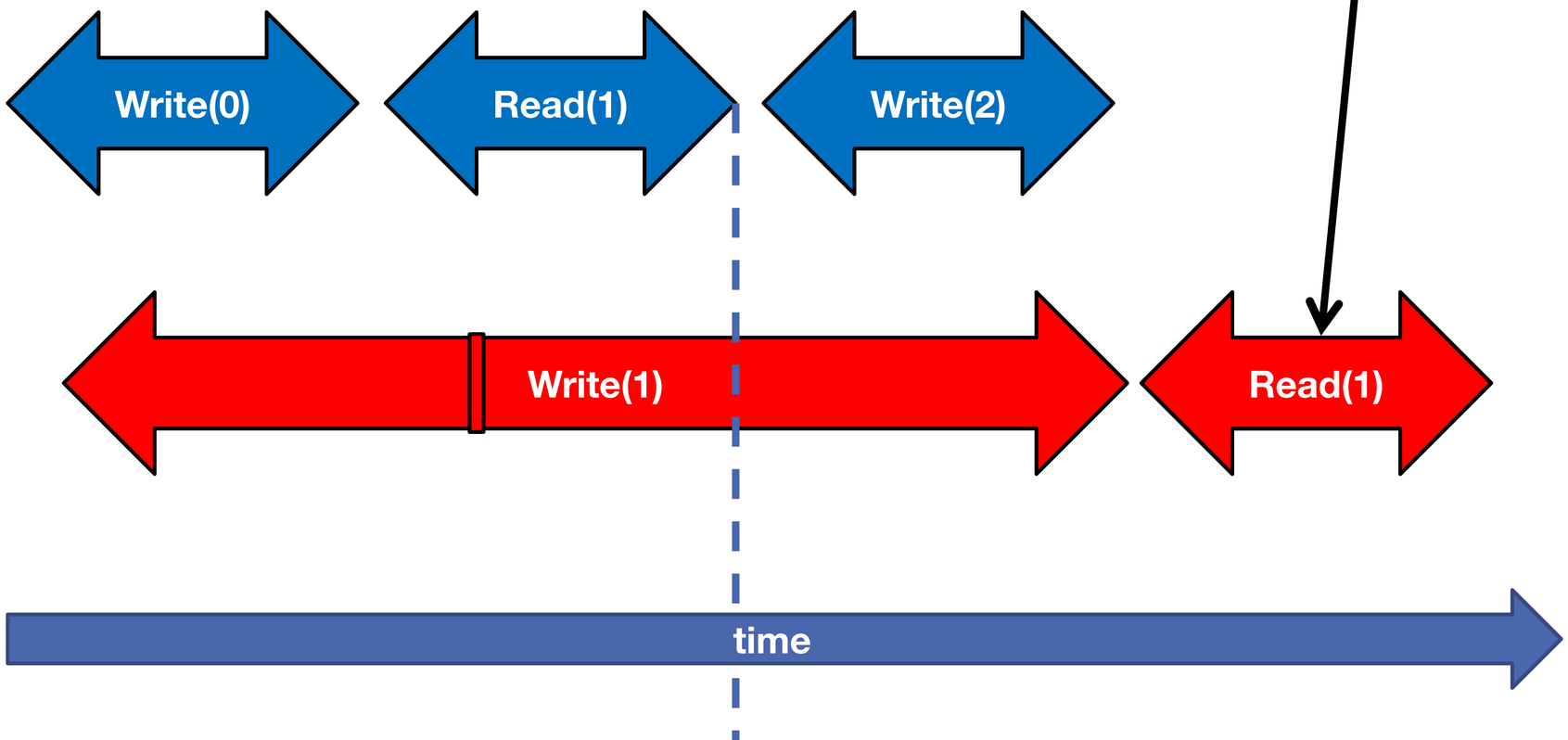


Linearizability Examples

- Is this linearizable?

No!

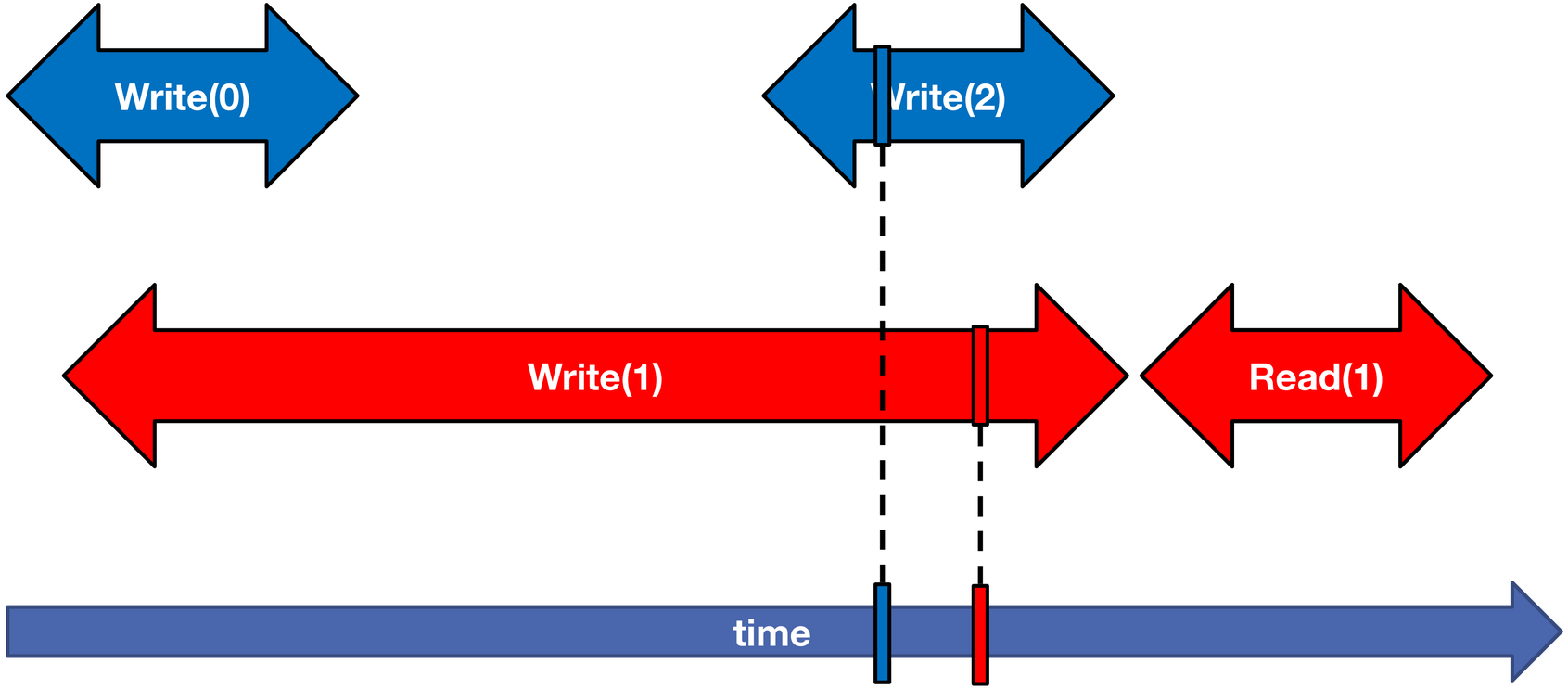
What if we change to Read(1)?



Linearizability Examples

- Is this linearizable?

Yes!



Linearizability is Compositional

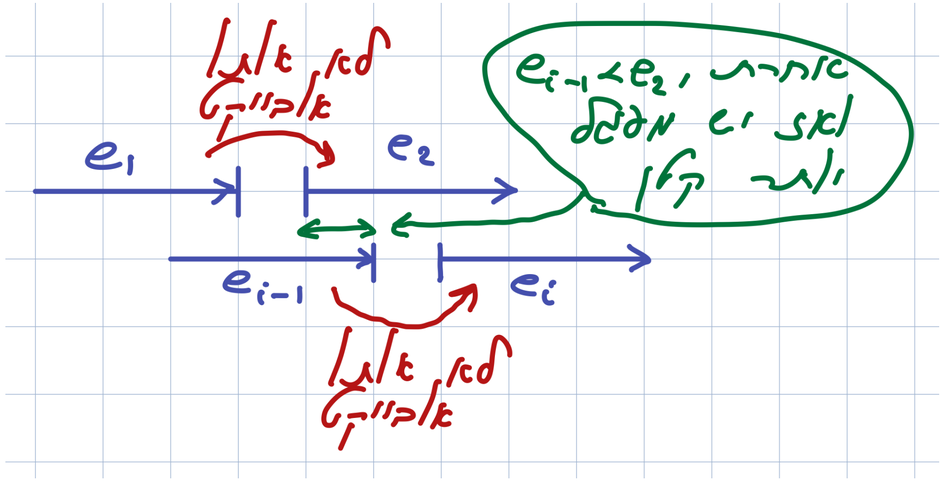
- Linearizability is compositional
- Why Does Composability Matter?
  - Modularity
  - Can prove linearizability of objects in isolation
  - Can compose independently-implemented objects

Proof Sketch

- Linearizability is compositional
- A history is linearizable $\Longleftrightarrow$ all sub-histories are linearizable
- $\Longrightarrow$ is obvious.
- $\Longleftarrow$: given a legal order on each object, build a legal order on everything.
- The order is a union of all objects orders plus all preceding relations.
- We need to show that there is no conflict, i.e., no cycle.
- Suppose in way of contradiction that there is a cycle $e_1, e_2, \ldots, e_n$

Proof Sketch

- The order is a union of the partial orders plus all preceding relations.
- Suppose in way of contradiction that there is a cycle $e_1, e_2, \ldots, e_n$.
- And let it be minimal
- They cannot all be of the same object, so let e_1 and e_2 relate to two different objects, and object x associate with e_1.
- Case 1: x appears again at $i \geq 3$.
- Let e_i be the first one related to x again. Then:
- Thus, e_1 precedes e_i, shorter cycle $e_1, e_i, \ldots, e_n$



Proof Sketch

- The order is a union of the partial orders plus all preceding relations
- Suppose in way of contradiction that there is a cycle $e_1, e_2, \ldots, e_n$
- And let it be minimal
- They cannot all be of the same object, so let e_1 and e_2 relate to two different objects, and object x associate with e_1.
- Case 1: x appears again at $i \geq 3$: shorter cycle $e_1, e_i, \ldots, e_n$
- Case 2: x does is not used again
- e_1 precedes e_2, e_n precedes e_1 implies e_n precedes e_2, shorter cycle $e_2, \ldots, e_n$.

Linearizability

- Linearizability today is the de-facto correctness property for concurrent programs

👻 Liveness 👻

Progress (Liveness): a good thing eventually happens

Counter Example

T1	T2
local := counter	local := counter
local++	local++
counter := local	counter := local

concurrent execution is not safe !

Use a Lock

Lock (L)

local := counter

local++

counter := local

Unlock(L)

Synchronize:
Only one thread can
acquire a lock

Compare and Swap (CAS)

- CAS (addr, expected, new)

Atomically:

```
If ( MEM[addr] == expected ) {
    MEM[addr] = new
    return (TRUE)
    }
else return (FALSE)
```

Use a Lock or a CAS

```
Lock (L)
local := counter
local++
counter := local
Unlock(L)
```

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Synchronize:

Only one thread can acquire a lock

Synchronize:

Only one thread changes from old in a concurrent CAS.

Use a Lock or a CAS

```
Lock (L)
local := counter
local++
counter := local
Unlock(L)
```

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Issues:
Efficiency, scalability, Fairness, progress guarantee, design complexity.

What Does Lock-Free Mean?

- First try: “never using a lock”.
- Well, is this using a lock? (word is initially zero.)

```
while (!CAS(&word,0,1)) {}
local := counter
local++
counter := local
word := 0
```

It is not easy to say if something is a “lock”.

What Does Lock-Free Mean?

- “No matter which interleaving is scheduled, my program will make progress.”.
- A.k.a. non-blocking.
- Our second example is lock-free in this sense.

```
START:
old := counter
new := old ++
if ( ! CAS(&counter,old,new) )
        goto START
```

Progress Guarantee

Lock-Freedom
If you schedule enough steps across all threads, one of them will make progress.

- Realistic (though difficult):
  Various lock-free data structures exist in the literature (stack, queue, hashing, skiplist, trees, etc.).

- Advantages:
  Worst-case responsiveness, scalability, no deadlocks, no livelocks, added robustness to threads fail-stop.

Observations

- Can progress be guaranteed when using locks ?
  - No !
  - Thread T_1 can hold a lock and not be scheduled while T_2 waits for the lock.

Observations

- Can progress be guaranteed when using locks ?
  - No !
- Is progress guaranteed when we do not employ locks?
  - No !
  - Computation can be blocked by fine-grained synchronization.

```
While ( ! CAS(&word,0,1) ) ;
    Work on shared space;
    word= 0;
```

Wait-Free and Lock-Free

An object is **wait-free** if **every** call completes in a finite number of steps

- Any thread must make progress; robust to worst-case scheduling
- Very strong, and harder to obtain.

An object is **lock-free** if **some** call finishes in a finite number of steps

- One thread must make progress; robust to worst-case scheduling
  - Other threads may starve
- Many lock-free algorithms exist and used in real systems

Wait-Free Implies Lock-Free

An object is **wait-free** if every call completes in a finite number of steps

An object is **lock-free** if some call finishes in a finite number of steps

Wait-free $\Longrightarrow$ Lock-free

(If all threads make progress then one thread makes progress)

An Example

An object is **wait-free** if every call completes in a finite number of steps

An object is **lock-free** if some call finishes in a finite number of steps

Is our counter Increment method lock-free or wait-free?

Increment Counter:

```
While(true) {
  old = counter
  new = old++
  flag = CAS(&counter, old, new)
  if (flag) break;
}
```

Liveness Properties

A lock-based queue:

```
public void enq(T x) throws FullException {
  lock.lock();
  try {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
     tail++;
  }  finally {
    lock.unlock();
  }
}
```

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Liveness Properties

No locks queue (not linearizable):

```
public void enq(T x) throws FullException {
  if (tail - head == items.length)
    throw new FullException();
  items[tail % items.length] = x;
  tail++;
}
```

```
public T deq() throws EmptyException {
  if (tail - head == 0)
    throw new EmptyException();
  T x = items[head % items.length];
  head++;
  return x;
}
```

Wait free!

Obstruction Freedom

- A method call executes in isolation if no other threads take steps
- The weakest progress guarantee definition:

An object is obstruction-free if, from any execution point, any thread that starts to execute in isolation, completes in a finite number of steps

Progress Guarantees

Great guarantee, but difficult to achieve.

Wait-Freedom

If you schedule enough steps of *any thread*, *it* will make progress.

Lock-Freedom

If you schedule enough steps across *all threads*, one of them will make progress.

Widely used

Obstruction-Freedom

If you let *any thread* run alone for enough steps, it will make progress.

Somewhat weak.

Summary

- Linearizability is the accepted correctness criteria
- Weaker criteria: sequential consistency, quiescent consistency

- Progress guarantees (liveness):
  - Lock-free is the widely used progress guarantee
  - Wait-free is stronger, obstruction-free is weaker