Concurrency: Foundations and Algorithms

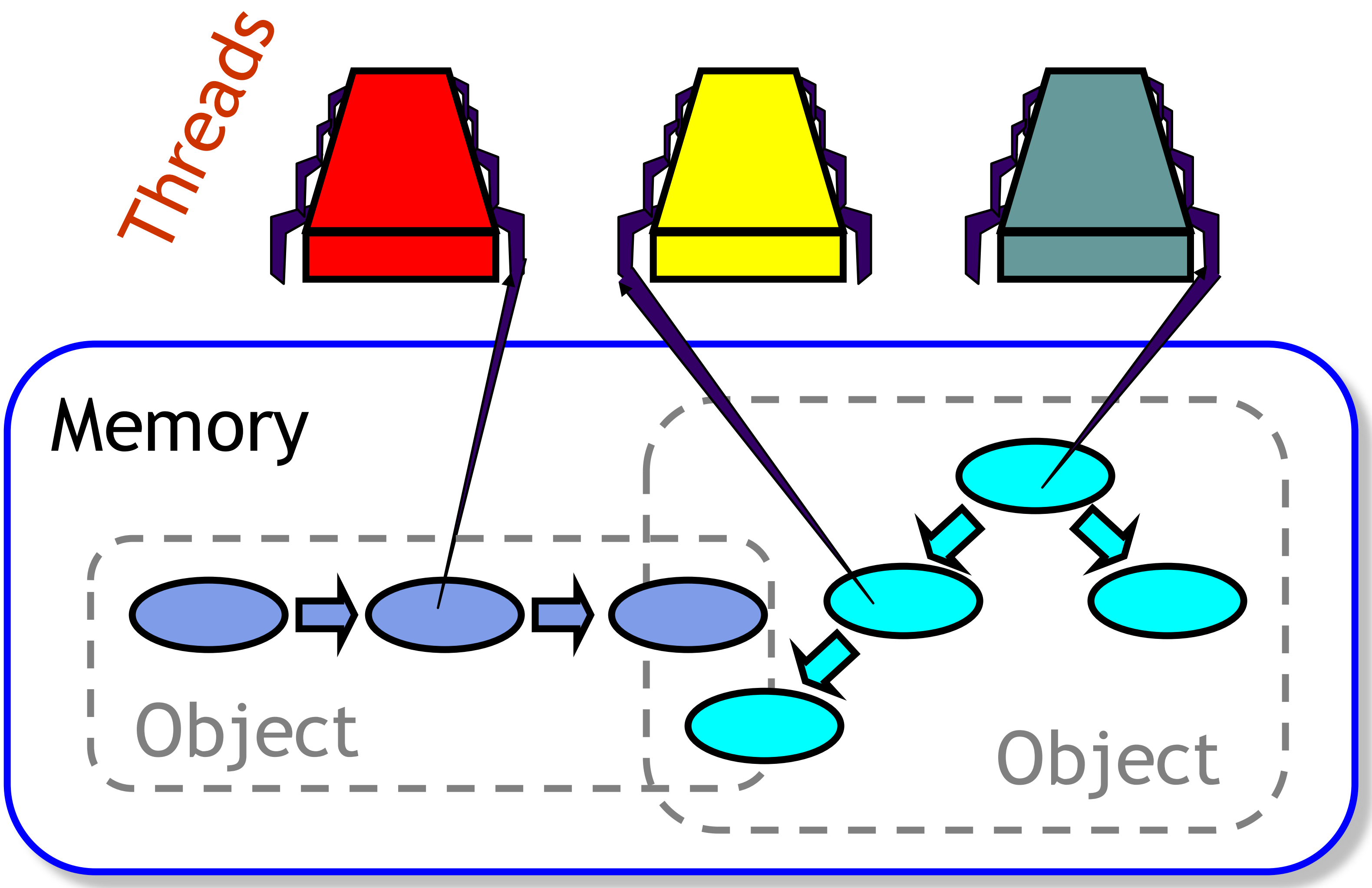
Concurrent Objects and Consistency

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

Based on slides by ***Maurice Herlihy*** *and* ***Nir Shavit***

Concurrent Computation

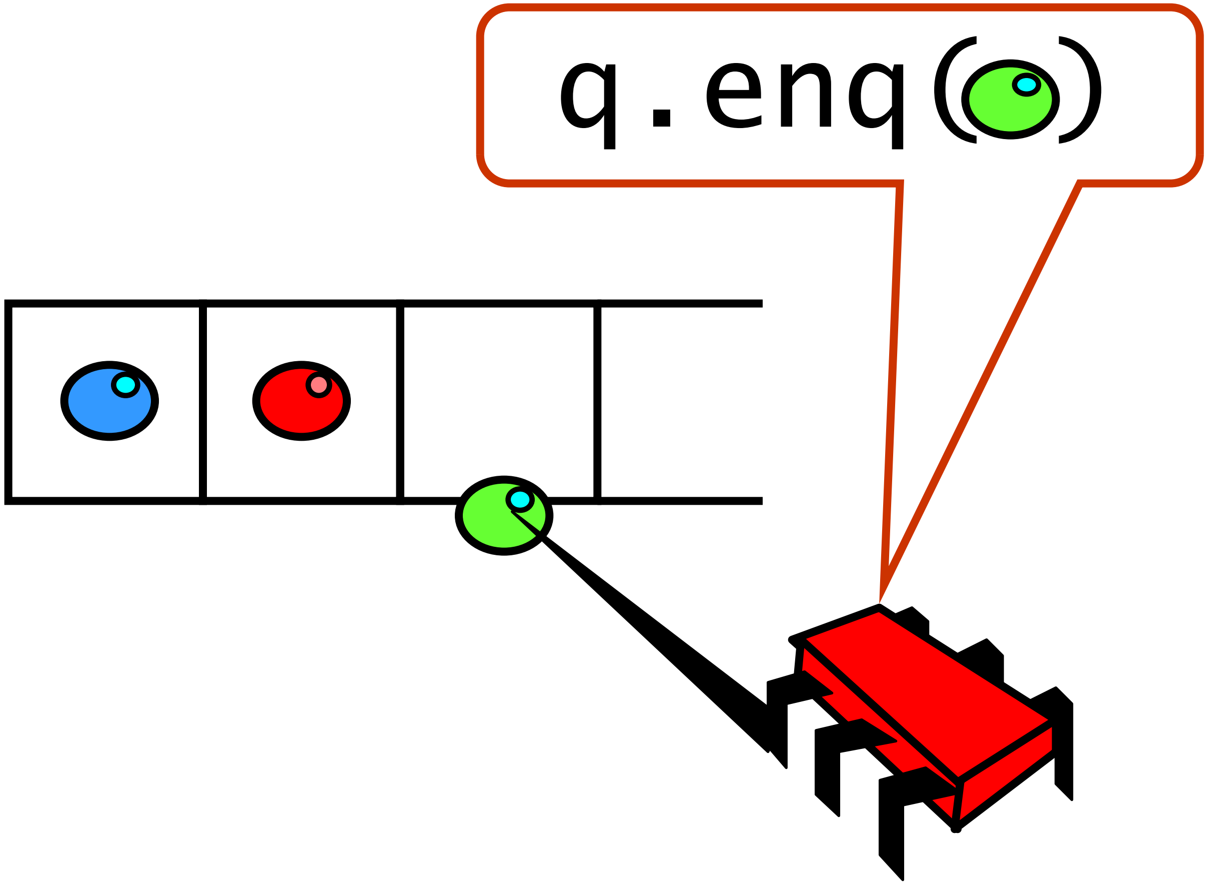


Objectivism

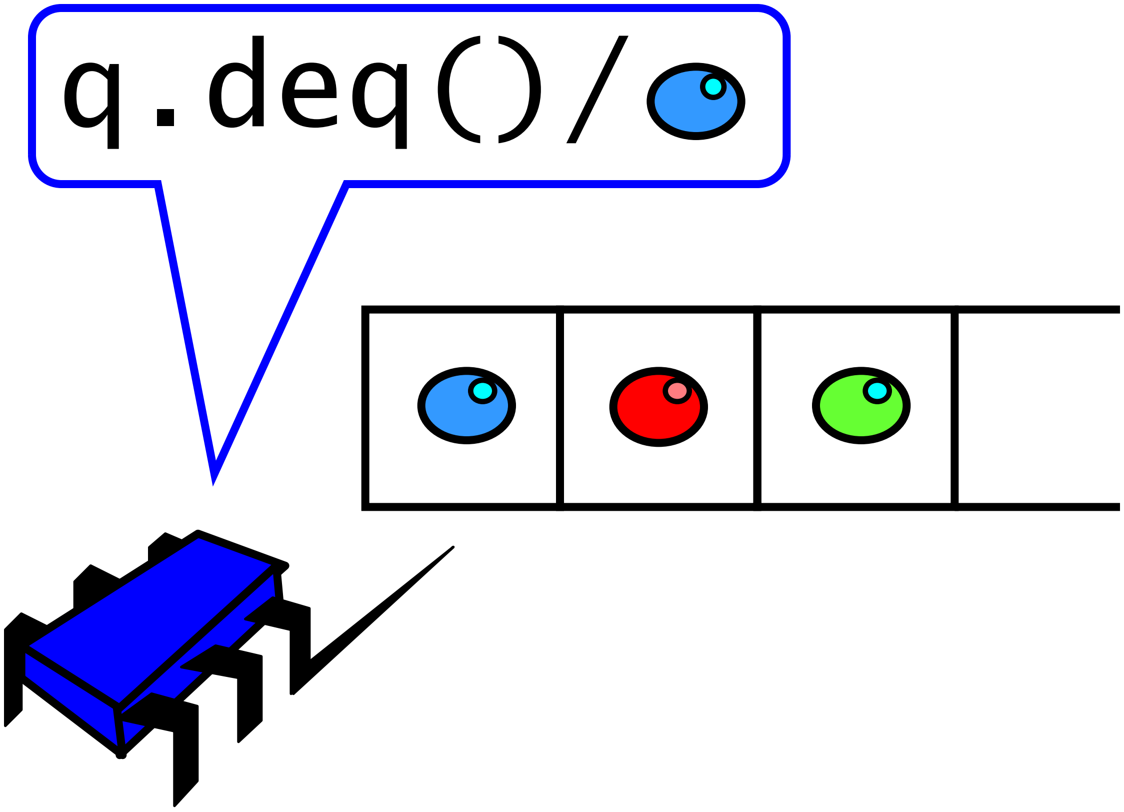
- What is a concurrent object?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we are right**?

This lecture

FIFO Queue: Enqueue Method

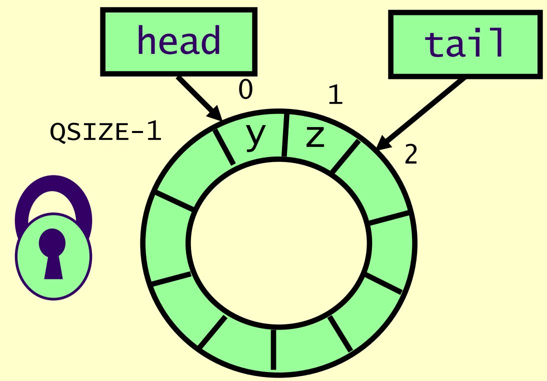


FIFO Queue: Dequeue Method



Implementation: enq()

```
public class Queue {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized
  void enq(Object x) {
    while (tail – head == QSIZE)
      this.wait();
    items[tail % QSIZE] = x;
    tail++;
    this.notifyAll();
  } …
}
```



Method execution is mutually exclusive

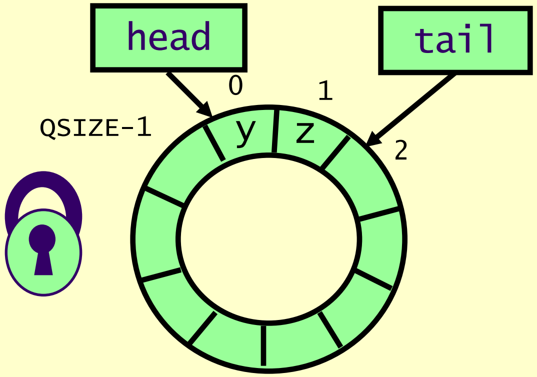
Release lock if need to wait

Here is where the queue is actually updated!

Notify all others that you release the lock

Implementation: deq()

```
public class Queue {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized
  Object deq() {
    while (tail – head == 0)
      this.wait();
    Object x = items[head % QSIZE];
    head++;
    this.notifyAll();
    return x;
  } …
}
```



We understand it is "correct" because all modifications are mutually exclusive…

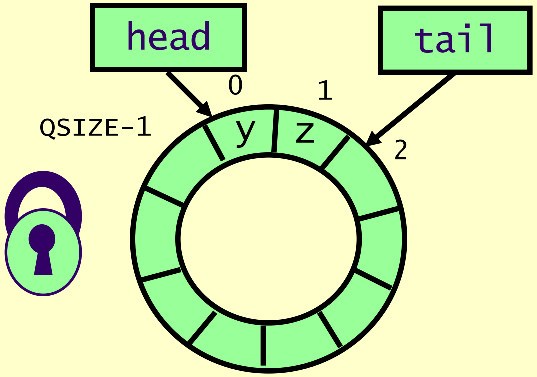
A Concurrent Implementation

- Now consider the following implementation
  - The same thing without mutual exclusion

- For simplicity, **only two threads:**
  - One thread only calls **`enq()`**
  - The other only **`deq()`**

Lock-free 2-Thread Queue

```
public class LockFreeQueue {
  volatile int head = 0,
               tail = 0;
  Object[QSIZE] items;
  public void enq(Item x) {
    while (tail-head == QSIZE); // busy-wait
    items[tail % QSIZE] = x; tail++;
  }
  public Item deq() {
    while (tail == head); // busy-wait
    Item item = items[head % QSIZE]; head++;
    return item;
  }
}
```



Queue updated without lock

How do we define "correct" when modifications are not exclusive?

Defining "Correct"

- Need a way to **specify** a concurrent queue object
- Need a way to prove that an algorithm **implements** the object's specification

- Lets talk about object specifications…

Sequential Objects

- Each object has a **state**
  - Usually given by a set of *fields*
  - Queue example: sequence of items
- Each object has a set of **methods**
  - Only way to manipulate state
  - Queue example: **`enq()`** and **`deq()`** methods

Sequential Specifications

- If (precondition)…
  - The object is in such-and-such a state…
  - Before you call the method…
- Then (postcondition)…
  - The method will return a particular value…
  - Or throw a particular exception…
- And (postcondition, con't)…
  - The object will be in some other state…
  - When the method returns

Pre- and Postconditions: deq()

- Precondition
  - Queue is non-empty
- Postcondition
  - Returns first item in queue
- Postcondition
  - Removes first item in queue

Pre- and Postconditions: deq()

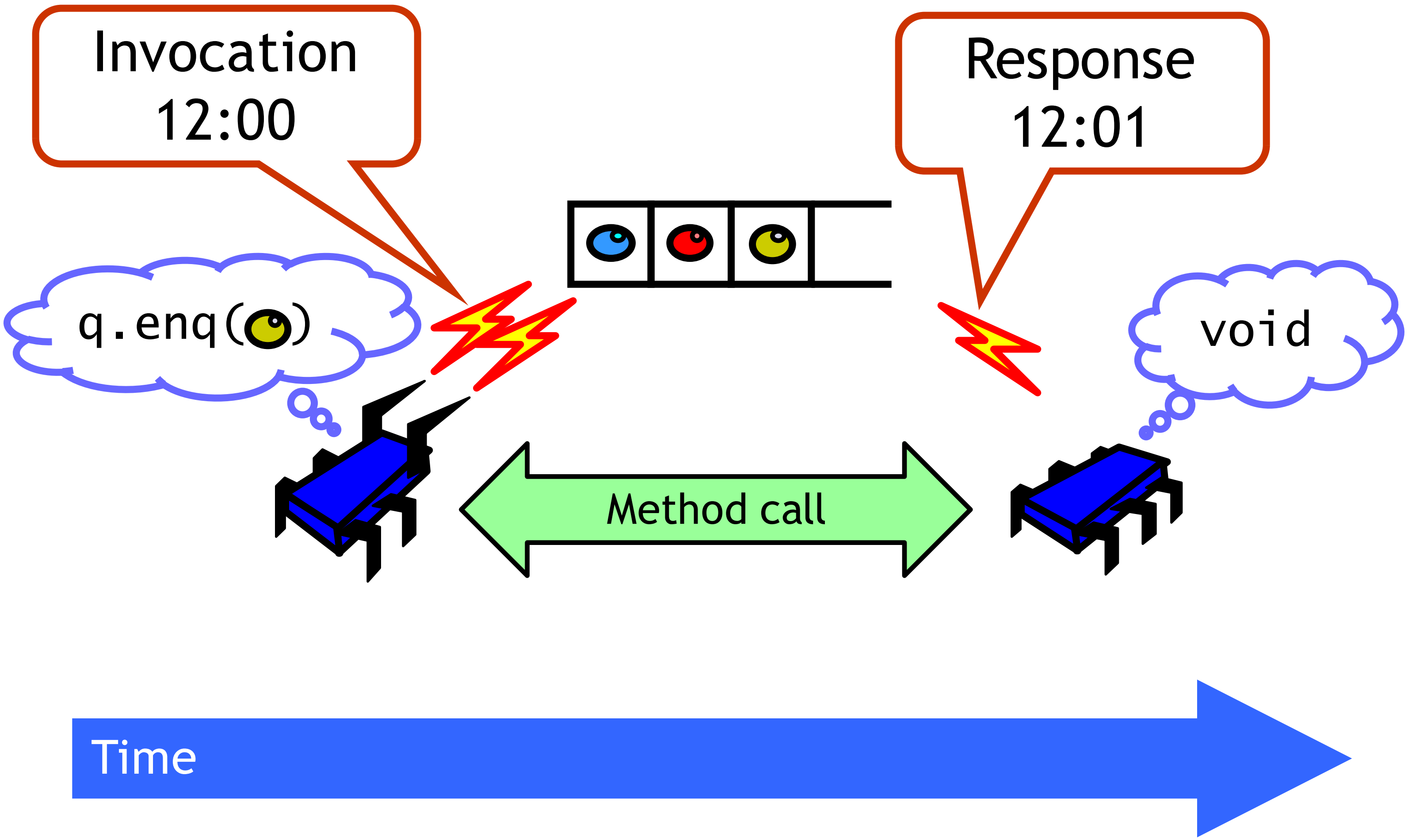
- Precondition
  - Queue is empty
- Postcondition
  - Throws `Empty` exception
- Postcondition
  - Queue state unchanged

Why Sequential Specifications Totally Rock

- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
- Documentation size linear in # of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

- What About Concurrent Specifications?
  - Methods? Documentation? Adding new methods?

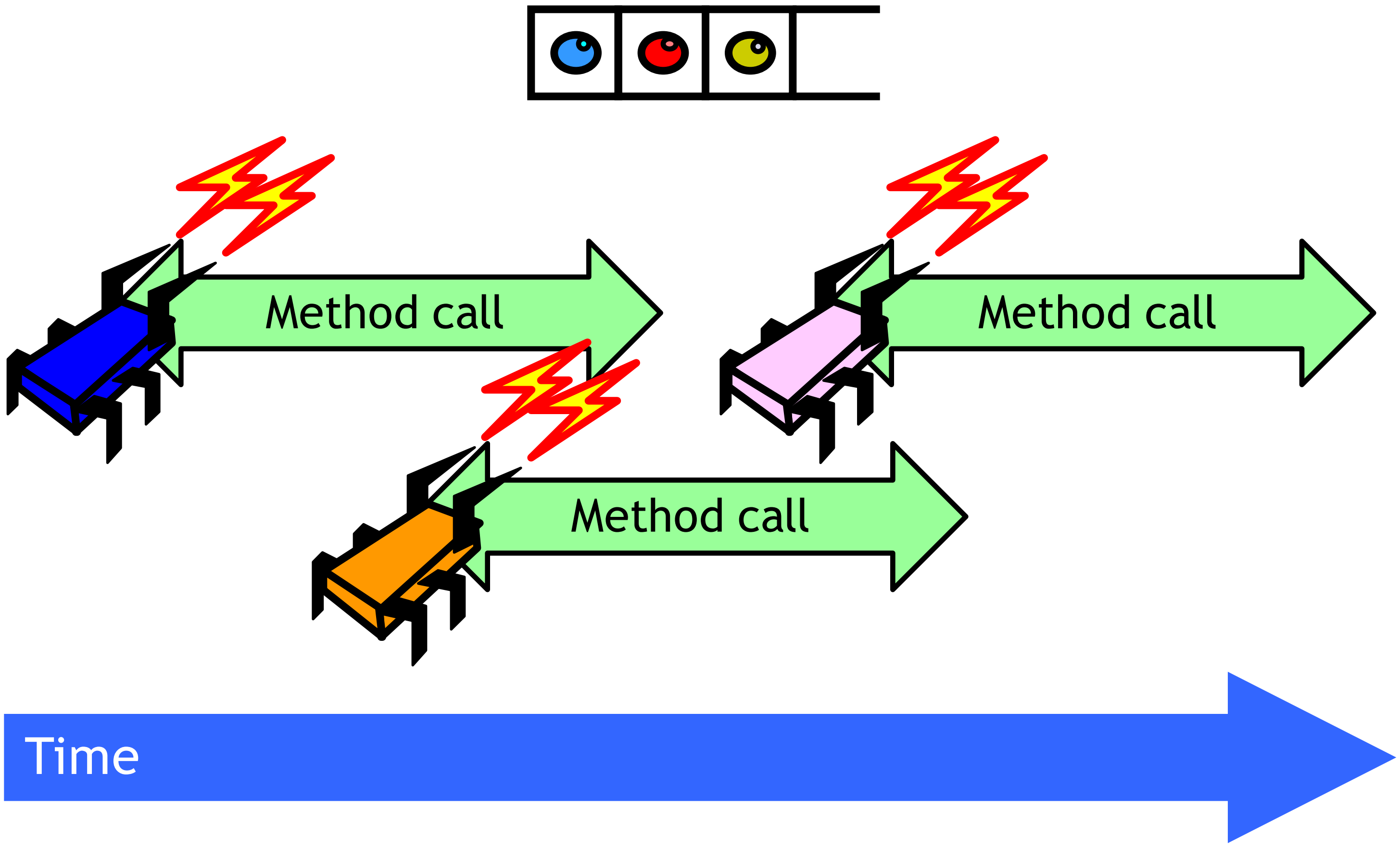
Methods Take Time



Sequential vs. Concurrent

- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval

Concurrent Methods Take Overlapping Time



Sequential vs. Concurrent

Sequential

- Object needs meaningful state only between method calls
- Each method described in isolation
- Can add new methods without affecting older methods

Concurrent

- Because method calls overlap, object might **never** be between method calls
- Must characterize **all** possible interactions with concurrent calls
  - What if 2 **enq()** overlap? Or 2 **deq()**? Or **enq()** and **deq()**?
- Everything can potentially interact with **everything else**

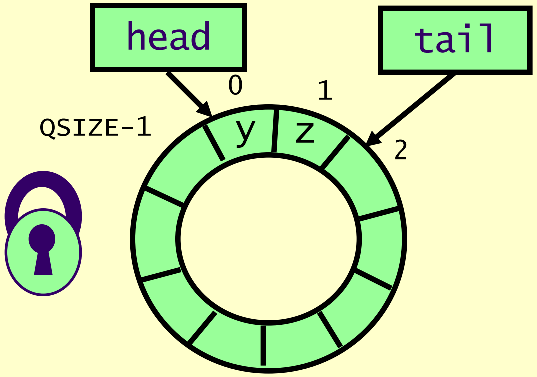
PANIC!

The Big Question

- 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

Intuitively…

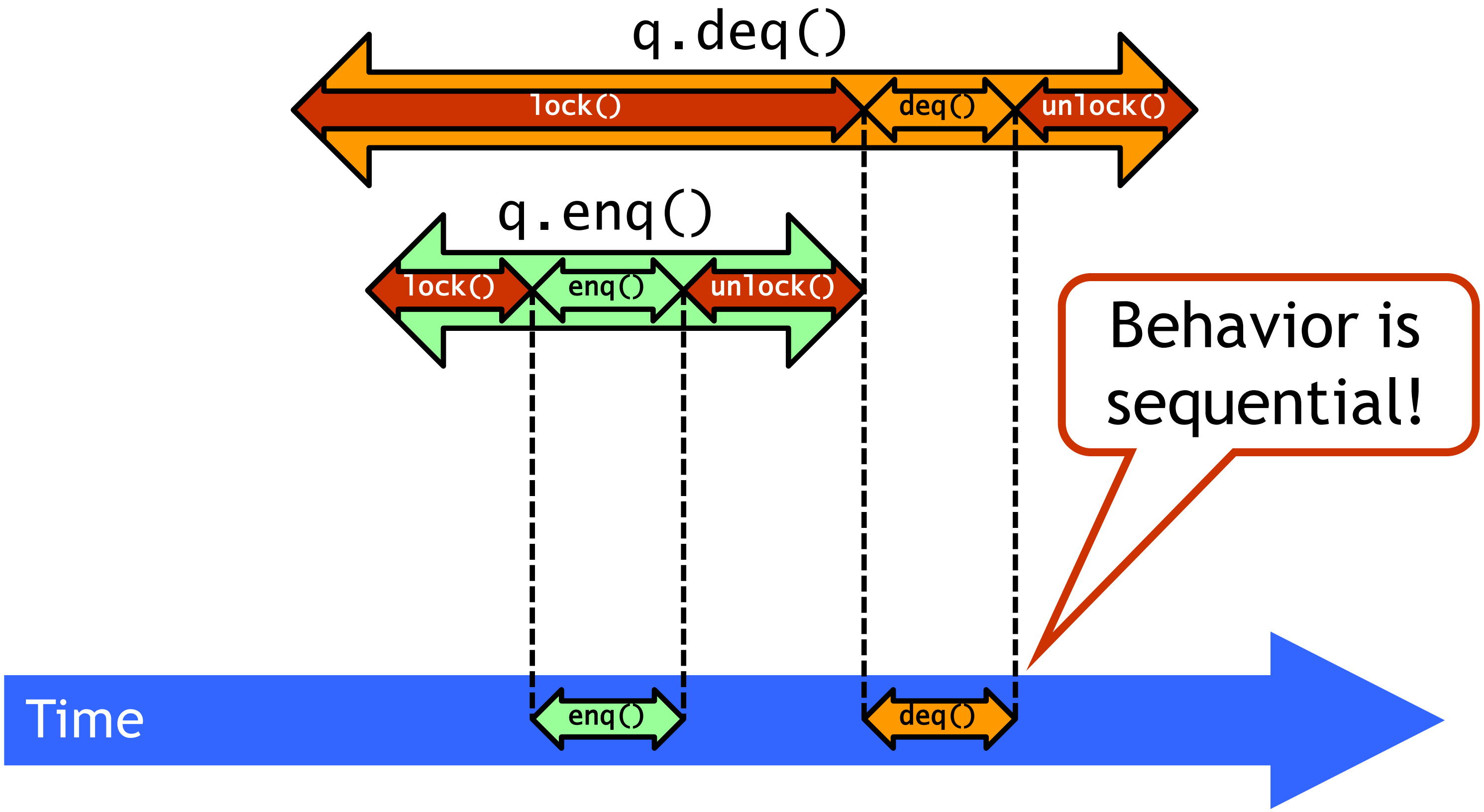
```
public class Queue {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized
  void enq(Object x) {
    while (tail – head == QSIZE)
      this.wait();
    items[tail % QSIZE] = x;
    tail++;
    this.notifyAll();
  } …
}
```



Queue is updated while holding lock (mutually exclusive)

Intuitively…

Let's capture the idea of describing the concurrent via the sequential



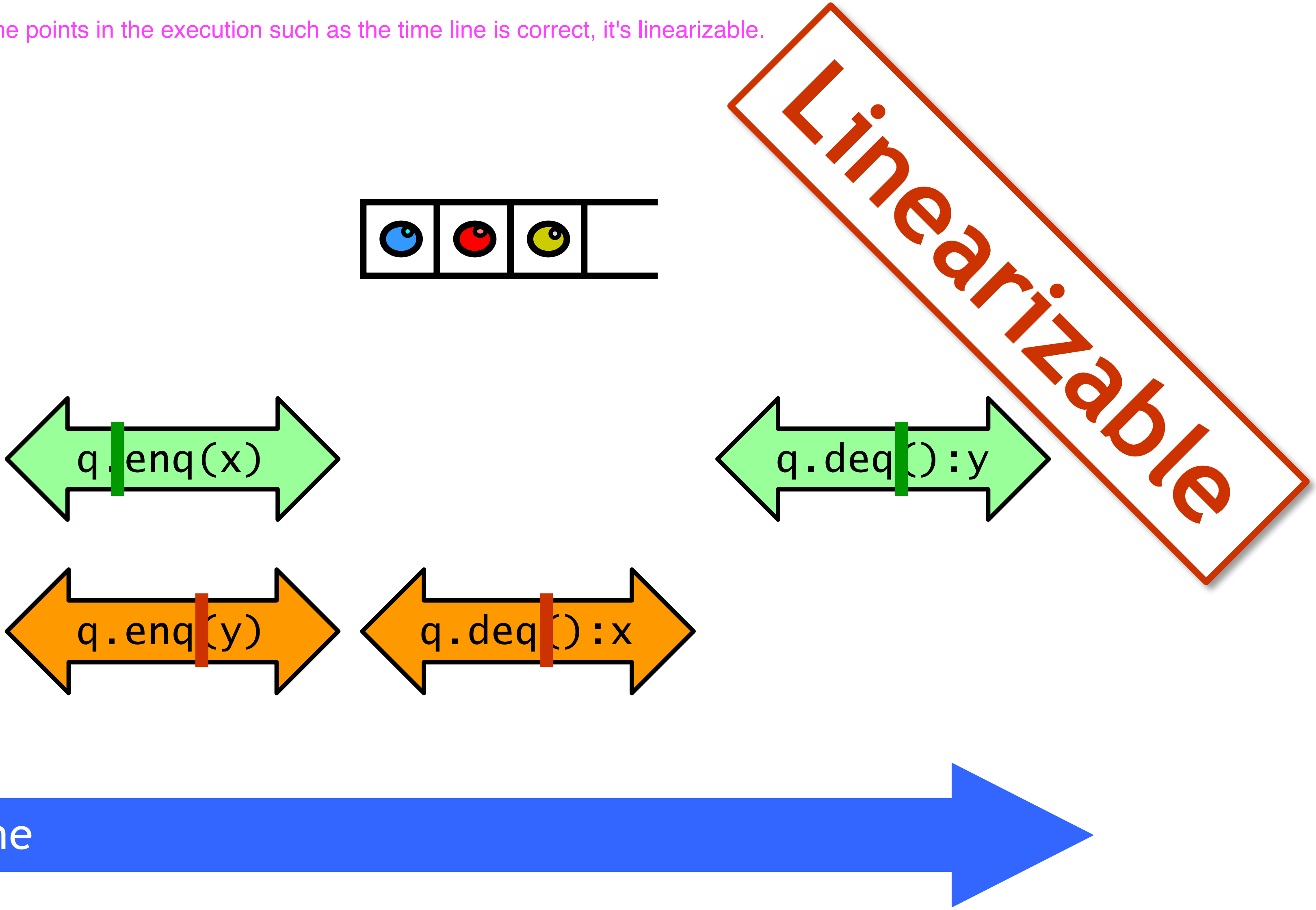
Linearizability

See previous graphic

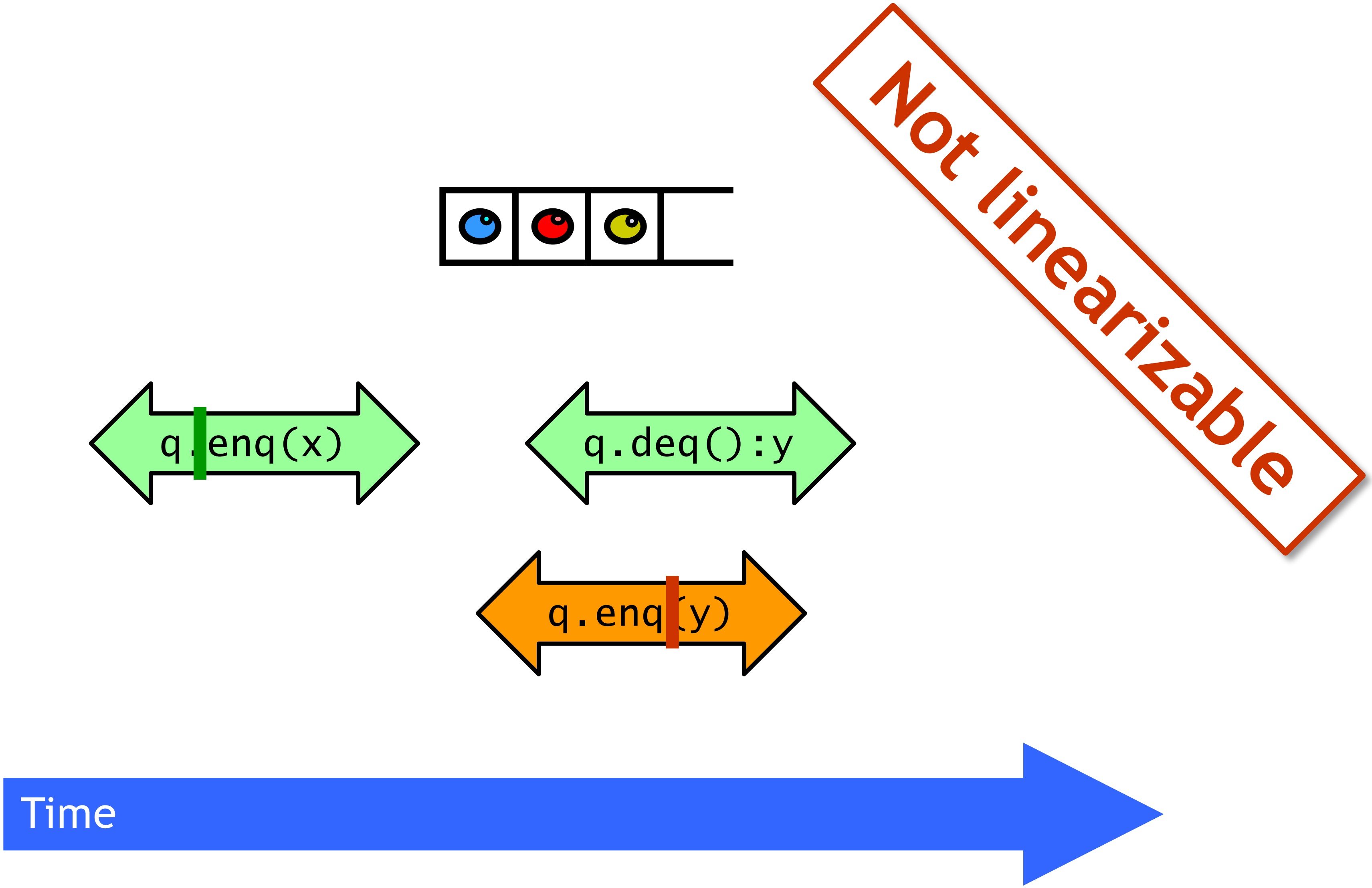
- Each method should
  - “Take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is **linearizable**
  - Formally, a **linearizable object** is an object all of whose possible executions are linearizable

Example

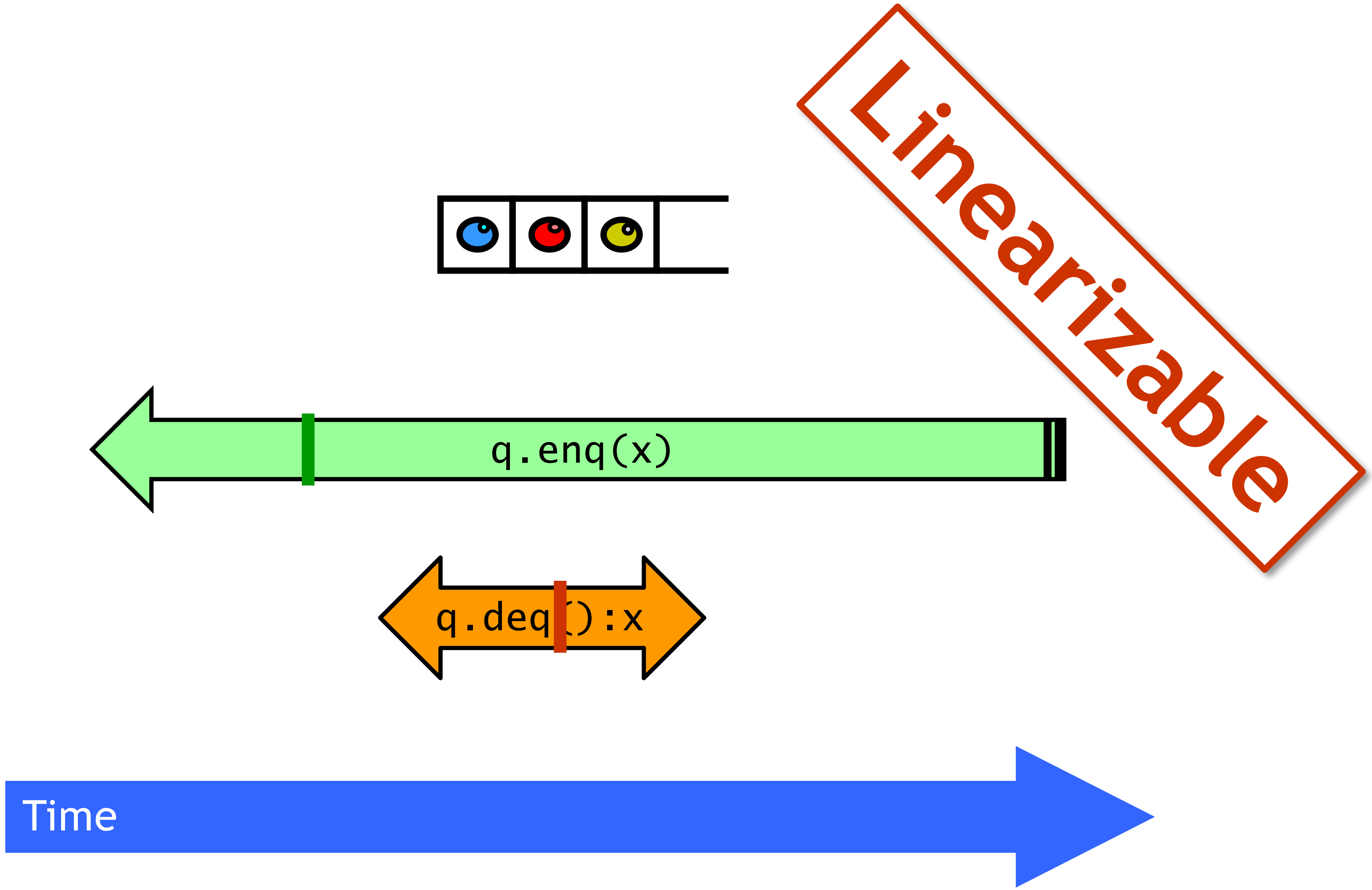
If we can find some points in the execution such as the time line is correct, it's linearizable.



Example

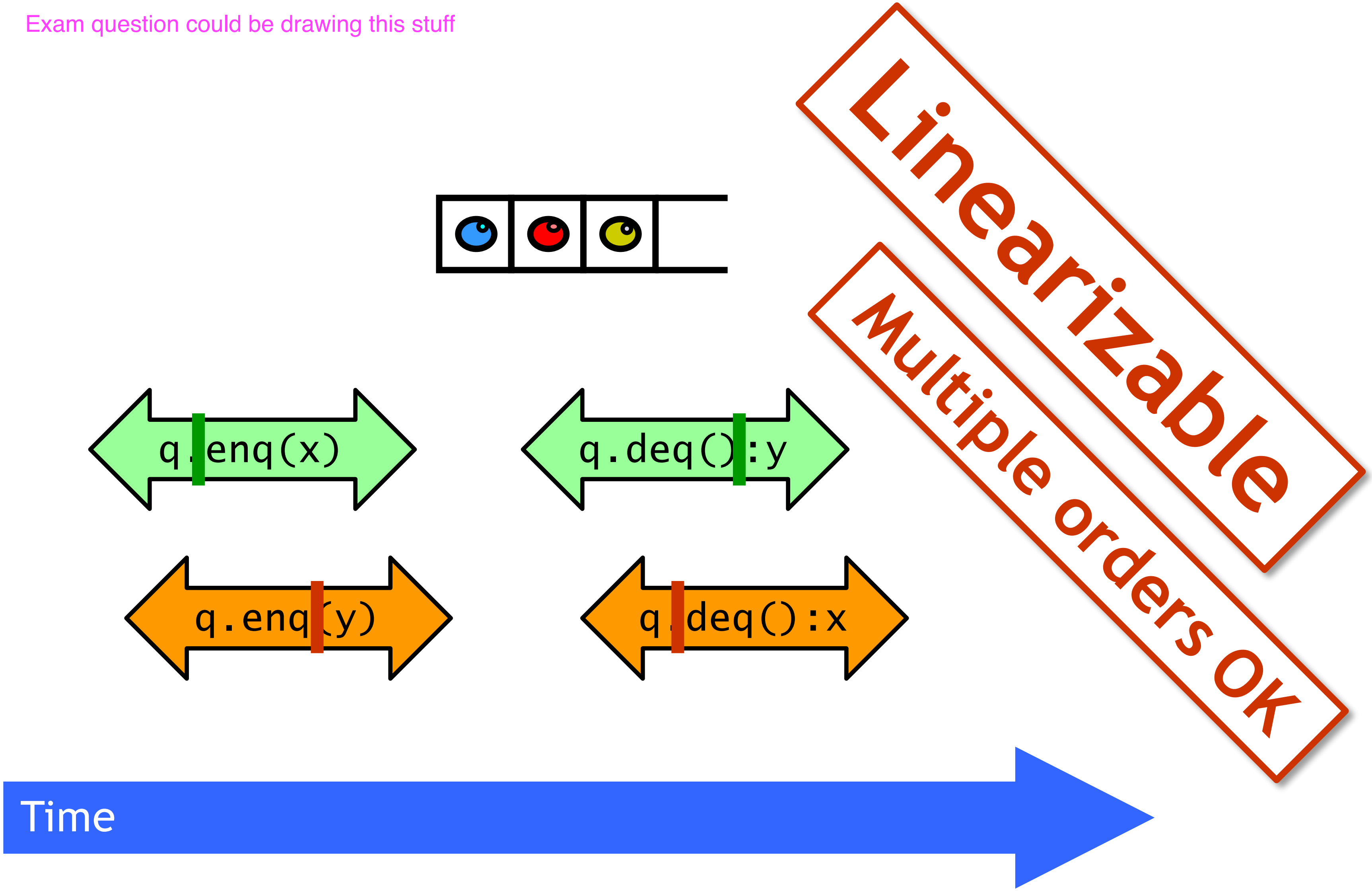


Example

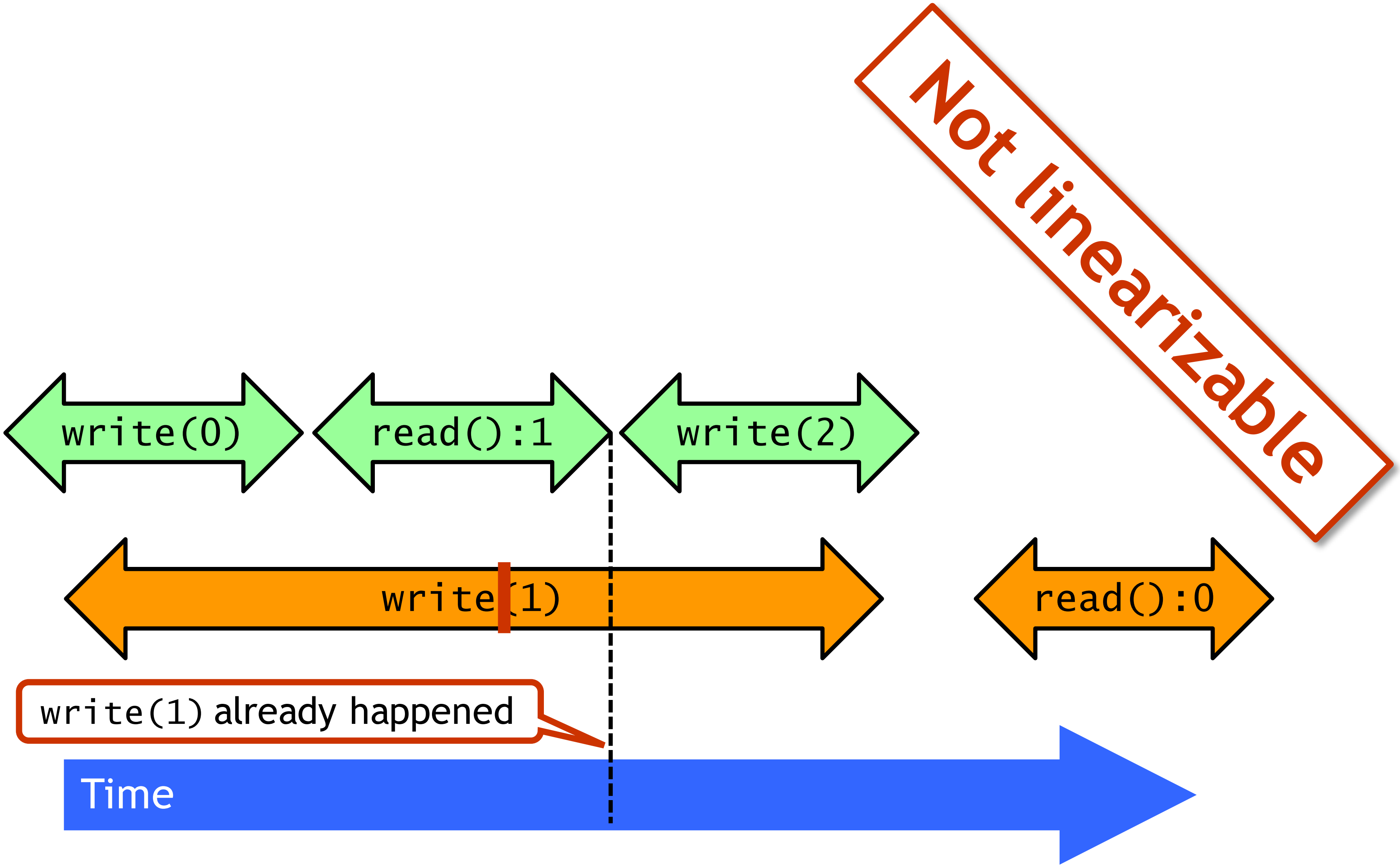


Example

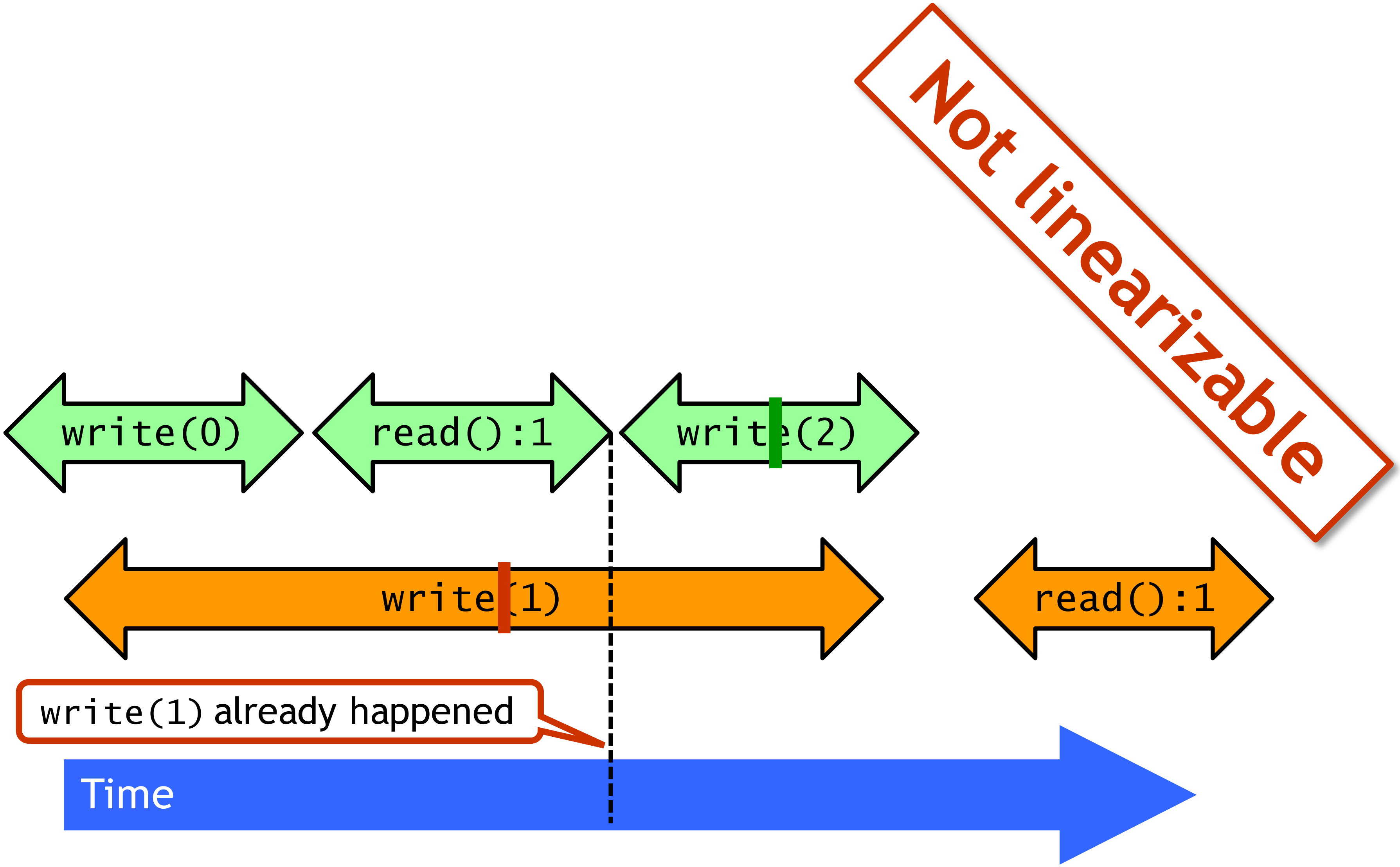
Exam question could be drawing this stuff



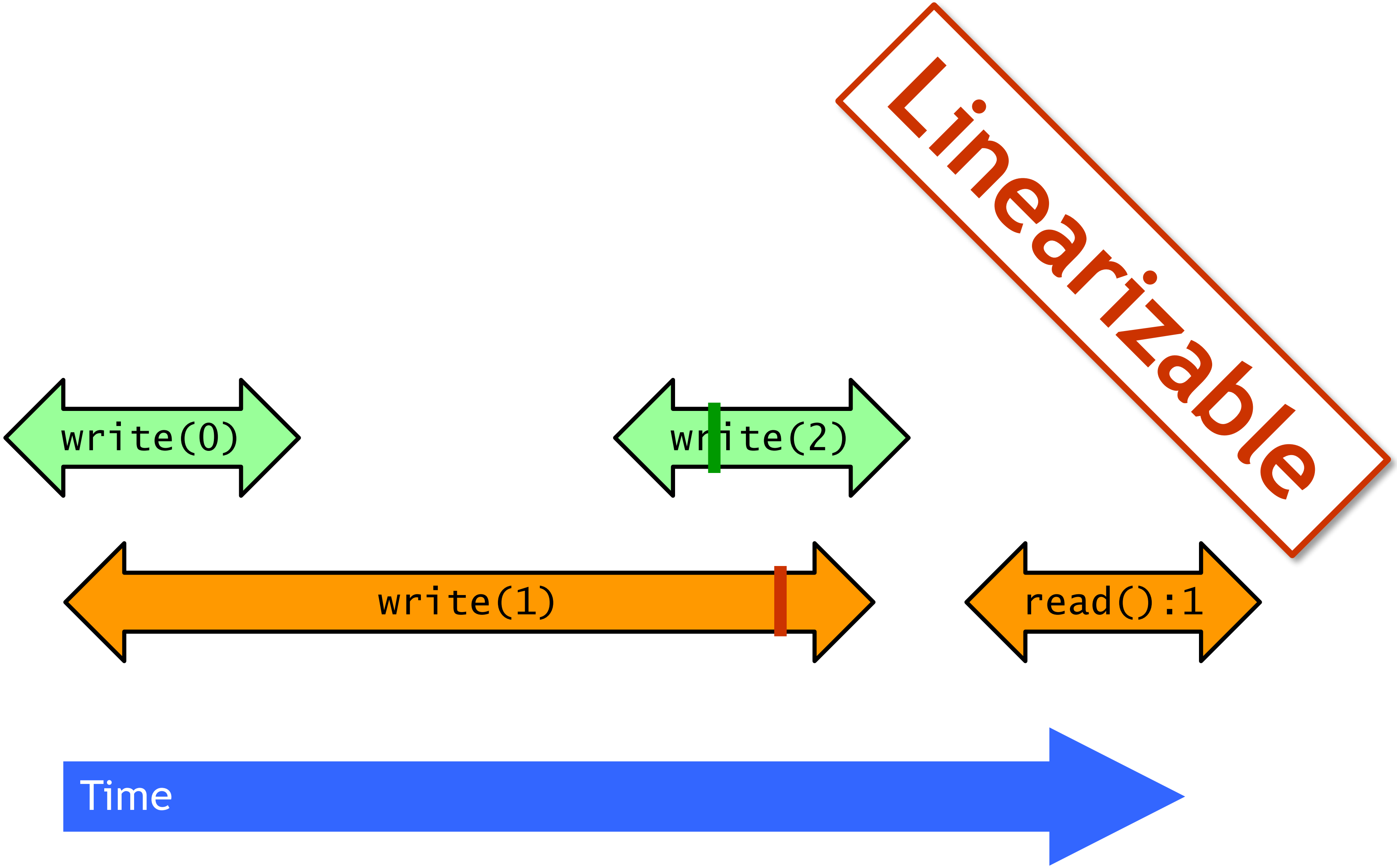
Read/Write Register Example



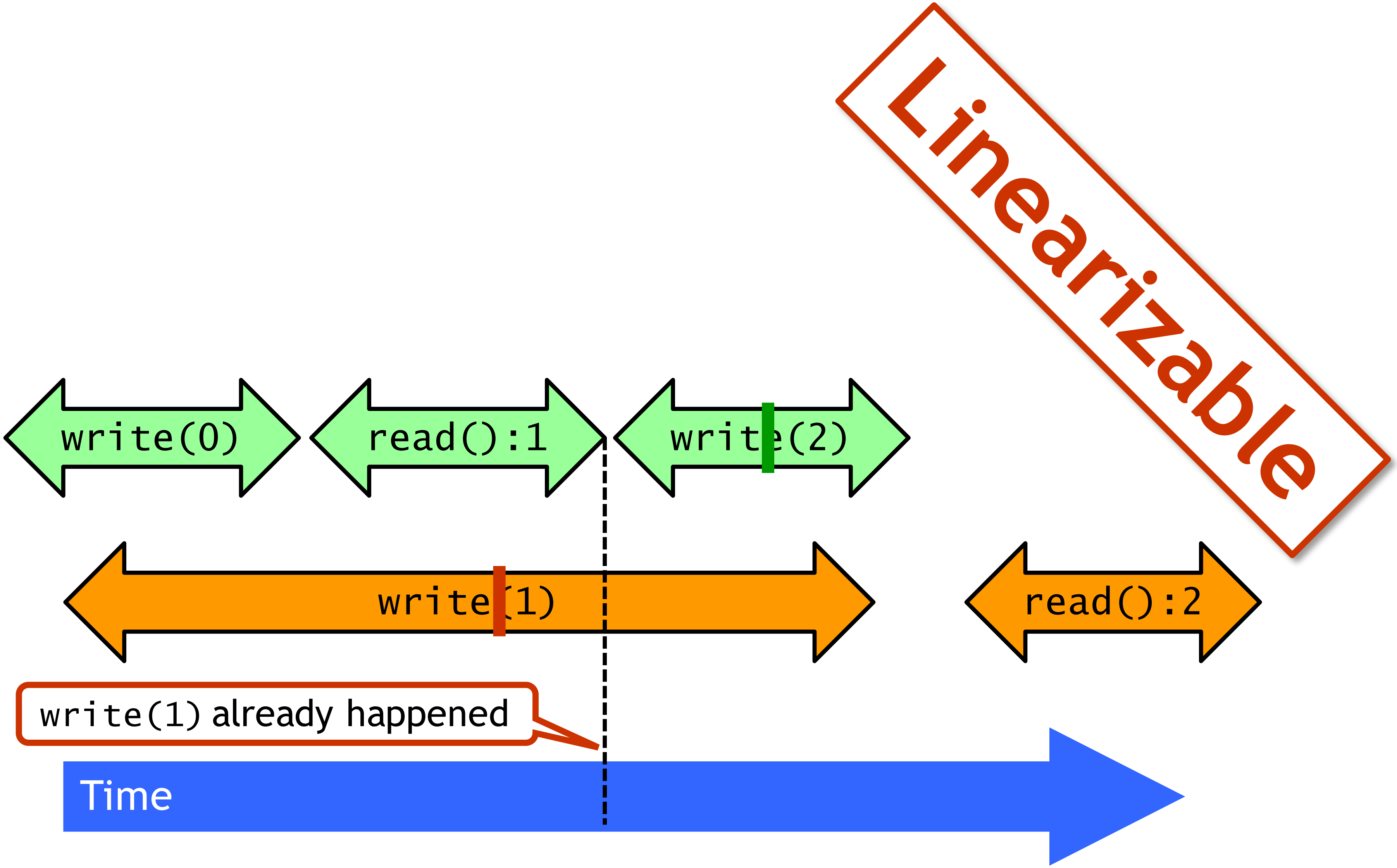
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Talking About Executions

- Why do we need to consider executions?
  - Can't we specify the linearization point of each operation without describing an execution?
- Not Always
  - In some cases, linearization point depends on the execution
- Let's define a formal model of executions
  - Define precisely what we mean (ambiguity is bad)
  - Allow reasoning, formal or informal

Split Method Calls into Two Events

- Invocation
  - Method name & arguments

    `q.enq(x)`
- Response
  - Result or exception

    `q.enq(x)` returns `void`

    `q.deq()` returns `x`

    `q.deq()` throws `Empty`

Invocation Notation

A q.enq(x)

Thread

Method

Object

Arguments

Response Notation

Method is implicit

`A q:void`

Thread

Object

Result

Response Notation

Method is implicit

`A q:Empty()`

Thread

Object

Exception

History: Describing an Execution

$$H = \begin{cases} \texttt{A q.enq(3)} \\ \texttt{A q:void} \\ \texttt{A q.enq(5)} \\ \texttt{B p.enq(4)} \\ \texttt{B p:void} \\ \texttt{B q.deq()} \\ \texttt{B q:3} \end{cases}$$

Sequence of invocations and responses

Definition

- Invocation & response **match** if

Object names agree

Thread names agree

```
A  q.enq(3)
A  q:void
```

Method call

Object Projections

```
      A q.enq(3)        A q.enq(3)
      A q:void          A q:void
H =   B p.enq(4)  H|q =
      B p:void
      B q.deq()         B q.deq()
      B q:3             B q:3
```

Thread Projections

H =

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```

H|B =

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Complete Subhistory

```
       A q.enq(3)
       A q:void
       A q.enq(5)
H =    B p.enq(4)
       B p:void
       B q.deq()
       B q:3
```

An invocation is pending if it has no matching response

May or may not have taken effect ⇒ discard pending invocations

Complete Subhistory

```
A q.enq(3)
A q:void
```

Complete(H) =

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Sequential Histories

H =

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

Match

Match

Match

Final pending invocation OK

Method calls of different threads do not interleave

Well-Formed Histories

Per-thread projections are sequential

```
H =  A q.enq(3)
     B p.enq(4)
     B p:void
     B q.deq()
     A q:void
     B q:3
```

```
H|B = B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

```
H|A = A q.enq(3)
      A q:void
```

Equivalent Histories

Threads see the same thing in both histories

$$\begin{cases} H|A = G|A \\ H|B = G|B \end{cases}$$

H =

```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

G =

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```

Sequential Specifications

- A sequential specification is some way of telling whether a
  - …single-thread, single-object history
  - …is legal
- Simple way is using
  - …pre and post-conditions
  - But plenty of other techniques exist

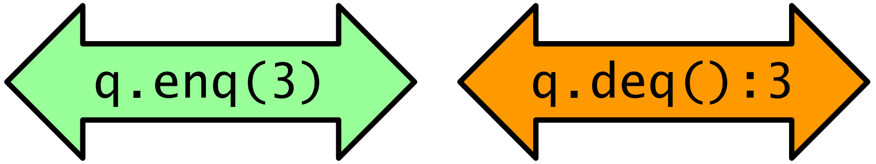
Legal Histories

- A sequential (multi-object) history **H** is legal if
  - For every object **x**
  - **H|x** is in the sequential specification for **x**

Precedence

A method call precedes another if response event precedes invocation event

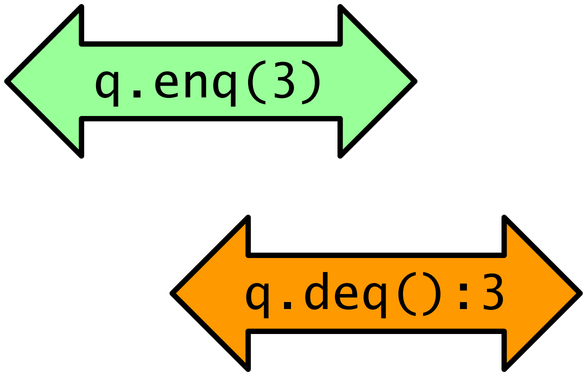
```
       A q.enq(3)
       B p.enq(4)
       B p:void
H =    A q:void
       B q.deq()
       B q:3
```



Non-Precedence

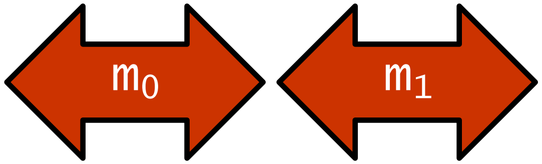
Some method calls *overlap* one another

```
H = A q.enq(3)
    B p.enq(4)
    B p:void
    B q.deq()
    A q:void
    B q:3
```



Notation

- Given
  - History **H**
  - Method executions m_0 and m_1 in **H**
- We say $m_0 \rightarrow_H m_1$, if
  - m_0 precedes m_1



- Relation $m_0 \rightarrow_H m_1$ is a
  - Partial order
  - Total order if **H** is sequential

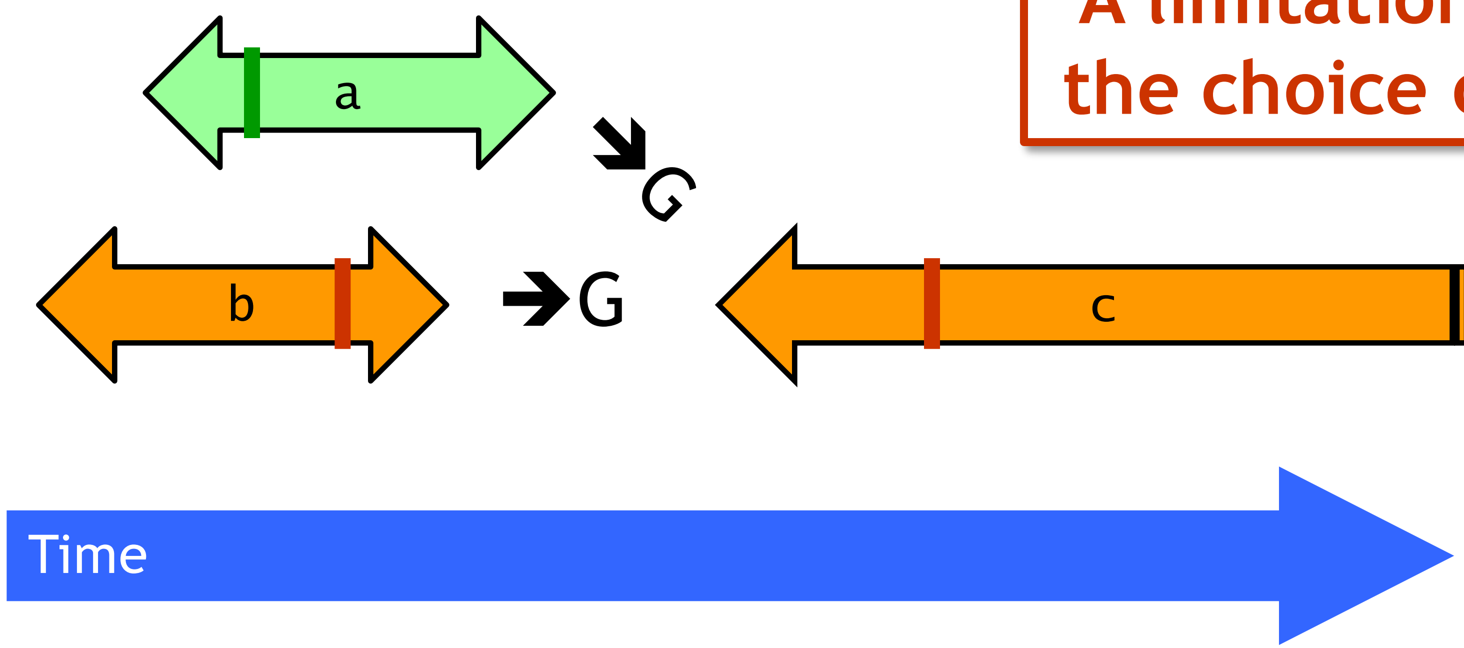
Linearizability

- History **H** is **linearizable** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that **G** is equivalent to
  - Legal sequential history **S**
  - Where $\rightarrow_G \subset \rightarrow_S$

What is $\rightarrow_G \subset \rightarrow_S$?

$\rightarrow_G = \{a \rightarrow c,\ b \rightarrow c\}$

$\rightarrow_S = \{a \rightarrow b,\ a \rightarrow c,\ b \rightarrow c\}$



A limitation on the choice of S!

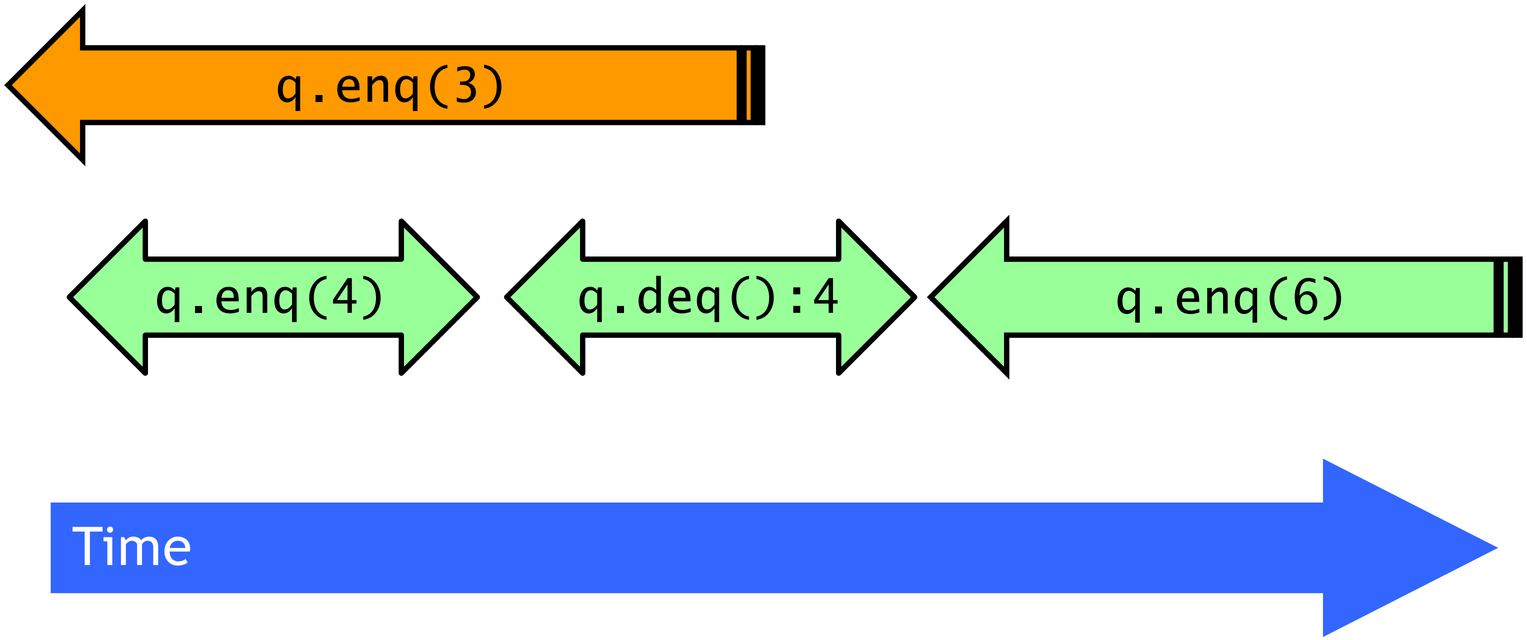
Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\rightarrow_G \subset \rightarrow_S$
  - Means that **S** respects “real-time order” of **G**

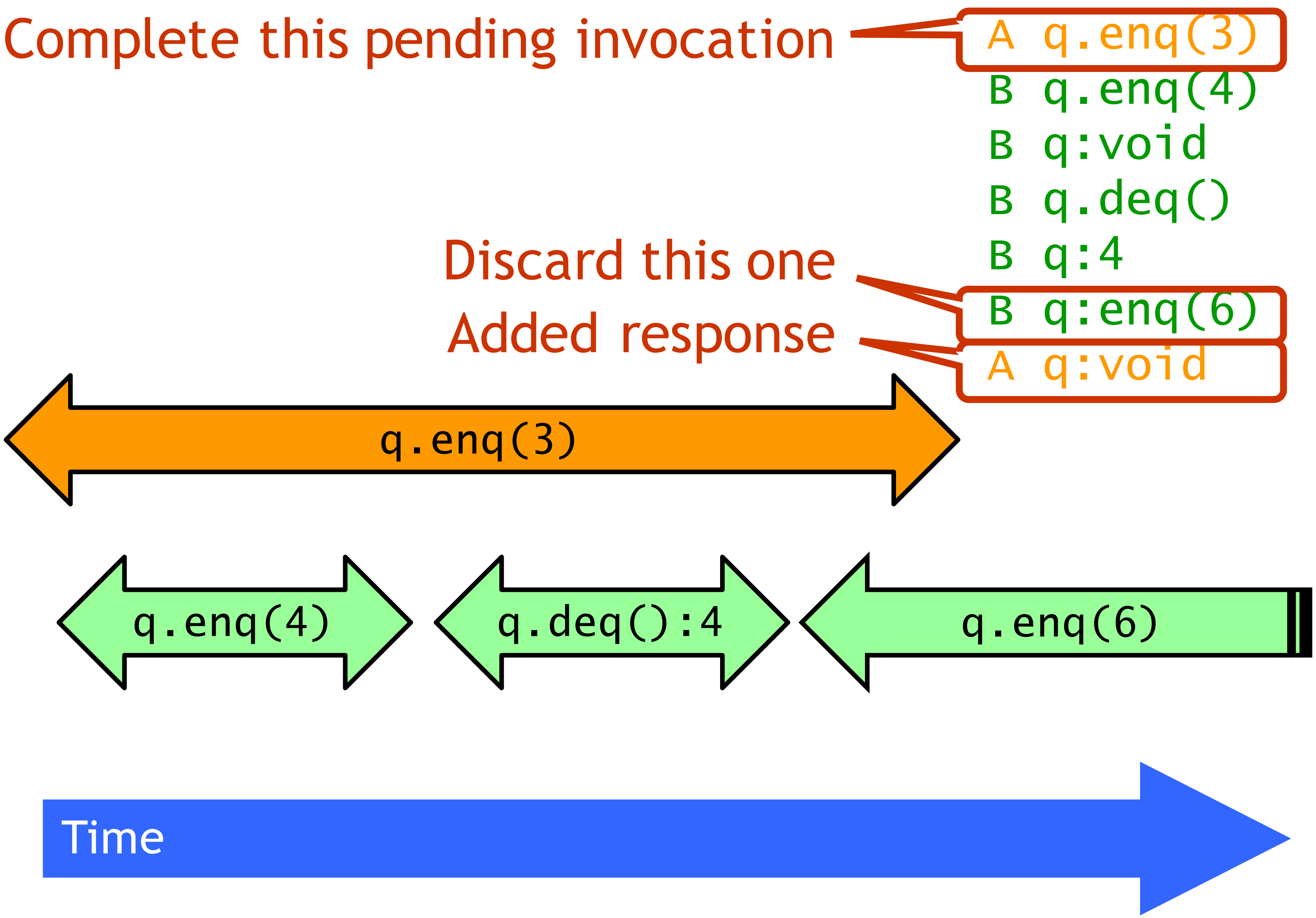
Example

Complete this pending invocation

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

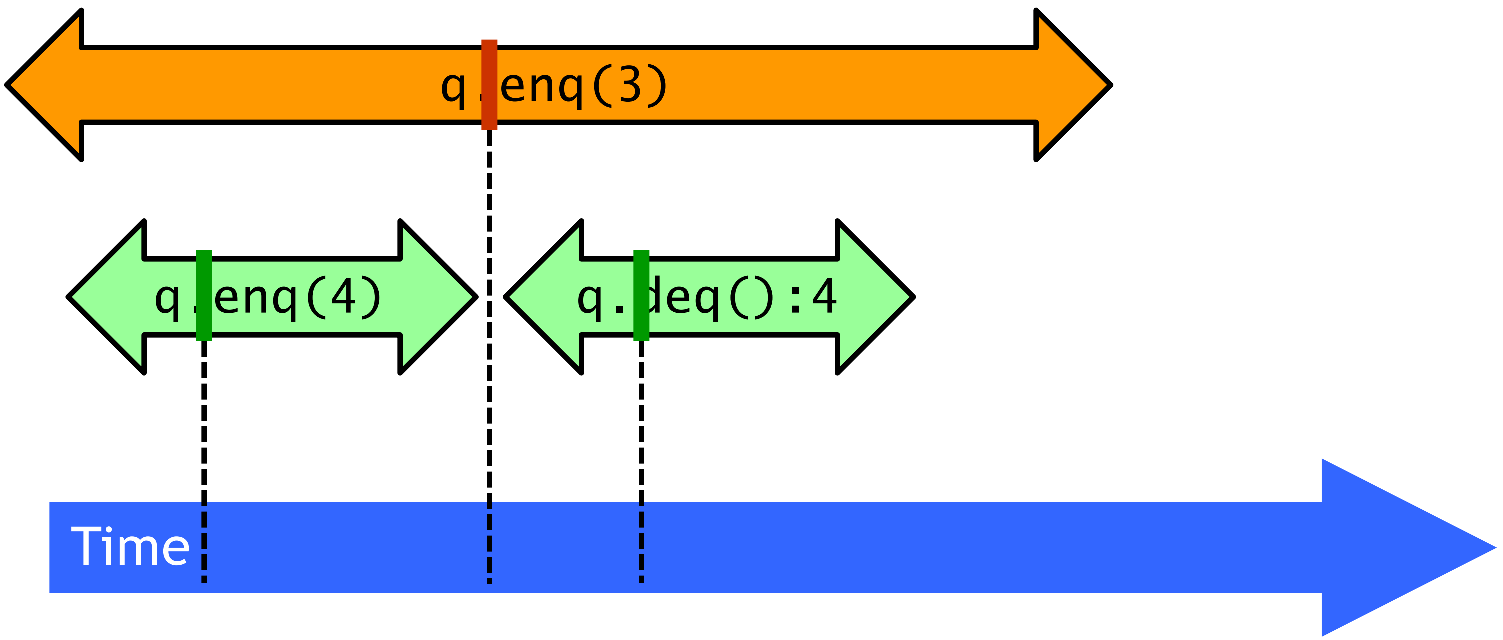

Complete this pending invocation
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
Discard this one
Added response
q.enq(3)
q.enq(4)
q.deq():4
q.enq(6)
Time

Example

```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```

Equivalent Sequential history

```
B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4
```



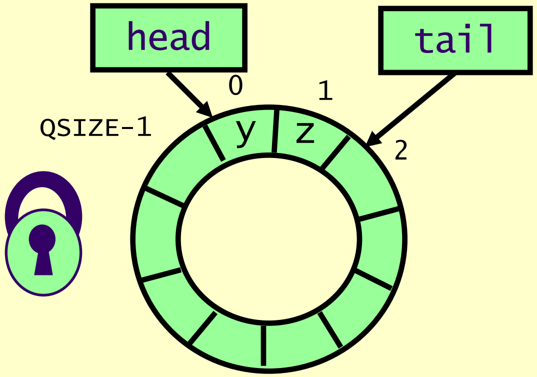
Locality Theorem

- History H is linearizable if and only if
  - For every object x
  - H|x is linearizable
- We care about objects only!

- Why Does Locality Matter?
  - Modularity
  - Can prove linearizability of objects in isolation
  - Can compose independently-implemented objects

Linearizability: Locking

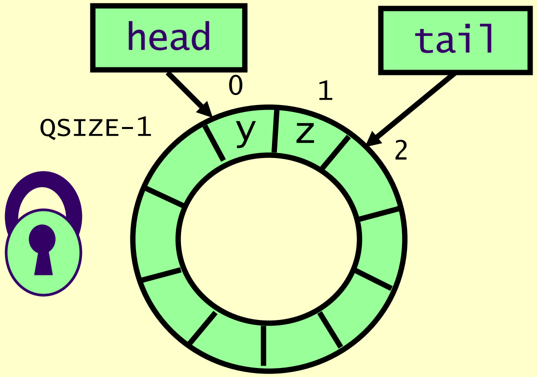
```
public class Queue {
  int head = 0, tail = 0;
  Object[QSIZE] items;

  public synchronized
  void enq(Object x) {
    while (tail – head == QSIZE)
      this.wait();
    items[tail % QSIZE] = x;
    tail++;
    this.notifyAll();
  } …
}
```



As we said, the linearization order is order lock acquired

Linearizability: Lock-free

```
public class LockFreeQueue {
  volatile int head = 0,
               tail = 0;
  Object[QSIZE] items;
  public void enq(Item x) {
    while (tail-head == QSIZE); // busy-wait
    items[tail % QSIZE] = x; tail++;
  }
  public Item deq() {
    while (tail == head); // busy-wait
    Item item = items[head % QSIZE]; head++;
    return item;
  }
}
```



Linearization order is order head and tail fields modified

Strategy

- Identify one atomic step where method "happens"
  - Critical section
  - Machine instruction
- Does not always work
  - Might need to define several different steps for a given method

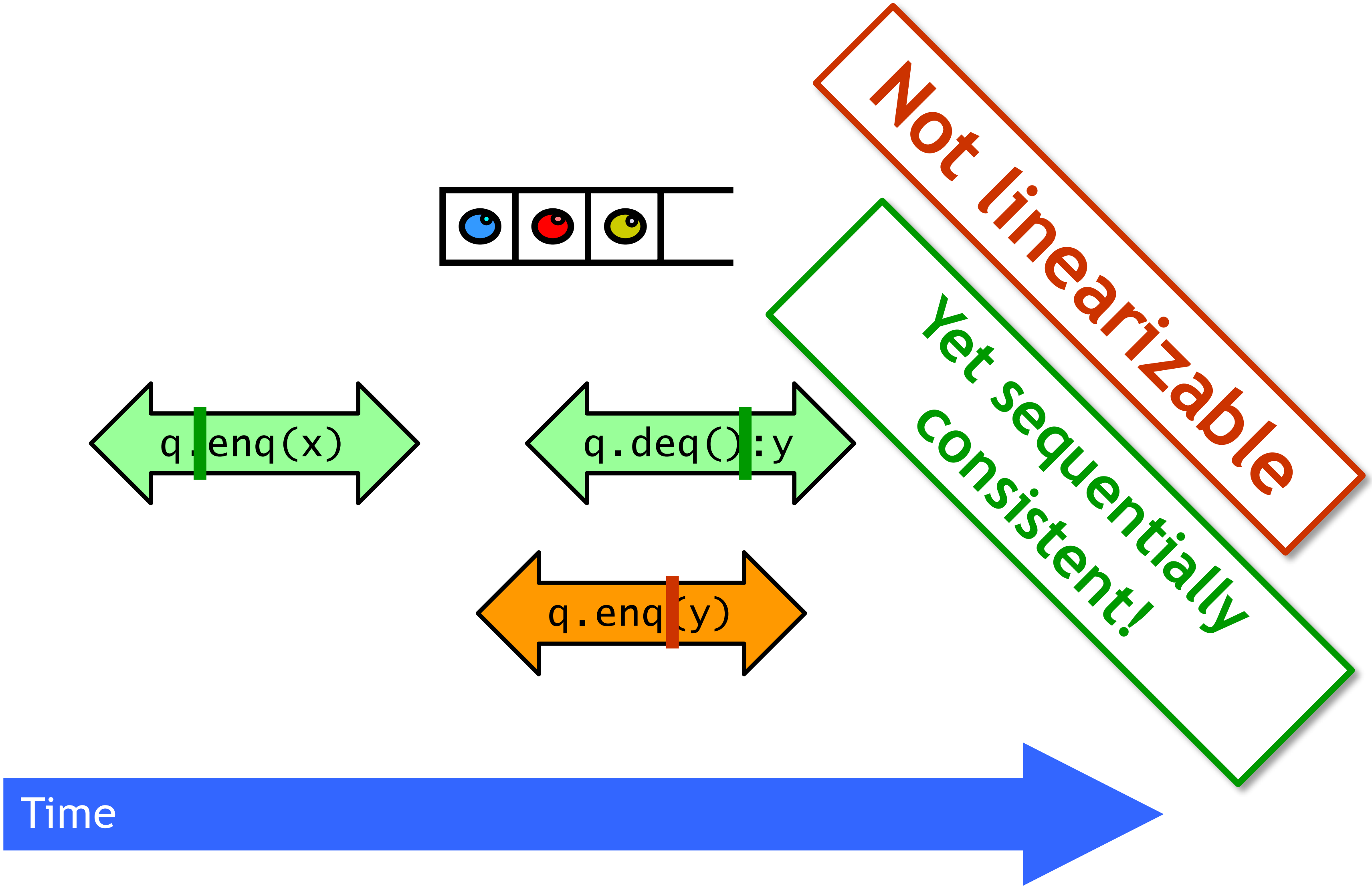
Alternative: Sequential Consistency

- History **H** is sequentially consistent if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that **G** is equivalent to a
  - Legal sequential history **S**
  - ~~Where $\rightarrow_G \subset \rightarrow_S$~~ Differs from linearizability

Alternative: Sequential Consistency

- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

Example

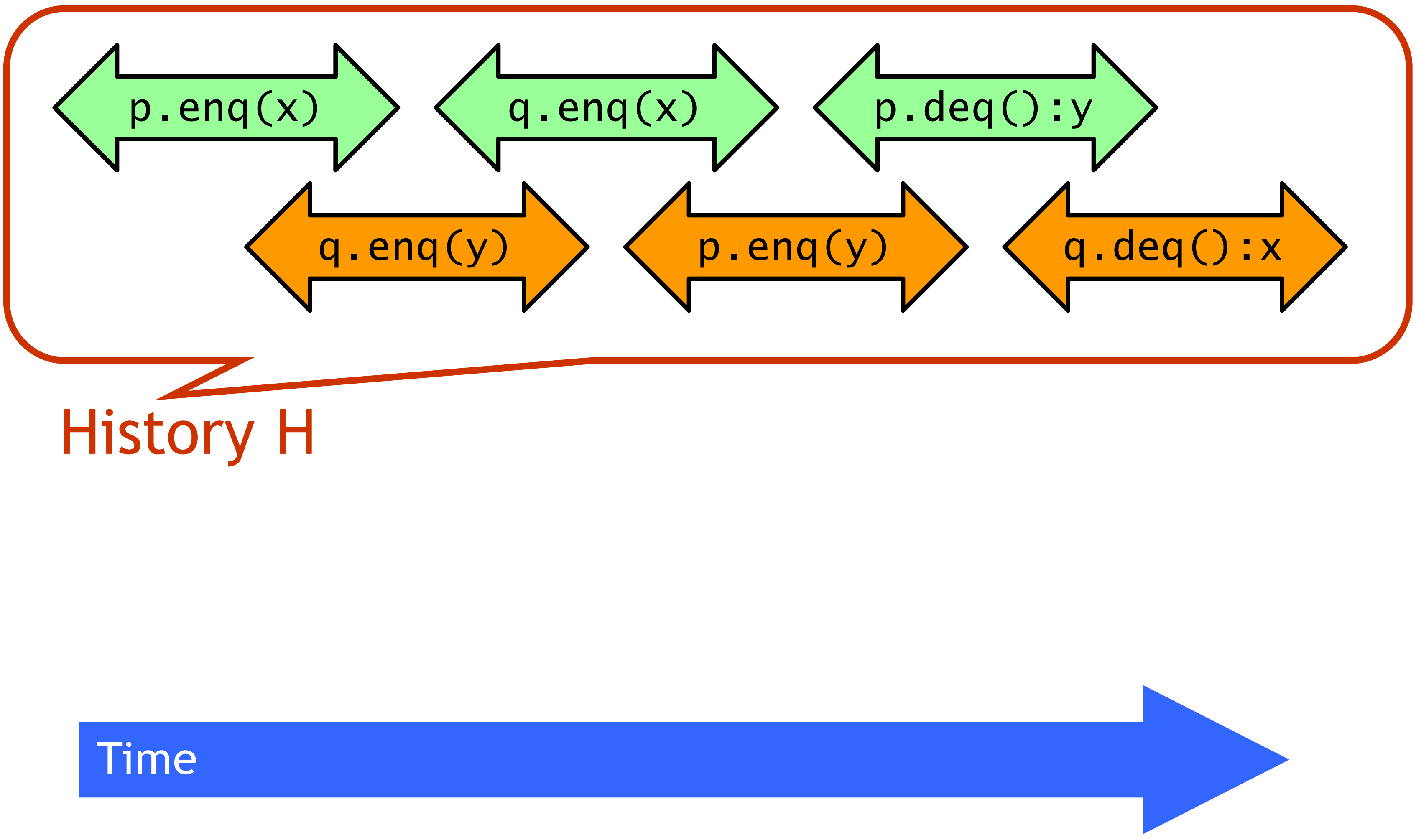

q.enq(x)
q.deq():y
q.enq(y)
Time
Not linearizable
Yet sequentially consistent!

Theorem

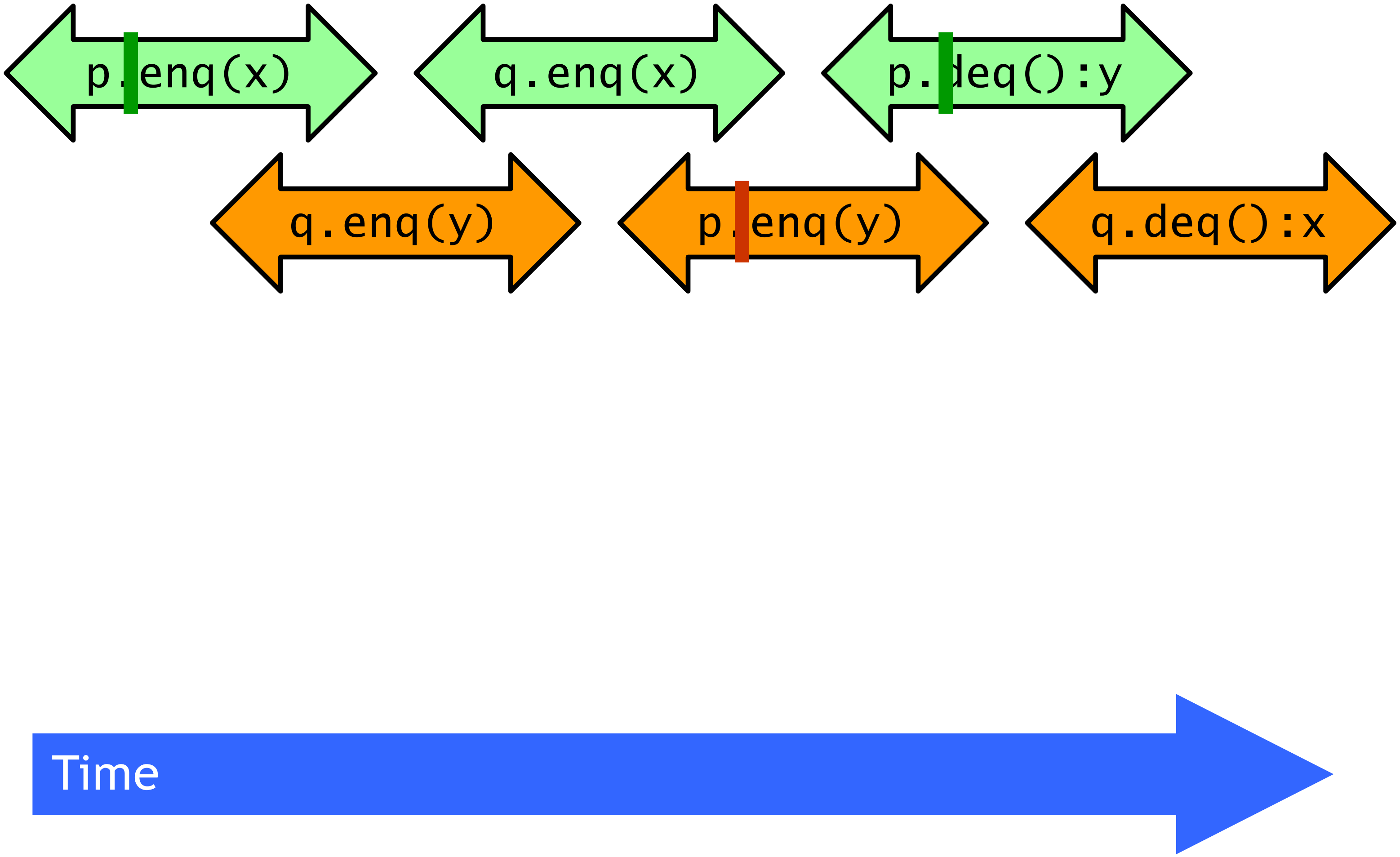
- Sequential consistency is not a local property

  (and thus we loose composability…)

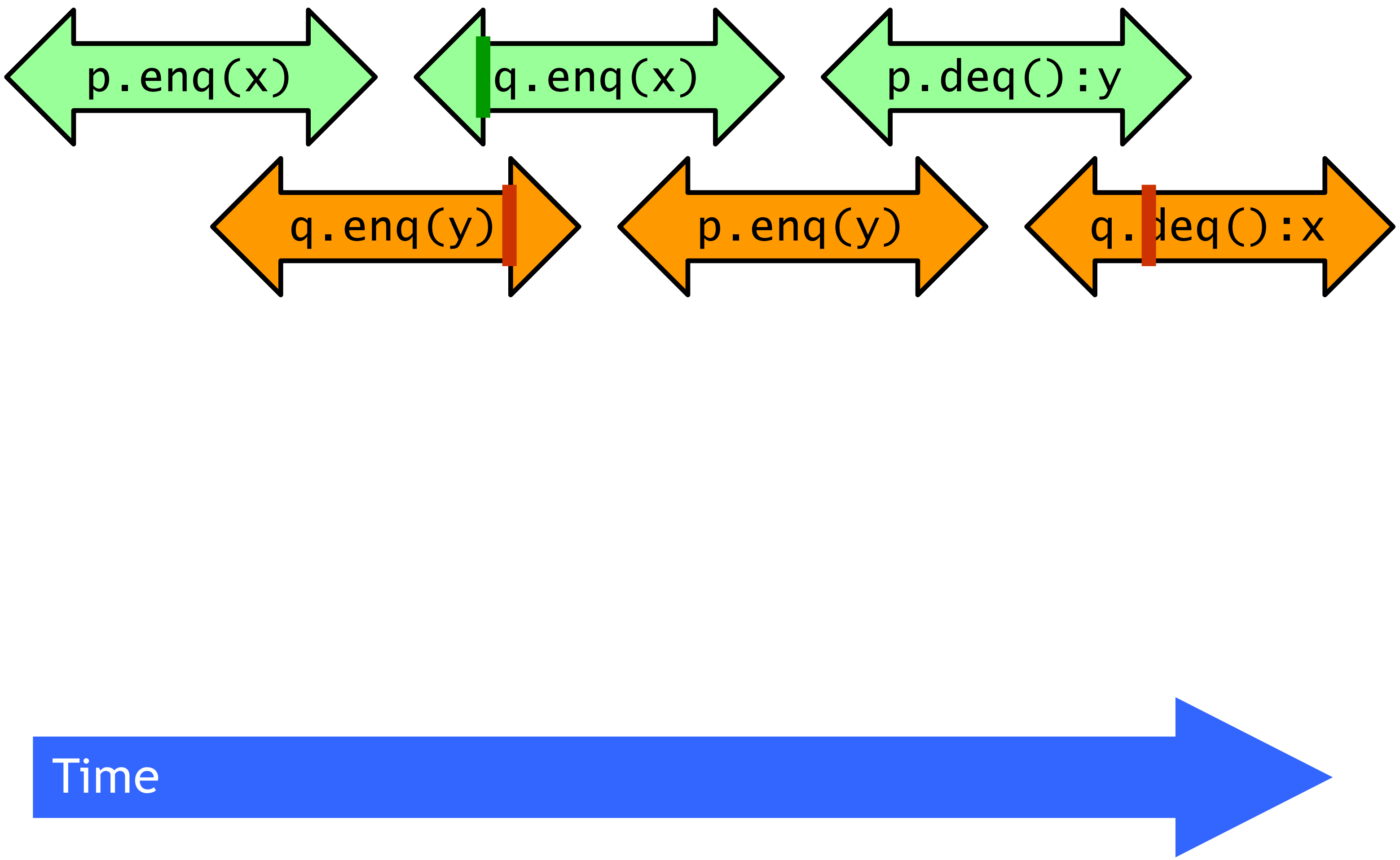
FIFO Queue Example



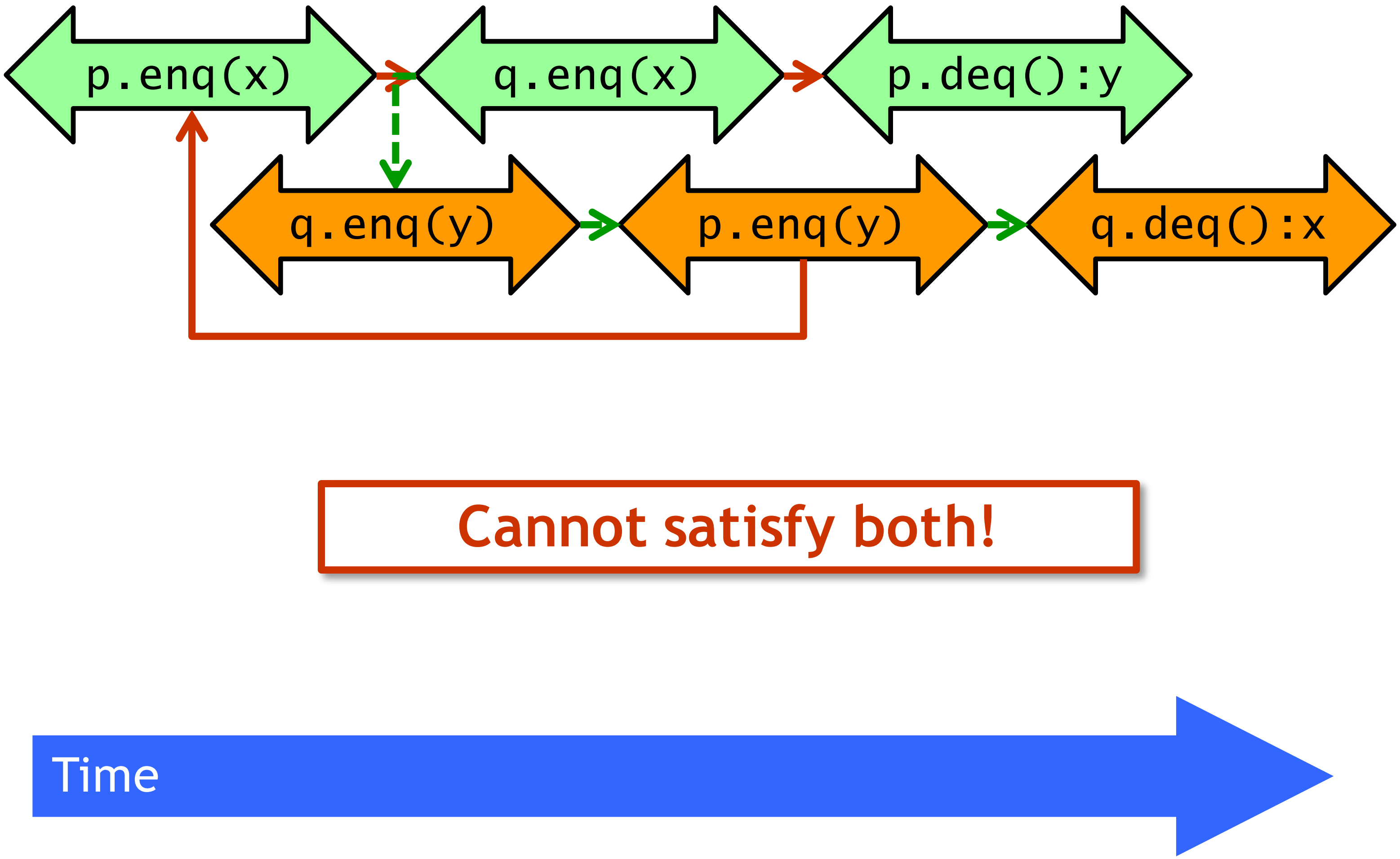
H|p Sequentially Consistent



H|q Sequentially Consistent



Ordering Imposed by p and q



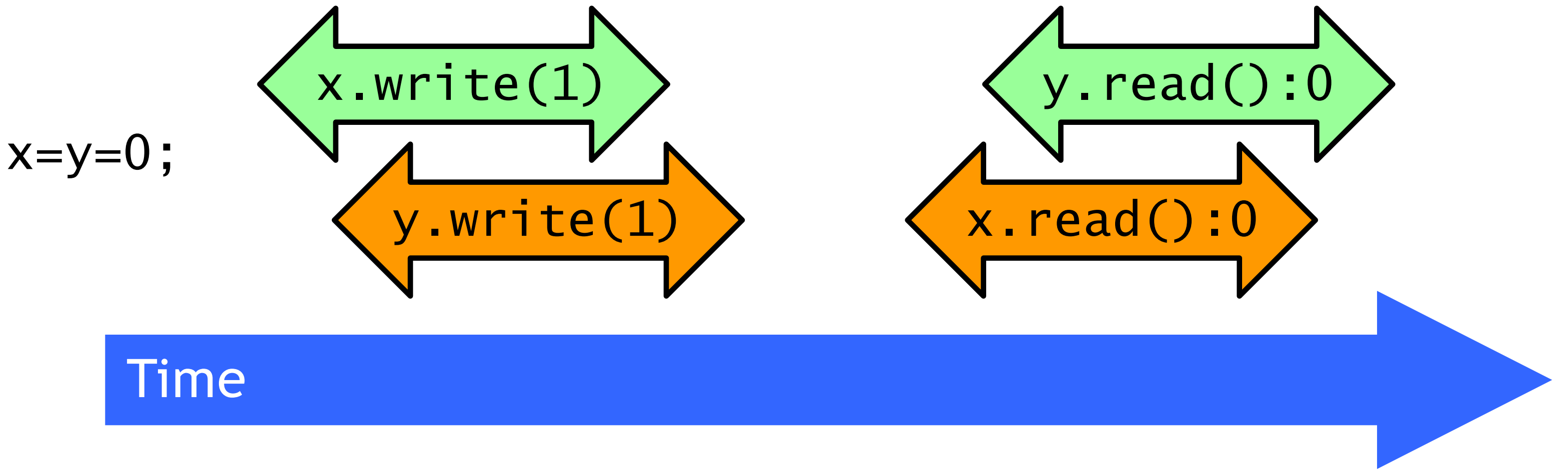
Cannot satisfy both!

Fact

- Most hardware architectures do not support sequential consistency
- Because they think it is too strong
- Here is another story…

The Flag Example

- Each thread's view is sequentially consistent
  - It went first
- Entire history is not sequentially consistent
  - Cannot both go first
- Is this behavior really so wrong?



Opinion 1: It is Wrong!

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It is non-negotiable!

Opinion 2: But It Feels So Right…

- Many hardware architects think that sequential consistency is too strong
- Too expensive to implement in modern hardware
- OK if flag principle
  - Violated by default
  - Honored by explicit request

Memory Hierarchy

- On modern multiprocessors, processors do not read and write directly to memory
- Memory accesses are very slow compared to processor speeds
- Instead, each processor reads and writes directly to a cache
  - To read a memory location: load data into cache and read from cache
  - To write a memory location: update cached copy and lazily write cached data back to memory

While Writing to Memory

- A processor can execute hundreds, or even thousands of instructions
- Why delay on every memory write?
- Instead, write back in parallel with rest of the program

Revisionist History

- Flag violation history is actually OK
  - Processors delay writing to memory
  - Until after reads have been issued
- Otherwise unacceptable delay between read and write instructions
- Who knew you wanted to synchronize?

Synchronizing

- Writing to memory = mailing a letter
- Vast majority of reads & writes
  - Not for synchronization
  - No need to idle waiting for post office
- If you want to synchronize
  - Announce it explicitly
  - Pay for it only when you need it

Explicit Synchronization

- Memory barrier instruction
  - Flush unwritten caches
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections
- Expensive

Volatile

- In Java, can ask compiler to keep a variable up-to-date with volatile keyword
- Also inhibits reordering, removing from loops, & other “optimizations”

Real-World Hardware Memory

- Weaker than sequential consistency
- But you can get sequential consistency at a price
- OK for expert, tricky stuff
  - Assembly language, device drivers, etc.
- Linearizability more appropriate for high-level software

Critical Sections

- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Like synchronized methods in Java
- Problems
  - Blocking
  - No concurrency

Summary

- Linearizability
  - Powerful specification tool for shared objects
  - Allows us to capture the notion of objects being “atomic”
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composablity
  - Good for high level objects

Summary

- Sequential Consistency
  - Not composable
  - Harder to work with
  - Good way to think about hardware models
- We will use **linearizability** in the remainder of this course unless stated otherwise