YSC3242: Parallel, Concurrent and Distributed Programming

Concurrent Objects
Part 2

Formal Model of Executions

- Define precisely what we mean
  - Ambiguity is bad when intuition is weak
- Allow reasoning
  - Formal
  - But mostly informal
    - In the long run, actually more important

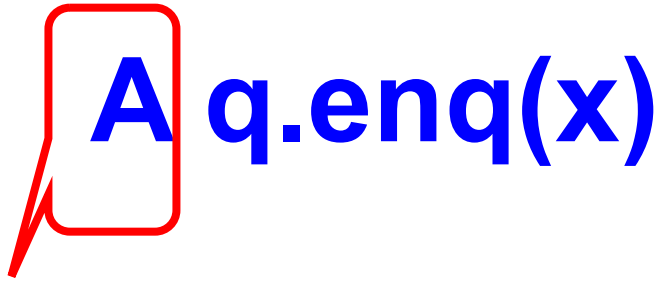
Split Method Calls into Two Events

- Invocation
  - method name & args
  - **`q.enq(x)`**
- Response
  - result or exception
  - **`q.enq(x)`** returns **`void`**
  - **`q.deq()`** returns **`x`**
  - **`q.deq()`** throws **`empty`**

Invocation Notation

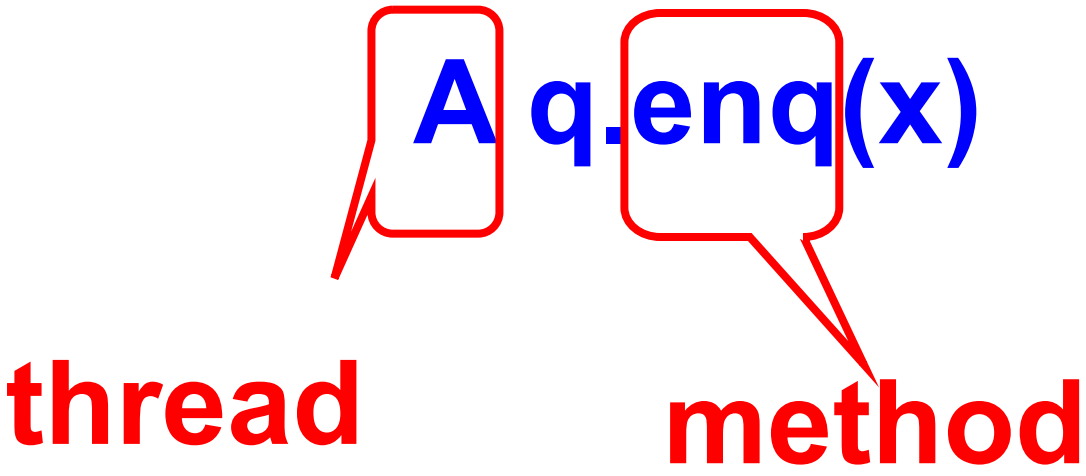
A q.enq(x)

Invocation Notation

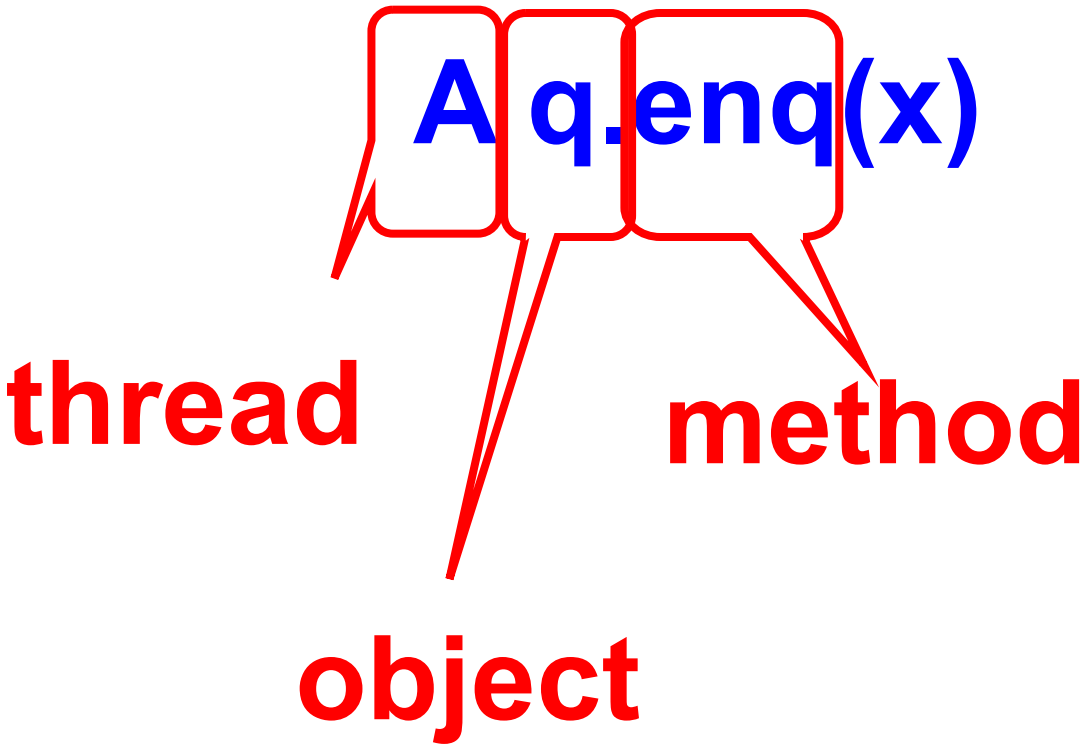


thread

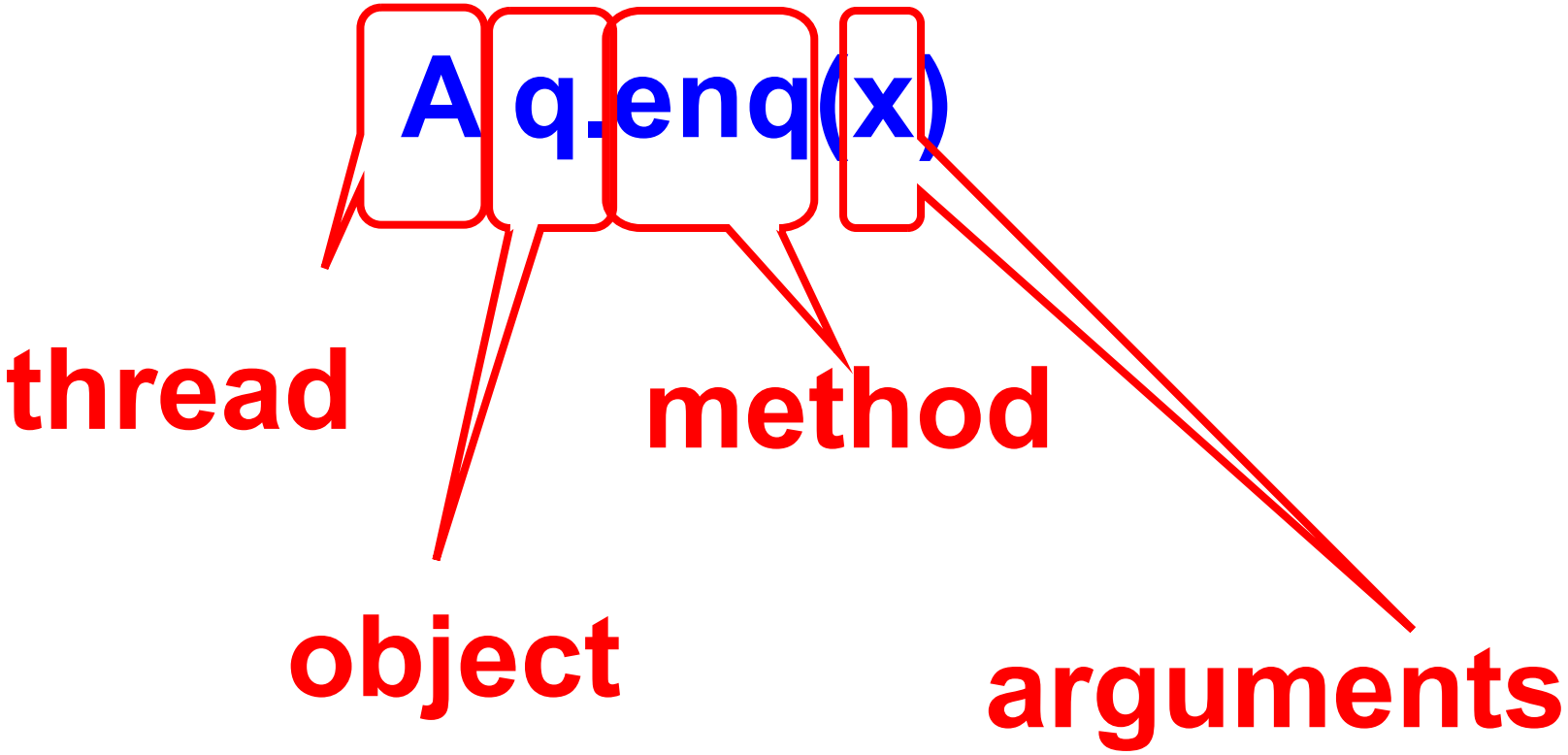
Invocation Notation



Invocation Notation



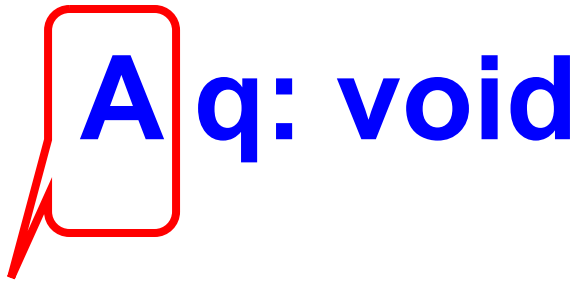
Invocation Notation



Response Notation

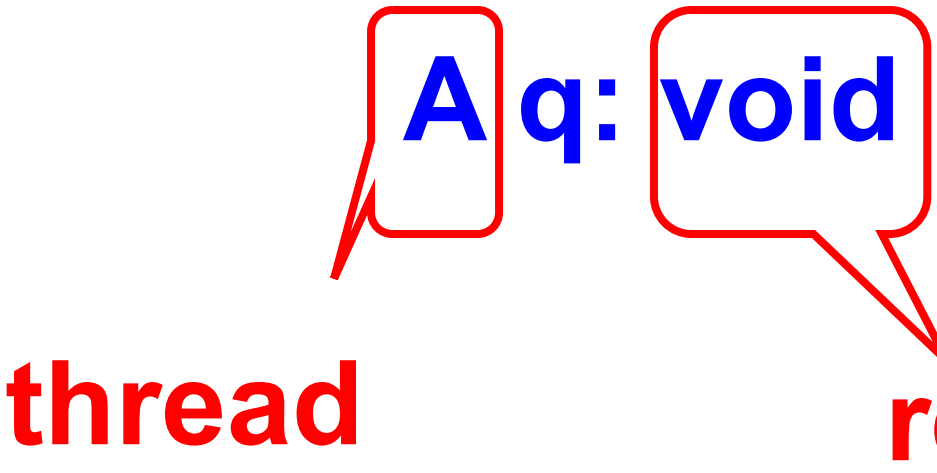
A q: void

Response Notation

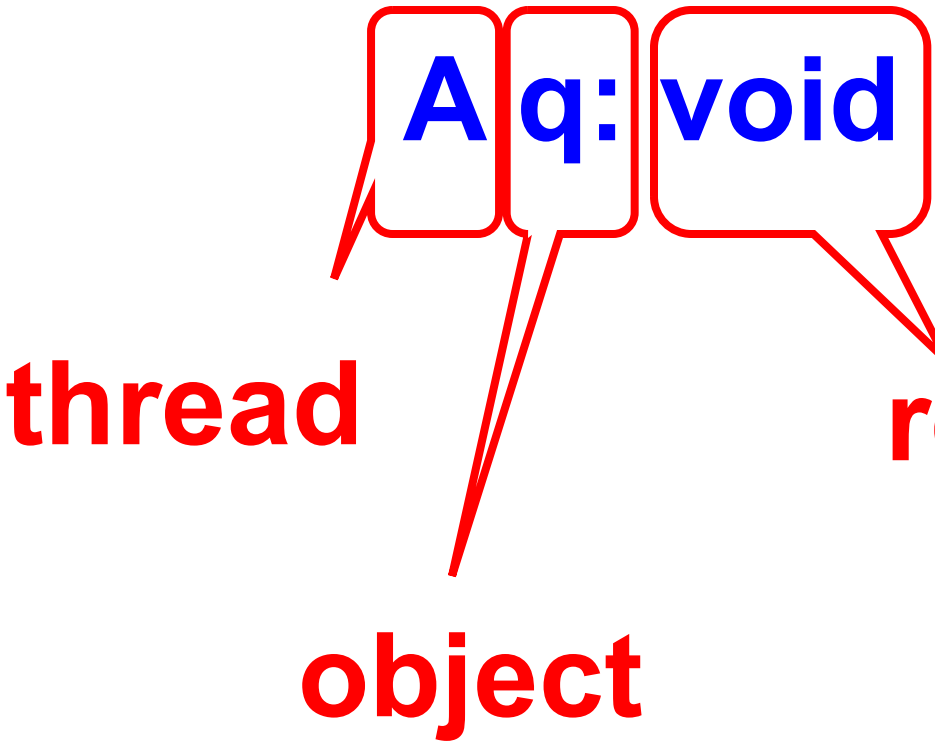


thread

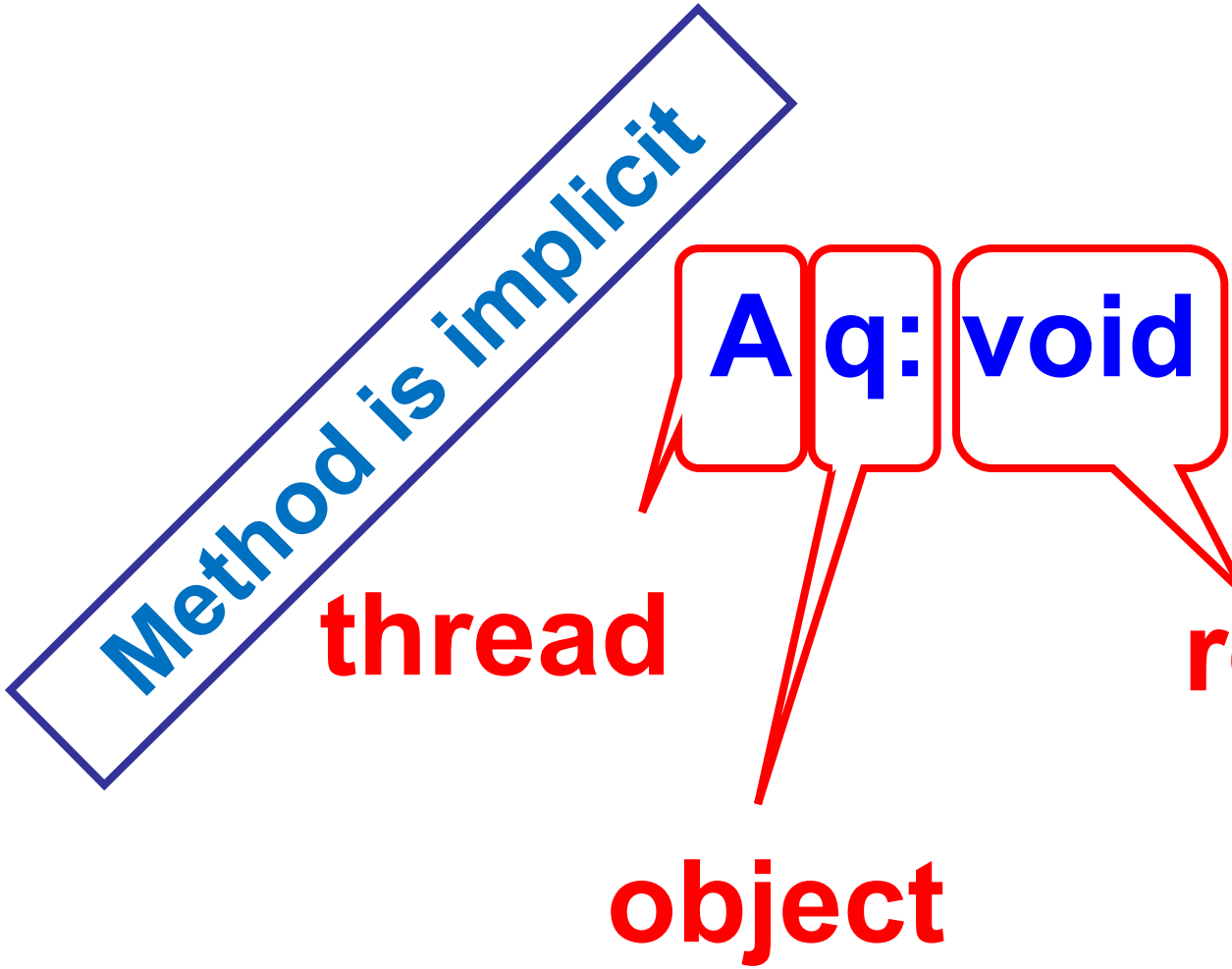
Response Notation



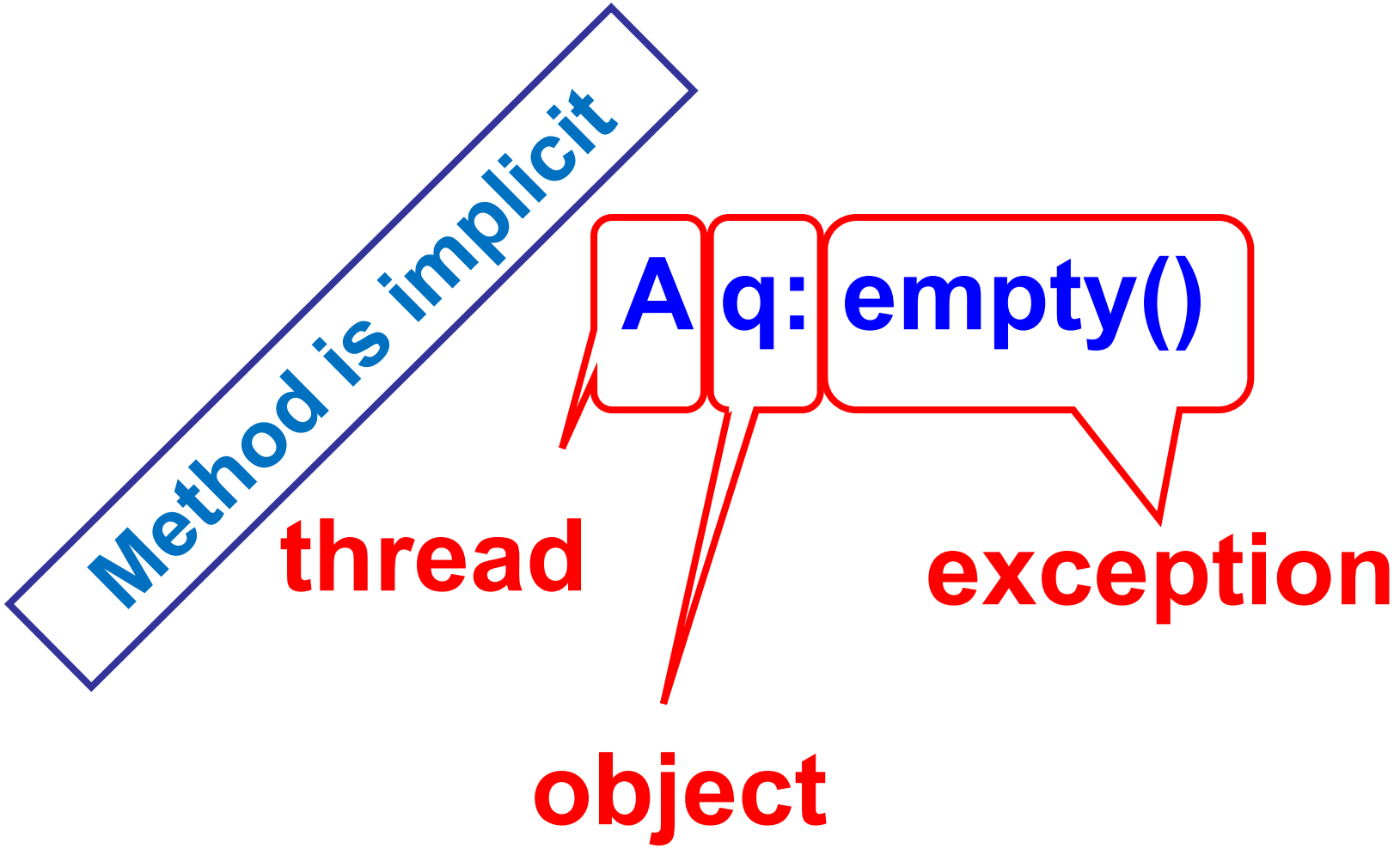
Response Notation



Response Notation



Response Notation



History - Describing an Execution

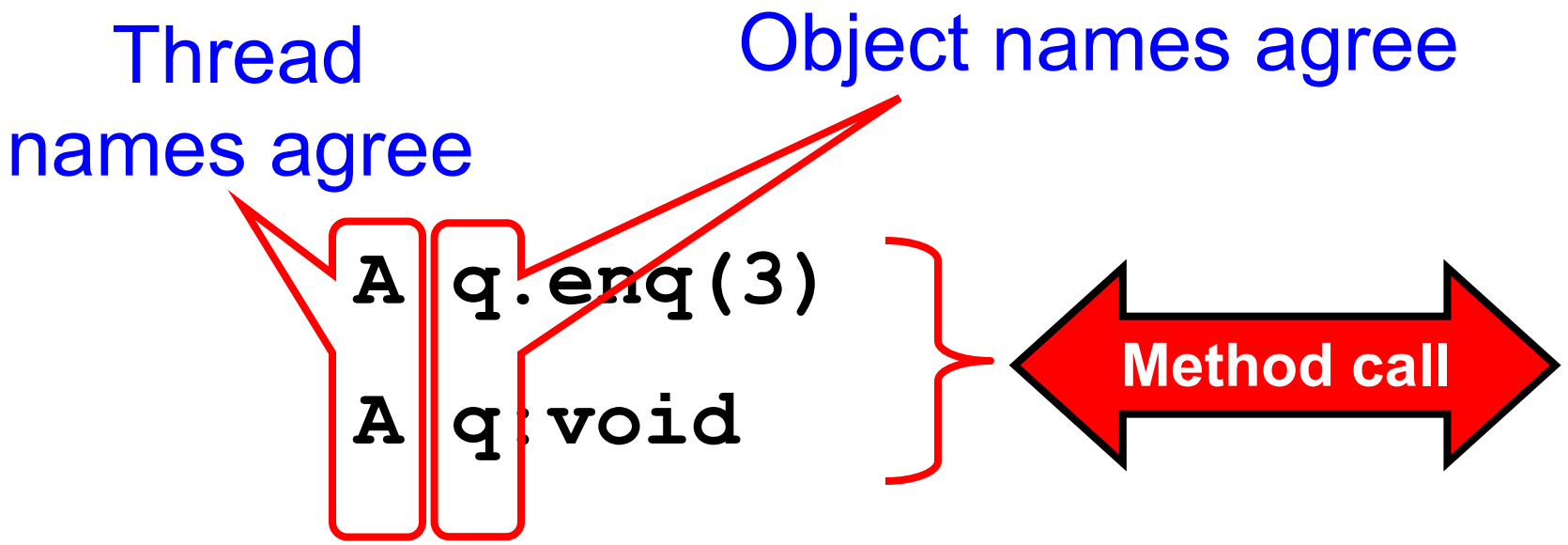
H =

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

Sequence of invocations and responses

Definition

- Invocation & response *match* if



Object Projections

H =

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

Object Projections

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

H|q =

```
B q.deq()
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
```

Thread Projections

H|B =

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

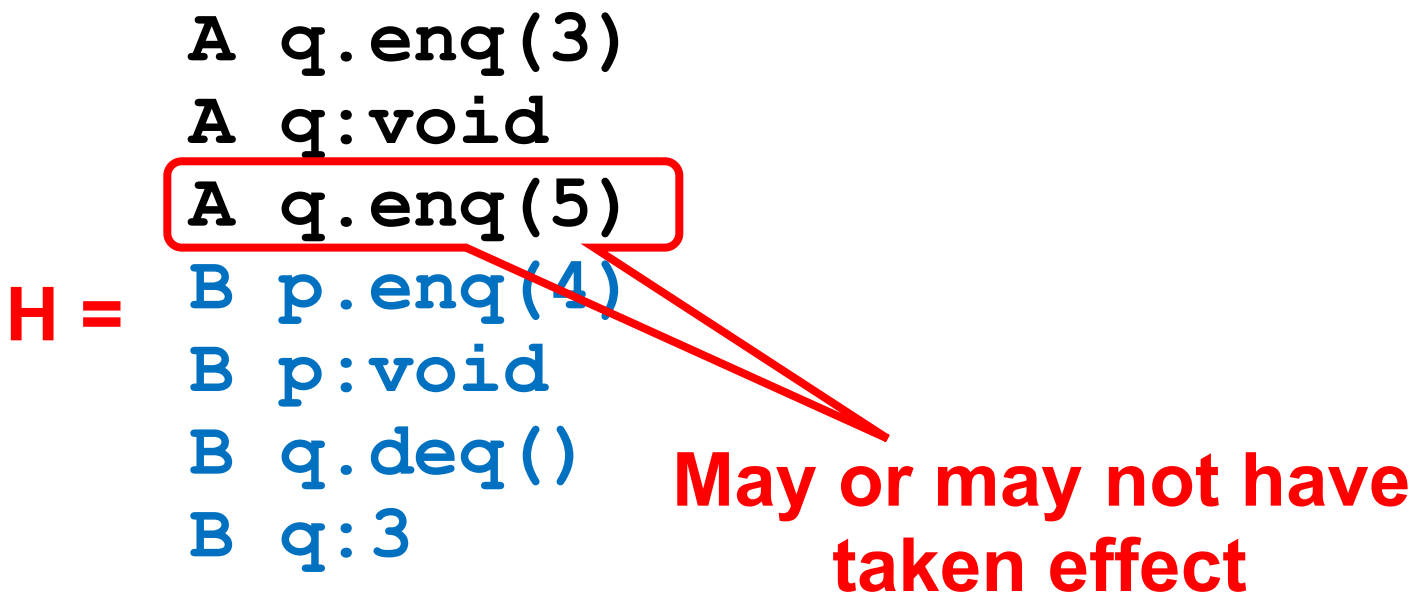
Complete Subhistory

H =

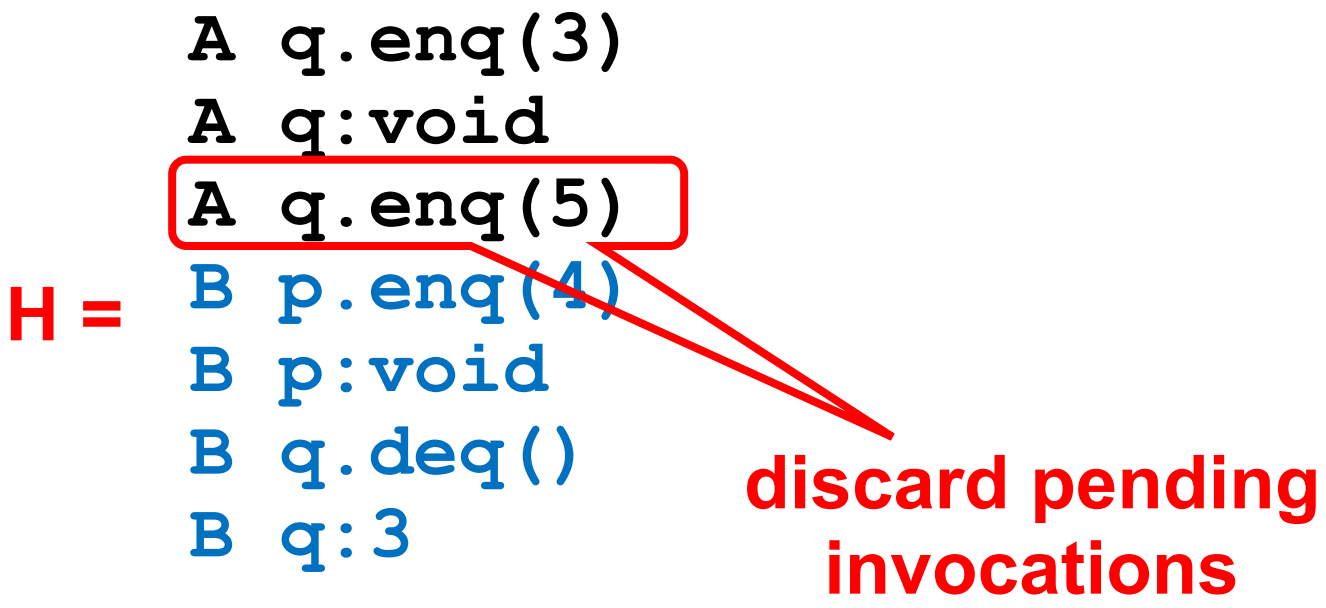
```
A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3
```

An invocation is *pending* if it has no matching response

Complete Subhistory



Complete Subhistory



Complete Subhistory

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

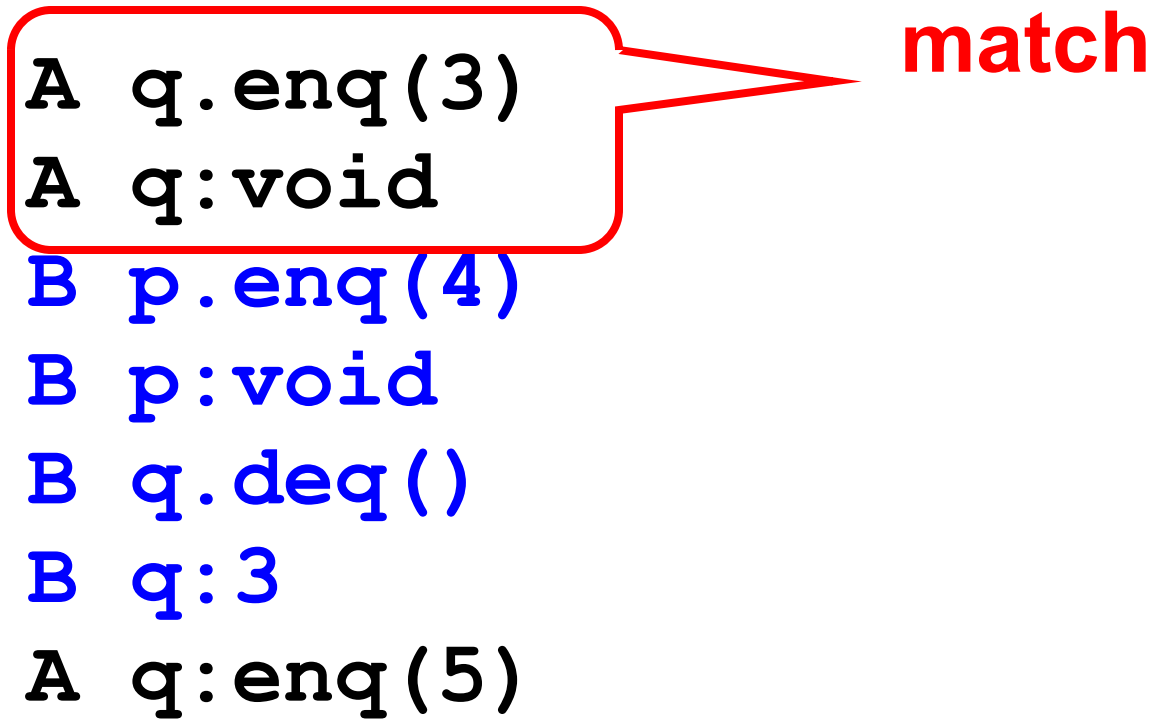
Complete(H) =

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

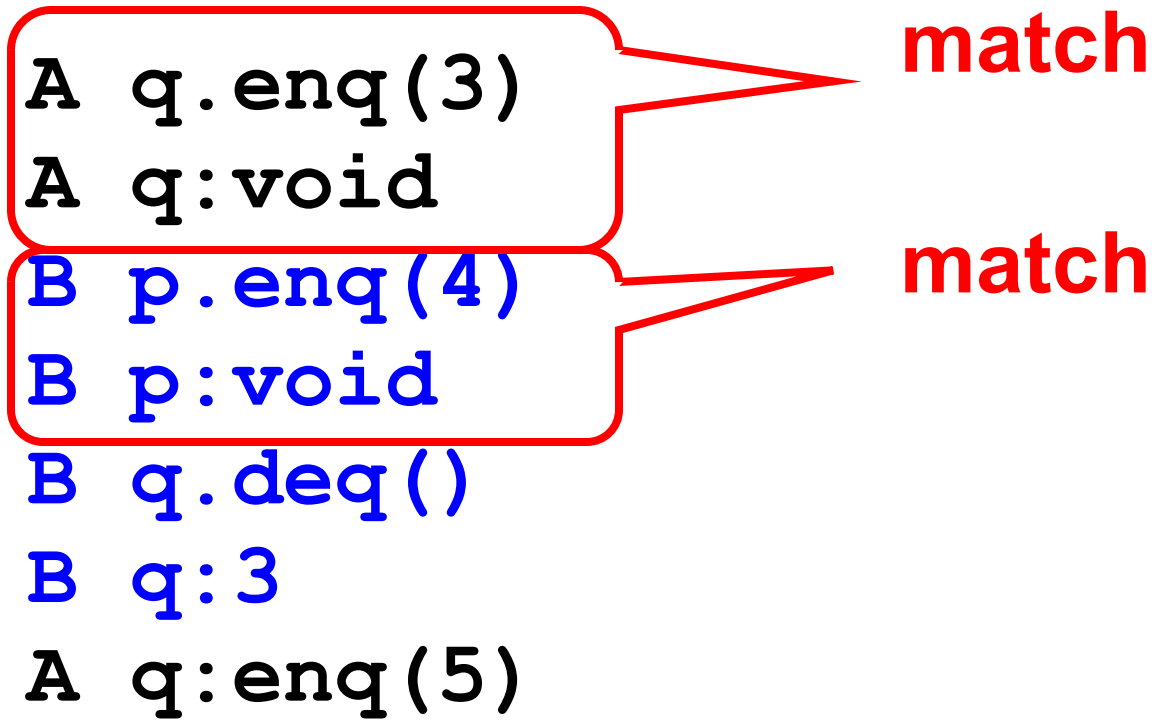
Sequential Histories

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

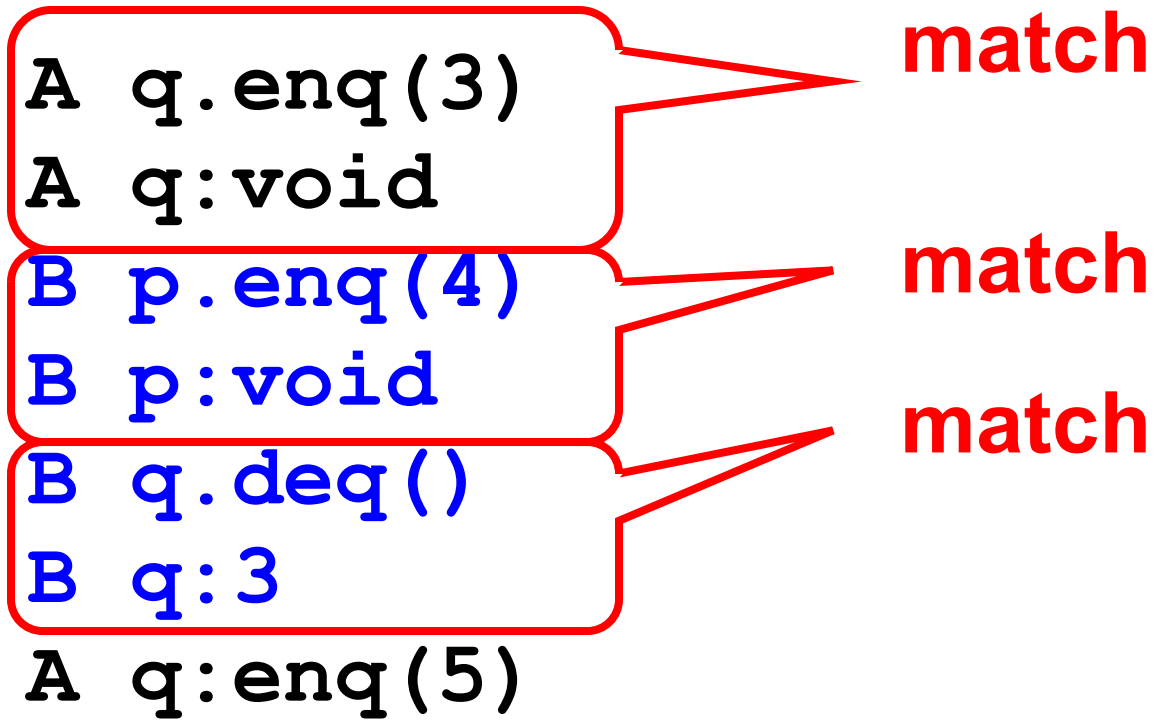
Sequential Histories



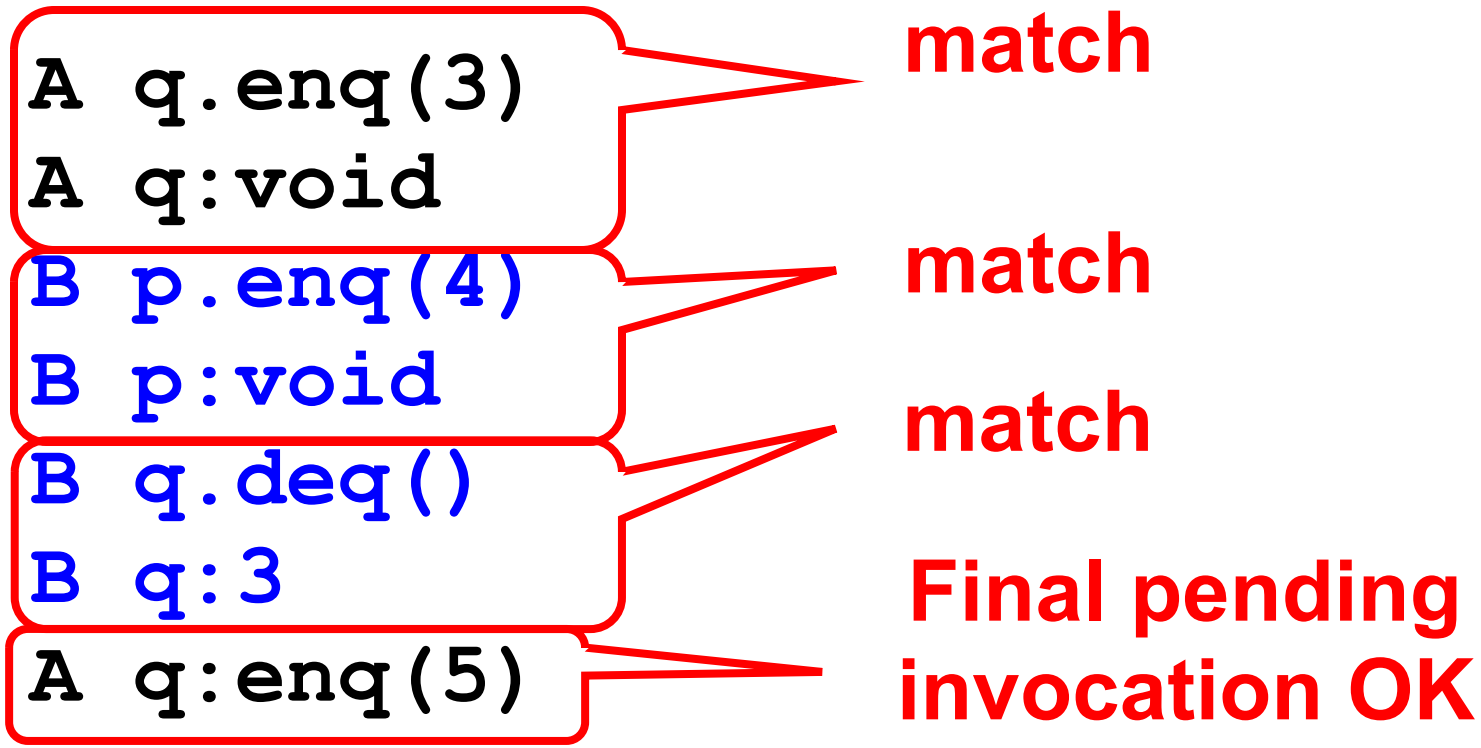
Sequential Histories



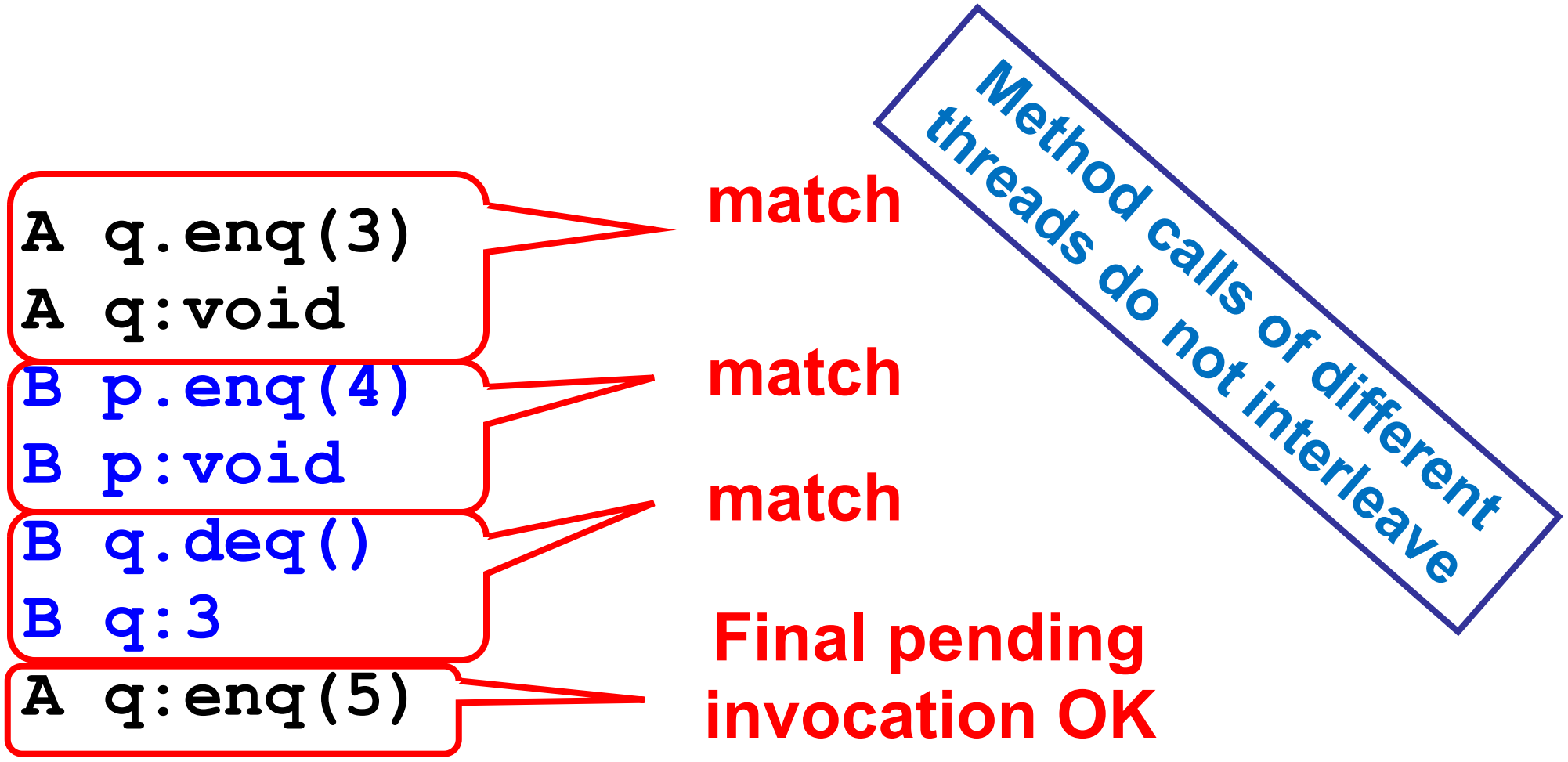
Sequential Histories



Sequential Histories



Sequential Histories



Well-Formed Histories

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

Well-Formed Histories

Per-thread projections 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
```

Well-Formed Histories

Per-thread projections 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

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

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

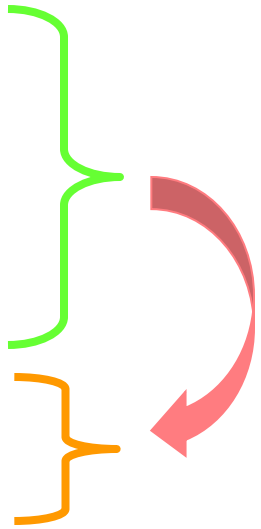
- A sequential specification is some way of telling whether a
  - Single-thread, single-object history
  - Is legal
- For example:
  - Pre and post-conditions
  - But plenty of other techniques exist …

Legal Histories

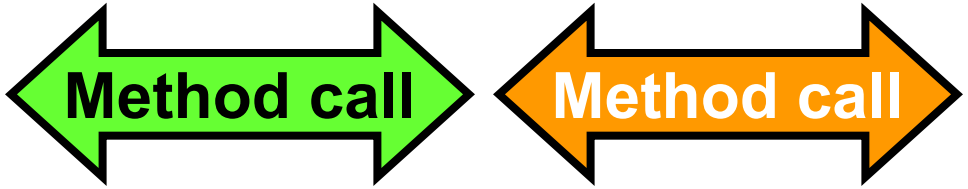
- A sequential (multi-object) history H is legal if
  - For every *object* **x**
  - **H|x** is in the sequential spec for **x**
  - *Not talking about threads now!*

Precedence

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



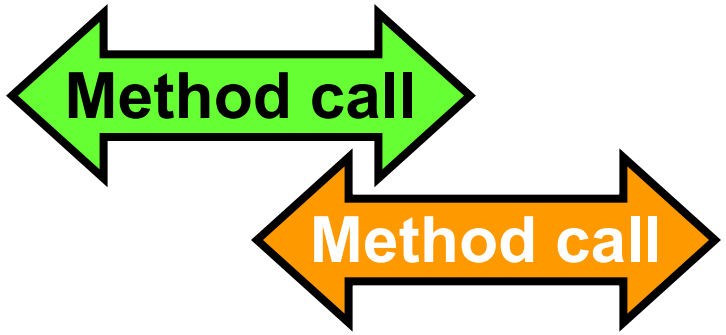
A method call precedes another if *response* event precedes *invocation* event



Non-Precedence

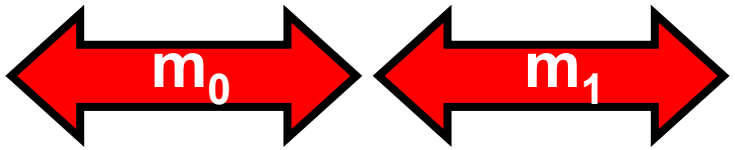
```
A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3
```

Some method calls overlap one another



Notation

- Given
  - History **H**
  - method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**
- We say $m_0 \rightarrow_{\mathbf{H}} m_1$, if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$
- Relation $\mathbf{m_0} \rightarrow_{\mathbf{H}} \mathbf{m_1}$ 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 $\rightarrow_G \subset \rightarrow_S$

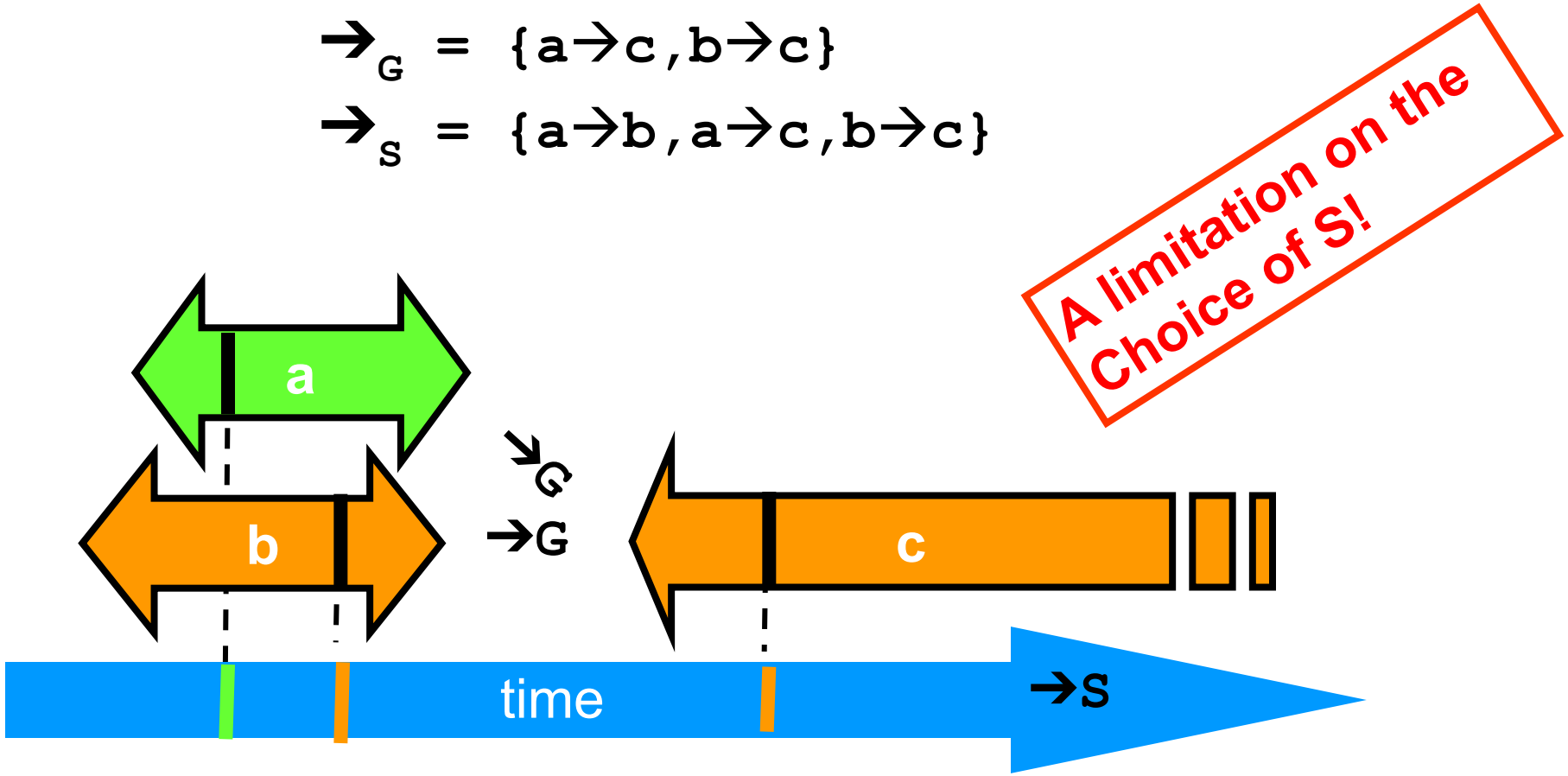
Remarks

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

Ensuring $\rightarrow_G \subset \rightarrow_S$

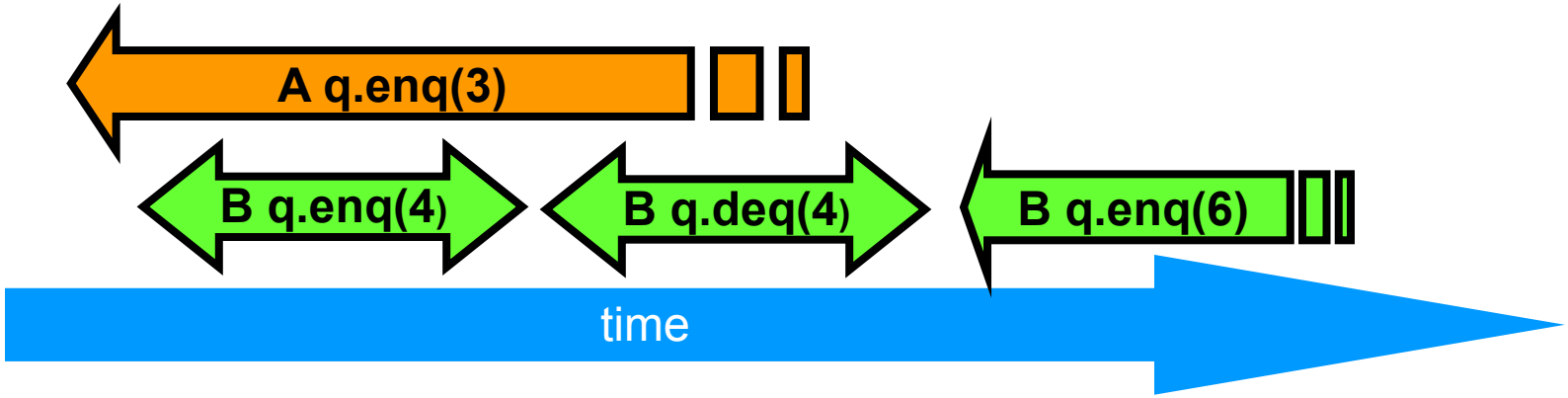
$\rightarrow_G$ = {a→c,b→c}

$\rightarrow_S$ = {a→b,a→c,b→c}



Example

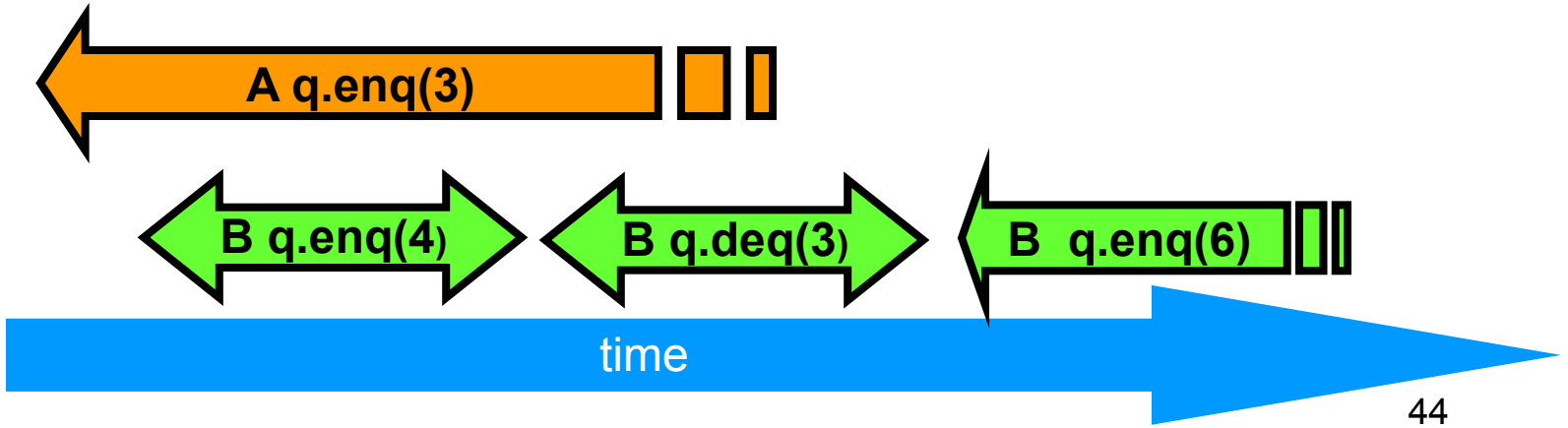
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



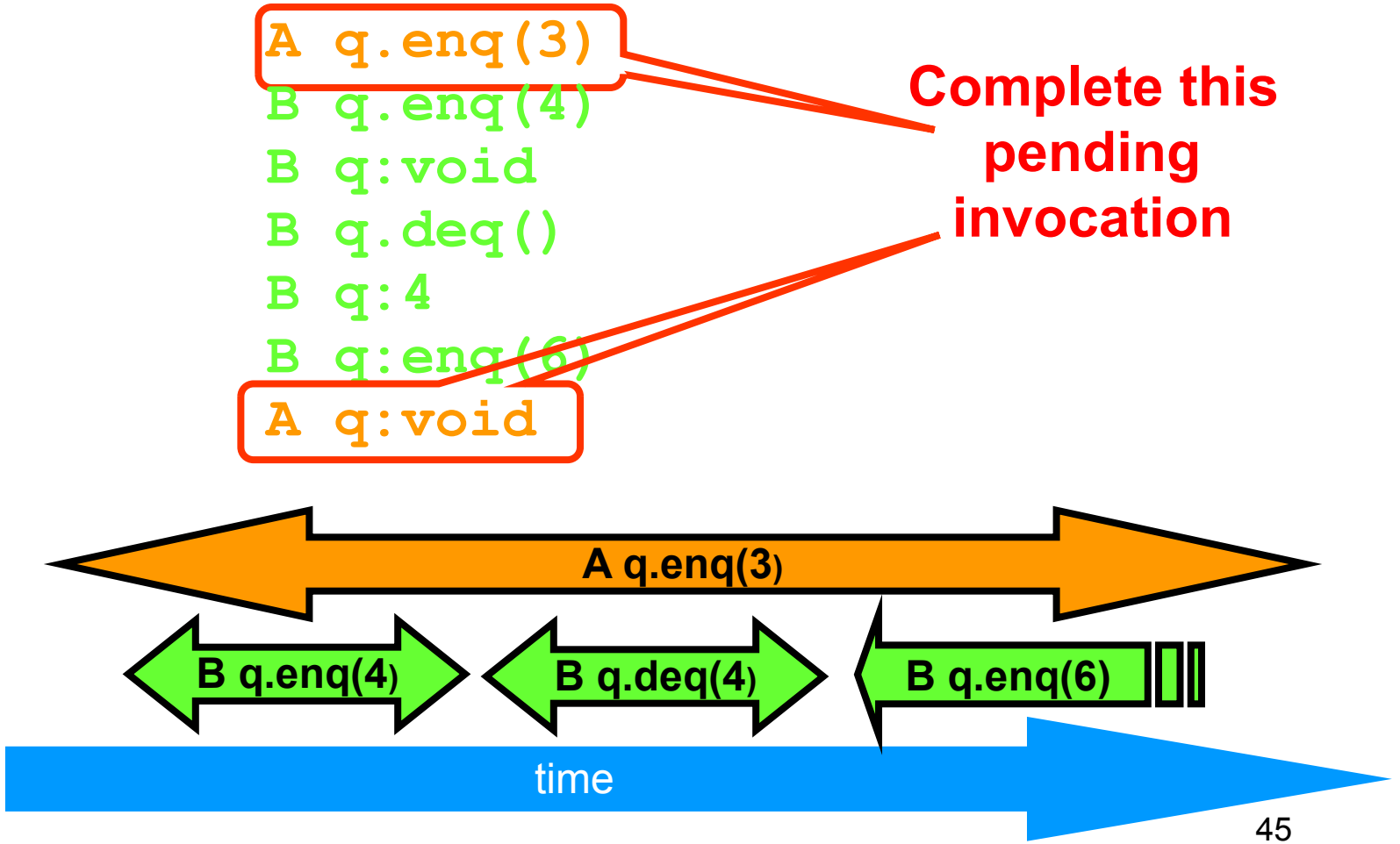
Example

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

Complete this pending invocation



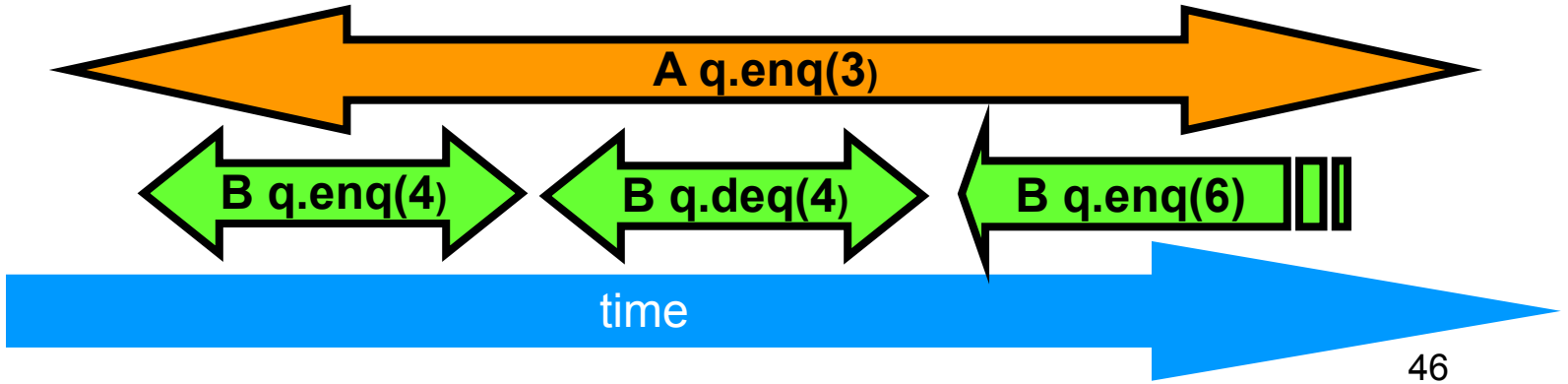
Example


A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
Complete this pending invocation
A q.enq(3)
B q.enq(4)
B q.deq(4)
B q.enq(6)
time

Example

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

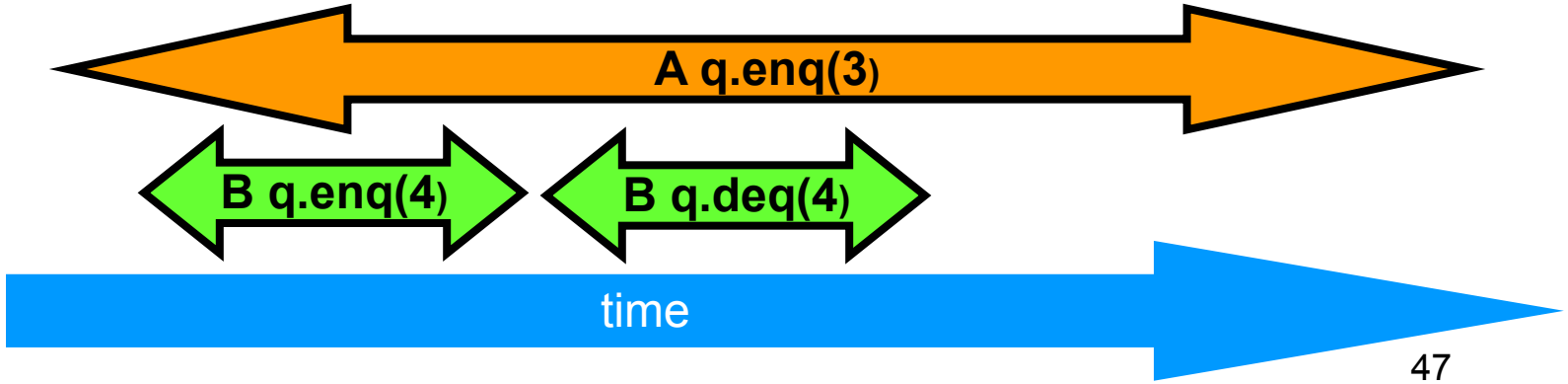
discard this one



Example

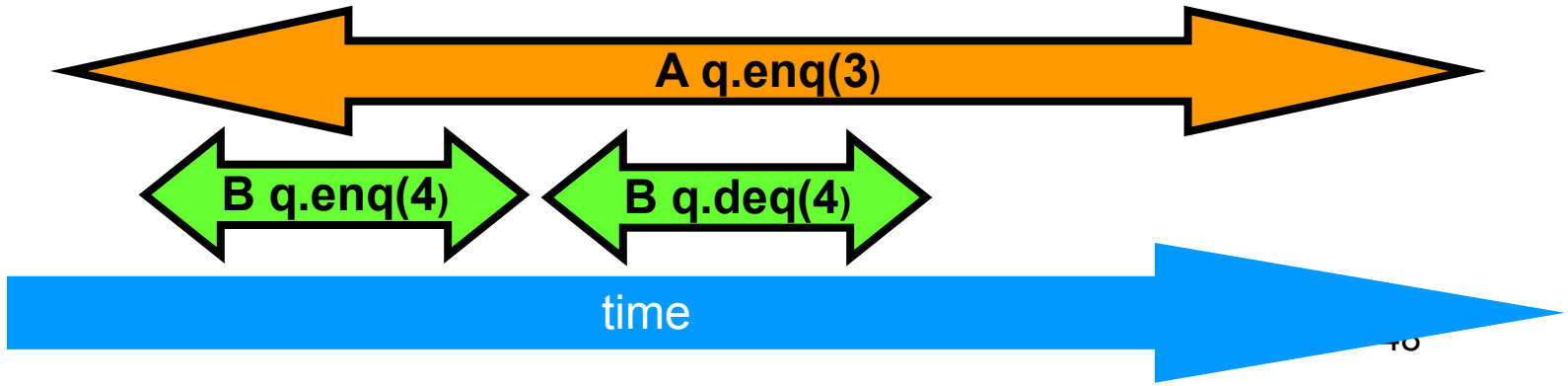
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4

A q:void
```



Example

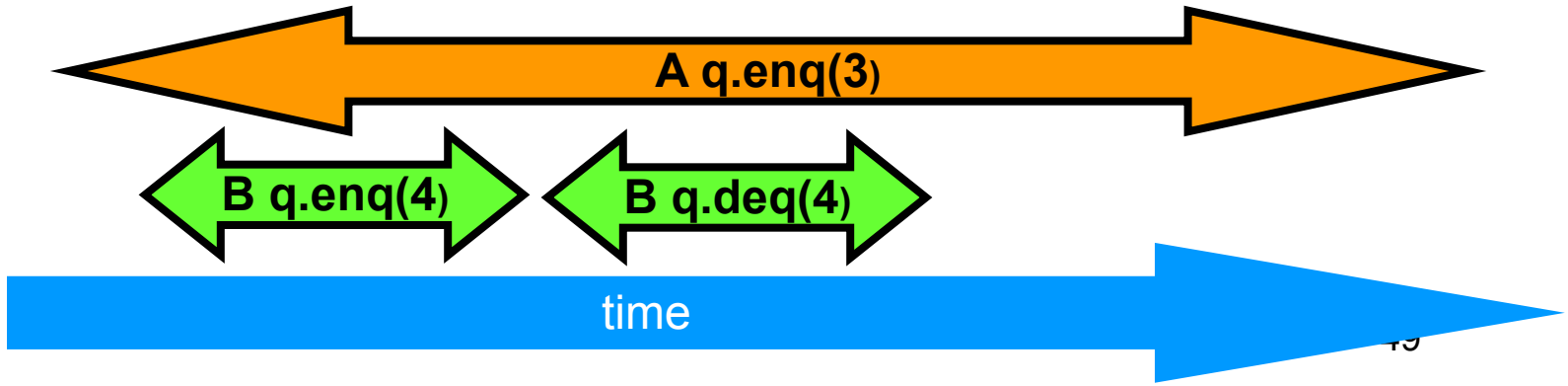
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



Example

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

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

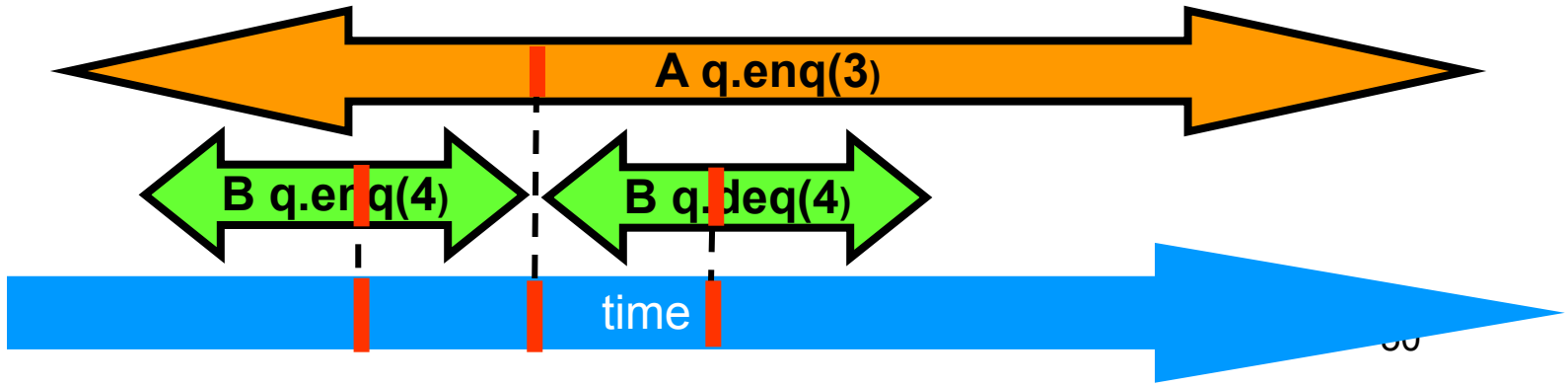


Example

```
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
```

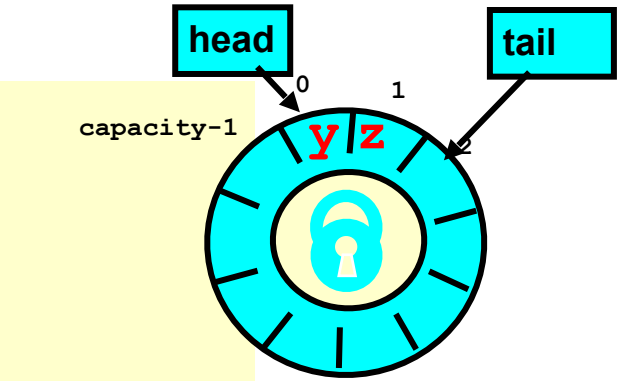


Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects
  - A history of two linearizable objects is linearizable

Reasoning About Linearizability: Locking

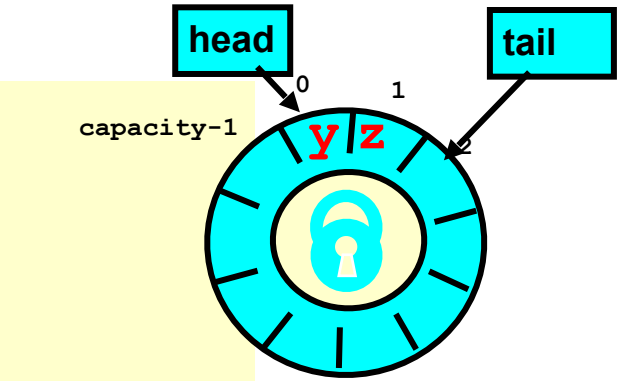
```
def deq() : T = {
  myLock.lock()
  try {
    if (tail == head) {
      throw EmptyException
    }
    val x = items(head % items.length)
    head = head + 1
    x
  } finally {
    myLock.unlock()
  }
}
```



Reasoning About Linearizability: Locking

```
def deq() : T = {
  myLock.lock()
  try {
    if (tail == head) {
      throw EmptyException
    }
    val x = items(head % items.length)
    head = head + 1
    x
  } finally {
    myLock.unlock()
  }
}
```

Linearization points are when locks are released



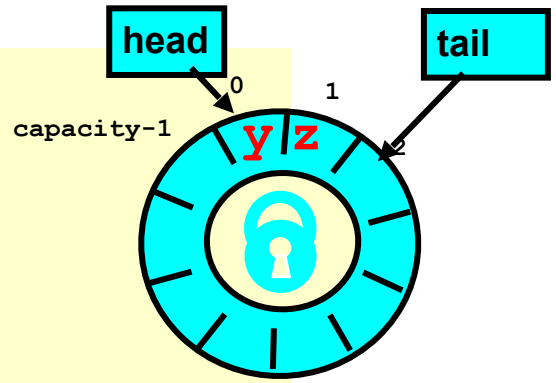
More Reasoning: Wait-free

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {

  @volatile
  private var head, tail: Int = 0
  private val items = new Array[T](capacity)

  def enq(x: T): Unit = {
    if (tail - head == items.length) throw FullException
    items(tail % items.length) = x
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```



More Reasoning: Wait-free

```
class LockFr ueue[T: ClassTag](val capacity: Int) {


  d, tail: Int = 0
  s = new Array[T](capacity)

  enq(x: T): Uni
    if (tail - head
    items(tail % ite
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```

Remember that there is only one enqueuer and only one dequeuer

Linearization order is order head and tail fields modified

Strategy

- Identify one atomic step where method “happens”
  - Critical section
  - Machine instruction
- Doesn’t always work
  - Might need to define several different steps for a given method

Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don’t leave home without it

Alternative: Sequential Consistency

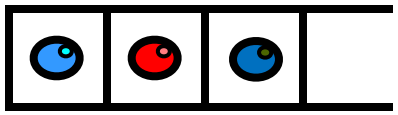
- History H is ***Sequentially Consistent*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to a
  - Legal sequential history **S**
  - Where $\rightarrow_G \subset \rightarrow_S$

Differs from linearizability

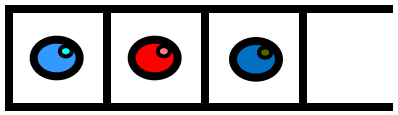
Sequential Consistency

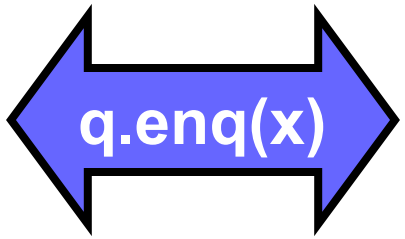
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

Example

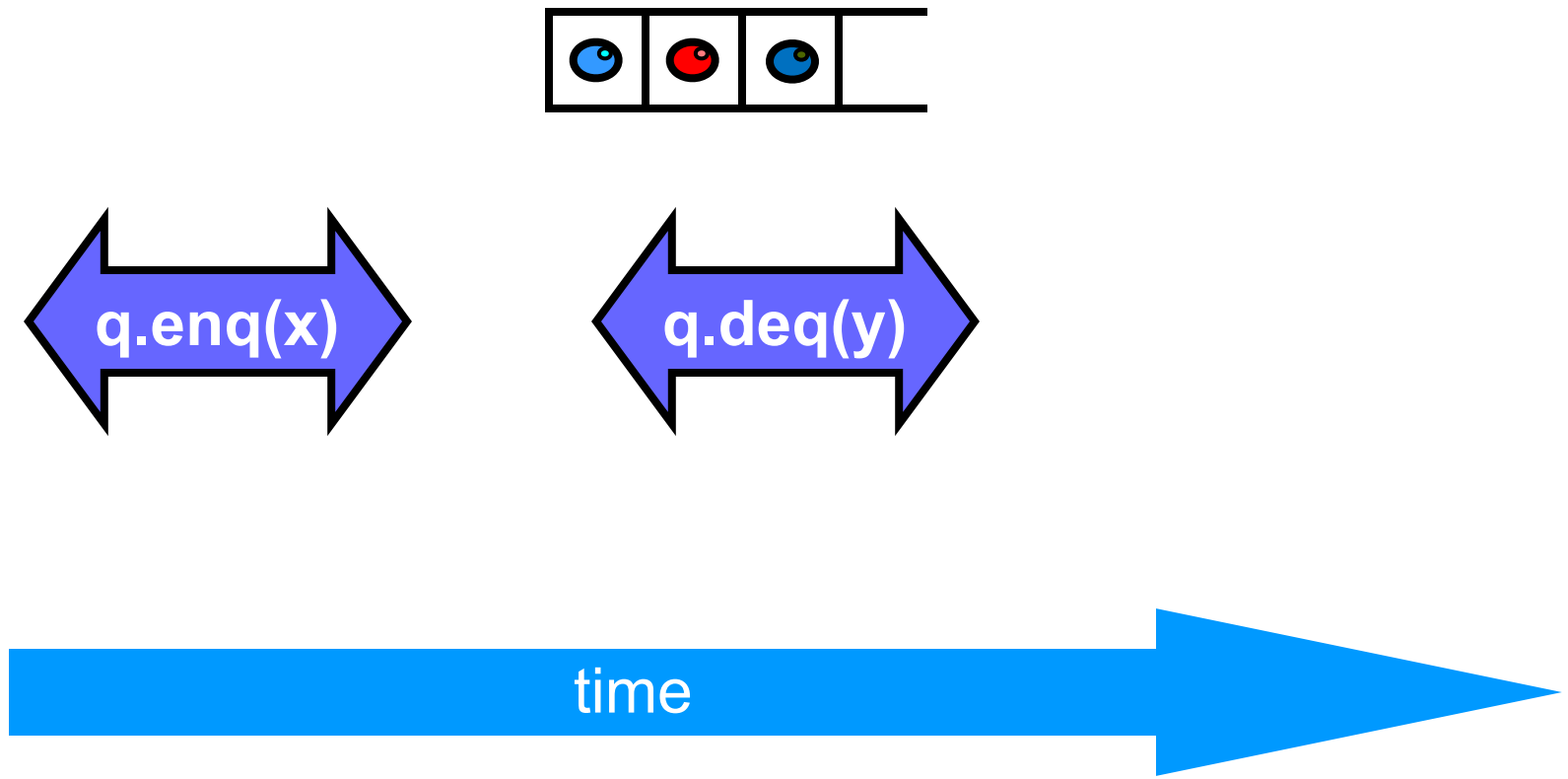


Example

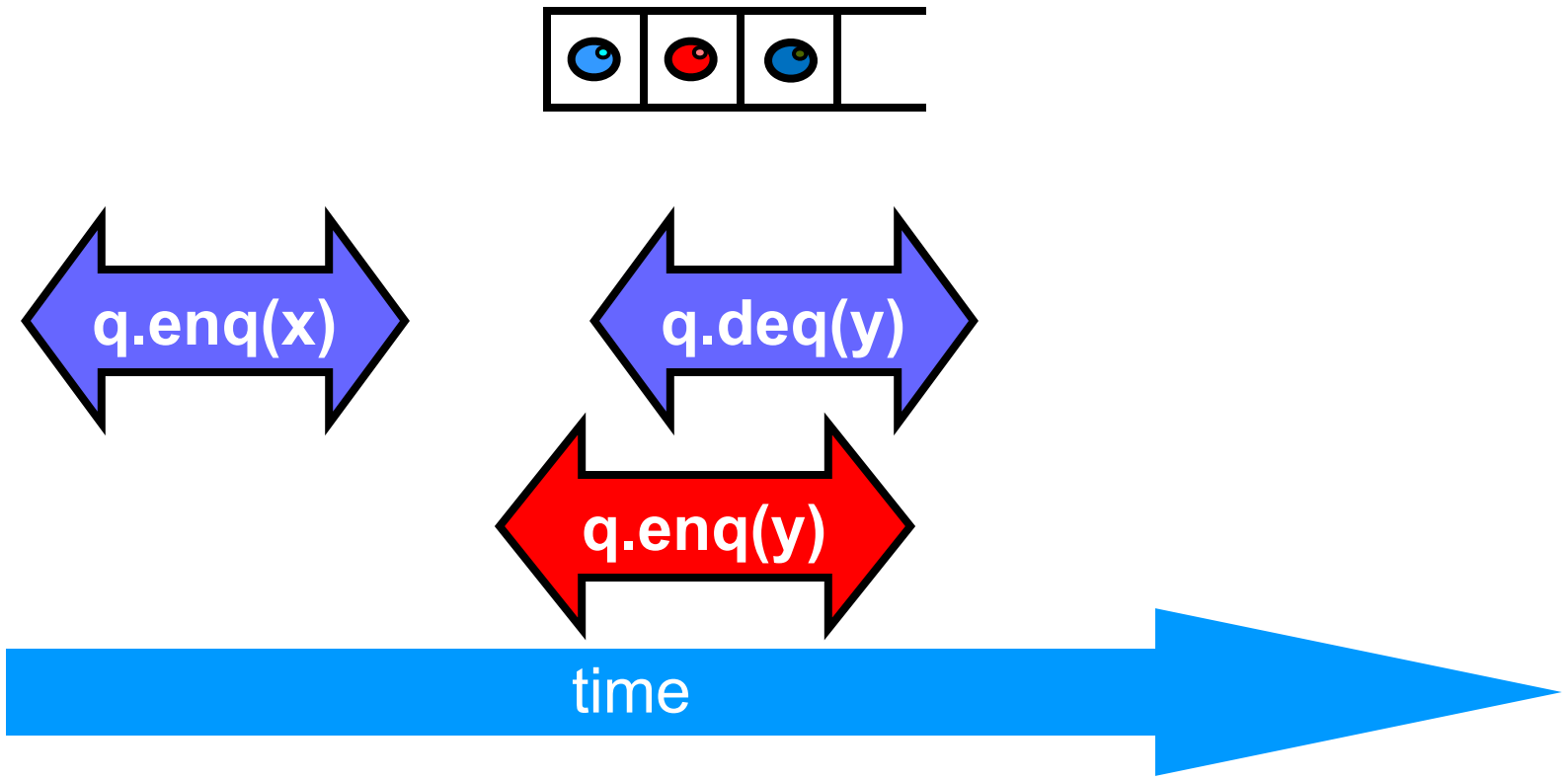




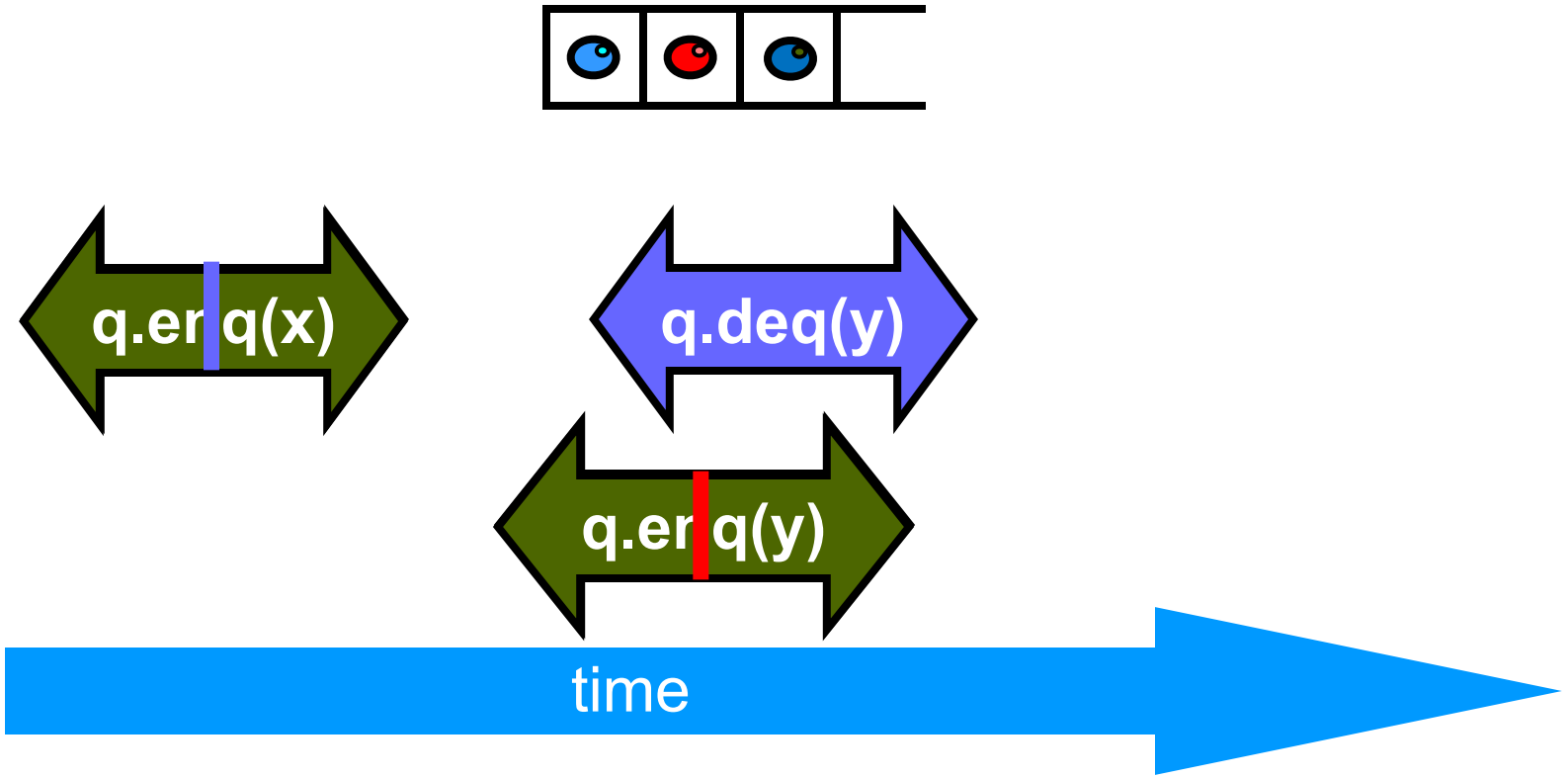
Example



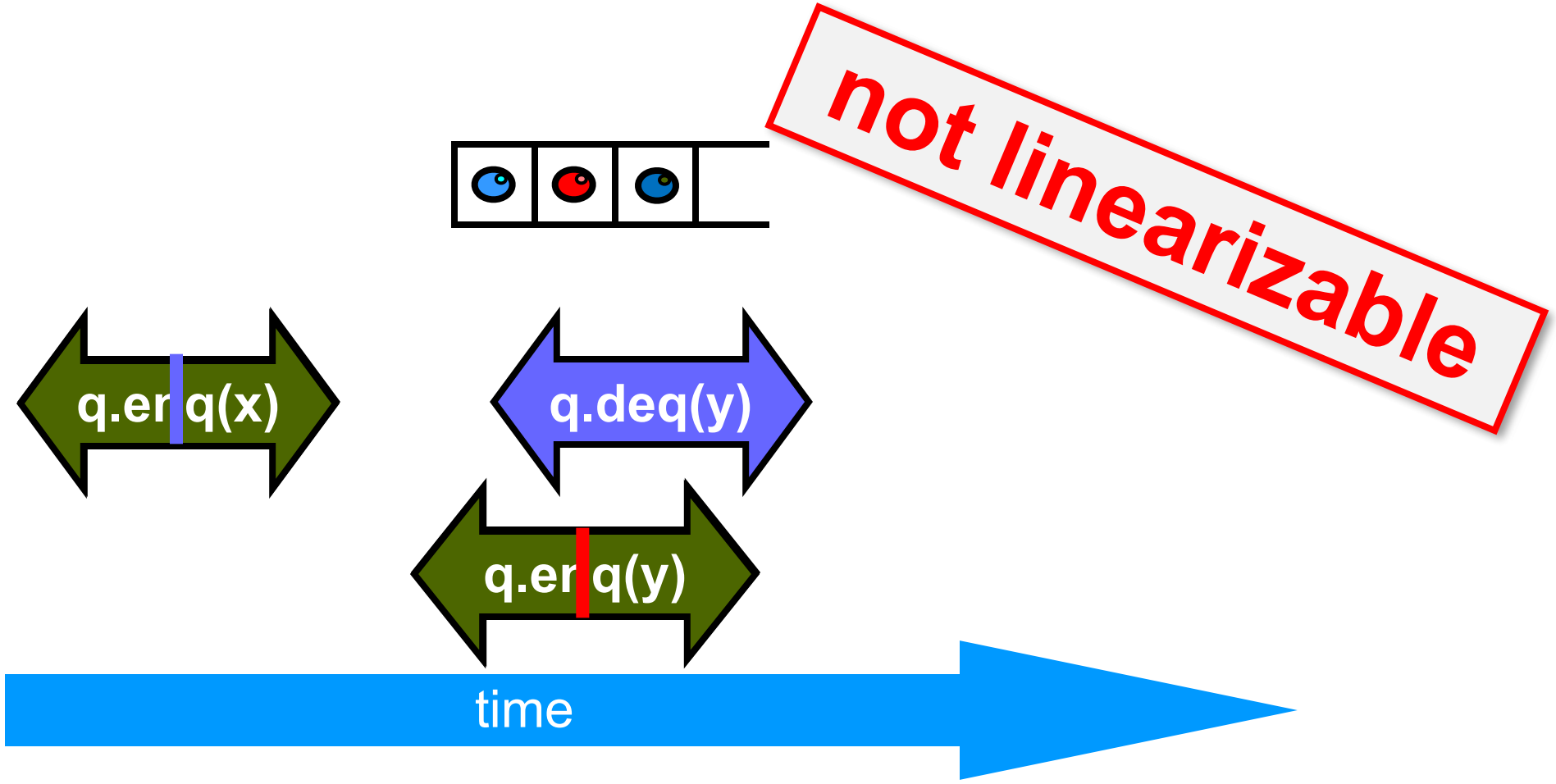
Example



Example

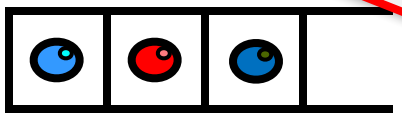


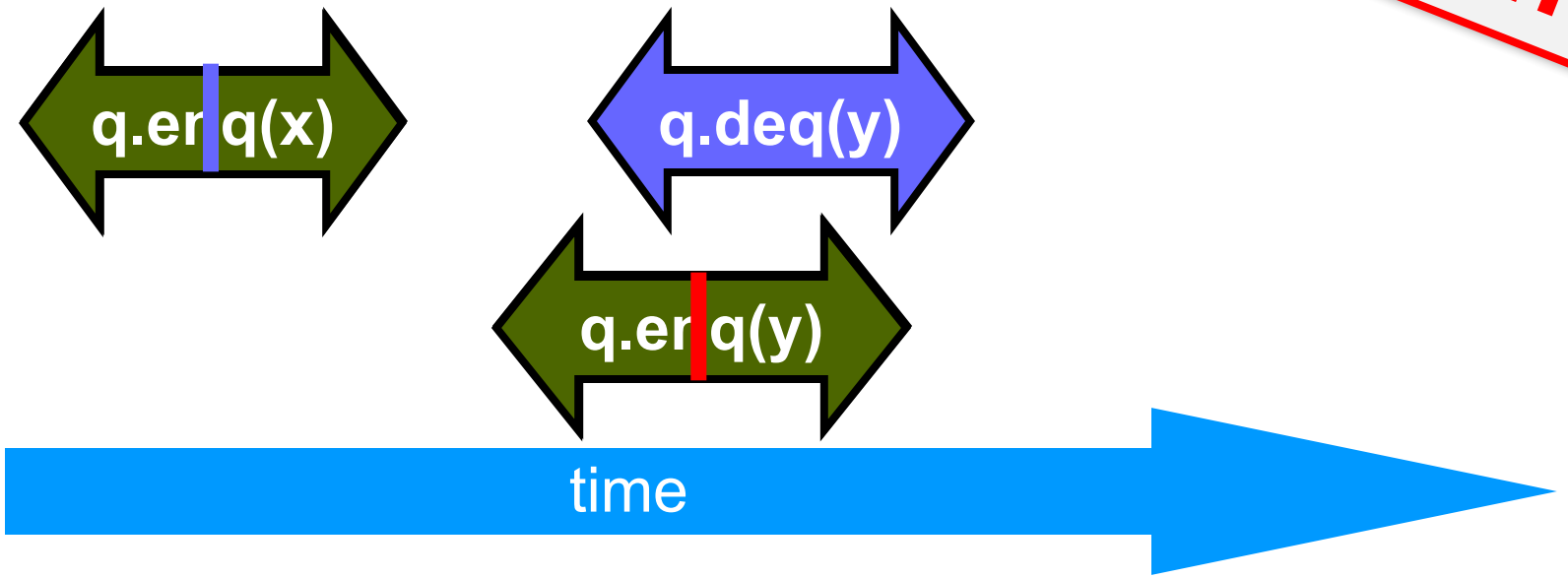
Example



Example

Yet Sequentially Consistent

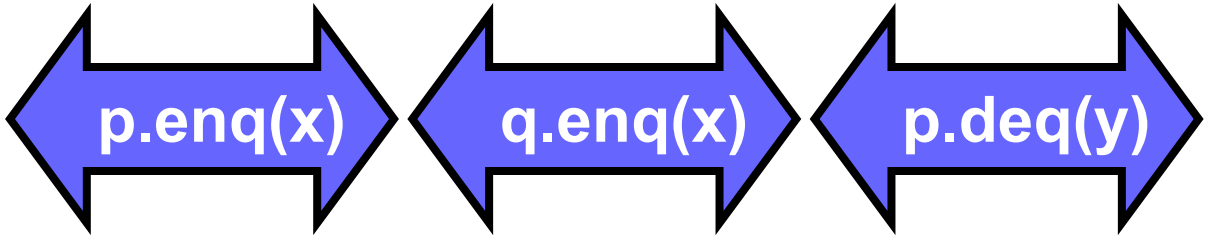




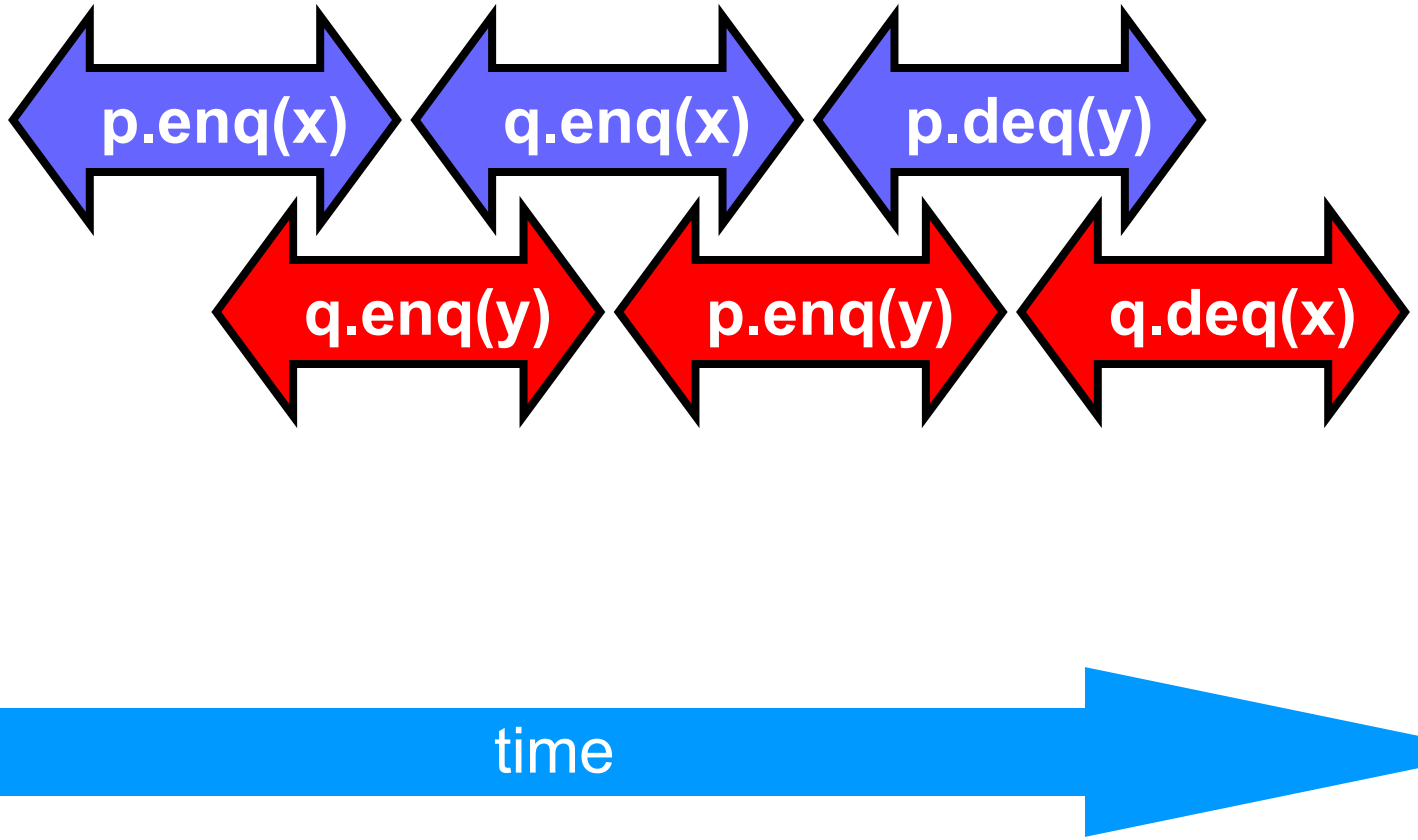
Theorem

Sequential Consistency is not composable

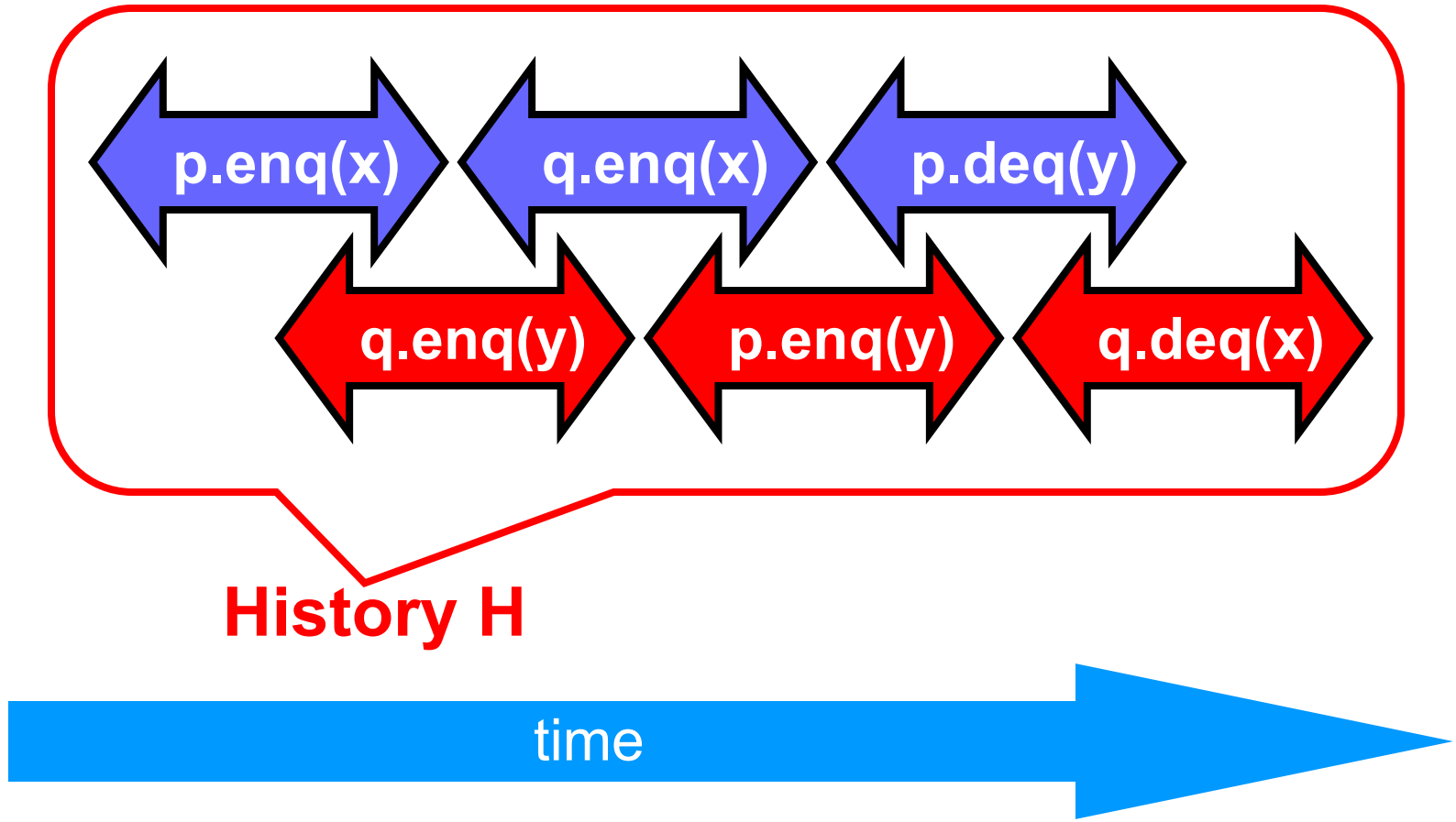
FIFO Queue Example



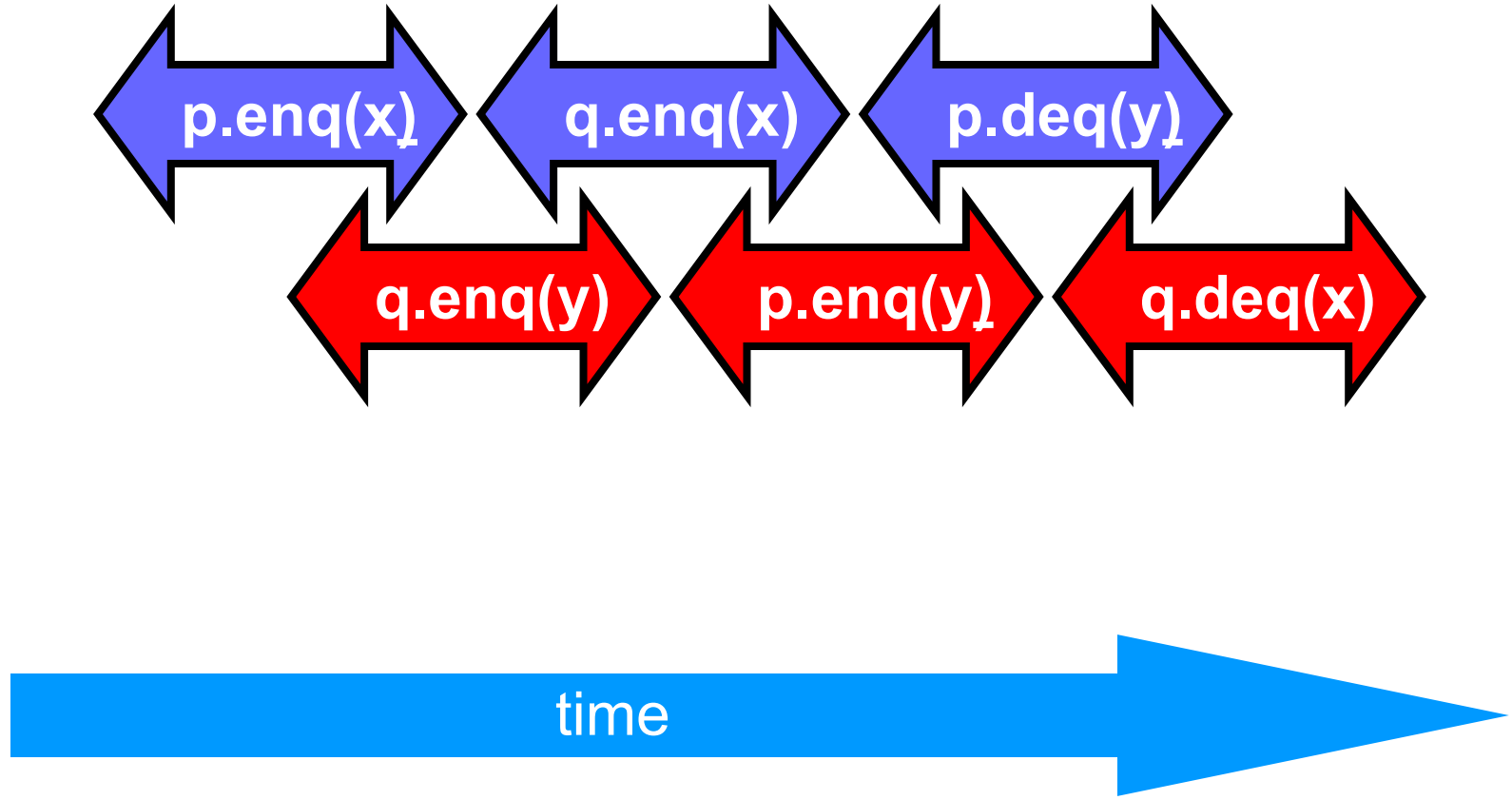
FIFO Queue Example



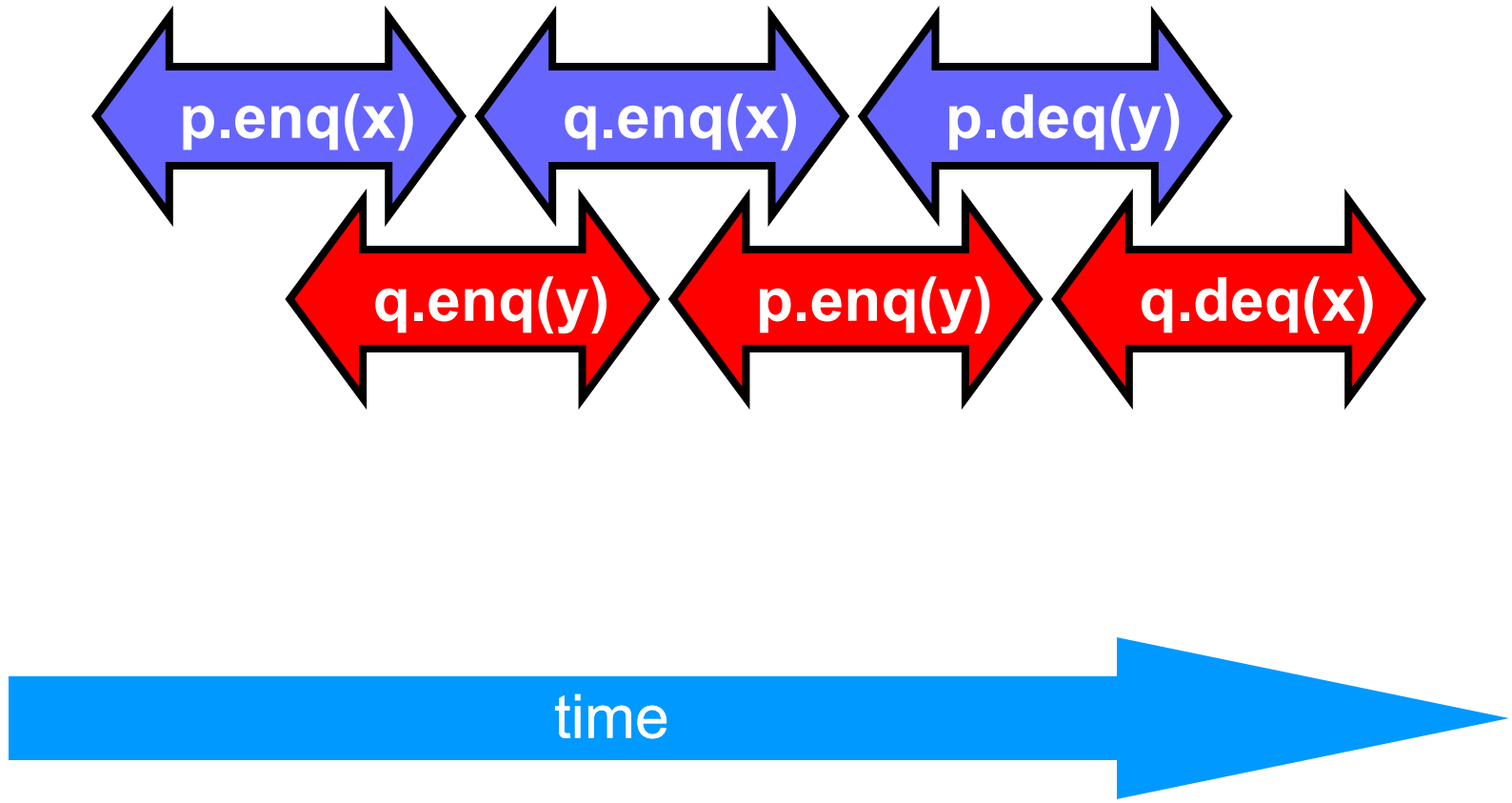
FIFO Queue Example



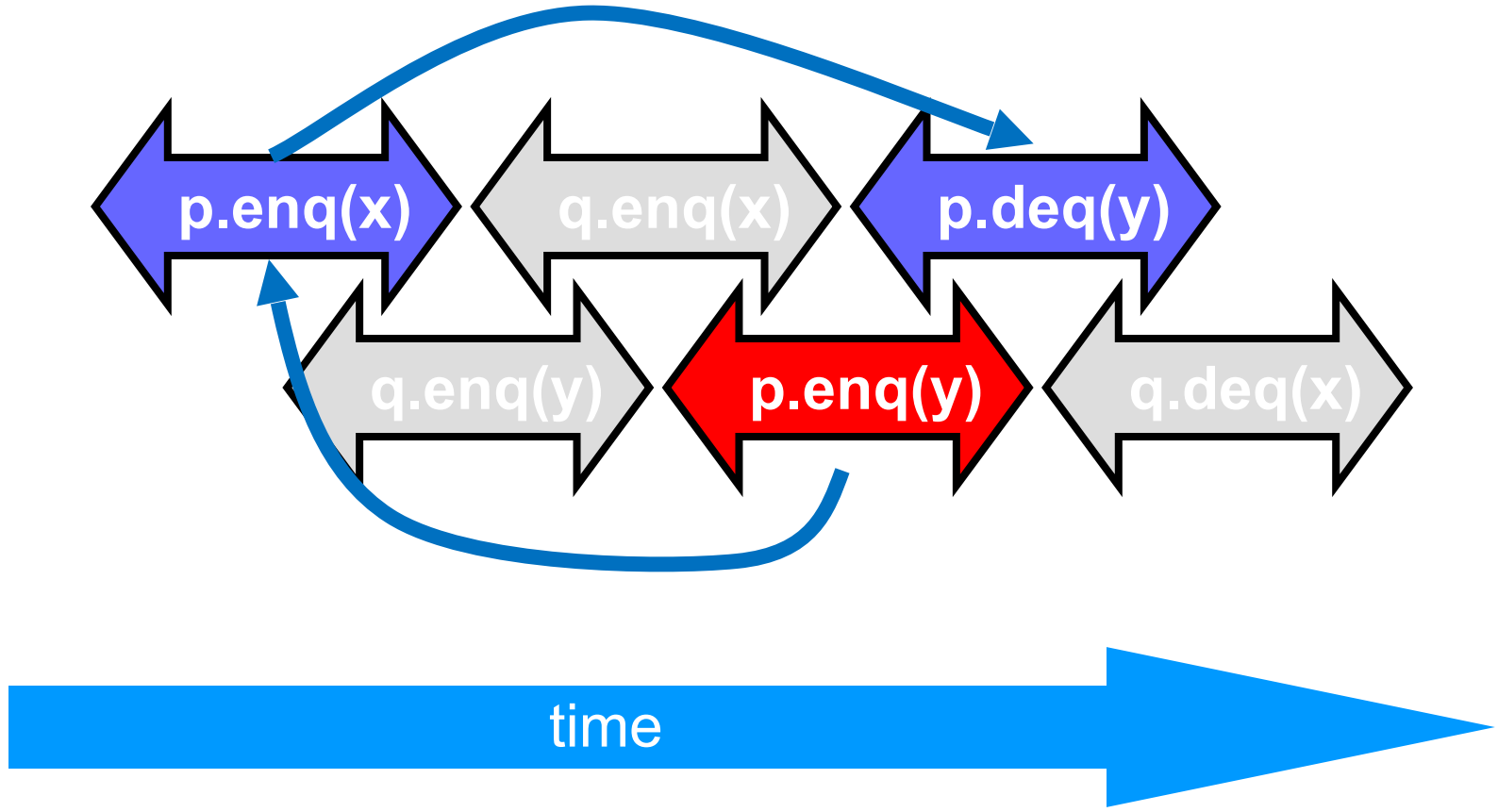
H|p Sequentially Consistent



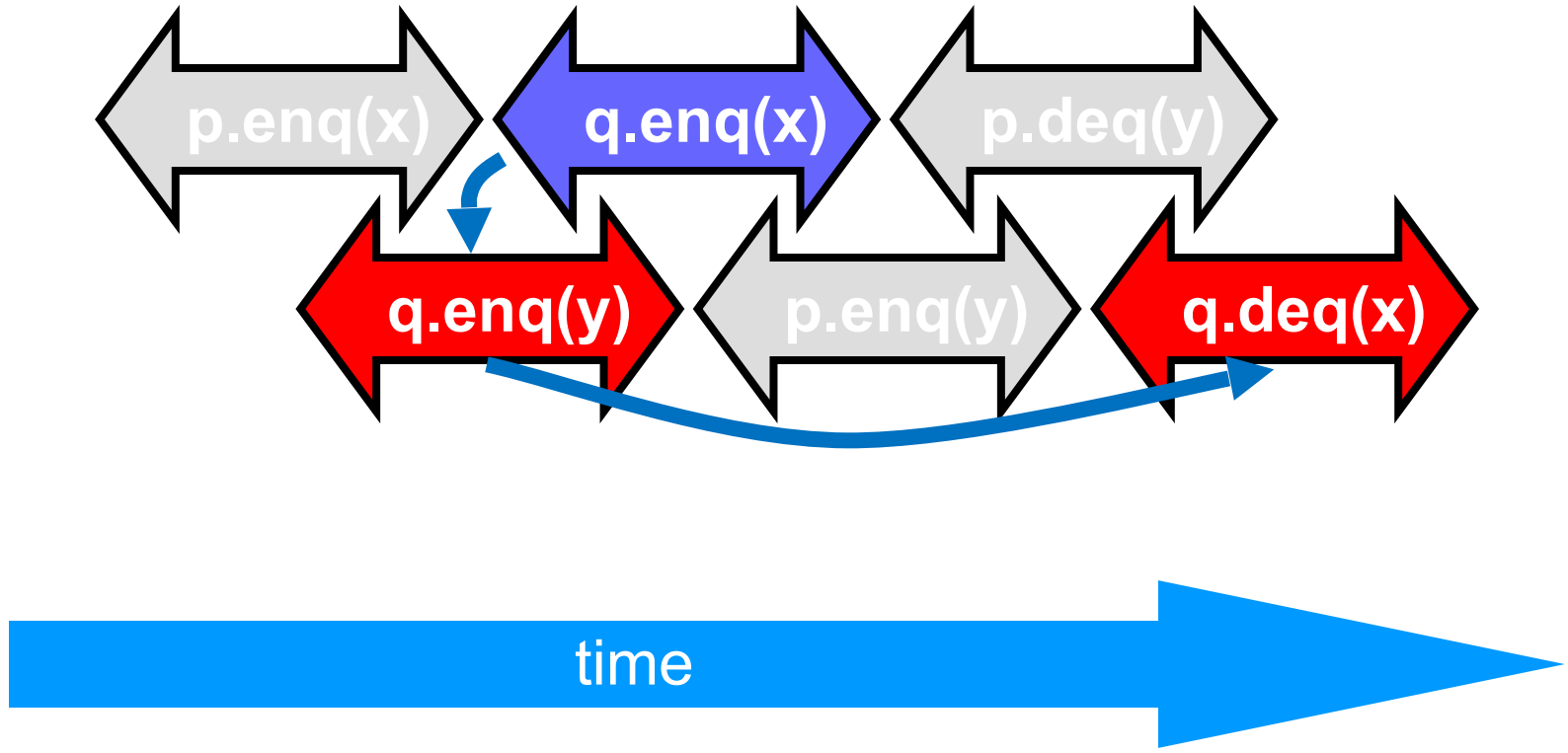
H|q Sequentially Consistent



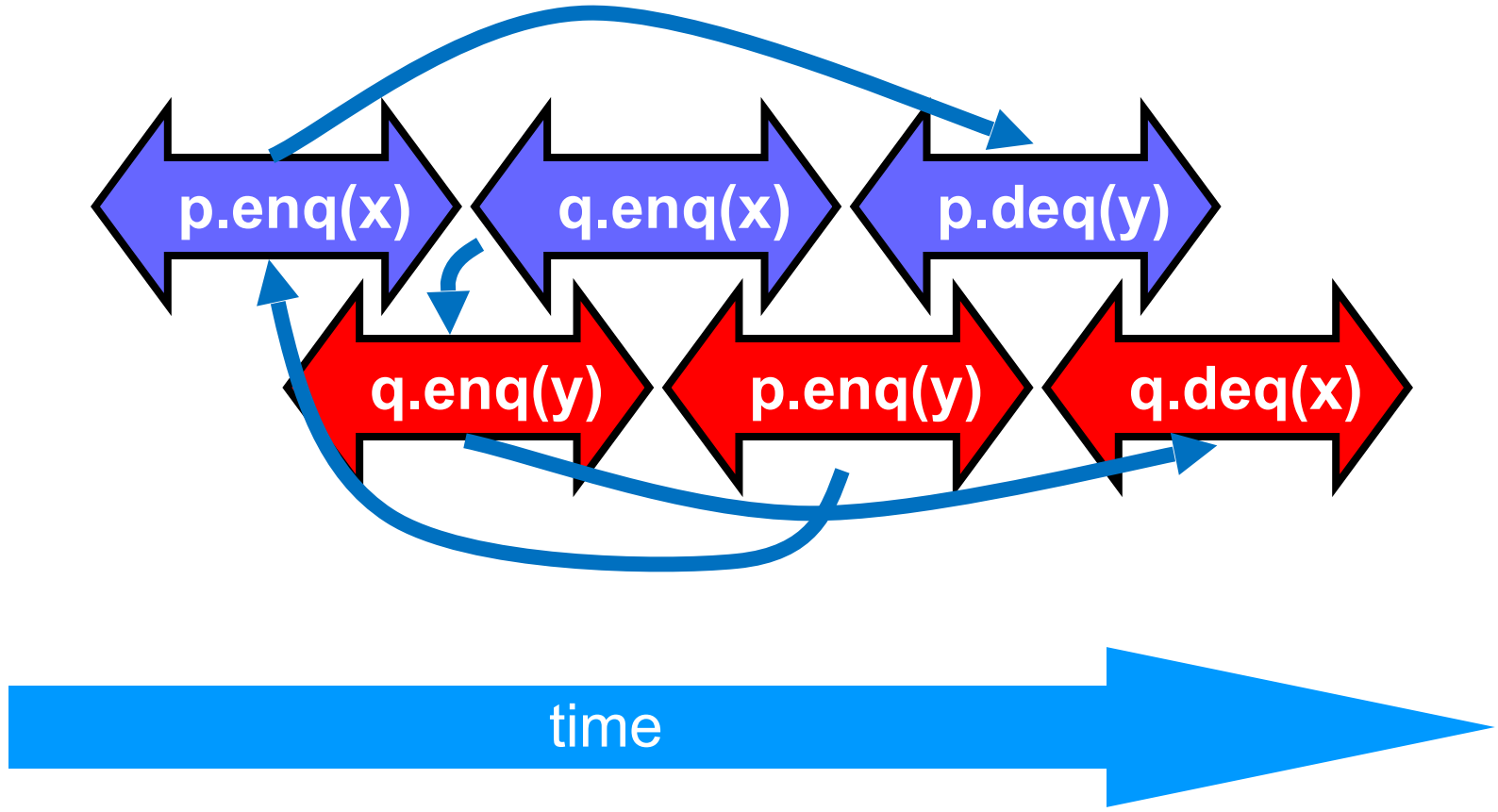
Ordering imposed by p



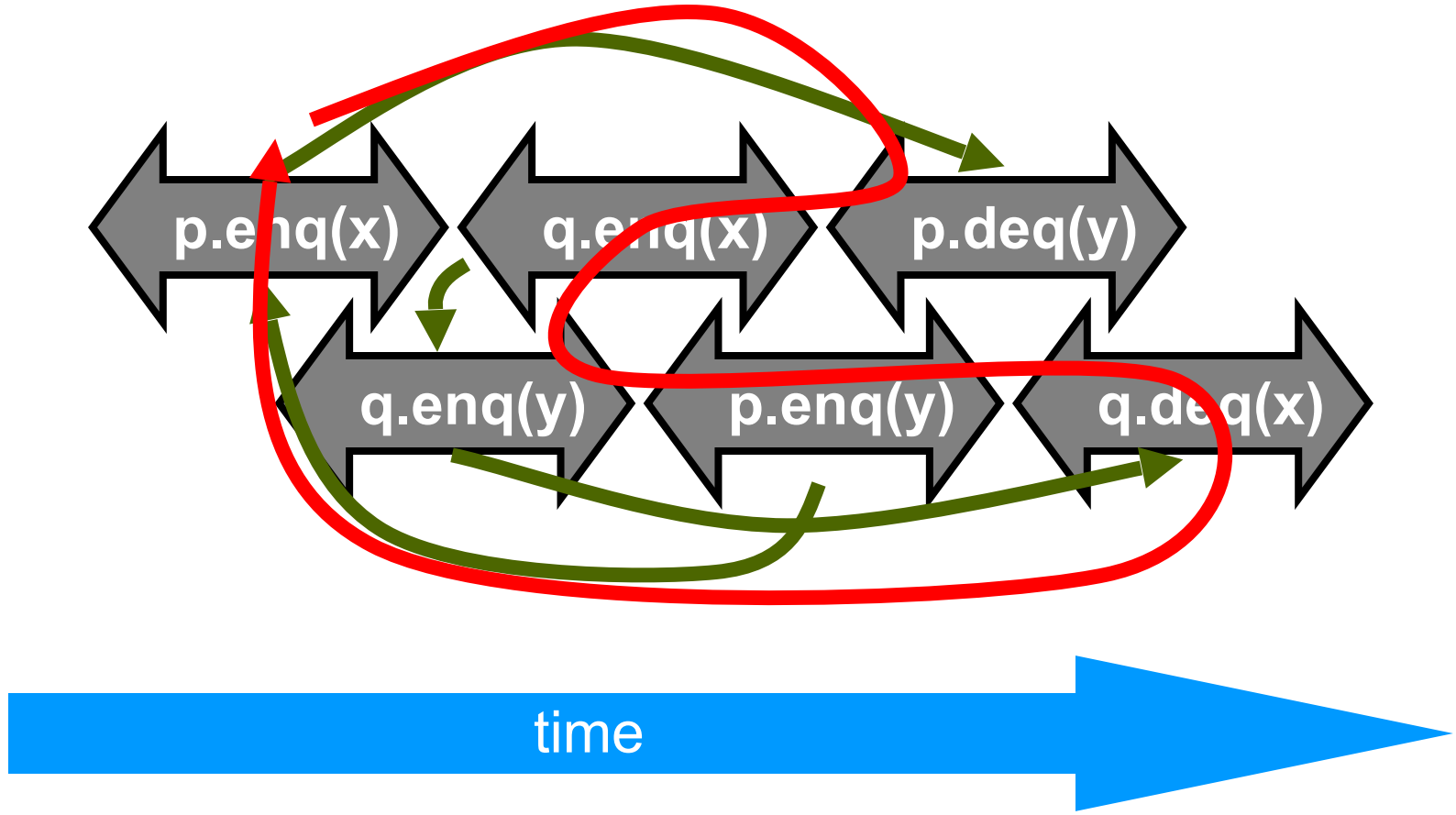
Ordering imposed by q



Ordering imposed by both



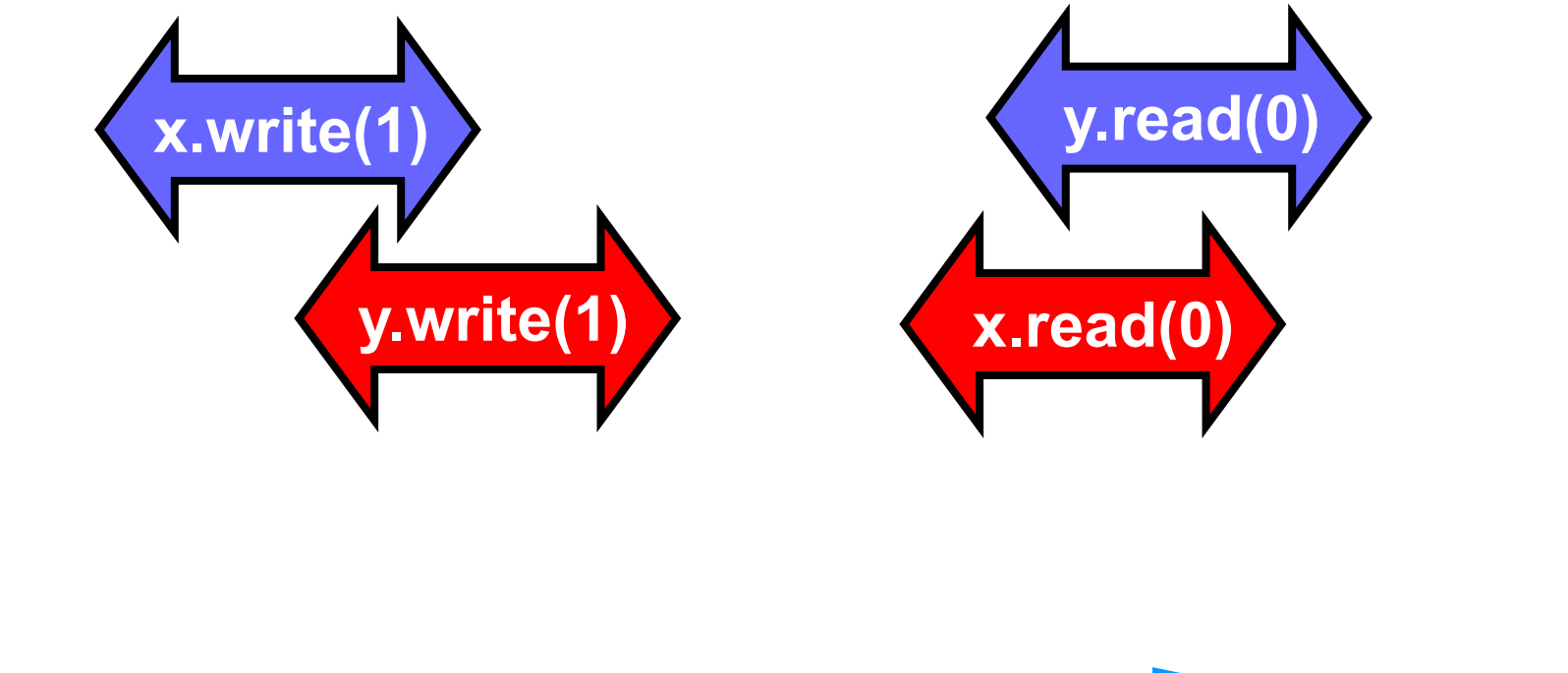
Combining orders



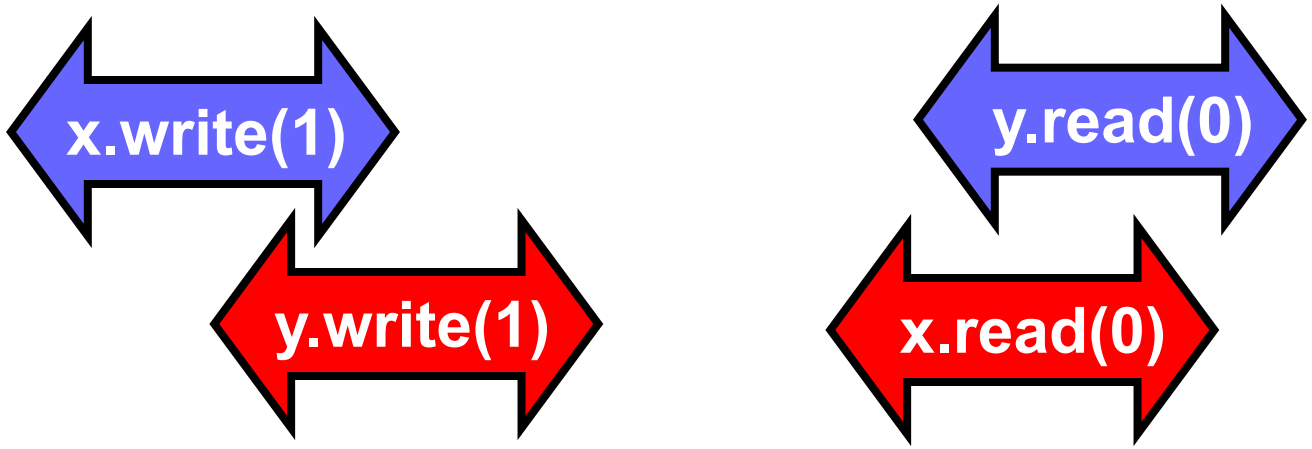
Fact

- Most hardware architectures don’t support sequential consistency
- Because they think it’s too strong
- Here’s another story …

The Flag Example

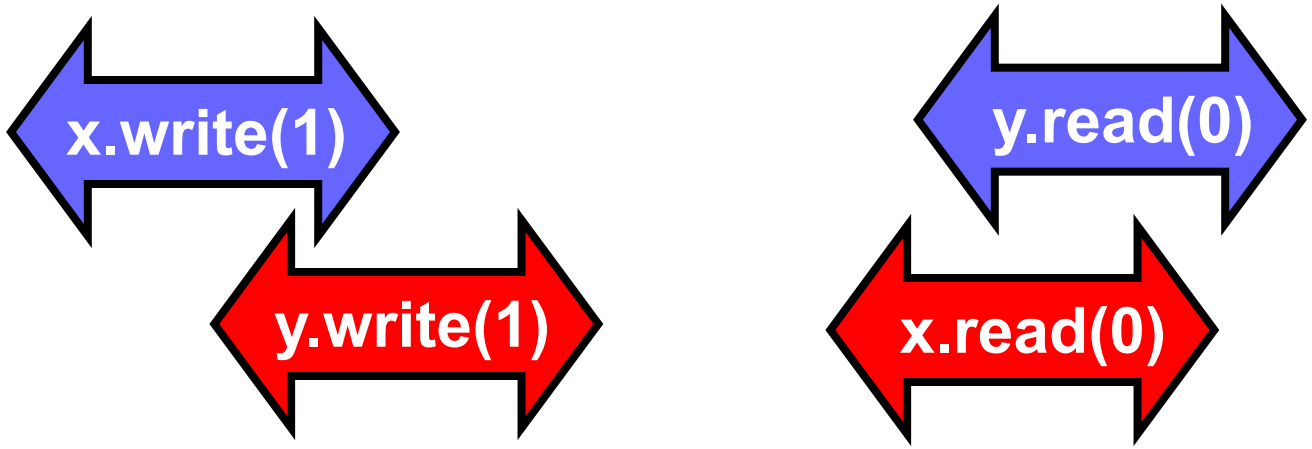


The Flag Example



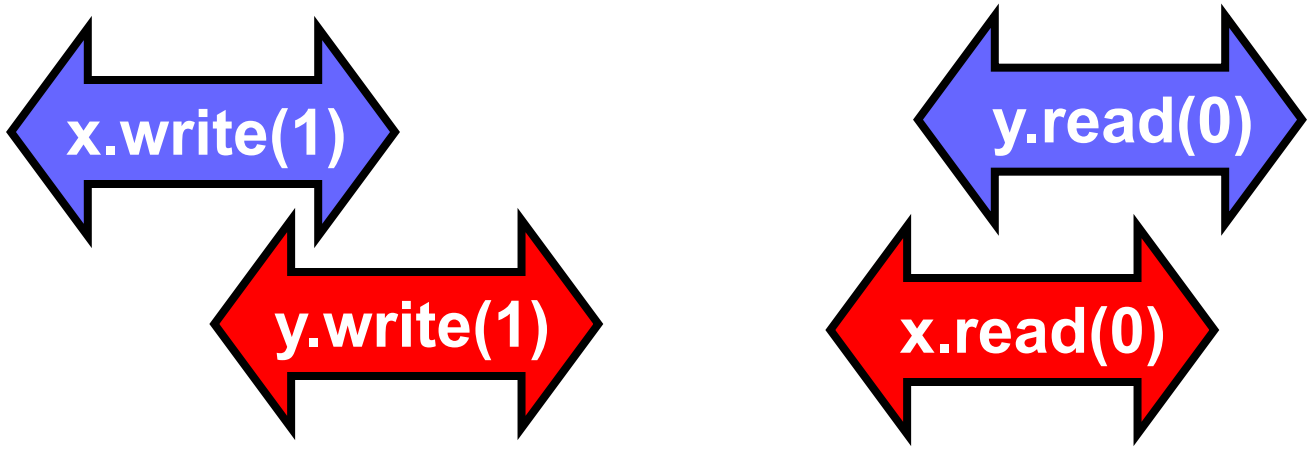
- Each thread's view is sequentially consistent
  - It went first

The Flag Example



- Entire history isn't sequentially consistent
  - Can't both go first

The Flag Example



- Is this behavior really so wrong?
  - We can argue either way …

Opinion: It's 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's non-negotiable!

Peterson's Algorithm

```
def lock(): Unit = {
  flag(i) = true
  victim = i
  while (flag(1 - i) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Crux of Peterson Proof

(1) $write_B(flag[B]=true)$➔

(3) $write_B(victim=B)$➔

(2) $write_A(victim=A)$➔$read_A(flag[B])$

➔ $read_A(victim)$

Crux of Peterson Proof

(1) $write_B(flag[B]=true)$ →

(3) $write_B(victim=B)$ →

(2) $write_A(victim=A)$ → $read_A(flag[B])$

→ $read_A(victim)$

Observation: proof relied on fact that if a location is stored, a later load by some thread will return this or a later stored value.

Opinion: 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

Hardware Consistency

Initially, a = b = 0.

Processor 0

```
mov 1, a     ;Store
mov b, %ebx  ;Load
```

Processor 1

```
mov 1, b     ;Store
mov a, %eax  ;Load
```

What are the final possible values of %eax and %ebx after both processors have executed?

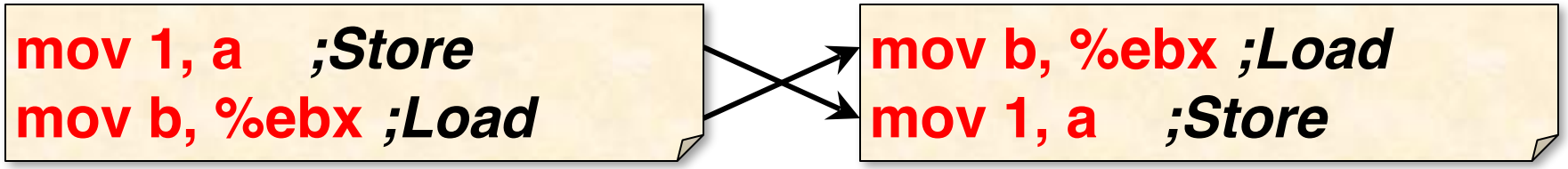
Sequential consistency implies that no execution ends with %eax= %ebx = 0

Hardware Consistency

- No modern-day processor implements sequential consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
- Why?
- Because most of performance is derived from a single thread's unsynchronized execution of code!

This is known as **Weak (Relaxed) Memory Semantics**

Weak-Memory Instruction Reordering



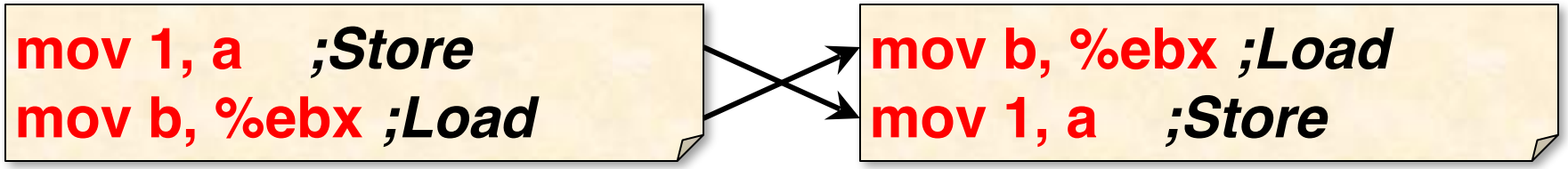
Program Order Execution Order

Q. Why might the hardware or compiler decide to reorder these instructions?

A. To obtain higher performance by covering load latency — *instruction-level parallelism*.

Slide used with permission of
Charles E. Leiserson

Weak-Memory Instruction Reordering



Program Order Execution Order

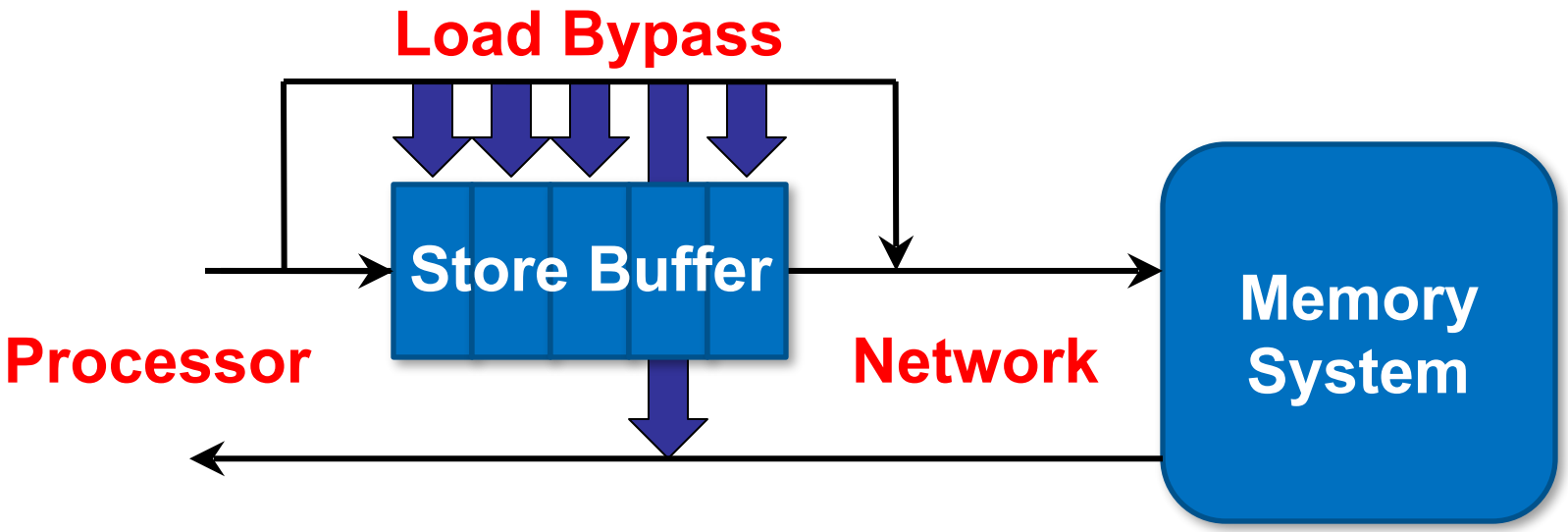
Q. When is it safe for the hardware or compiler to perform this reordering?

A. When a ≠ b.

A′. And there's no concurrency.

Slide used with permission of
Charles E. Leiserson

Hardware Reordering



- Processor can issue stores faster than the network can handle them ⇒ store buffer.
- Loads take priority, bypassing the store buffer.
- Except if a load address matches an address in the store buffer, the store buffer returns the result.

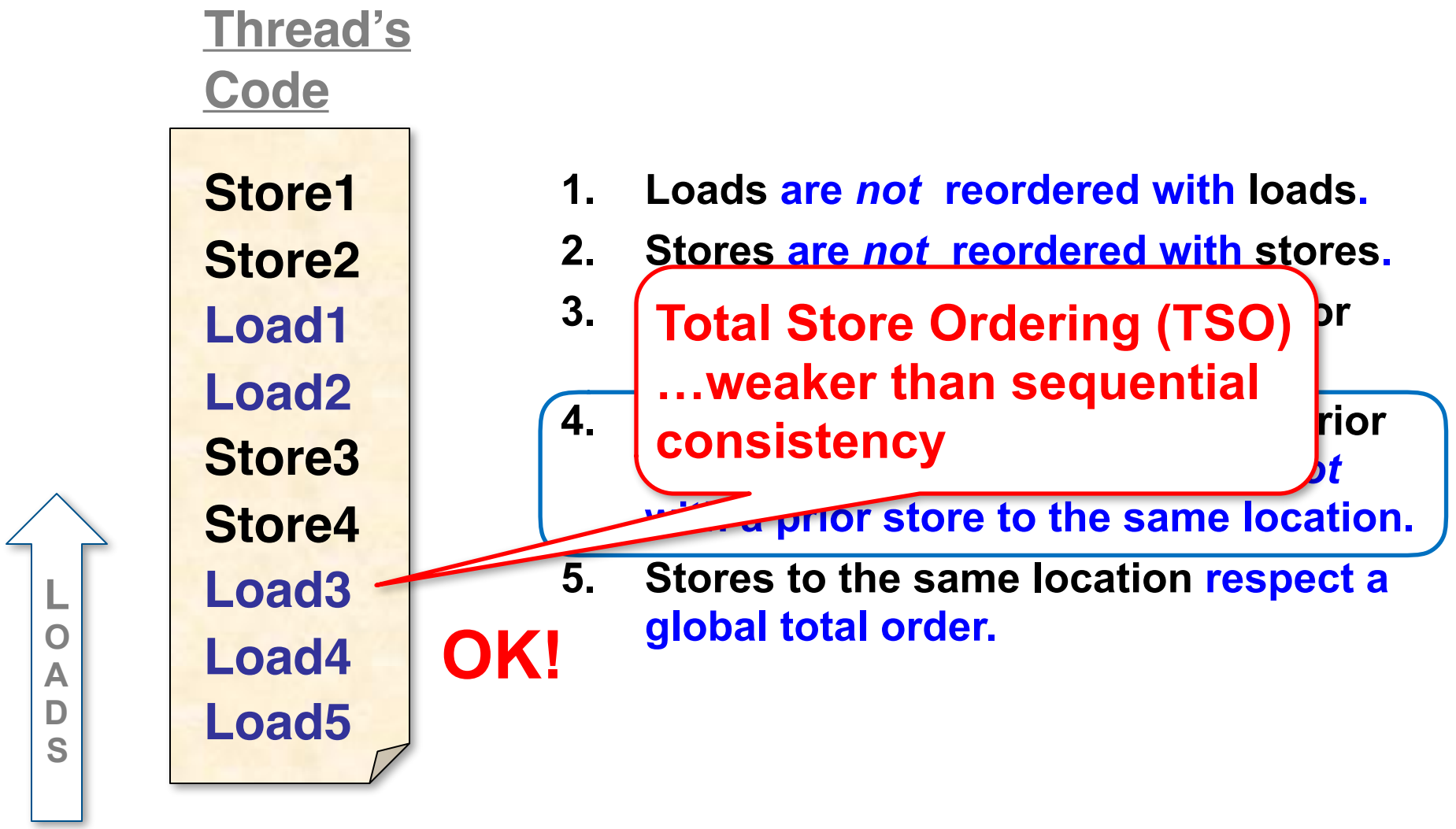
Slide used with permission of
Charles E. Leiserson

X86 Relaxed Memory Model

Thread's Code

1. Loads are *not* reordered with loads.
2. Stores are *not* reordered with stores.
3. Stores are *not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location but *not* with a prior store to the same location.
5. Stores to the same location respect a global total order.

X86 Relaxed Memory Model



Memory Barriers (Fences)

- A *memory barrier* (or *memory fence*) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.
- A memory barrier can be issued explicitly as an instruction (x86: mfence)
- The typical cost of a memory fence is comparable to that of an L2-cache access.

X86 Relaxed Memory Model

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5

1. Loads
2. Store
3. Store
   loads
4. A loa
   store
   with a pr
5. Stores to the same location respect a global total order.

Total Store Ordering +
properly placed memory barriers
= sequential consistency

Memory Barriers

- Explicit Synchronization
- Memory barrier will
  - Flush write buffer
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections

Java/Scala Volatile Variables

- In Java, can ask compiler to keep a variable up-to-date by declaring it **volatile**
- In Scala, use **@volatile** annotation
- Adds a memory barrier after each store
- Inhibits reordering, removing from loops, & other “compiler optimizations”

Demo: Realistic Locks

Summary: Real-World

- Hardware weaker than sequential consistency
- Can get sequential consistency at a price
- Linearizability better fit for high-level software

Linearizability

- Linearizability
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composablity

Summary: Correctness

- Sequential Consistency
  - Not composable
  - Harder to work with
  - Good way to think about hardware models

- We will use *linearizability* as our consistency condition in the remainder of this course unless stated otherwise