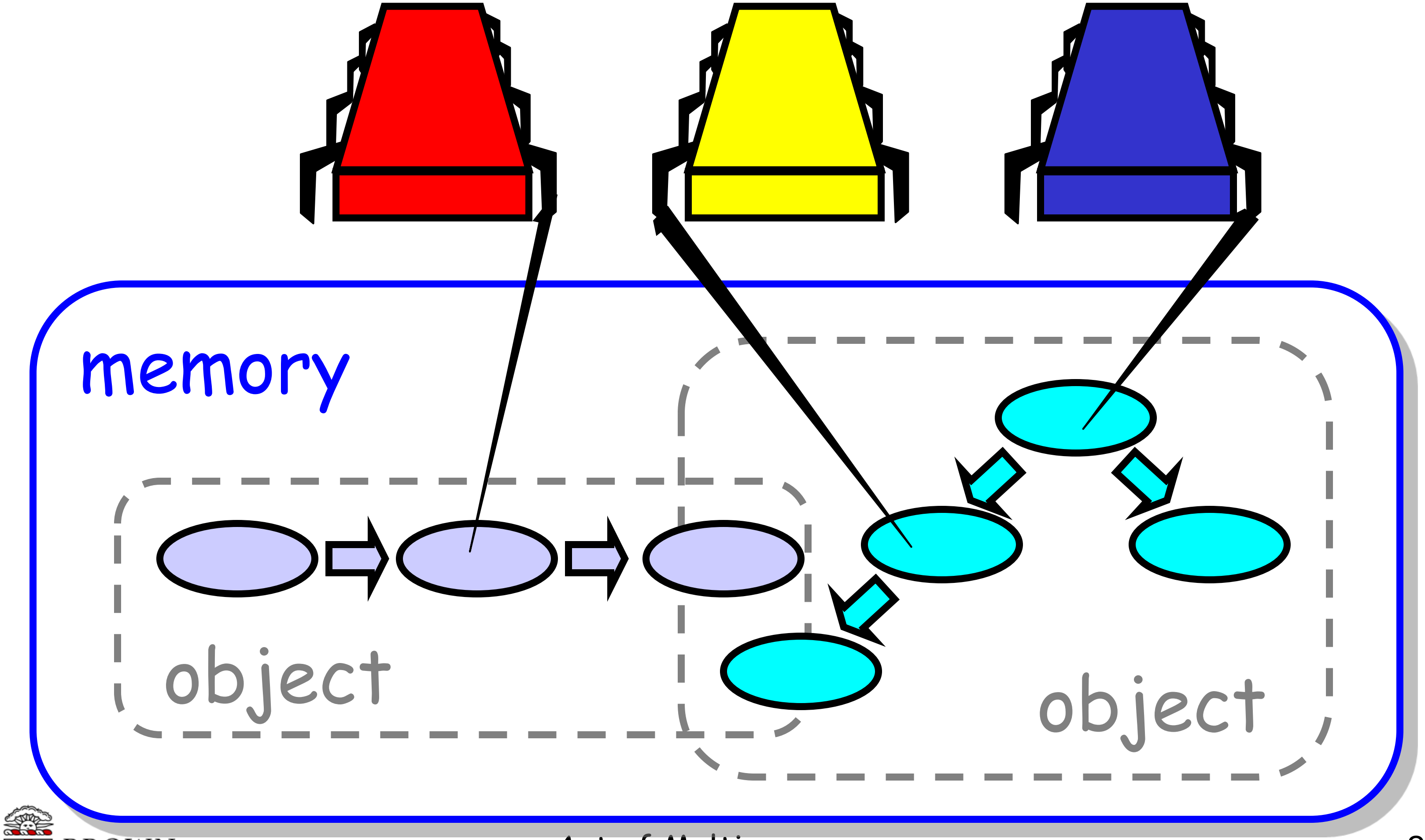
Concurrent Objects

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Concurrent Computaton



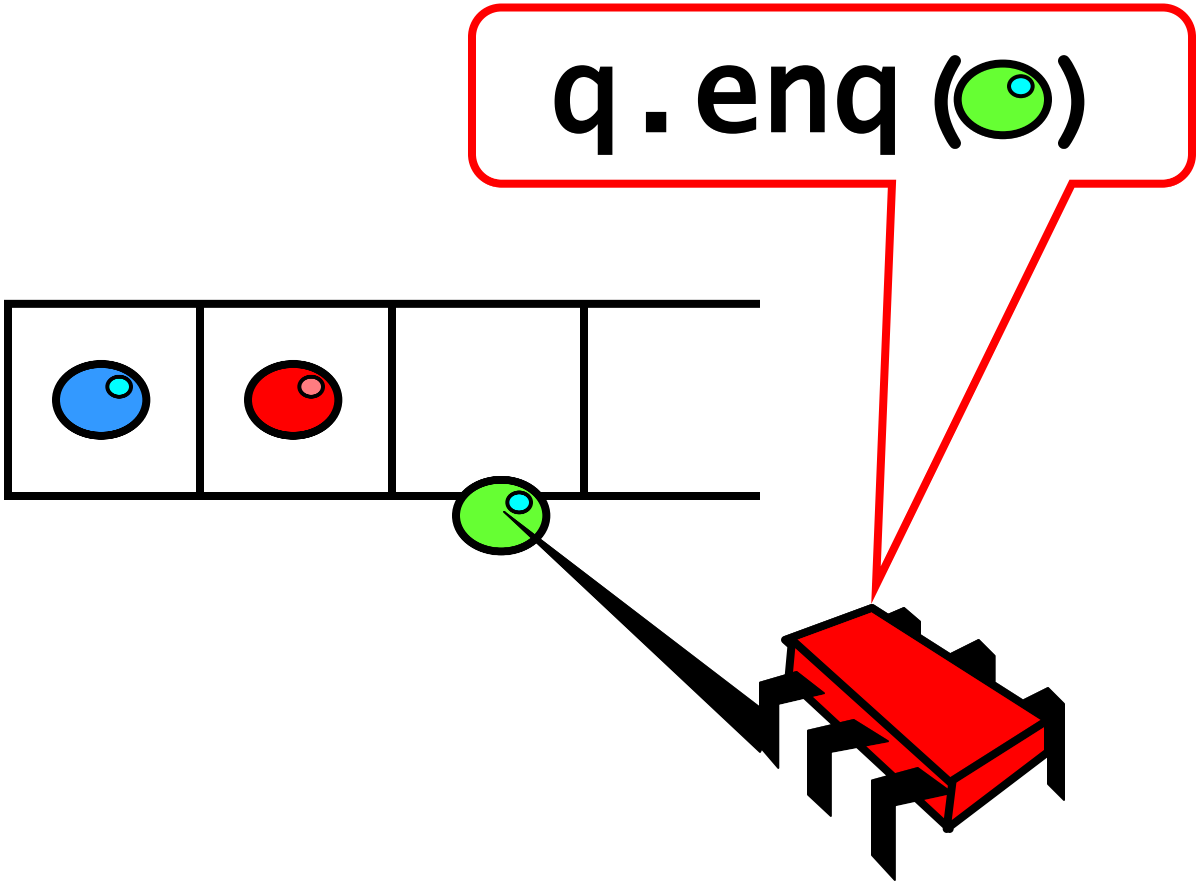
Objectivism

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

Objectivism

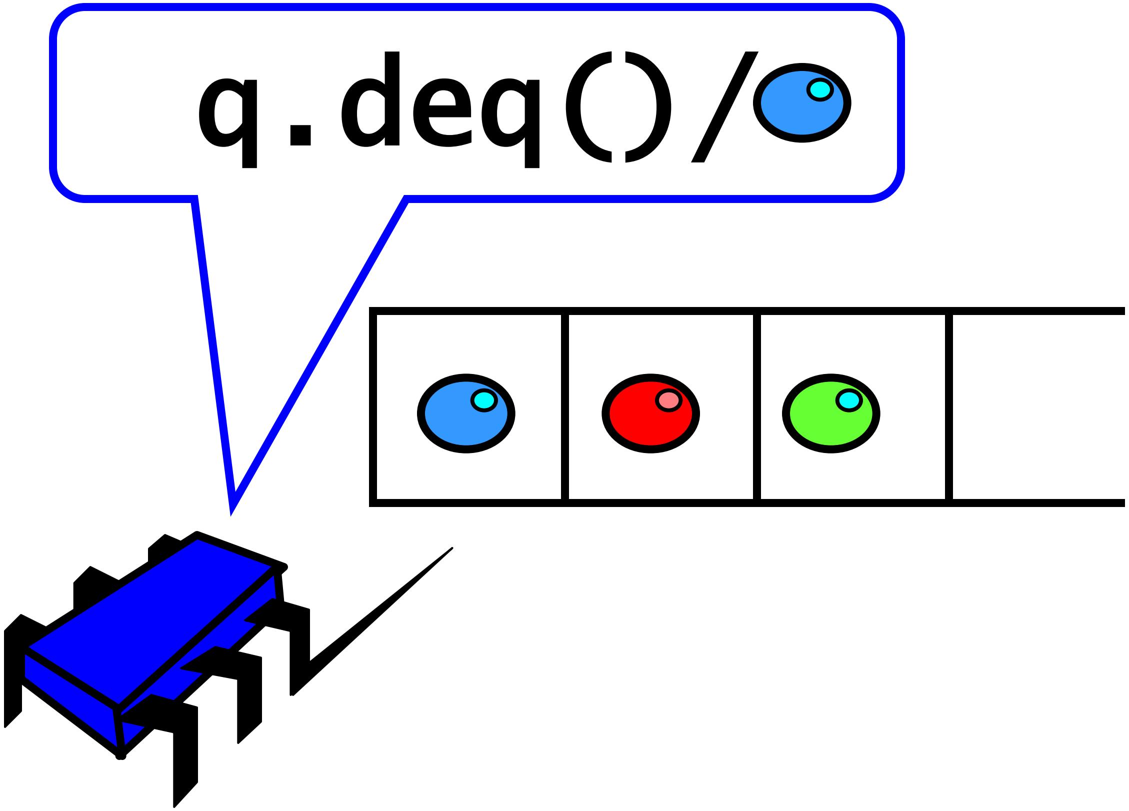
- What is a concurrent object?
  - How do we **describe** one?
  - How do we **tell if we're right**?

FIFO Queue: Enqueue Method



q.enq(●)

FIFO Queue: Dequeue Method

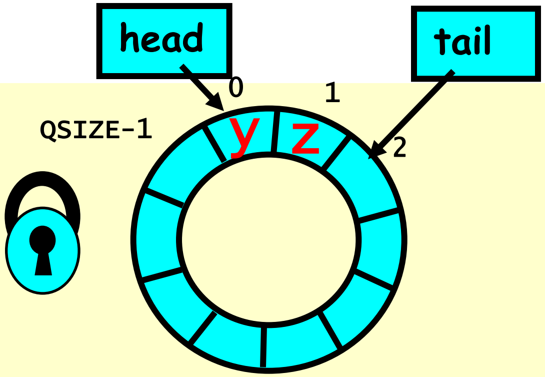


q.deq()/●

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Implementation: Deq



```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Method calls mutually exclusive

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Is queue empty?

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Release lock if need to wait

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

actual update

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Notify waiting threads that you put something in the queue

Implementation: Deq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Should be correct because modifications are mutually exclusive…

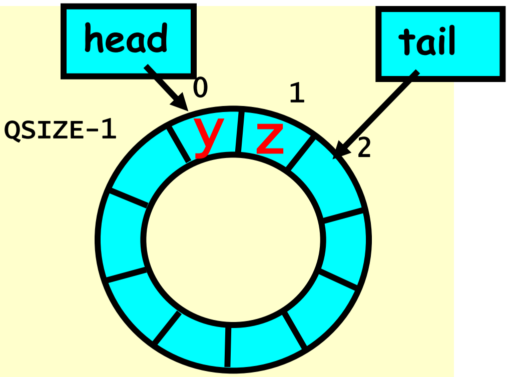
Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Lock-free 2-Thread Queue

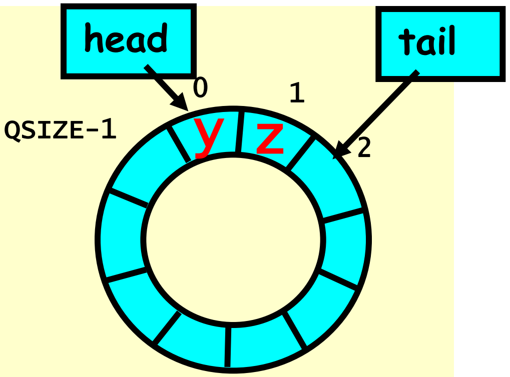
```
public class LockFreeQueue {

  int head = 0, tail = 0;
  Item[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;
}}
```

Lock-free 2-Thread Queue



```
public class LockFreeQueue {

  int head = 0, tail = 0;
  Item[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;
}}
```

Lock-free 2-Thread Queue



```
public class LockFreeQueue {

  int head = 0, tail = 0;
  Item[QSIZE] items;

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

Queue is up ut a lock!

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

Defining correct concurrent queue implementations

- 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 for Dequeue

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

Pre and PostConditions for Dequeue

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

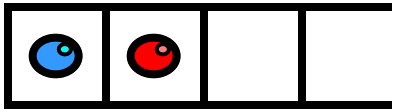
Why Sequential Specifications Totally Rock

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

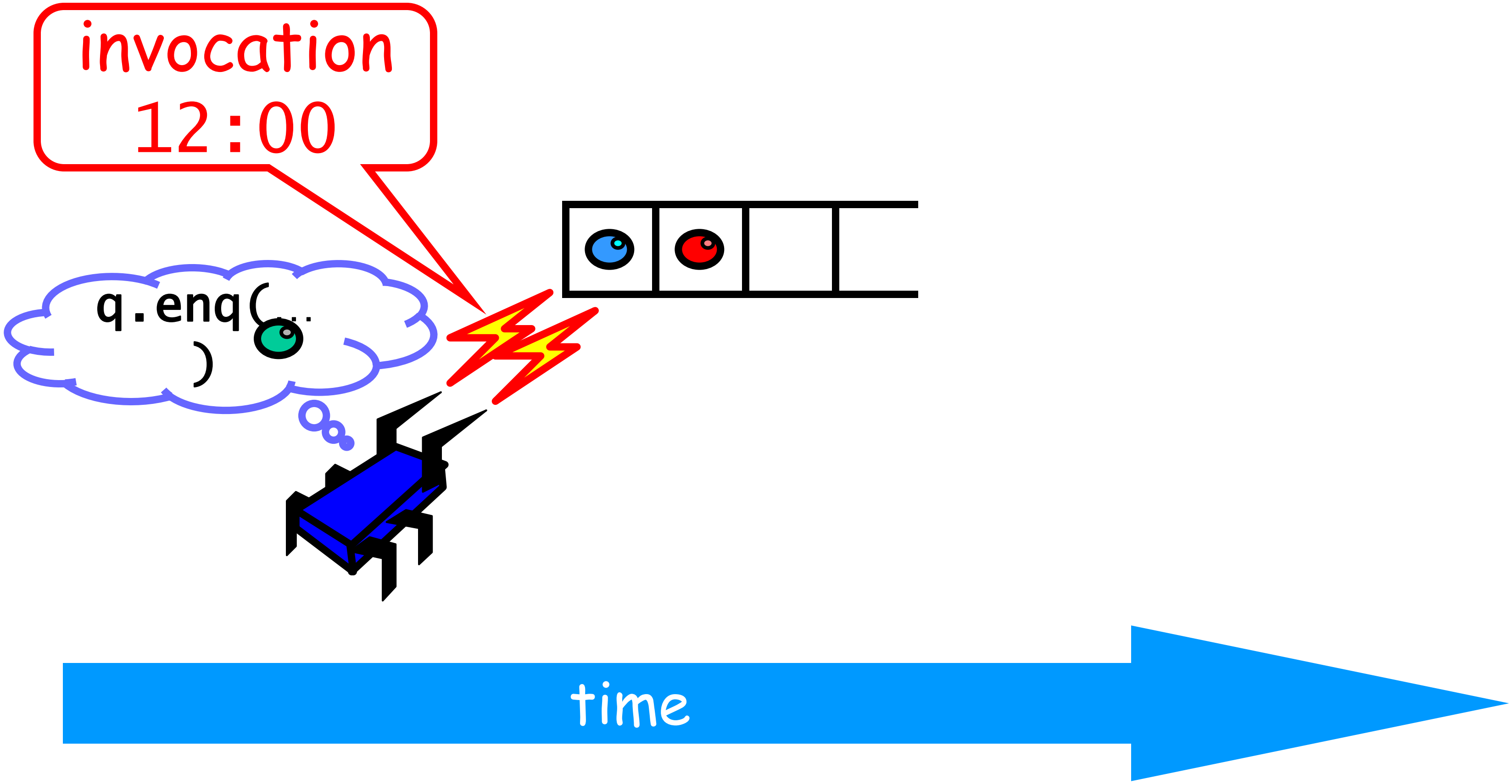
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

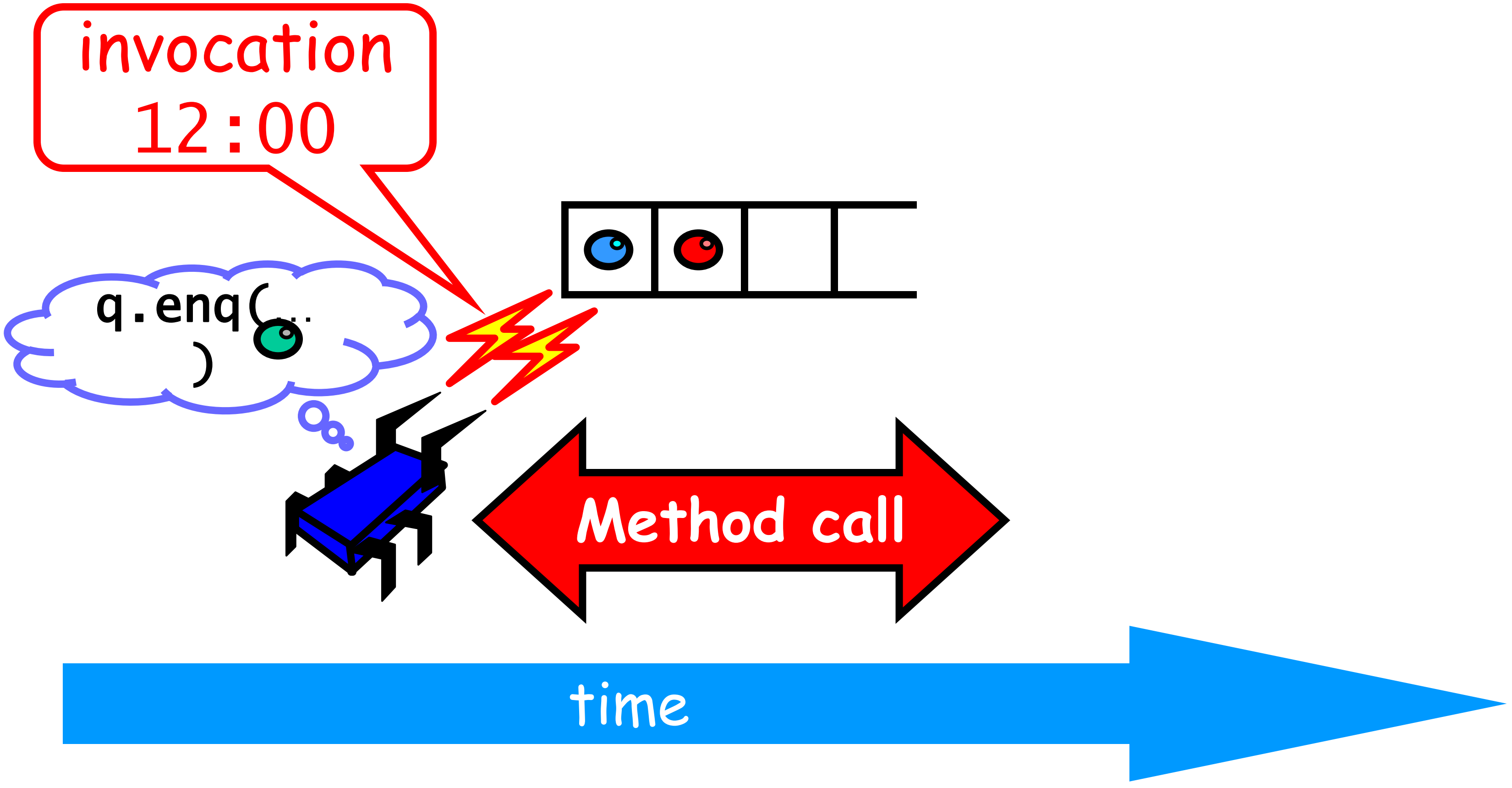
Methods Take Time



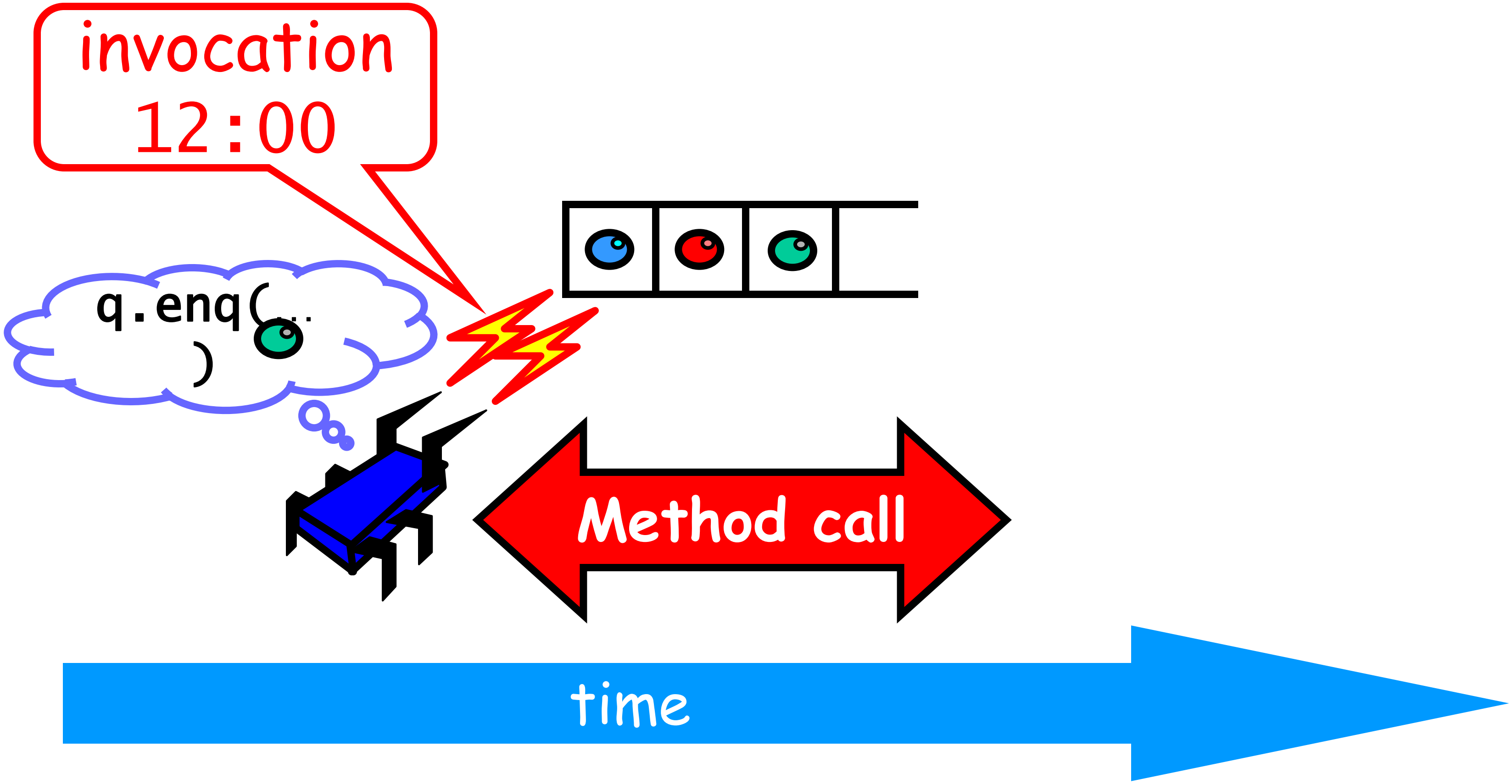
Methods Take Time



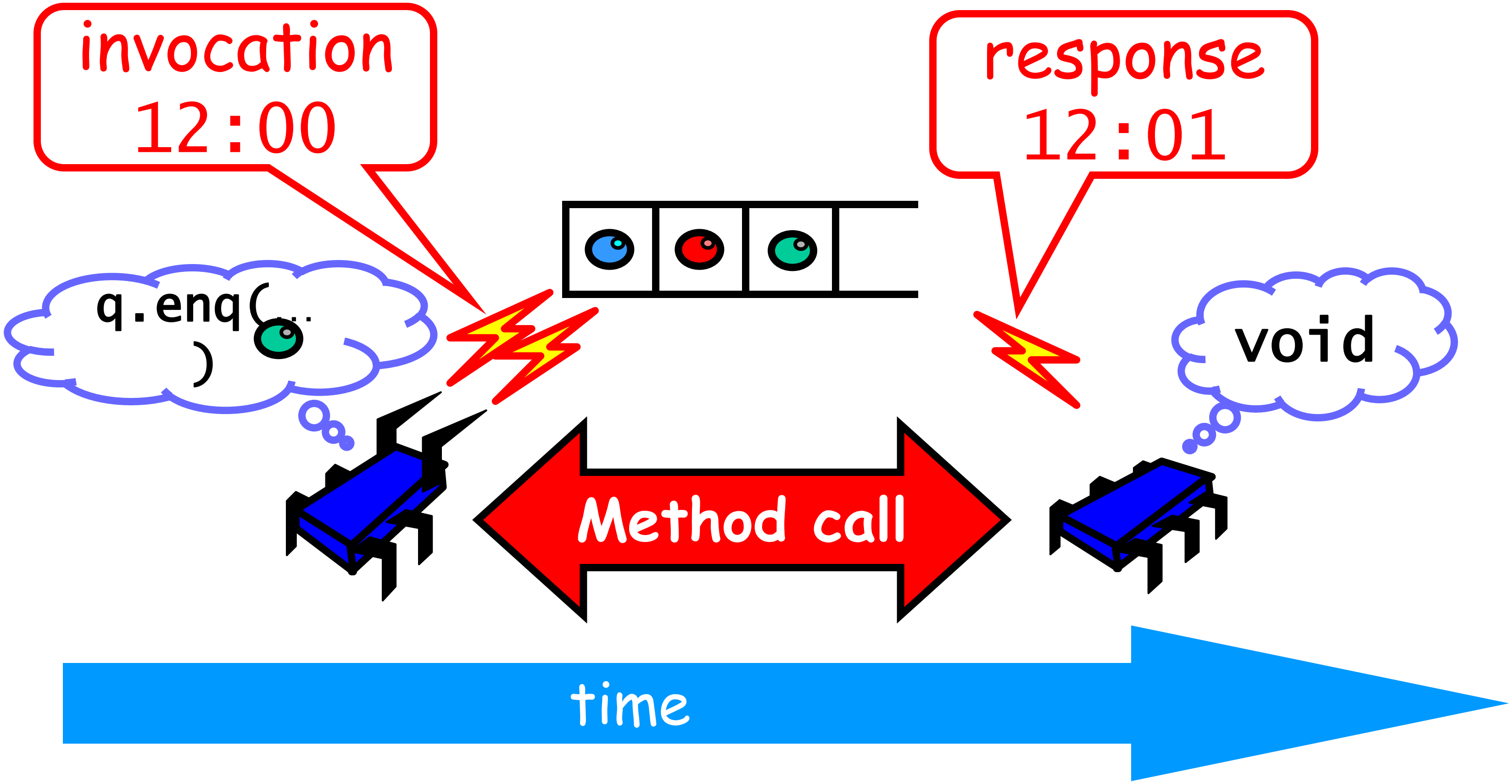
Methods Take Time



Methods Take Time



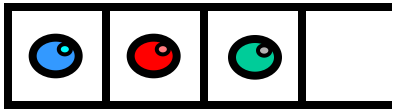
Methods Take Time



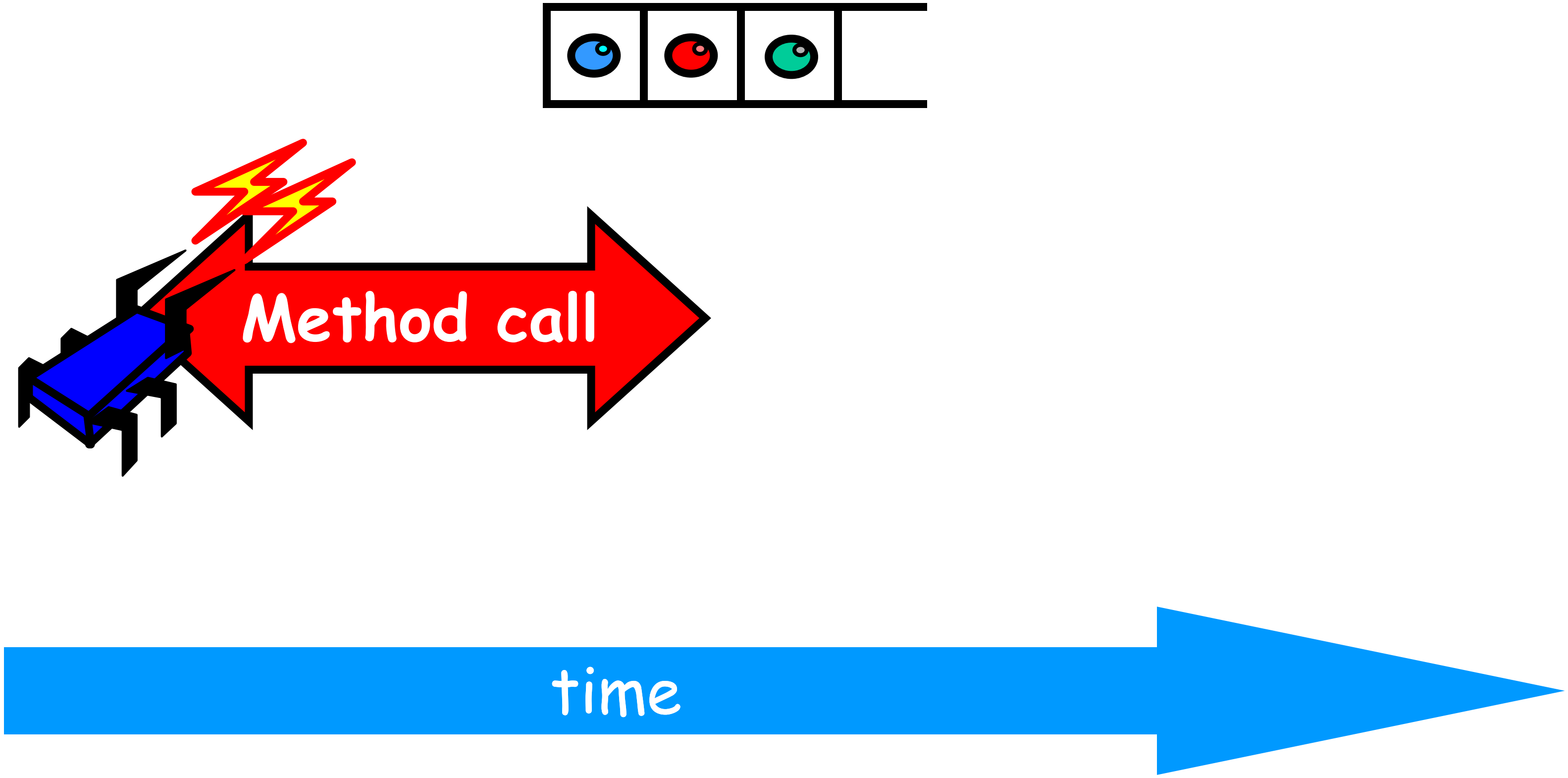
Sequential vs Concurrent

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

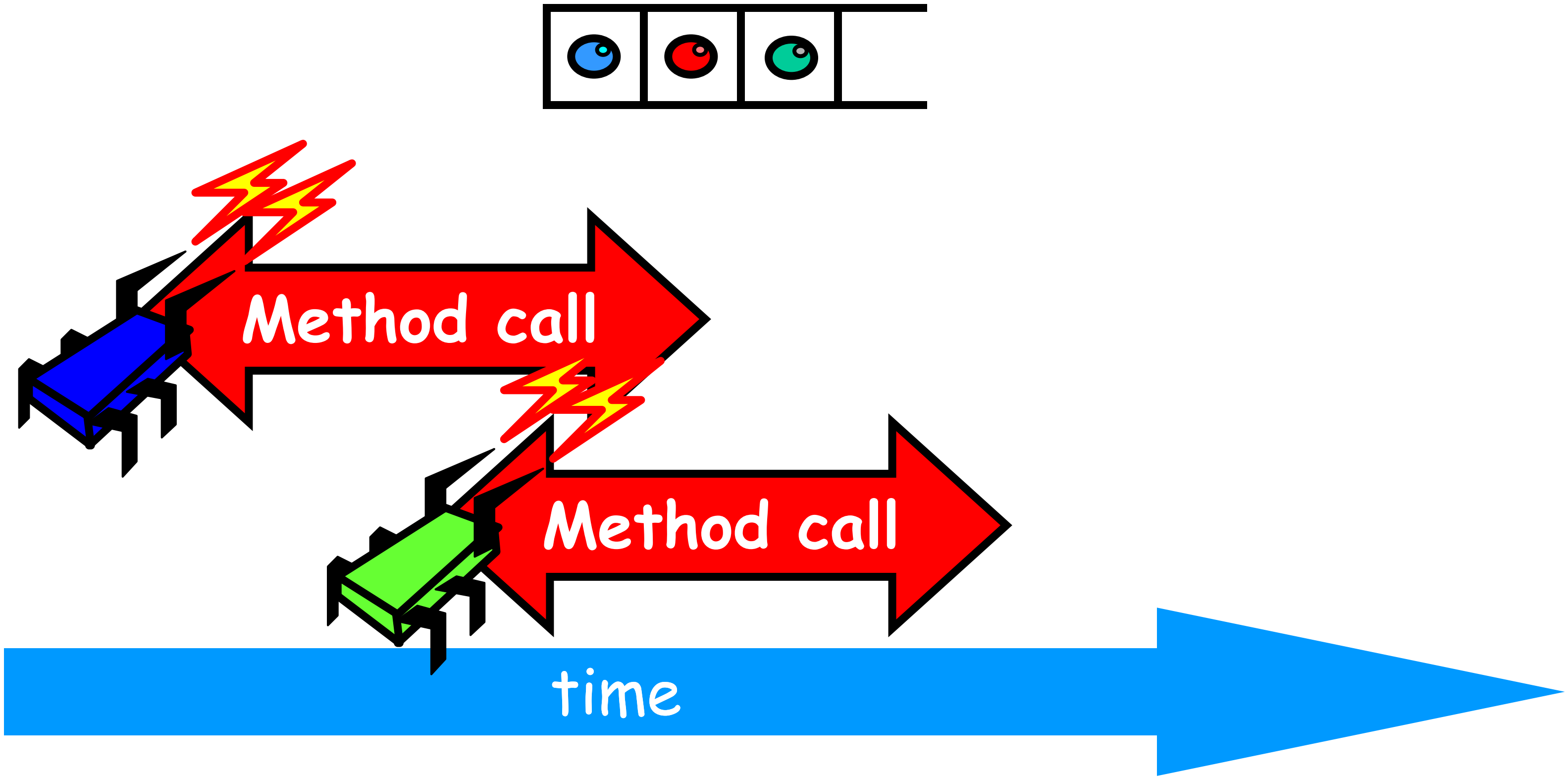
Concurrent Methods Take Overlapping Time



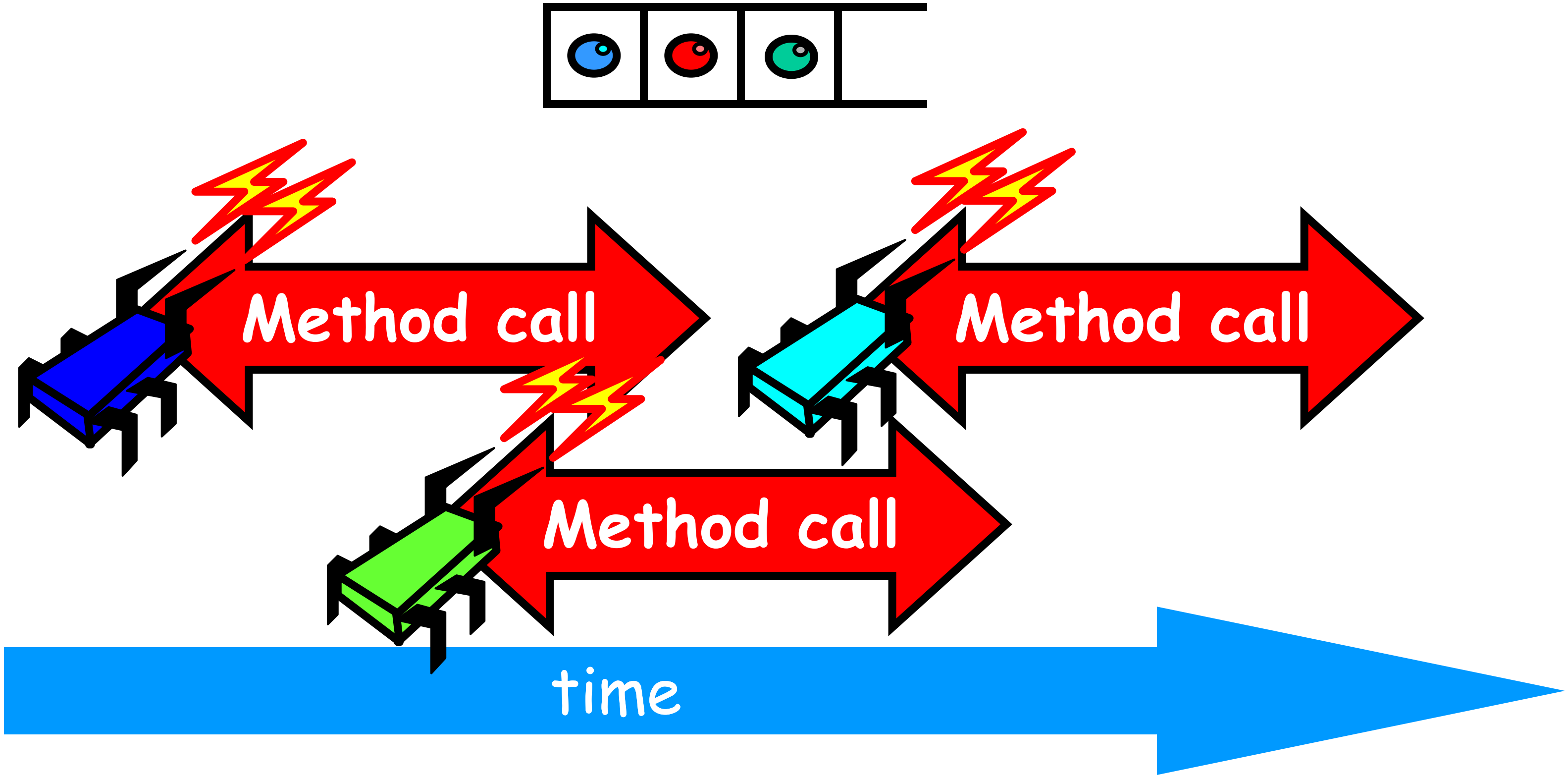
Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only between method calls
- Concurrent
  - Because method calls overlap, object might ***never*** be between method calls

Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent
  - Must characterize ***all*** possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? …

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - 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<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

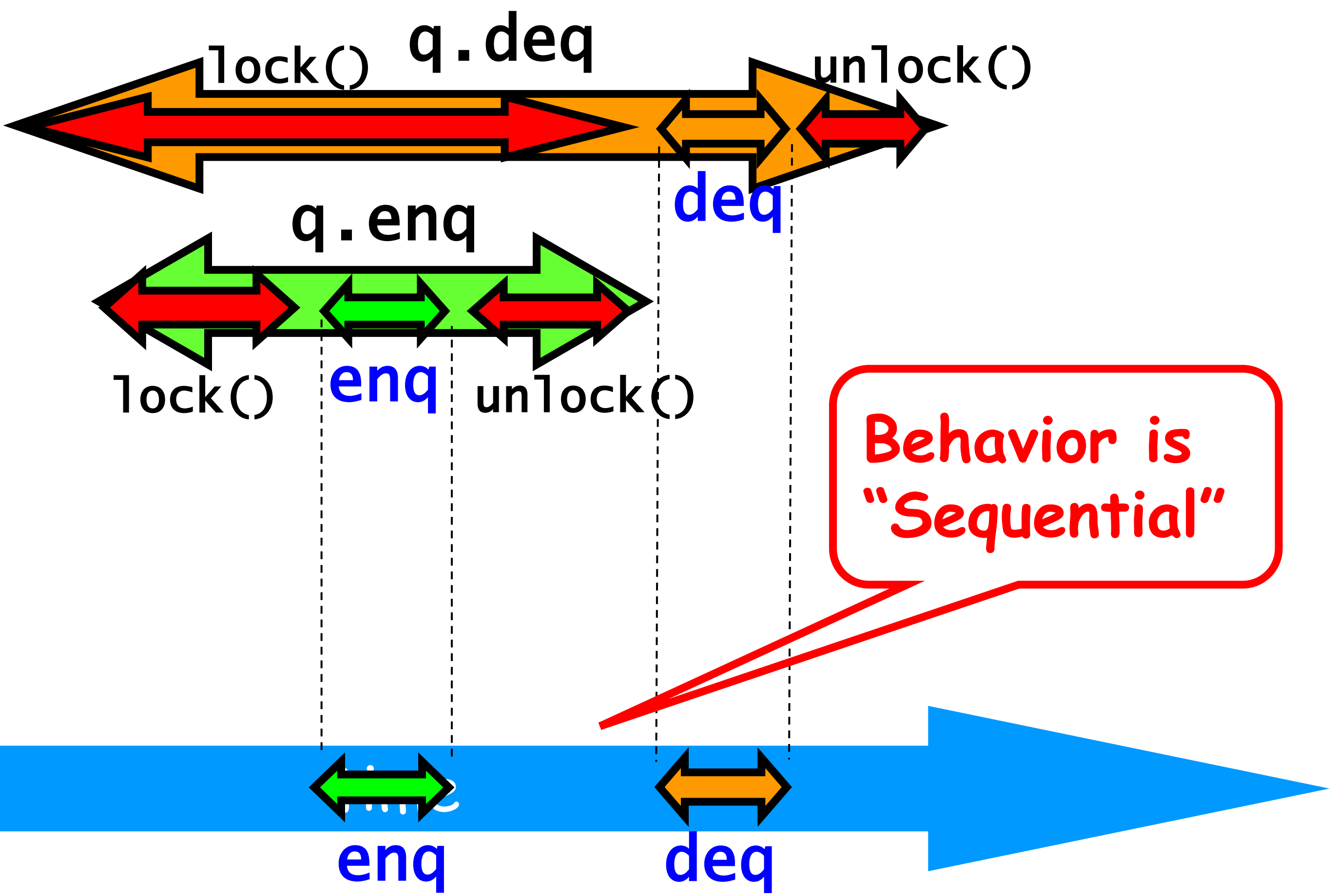
Intuitively…

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Queue is updated while holding lock (mutually exclusive)

Intuitively

Lets capture the idea of describing the concurrent via the sequential



Behavior is "Sequential"

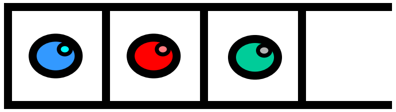
Linearizability

- 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™**

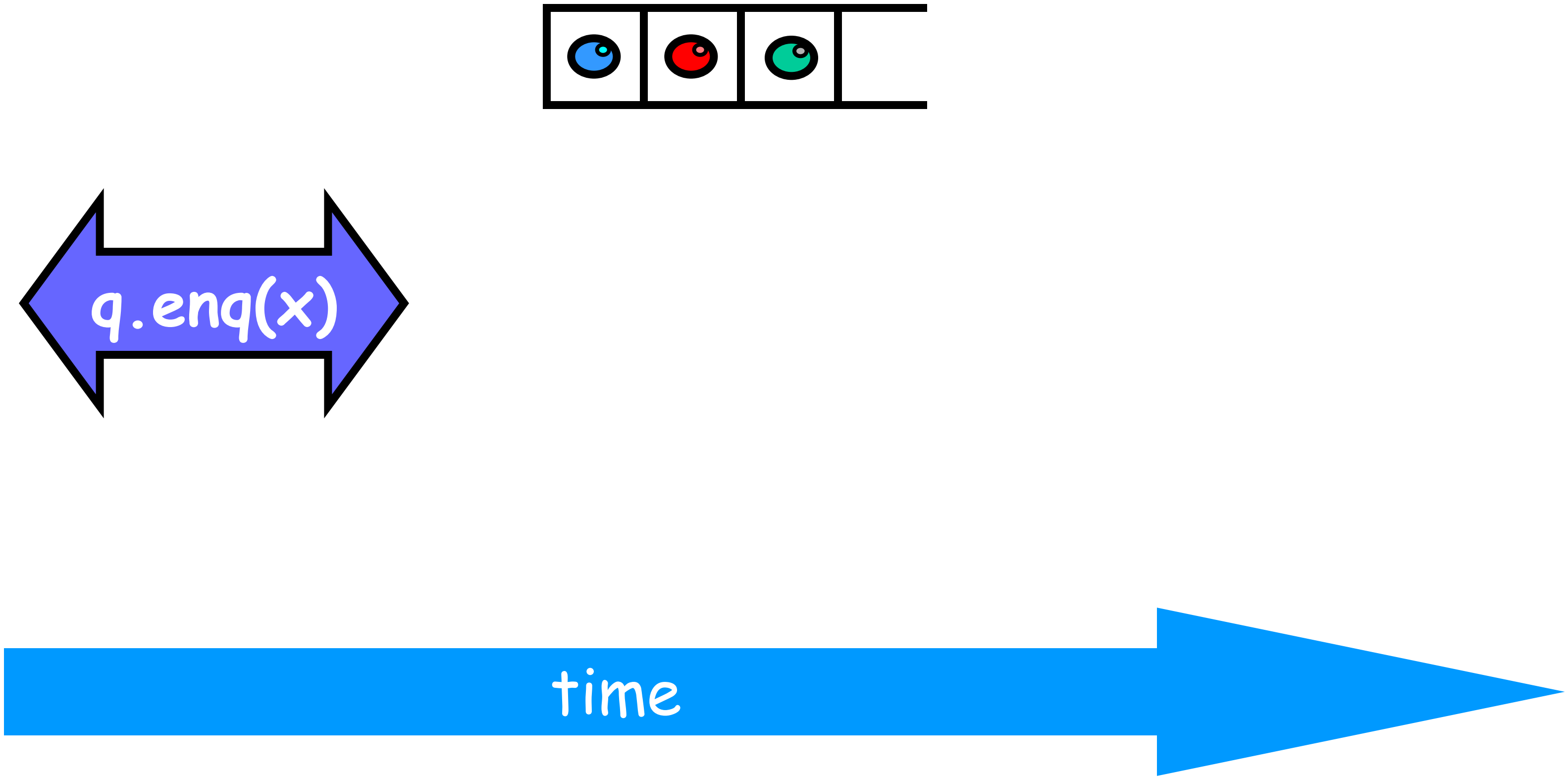
Is it really about the object?

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Sounds like a property of an execution…
- A linearizable object: one all of whose possible executions are linearizable

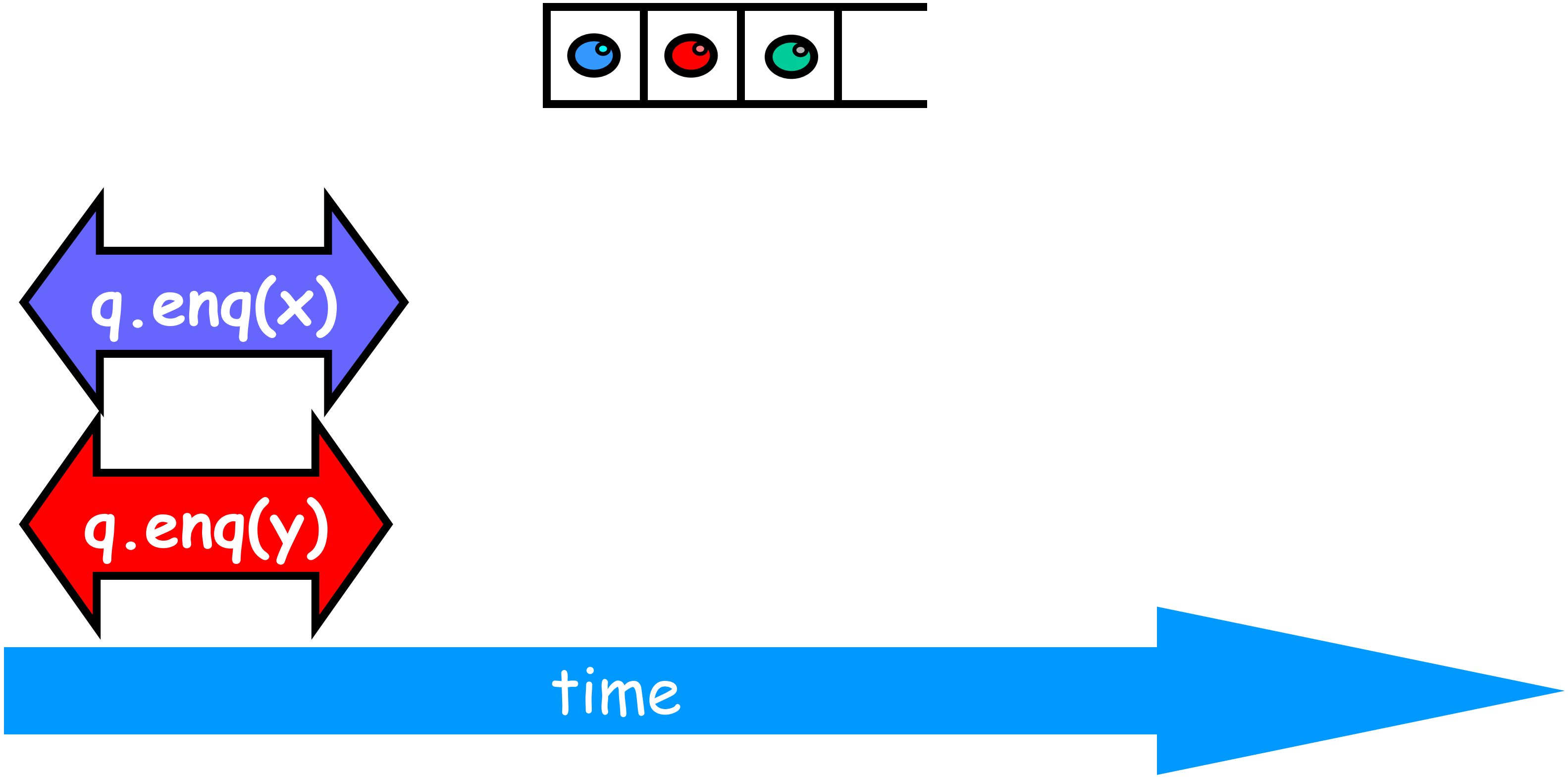
Example



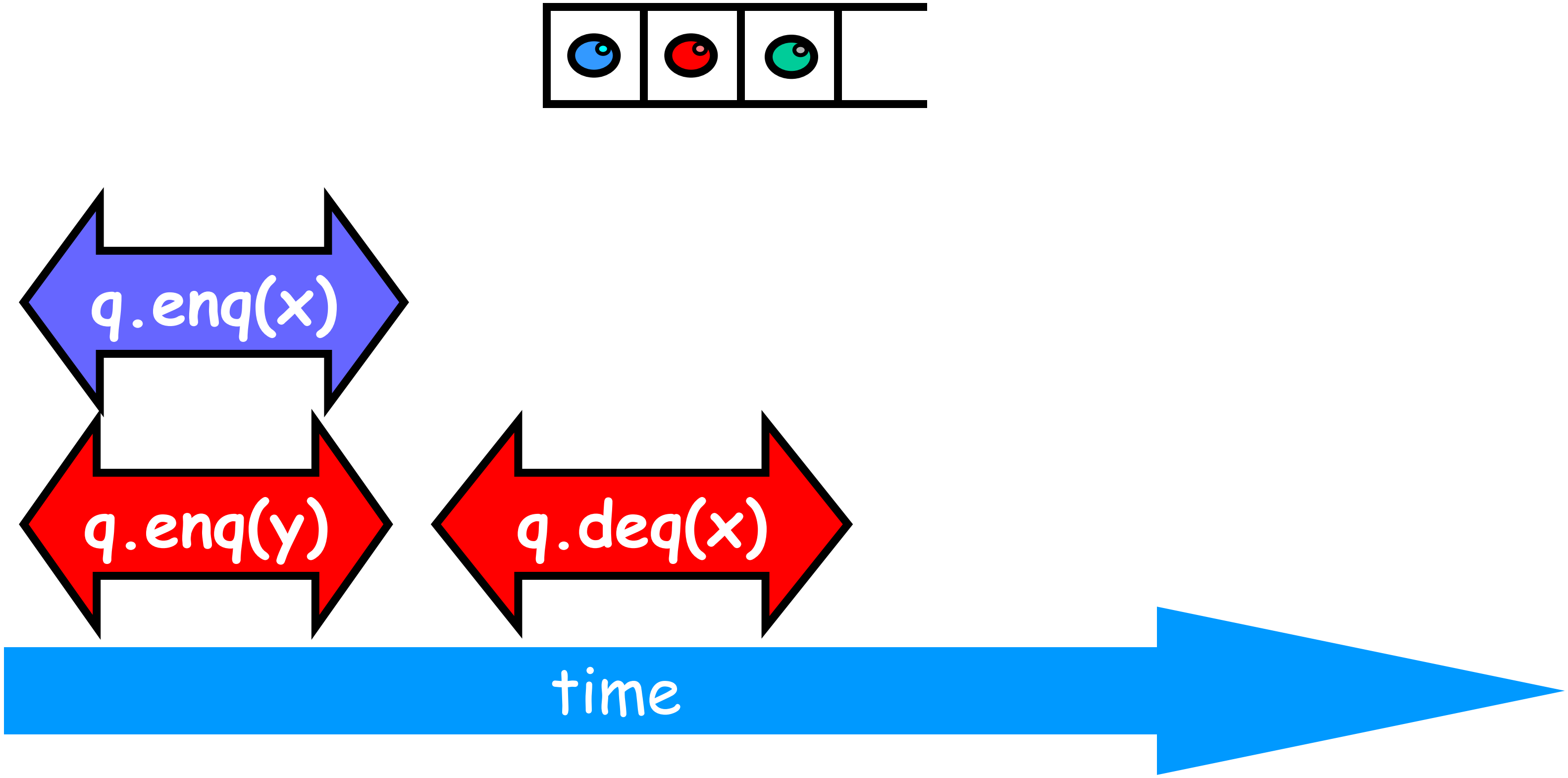
Example



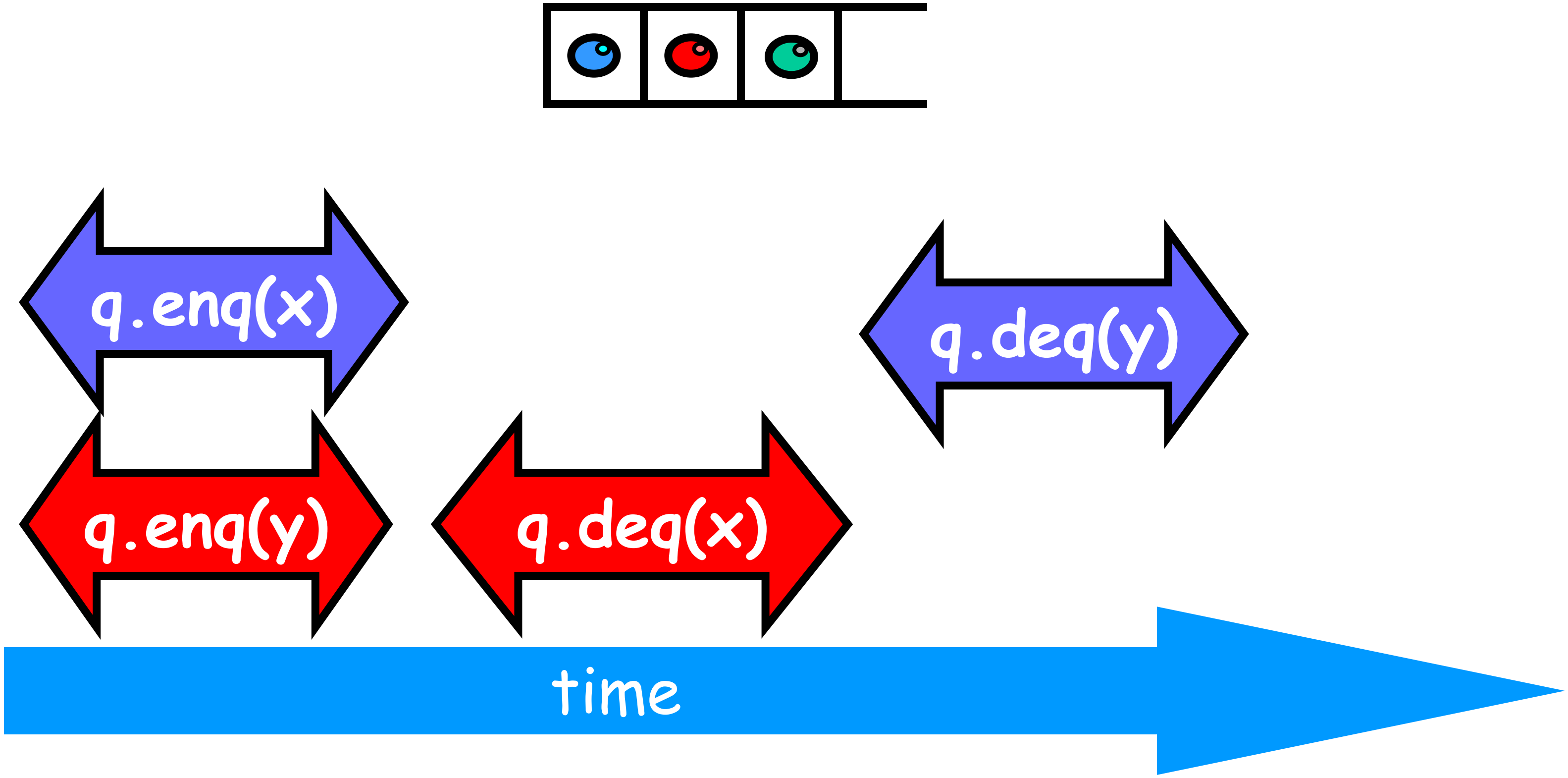
Example



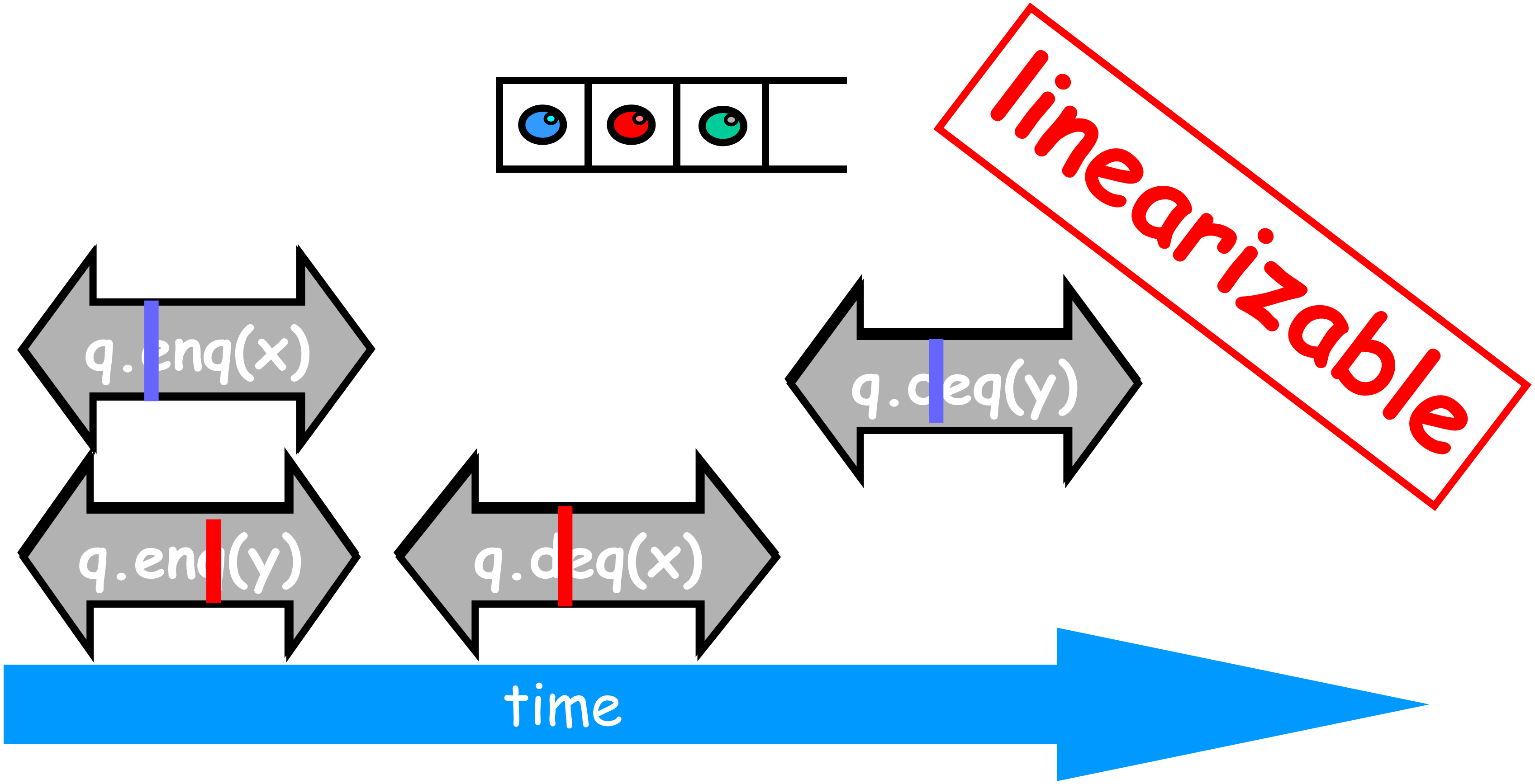
Example



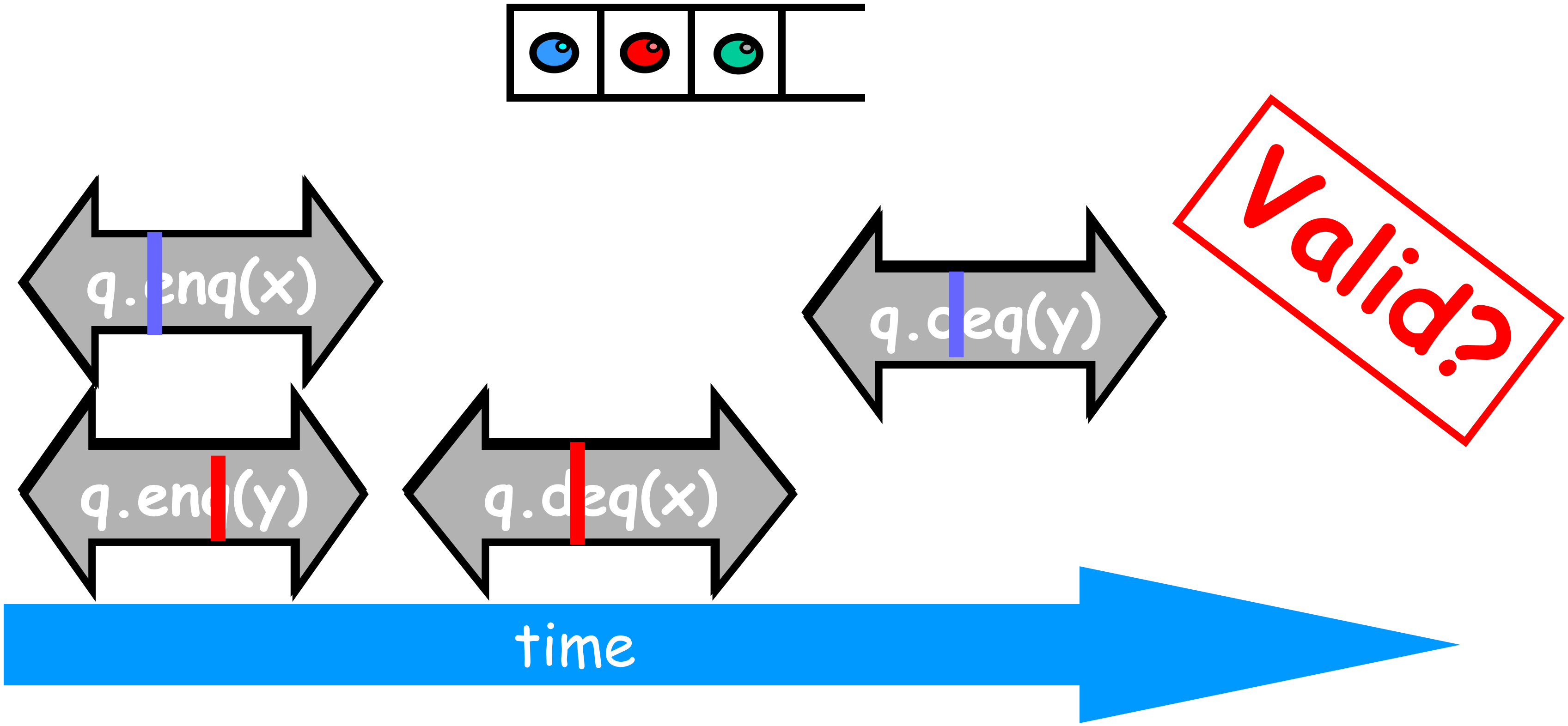
Example



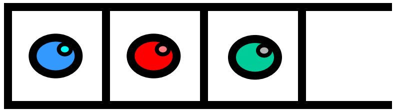
Example



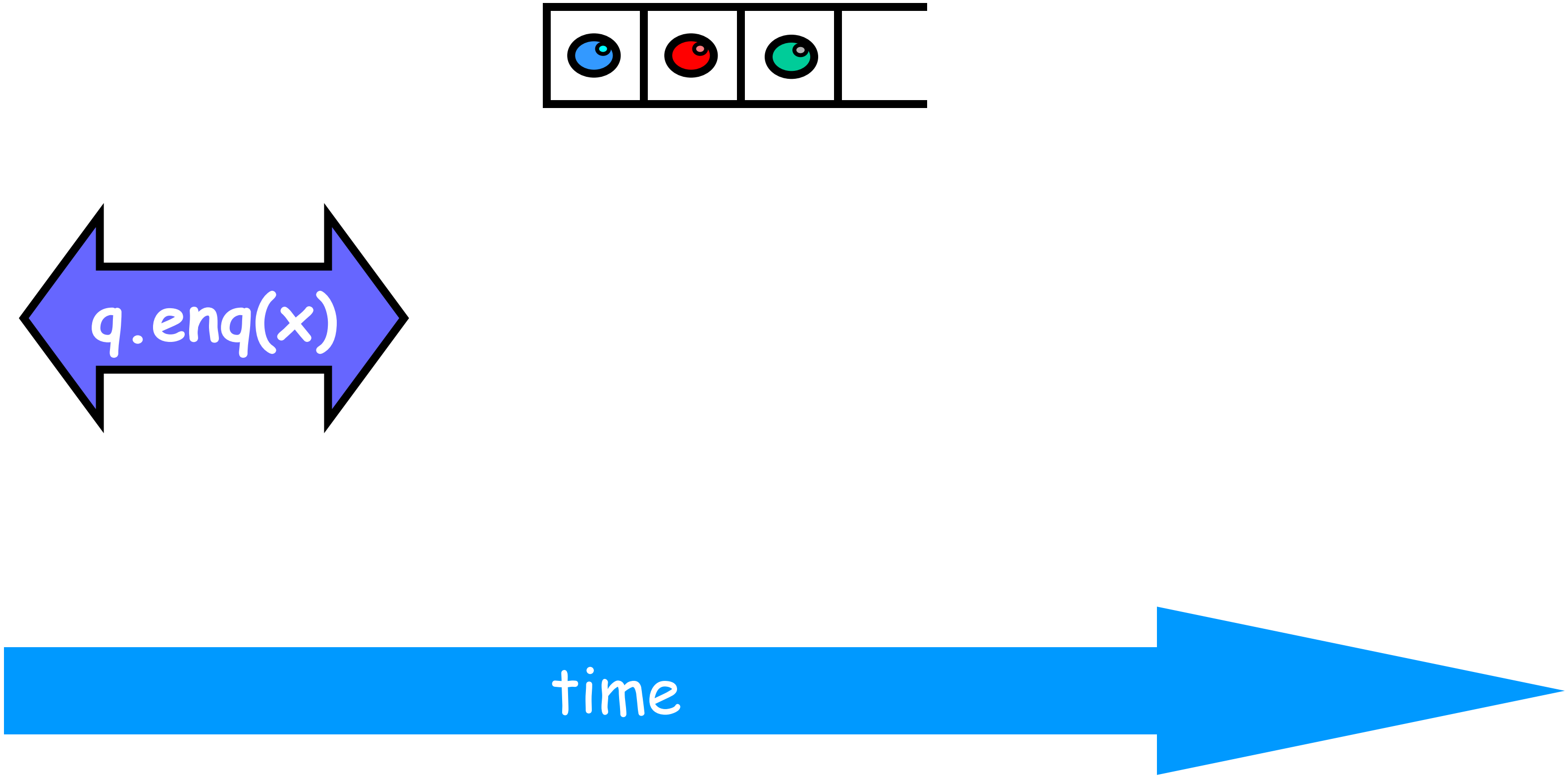
Example



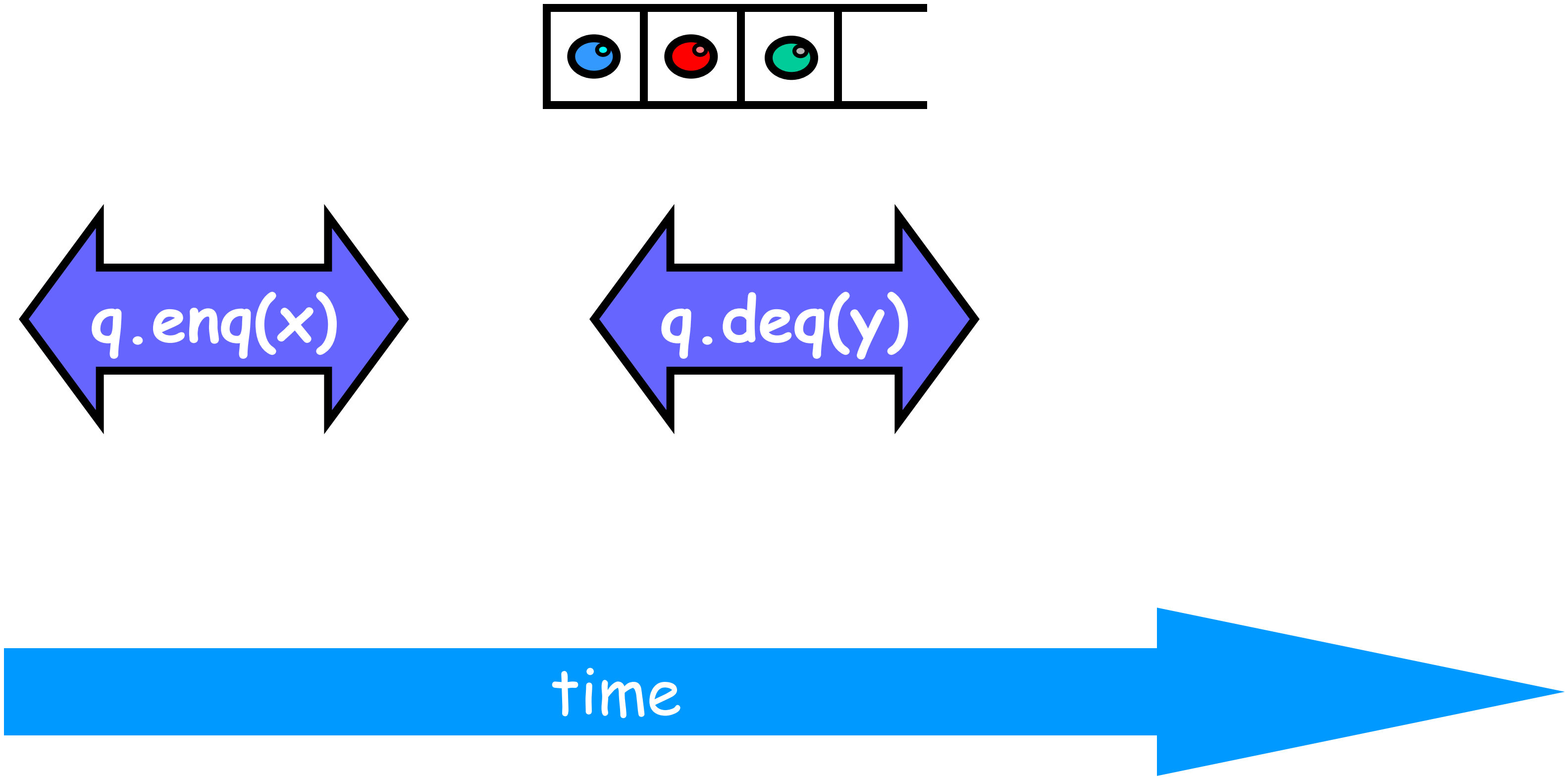
Example



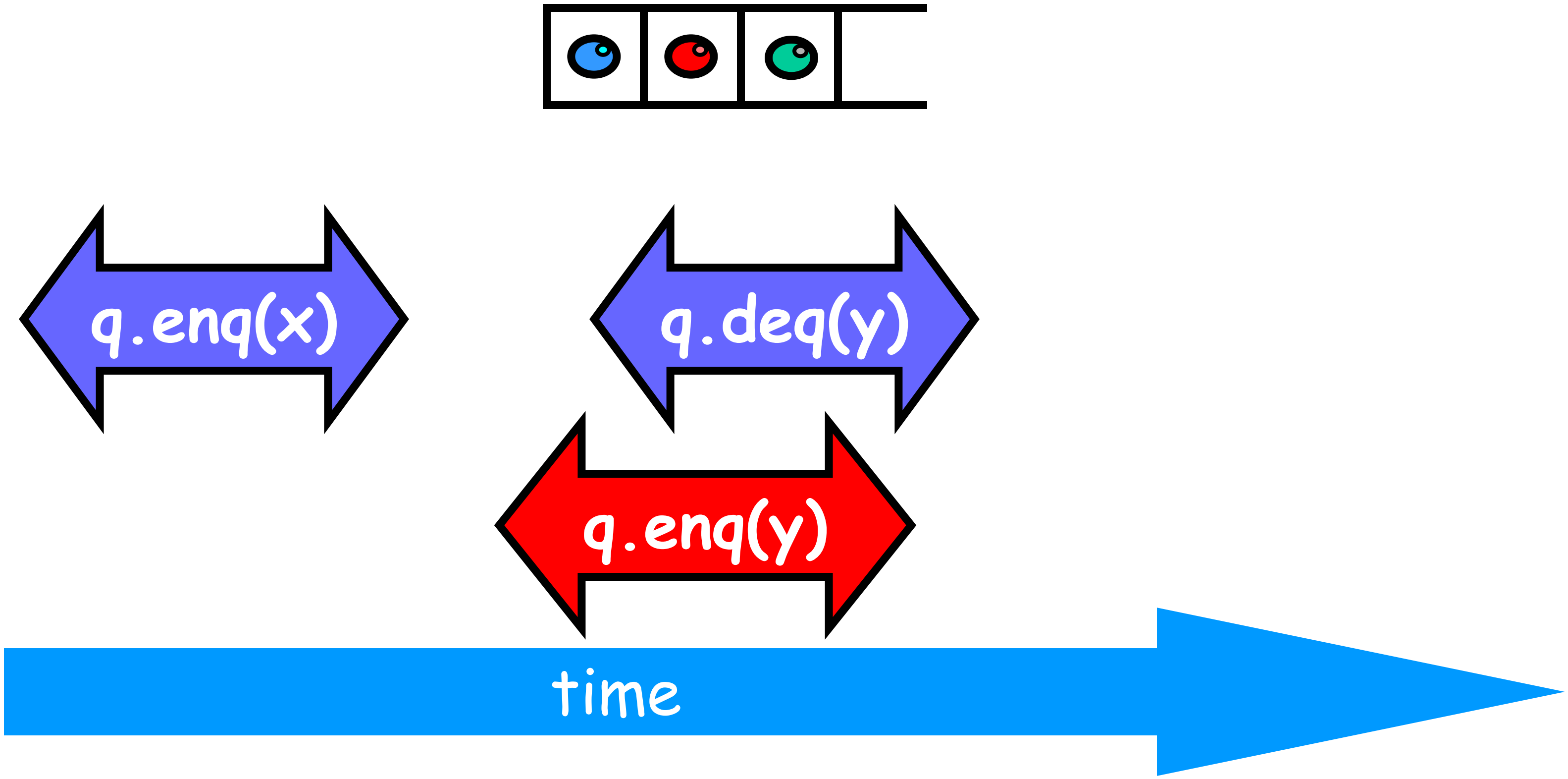
Example



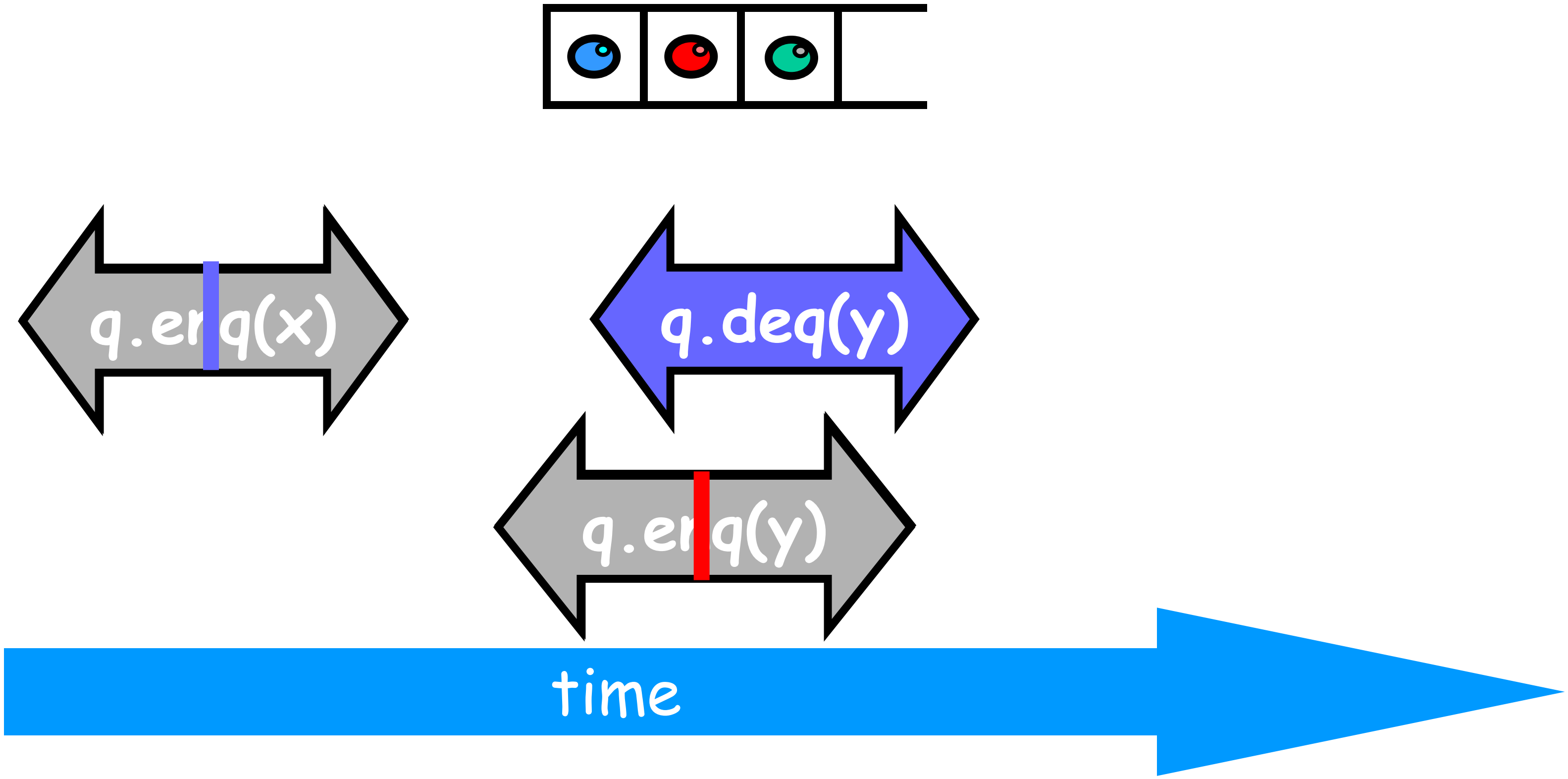
Example



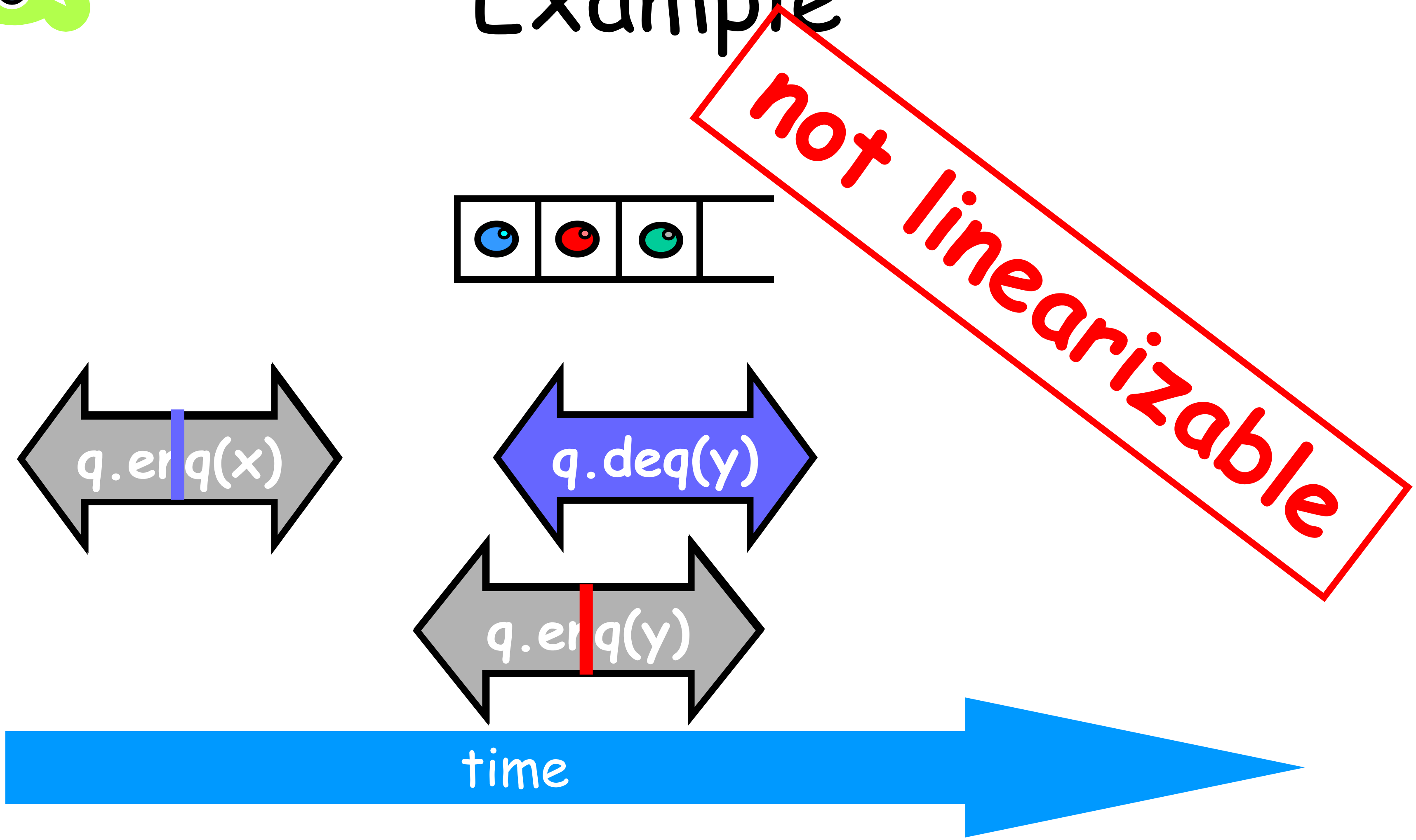
Example



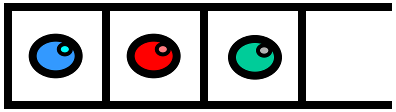
Example



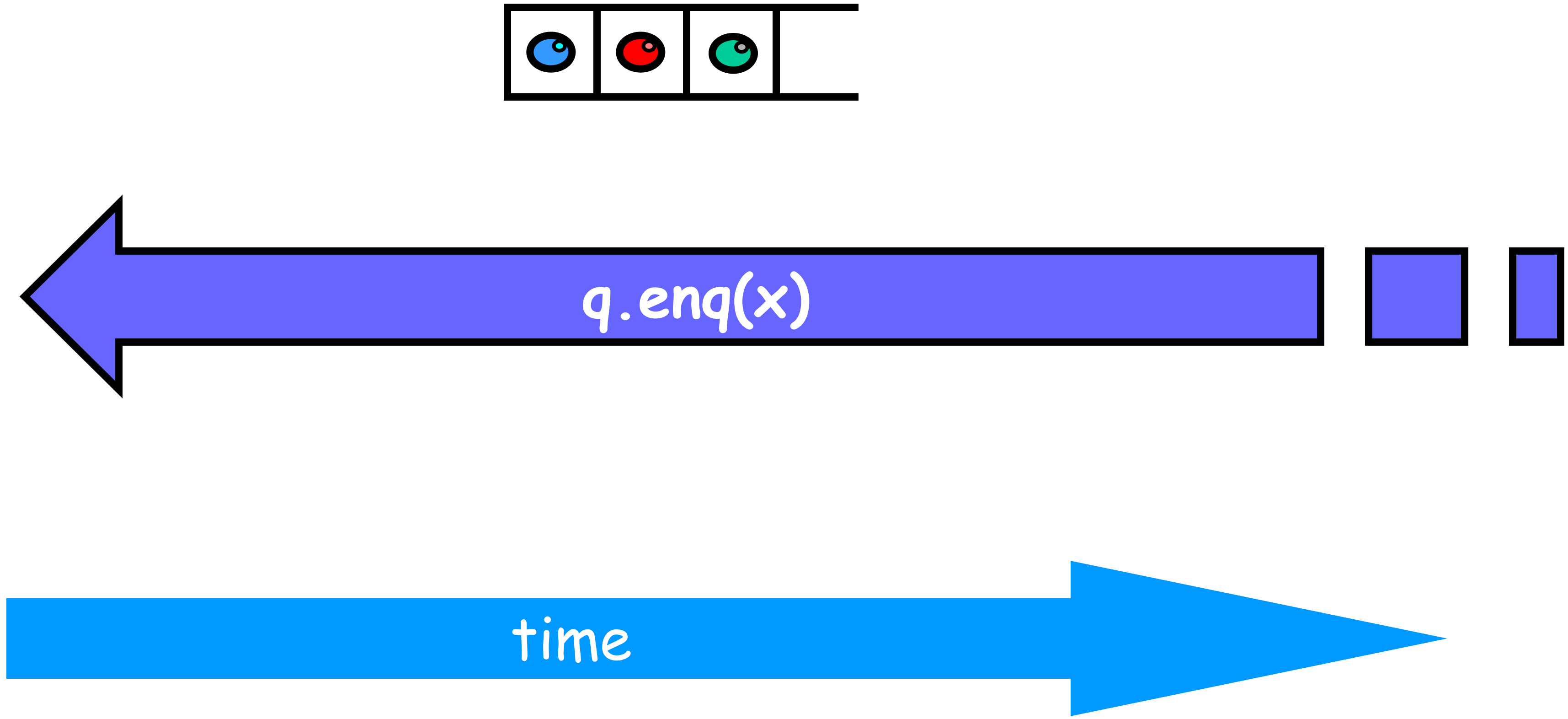
Example



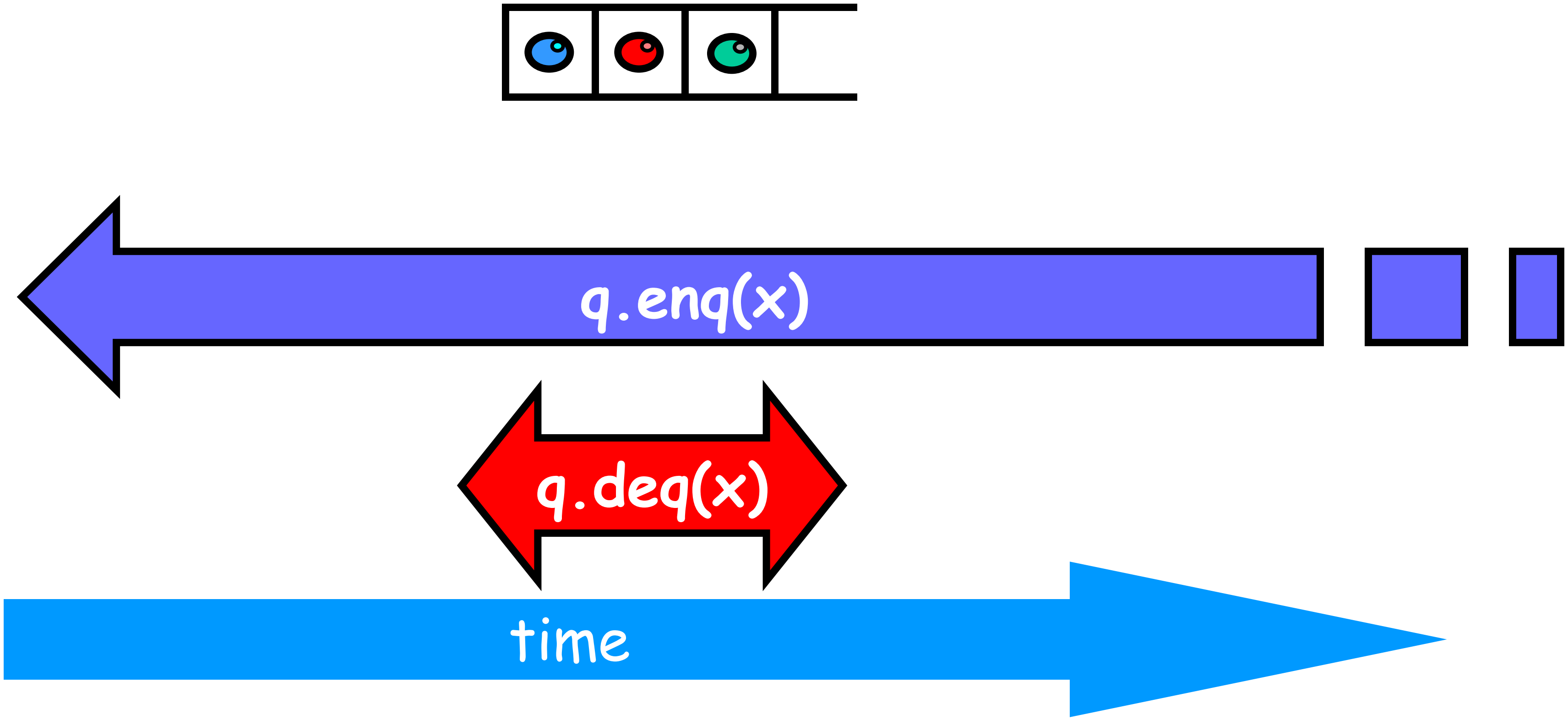
Example



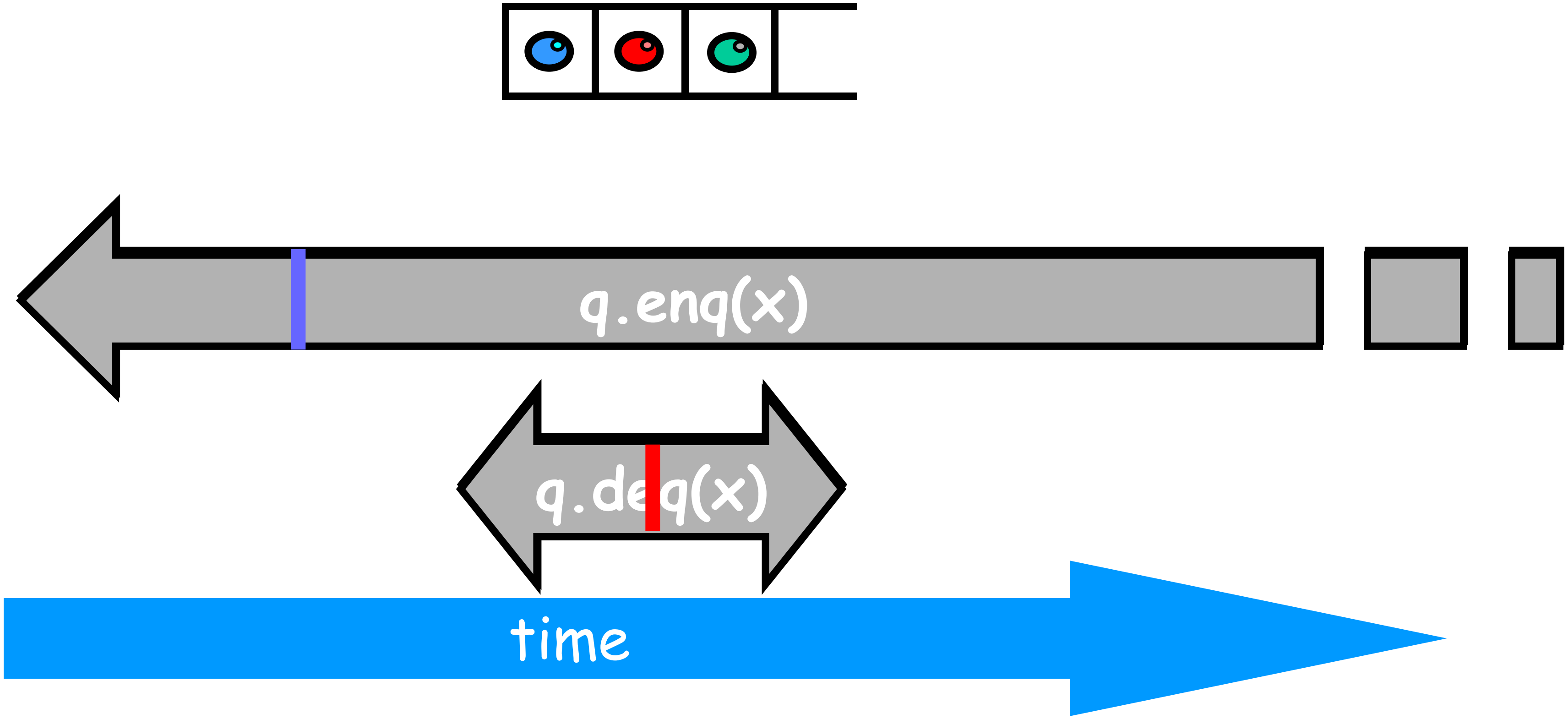
Example



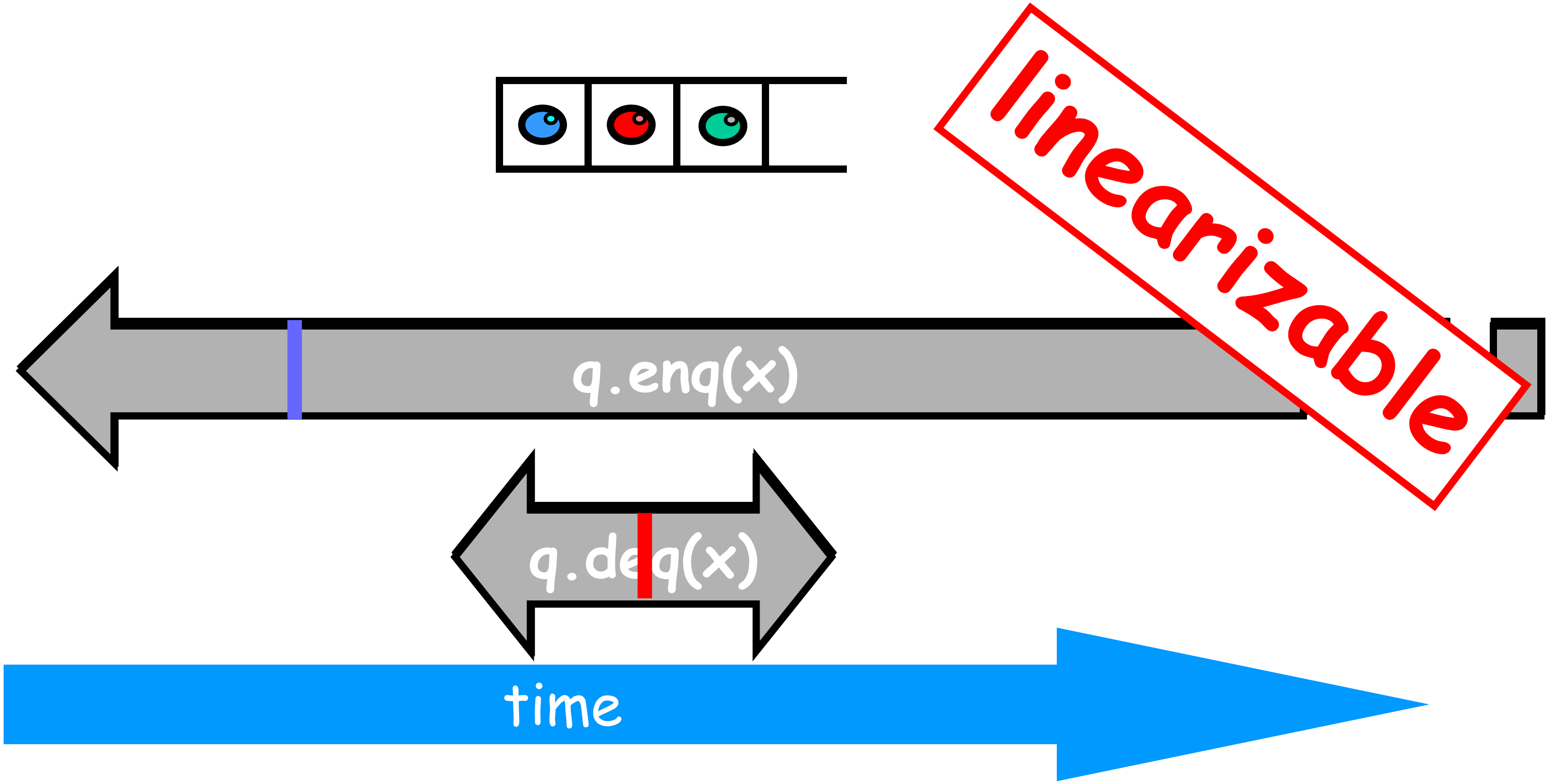
Example



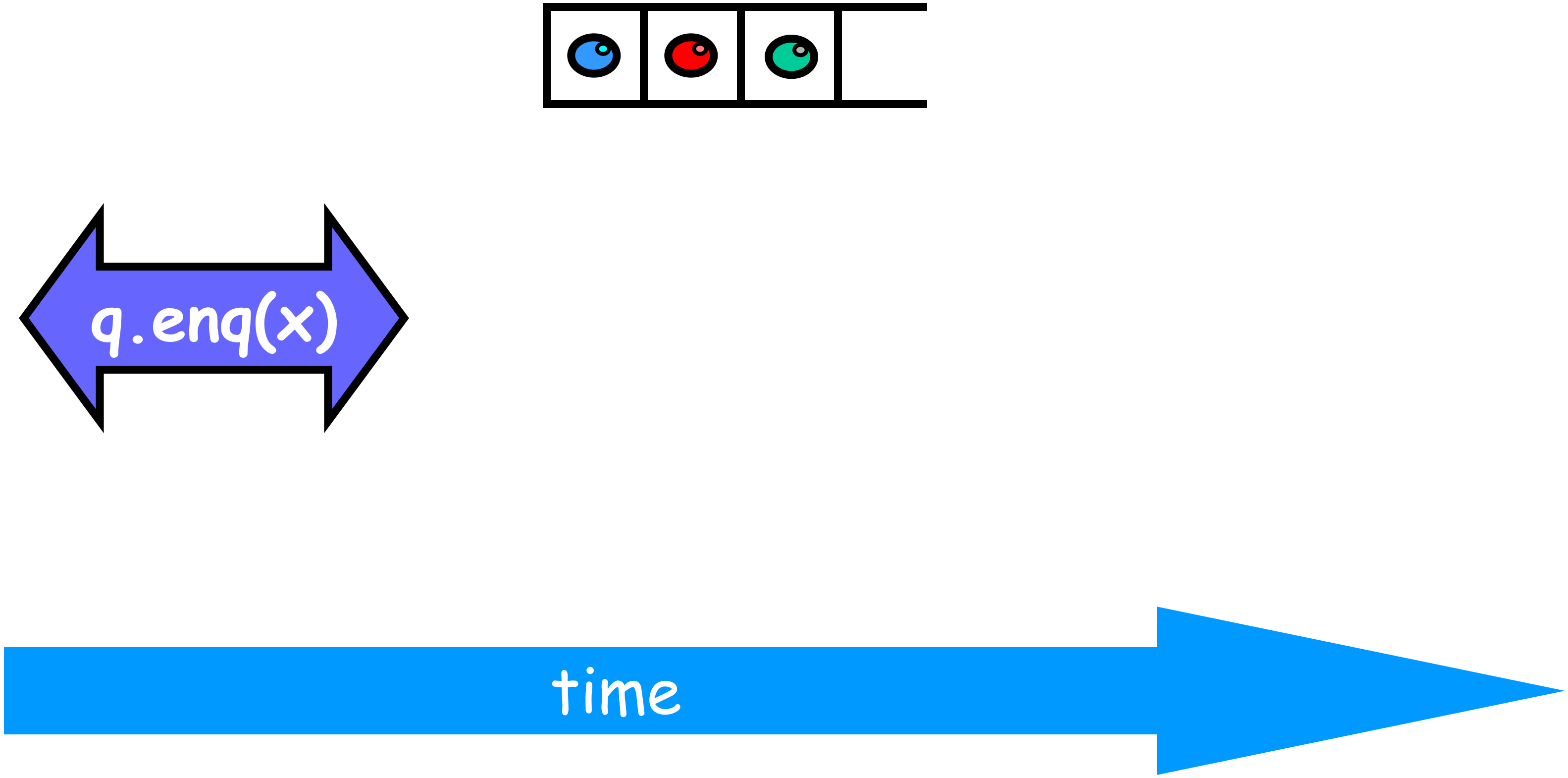
Example



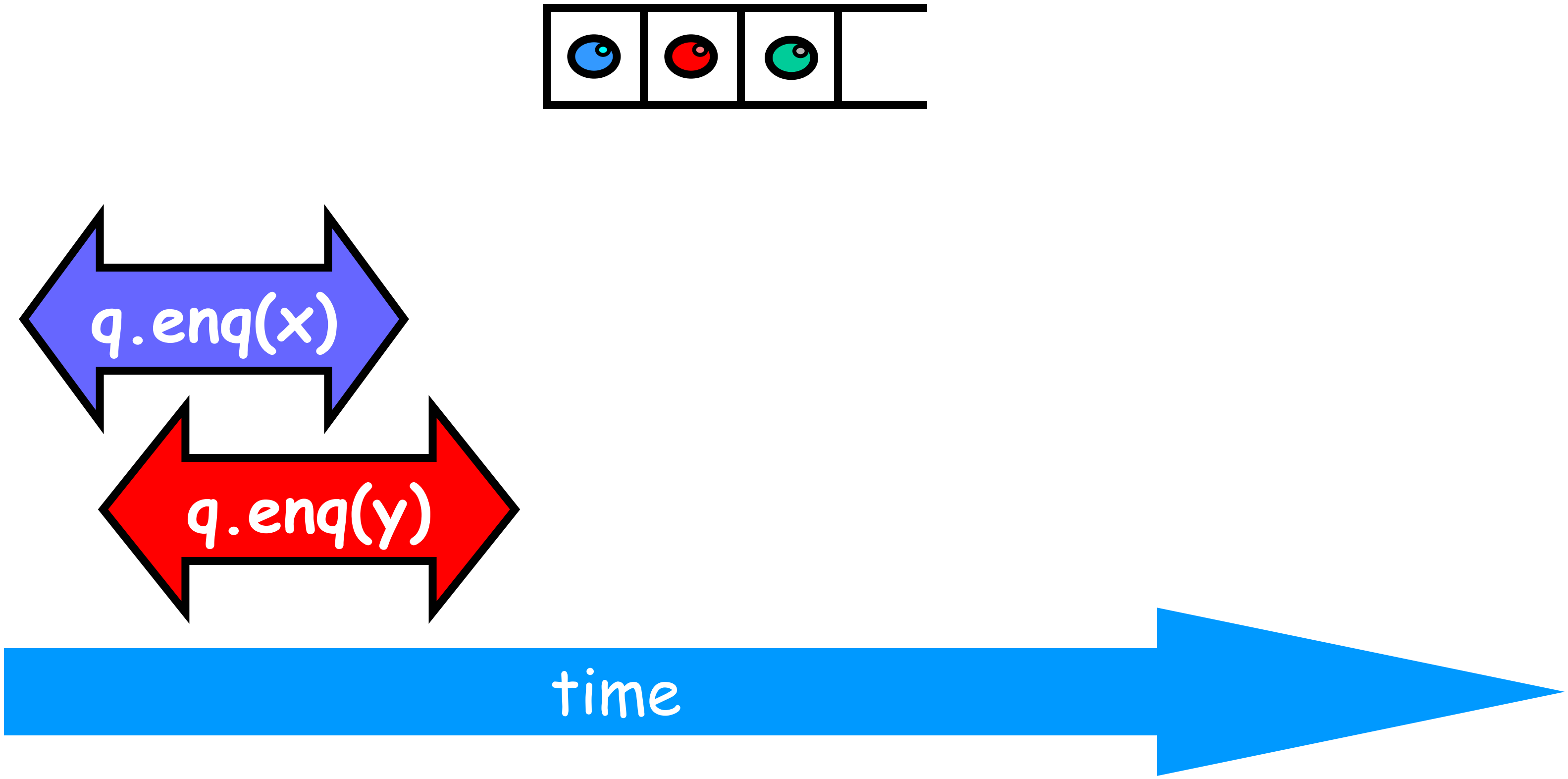
Example



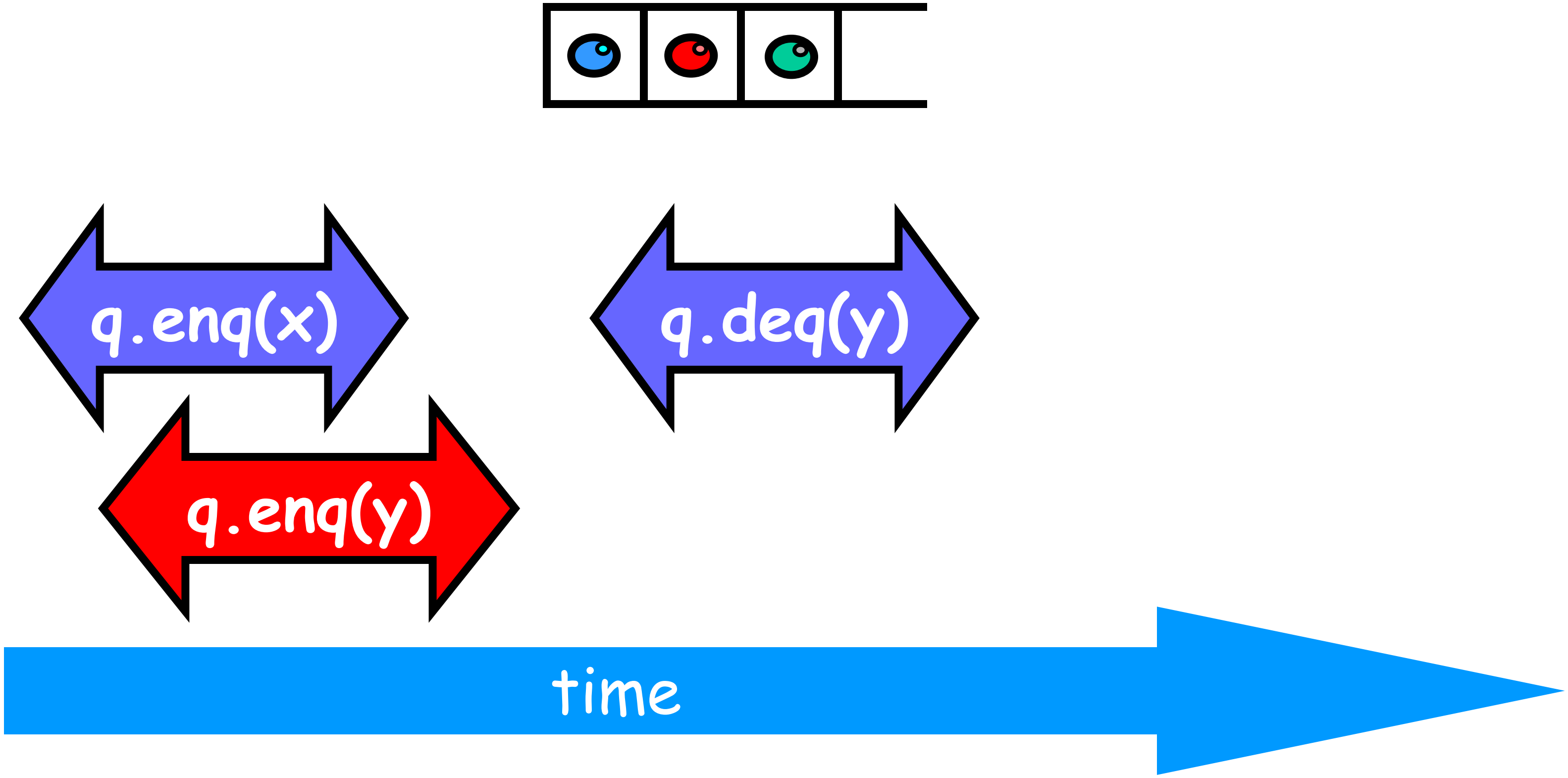
Example



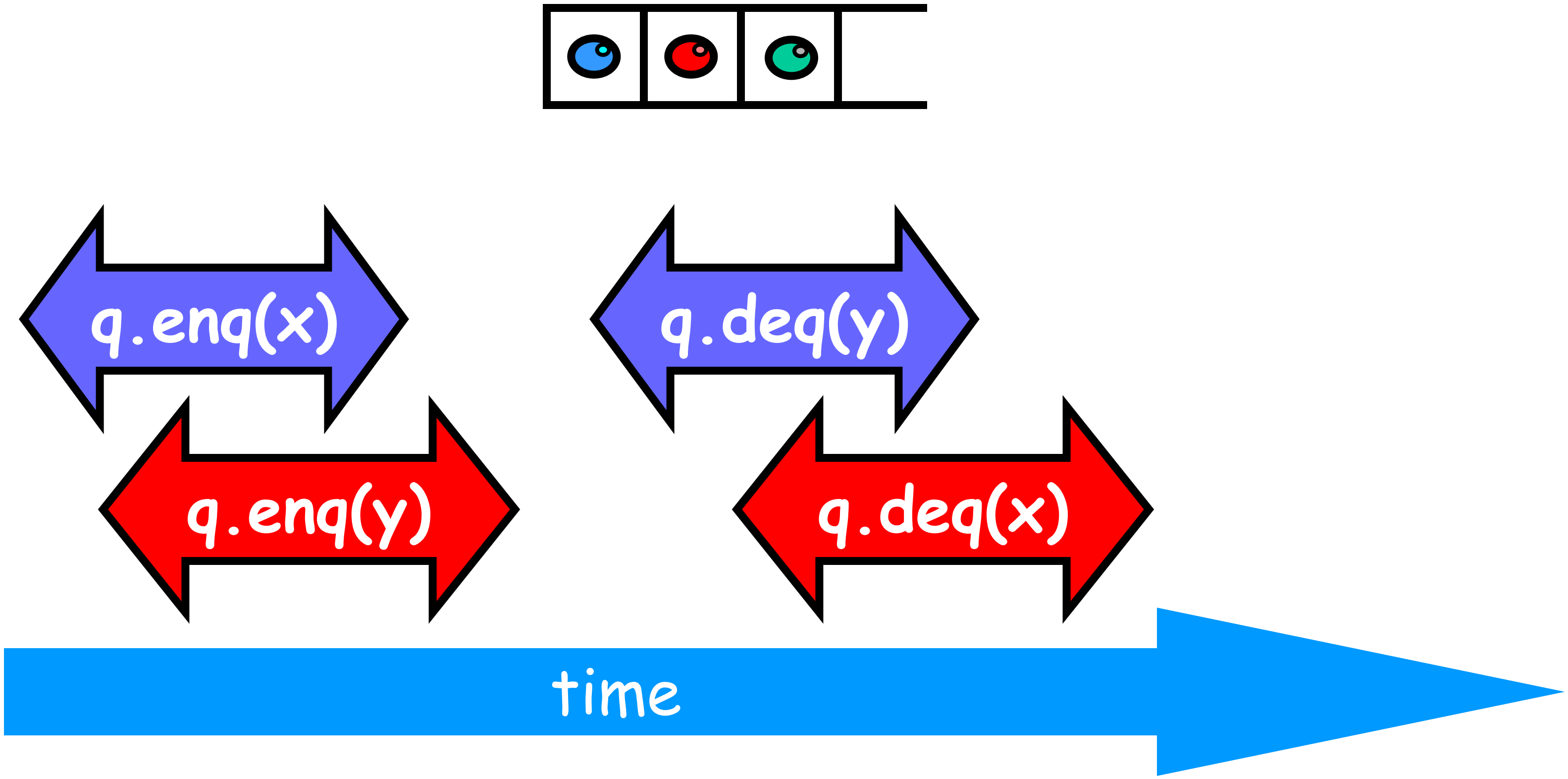
Example



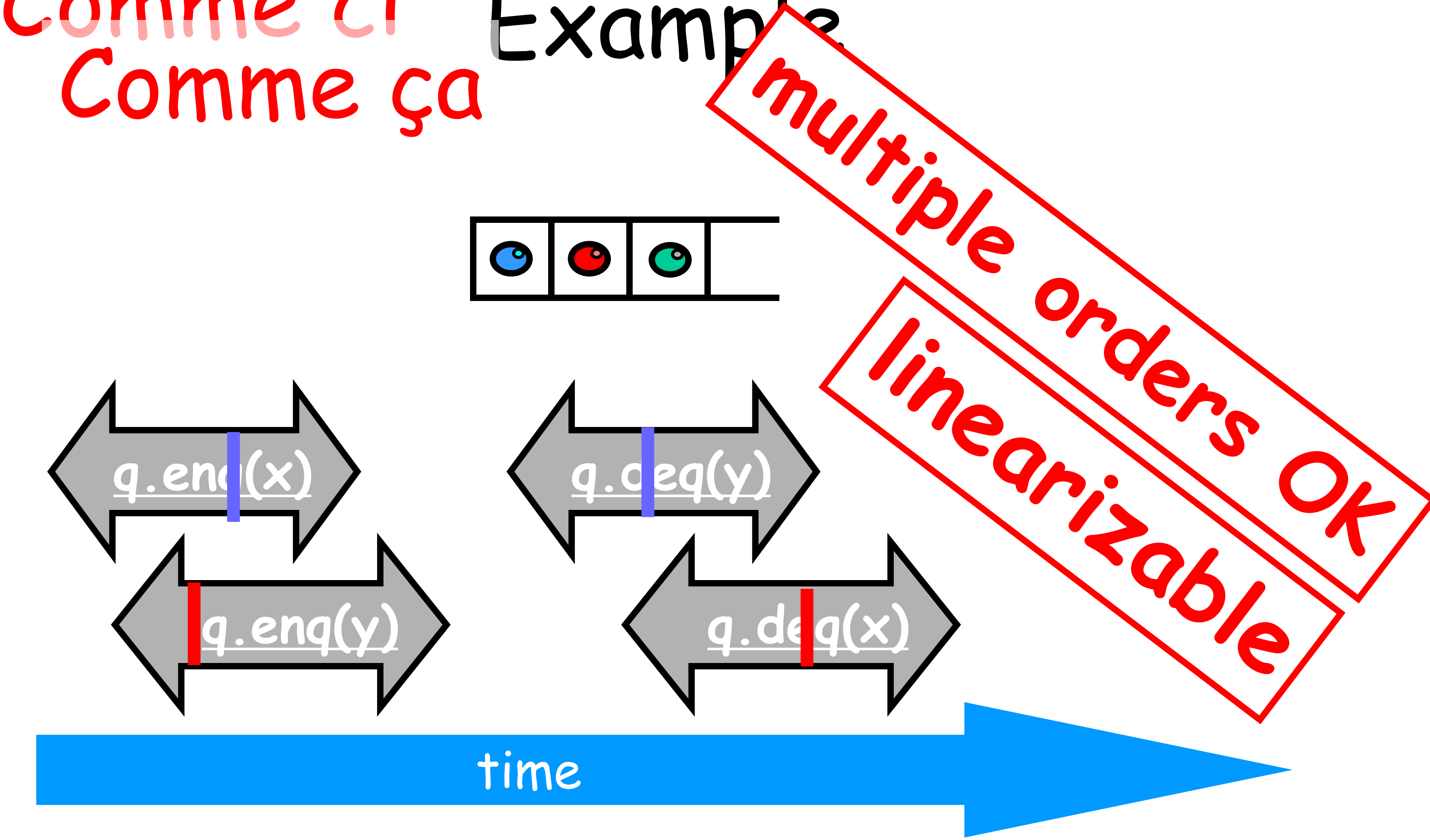
Example



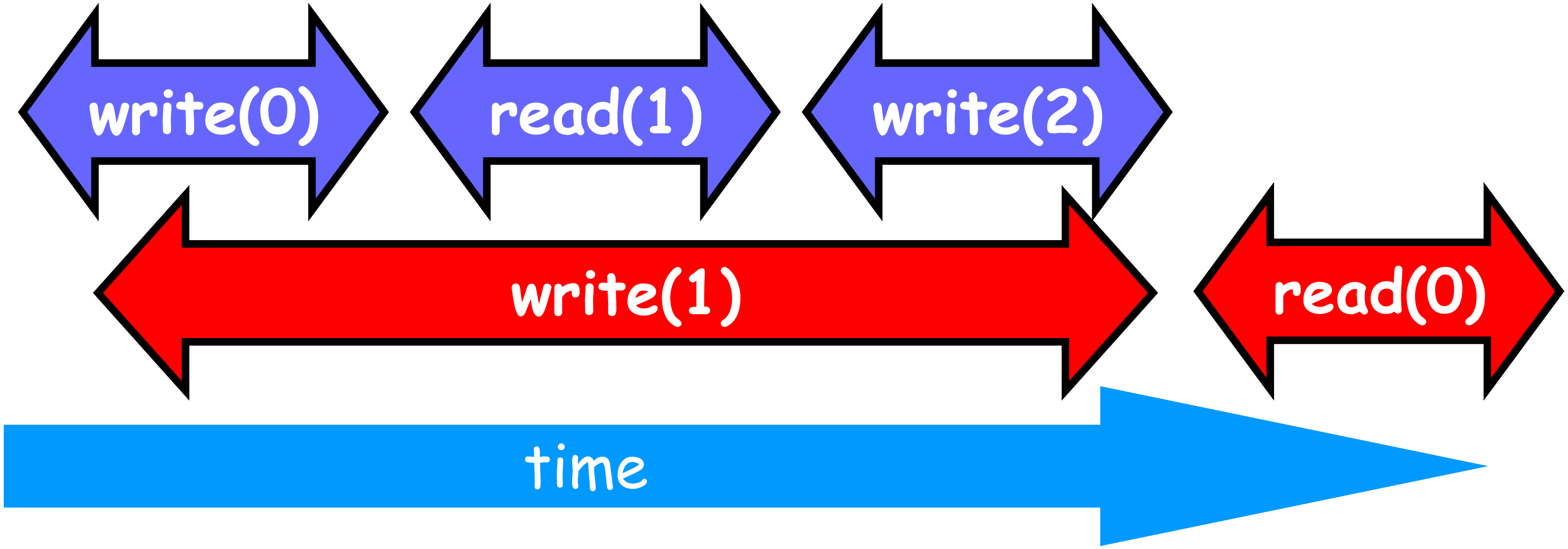
Example



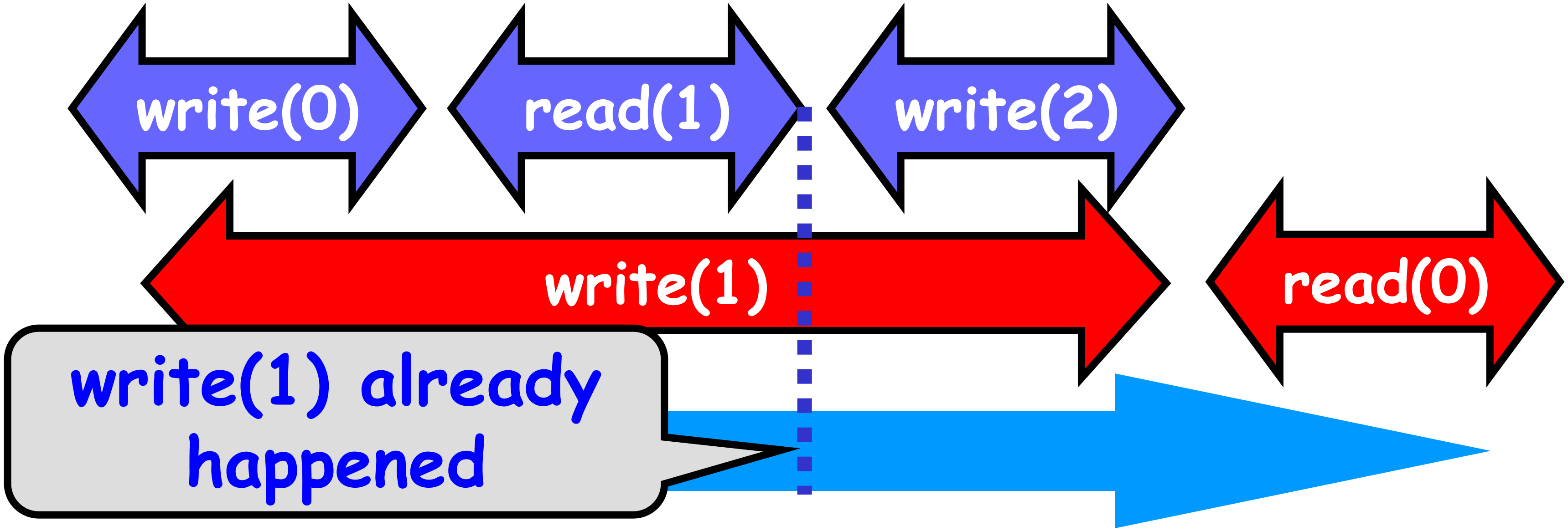
Comme ci Comme ça Example



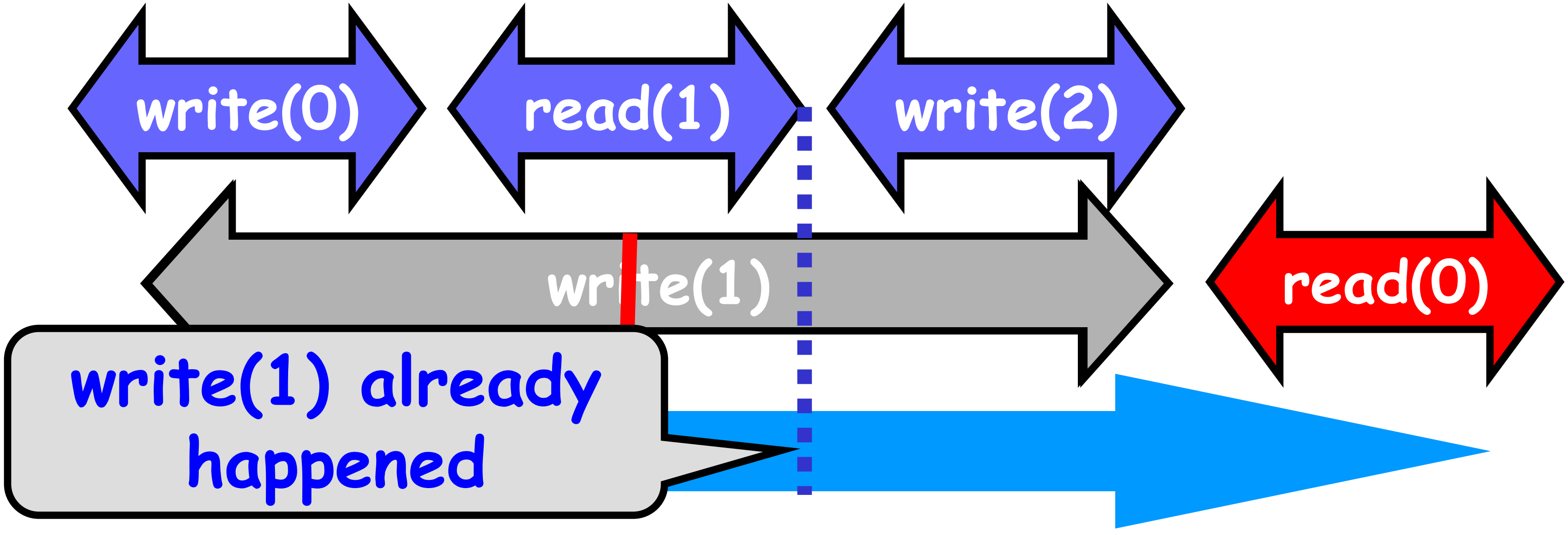
Read/Write Register Example



Read/Write Register Example

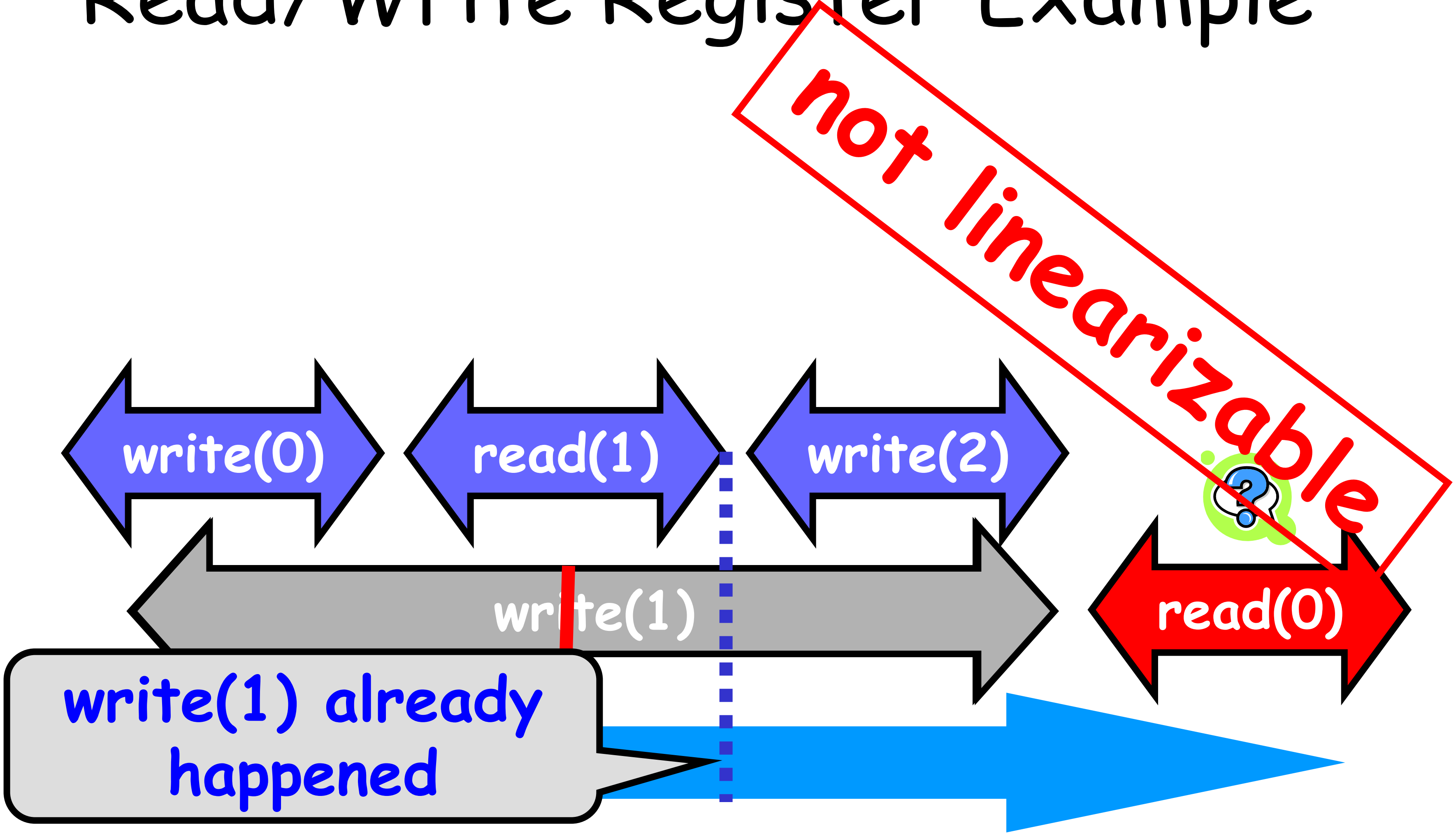


Read/Write Register Example

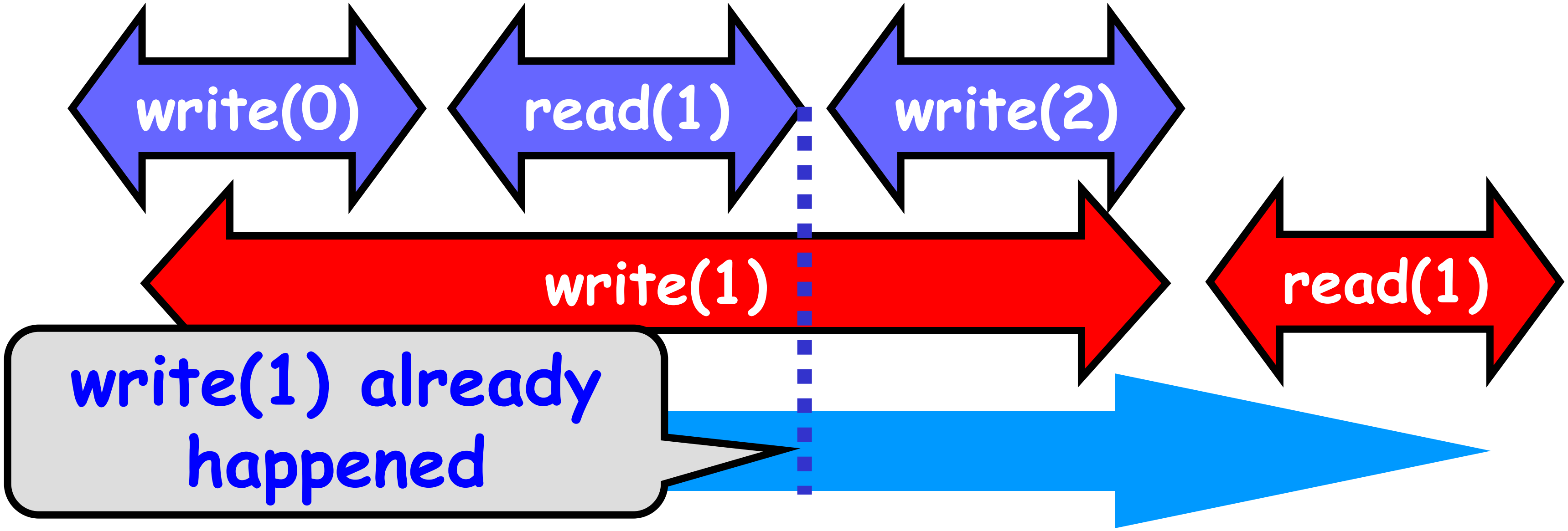


write(1) already happened

Read/Write Register Example

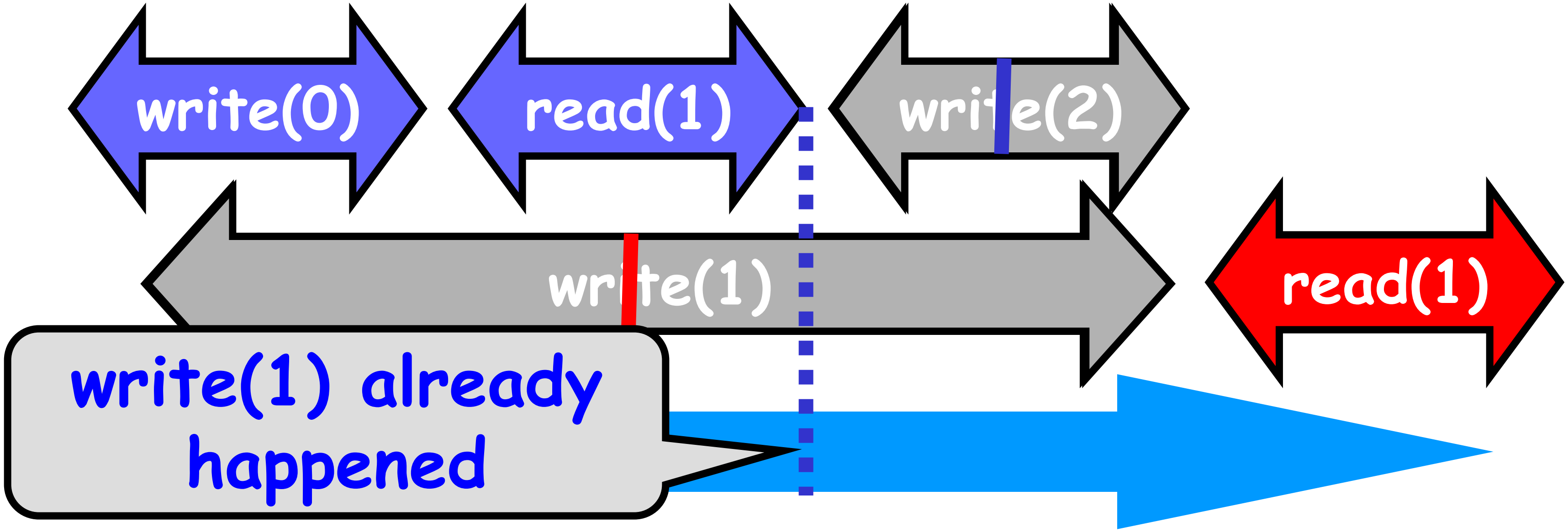


Read/Write Register Example



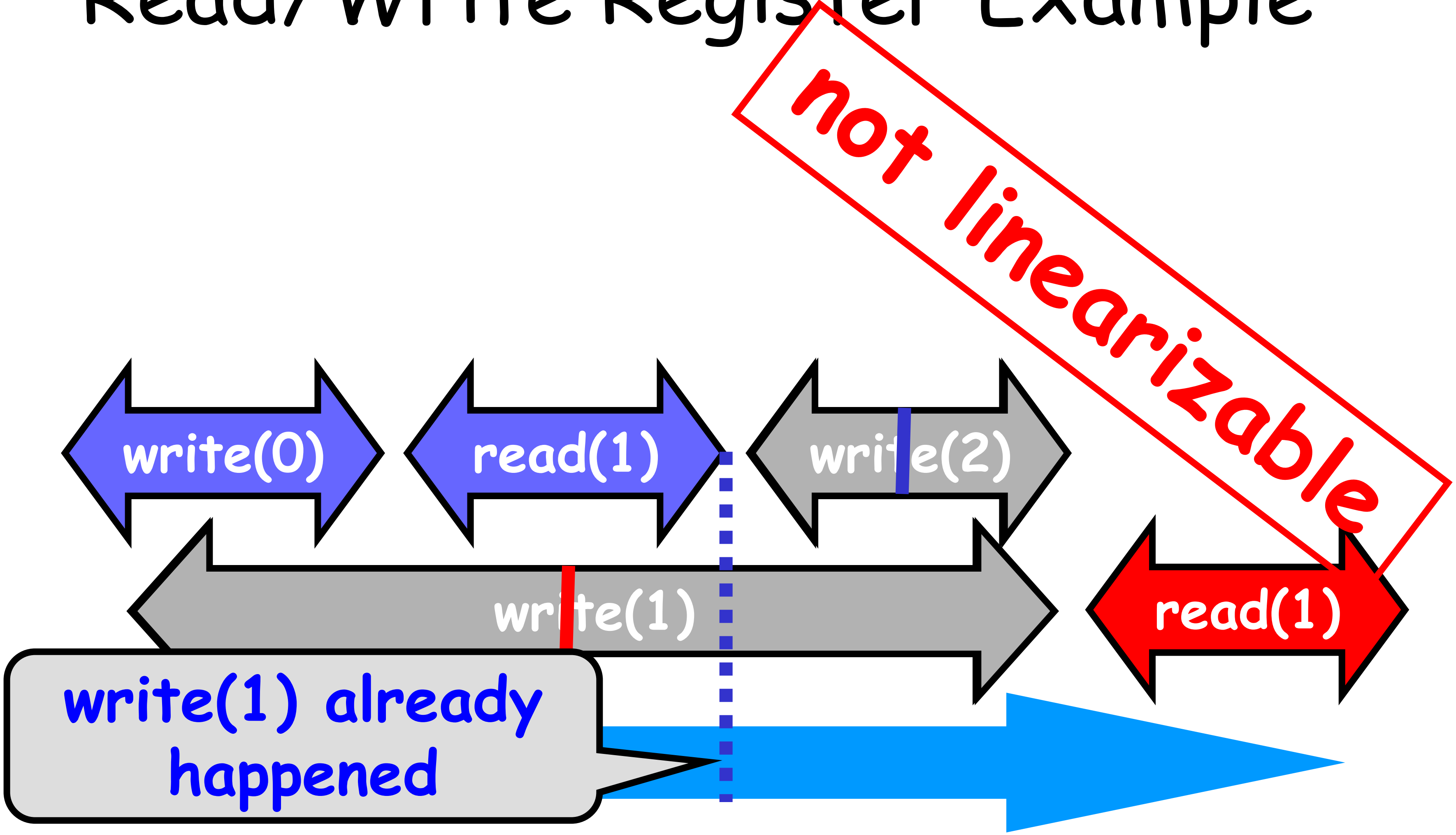
write(1) already happened

Read/Write Register Example

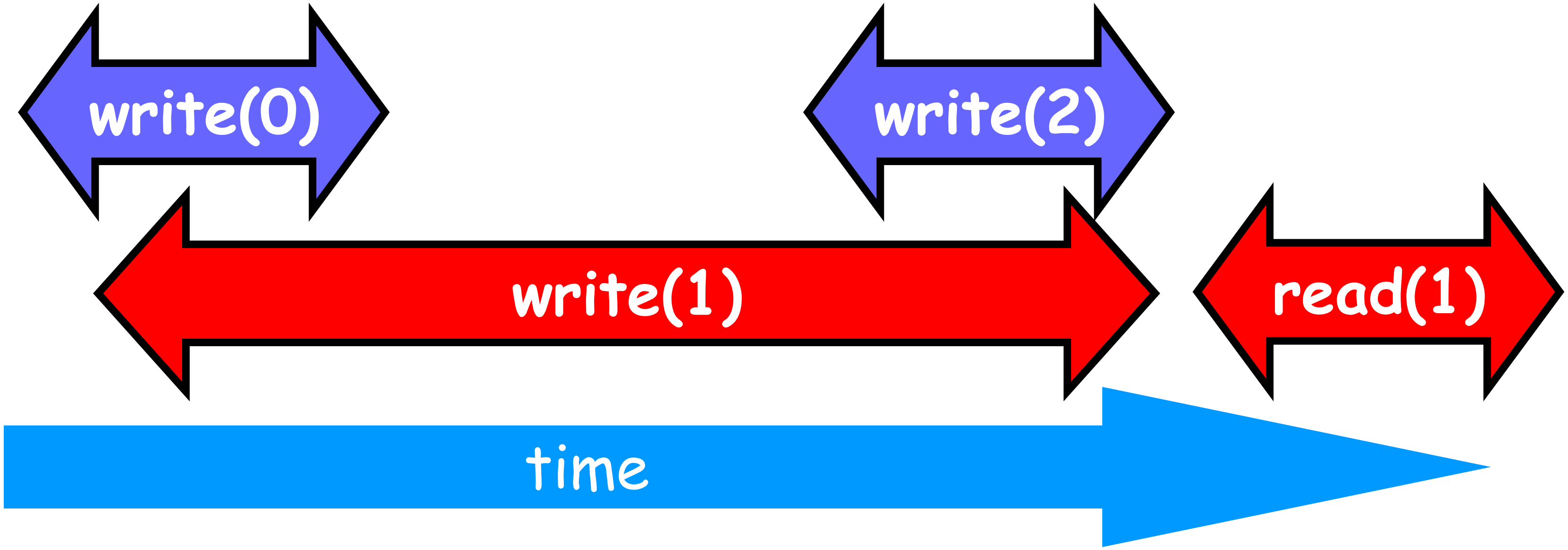


write(1) already happened

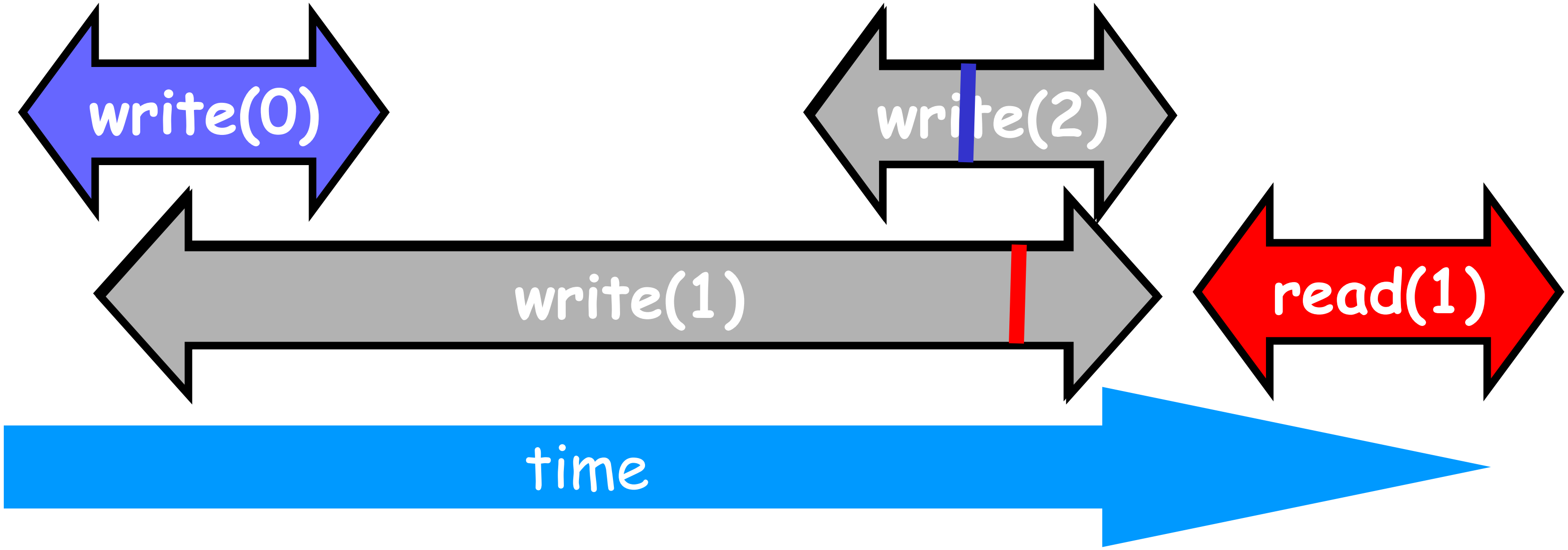
Read/Write Register Example



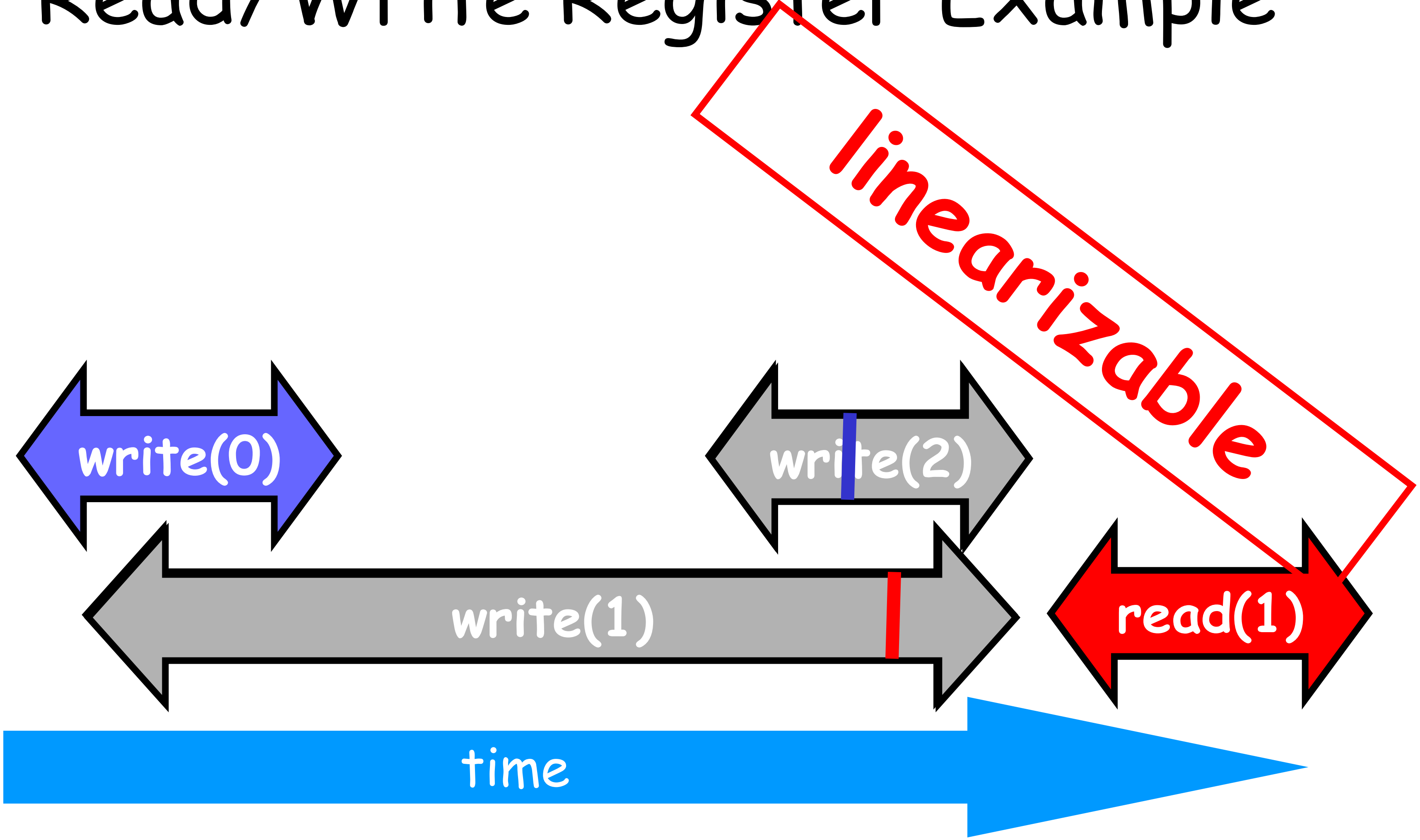
Read/Write Register Example



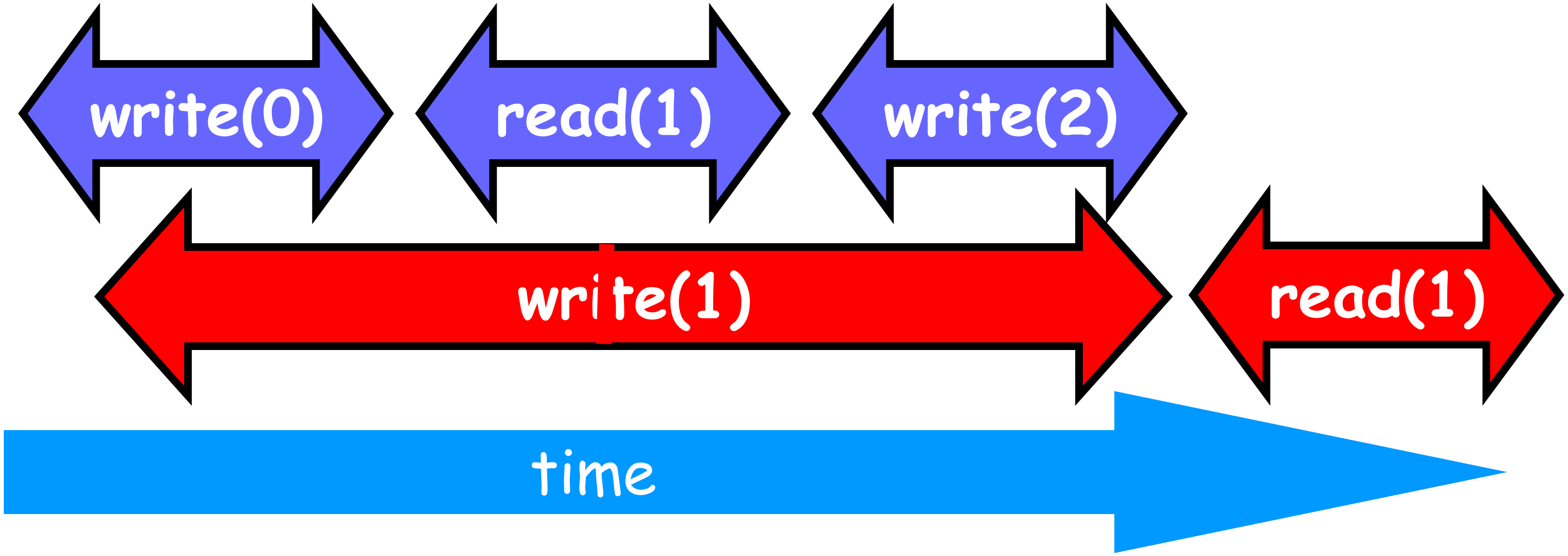
Read/Write Register Example



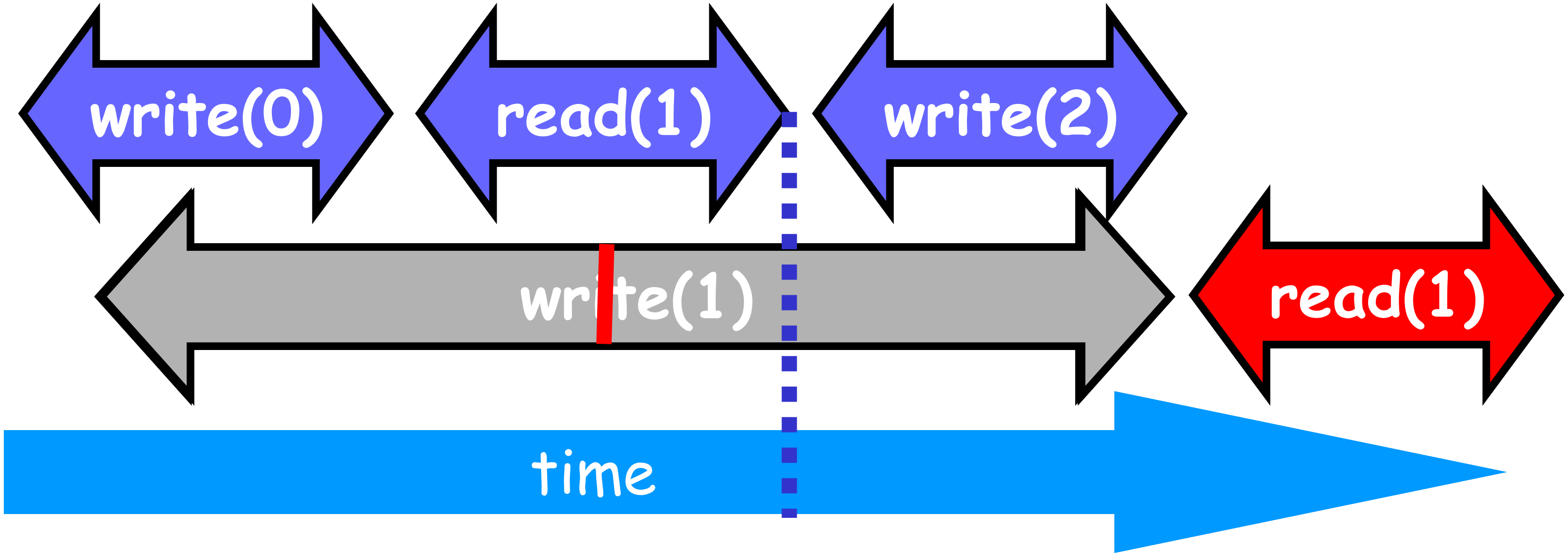
Read/Write Register Example



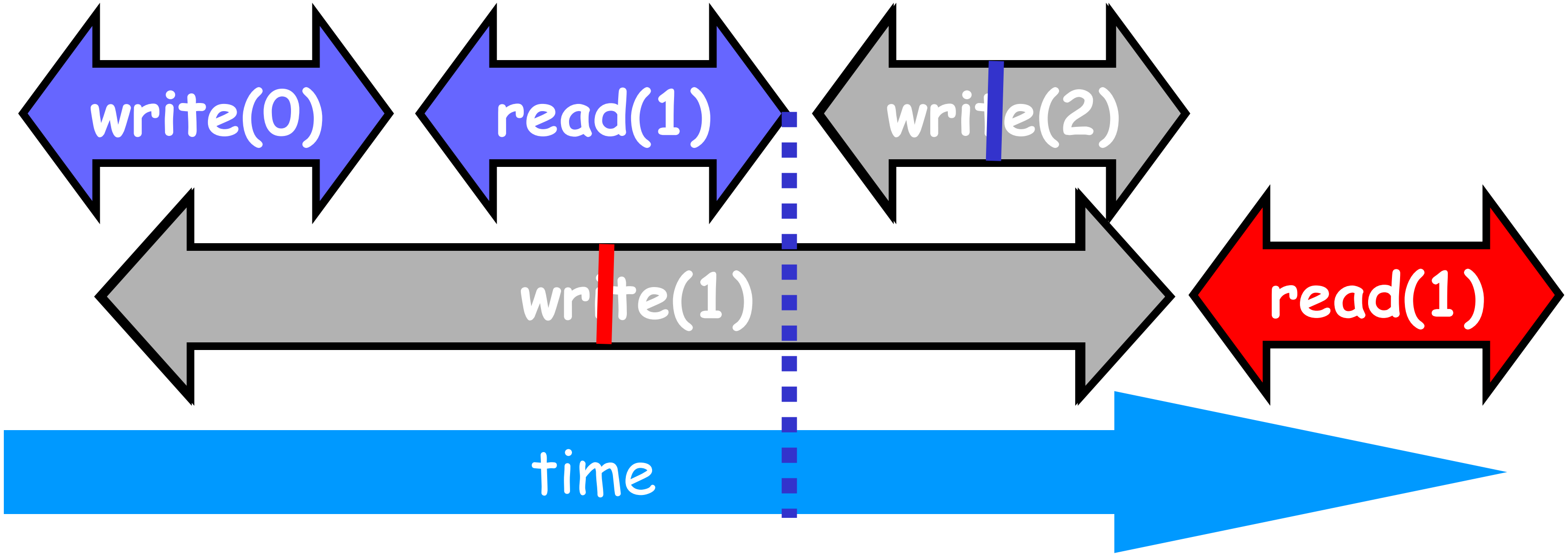
Read/Write Register Example



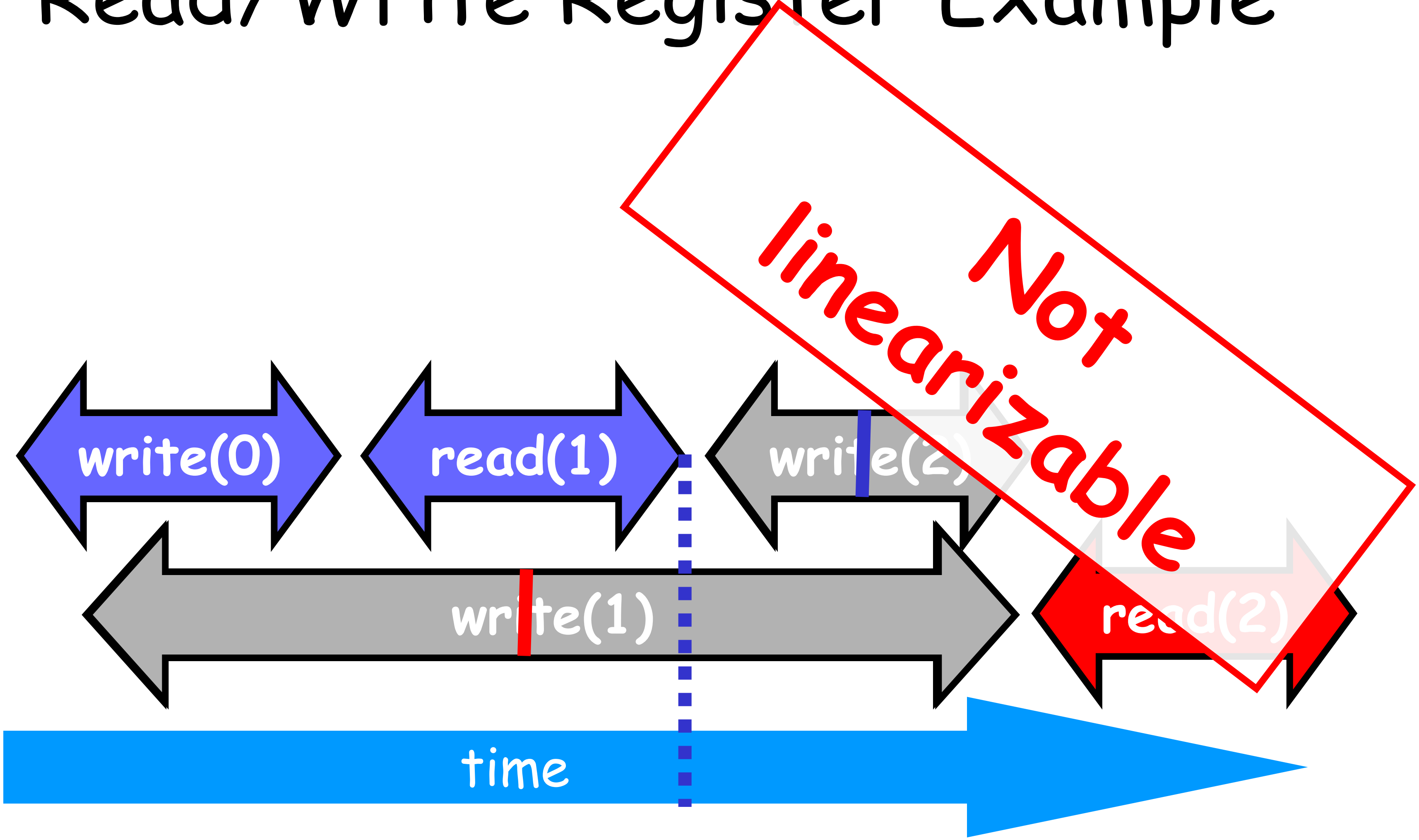
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Talking About Executions

- Why?
  - 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

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
    - Ask me why!

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

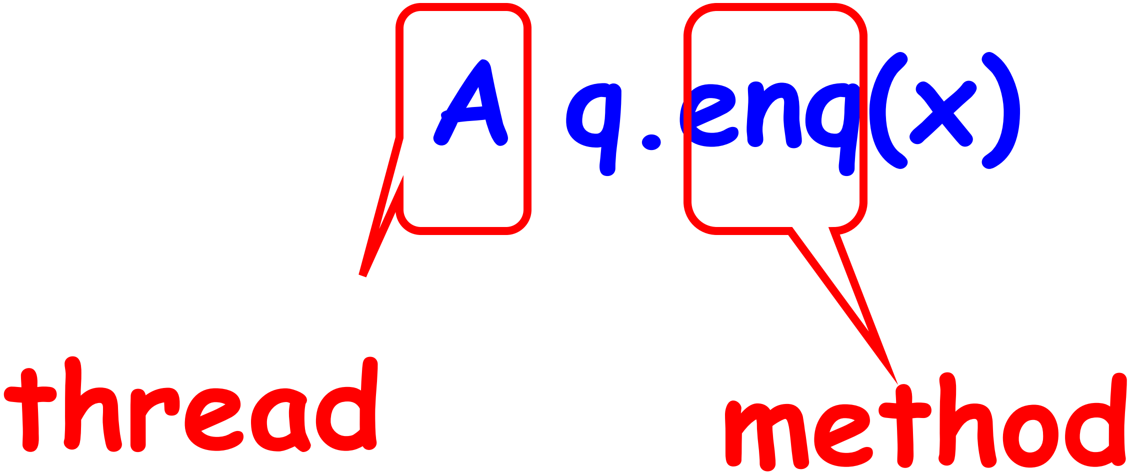
A q.enq(x)

Invocation Notation

A q.enq(x)

thread

Invocation Notation

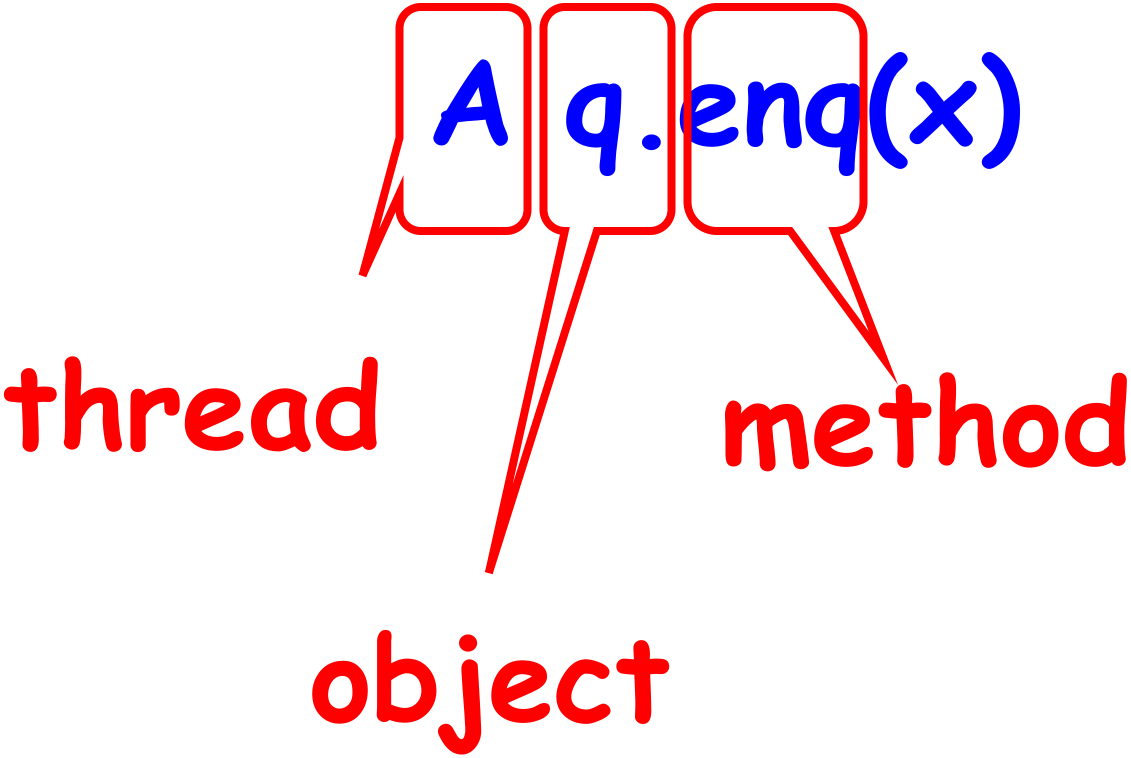


A q.enq(x)

thread

method

Invocation Notation



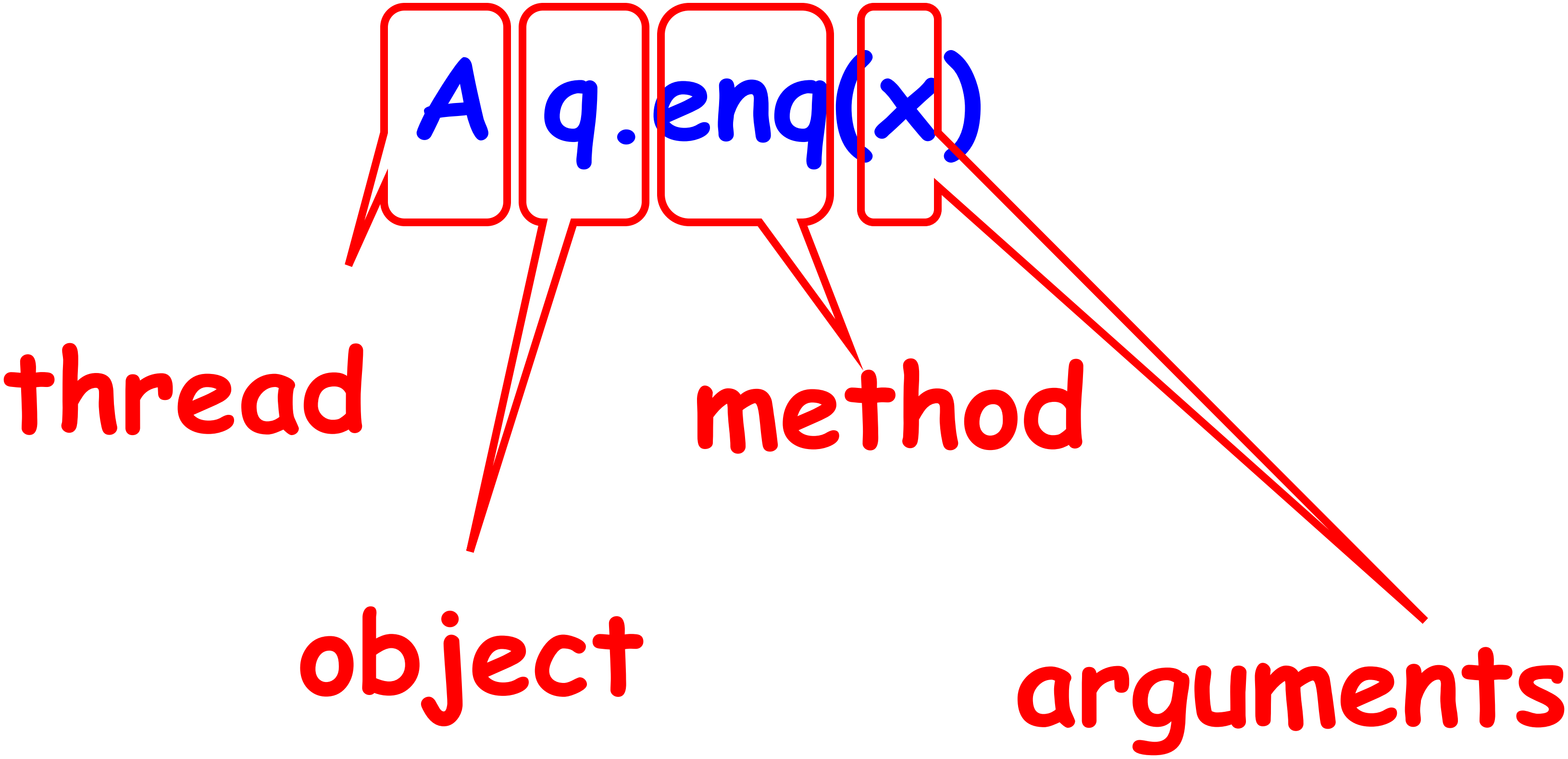
A q.enq(x)

thread

method

object

Invocation Notation



A q.enq(x)

thread

object

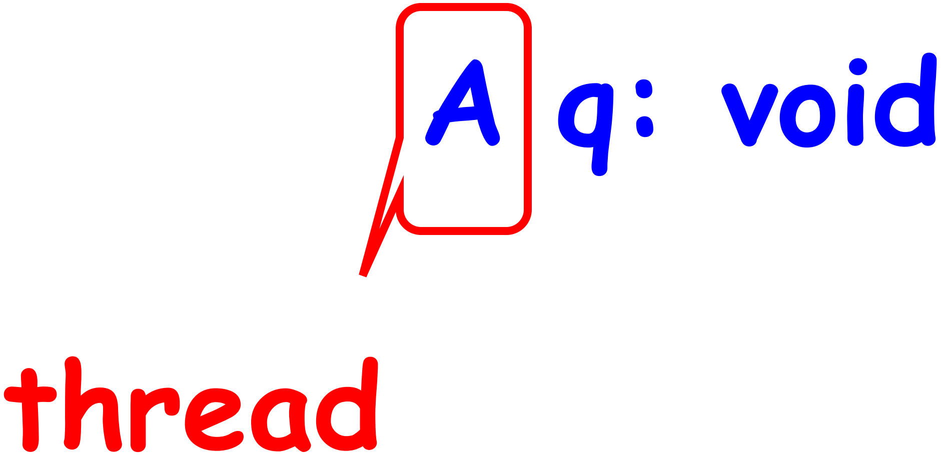
method

arguments

Response Notation

A q: void

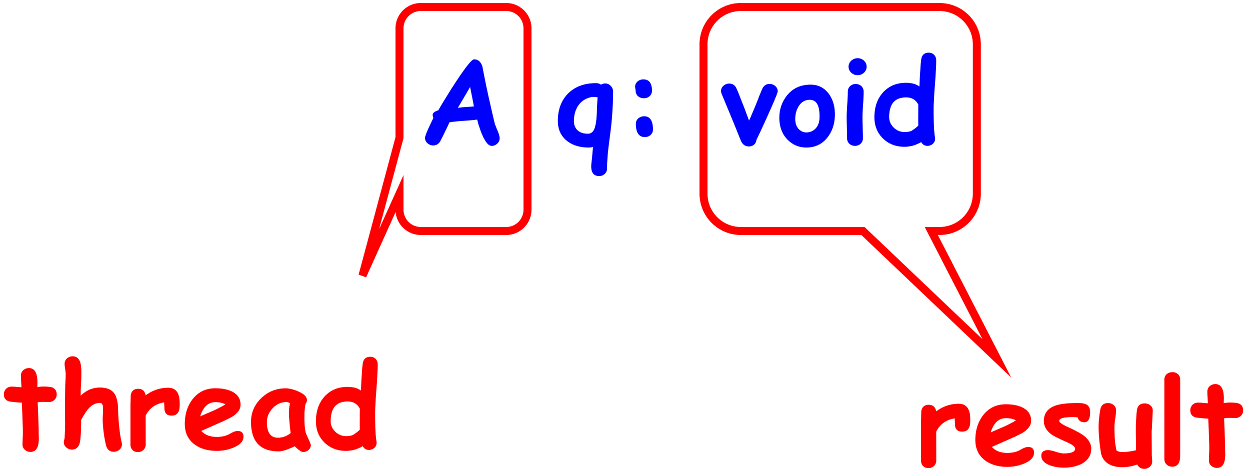
Response Notation



A q: void

thread

Response Notation

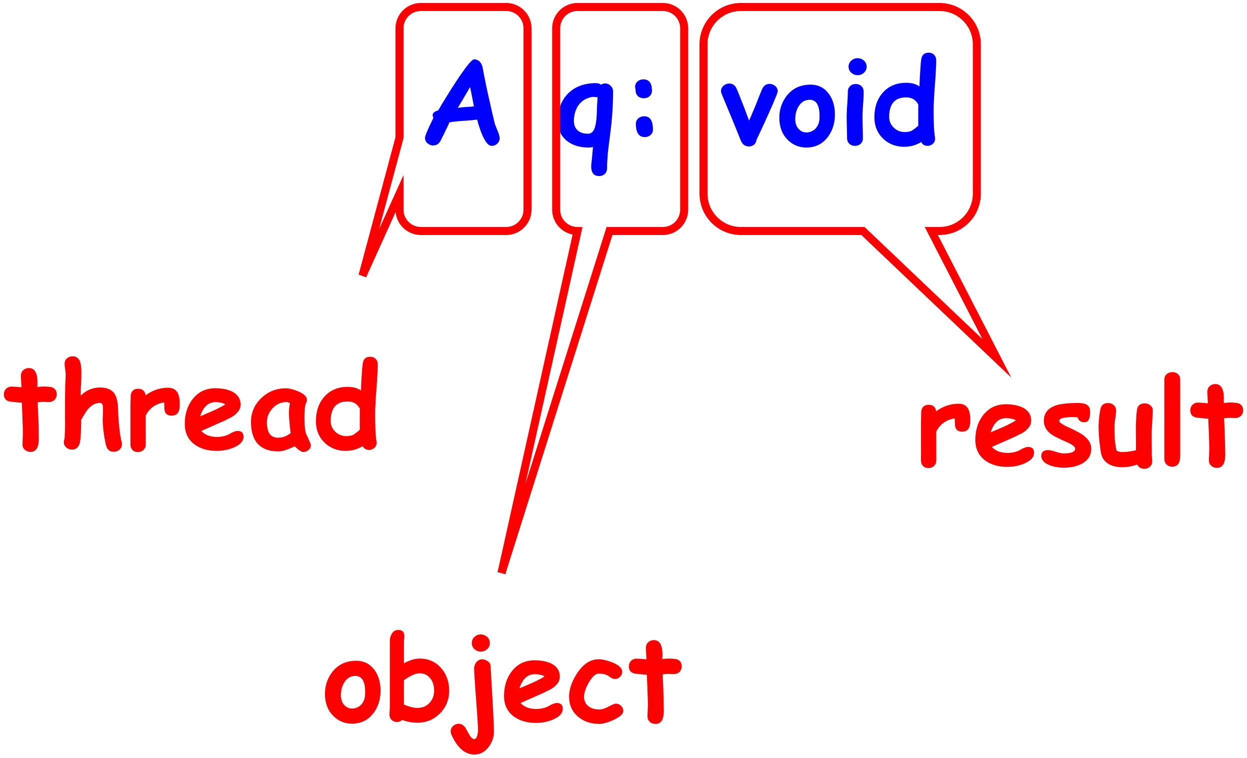


A q: void

thread

result

Response Notation



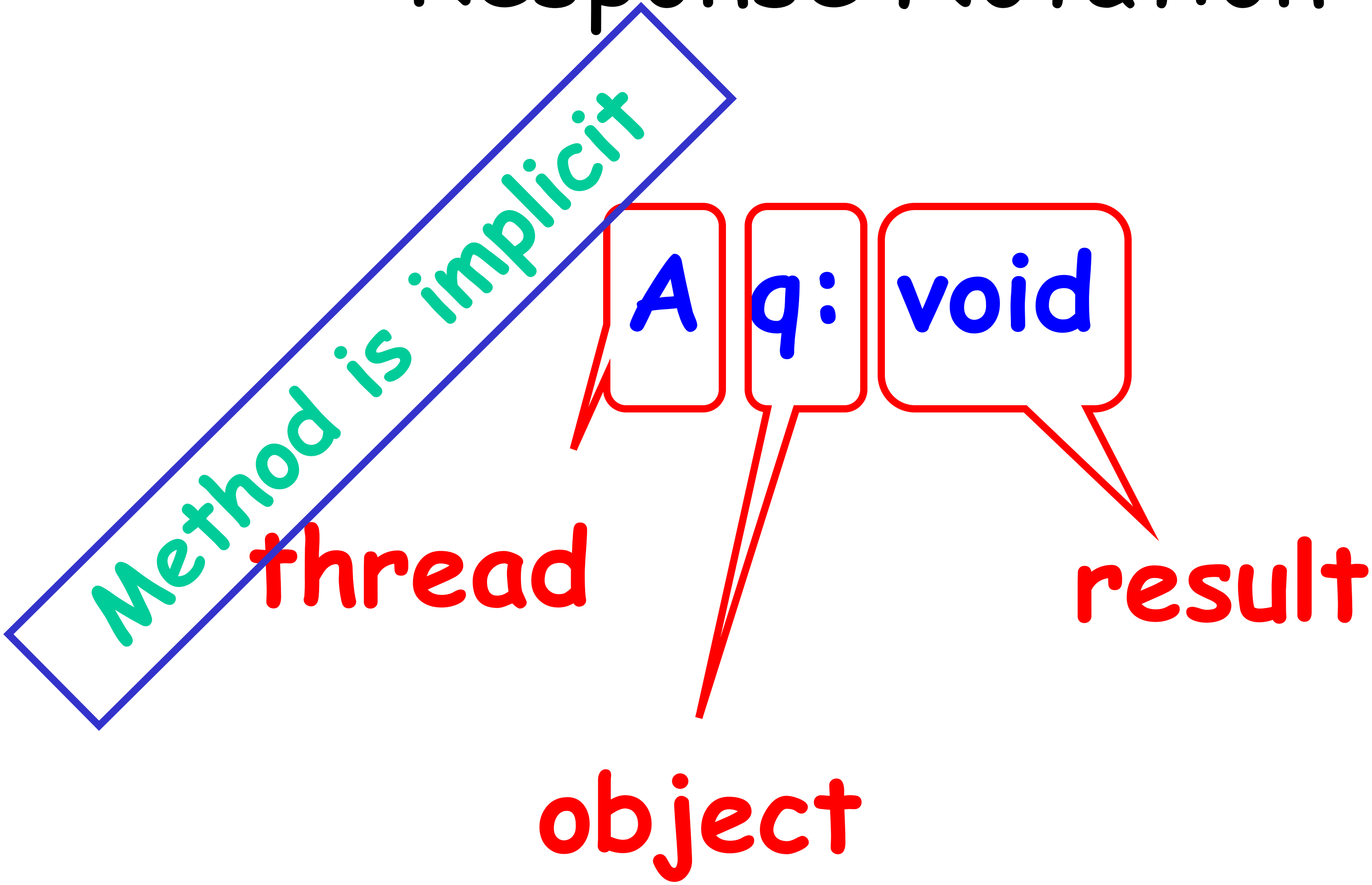
A q: void

thread

object

result

Response Notation



A q: void

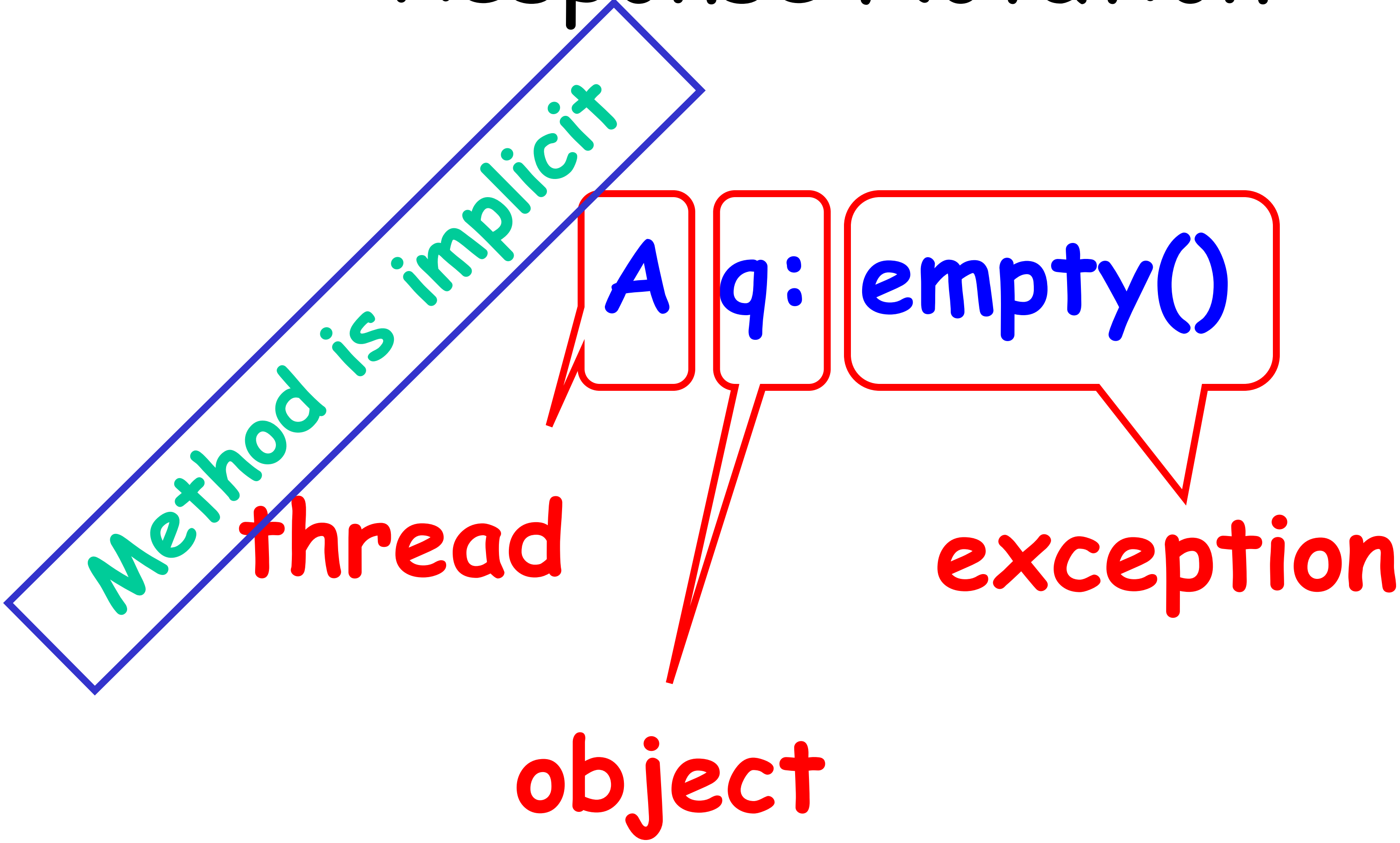
Method is implicit

thread

object

result

Response Notation



A q: empty()

thread

object

exception

Method is implicit

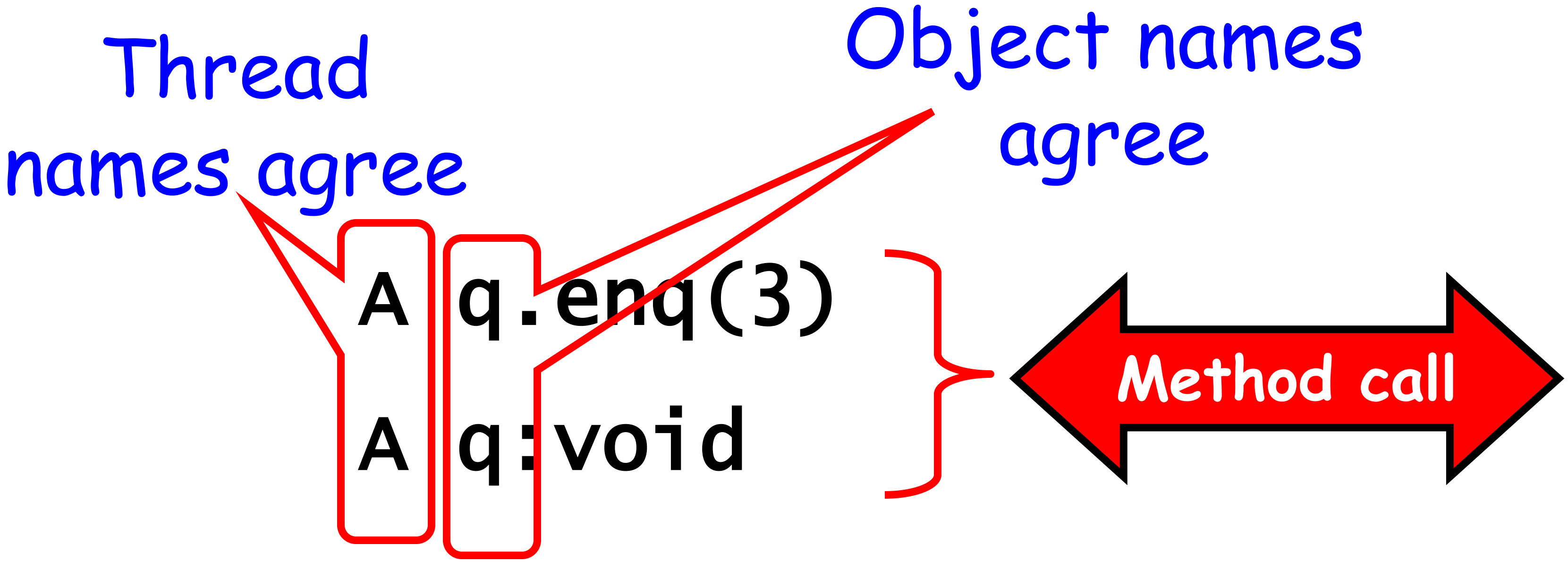
History - Describing an Execution

```
      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
```

Sequence of invocations and responses

Definition

- Invocation & response *match* if



Thread names agree

Object names agree

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

Method call

Object Projections

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

```
      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 respnse

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
```

May or may not have taken effect

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
```

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

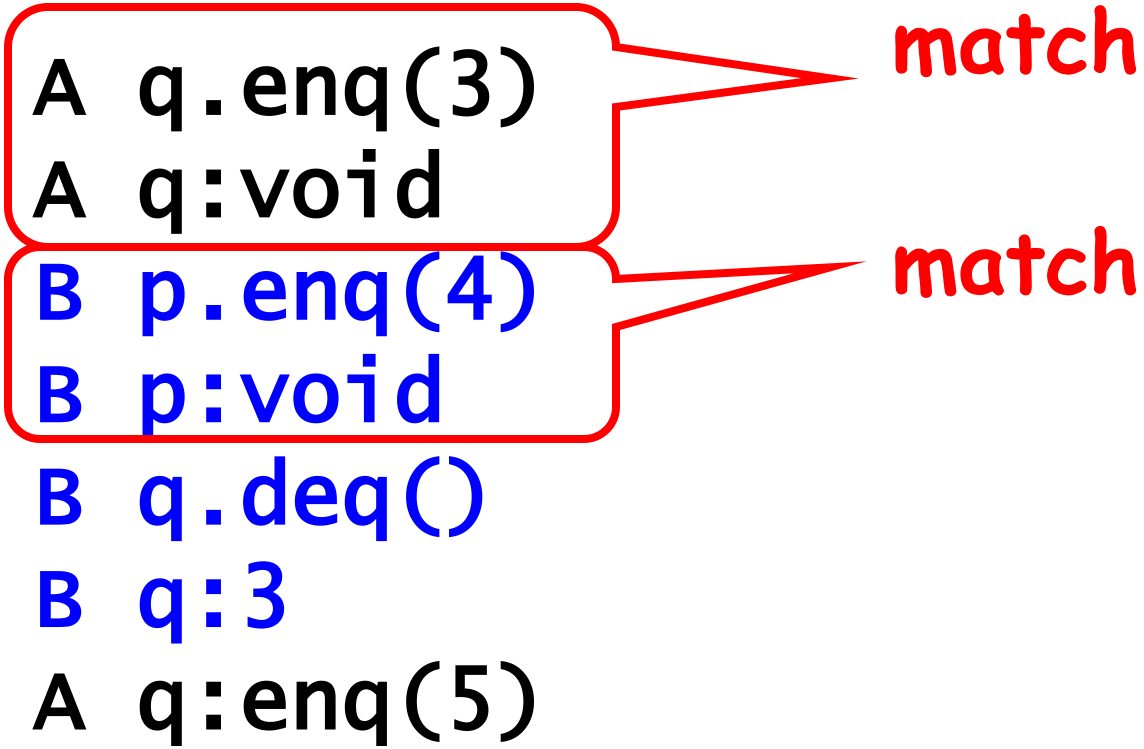
```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

Sequential Histories

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

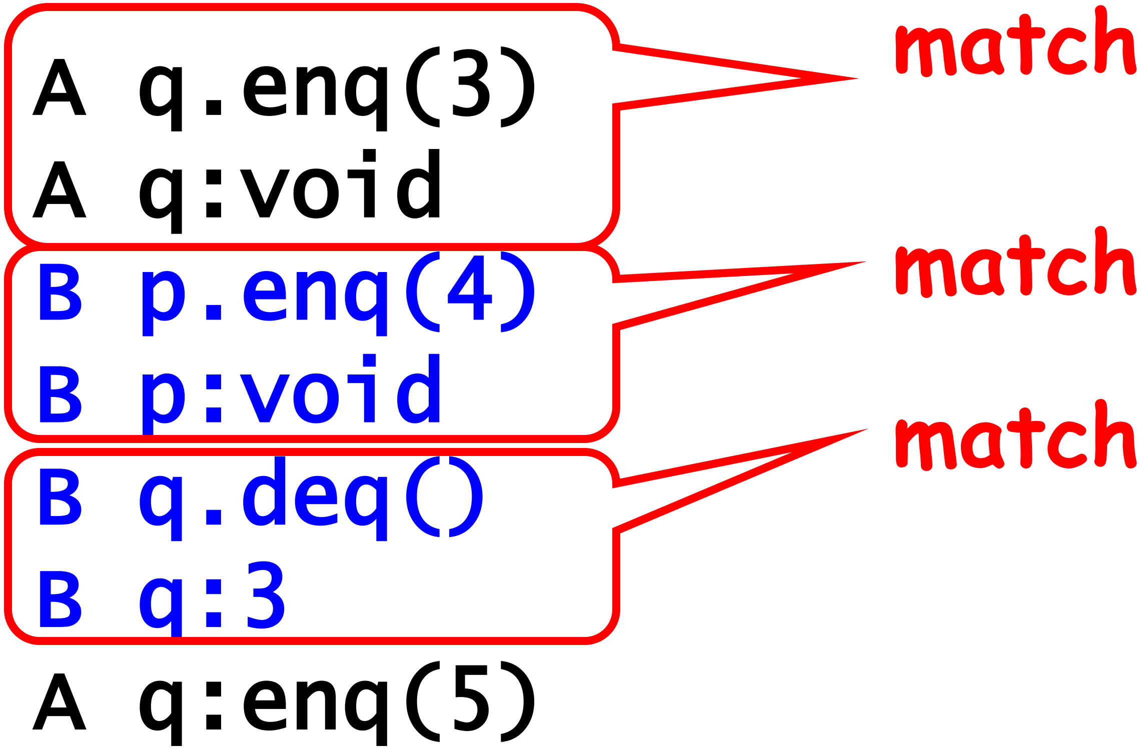
match

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



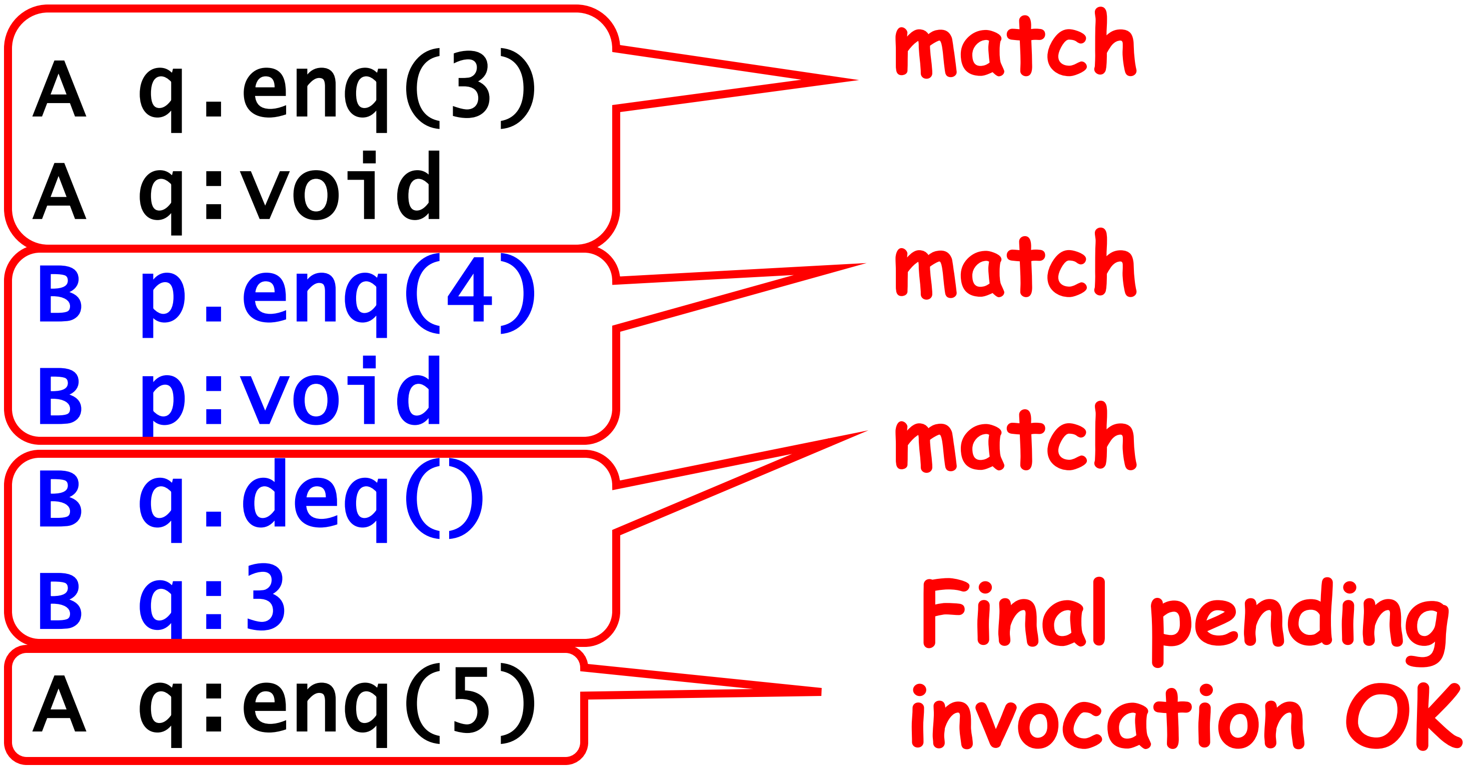
```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

match

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)
```

match

match

match

Final pending invocation OK

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)
```

match

match

match

Final pending invocation OK

Method calls of different threads do not interleave

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

$$\left\{\begin{array}{l} H|A = G|A \\ H|B = G|B \end{array}\right.$$

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

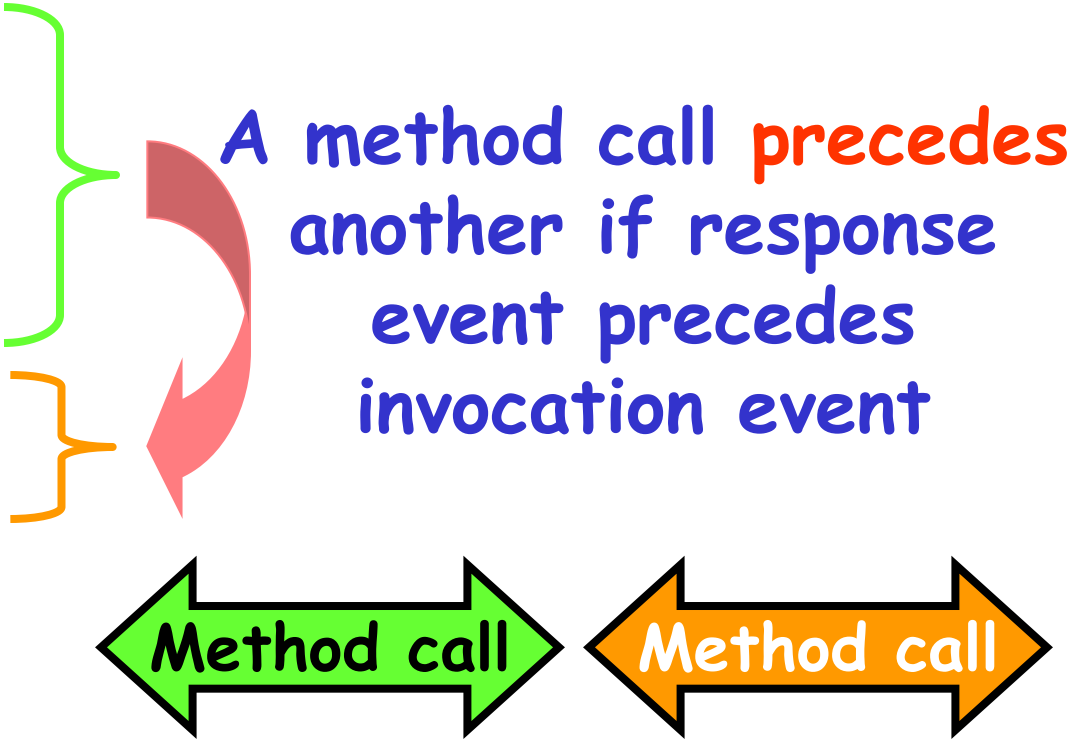
- A sequential specification is some way of telling whether a
  - Single-thread, single-object history
  - Is legal
- For example:
  - Pre and post-conditions
  - But plenty of other techniques exist …

Legal Histories

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

Precedence

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

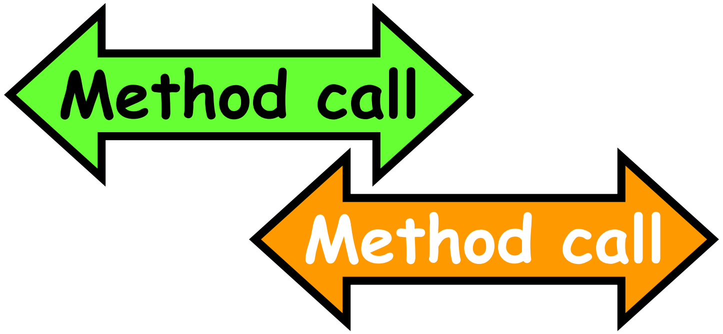


A method call **precedes** another if response event precedes invocation event

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

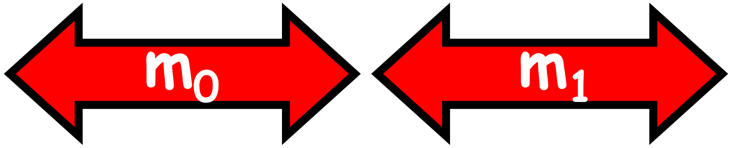


Method call

Method call

Notation

- Given
  - History **H**
  - method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**
- We say $m_0 \rightarrow_H m_1$, if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$



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

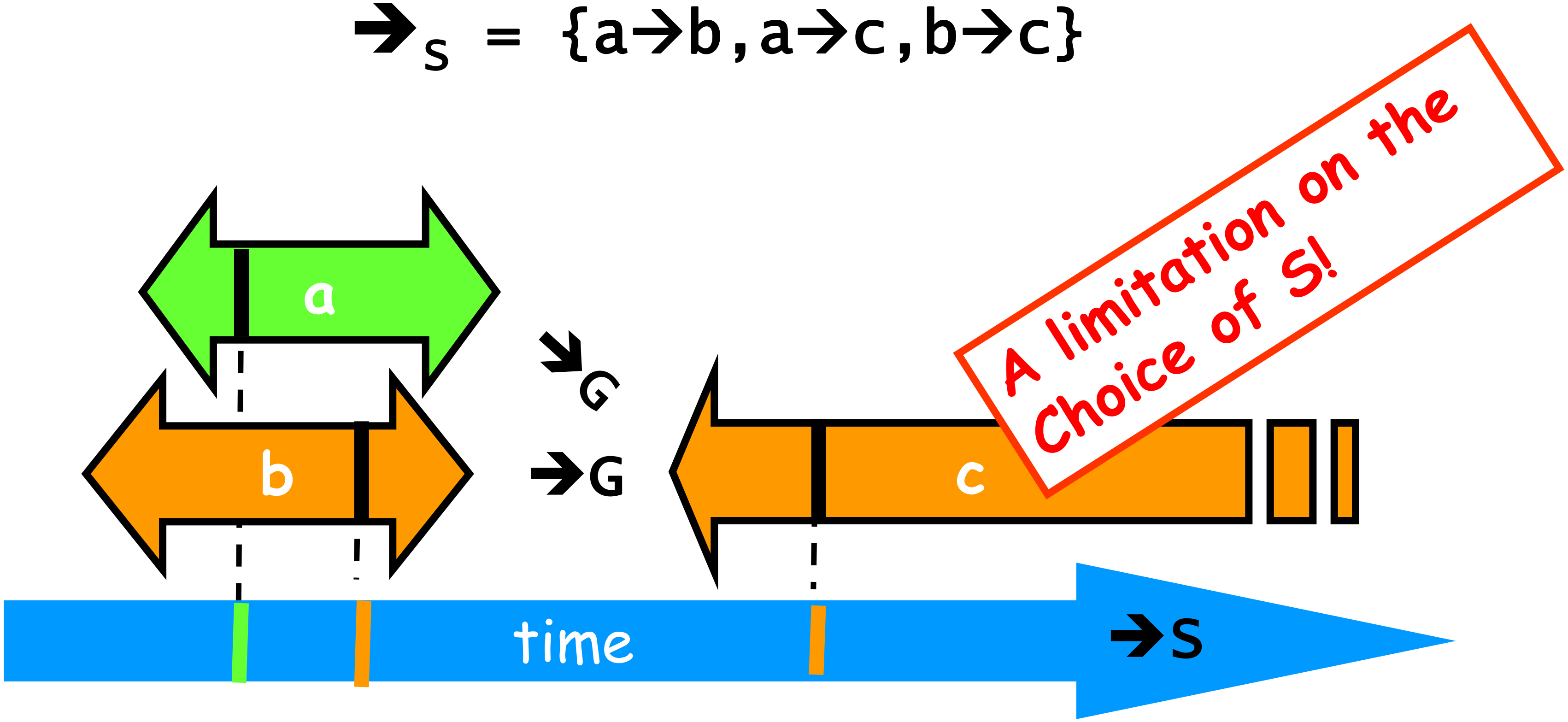
Linearizability

- 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 $\rightarrow_G \subset \rightarrow_S$

What is $\rightarrow_G \subset \rightarrow_S$

$\rightarrow_G$ = {a→c,b→c}

$\rightarrow_S$ = {a→b,a→c,b→c}



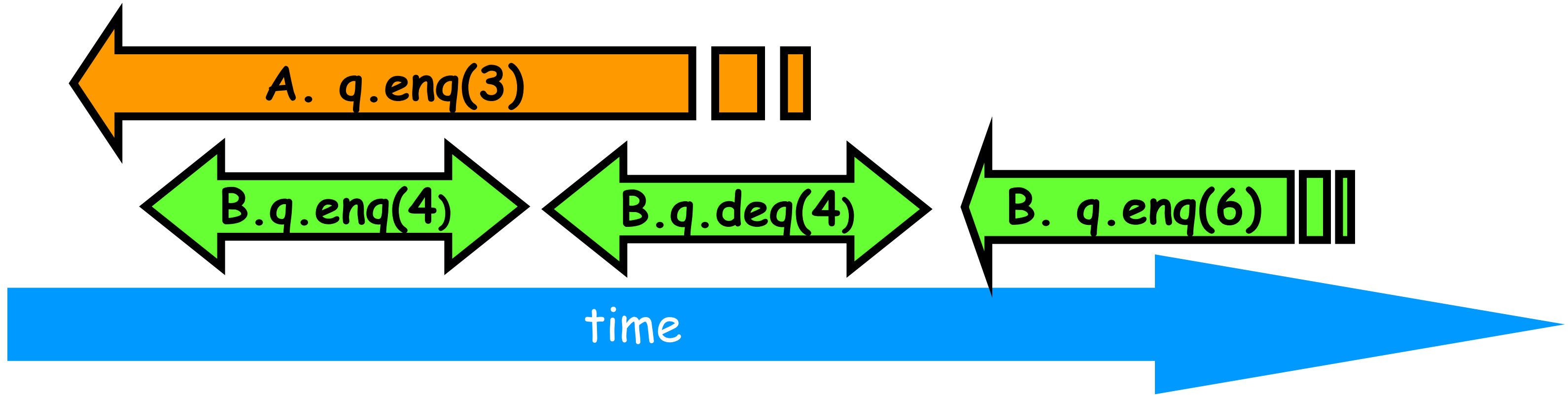
A limitation on the Choice of S!

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

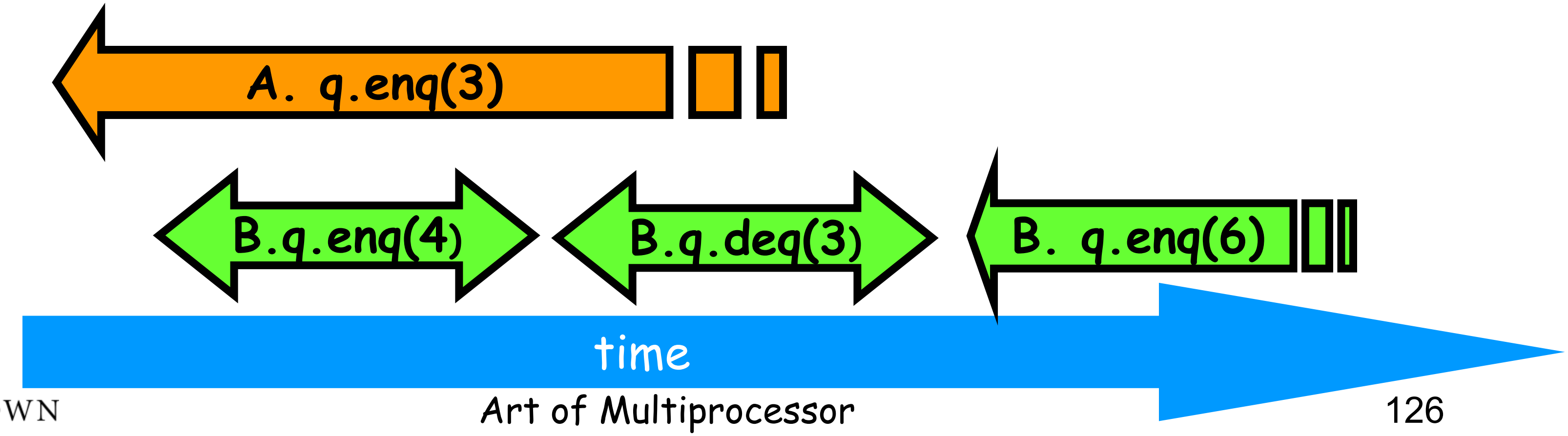
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



Example

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

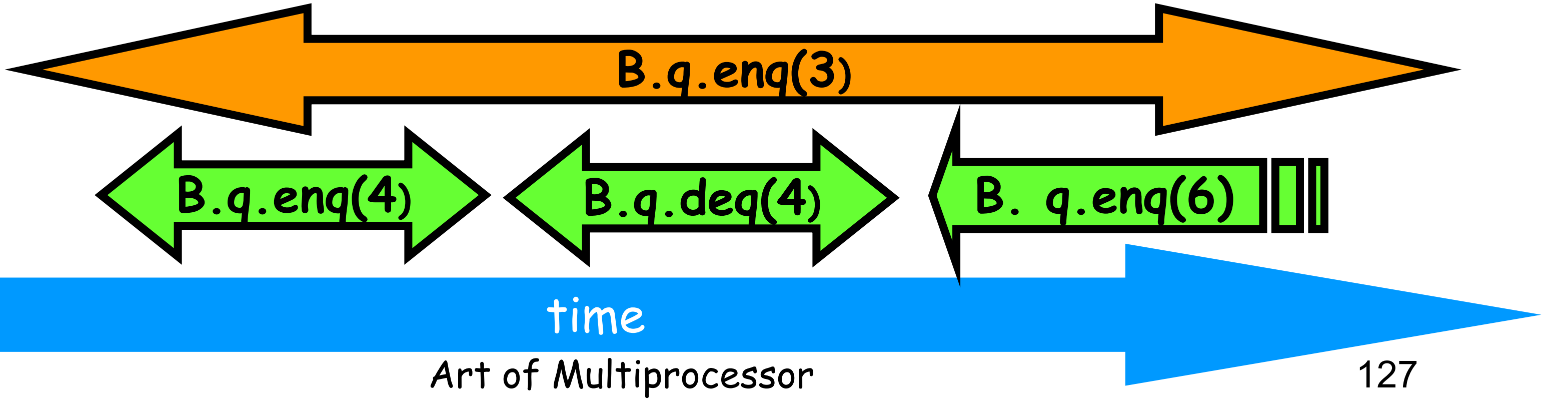
Complete this pending invocation



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
```

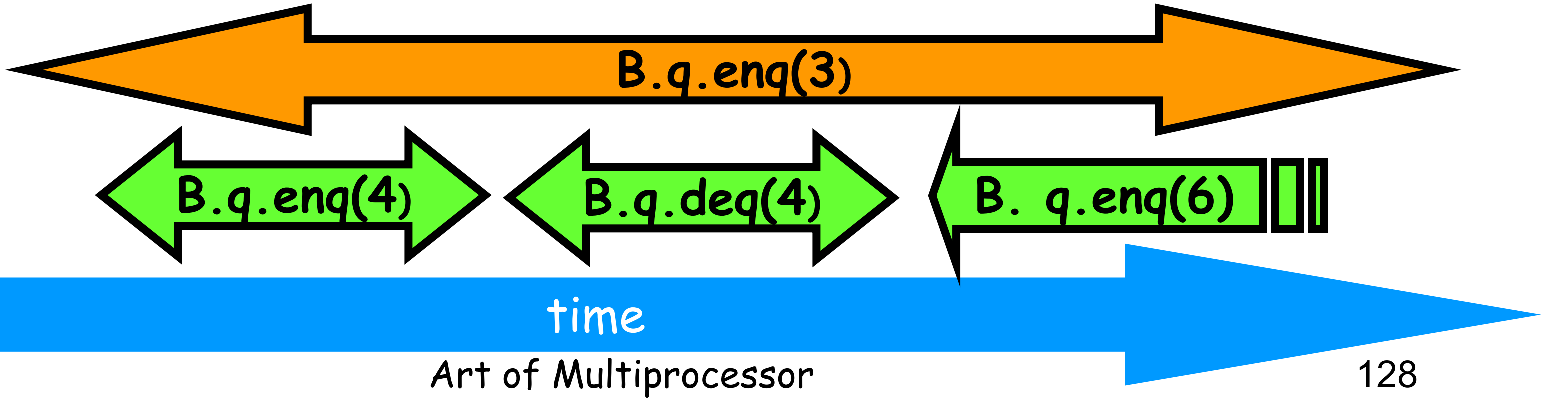
Complete this pending invocation



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
```

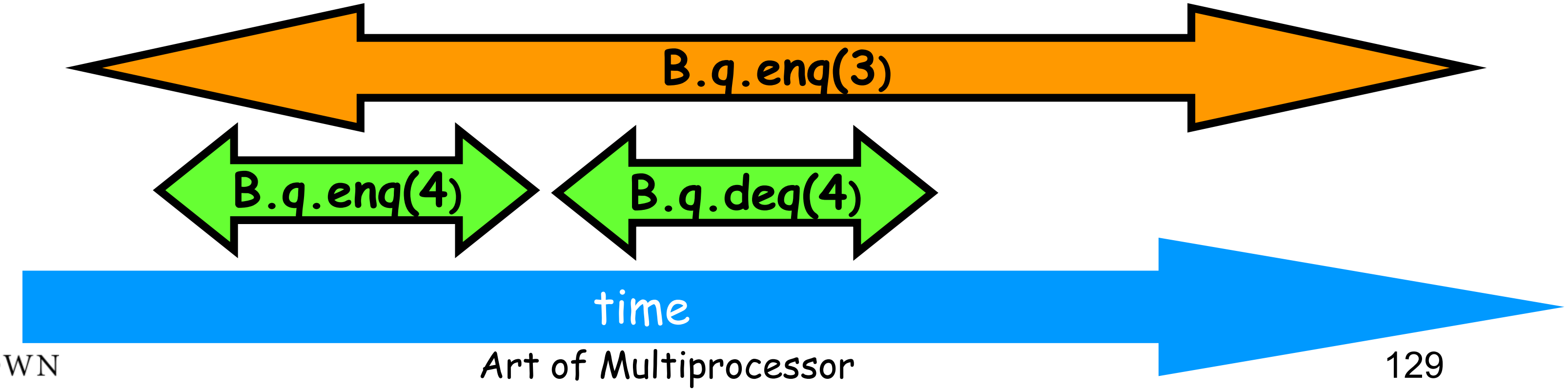
discard this one



Example

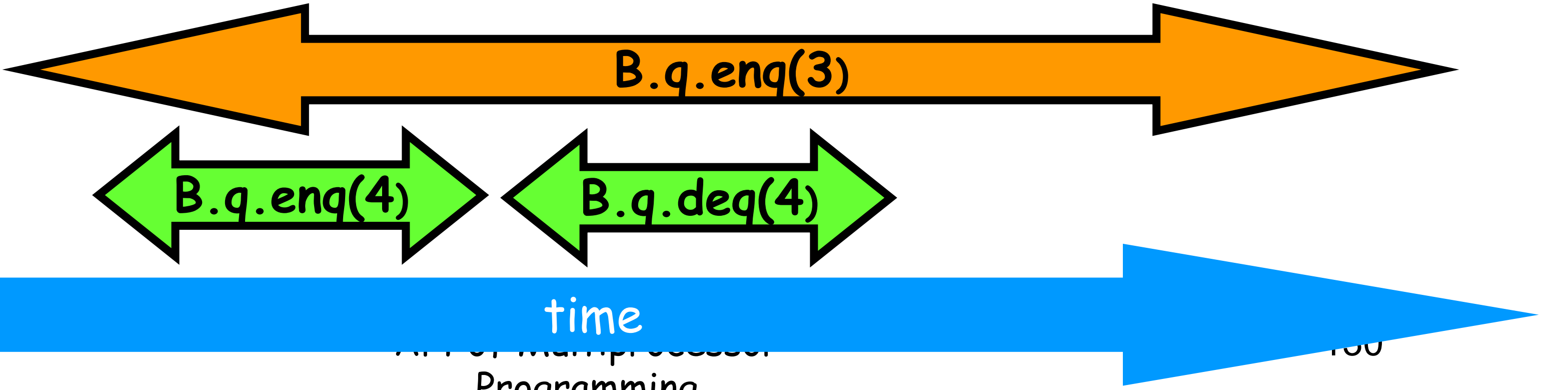
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4

A q:void
```

discard this one



Example

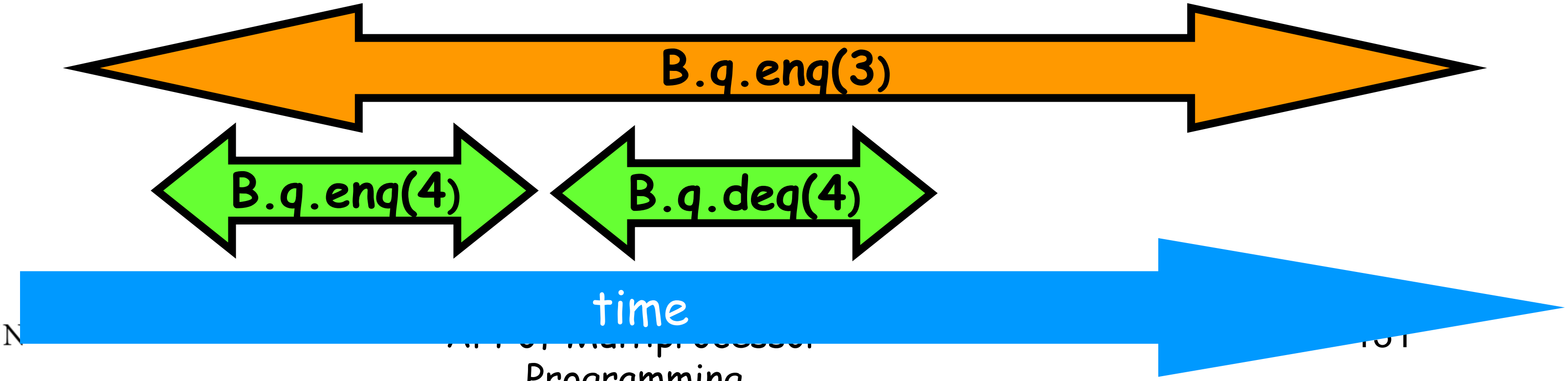
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



Example

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

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

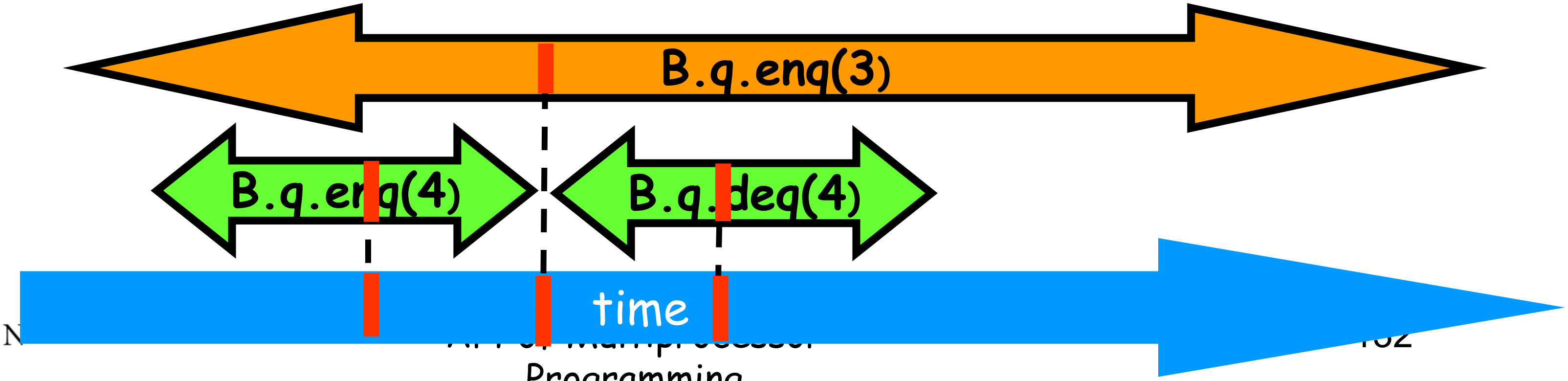


Example

```
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
```



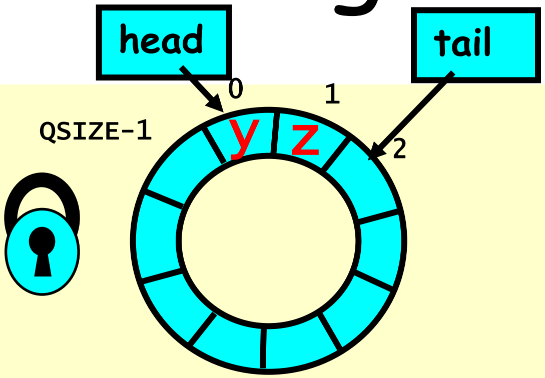
Composability Theorem

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

Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

Reasoning About Lineraizability: Locking



```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

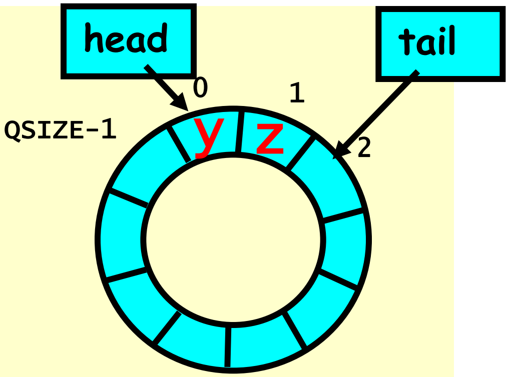
Implementation: Enq

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

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

Linearization order is order lock released

More Reasoning: Lock-free

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  Item[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;
}}
```



More Reasoning

```
public class LockFreeQ

  int head = 0, tail =
  Item[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
- 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**

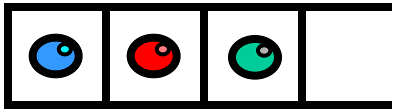
~~Where $\rightarrow_G \subset \rightarrow_S$~~

Differs from linearizability

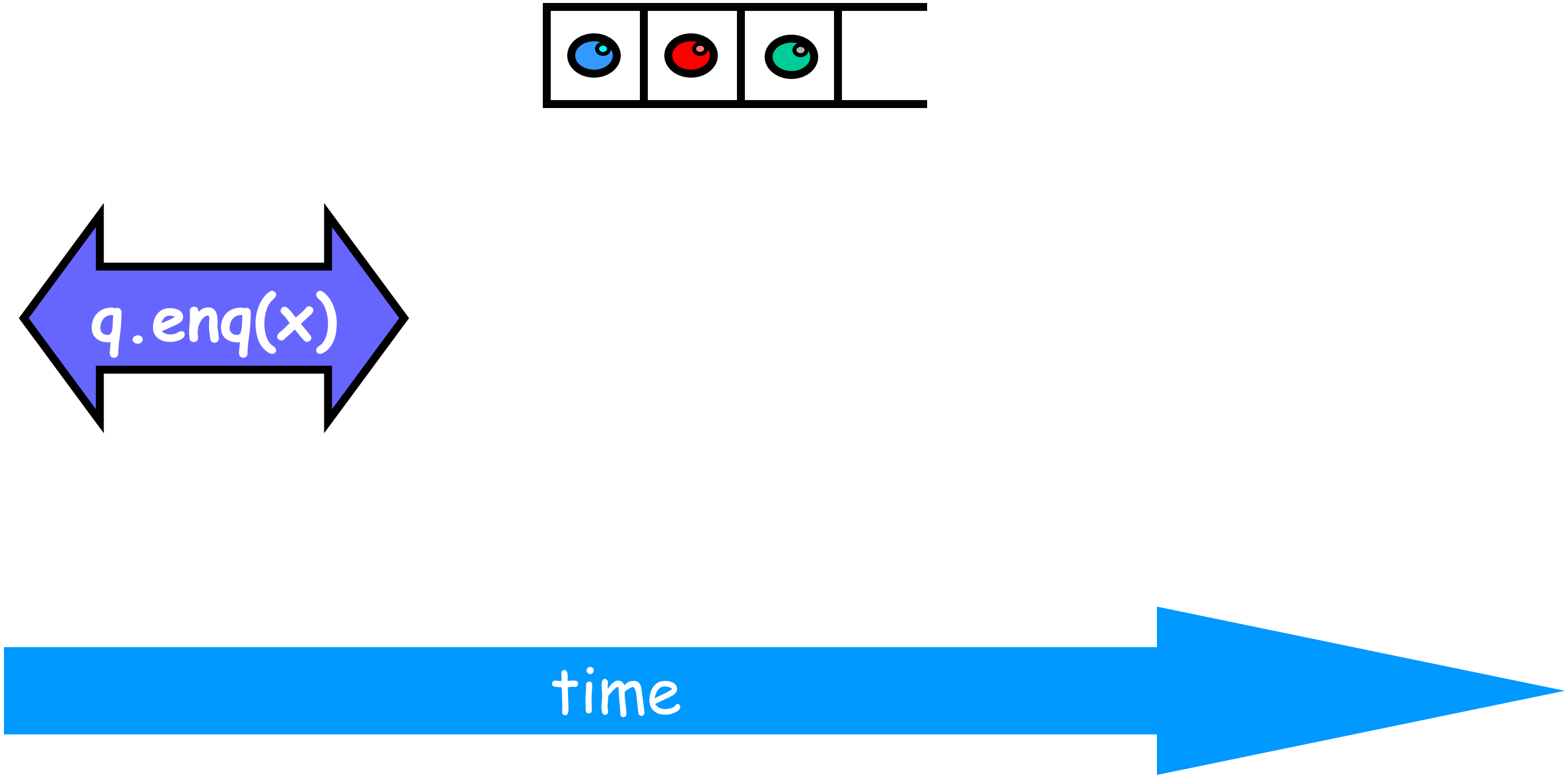
Alternative: Sequential Consistency

- 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

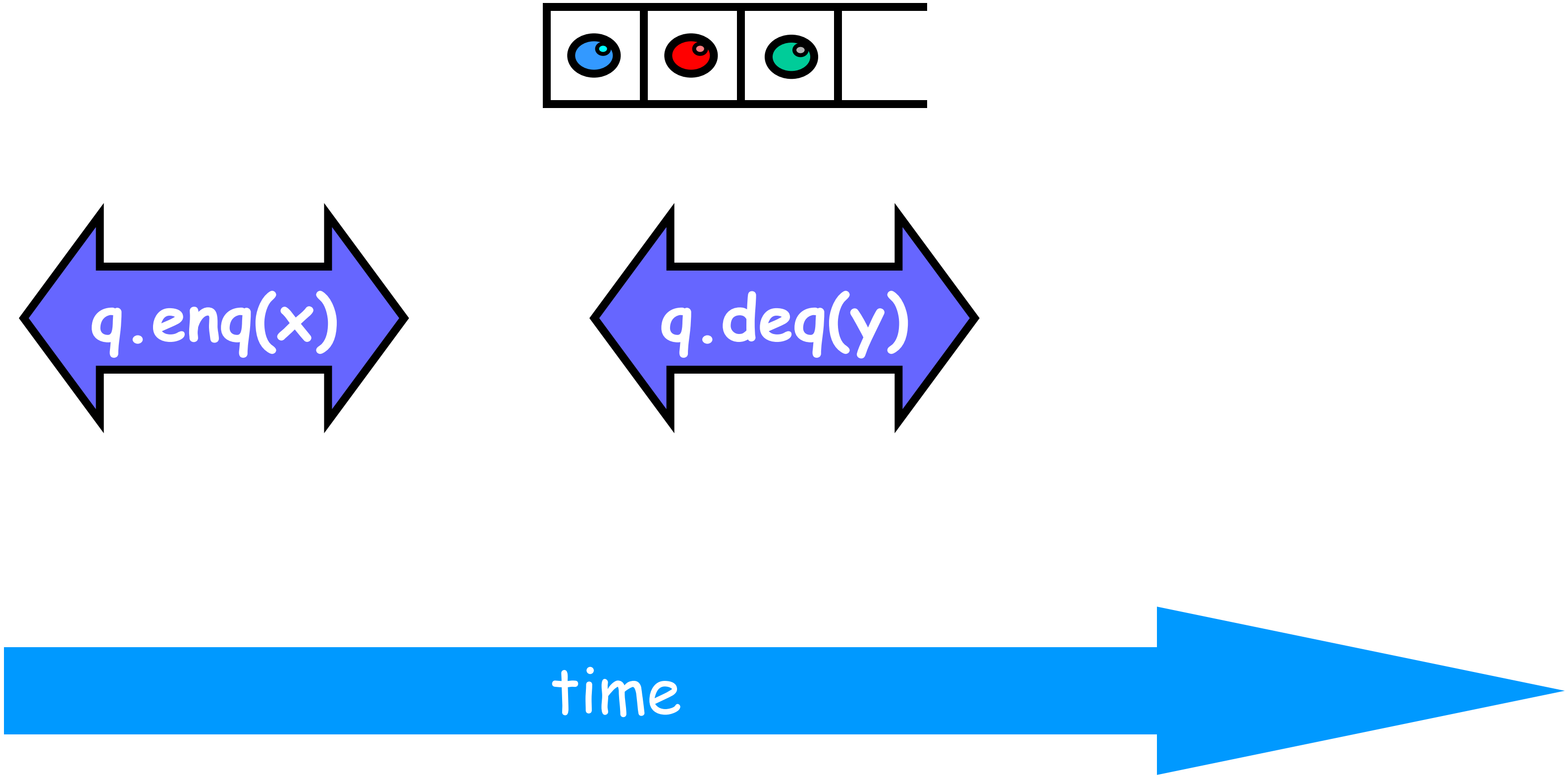
Example



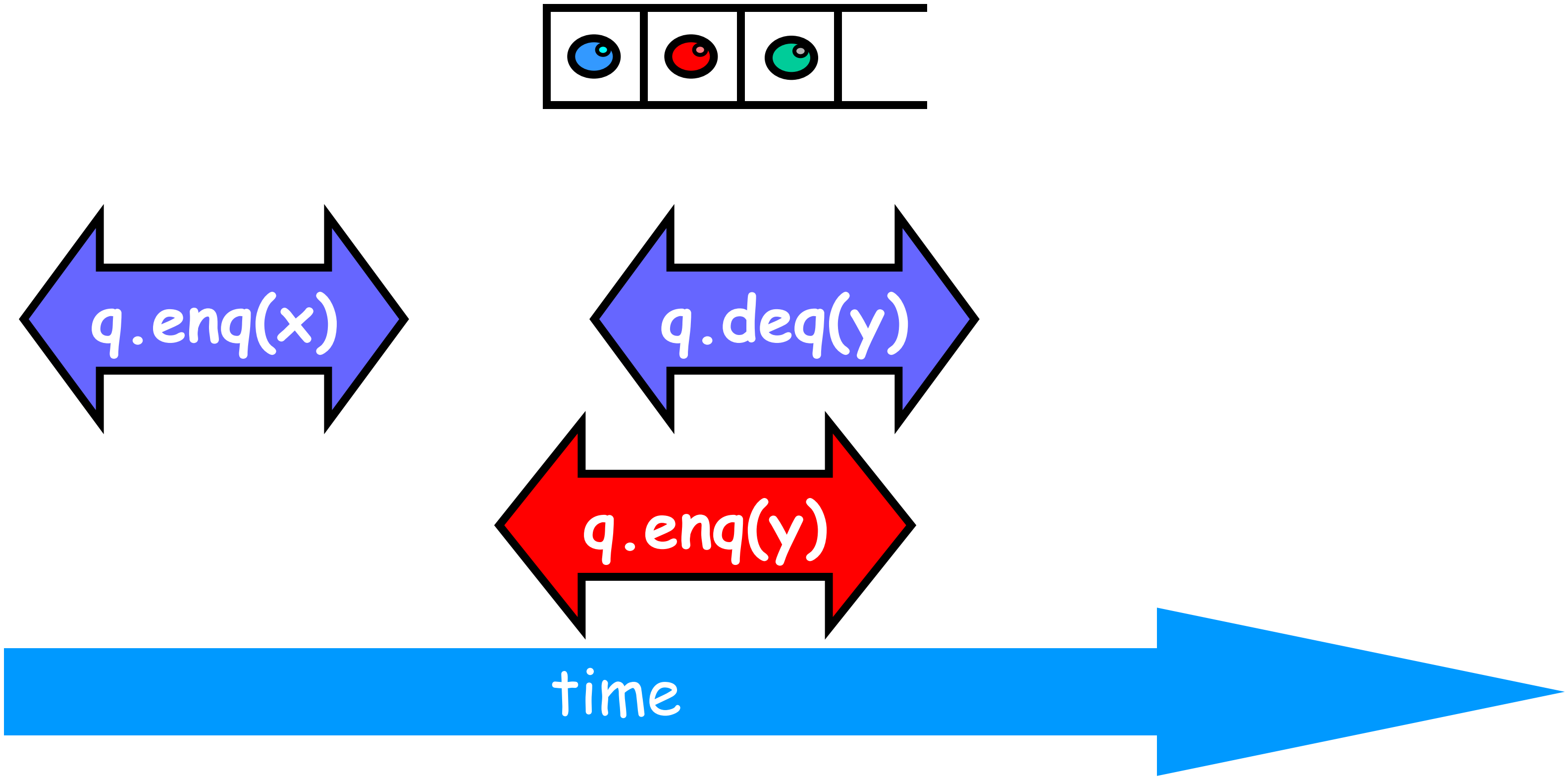
Example



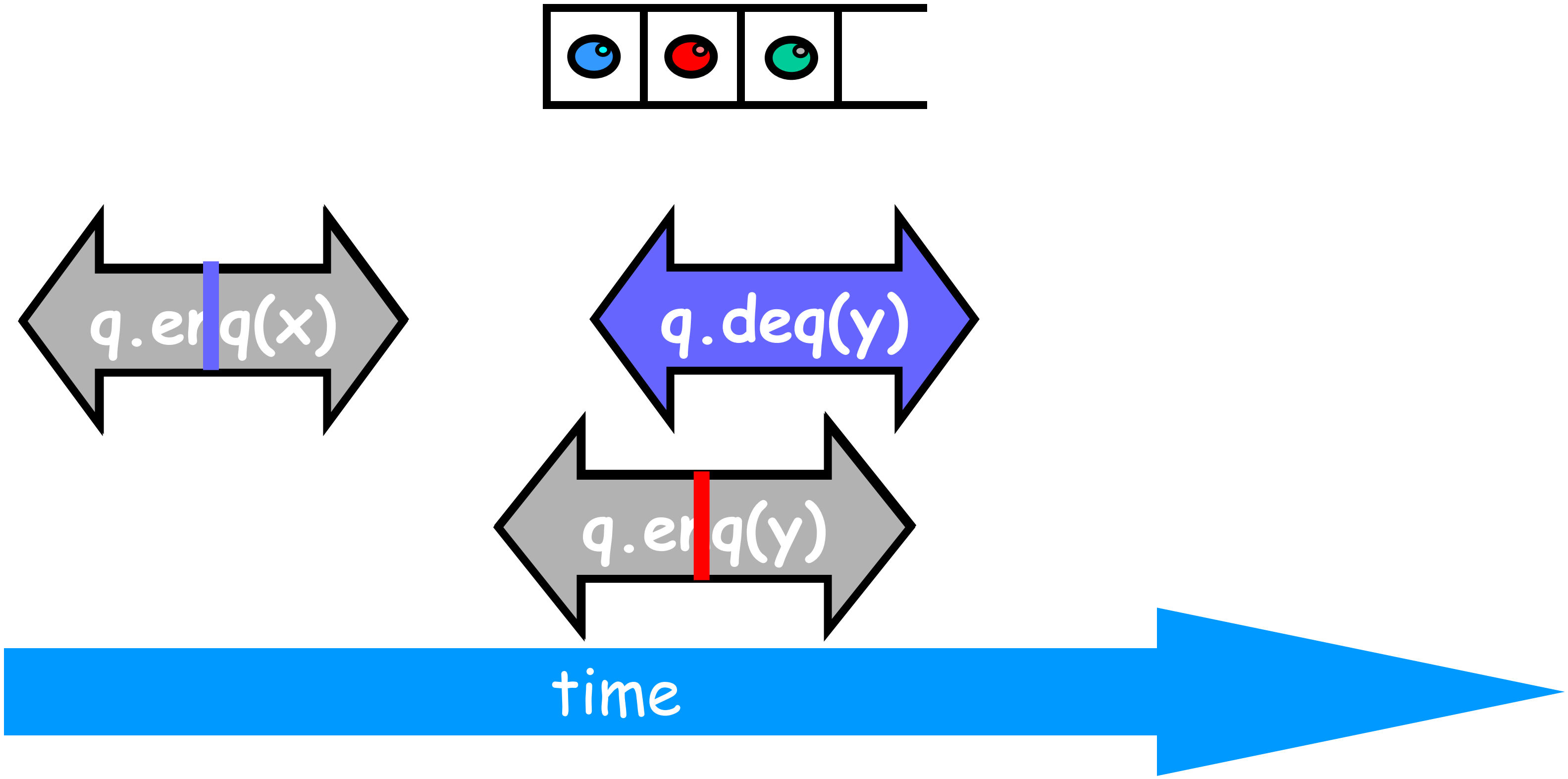
Example



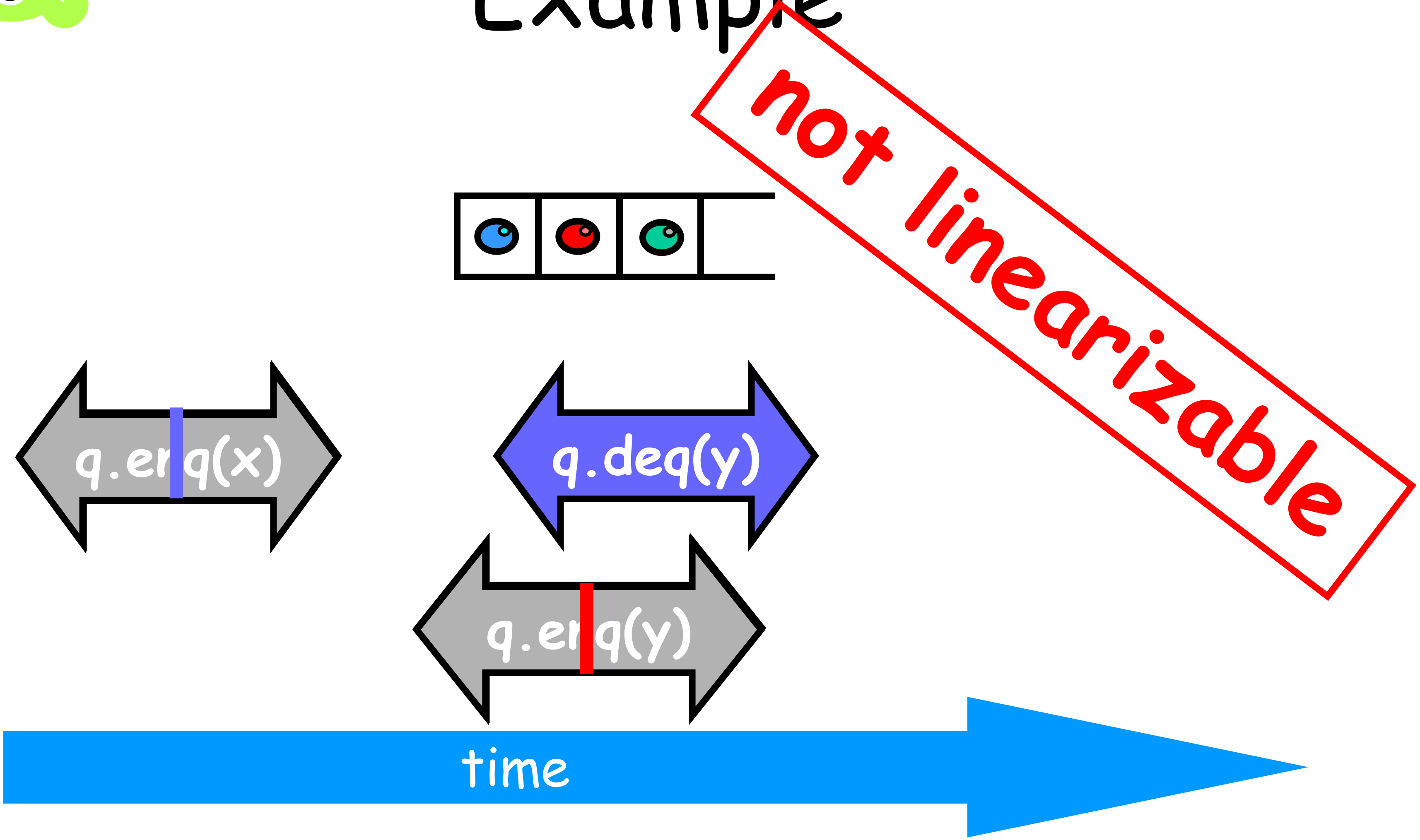
Example



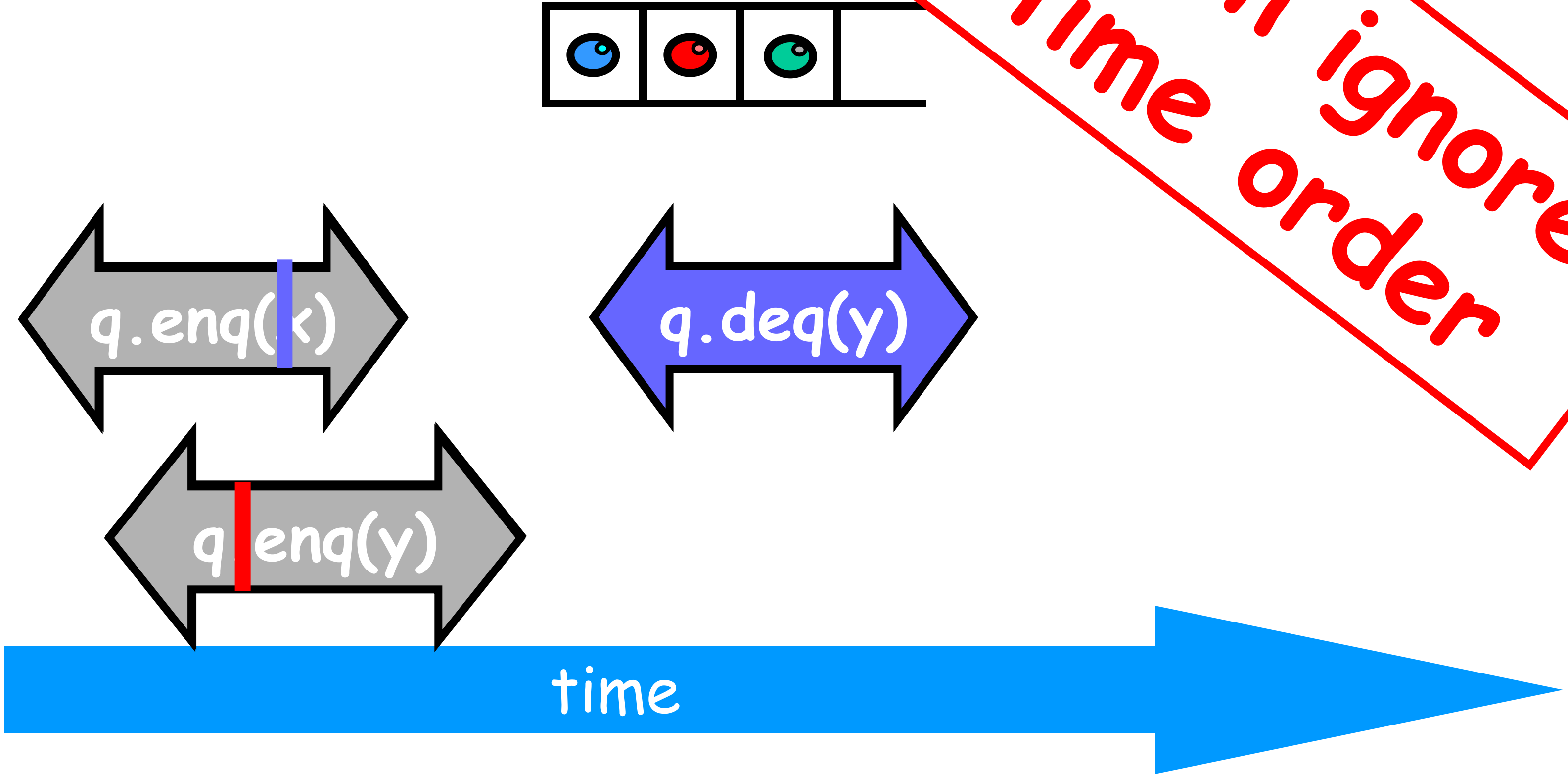
Example

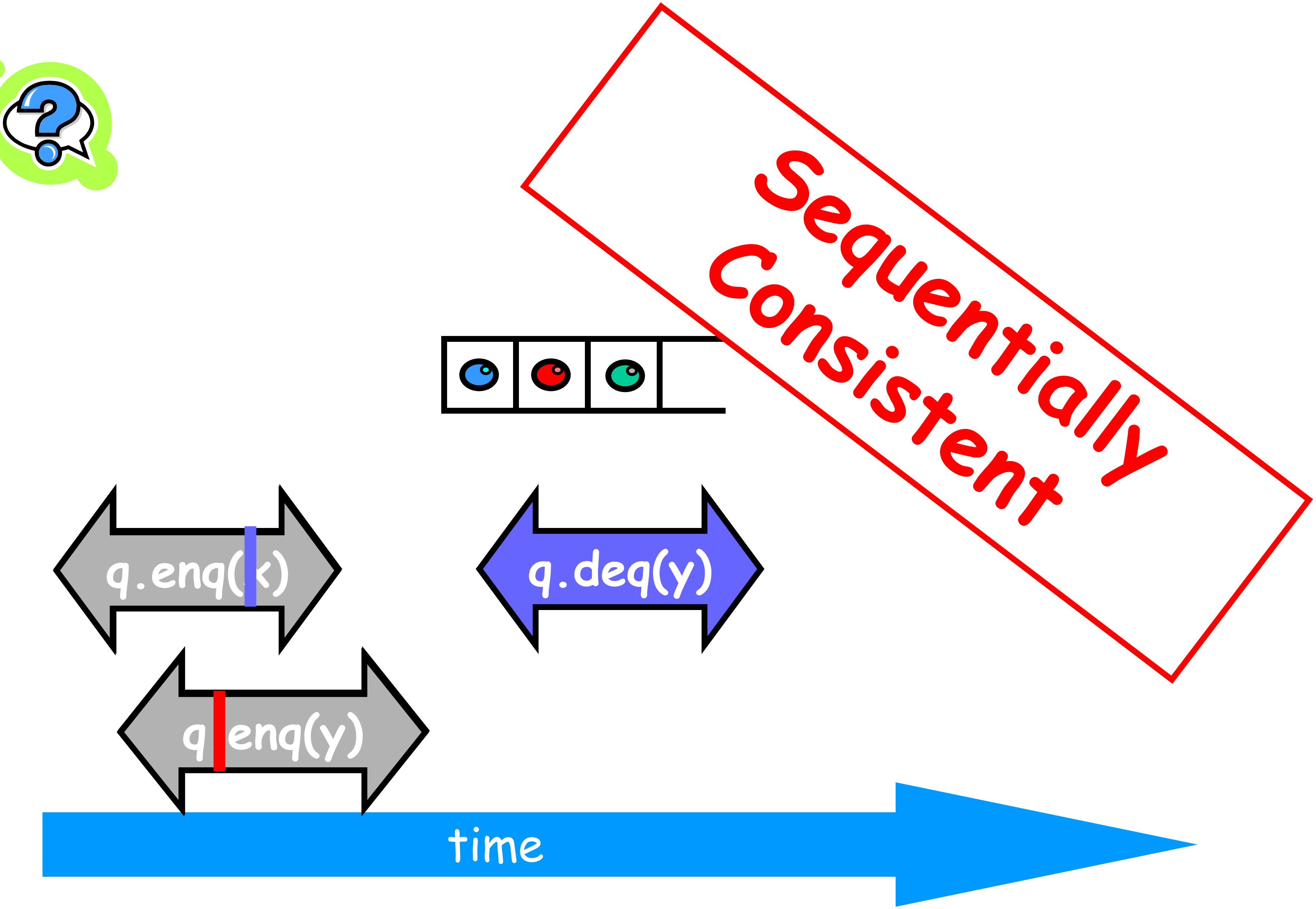


Example



If we can ignore real-time order



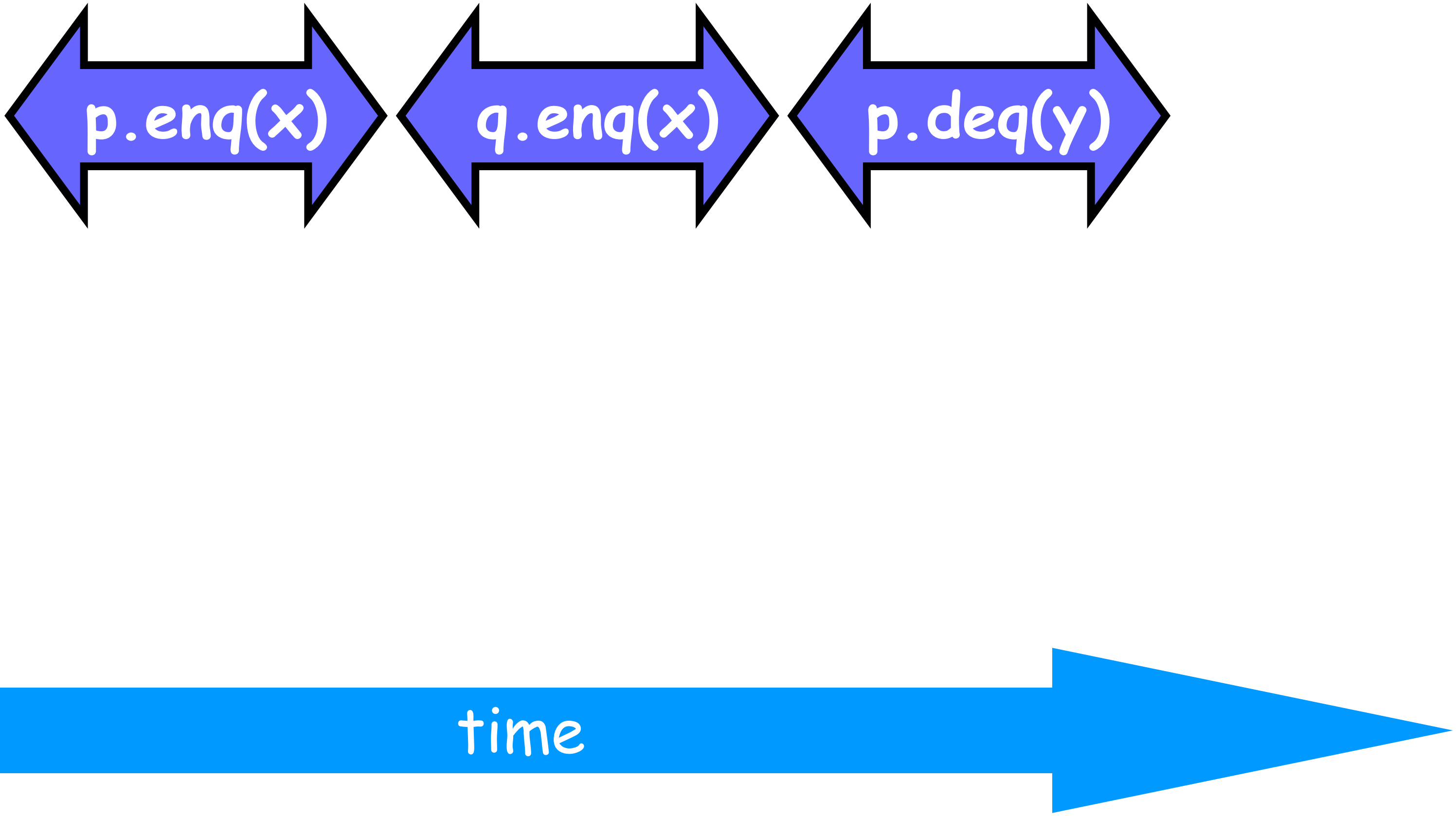

Sequentially Consistent
q.enq(x)
q.enq(y)
q.deq(y)
time

Theorem

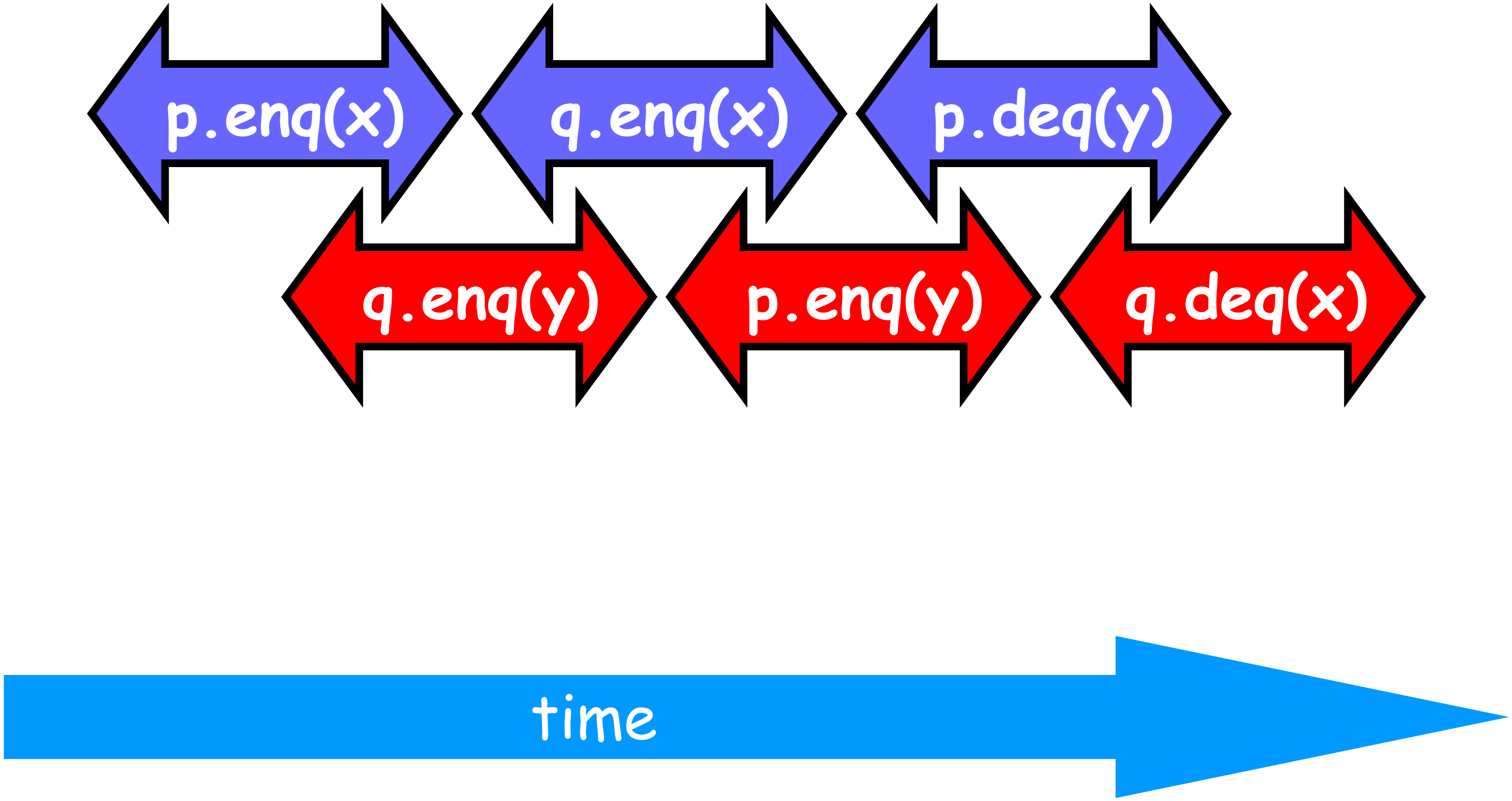
Sequential Consistency is not a local property

(and thus we lose compasability…)

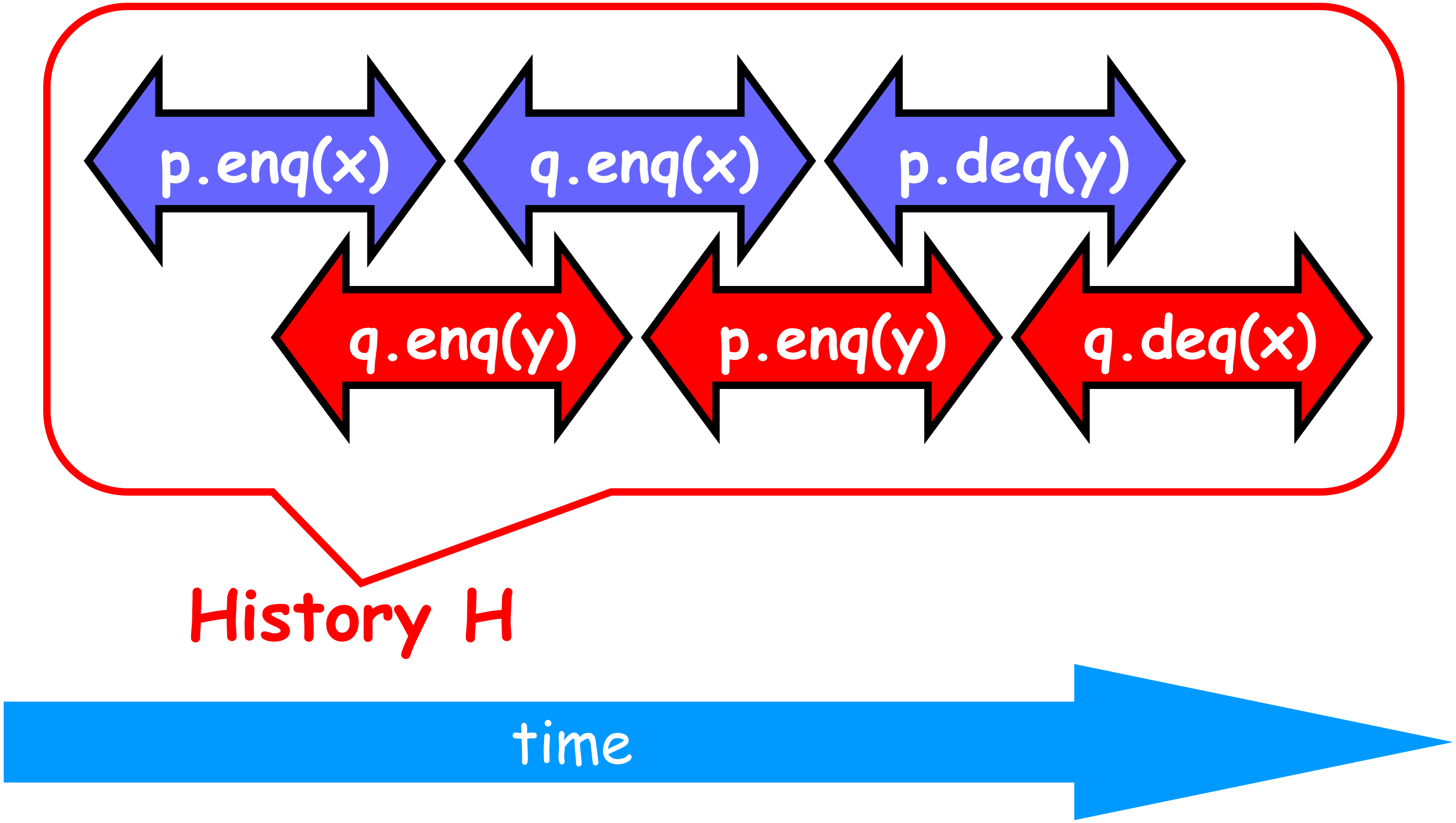
FIFO Queue Example



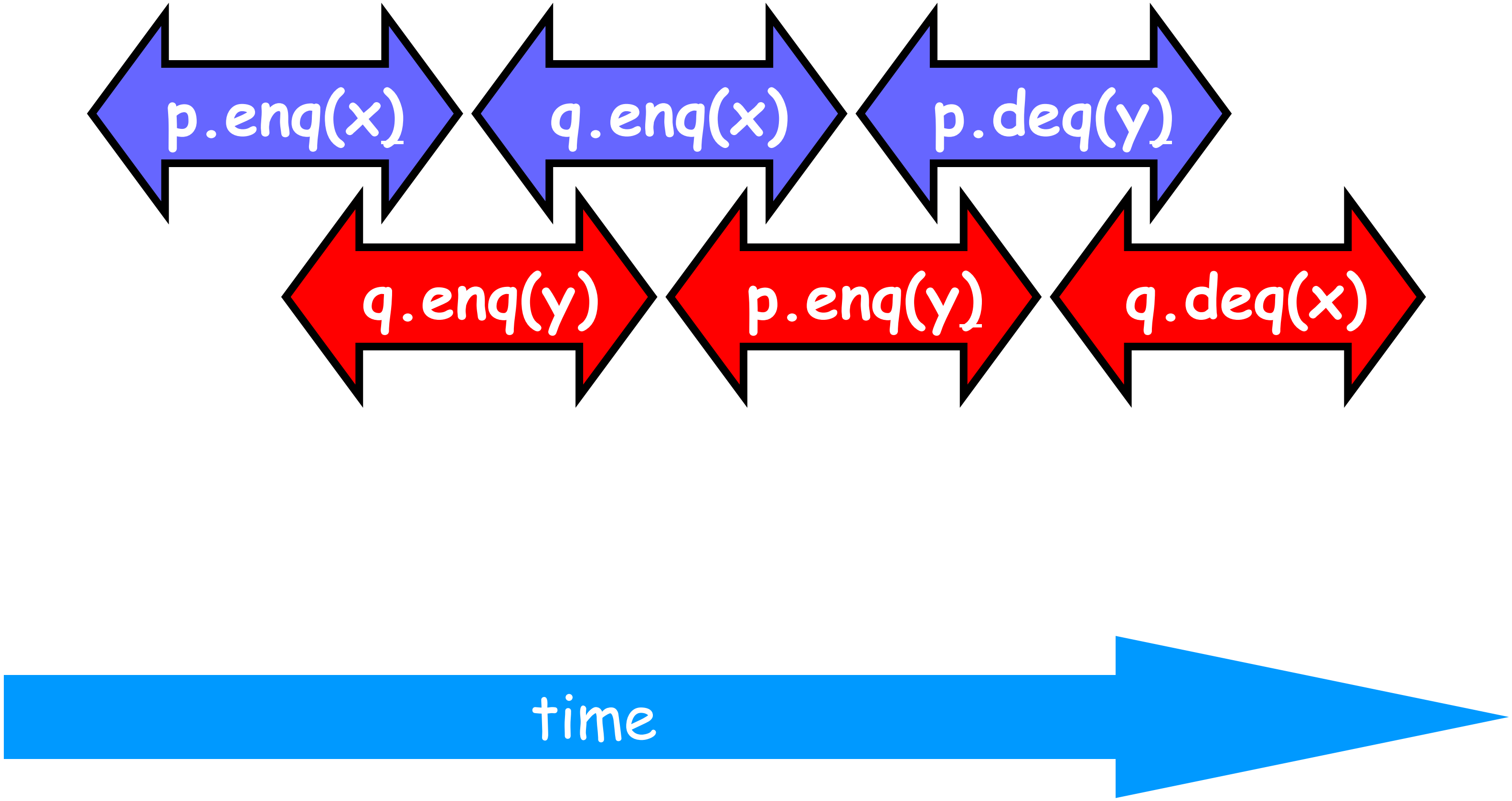
FIFO Queue Example



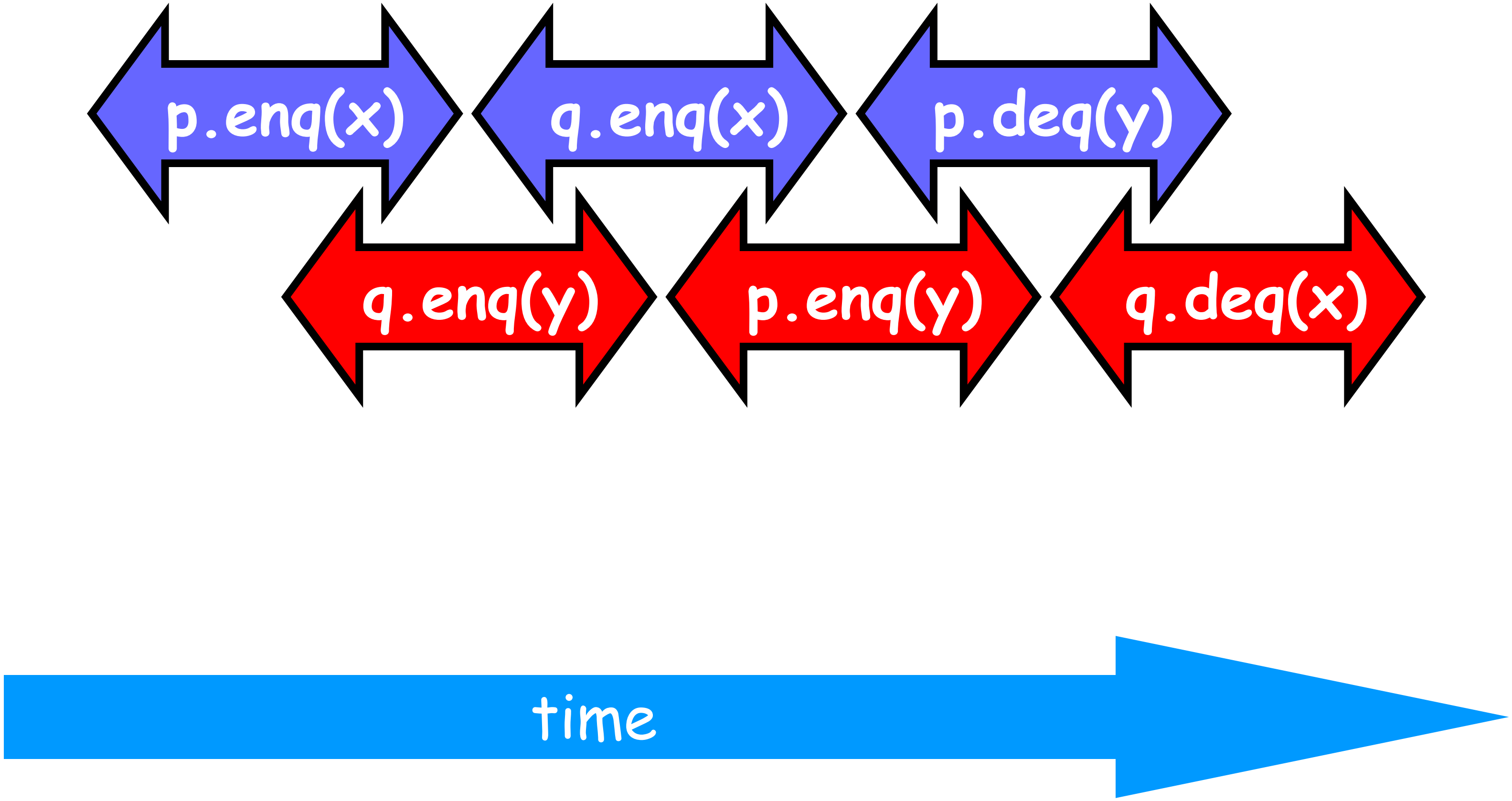
FIFO Queue Example



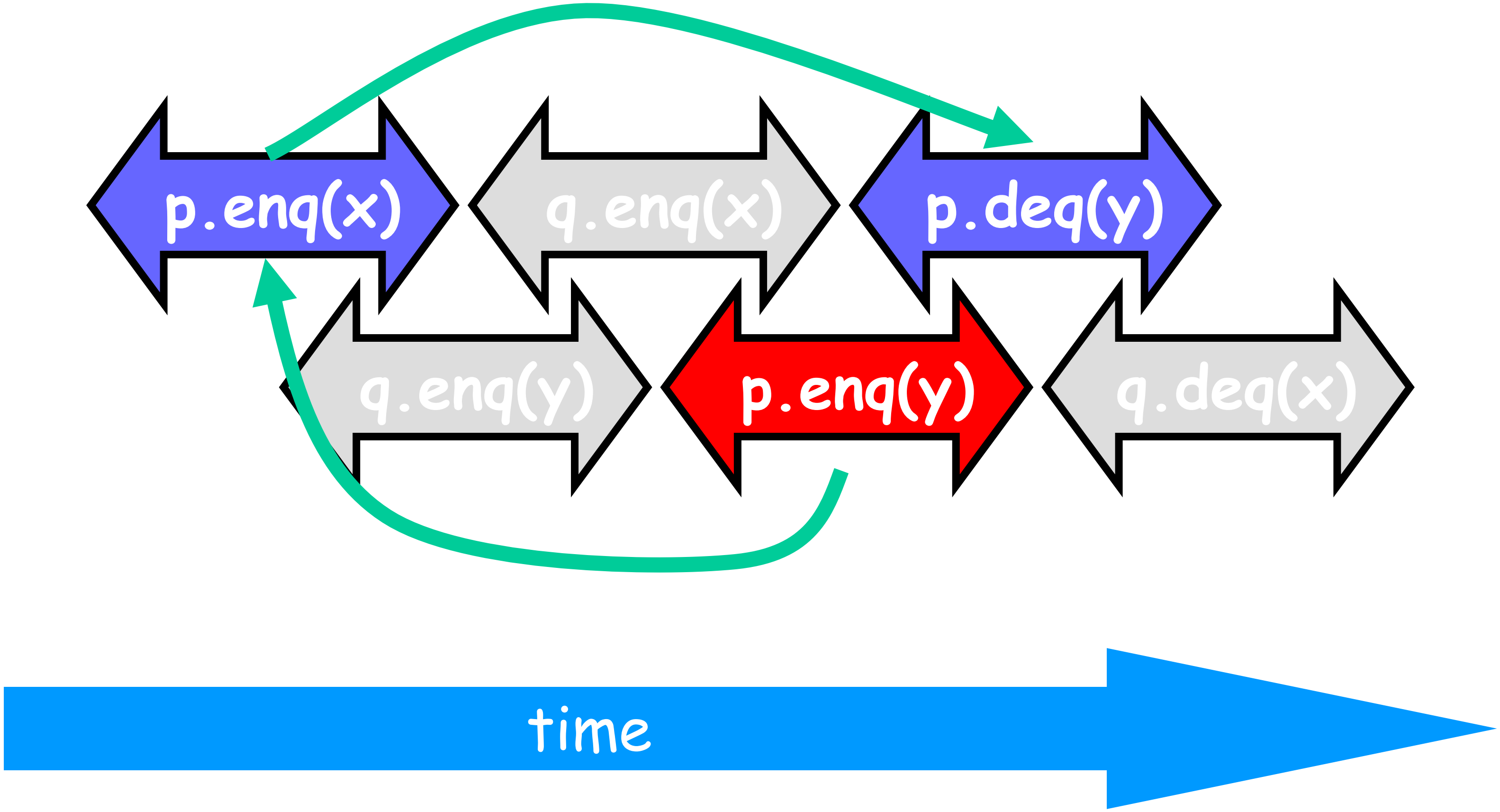
H|p Sequentially Consistent



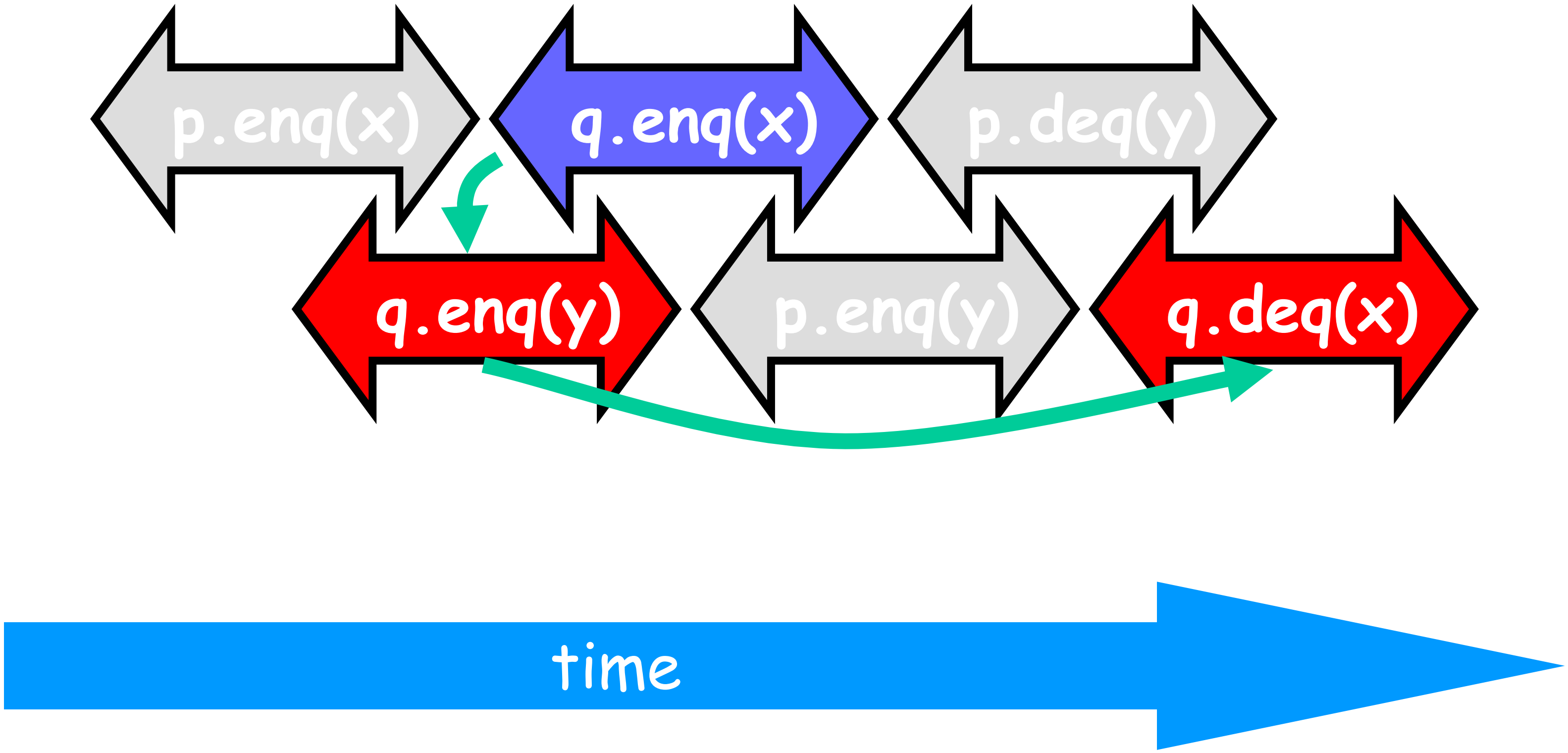
H|q Sequentially Consistent



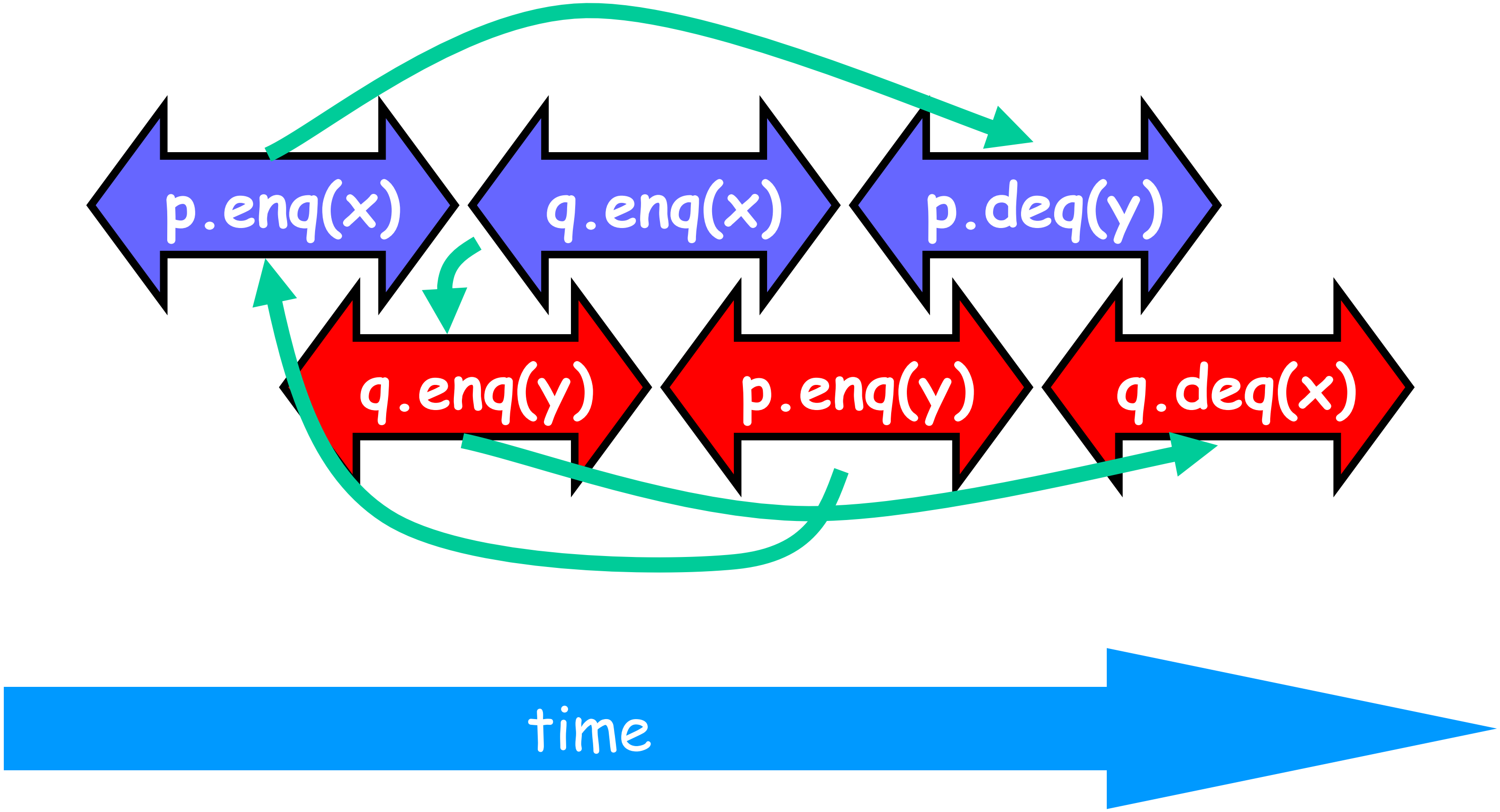
Ordering imposed by p



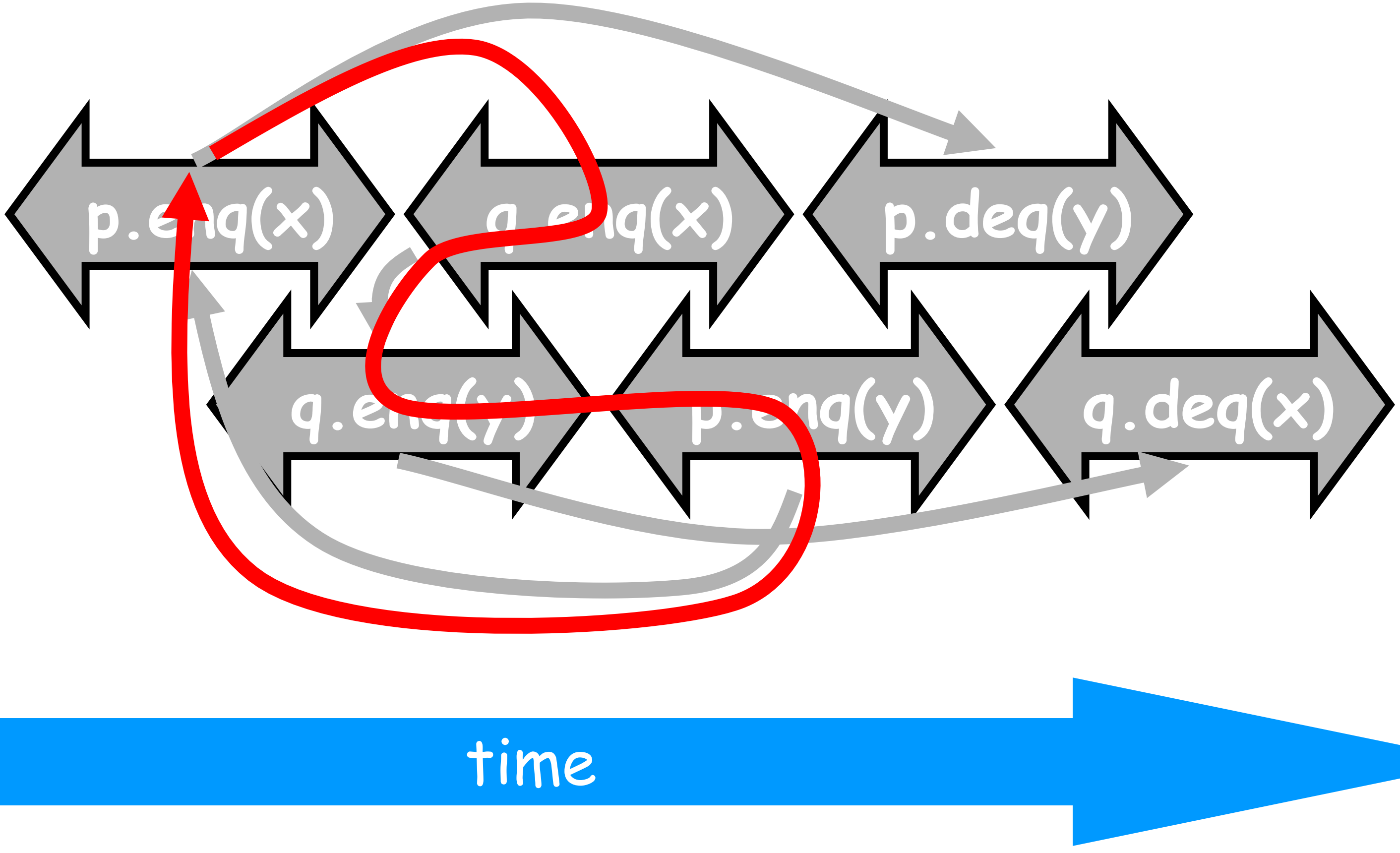
Ordering imposed by q



Ordering imposed by both



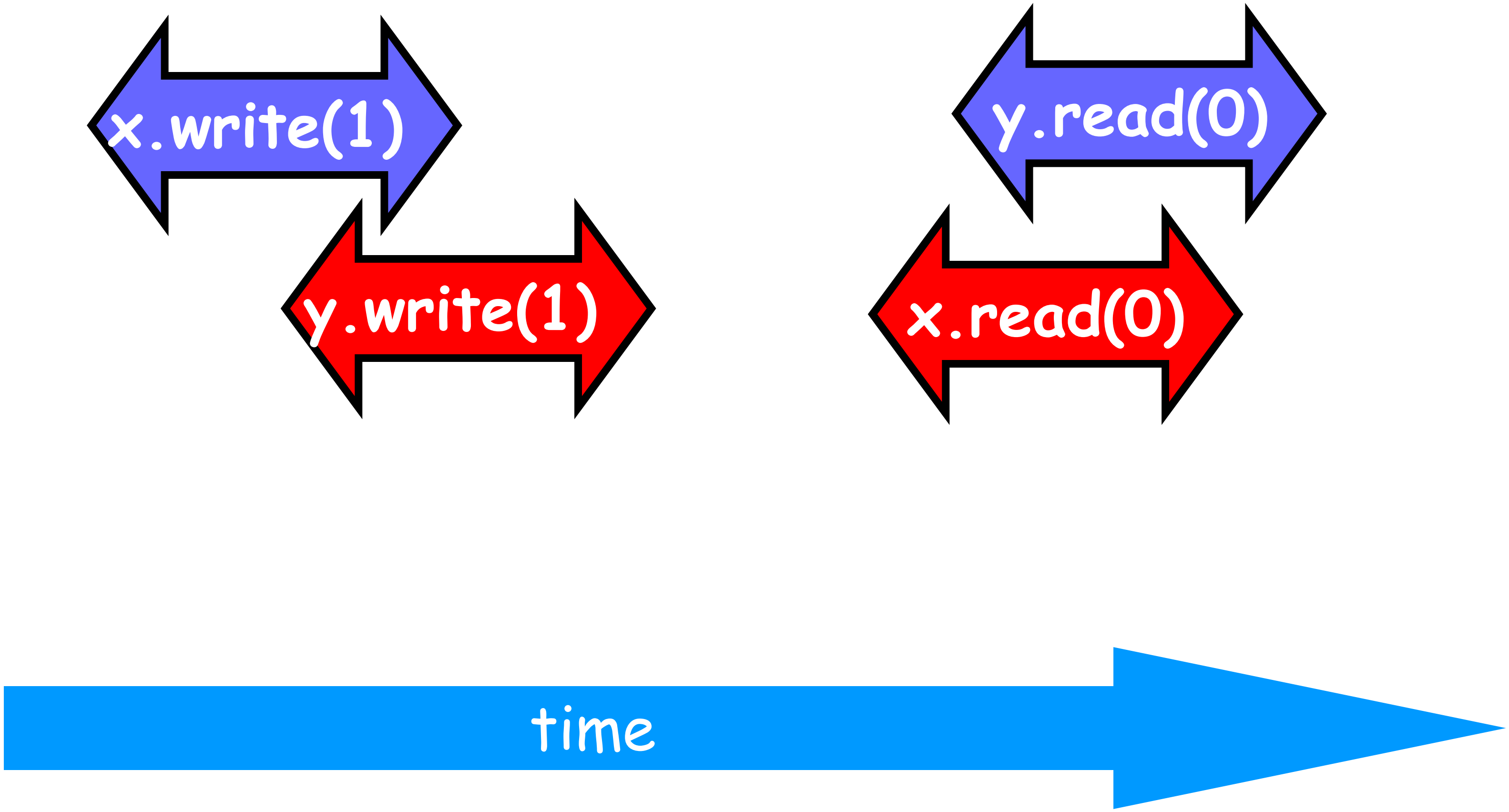
Combining orders



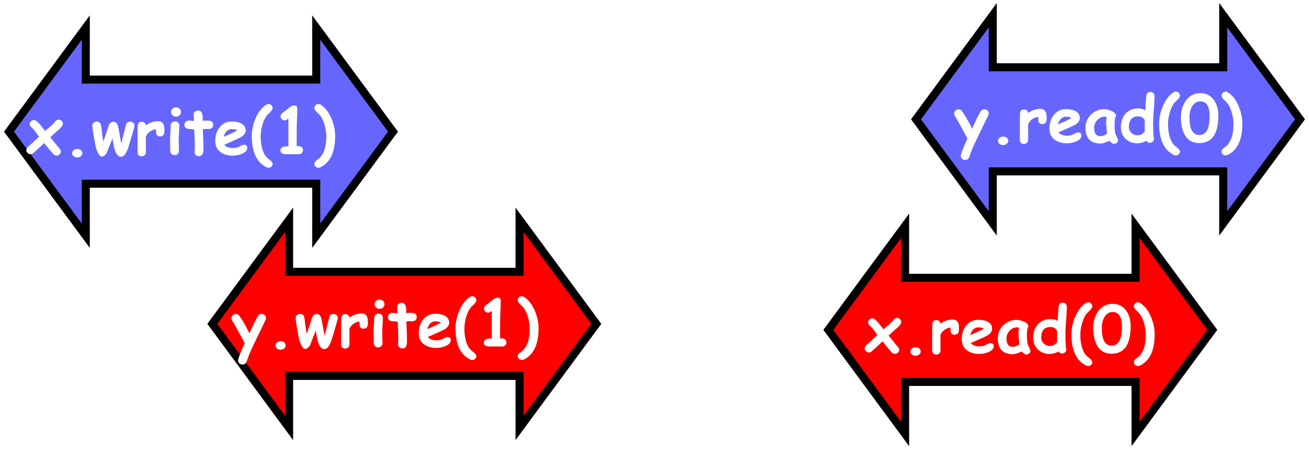
Fact

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

The Flag Example

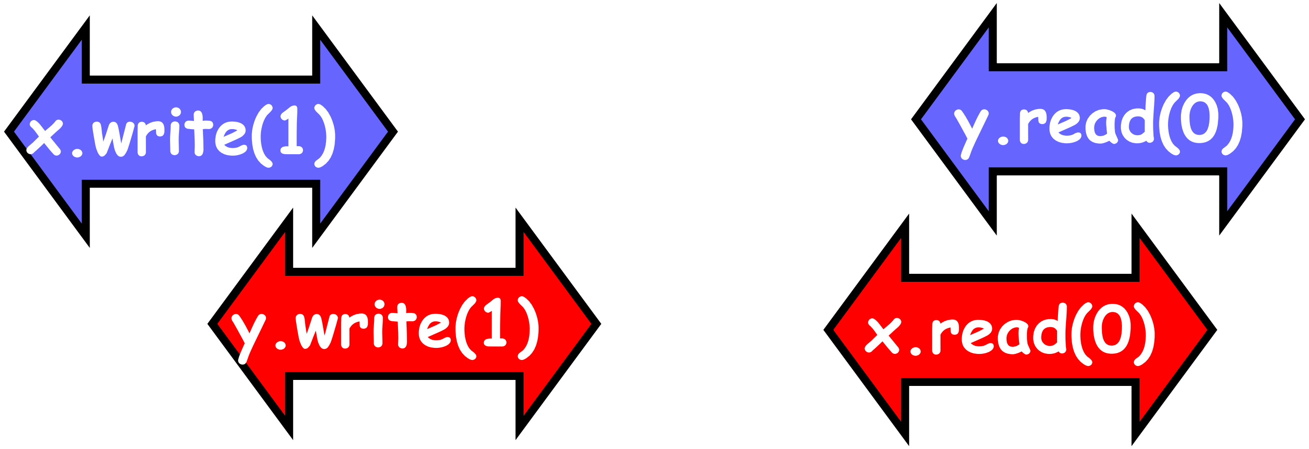


The Flag Example



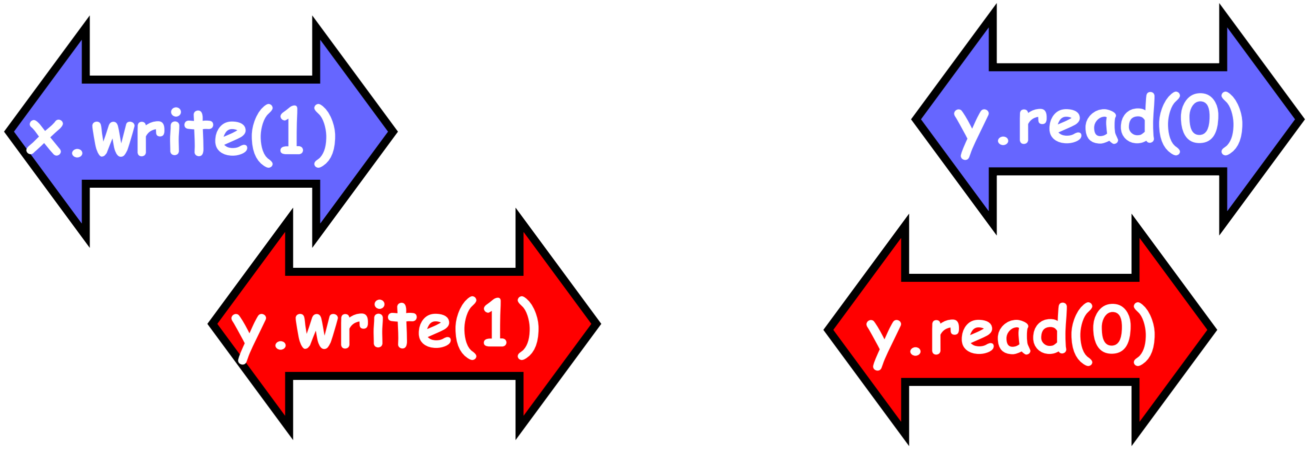
- Each thread's view is sequentially consistent
  - It went first

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 …

Opinion1: 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!

Opinion2: 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

Memory Operations

- To read a memory location,
  - load data into cache.
- To write a memory location
  - update cached copy,
  - 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?

Who knew you wanted to synchronize?

- 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

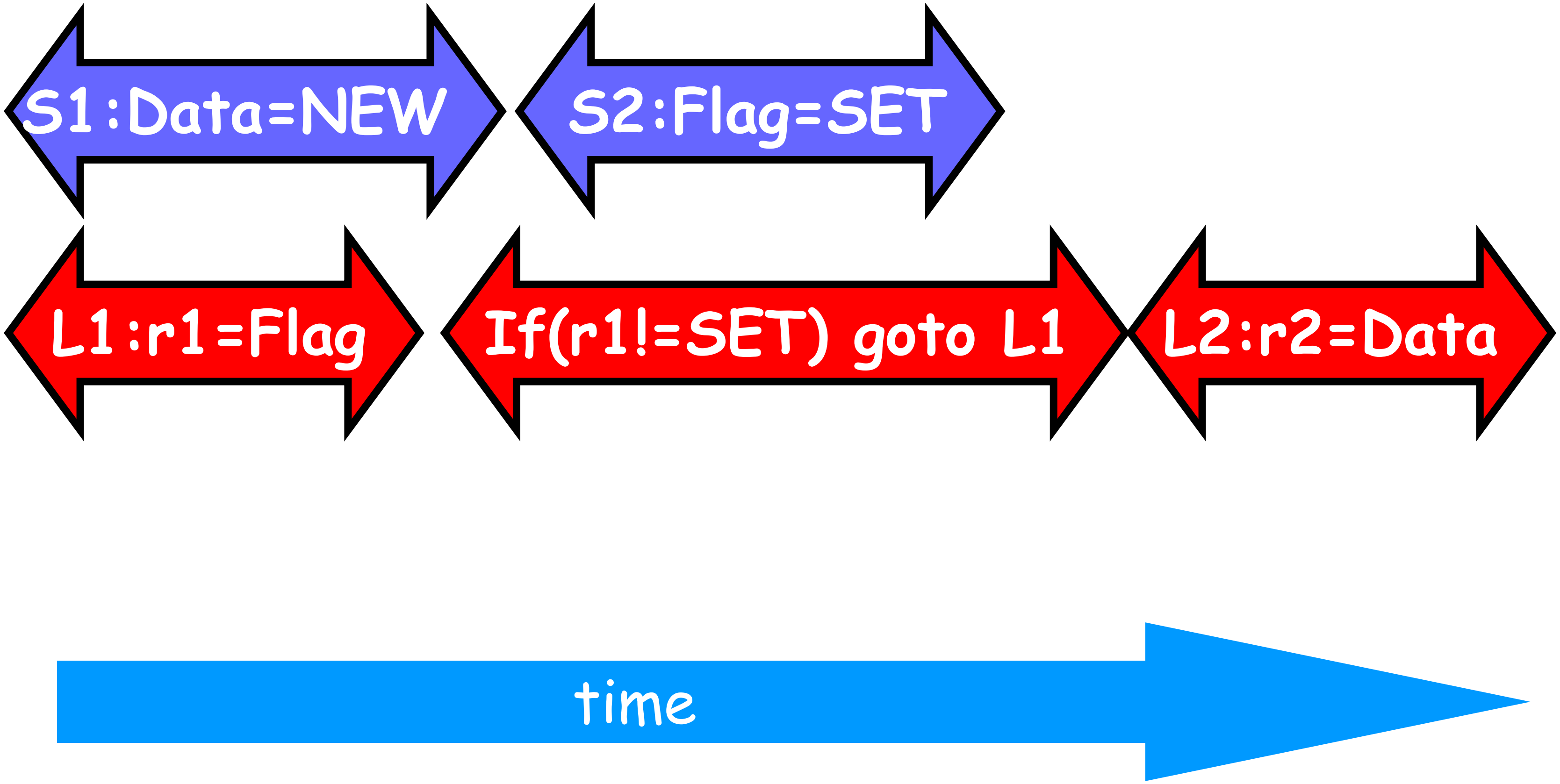
Summary

- Linearizability
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composablity
  - Good for high level objects

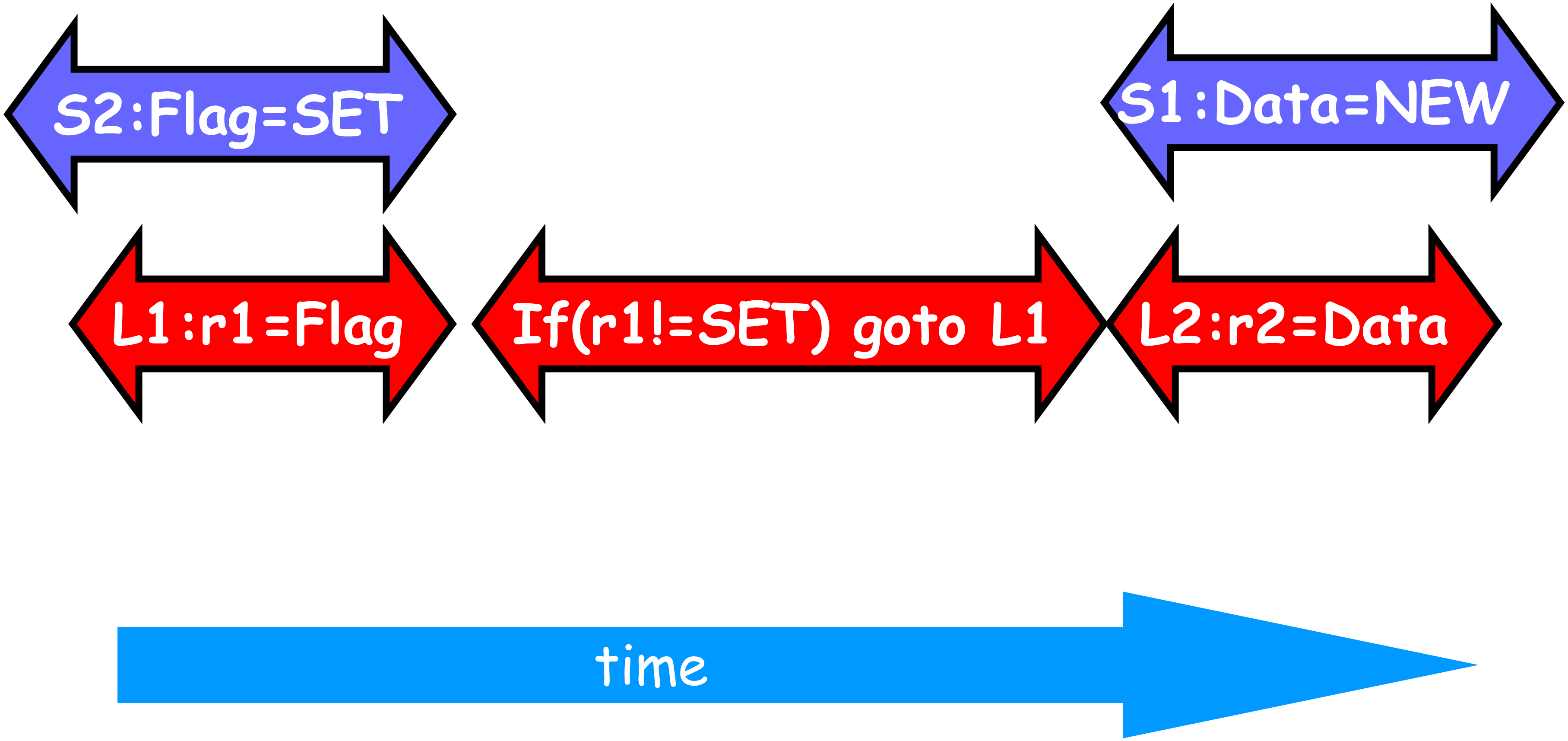
Summary

- 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

With multi core, What happens ?



S1 and S2 can be reordered



S2:Flag=SET

S1:Data=NEW

L1:r1=Flag

If(r1!=SET) goto L1

L2:r2=Data

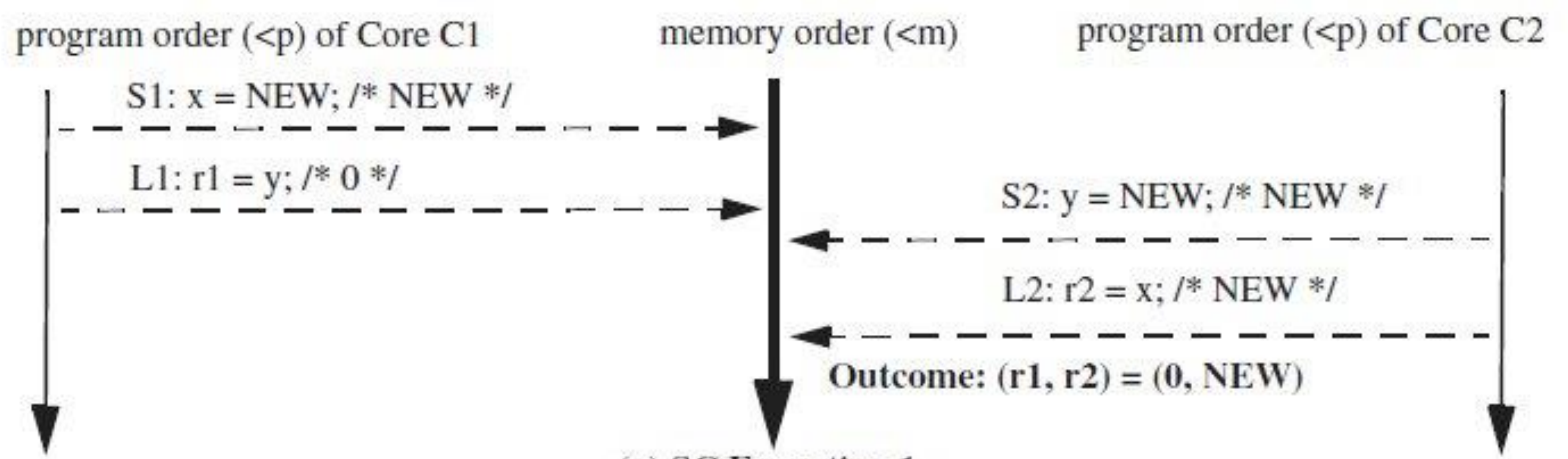
time

Dekker's algorithm

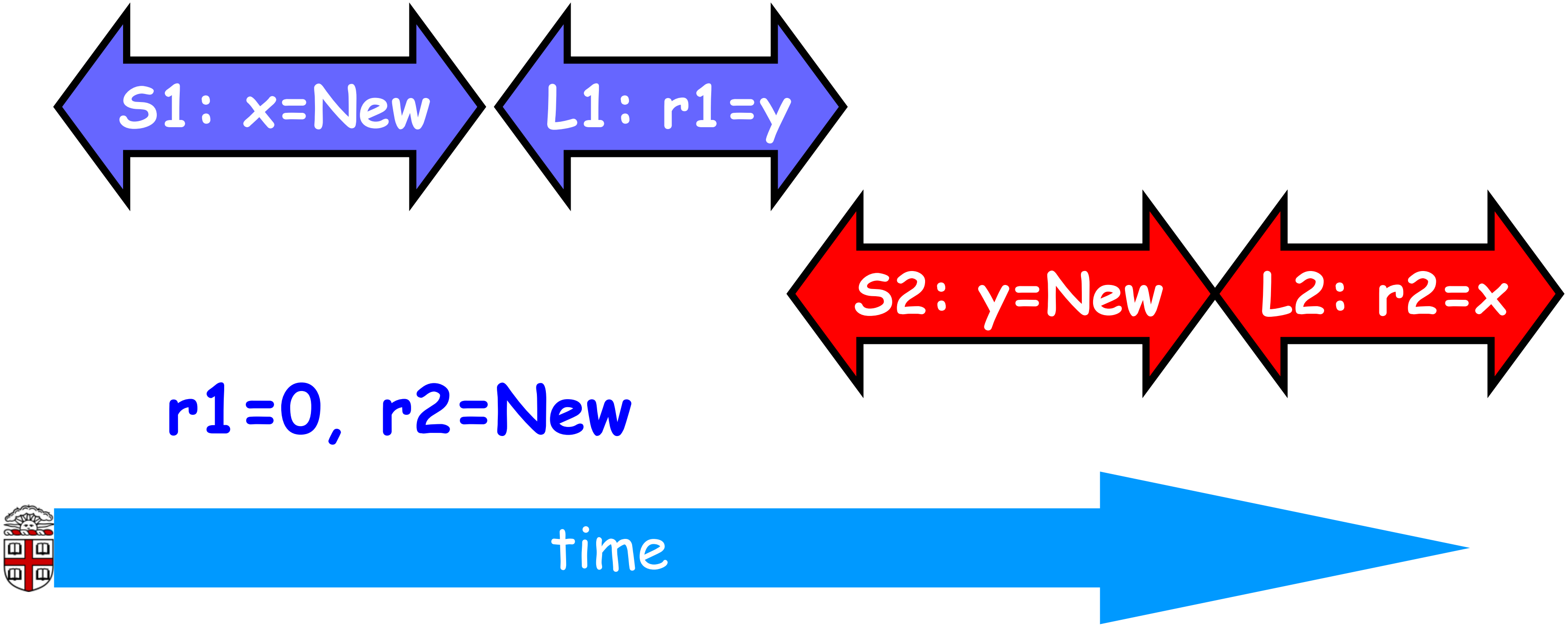
Initially, x=0 & y=0

```
Core C1       Core C2
S1: x=New;  S2: y=New;
L1: r1=y;     L2: r2=x;
```

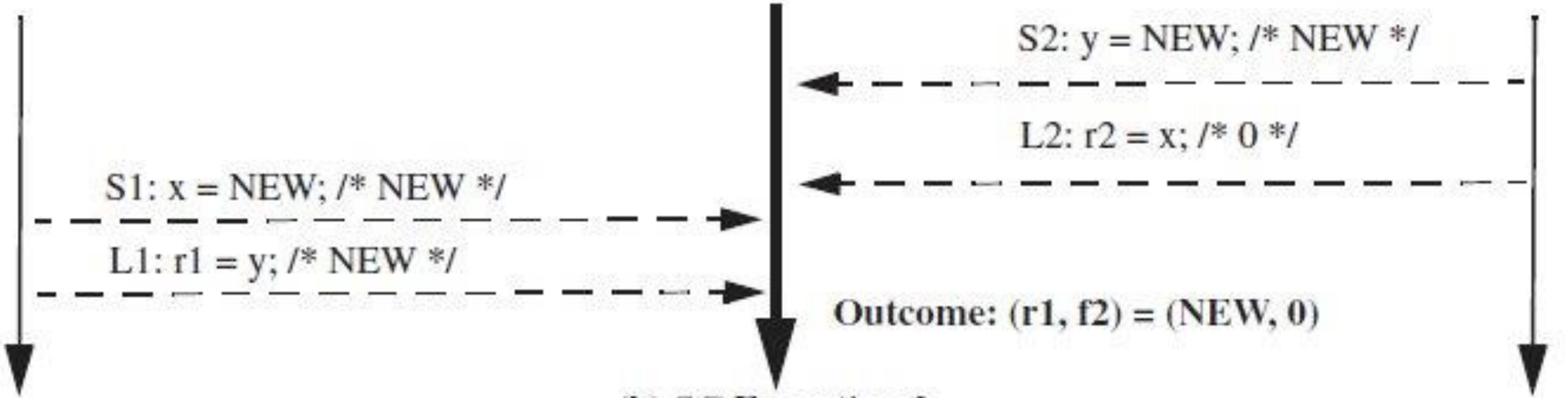
Can both r1 and r2 be set to 0 ?



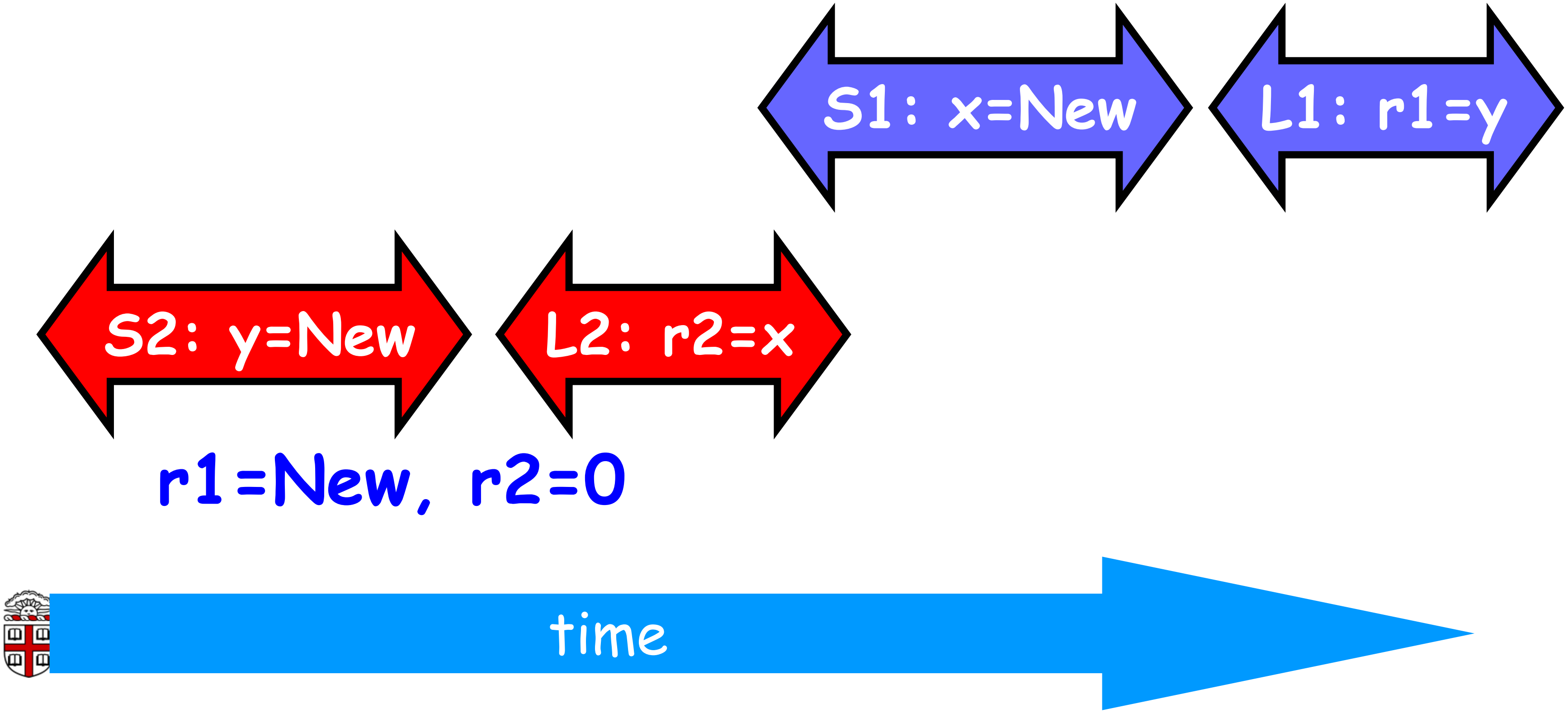
(a) SC Execution 1



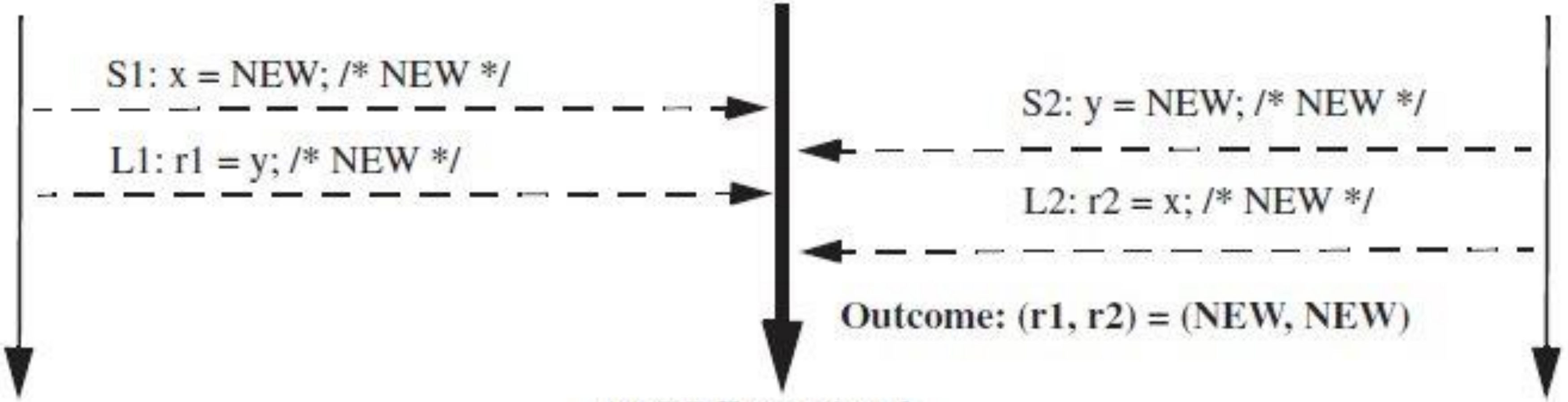
r1=0, r2=New



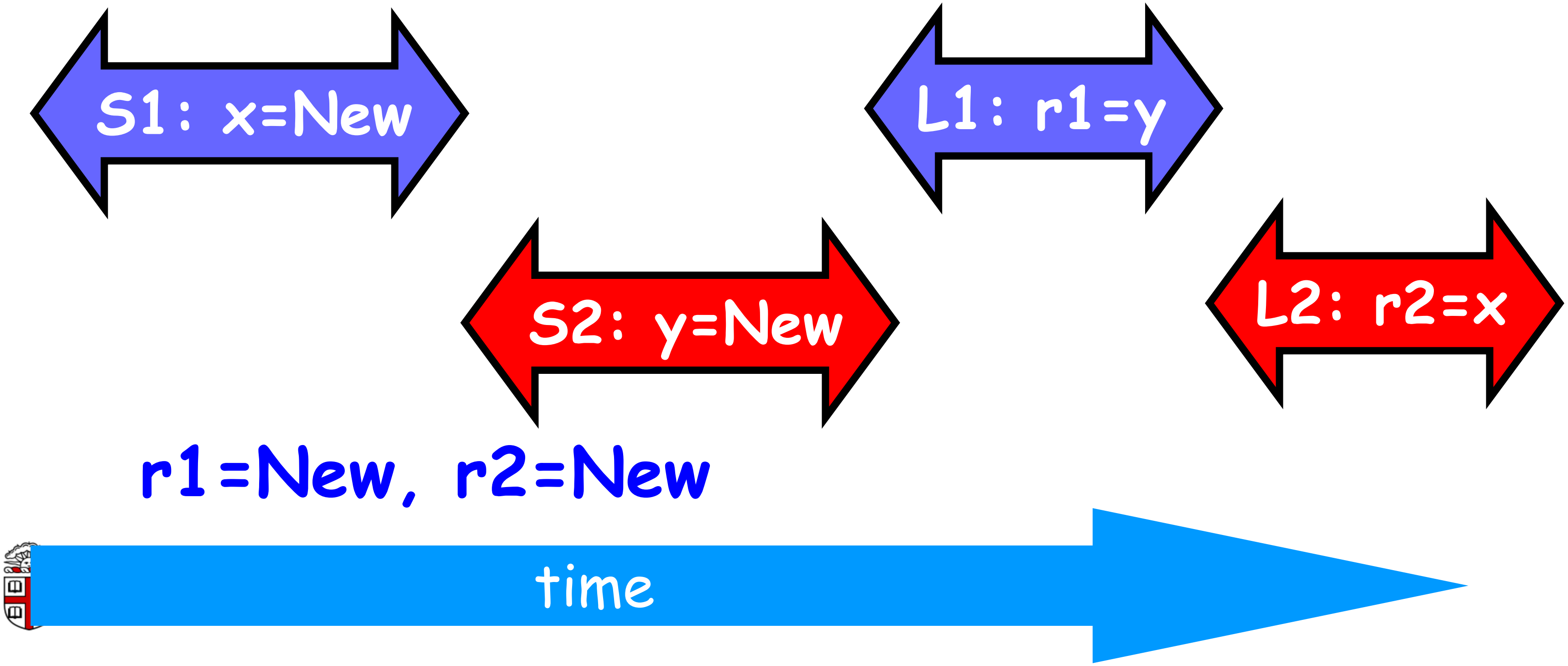
(b) SC Execution 2



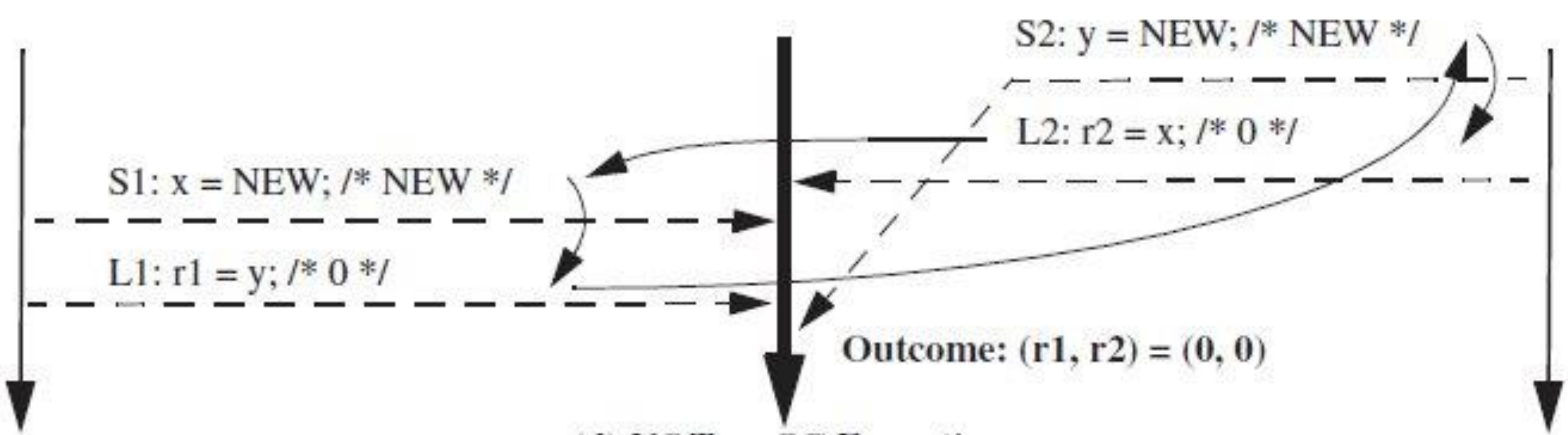
r1=New, r2=0



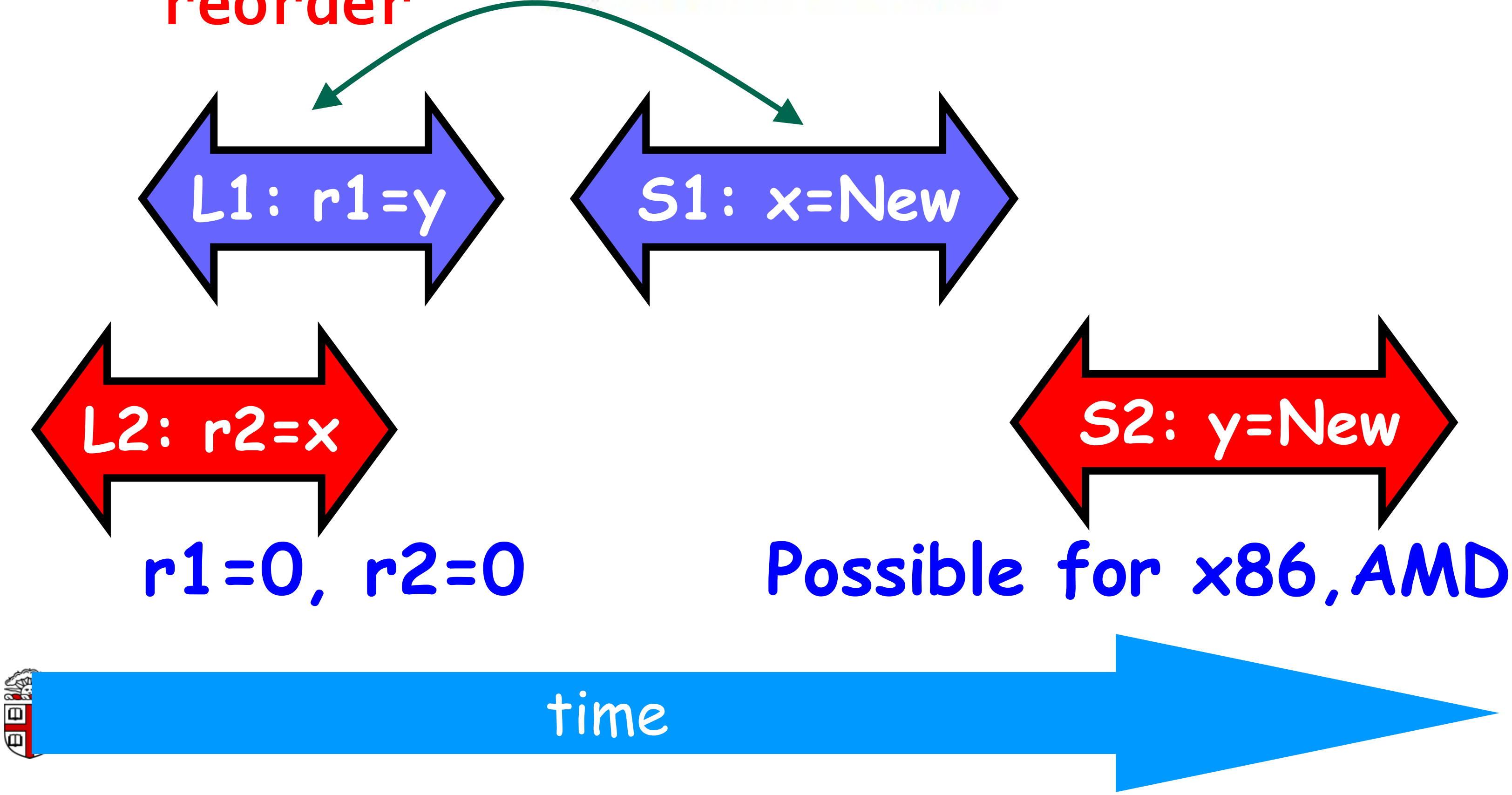
(c) SC Execution 3



r1=New, r2=New



(d) NOT an SC Execution



r1=0, r2=0

Possible for x86,AMD

An SC execution requires

- All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses
- Every load gets its value from the last store before it to the same address

SC ordering Rules

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

X denotes an enforced ordering

Total Store Order

- SPARC
- X86

- Remnants of the write buffer
- Write takes longer
  - It sometimes does, too
  - Needs to get the write permission

Total Store Order

- Load -> Load
- Load -> Store
- Store -> Store
- Store -> Load (omitted for TSO)

- Omitting the 4th constraint allows each core to use a write buffer

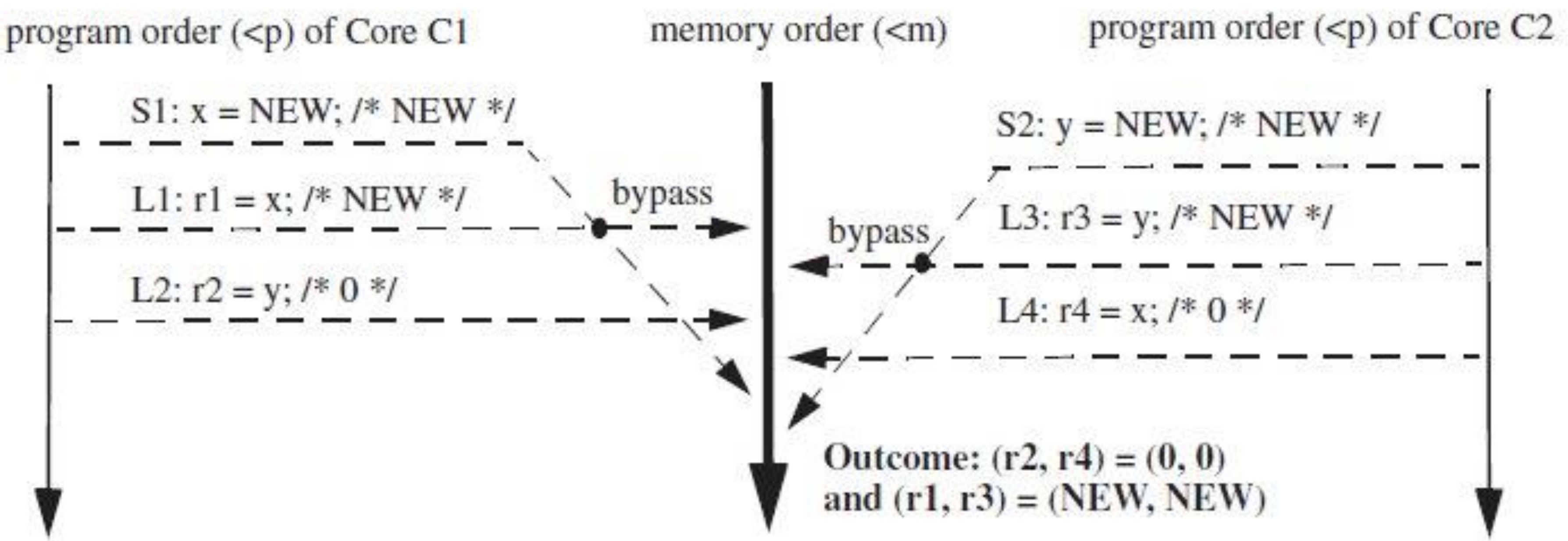
TSO behavior

Initially, x=0 & y=0

Core C1	Core C2
S1: x=New;	S2: y=New;
L1: r1=x;	L3: r3=y;
L2: r2=y;	L4: r4=x;

If r2=0 & r4=0, can r1 or r3 be set to 0 ?

Bypassing



r1=New & r3=New !!!

To make TSO sequentially consistent

- Use FENCE
- FENCE ensures the memory operations before the FENCE get placed before the memory operations after the FENCE on the core

TSO ordering Rules

Operation 1	Operation 2			
	Load	Store	RMW	FENCE
Load	X	X	X	X
Store	B	X	X	X
RMW	X	X	X	X
FENCE	X	X	X	X

B denotes bypassing if on the same address