DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

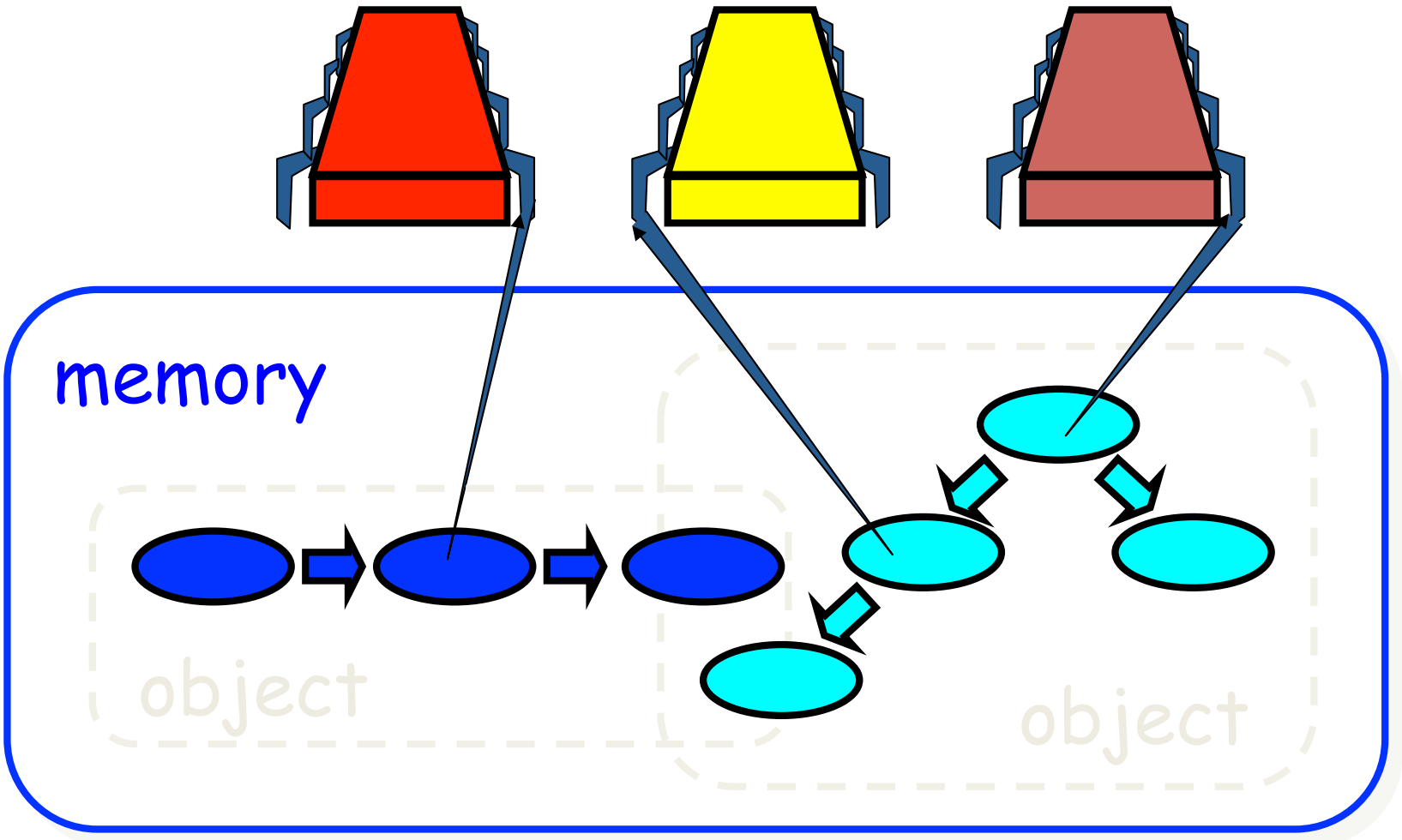
Lecture 2

Concurrent Objects

Mads Dam

Autumn/Winter 2011

Concurrent Computaton



Objects

Objects have ***state***

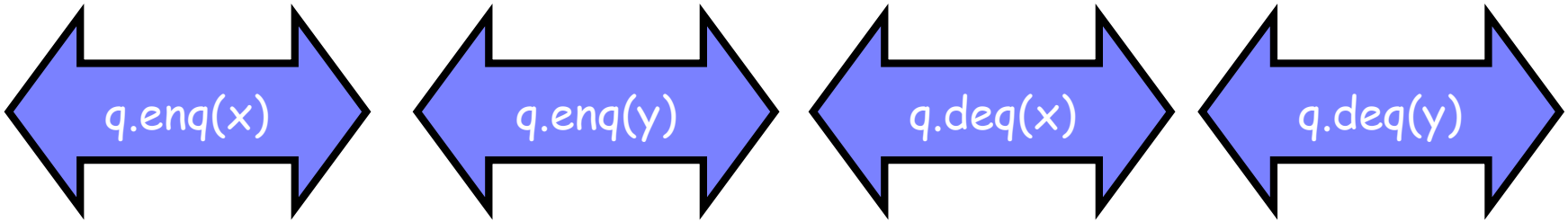
- Usually given by a set of ***fields***

Object have ***methods***

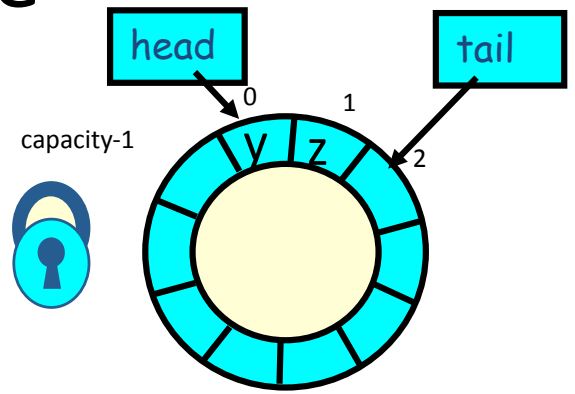
- Only way to manipulate state

Example, a FIFO queue:

- Fields: Sequence of items
- Methods: enq and deq

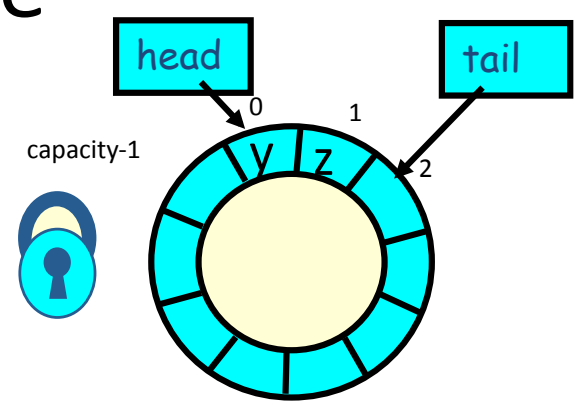


A Concurrent FIFO Queue




```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

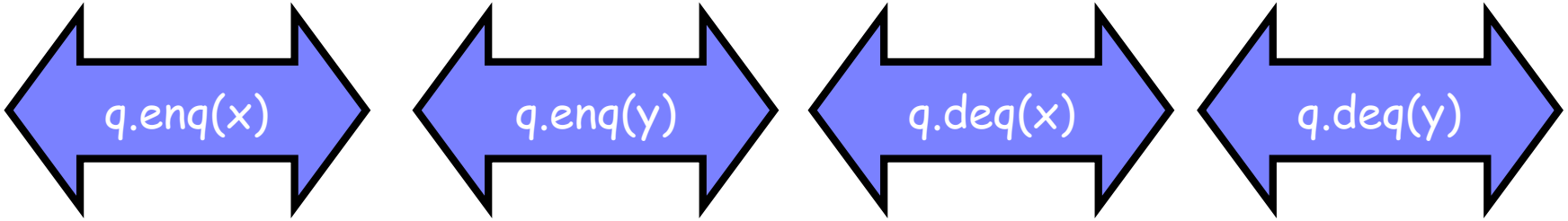
A Concurrent FIFO Queue




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

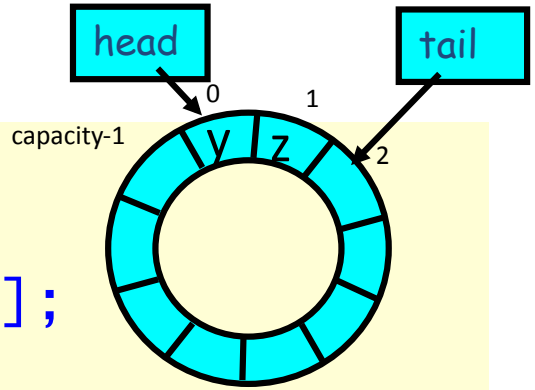
Is The Queue Correct?

- Calls are effectively sequentialized
- Behaviour is familiar:



- (In fact the calls overlap slightly but we ignore that for now)
- Now let's try the same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
    - The producer thread
  - The other deq only
    - The consumer thread

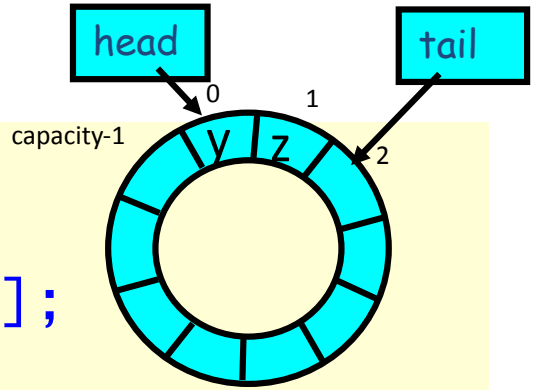
Wait-free 2-Thread Queue




```
public class WaitFreeQueue {
  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

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

Wait-free 2-Thread Queue



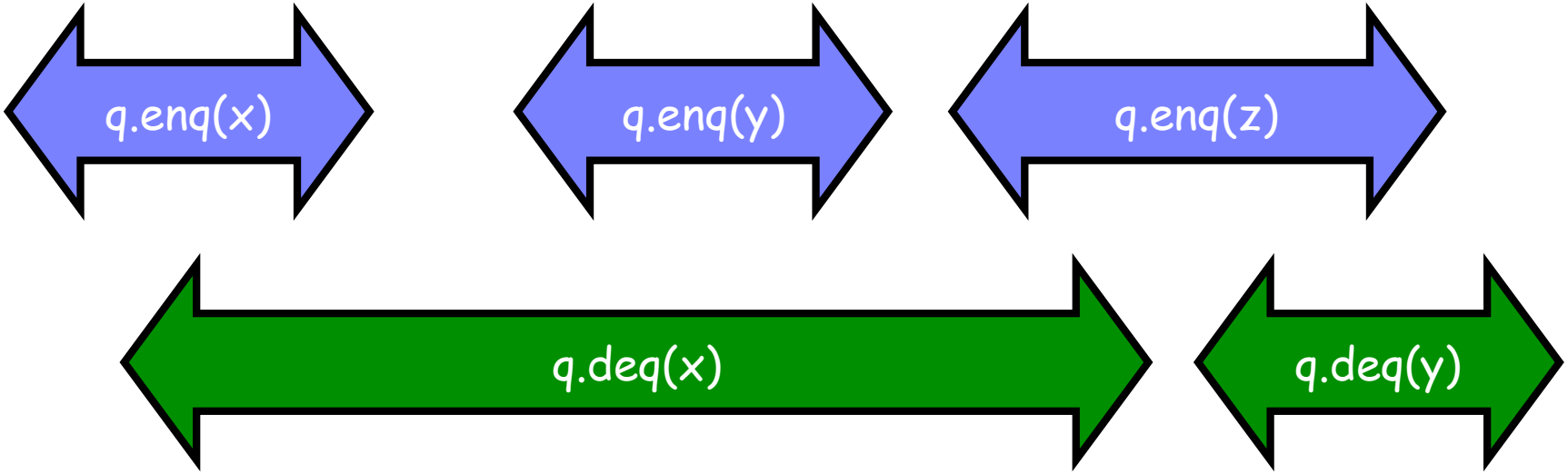

```
public class WaitFreeQueue {
  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

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

Ooops: No locks!

Is the Wait-Free Queue Correct?

The behaviour of the queue may now look something like this:



Do we want this behaviour? Or not? Specify!

Need to be able to answer this question

Need to know when an implementation is correct

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, cont'd)
  - the object will be in some other state
  - when the method returns.

Pre and PostConditions for Deq

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

- 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
  - Elsewhere it does not matter
- 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 ?

- Method state potentially matters everywhere
  - Method calls may interfere because of shared variable accesses
  - So intermediate states matter
- Documentation size?
  - In worst case the only meaningful specification is the method itself
- Can add new methods?
  - In worst case only by exploring all possible interactions with existing methods
- Is there any rescue from this morass?

Sequential vs Concurrent

Sequential:

- Methods take time
- Sure, but their *effect* might as well be instantaneous
- Objects need meaningful state only between method calls

Concurrent:

- Method call is not an event
- Method call is an interval
- Since method calls overlap object might never be between method calls

Sequential vs Concurrent

Sequential:

- Each method described in isolation
- Can add new method without affecting old ones

Concurrent:

- Must characterize all possible interactions between concurrent calls
  - What if two deq's overlap? Two enq's? One enq and one deq? Etc? When and how do they overlap?
- Everything can potentially interact with everything else

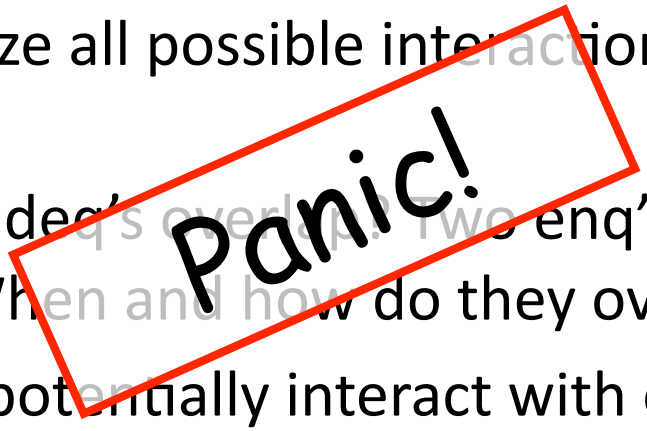
Sequential vs Concurrent

Sequential:

- Each method described in isolation
- Can add new method without affecting old ones

Concurrent:

- Must characterize all possible interactions between concurrent calls
  - What if two deq's overlap? Two enq's? One enq and one deq? Etc? When and how do they overlap?
- Everything can potentially interact with everything else



Intuitively…

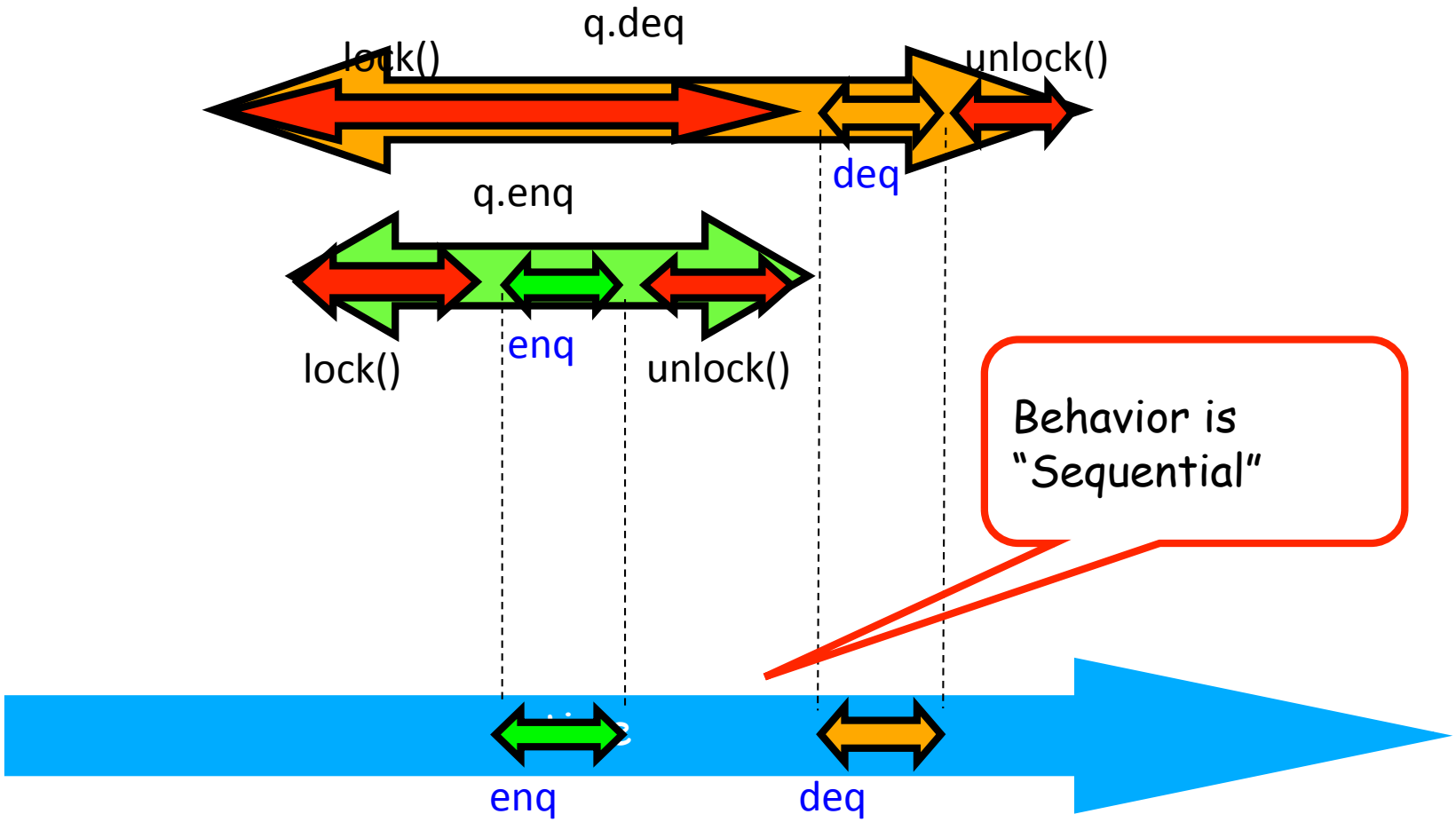
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Intuitively…

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

All modifications of queue are done mutually exclusive

Lets capture the idea of describing
the concurrent via the sequential



Linearizability

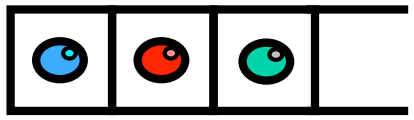
The effect of each method is

- Instantaneous
- Takes place inbetween invocation and response events
- Methods can use locks to achieve this, or not
- Call such an execution linearizable

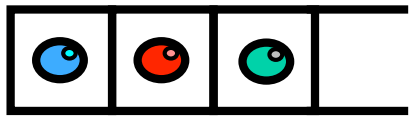
Linearizable object:

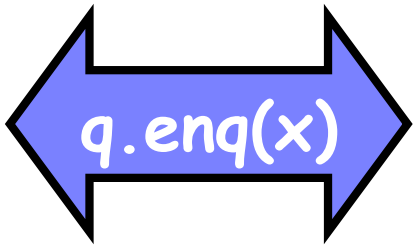
- Each execution is linearizable
- More formal definition to come …

Example

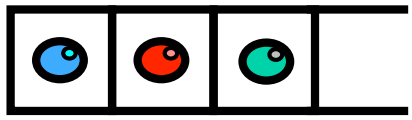


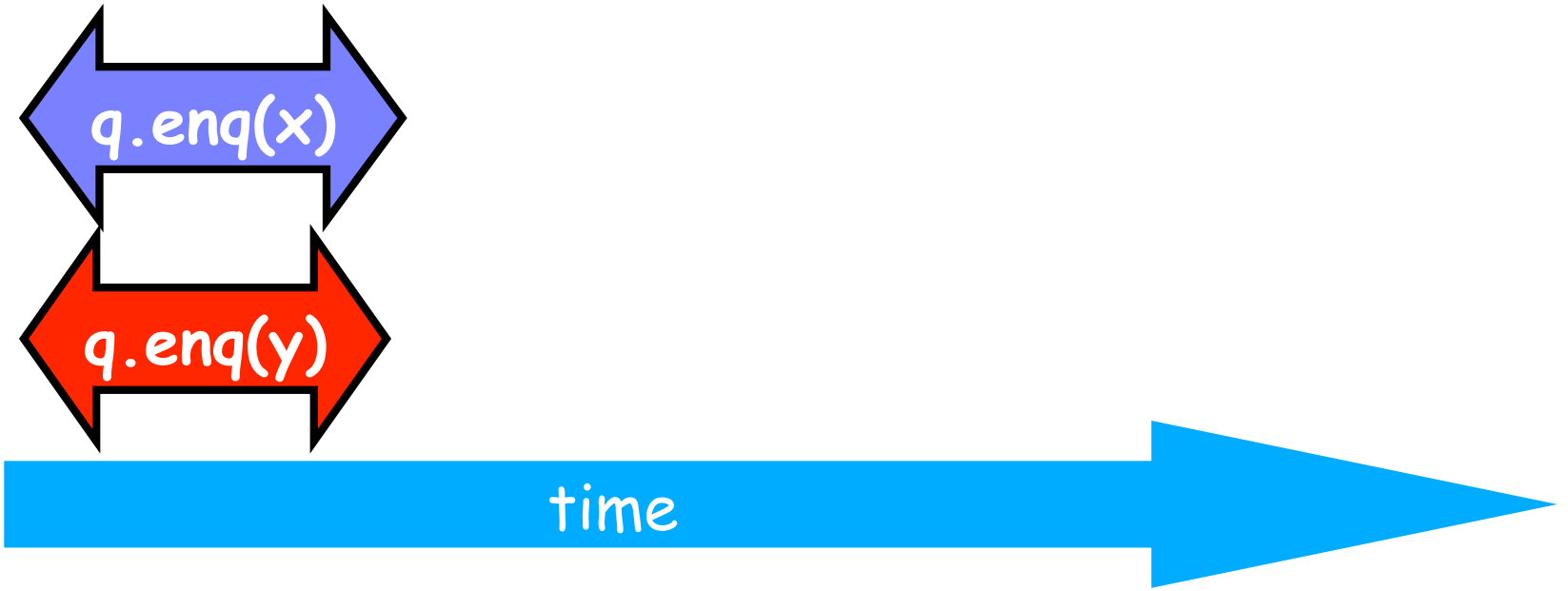
Example



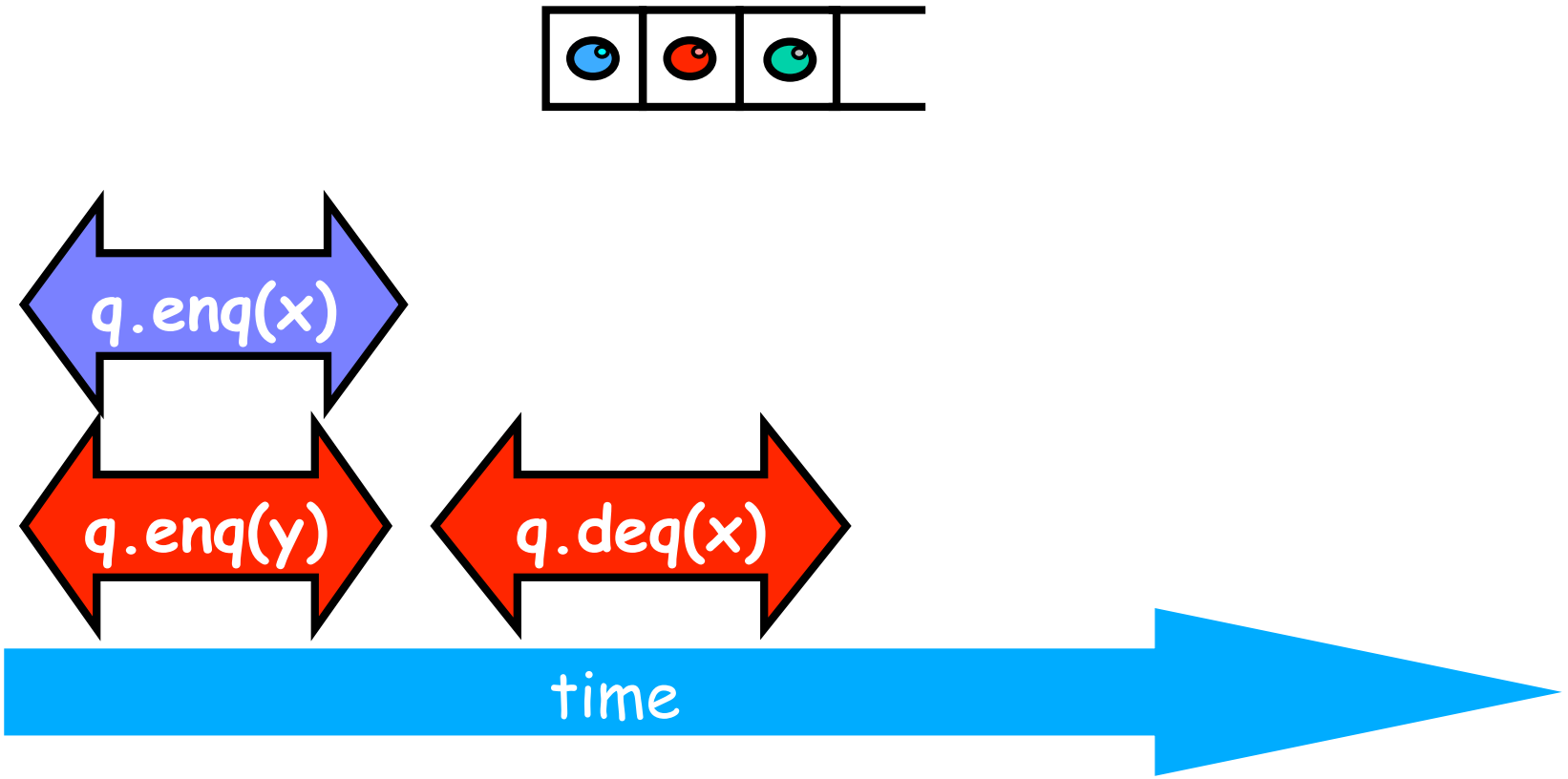


Example

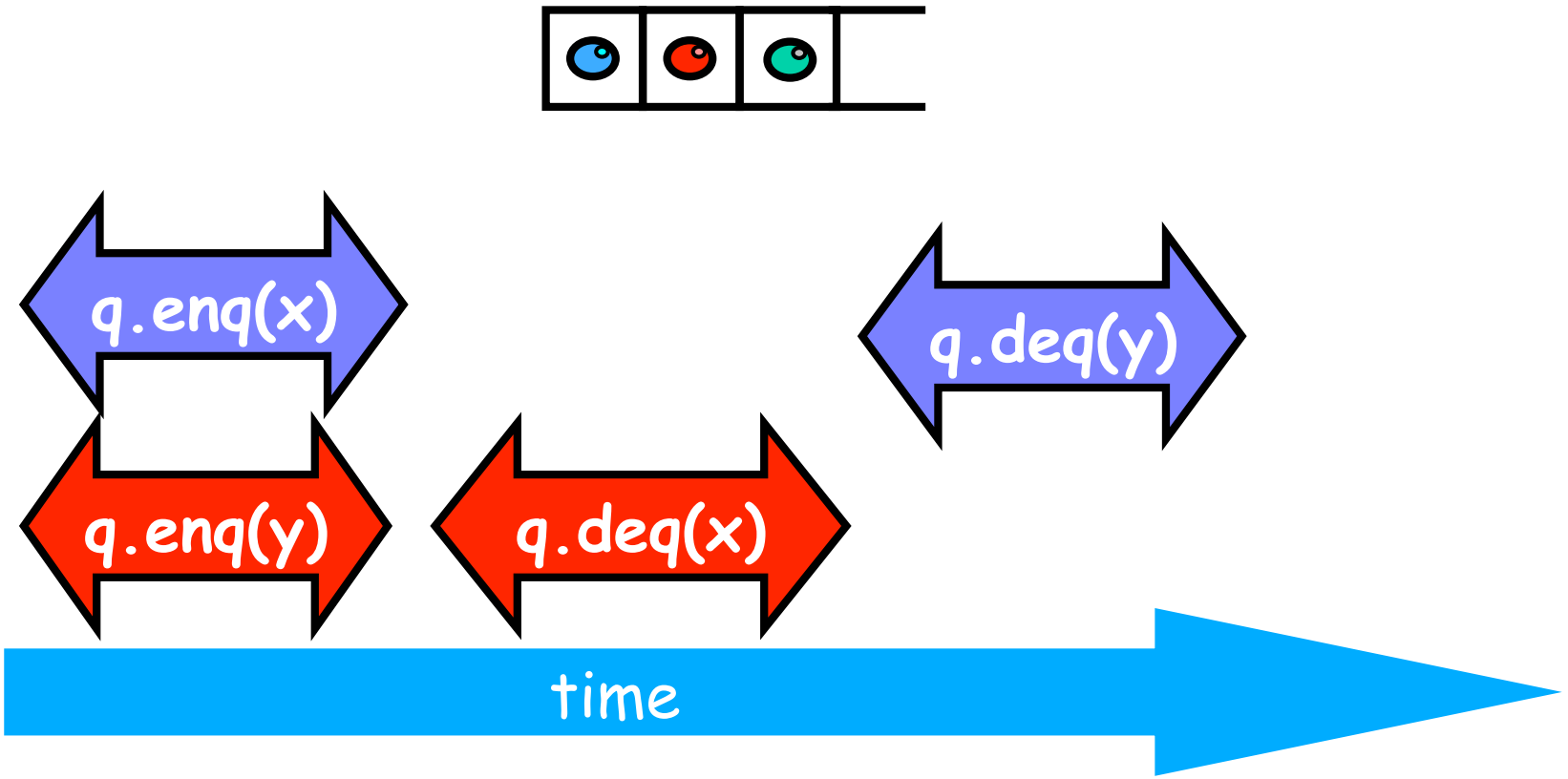




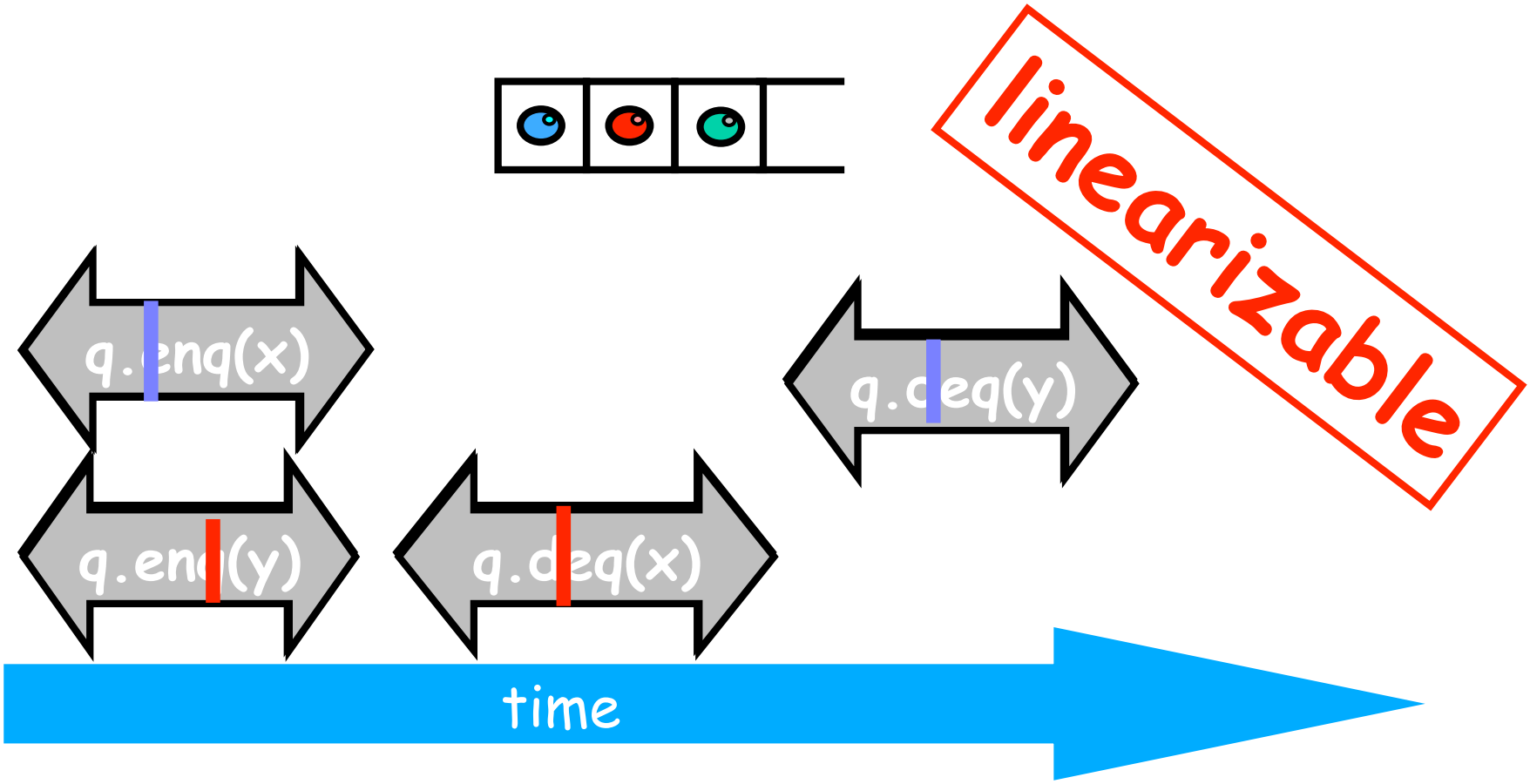
Example



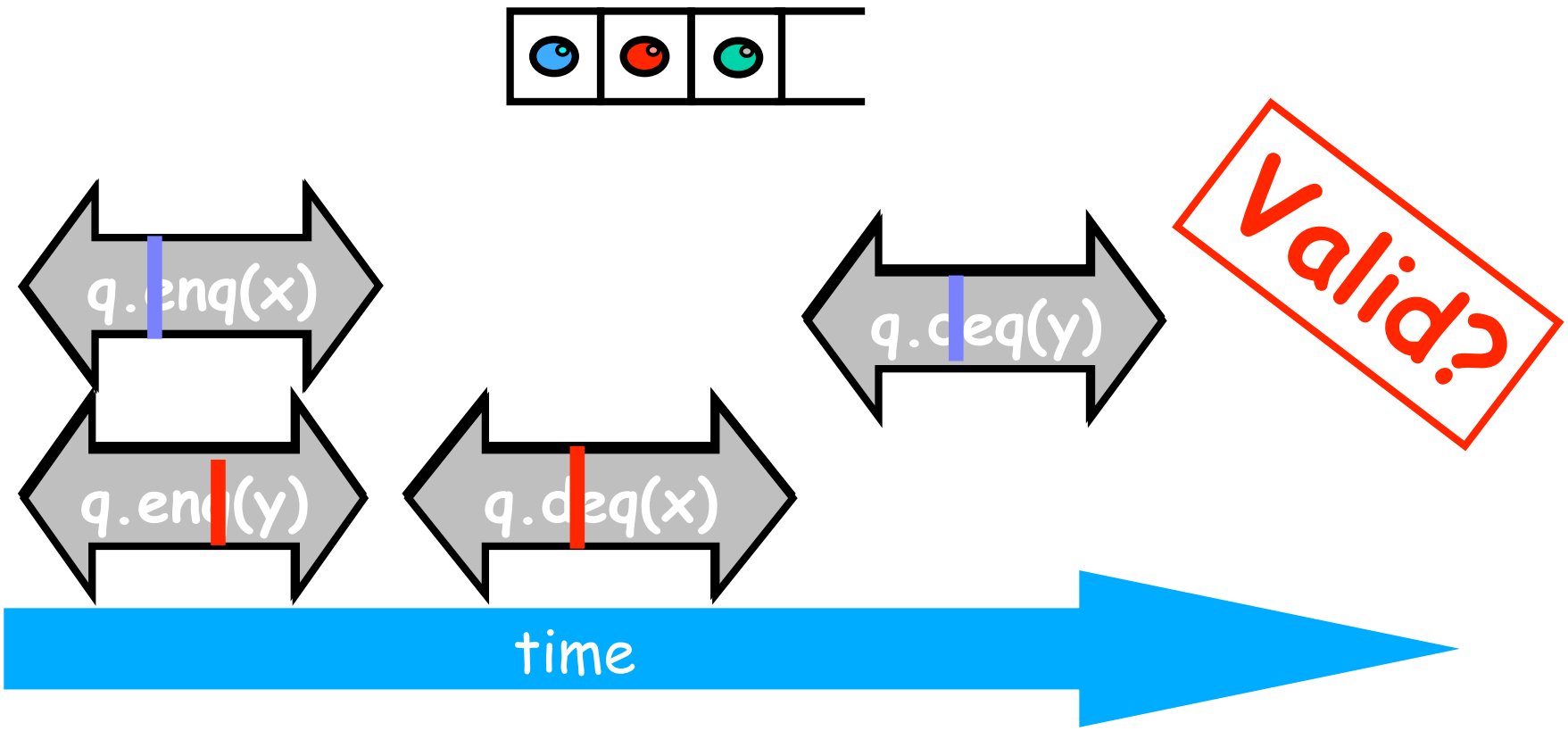
Example



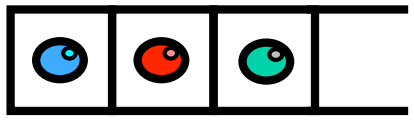
Example



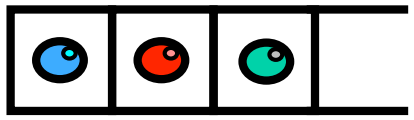
Example

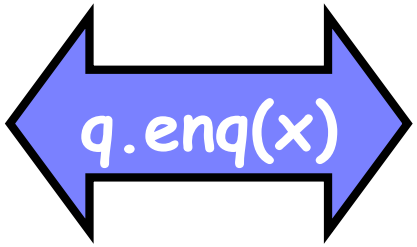


Example

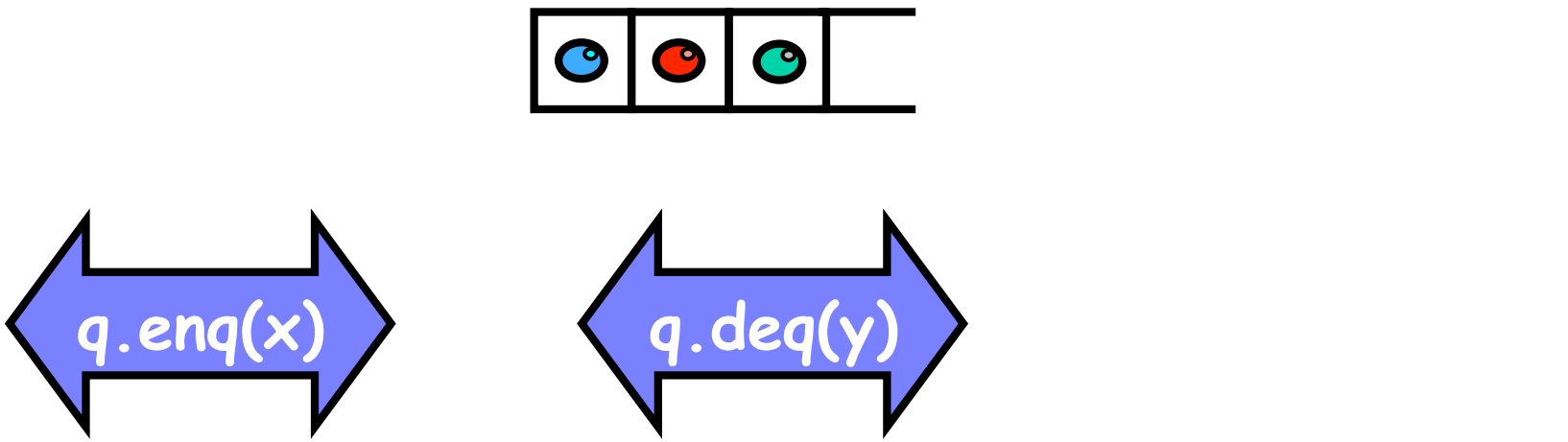


Example

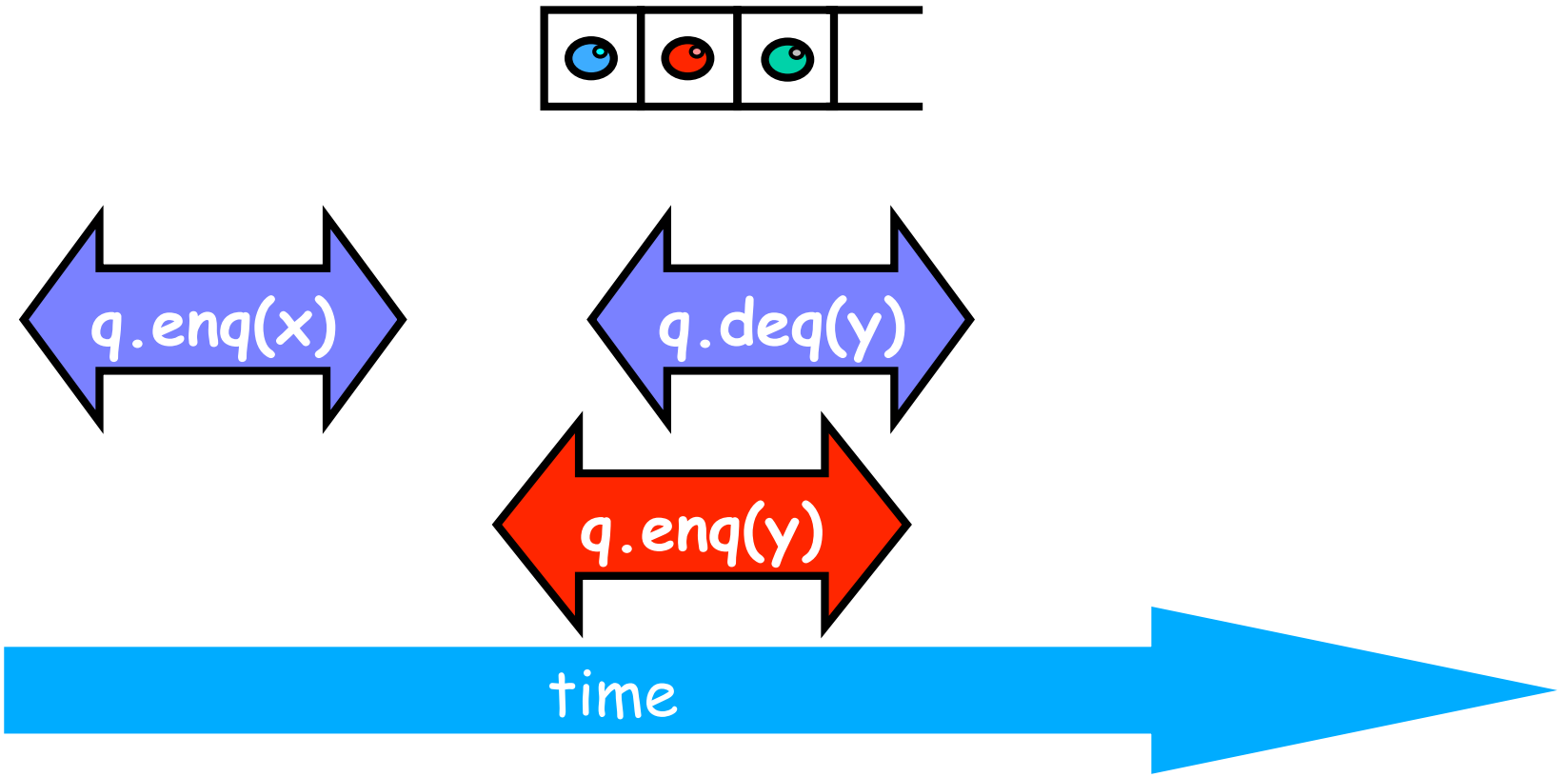




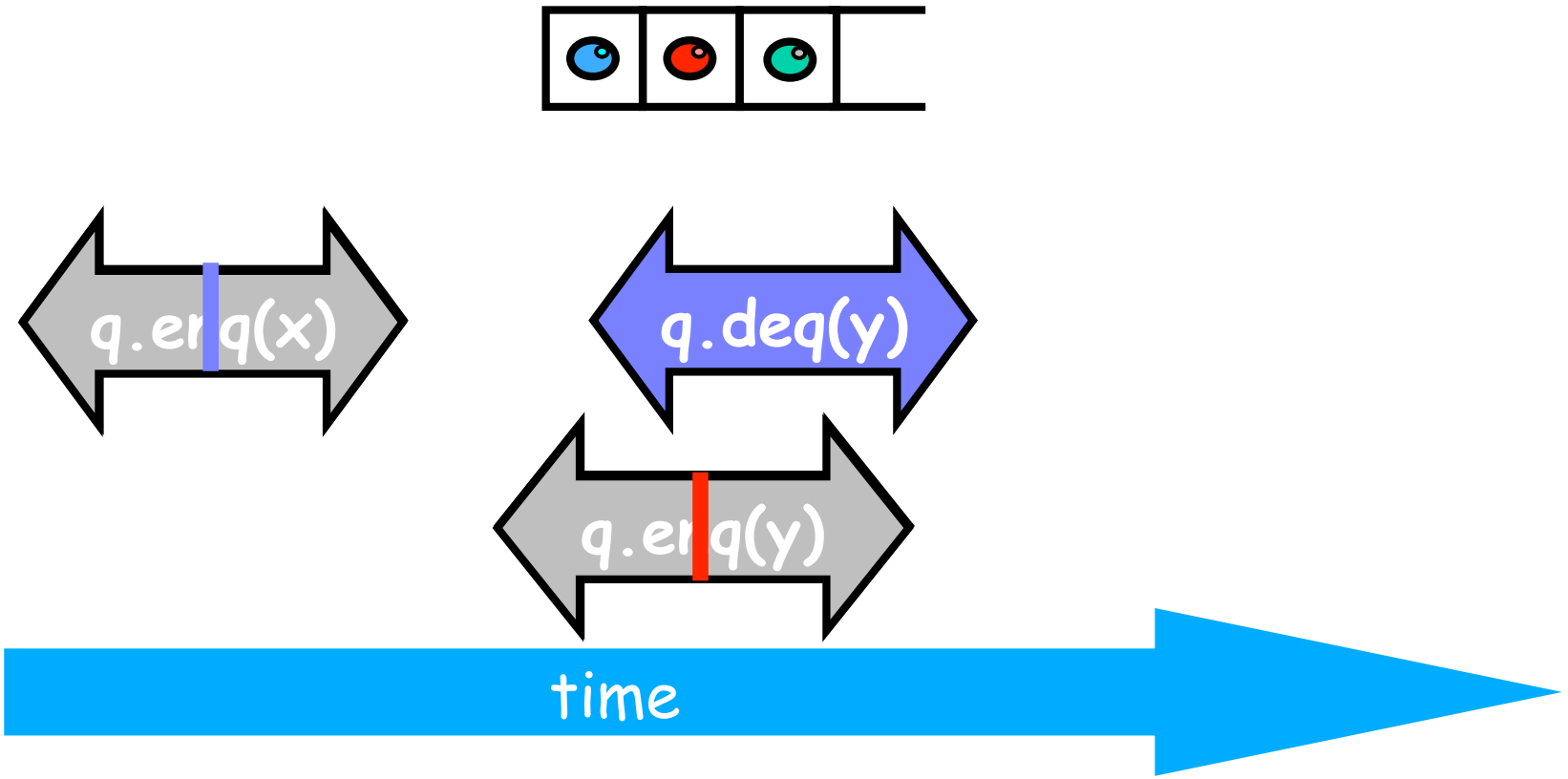
Example



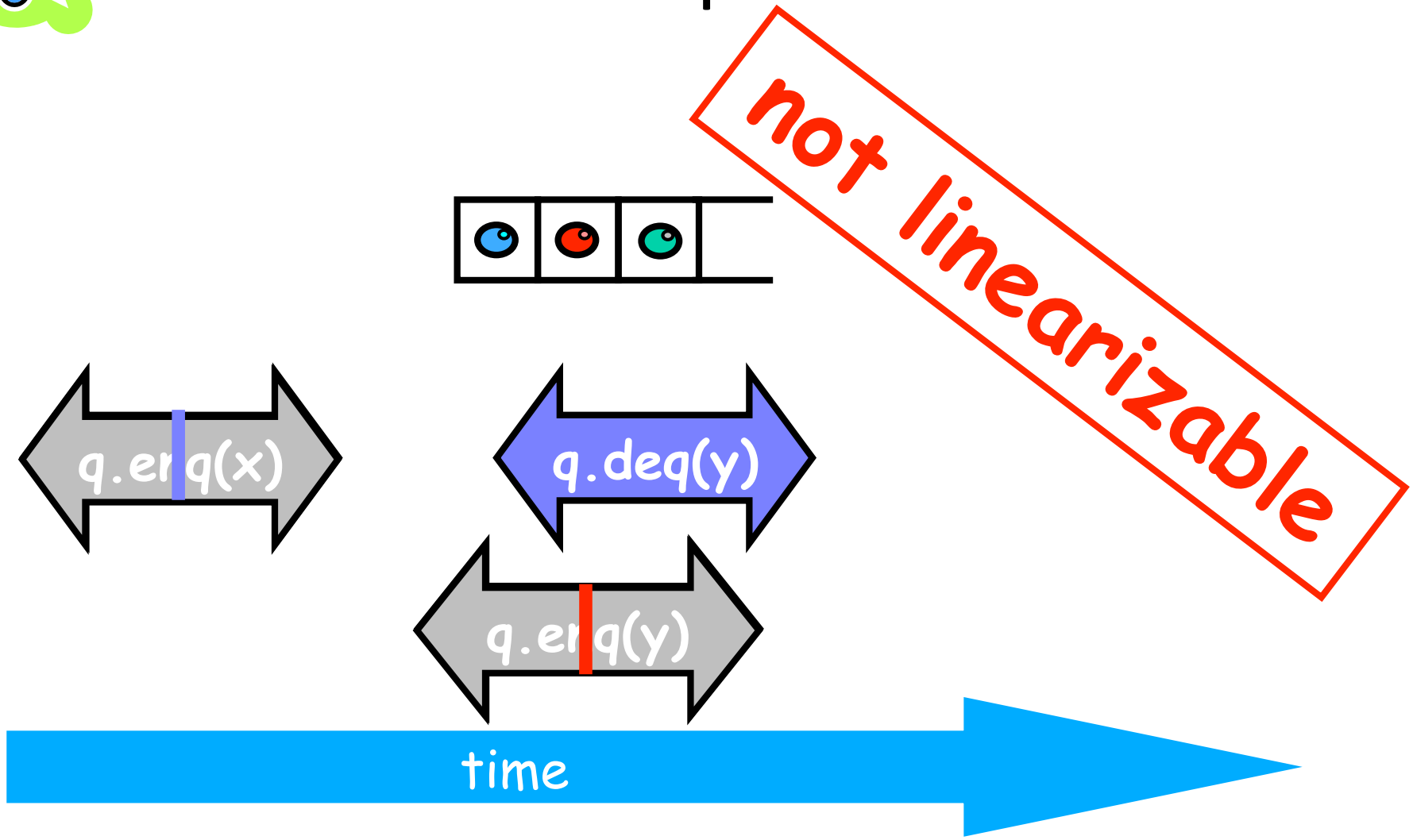
Example



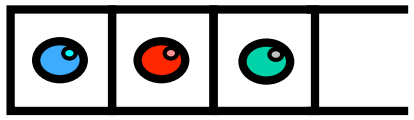
Example



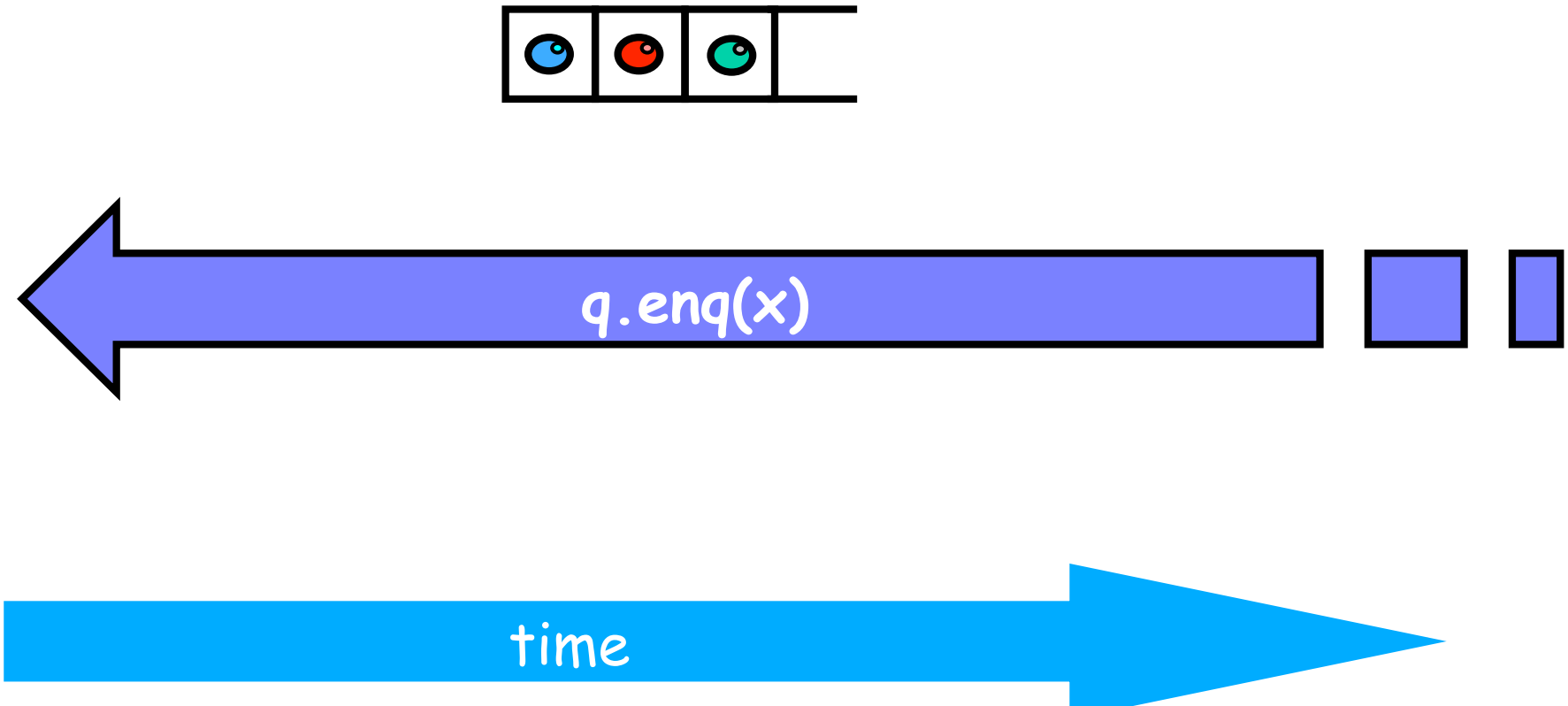
Example



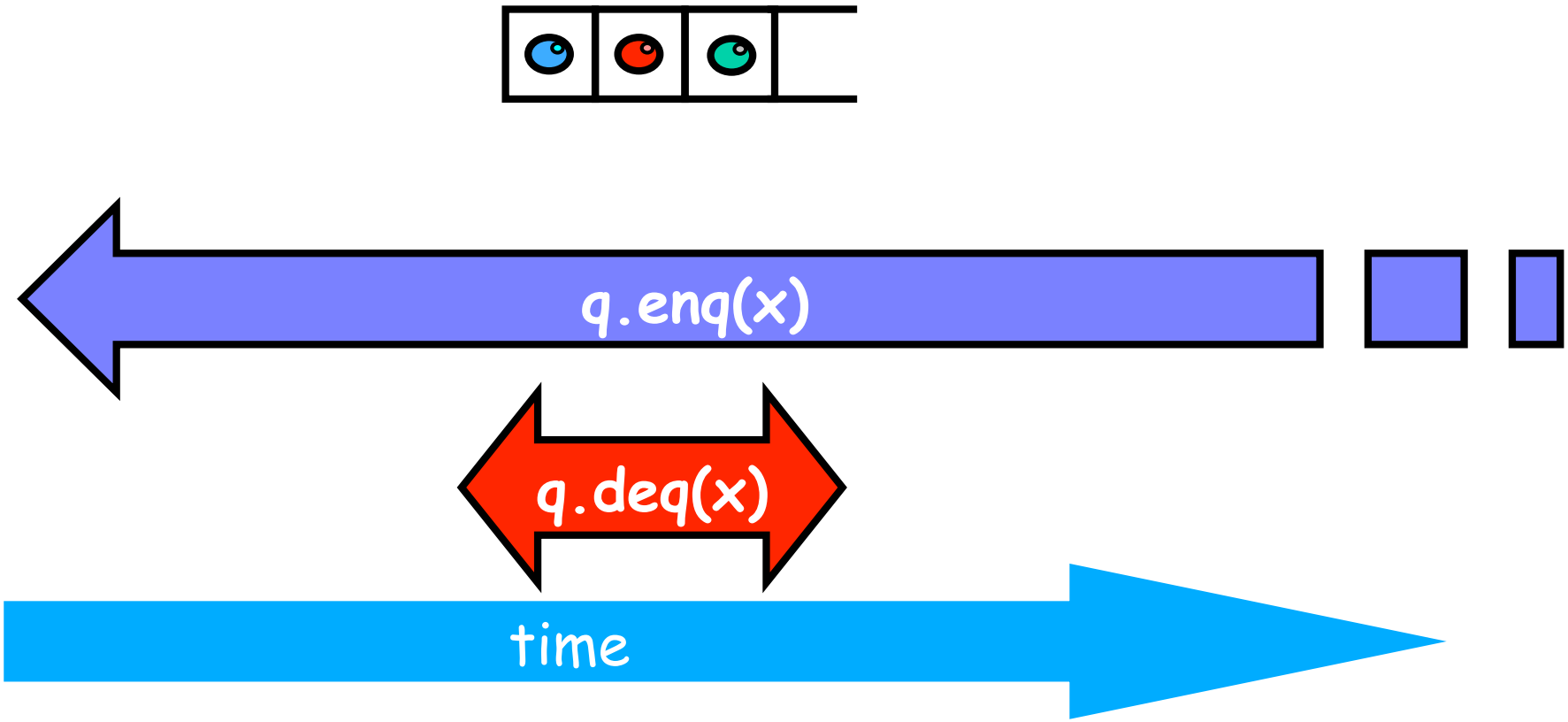
Example



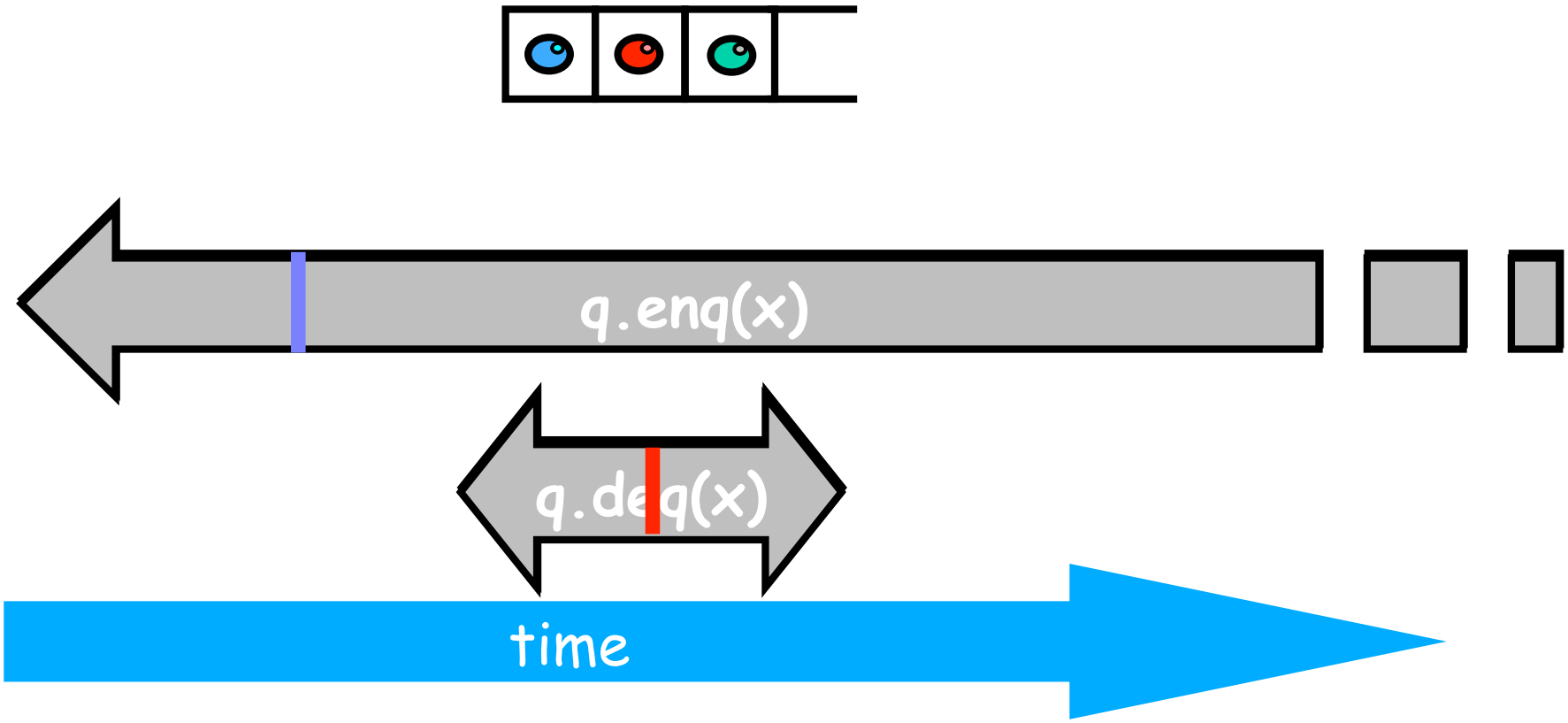
Example



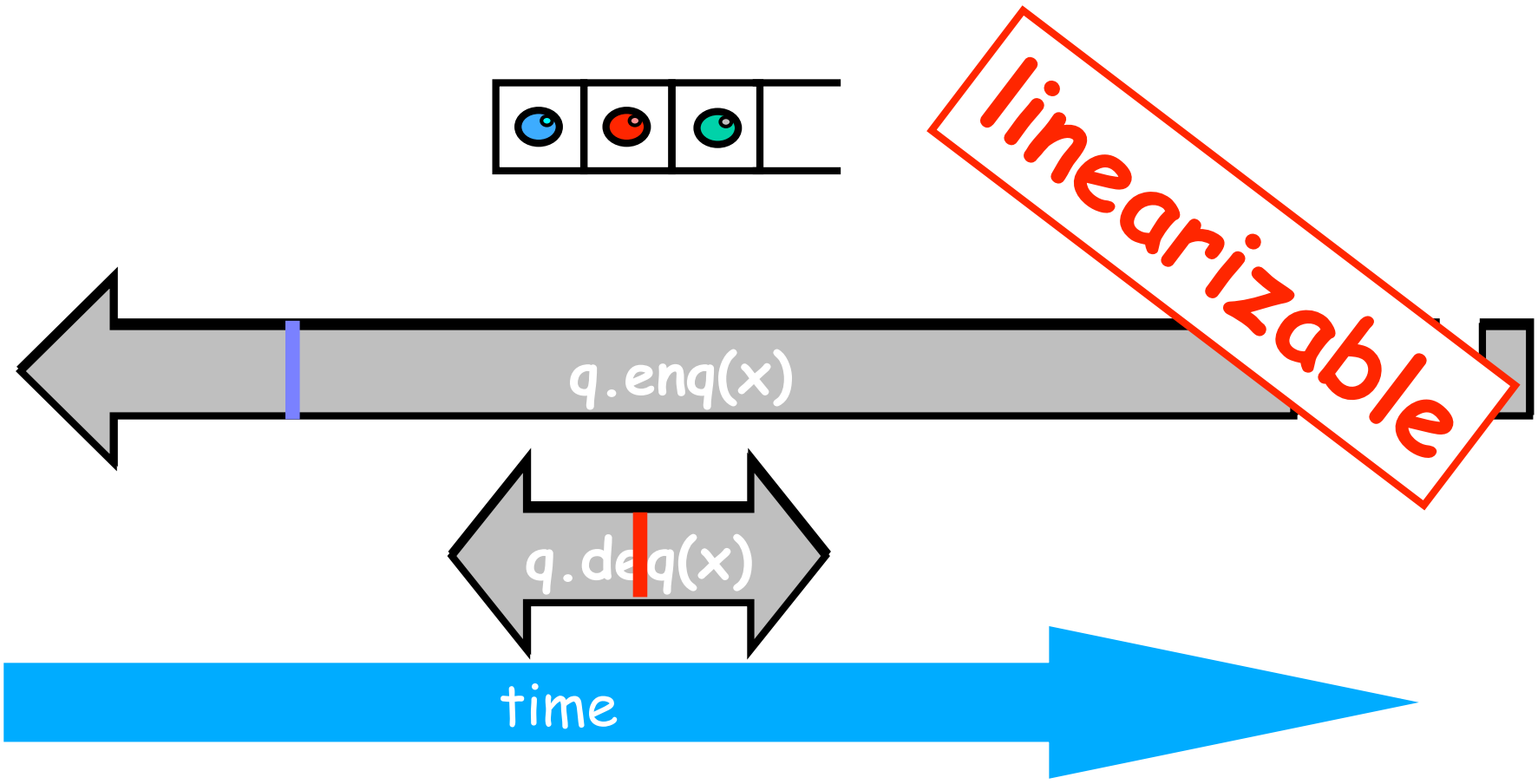
Example



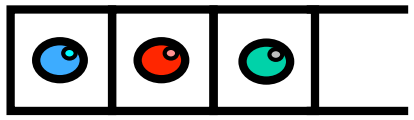
Example

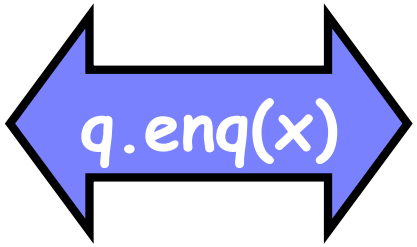


Example

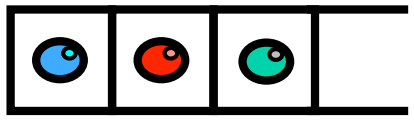


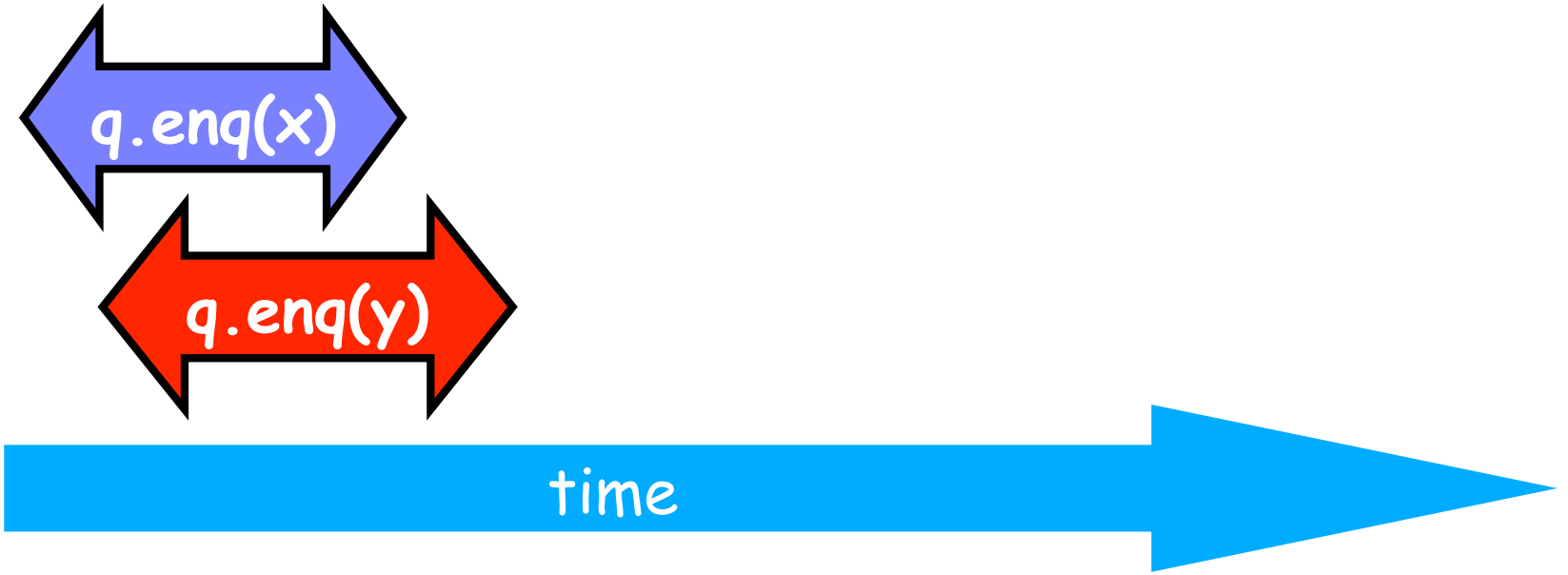
Example



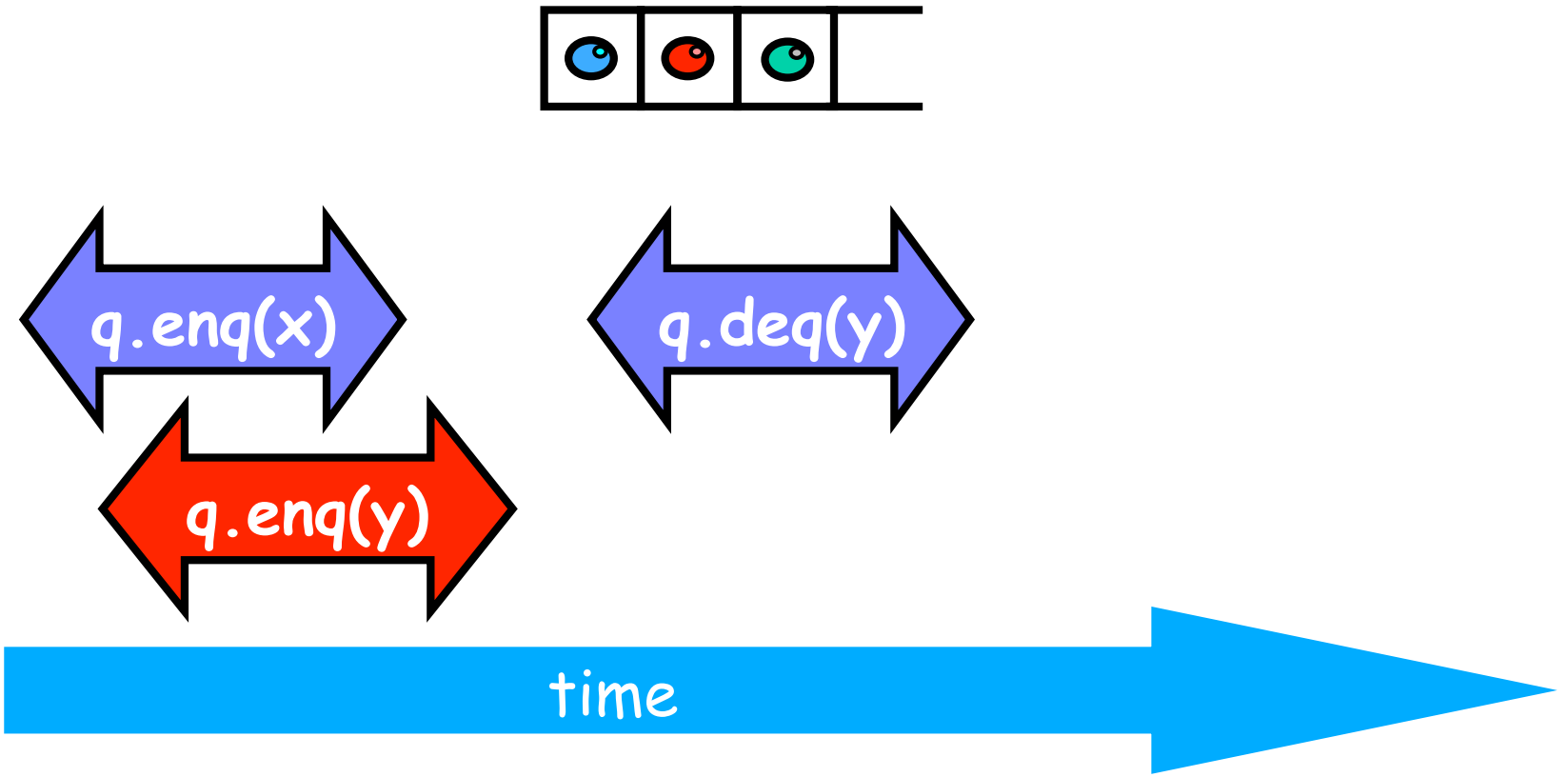


Example

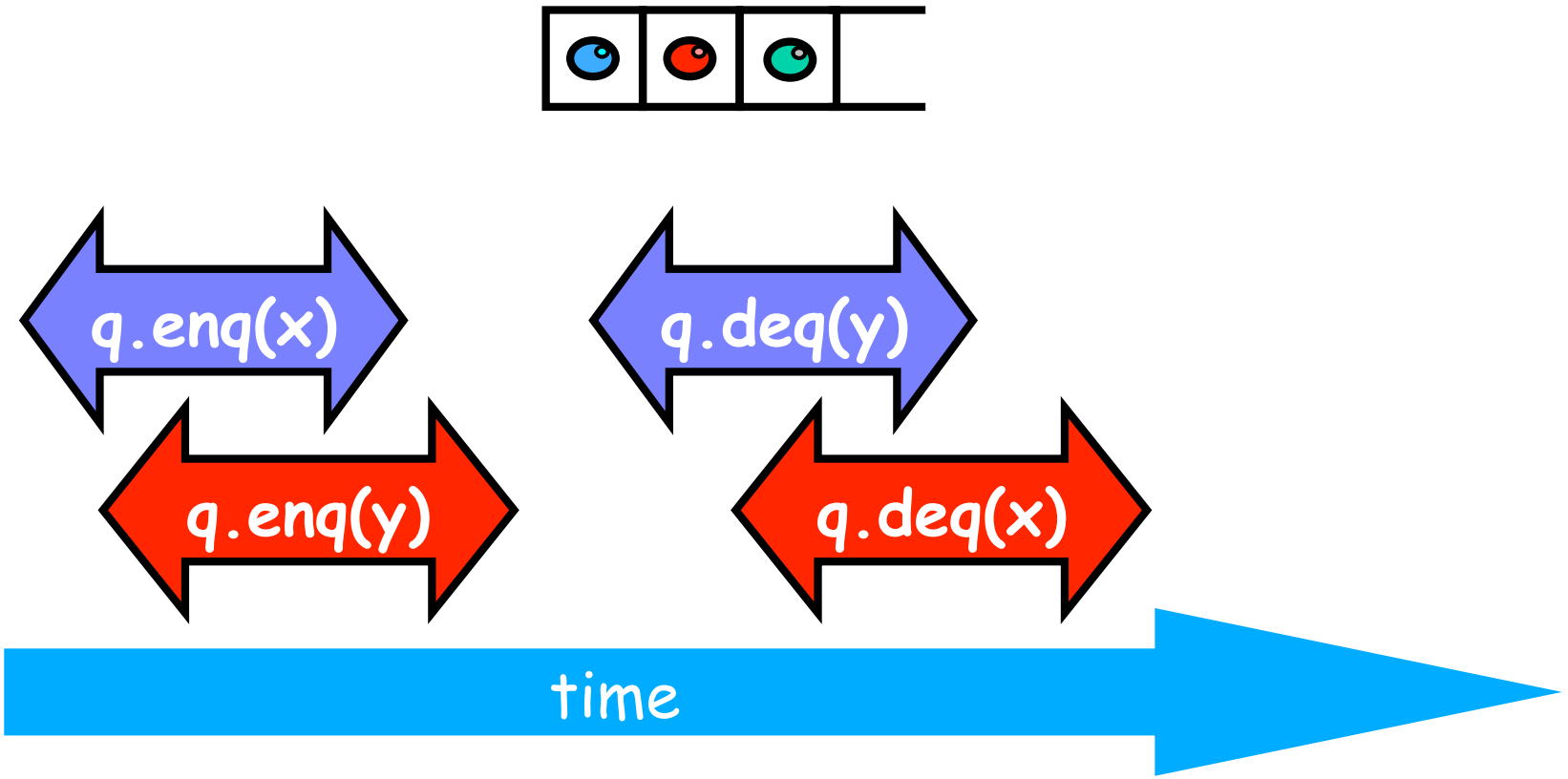




Example

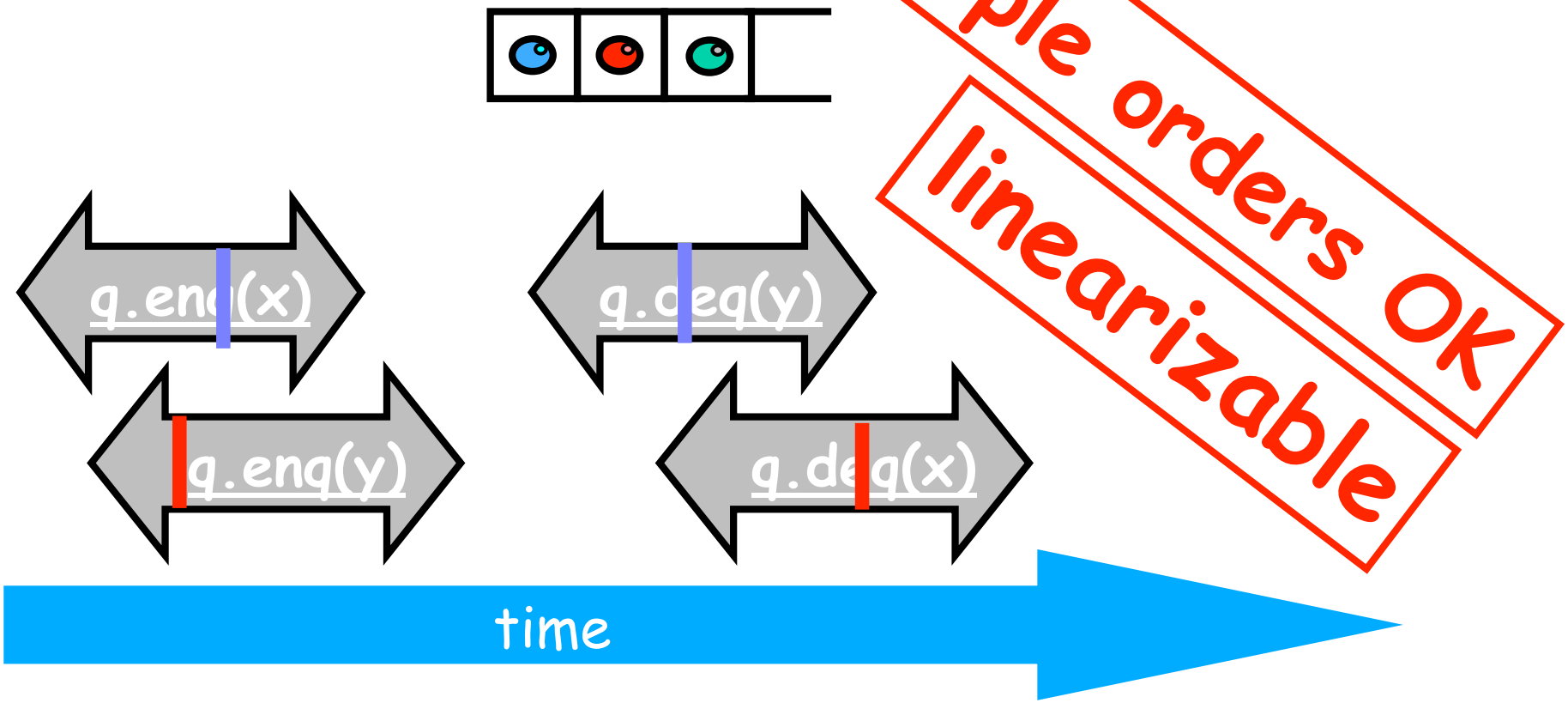


Example

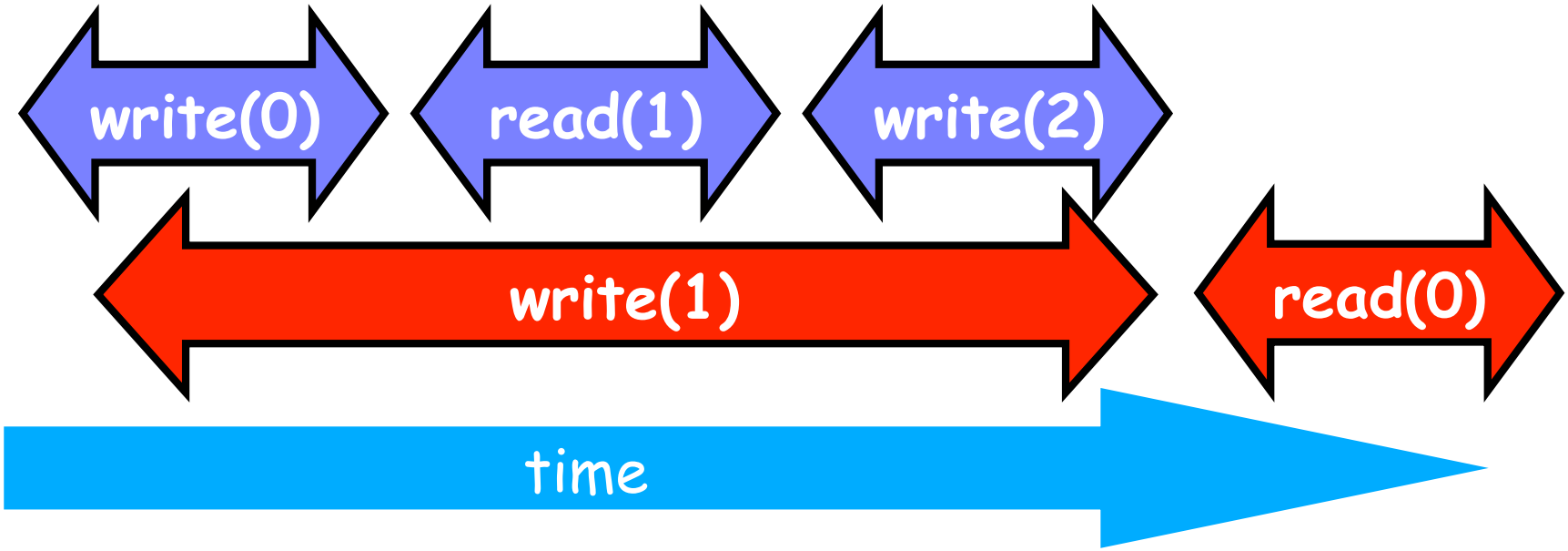


Comme ci
Comme ça

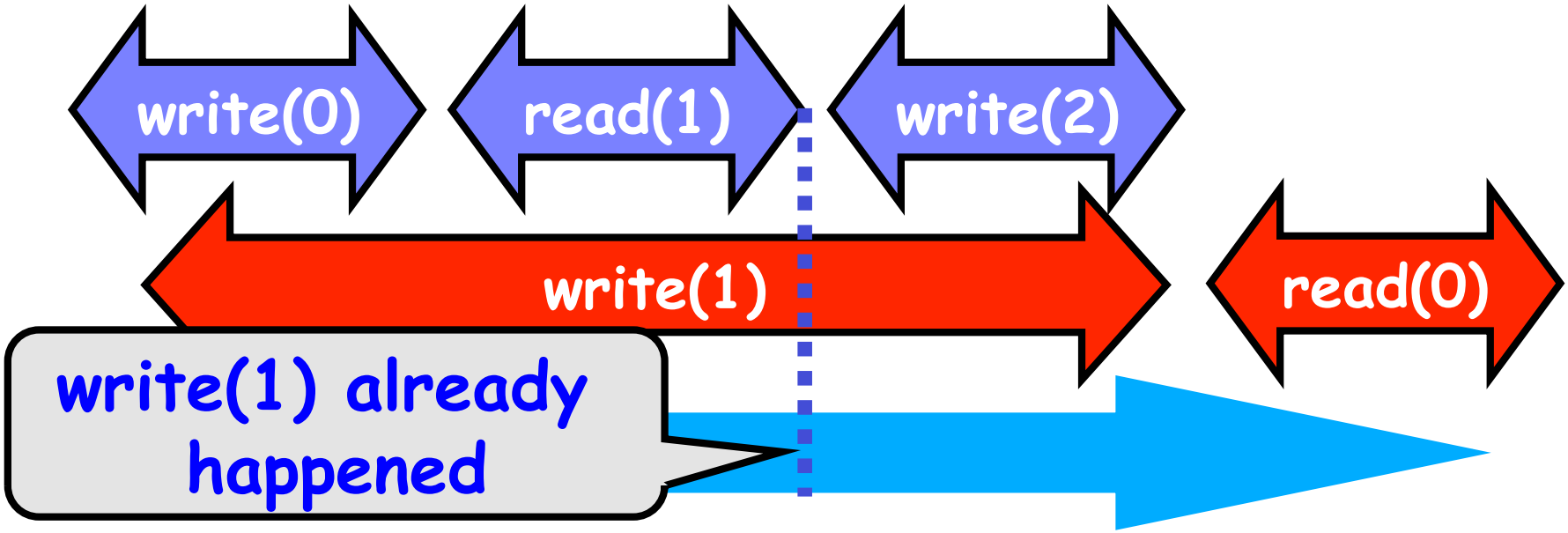
Example



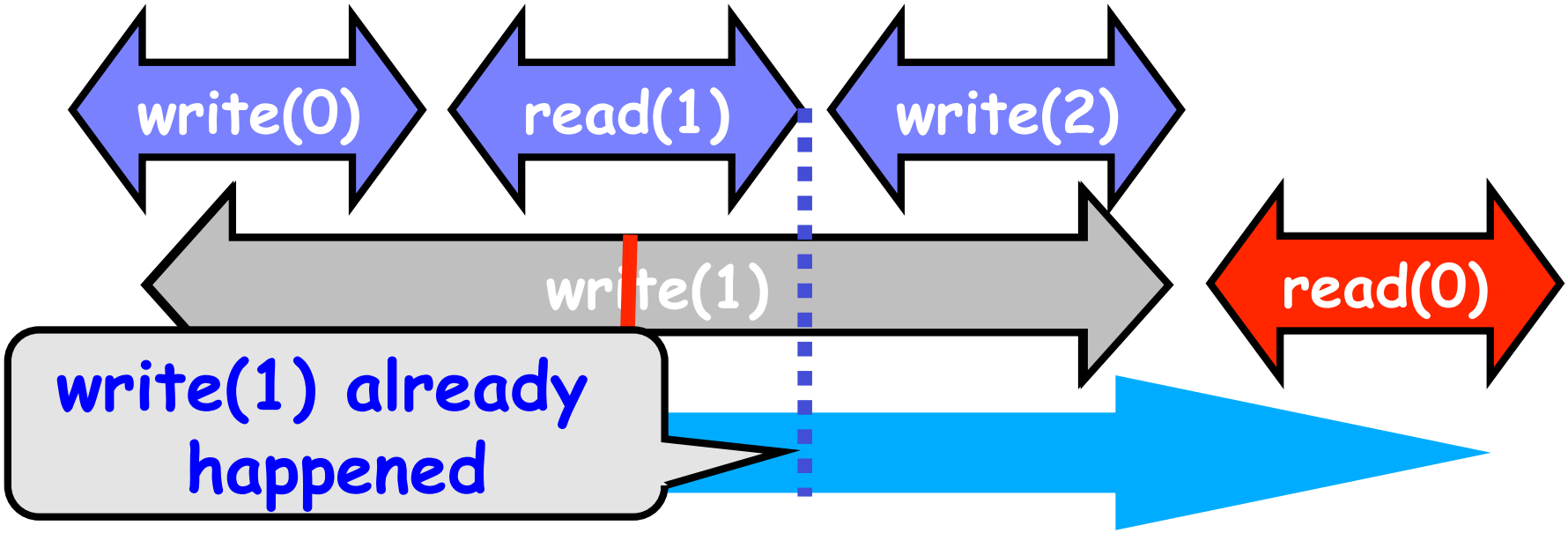
Read/Write Register Example



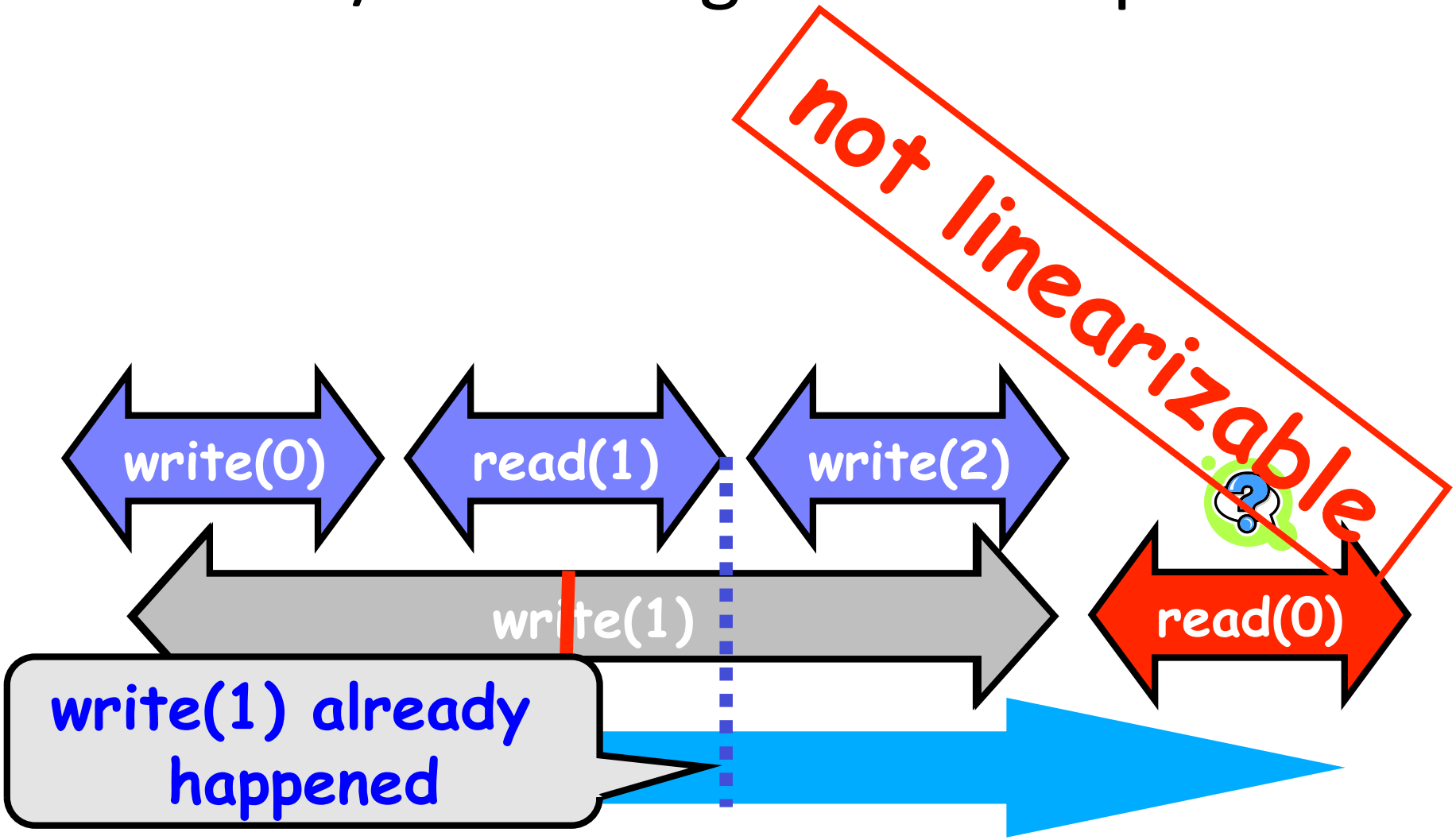
Read/Write Register Example



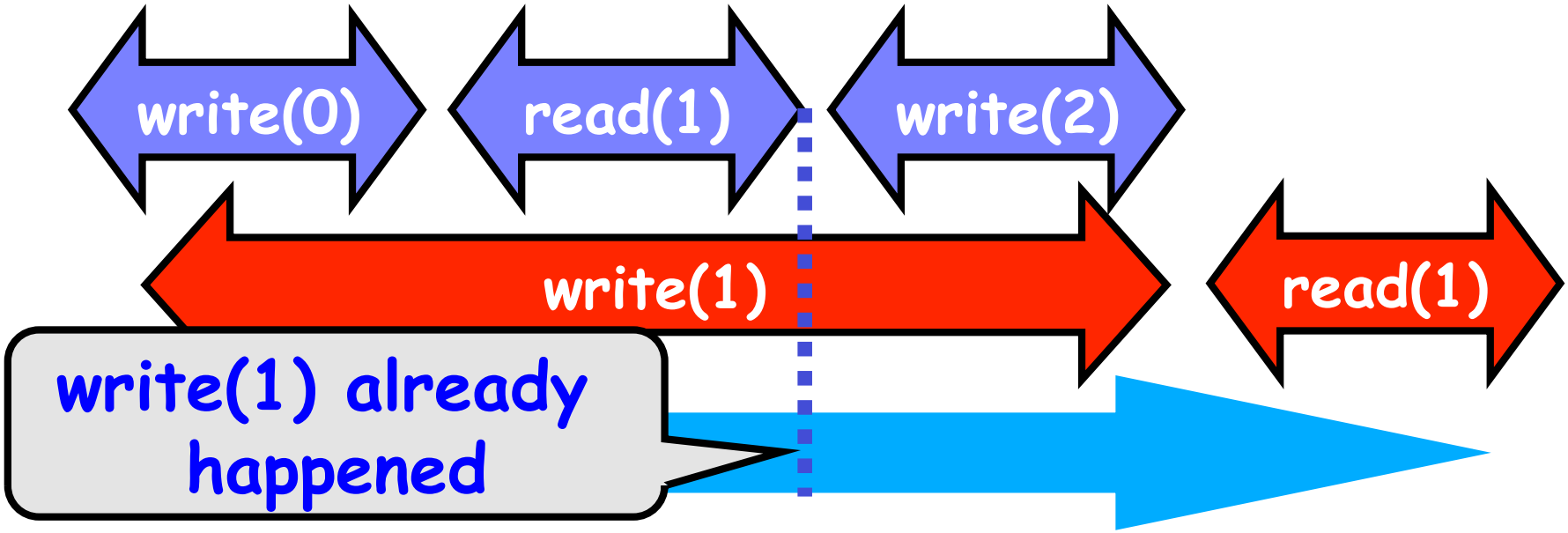
Read/Write Register Example



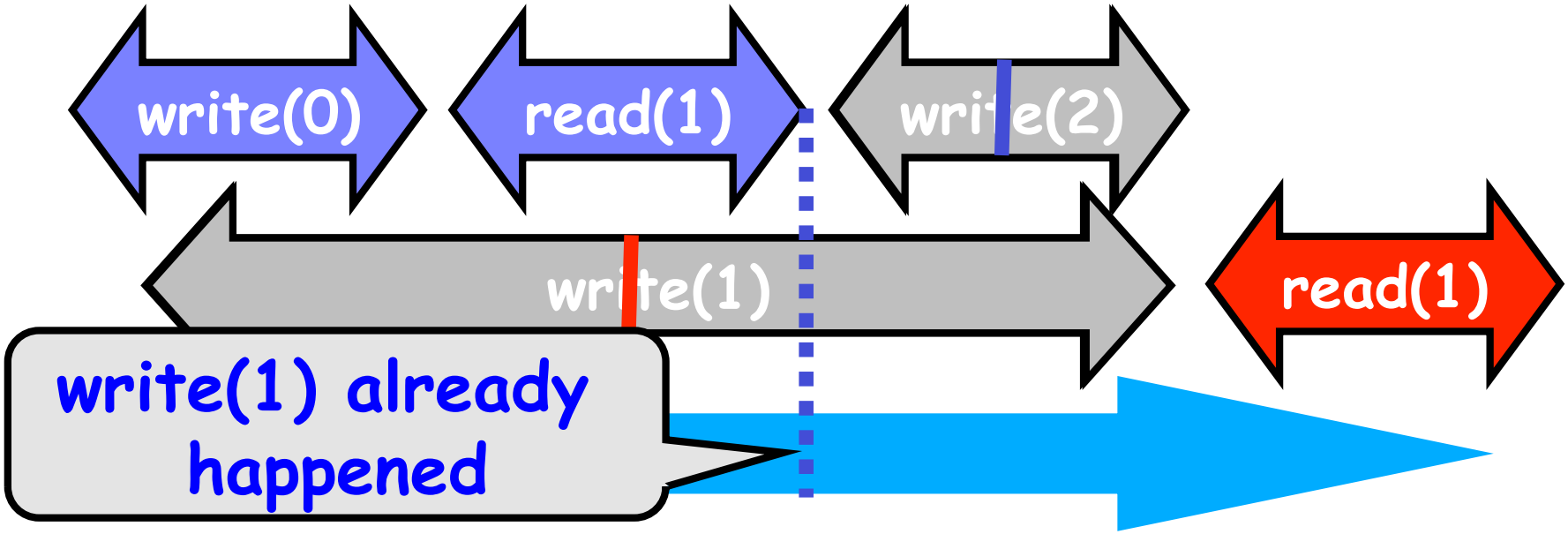
Read/Write Register Example



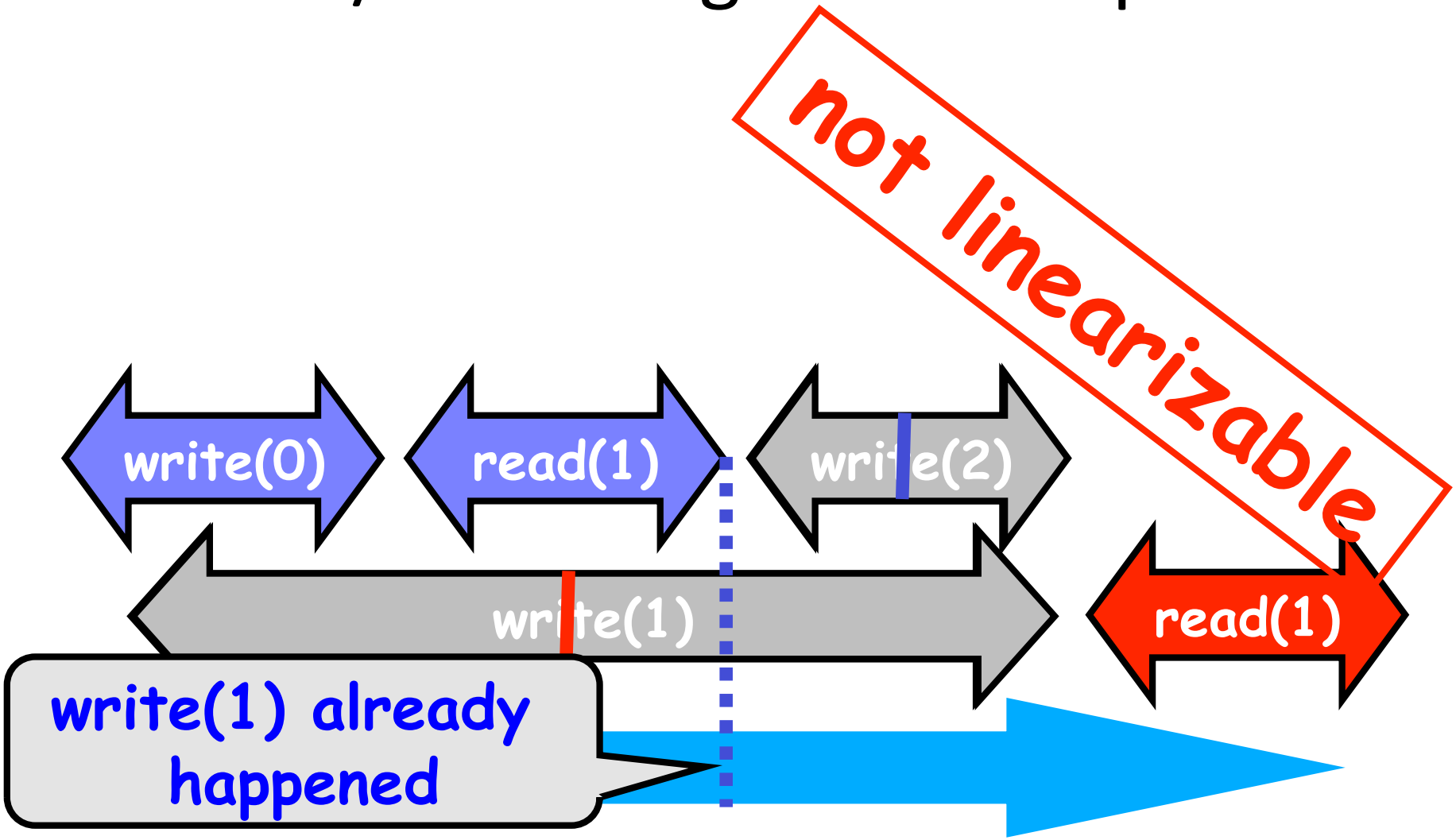
Read/Write Register Example



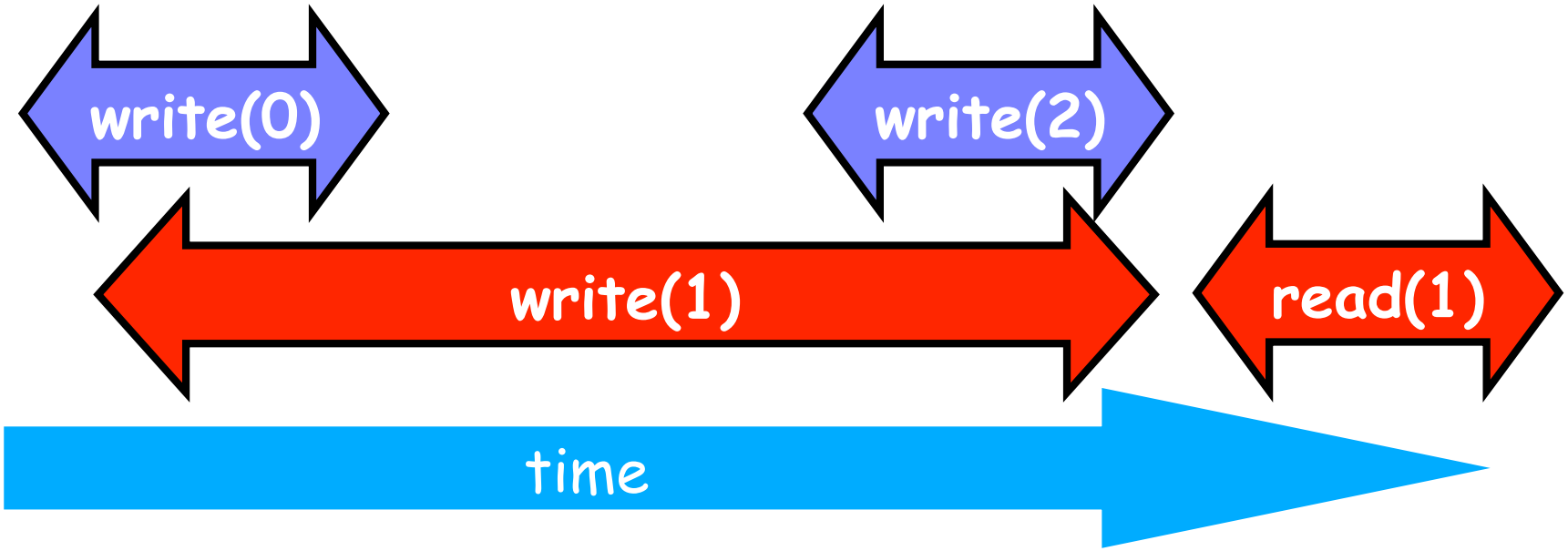
Read/Write Register Example



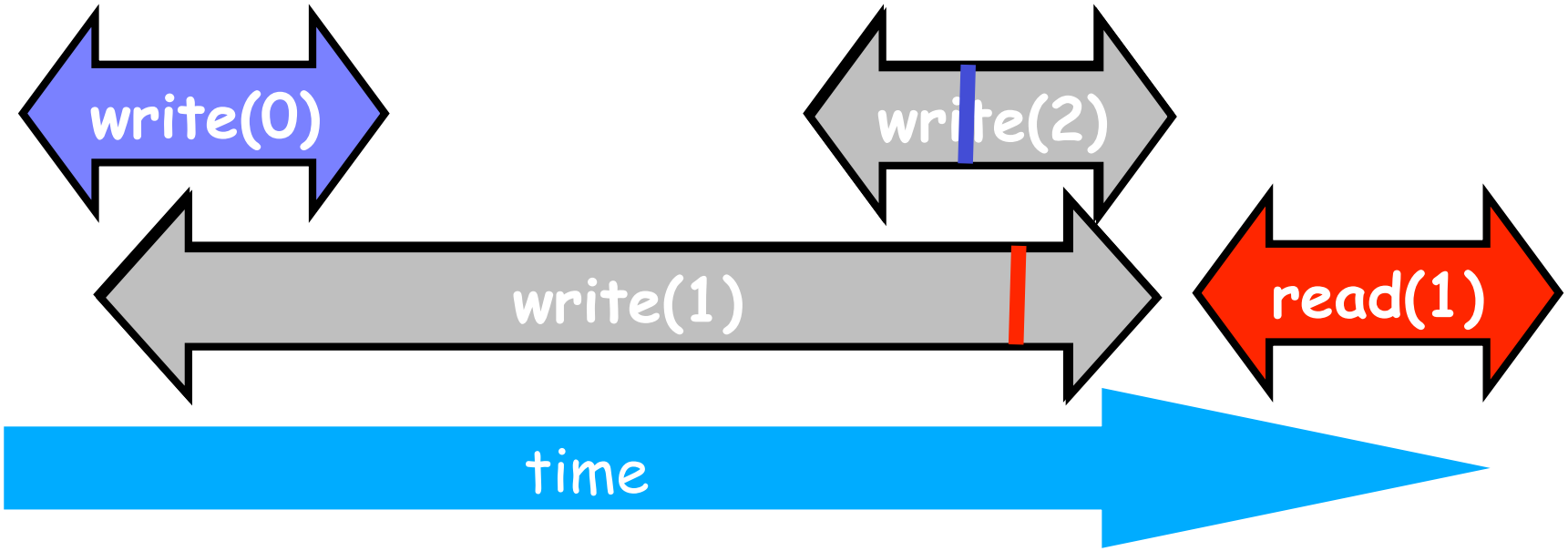
Read/Write Register Example



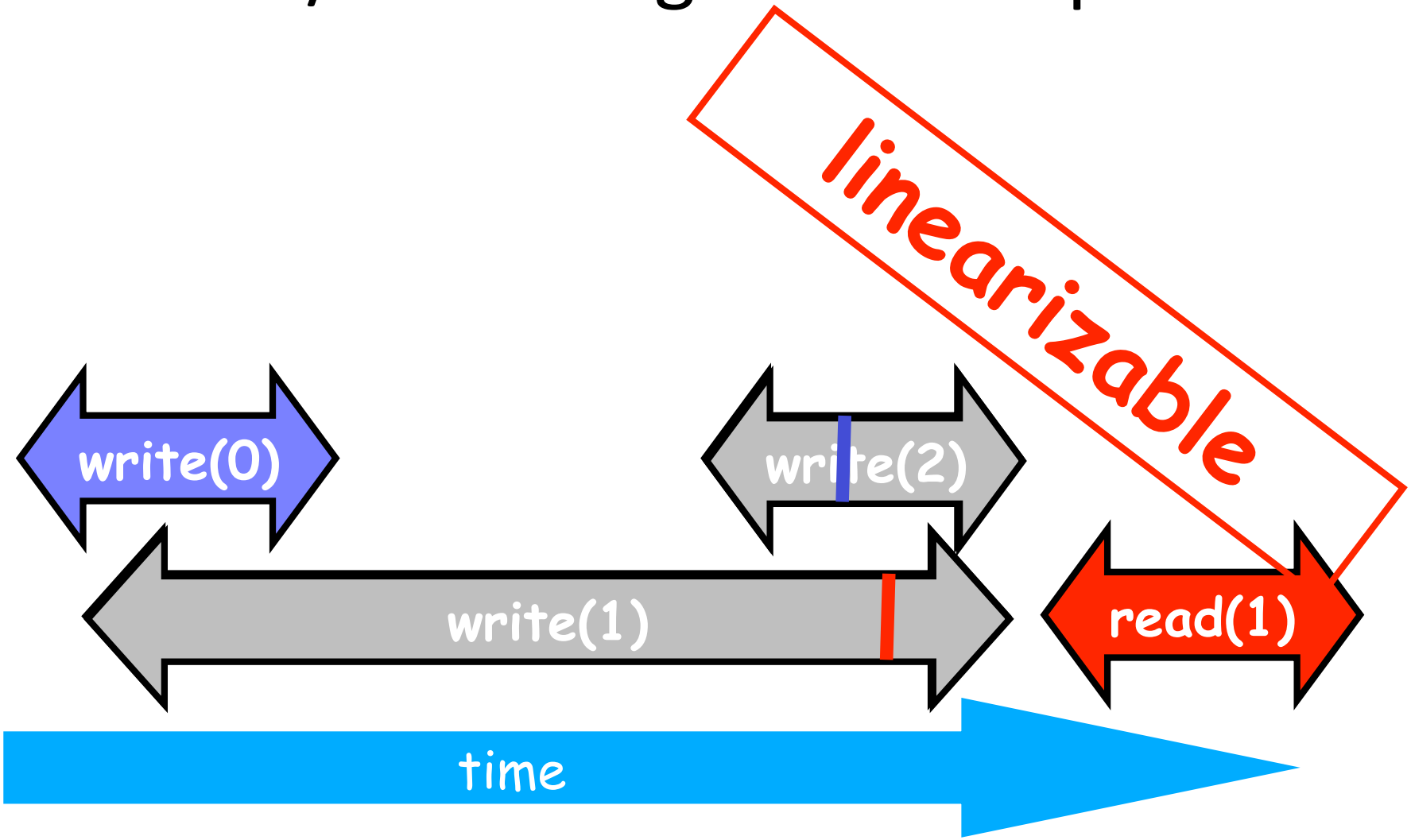
Read/Write Register Example



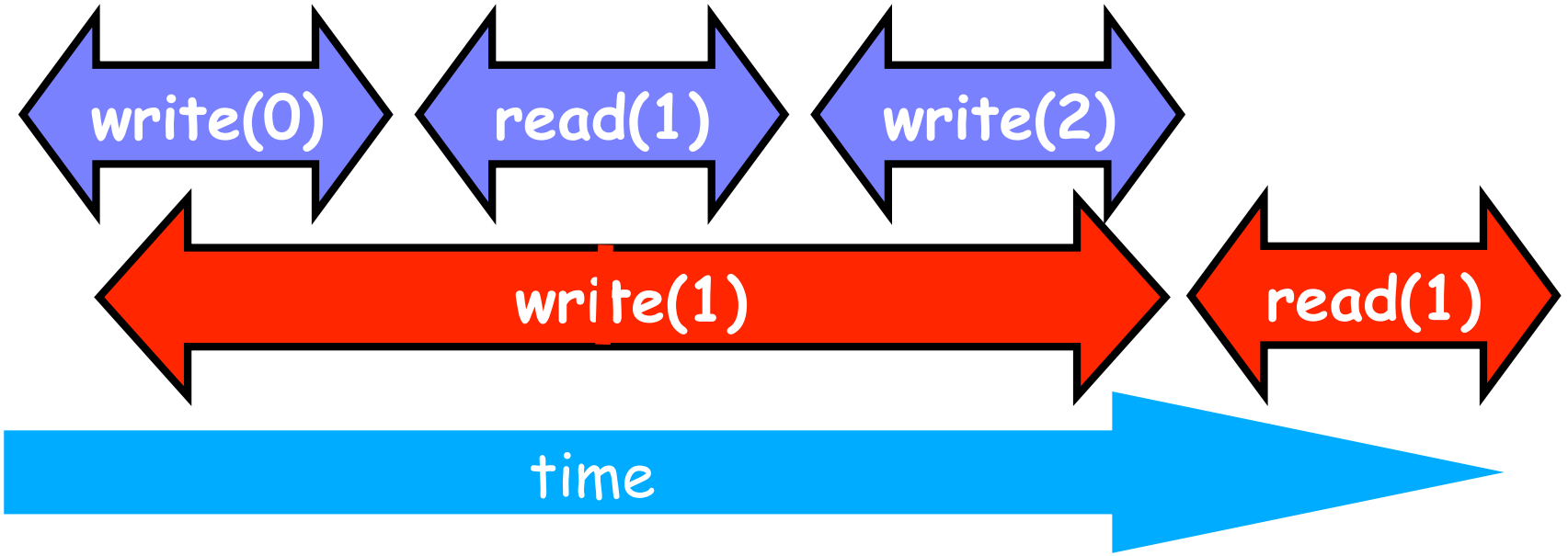
Read/Write Register Example



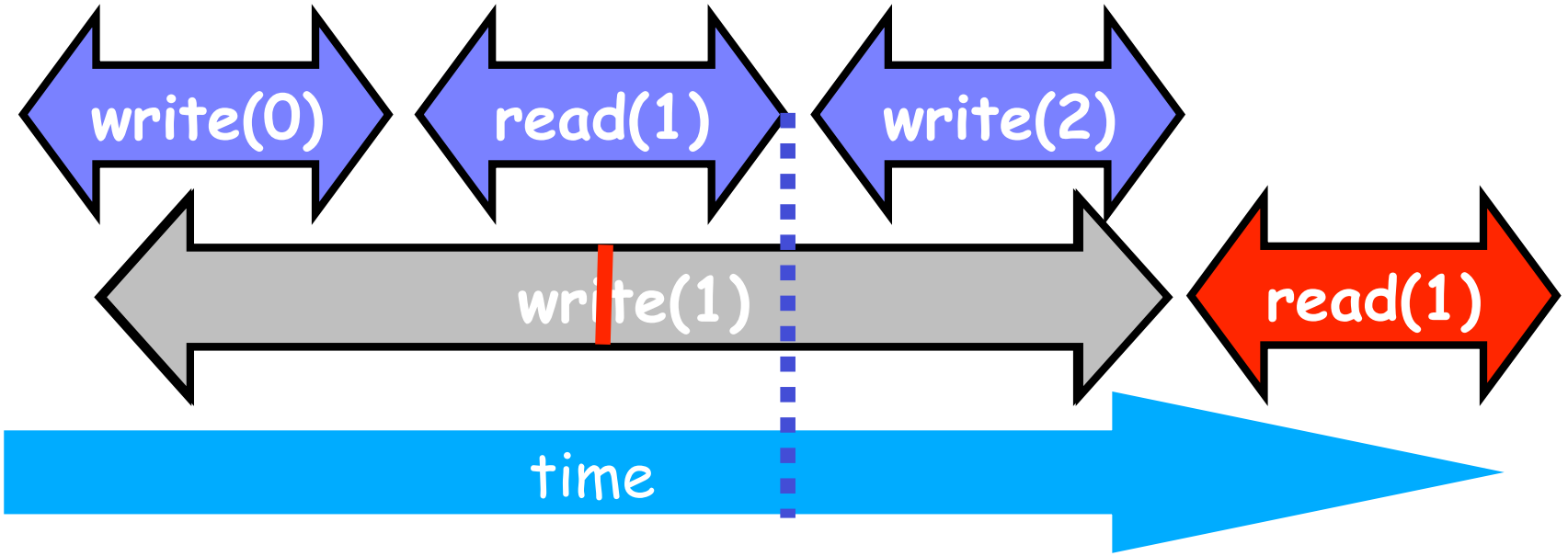
Read/Write Register Example



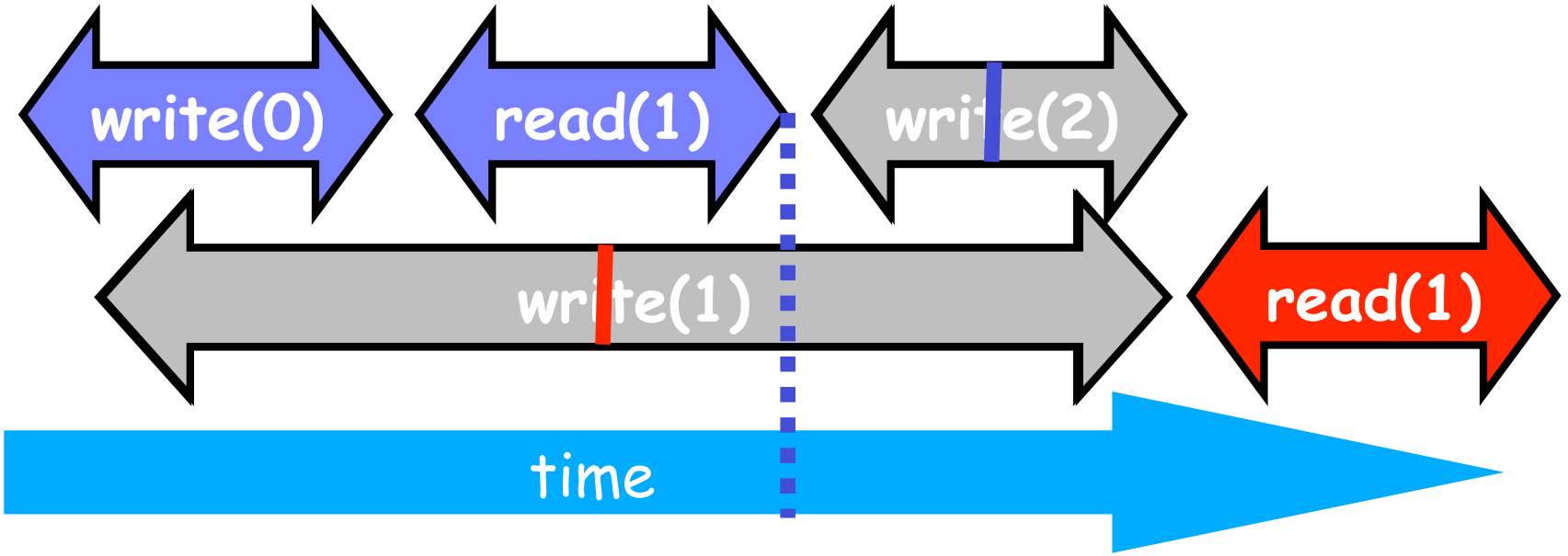
Read/Write Register Example



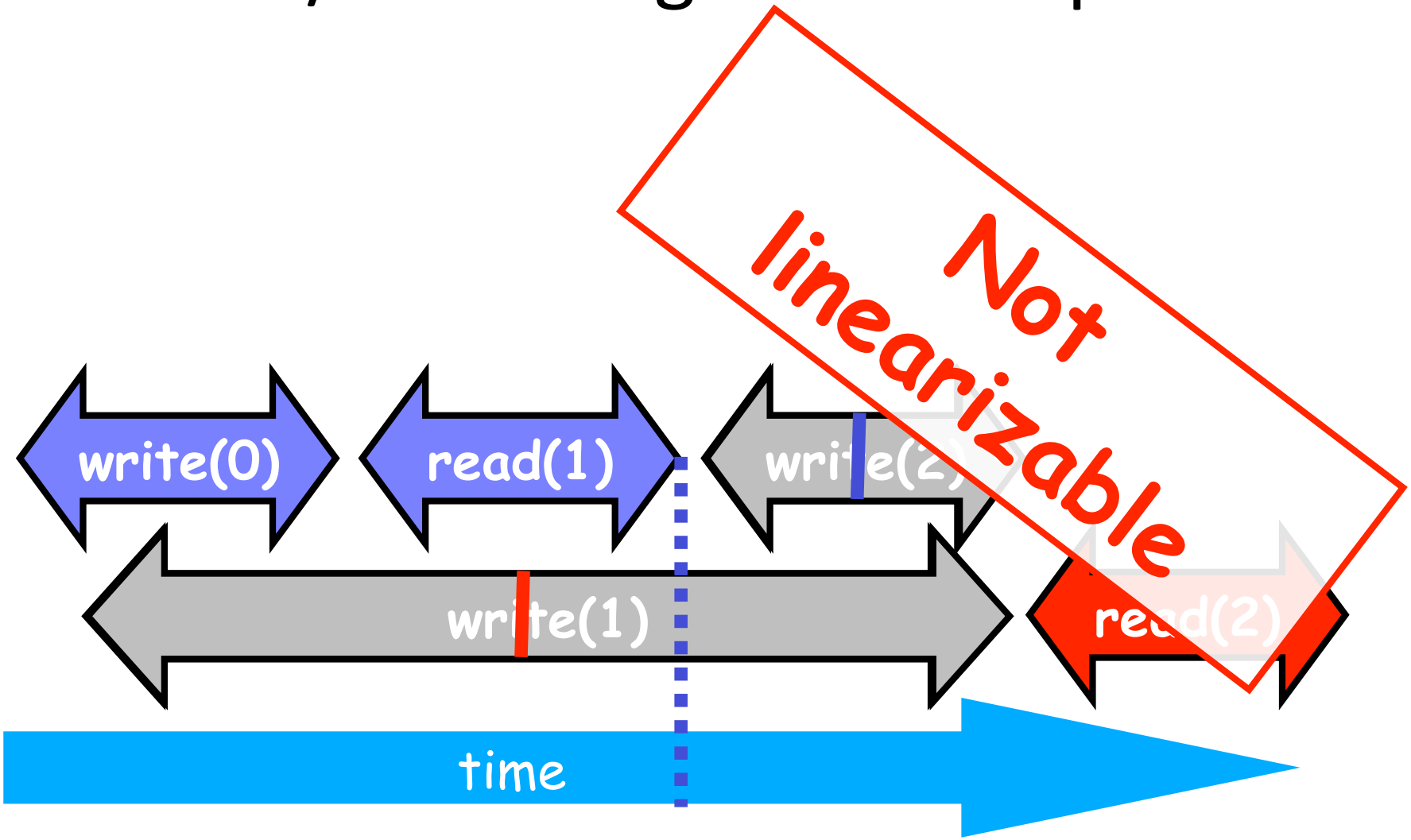
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Formal Model

- Goal: Build formal model of computations on shared objects
- So that we pin down concepts precisely
- And can prove properties about them
- Formalization:
  - Can be a goal by itself
  - Here: Tool for understanding
  - Do not go overboard!
- Our terminology slightly different from Herlihy/Shavit's
- You can use either
- Good to be clear about this *when doubt may arise*

States and Executions

Object state:

- Assignment to fields

Thread state:

- Assignment to local variables
- Assignment to program counter, operand stacks and activation record stacks

System state:

- State of all objects and threads

Execution:

- Sequences of system/thread states
- Can be finite or infinite – only finite for now
- Produced by running the program

Example

Initial state: head = 0, tail = 0, items array allocated + initialized

Producer and consumer threads initialized, program counters at initial states

Execution: Sequence of transitions from state to state

- Control in producer or consumer thread
- Inside or outside enq/deq methods

```
public class WaitFreeQueue {
  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

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

Histories/Traces

Executions produce histories – or traces – of events

How – next slides

Here:

behaviour(program *P*)

=

{ traces *t* | exists execution of *P* such that

t = trace(execution) }

Specification *Spec* = set of traces

Correctness of program *P*: *R*(behaviour(*P*),*Spec*)

Question is: What is *R*?

Events

This lecture: Events = method calls and returns

Call event – method invocation:

$A: x.m(a_1,\ldots,a_n)$

thread object method arguments

Normal return event:

$A: x.\text{ok}(r_1,\ldots,r_m)$

result values

No thread id given

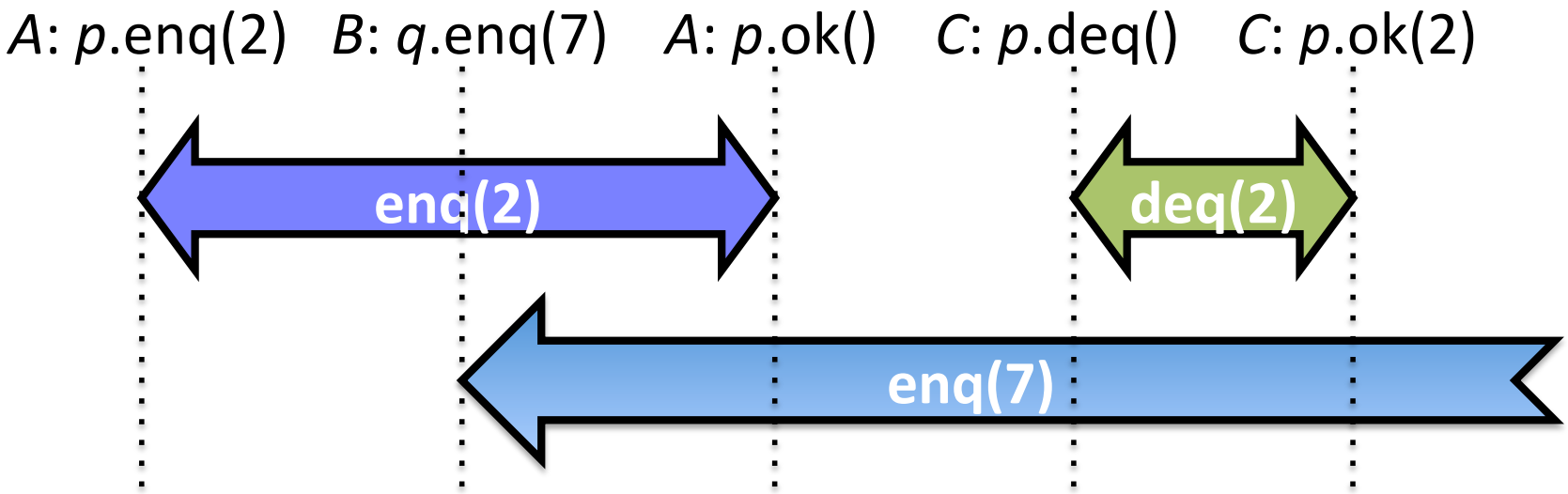
Exceptional return event:

$A: x.e(r_1,\ldots,r_m)$

exception

Traces - Matching Events

Trace: Finite or infinite sequence of events



Matching response:

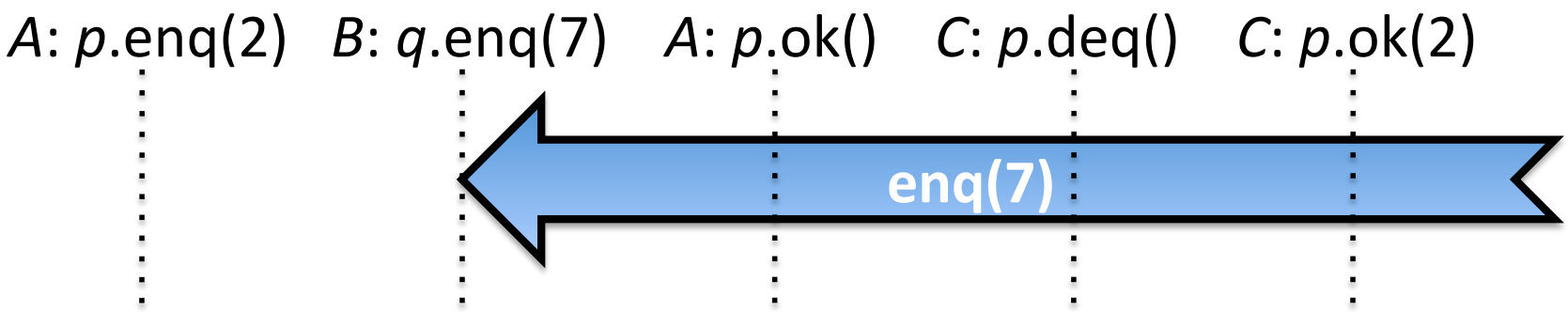
A: $x.m(a_1,\ldots,a_n)$ matches B: $y.\text{ok}(r_1,\ldots,r_m)$ iff

$A = B$ and $x = y$

Method call: Matching call-return pair – interval (lecture 1)

Complication – Incomplete Calls

Pending invocation



Pending: No matching response

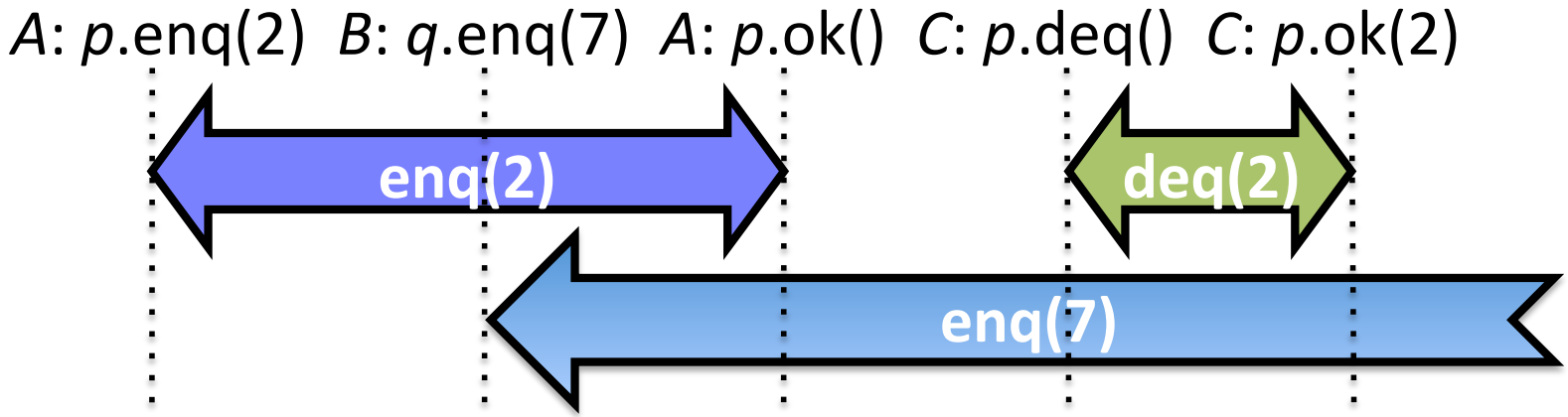
Extension of trace *H*:

- *H* followed by matching responses to unmatched calls in *H*

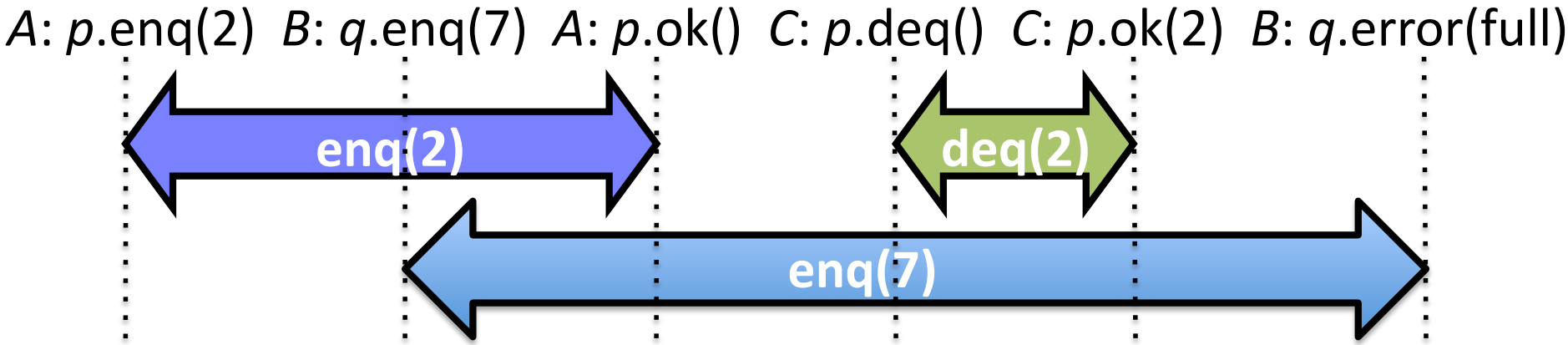
Complete(*H*):

- Subsequence of *H* with unmatched calls removed

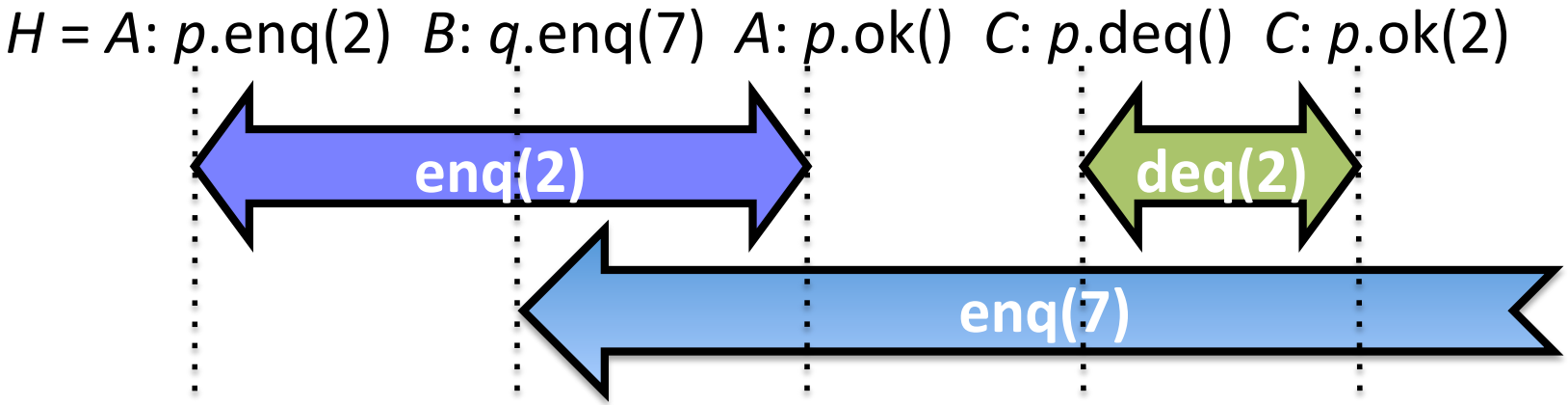
Example - Extension



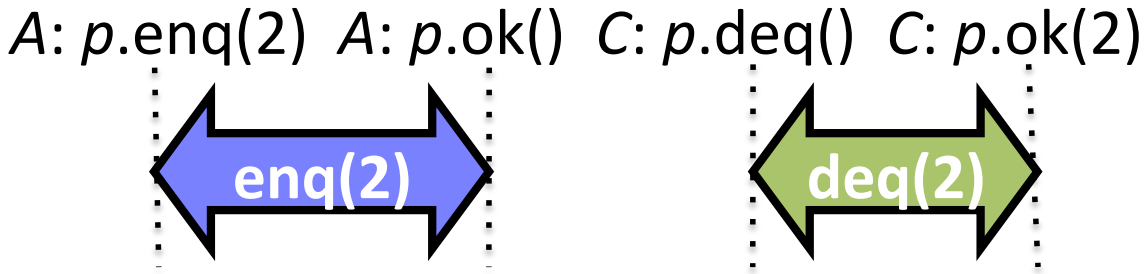
Extension:



Example - Completion



complete(*H*):



Subtraces and Equivalence

Thread subtrace $H|A$:

- Subsequence of H with all events not from A removed

$H = A$: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)

$H|A = A$: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)

Object subtrace $H|x$:

- Do. with all events not on x removed:

$H|p = A$: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)

Sequentiality

Sequential trace:

- First event is call. All calls immediately followed by matching response

  A: *p*.enq(2) *A*: *p*.ok() *C*: *p*.deq() *C*: *p*.ok(2)

Claim: Thread subtraces are always sequential

  – In the absence of recursion

Claim: Object subtraces may not be sequential. Why?

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

For a sequential multi-object trace *H*:

- *H* is *legal* iff
- *H*|*x* is in the sequential specification of *x*, for all *x*

Equivalence

Traces H_1 and H_2 are *equivalent* if $H_1|A = H_2|A$ for all A

Equivalence: Threads see the same subtraces in both

H_1 = A: p.enq(2) A: p.ok() C: p.deq() C: p.ok(2)

H_2 = A: p.enq(2) C: p.deq() A: p.ok() C: p.ok(2)

Condition:

- $H_1 \mid A = H_2 \mid A$
- $H_1 \mid B = H_2 \mid B$

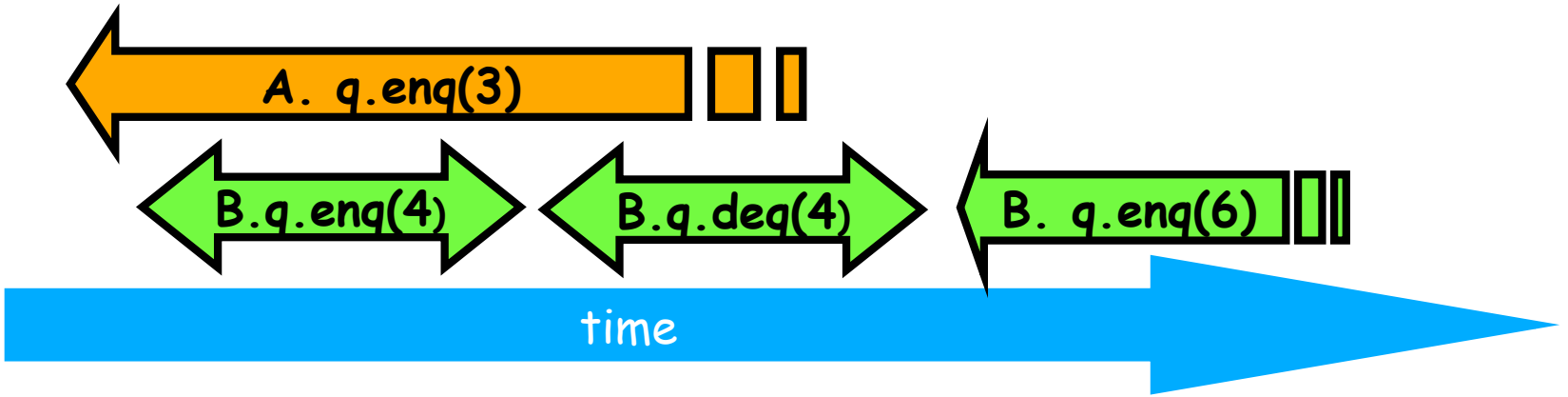
Linearizability

Definition: Trace *H* is linearizable if

- There is a legal sequential multi-object trace *S*
- *H* has an extension *H'*
  - (So pending calls can get response)
- *complete*(*H'*) is equivalent to *S*
  - (Not all pending calls need get a response)
- If m_0 -> m_1 in *H* then m_0 -> m_1 in *S*
  - Method calls that are sequential in *H* must appear in the right order
  - Recall the interval ordering -> from last lecture

Example

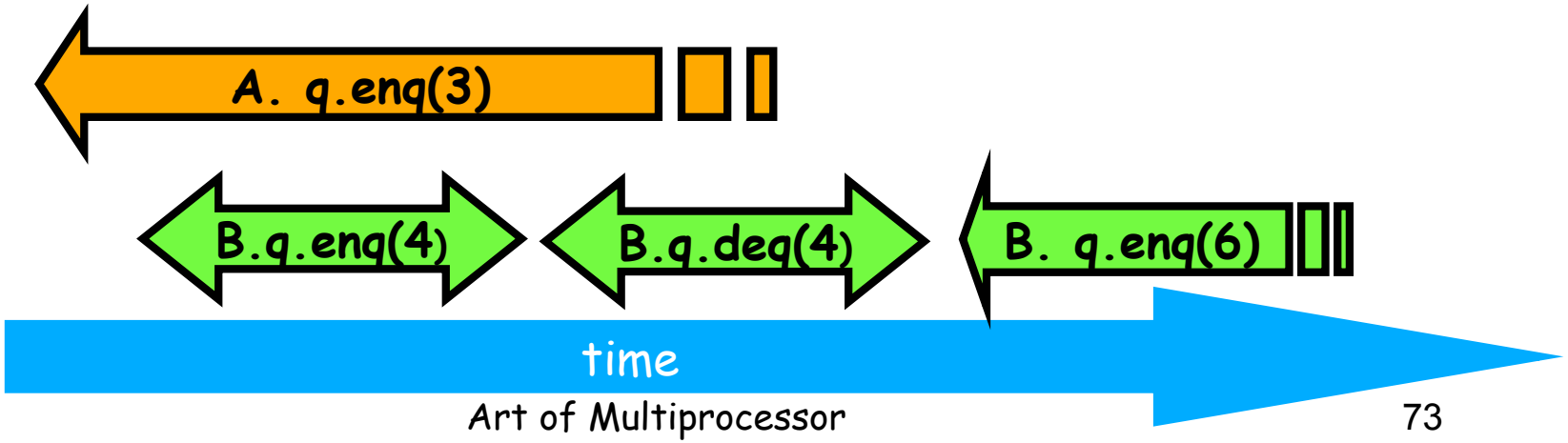
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



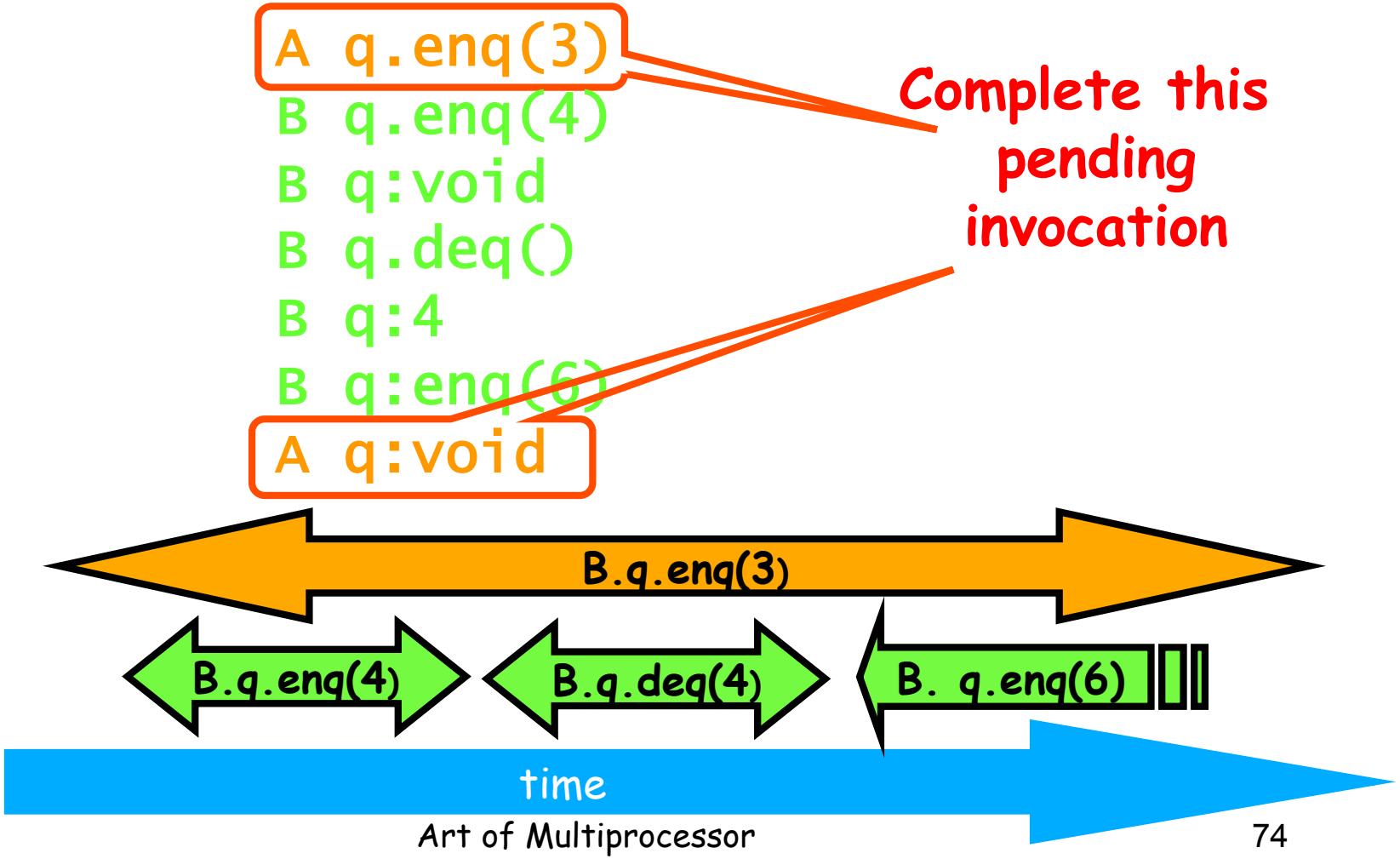
Example

```
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



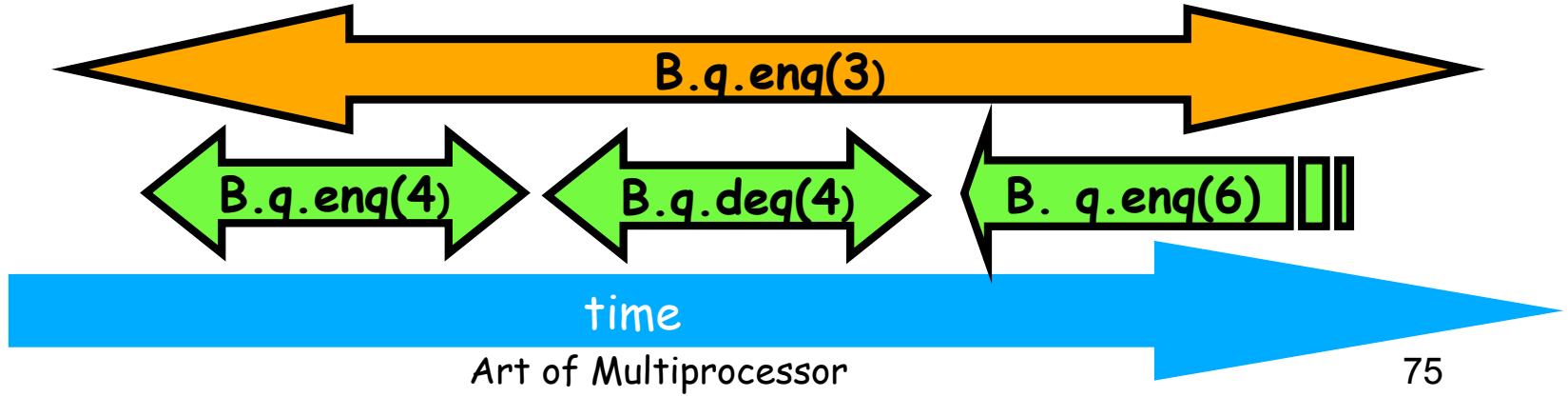
Example


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
B.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

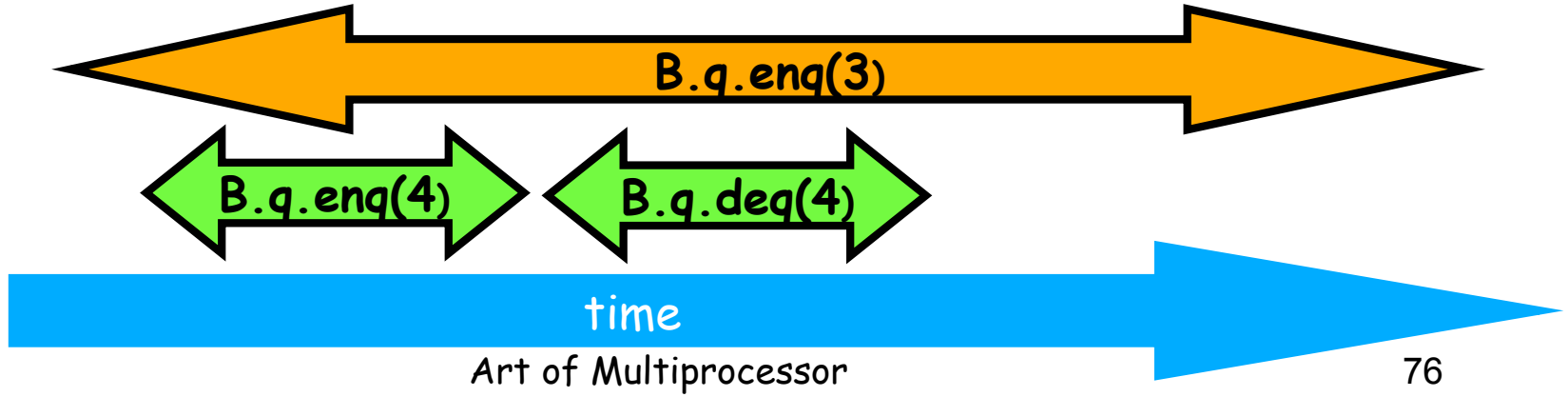


Example

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

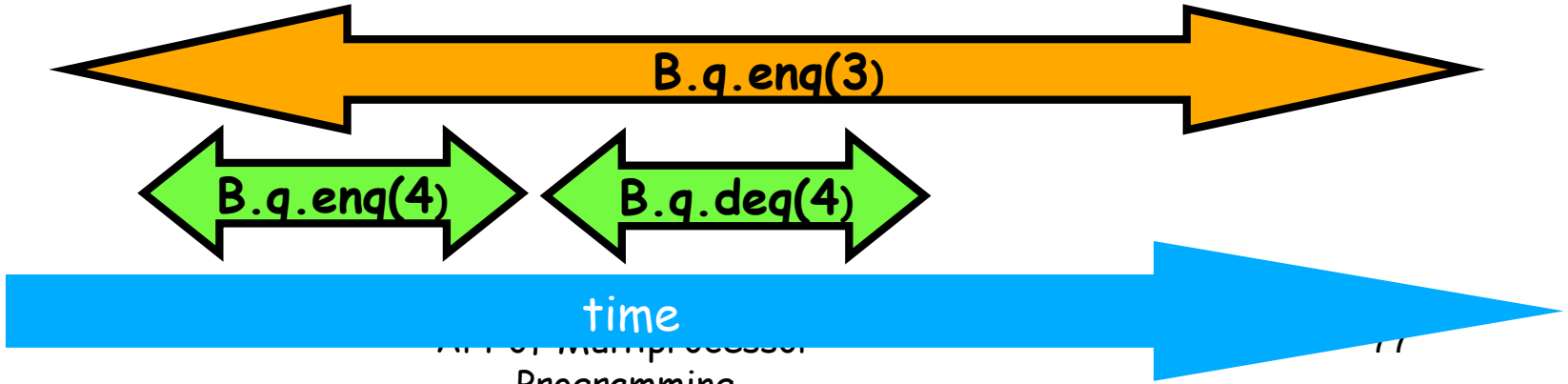
discard this one

A q:void



Example

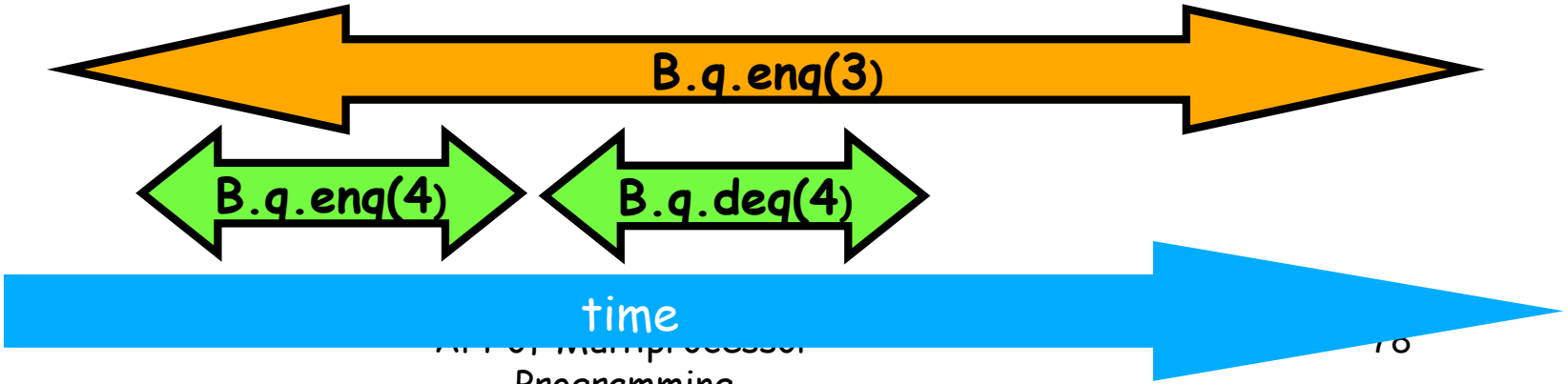
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



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
```

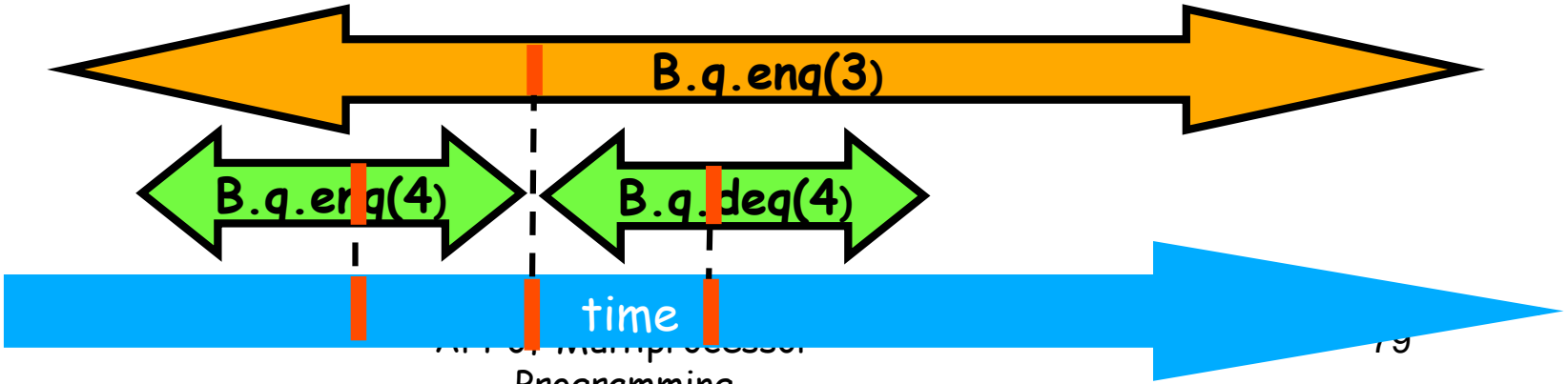


Example

```
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
```



Composability

Theorem: *H* is linearizable iff for each object *x*, *H* | *x* is linearizable

Proof: See Herlihy-Shavit

This is important!

- Linearizable objects => linearizable systems
- Otherwise might have to impose extra scheduler or locks to ensure linearizability at system level

Proving Linearizability

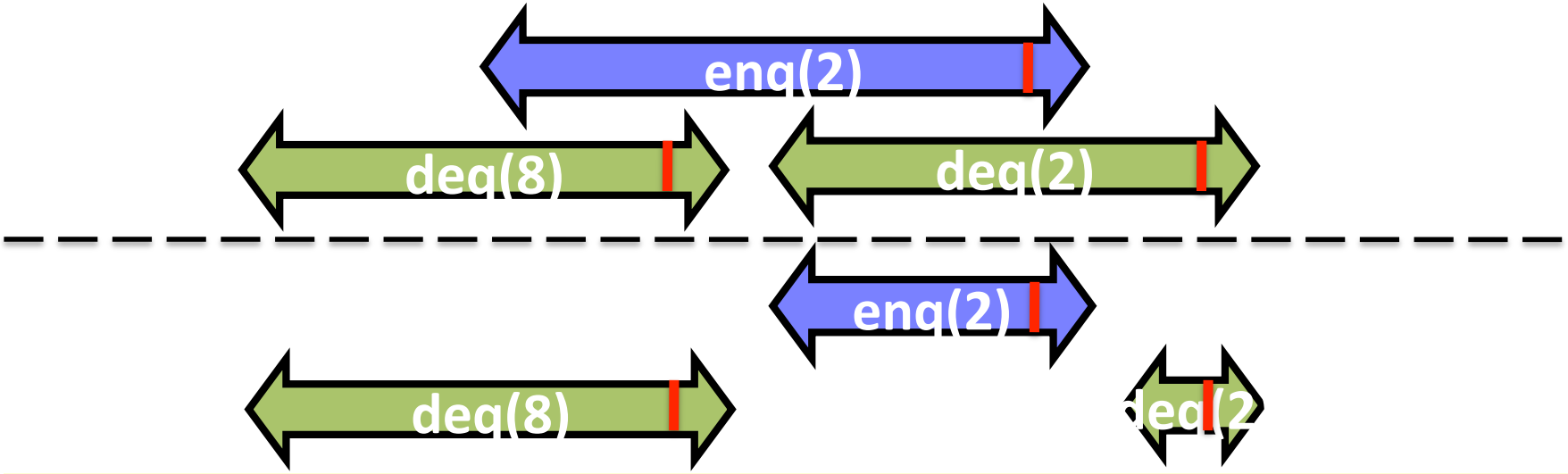
Recall: Trace *H* is linearizable if

- There is a legal sequential multi-object trace *S*
- *H* has an extension *H'*
- *complete*(*H'*) is equivalent to *S*
- If m_0 -> m_1 in *H* then m_0 -> m_1 in *S*

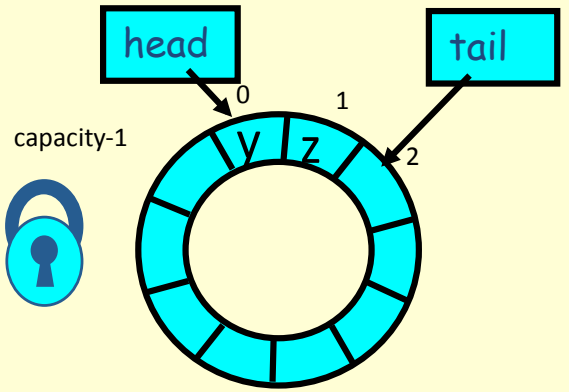
Strategy:

- Consider each object in turn – ok by composability
- Identify a single linearization point in each method
  - (Not always possible, alas)
- Show that all method events can be "shifted" to the linearization point

Proving Linearizability – with Locking




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



Linearization point

The Lock-free Case

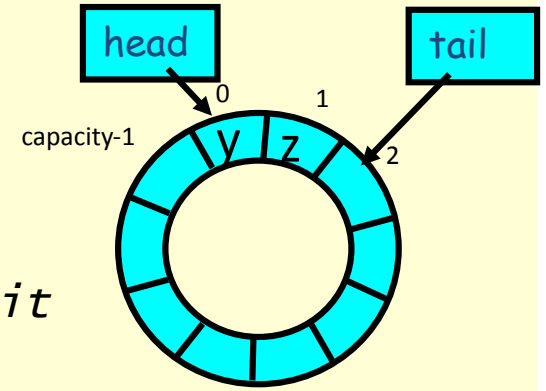
Linearization points where tail/head updated

Remember: Only one producer and one consumer!

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

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



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

Definition: Trace *H* is sequentially consistent if

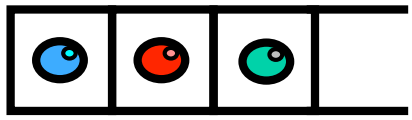
- There is a legal sequential multi-object trace *S*
- *H* has an extension *H'*
- *complete*(*H'*) is equivalent to *S*
- ~~If m_0 -> m_1 in *H* then m_0 -> m_1 in *S*~~

Not necessary to preserve precedence order between threads

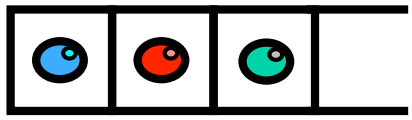
But program order must be preserved !

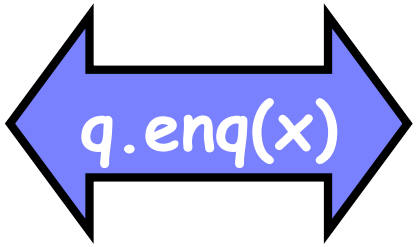
Program order: Per thread precedence order - equivalence

Example: Sequential Consistency

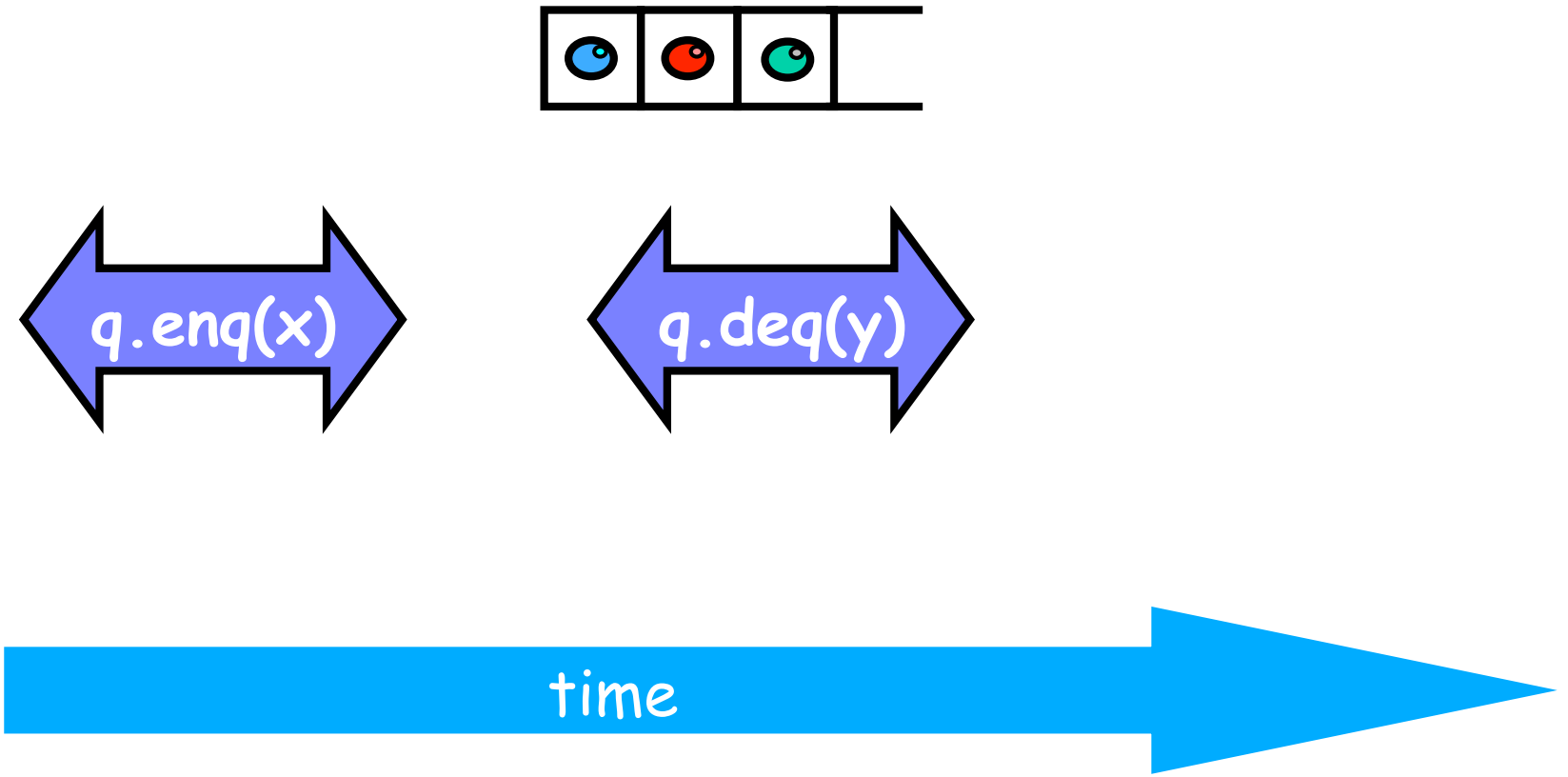


Example: Sequential Consistency

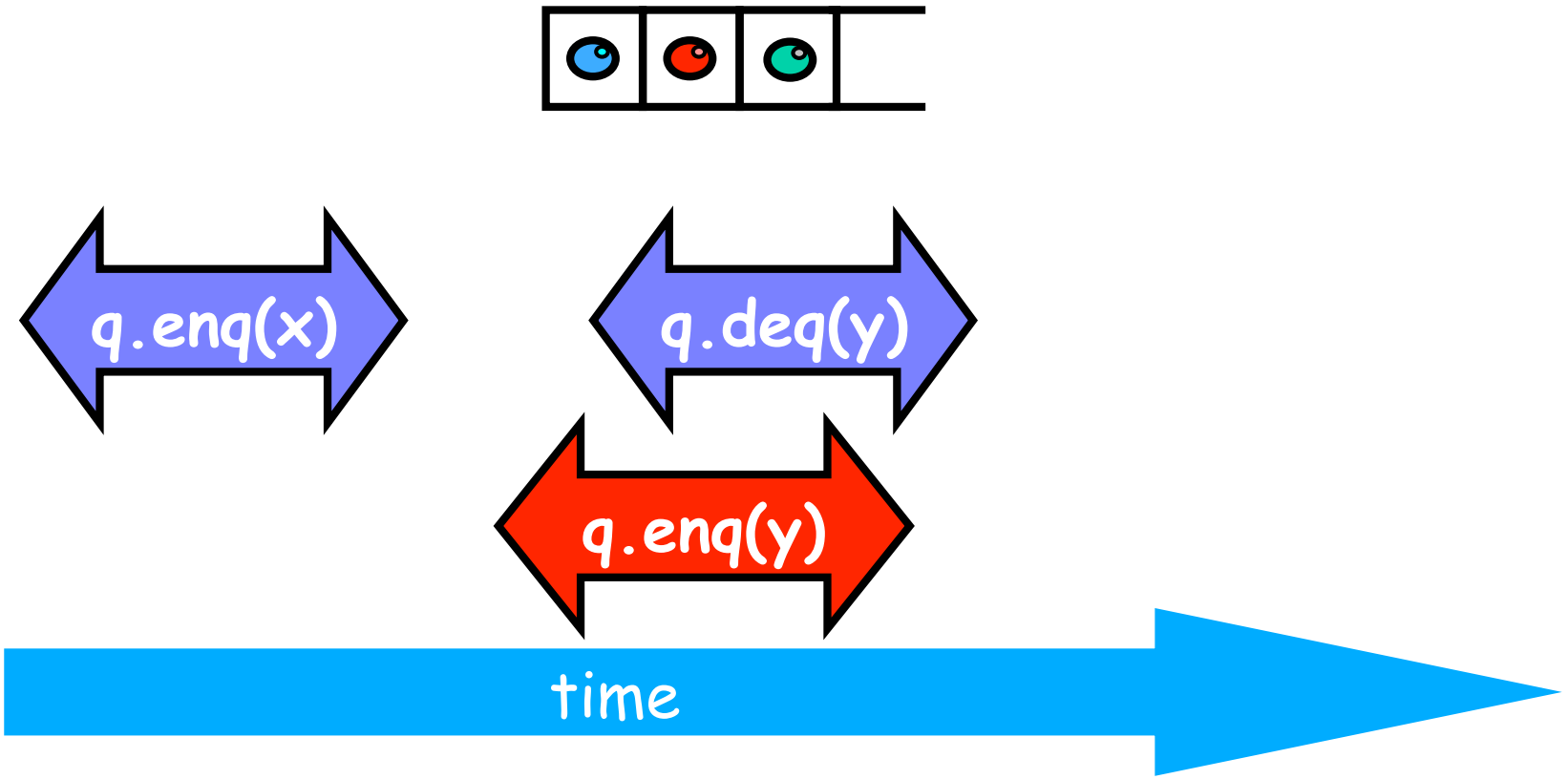




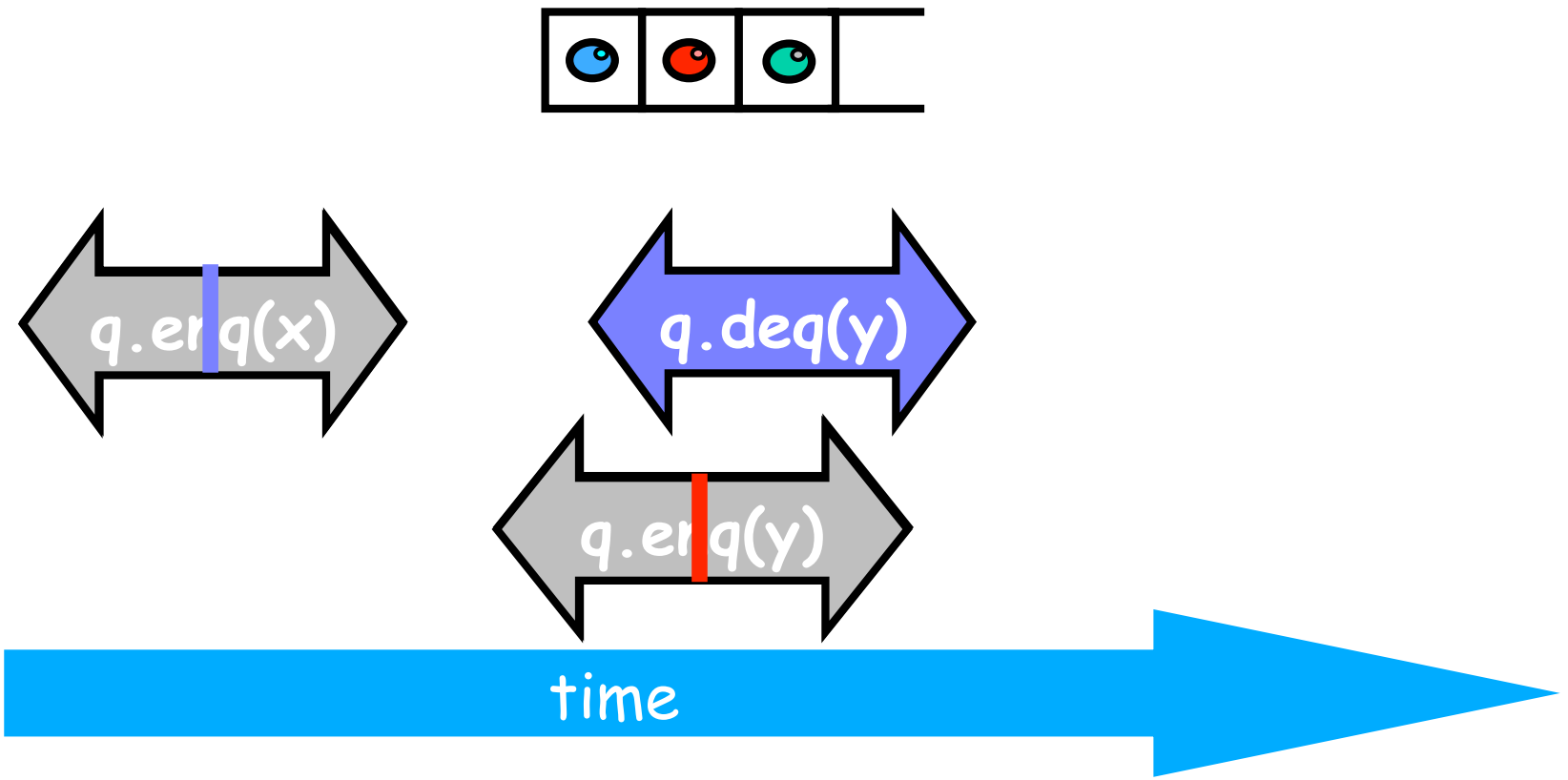
Example: Sequential Consistency



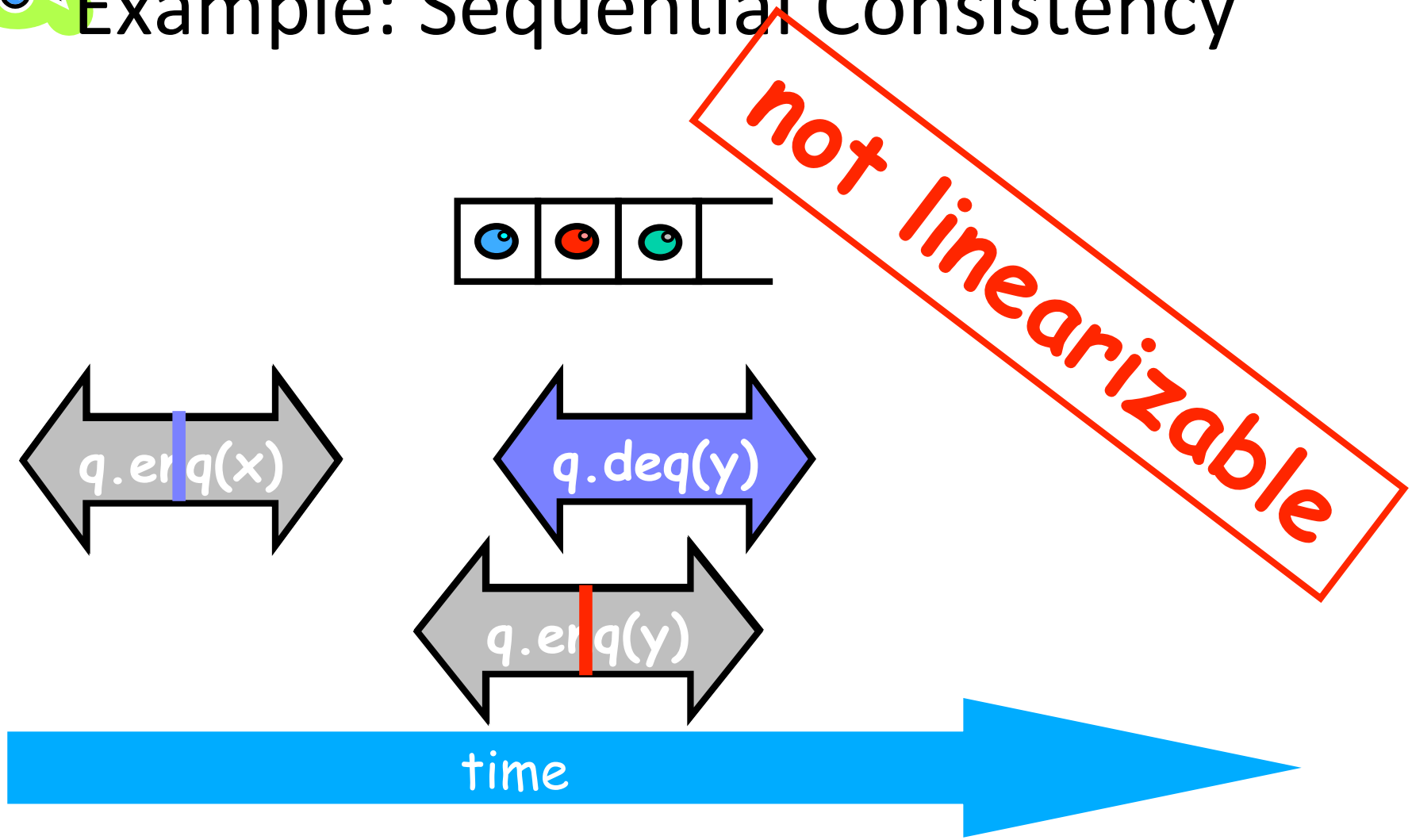
Example: Sequential Consistency



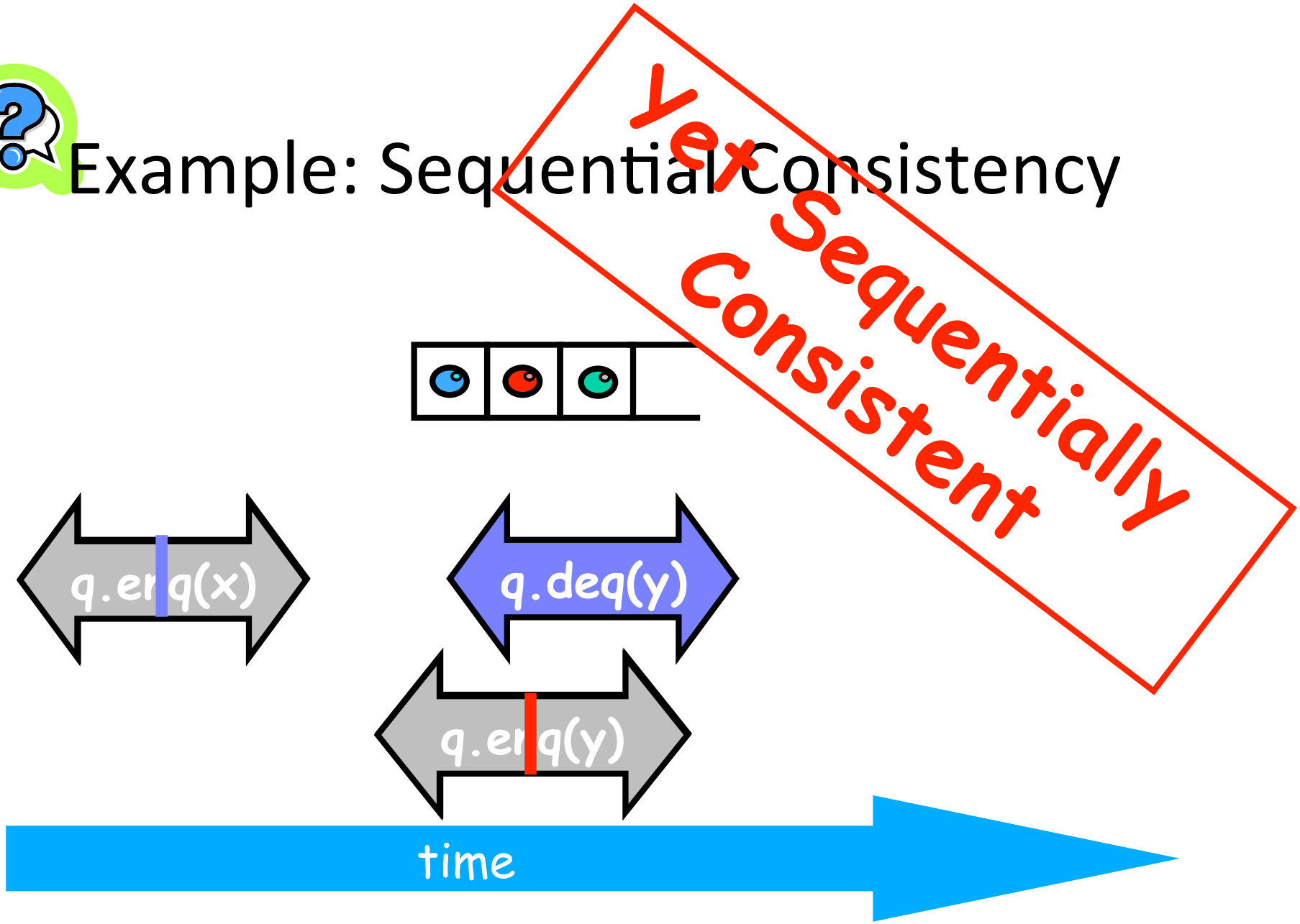
Example: Sequential Consistency



Example: Sequential Consistency



Example: Sequential Consistency

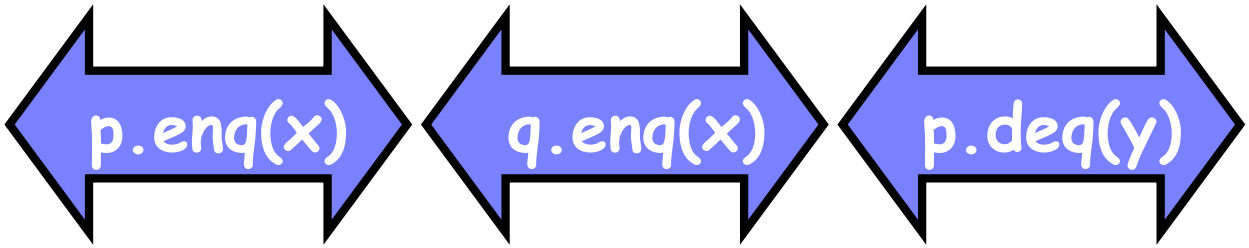


Theorem

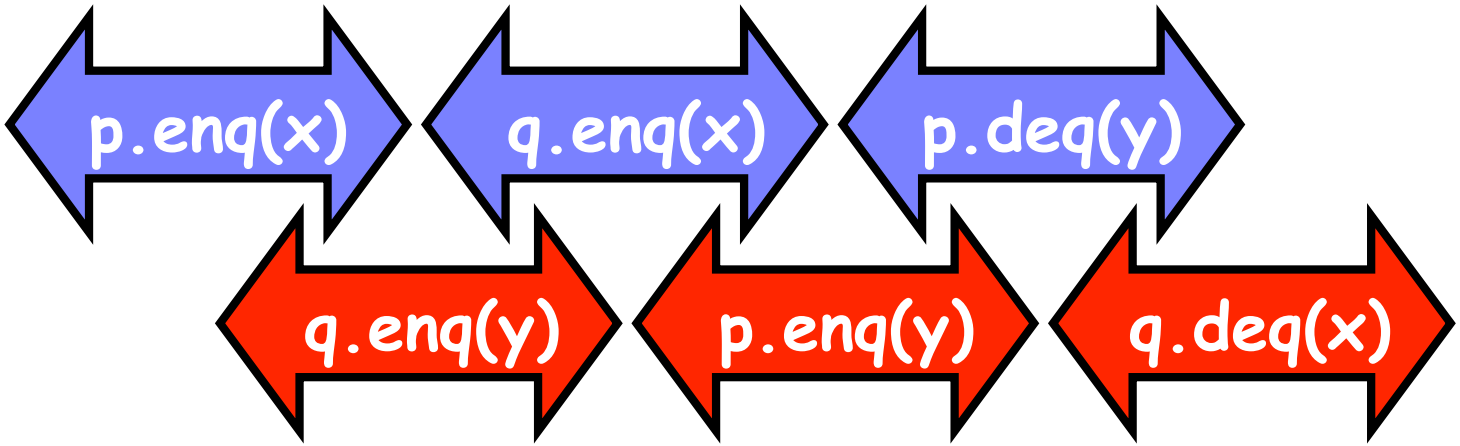
Sequential Consistency is not a local property

(and thus we lose composability…)

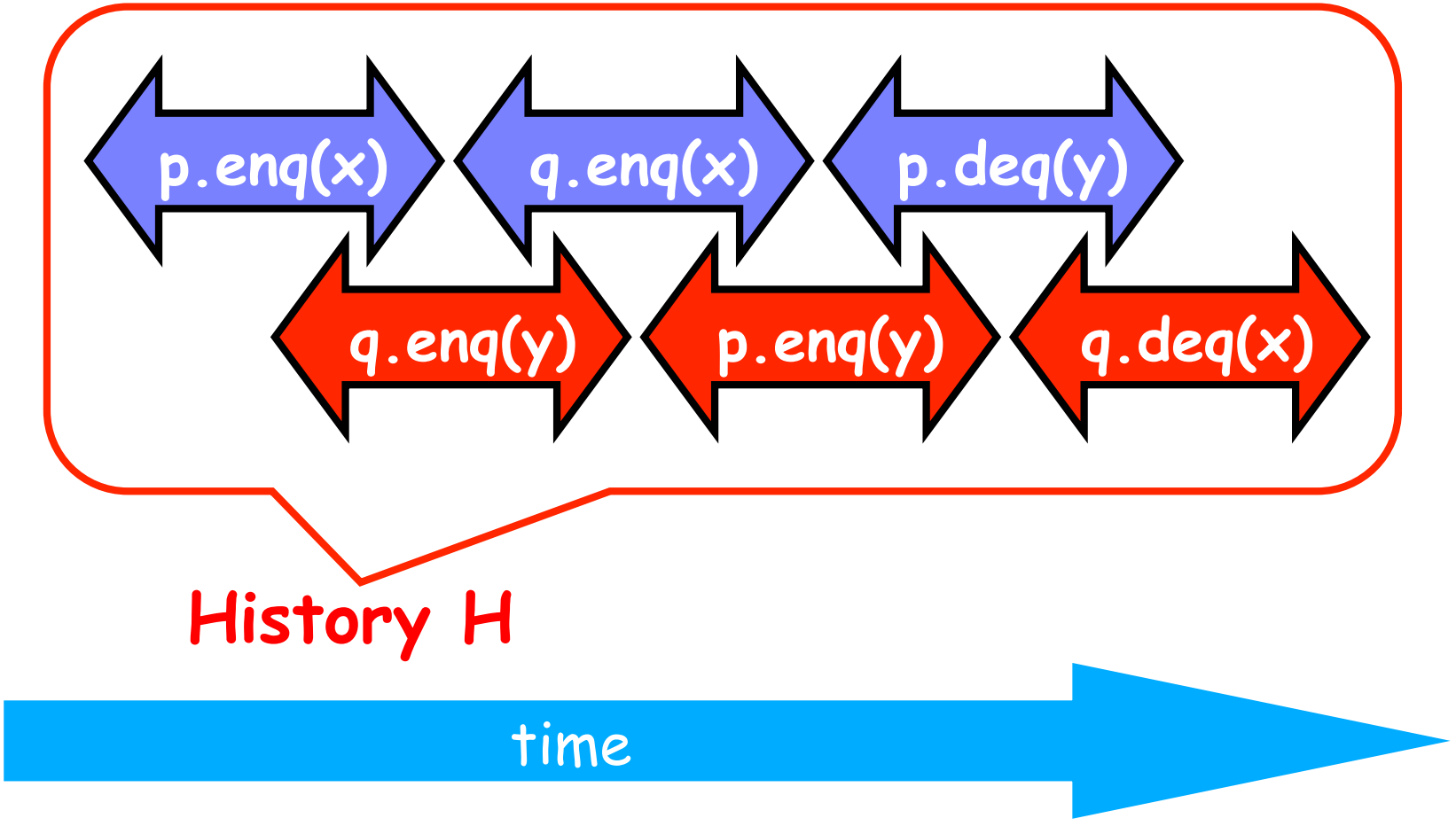
FIFO Queue Example



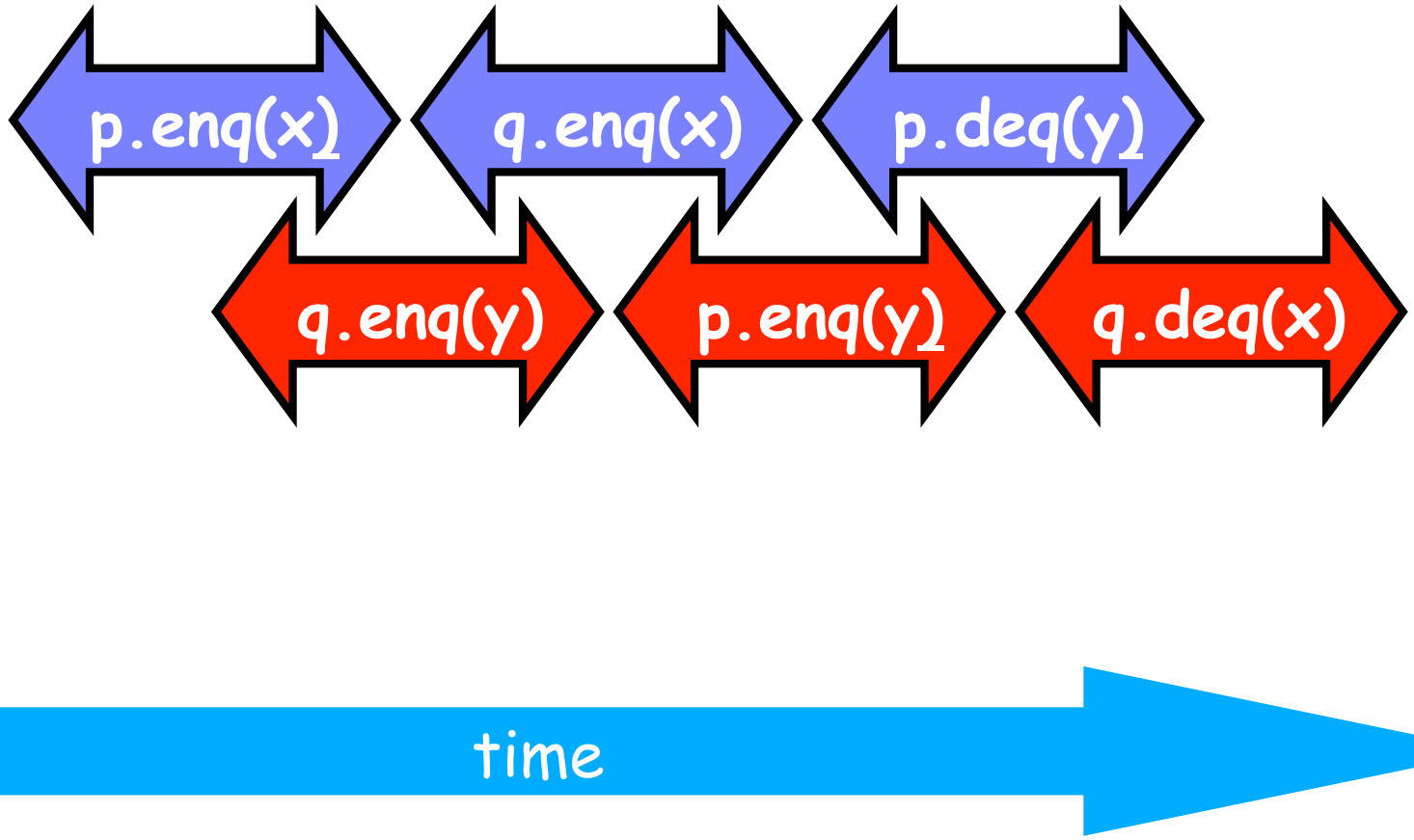
FIFO Queue Example



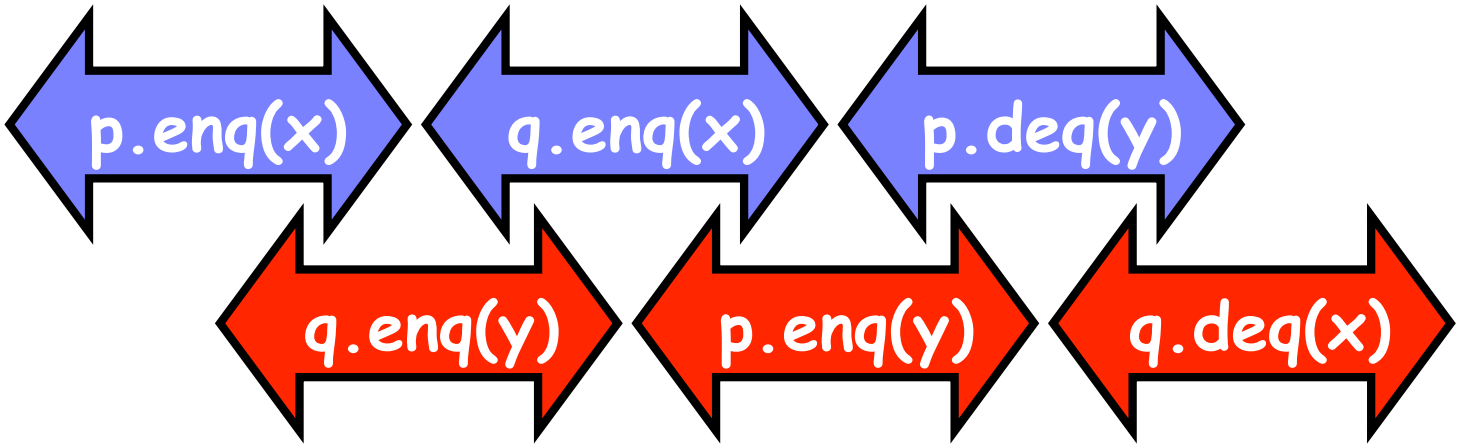
FIFO Queue Example



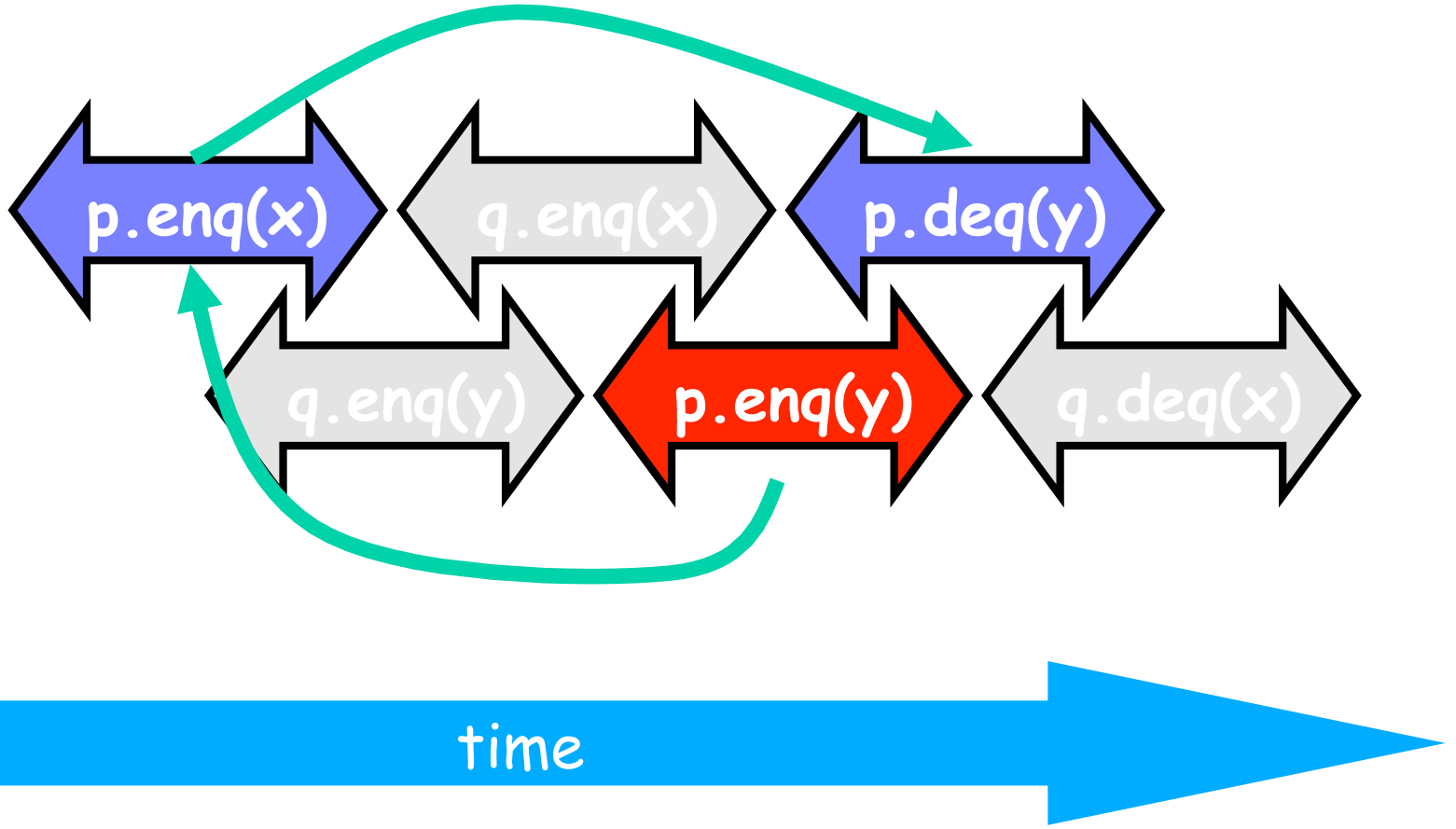
H|p Sequentially Consistent



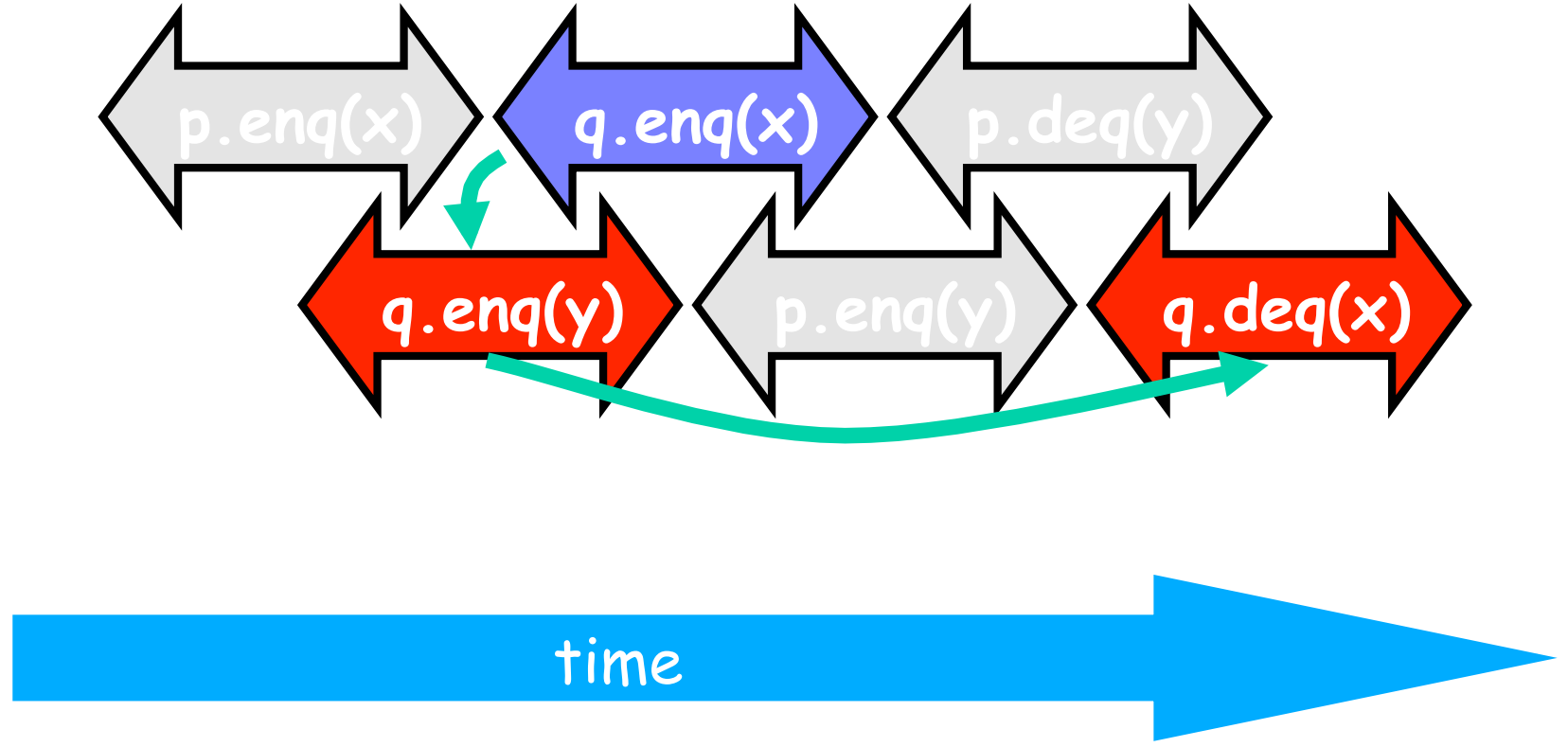
H|q Sequentially Consistent



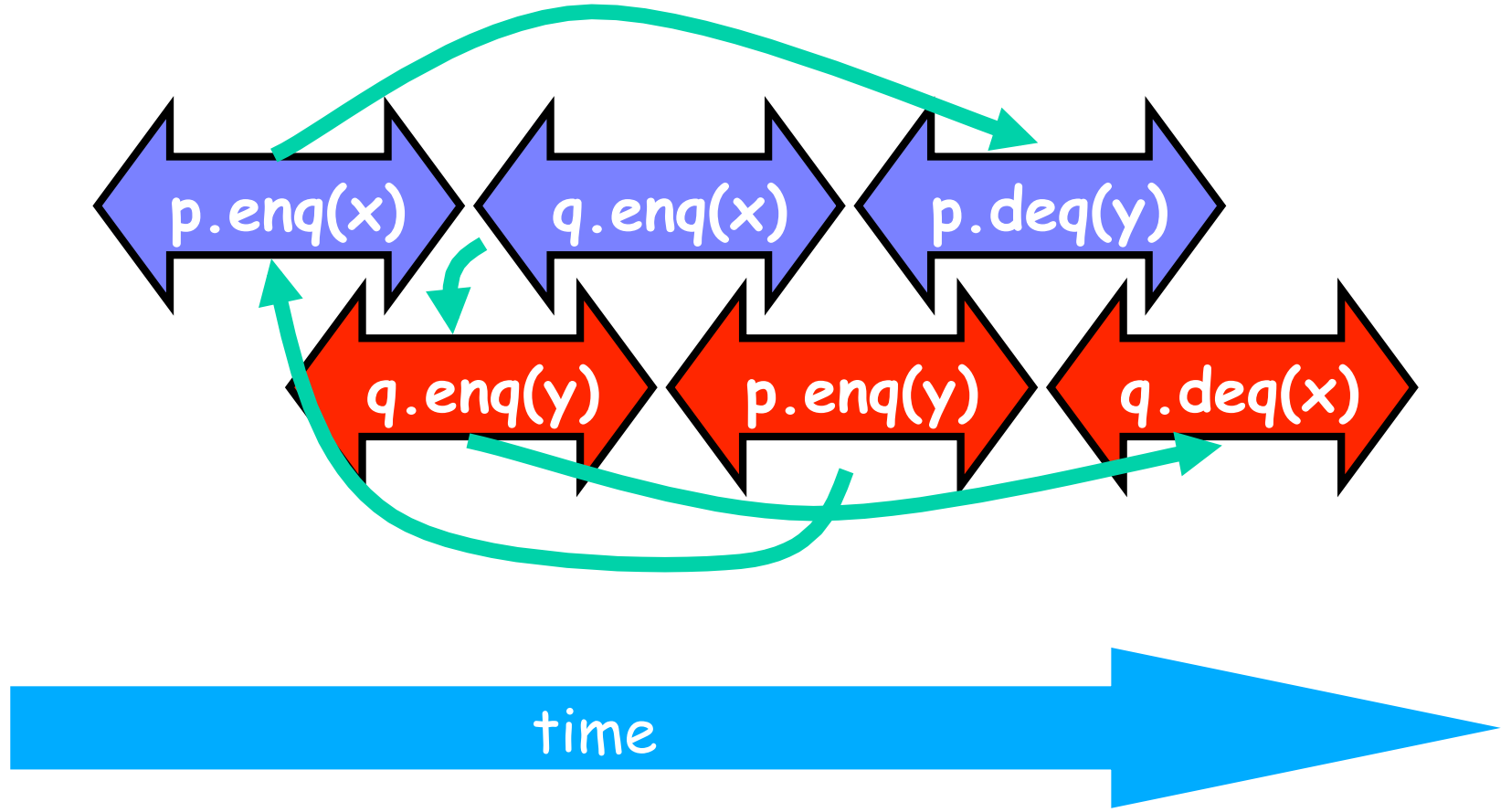
Ordering imposed by p



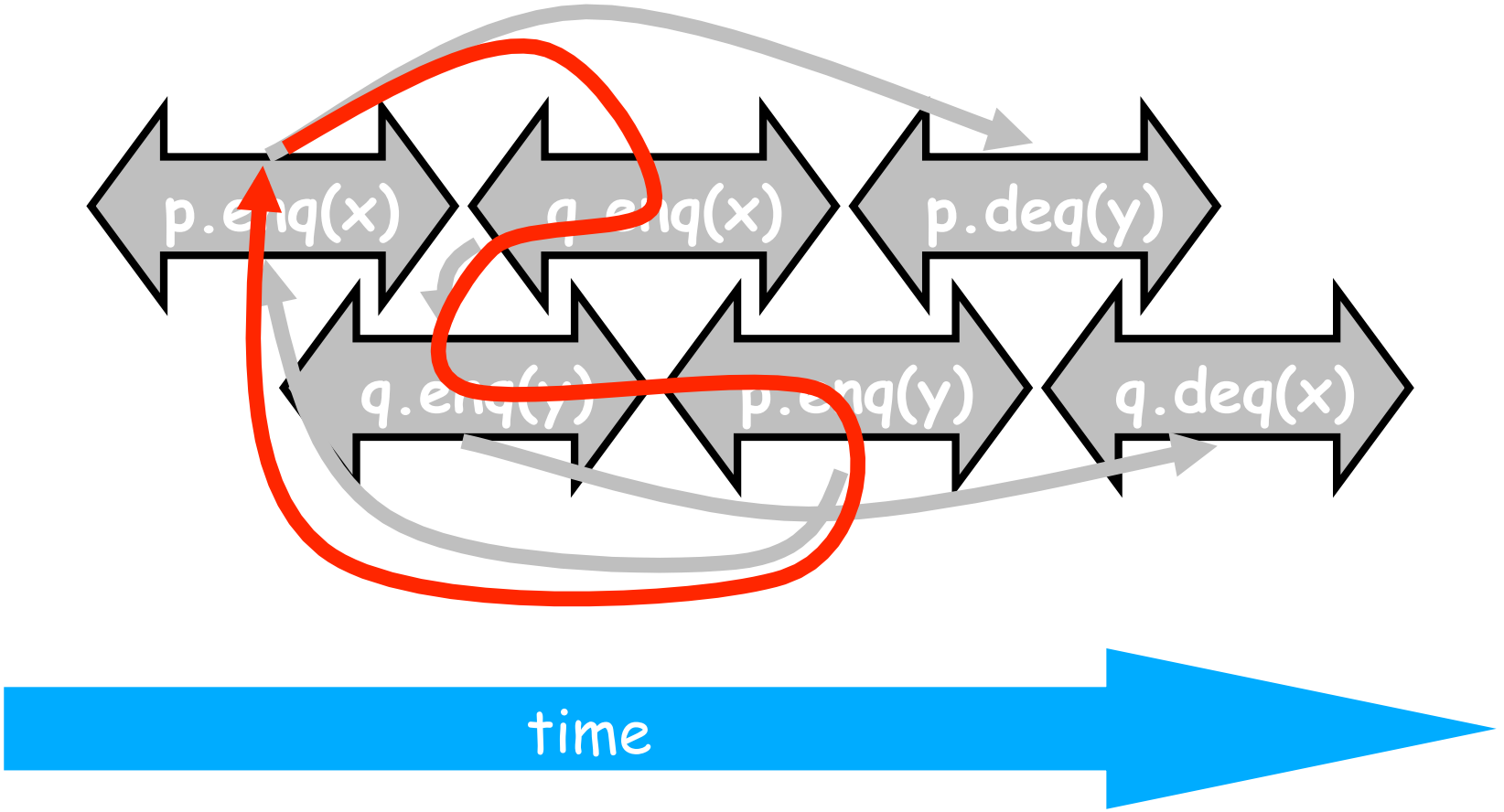
Ordering imposed by q



Ordering imposed by both



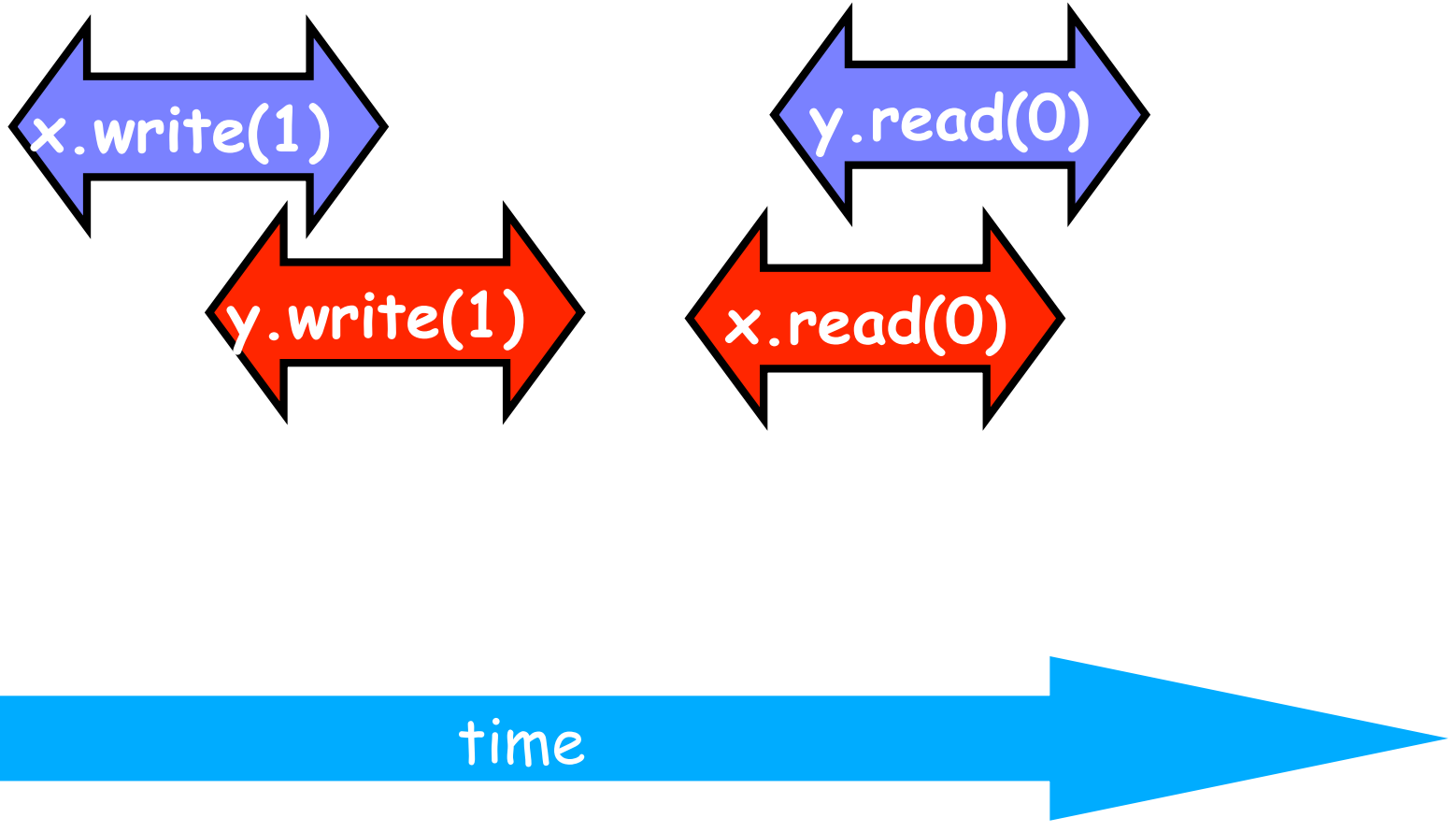
Combining orders



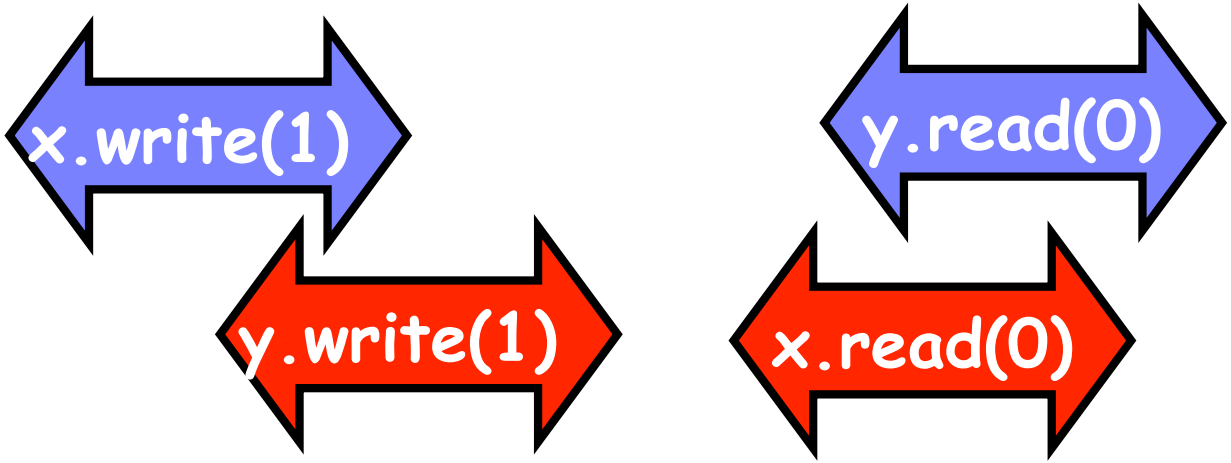
Fact

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

Example



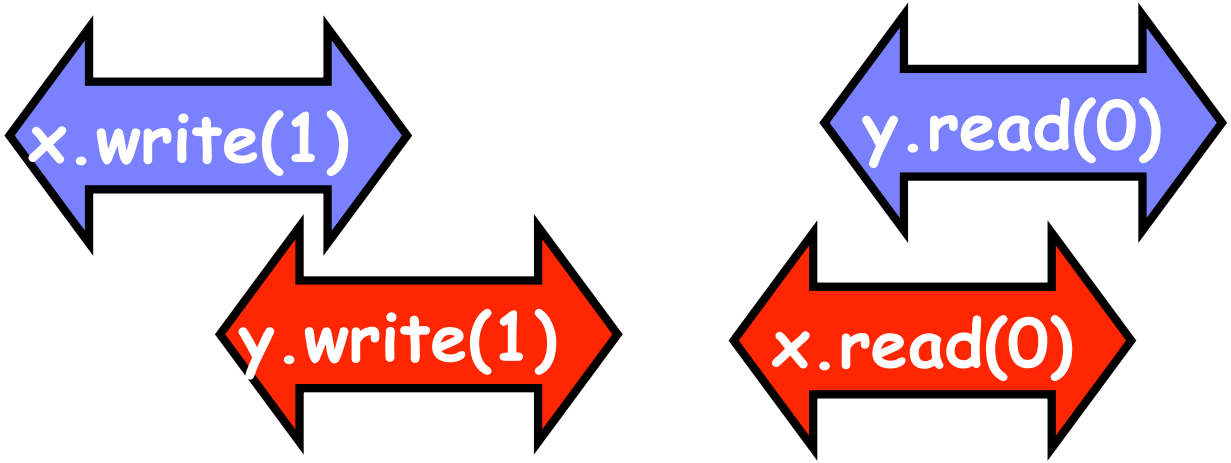
Example



Each thread's view is sequentially consistent

– It went first

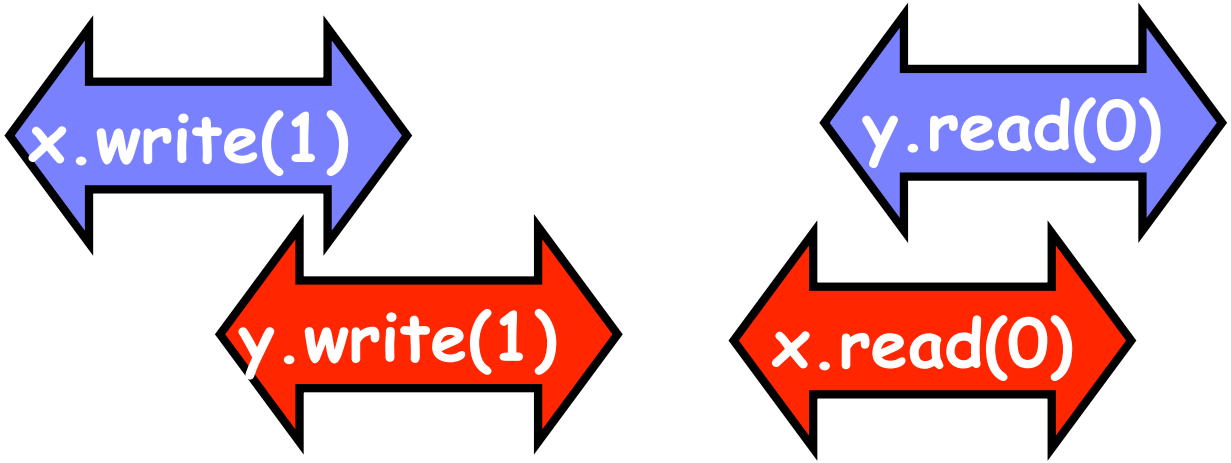
Example



Entire history isn't sequentially consistent

- Can't both go first
- Highly counterintuitive

Example

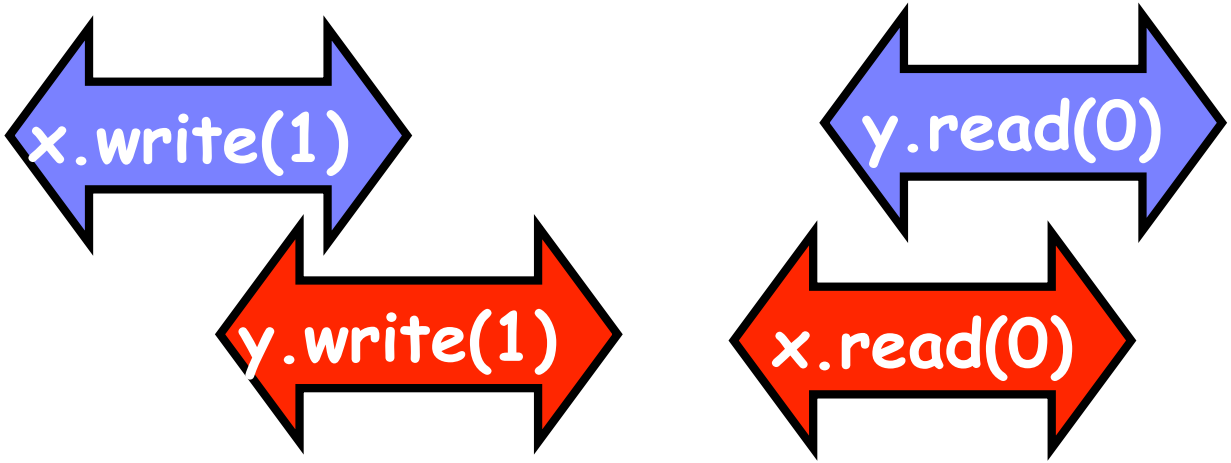


But what if each thread executes on its own core?

Read and writes address cores own cache

Writes and reads to main memory buffered

Example

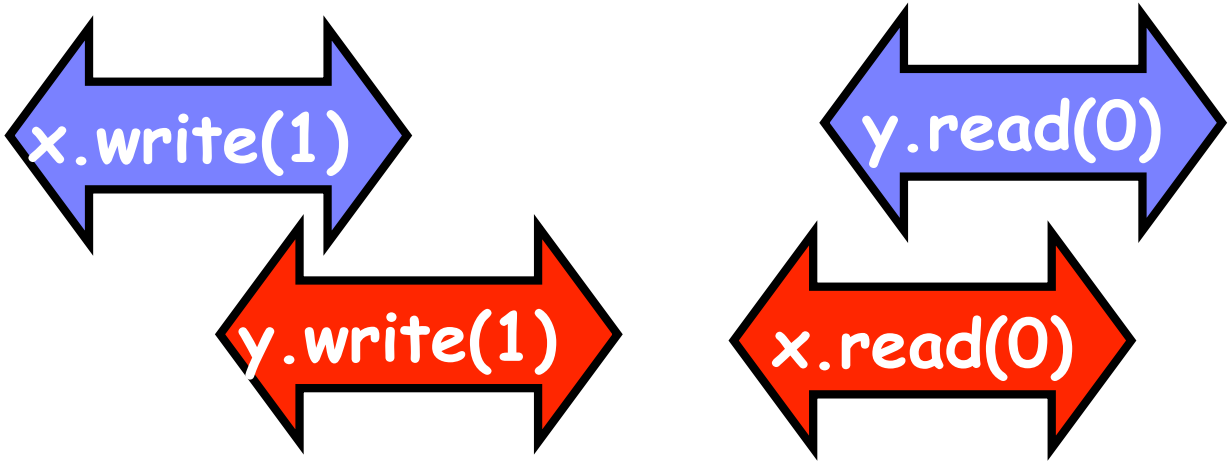


But what if each thread executes on its own core?

Read and writes address cores own cache

Writes and reads to main memory buffered

Example



Is this behavior wrong?

– We can argue either way …

The Hardware View

- Many hardware architects think that sequential consistency is too strong
- Too expensive to implement in modern hardware
- OK if fiction of sequentiality
  - 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.
- 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
- Processors delay writing to memory
- Until reads have been issued
- Otherwise get severe performance degradation
- If you need to synchronize, say so

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
- 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 experts, tricky stuff
  - assembly language, device drivers, etc.
- Linearizability more appropriate for high-level software

Progress Conditions

- *Deadlock-free:* some thread trying to acquire the lock eventually succeeds.
- *Starvation-free:* every thread trying to acquire the lock eventually succeeds.
- *Lock-free:* some thread calling a method eventually returns.
- *Wait-free:* every thread calling a method eventually returns.

	Non-Blocking	**Blocking**
Everyone makes progress	**Wait-free**	**Starvation-free**
Someone makes progress	**Lock-free**	**Deadlock-free**

Summary

- Concurrent access to objects need thought
- Linearizability:
  - Useful high-level model
  - Local and composable
- Locks: Easy way to obtain linearizability
  - But expensive
  - Deadlock freedom and starvation freedom can be obtained
- Can do better without locks
  - Trickier, but can get better progress properties
- For better performance yet need to use weaker memory models
  - Even trickier