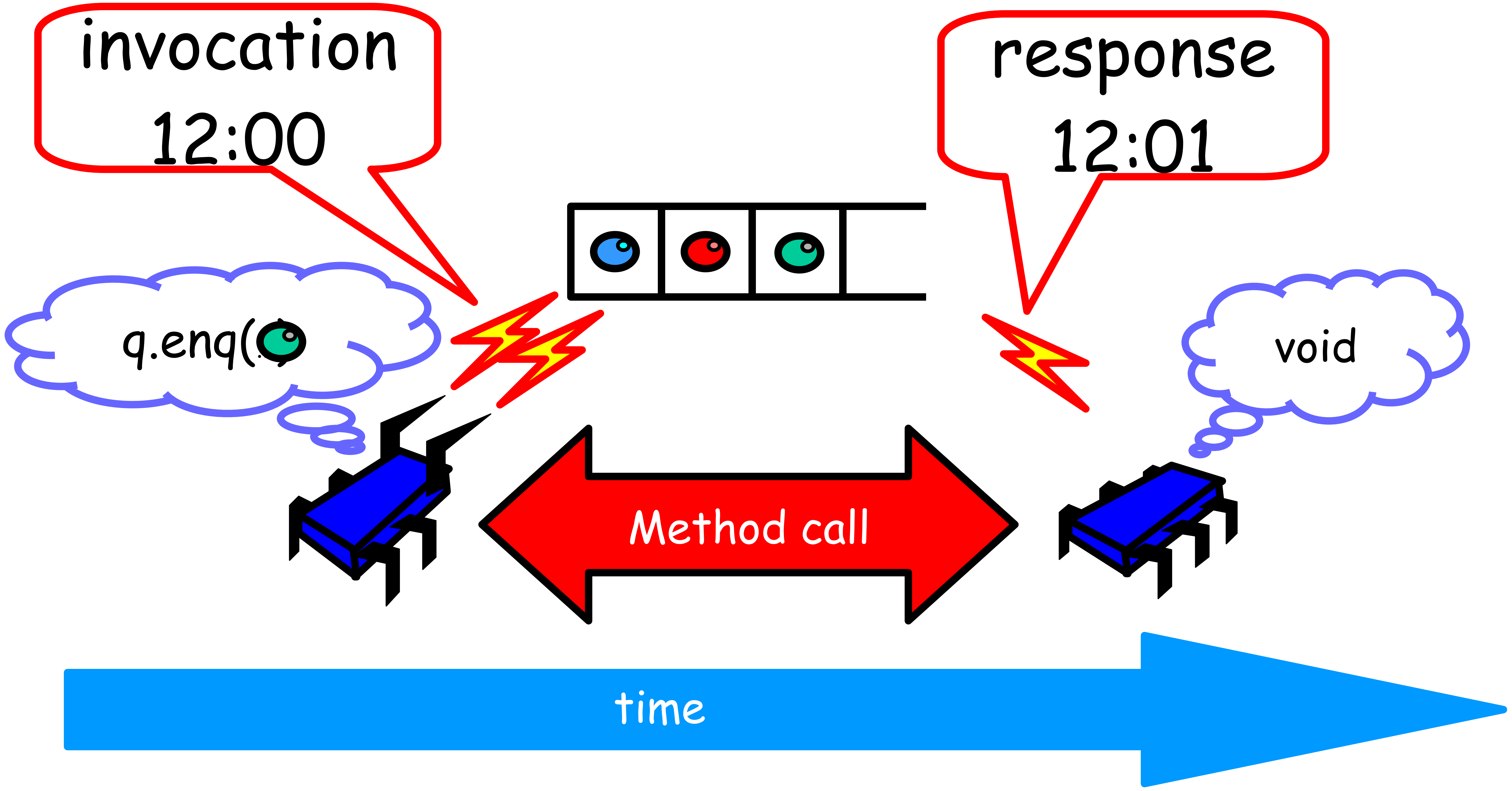
Concurrency in Java

CS 475, Spring 2019
Concurrent & Distributed Systems

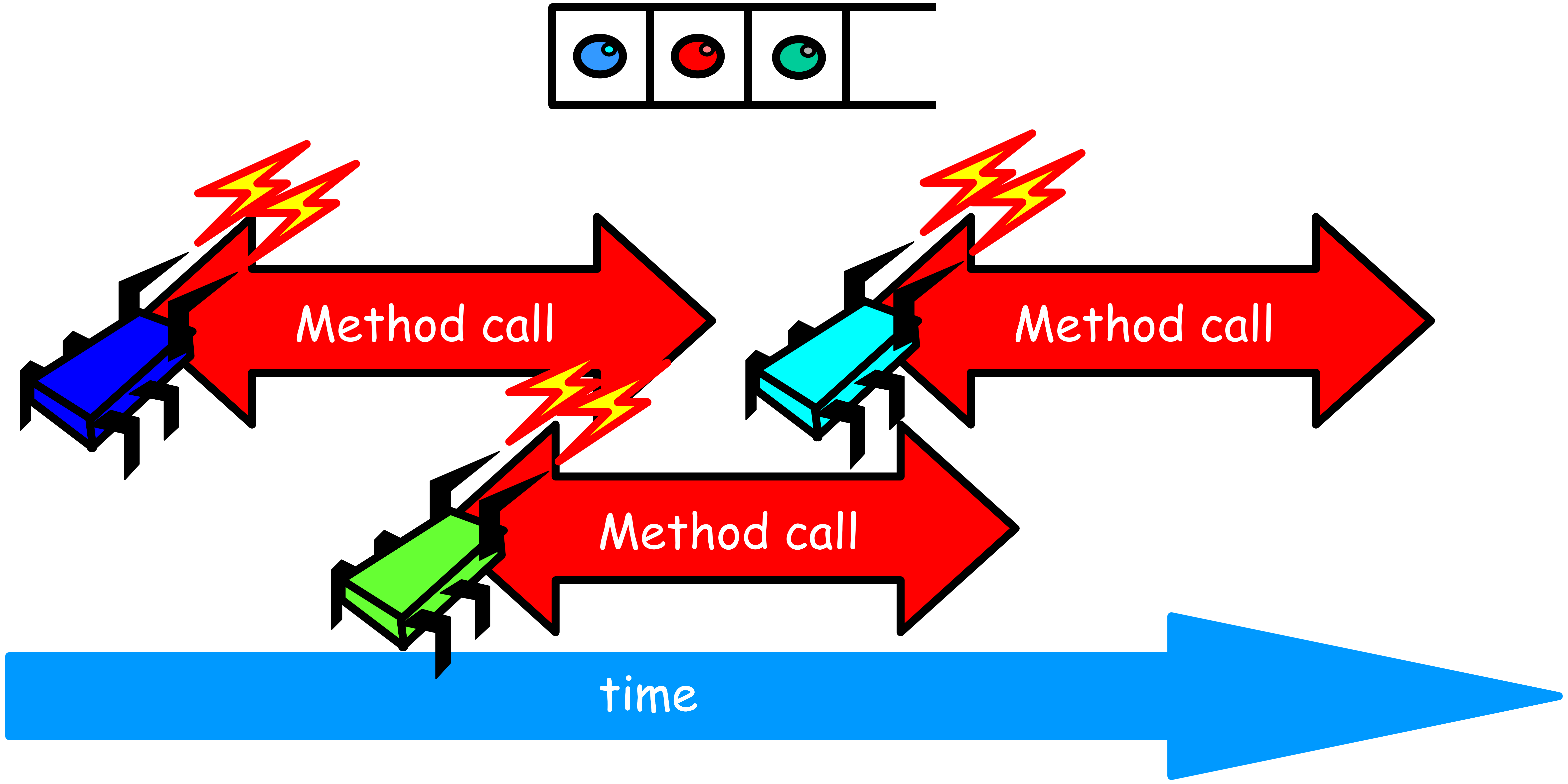
GEORGE MASON UNIVERSITY

With material from Herlihy & Shavit, Art of Multiprocessor Programming

Methods Take Time

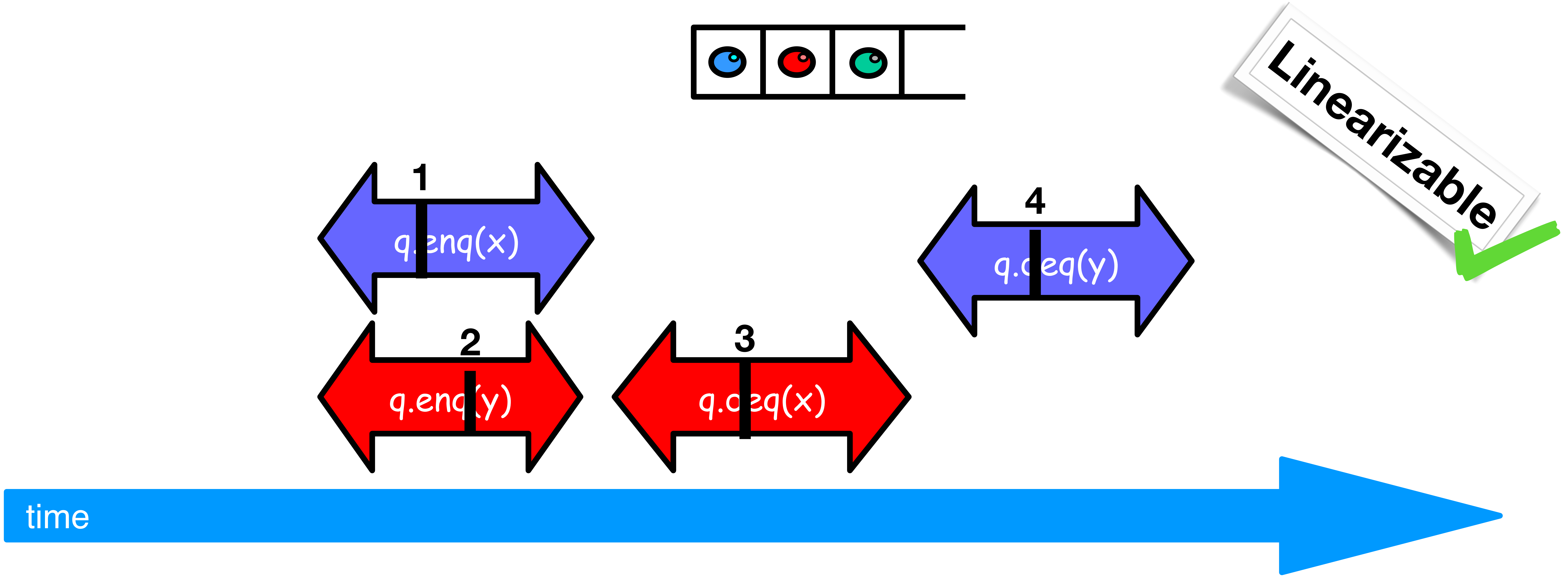


Concurrent Methods Take Overlapping Time



Example: Linearizable?

Reminder: Linearizable means: each method takes effect instantaneously, sometime in its observed time window



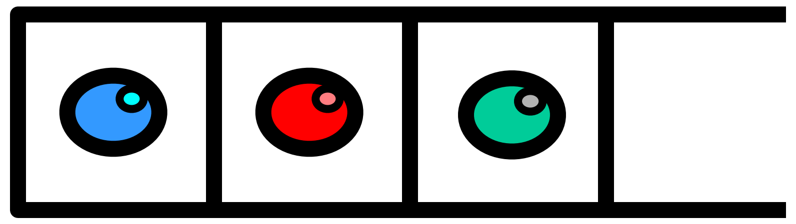
Today

- From our model of strict happens-before to… how programming languages really work
- Plus condition variables, monitors and semaphores
- Reading: H&S 3.8, 8.1-8.5
- Note: HW1 on Monday! https://www.jonbell.net/gmu-cs-475-spring-2019/homework-1/
  - Several new hints/updates this week

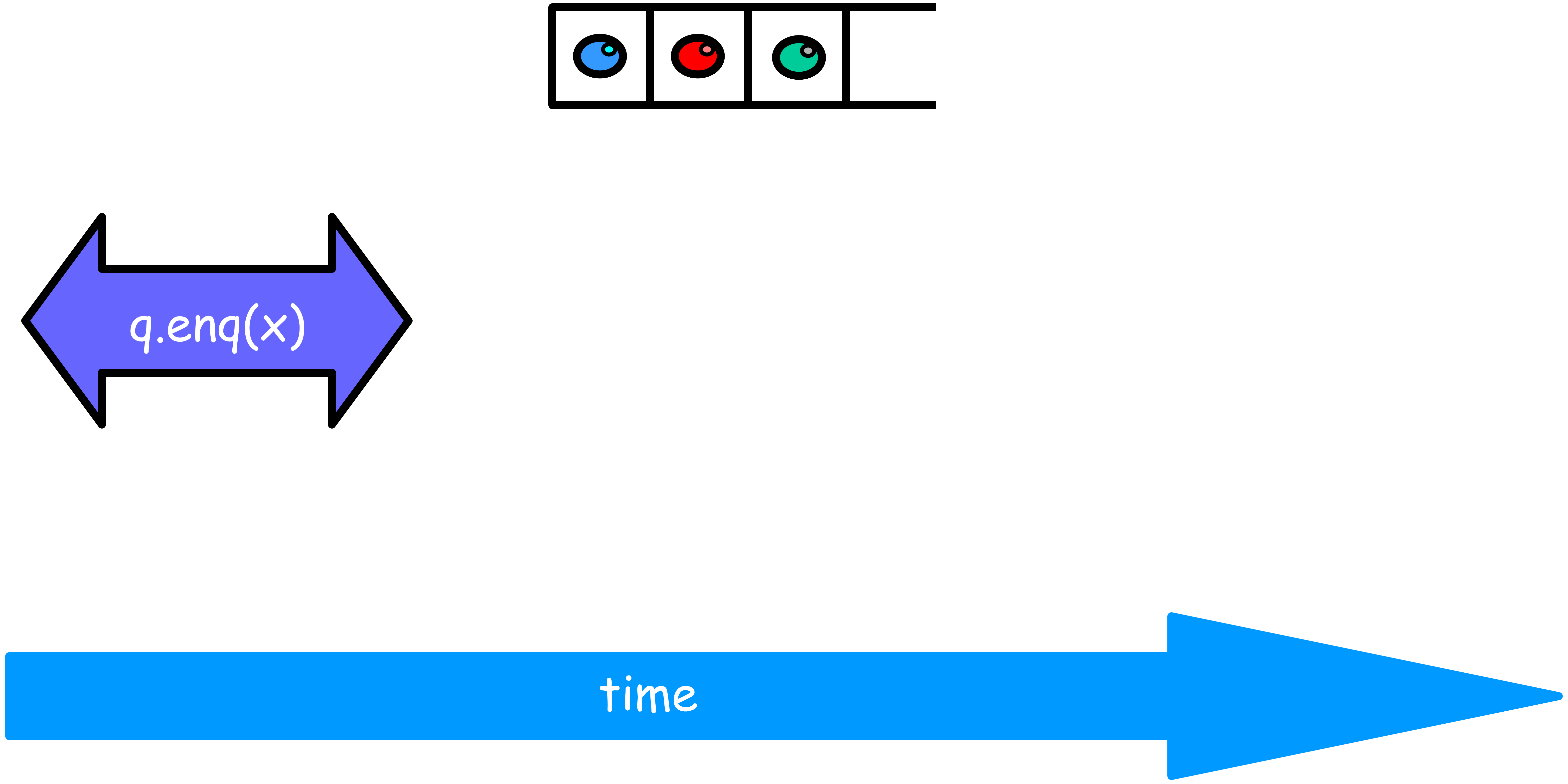
Alternative: Sequential Consistency

- 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
- Formulation:
  - There is some total order of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)

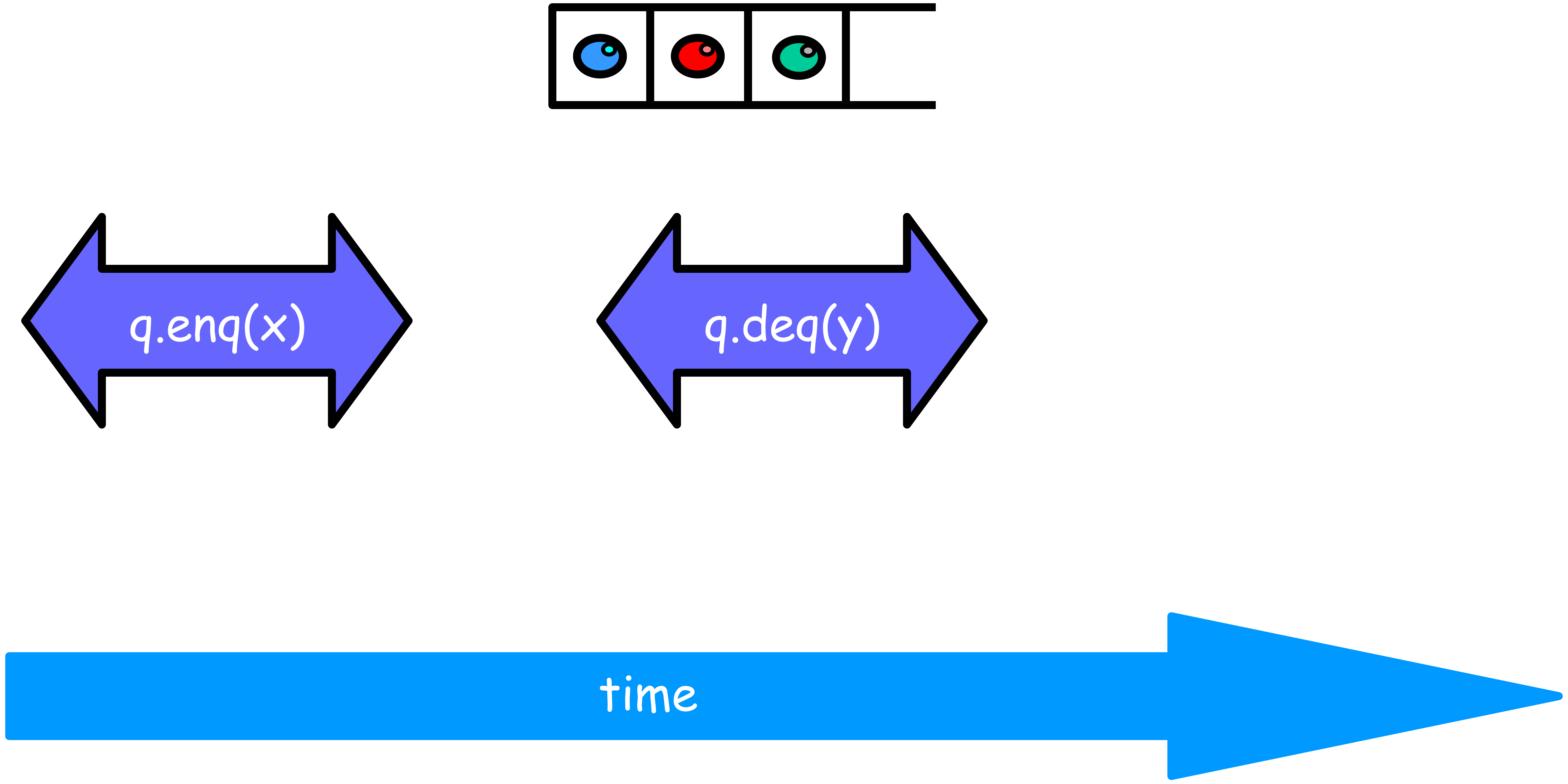
Example



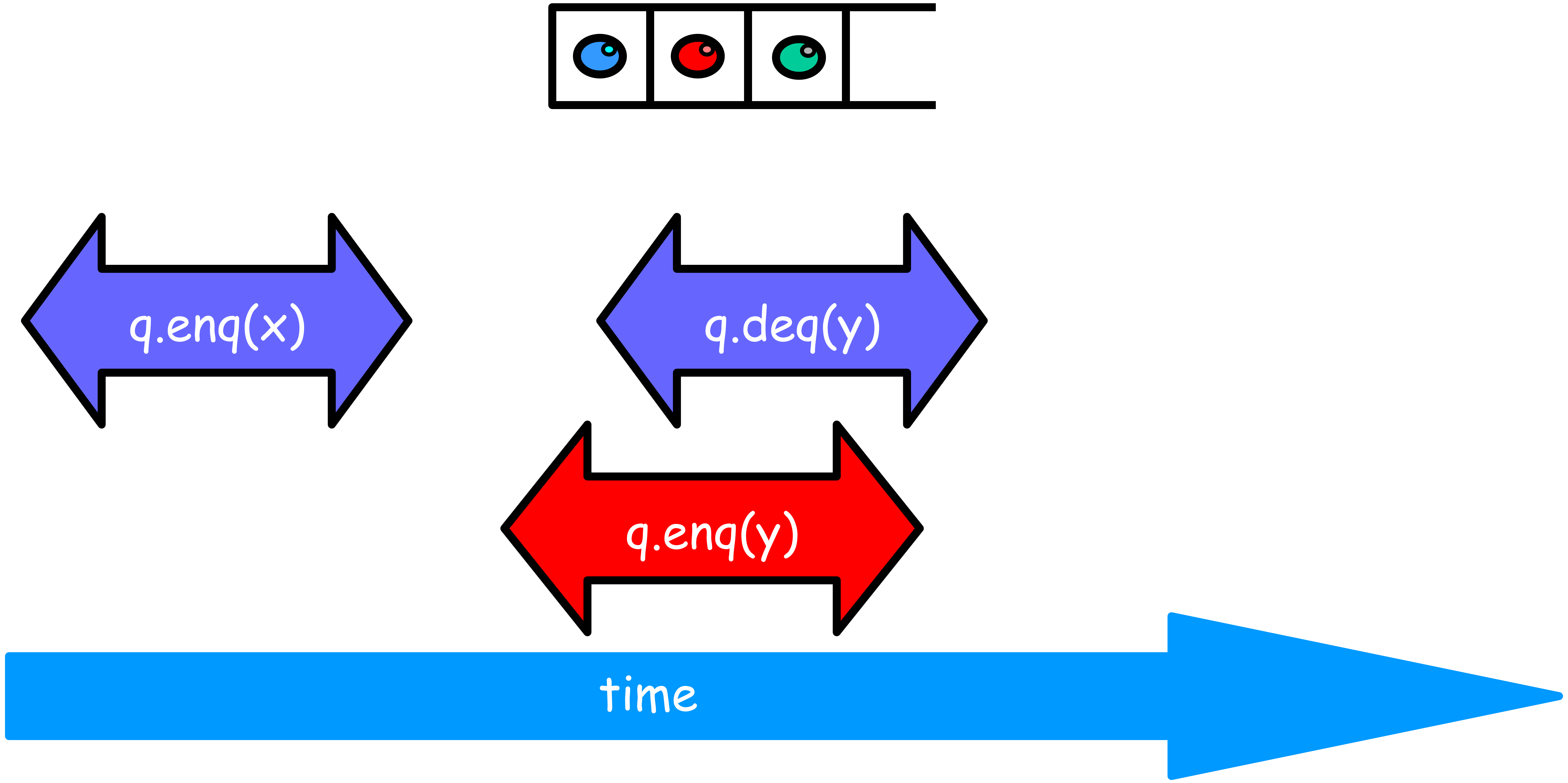
Example



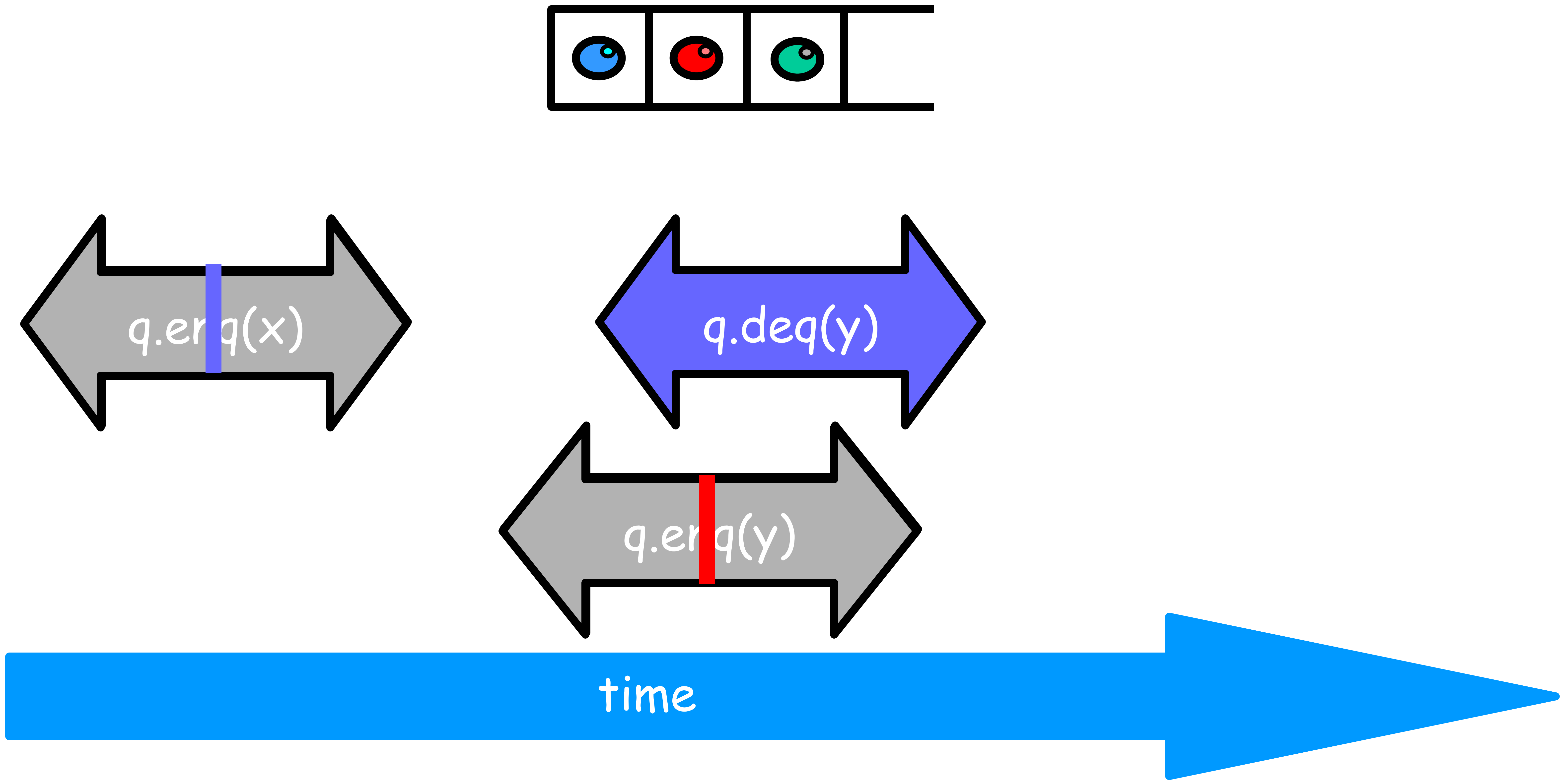
Example



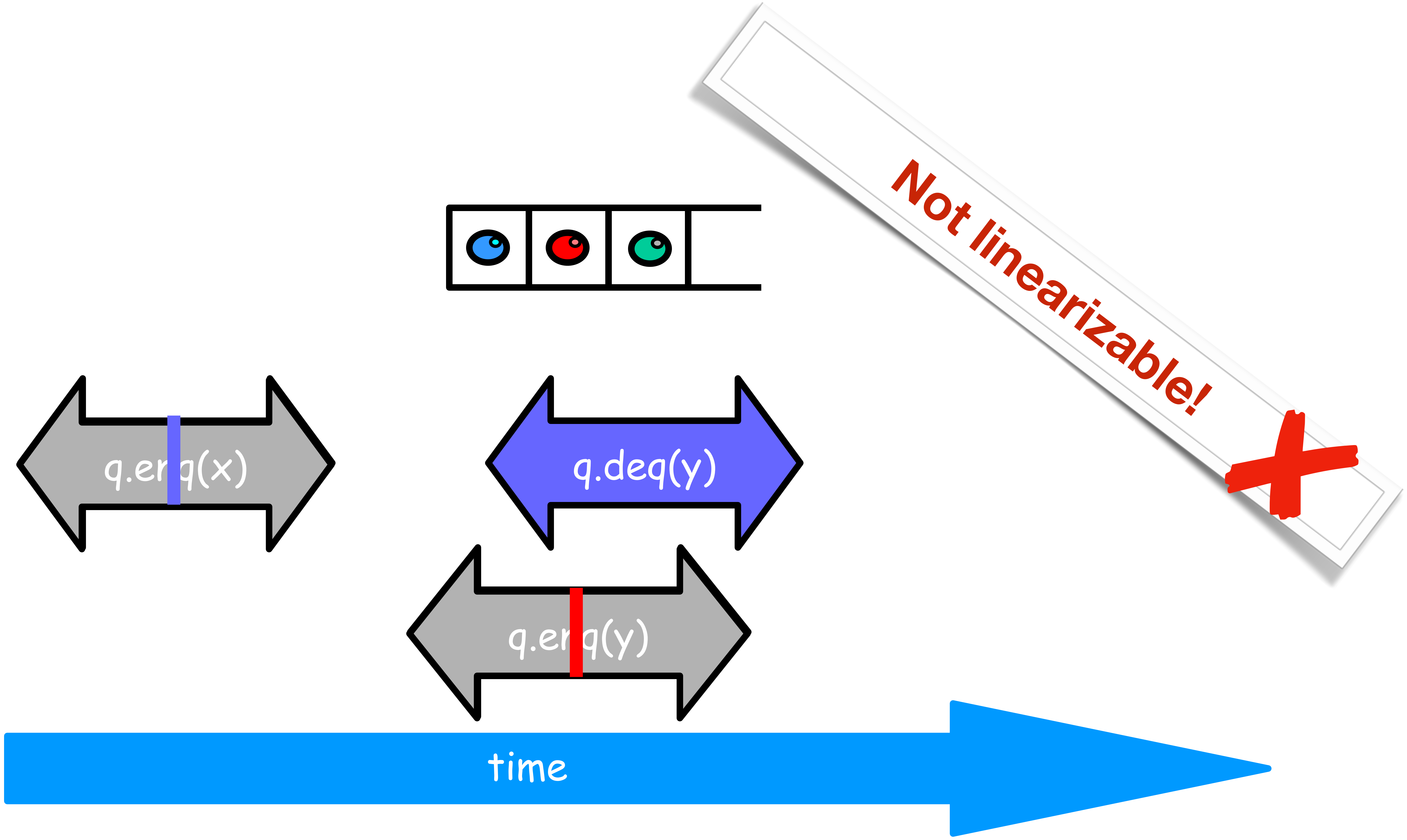
Example



Example

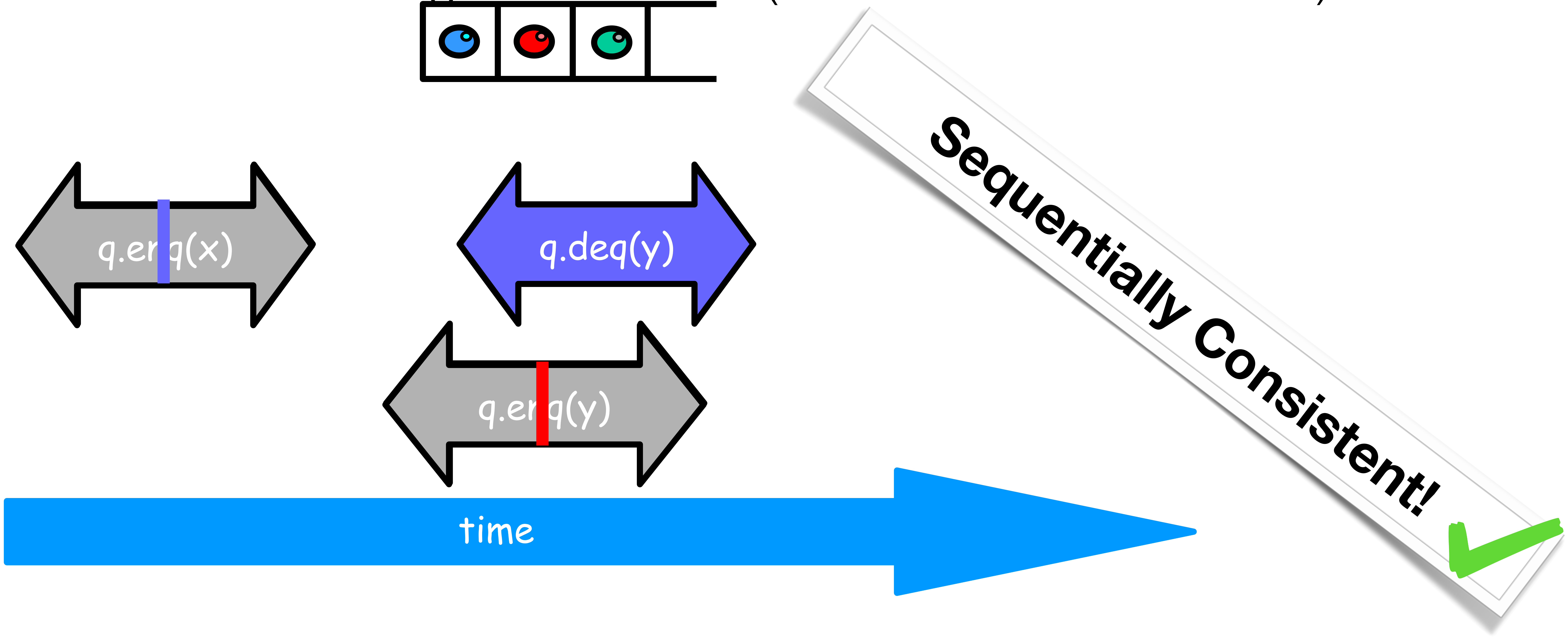


Example



Example

- Reminder, sequentially consistent if there is some total order of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)



Sequential Consistency vs Linearizability

Linearizable

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

Sequential Consistency

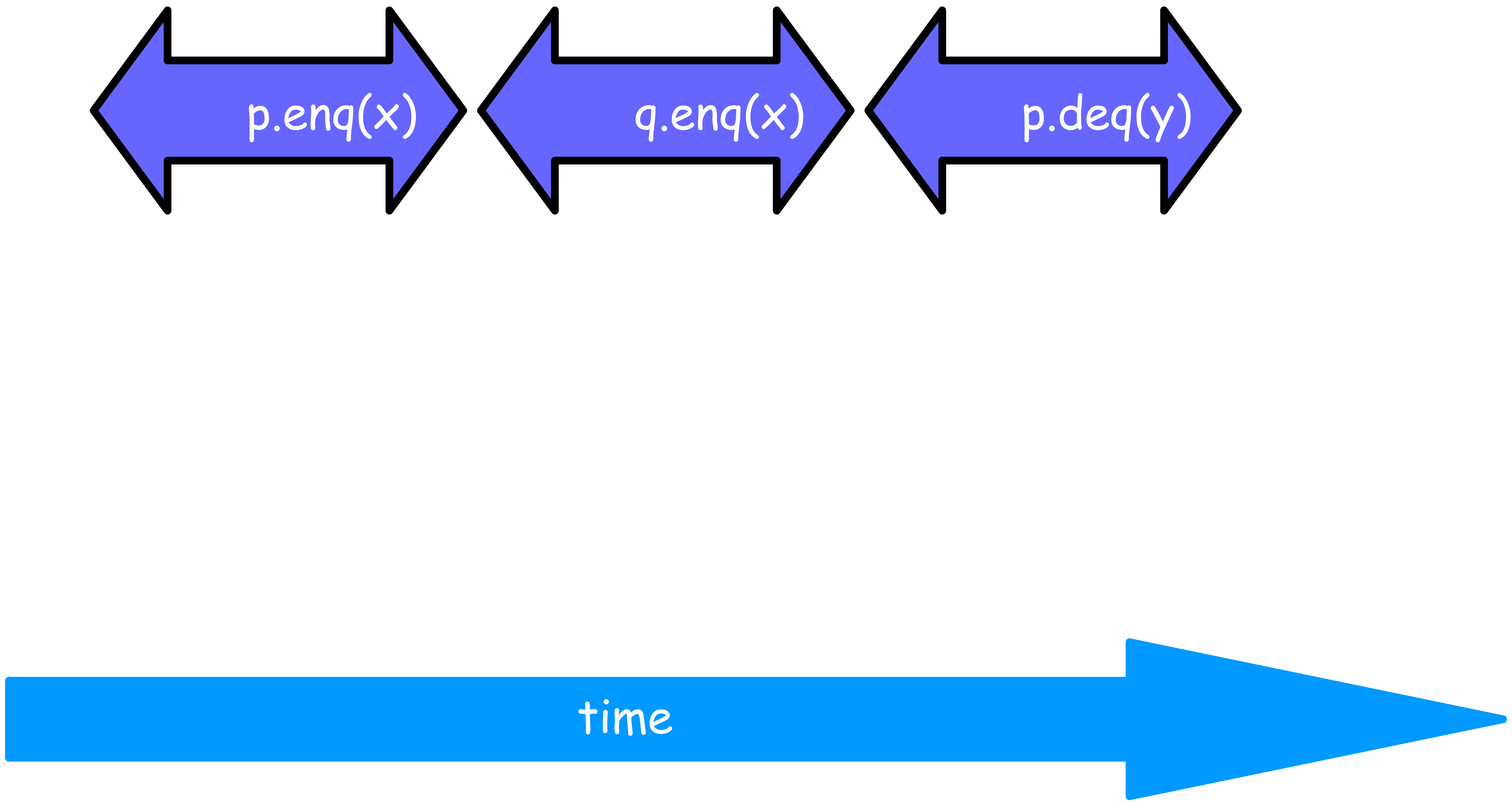
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 0

Theorem

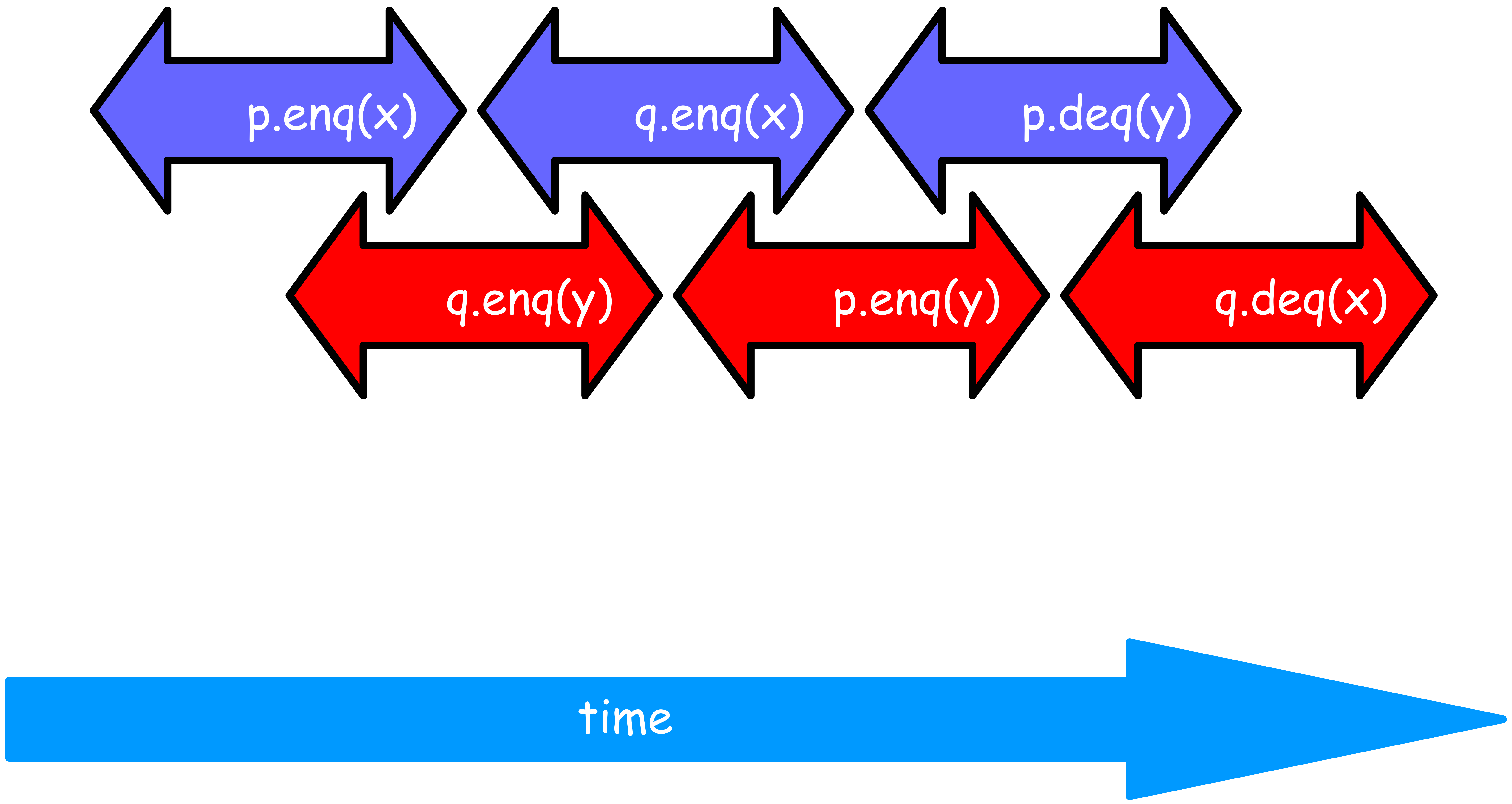
Sequential Consistency is not a local property

(and thus we lose composability…)

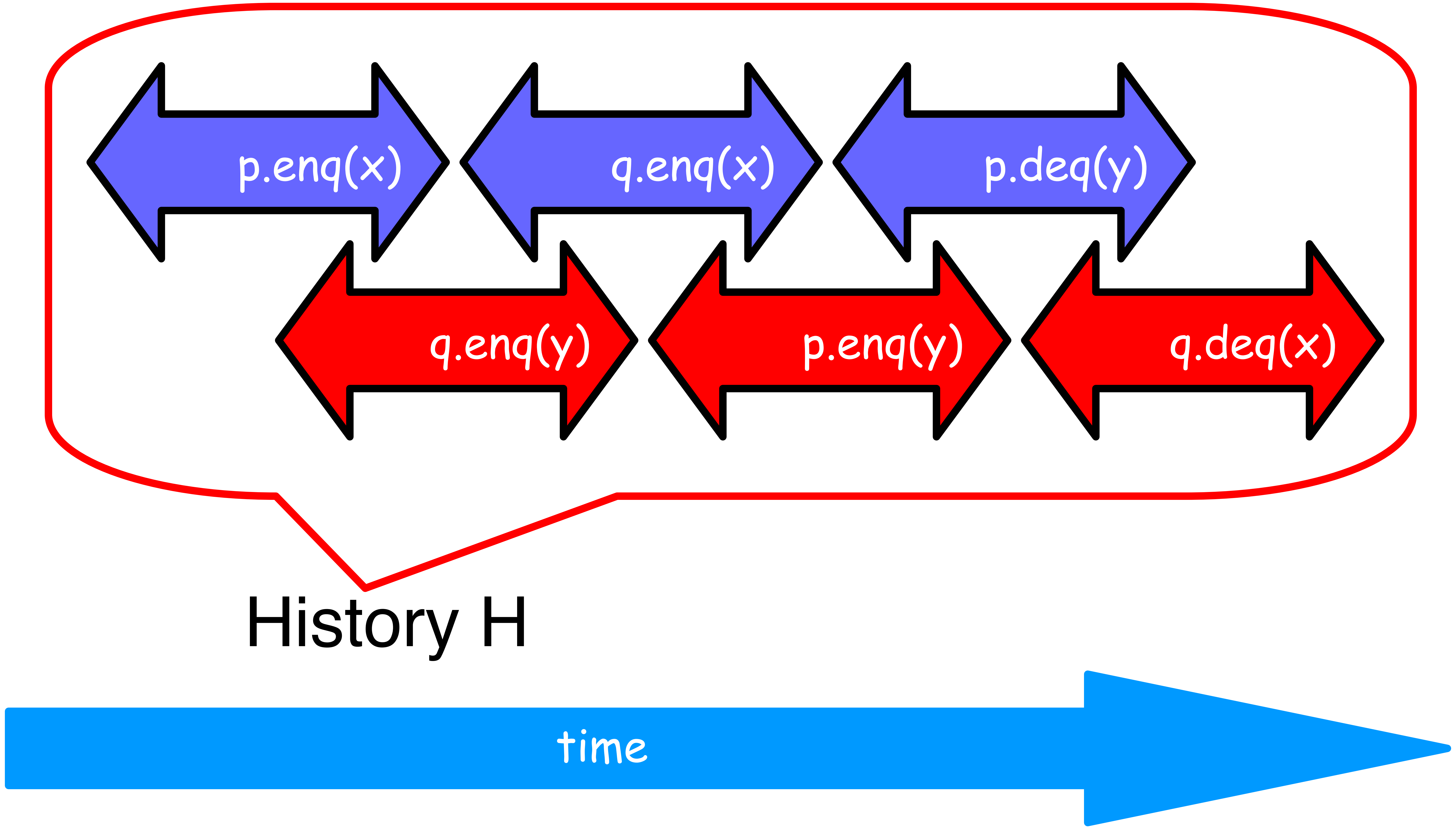
FIFO Queue Example



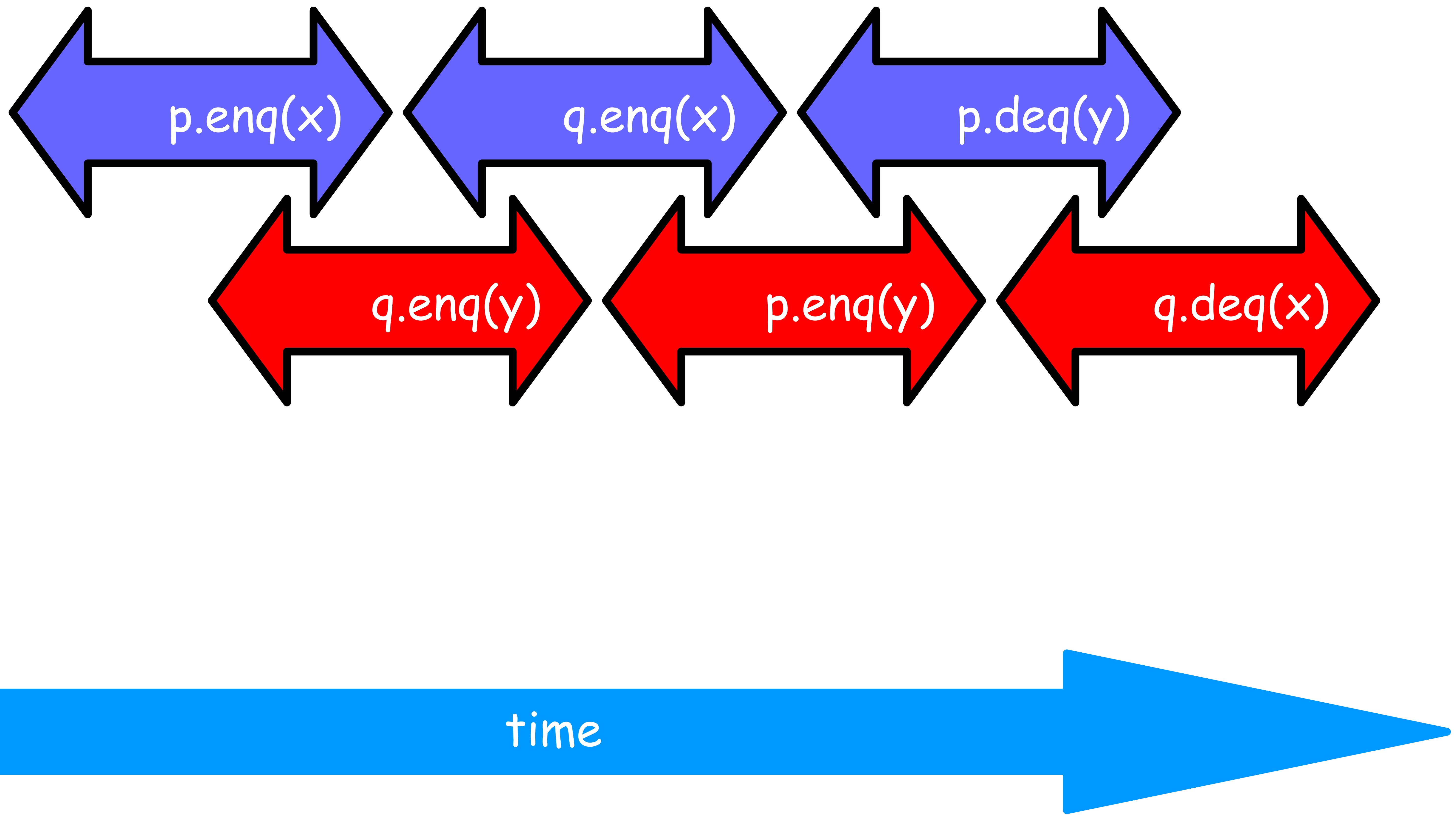
FIFO Queue Example



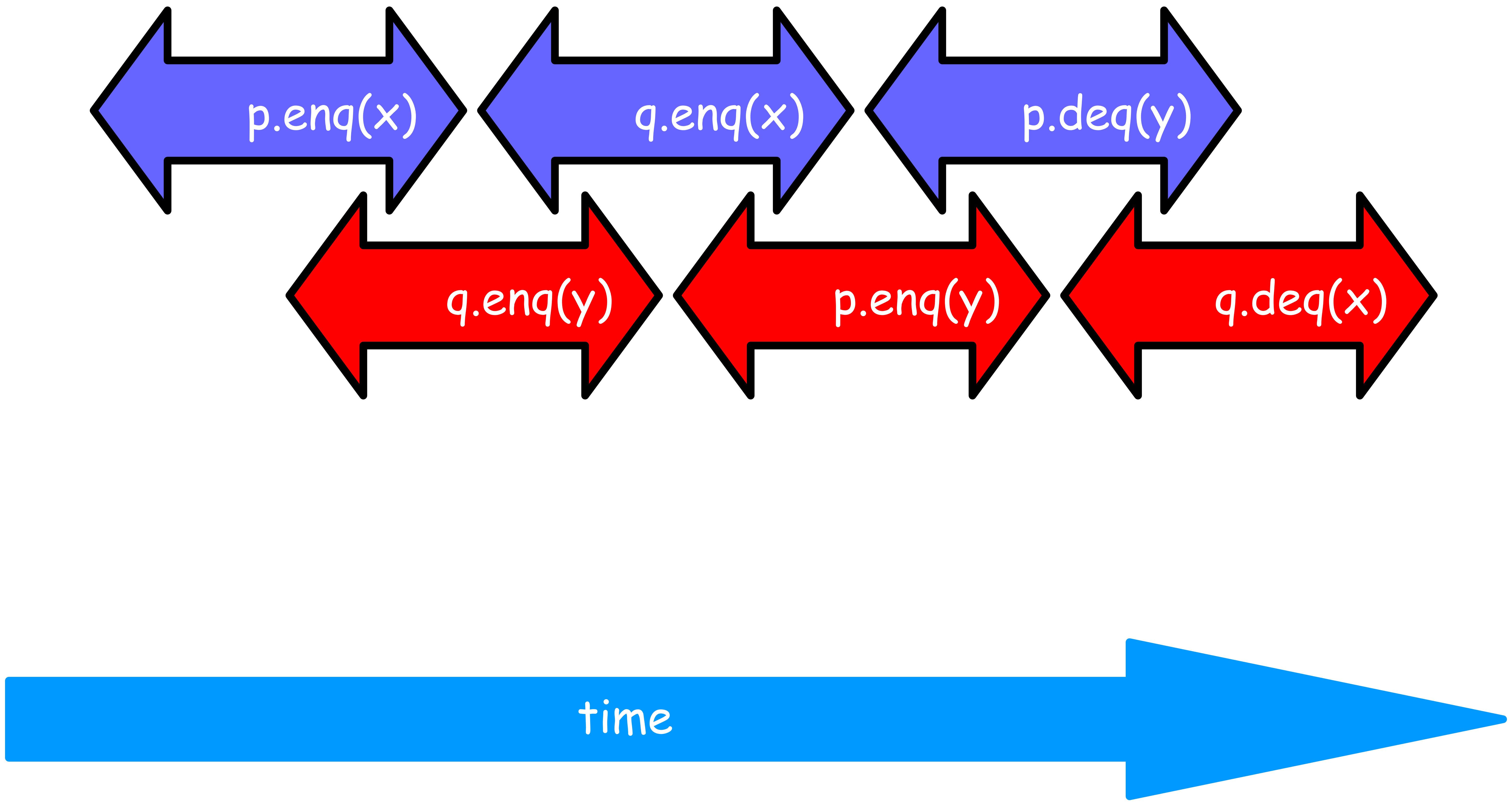
FIFO Queue Example



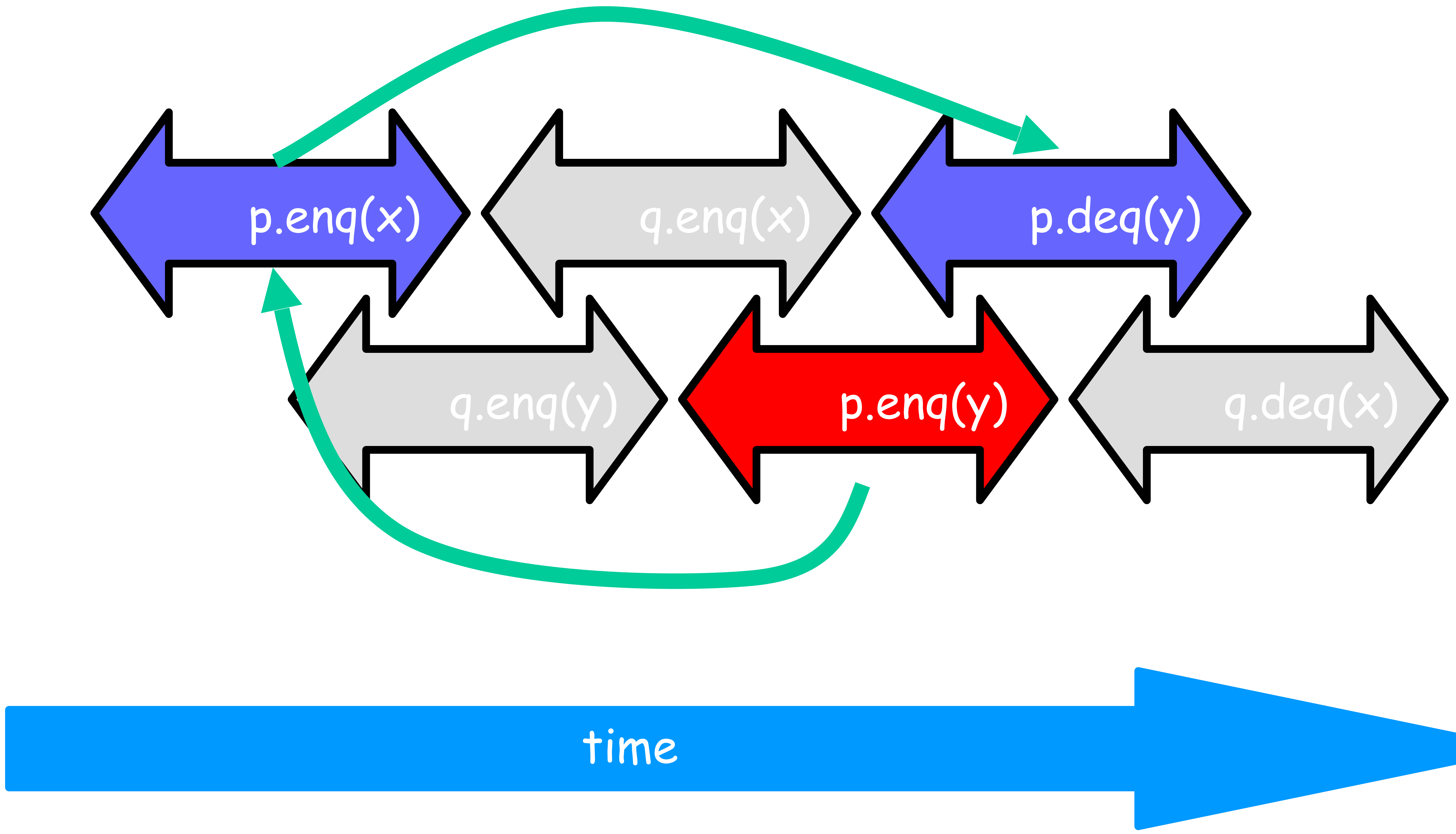
H|p Sequentially Consistent



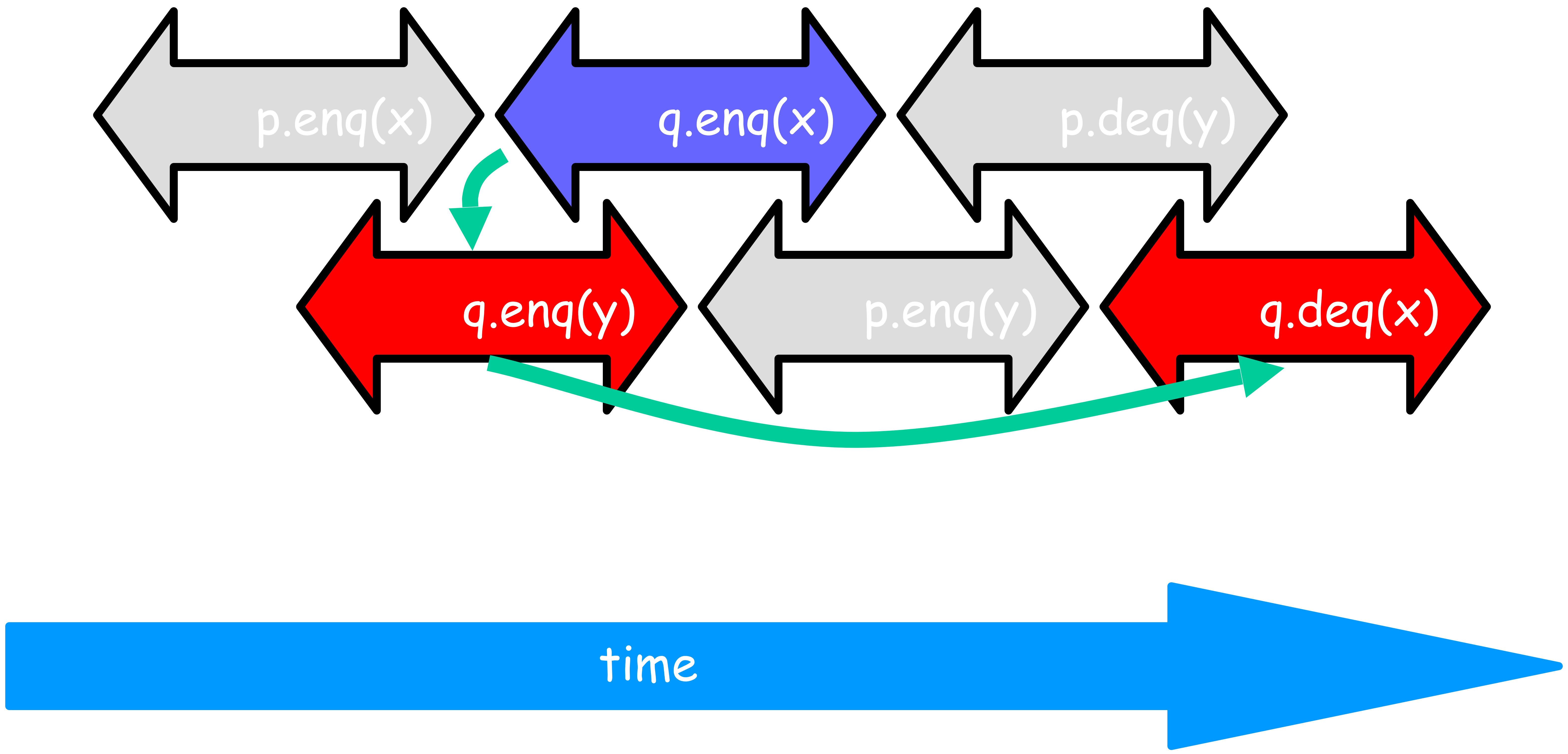
H|q Sequentially Consistent



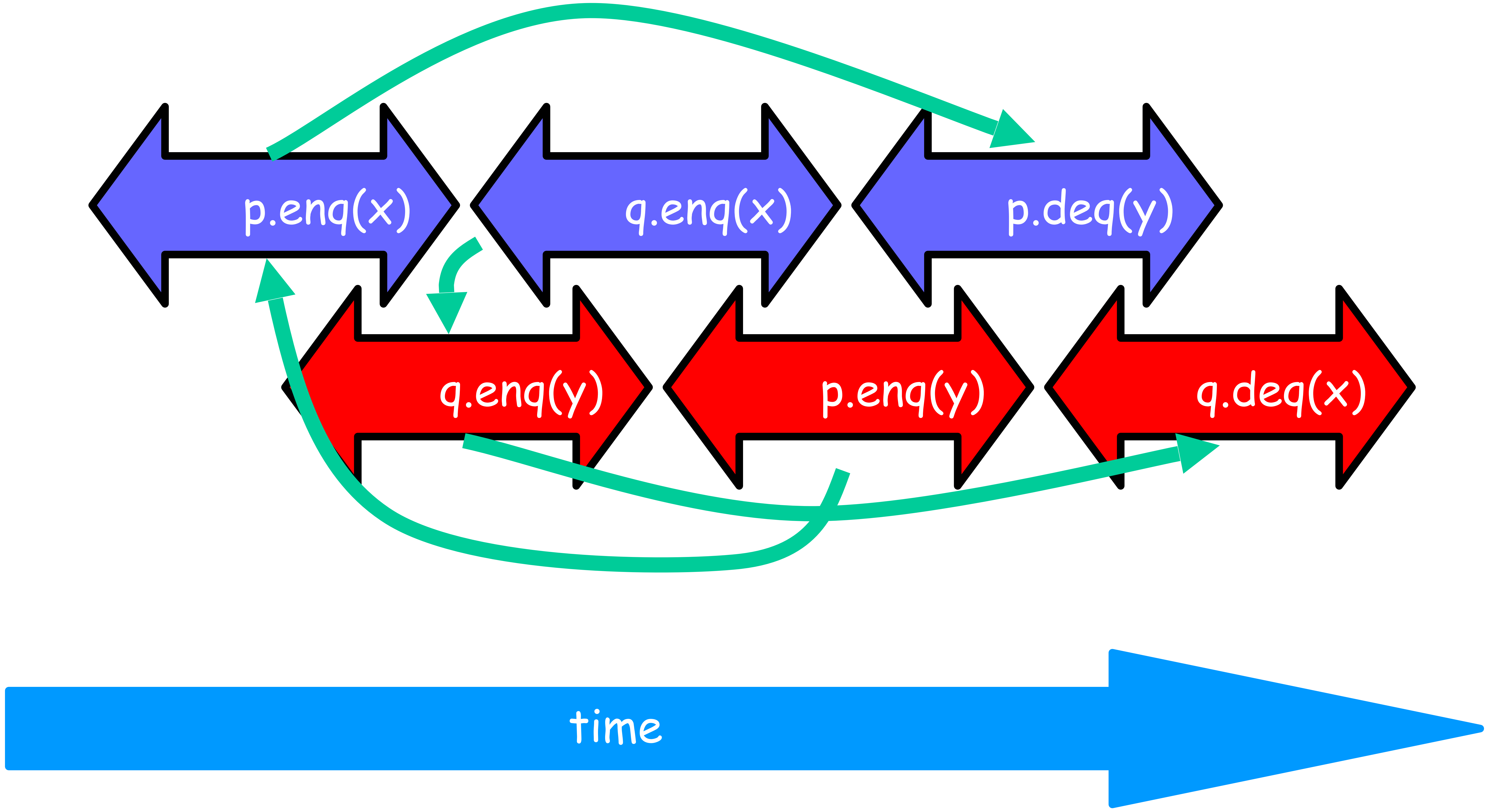
Ordering imposed by p



Ordering imposed by q

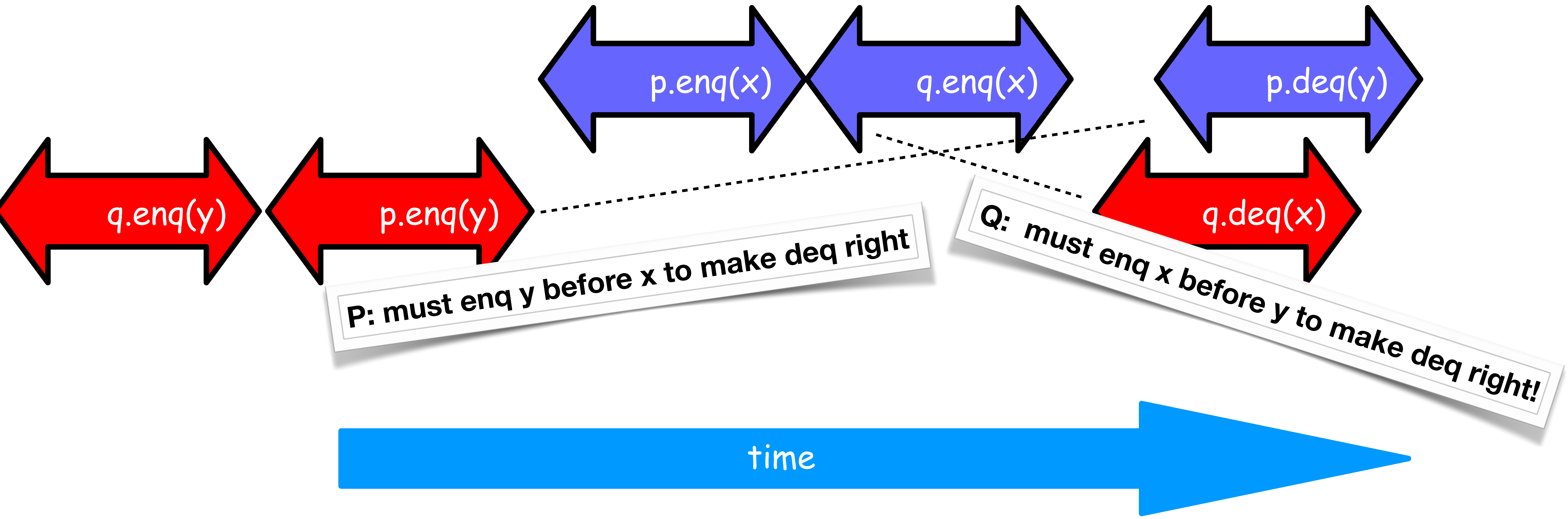


Ordering imposed by both



Sequentially Consistent Overall?

The only way to make this work is if these operations are reordered: but we can not reorder operations within a thread!



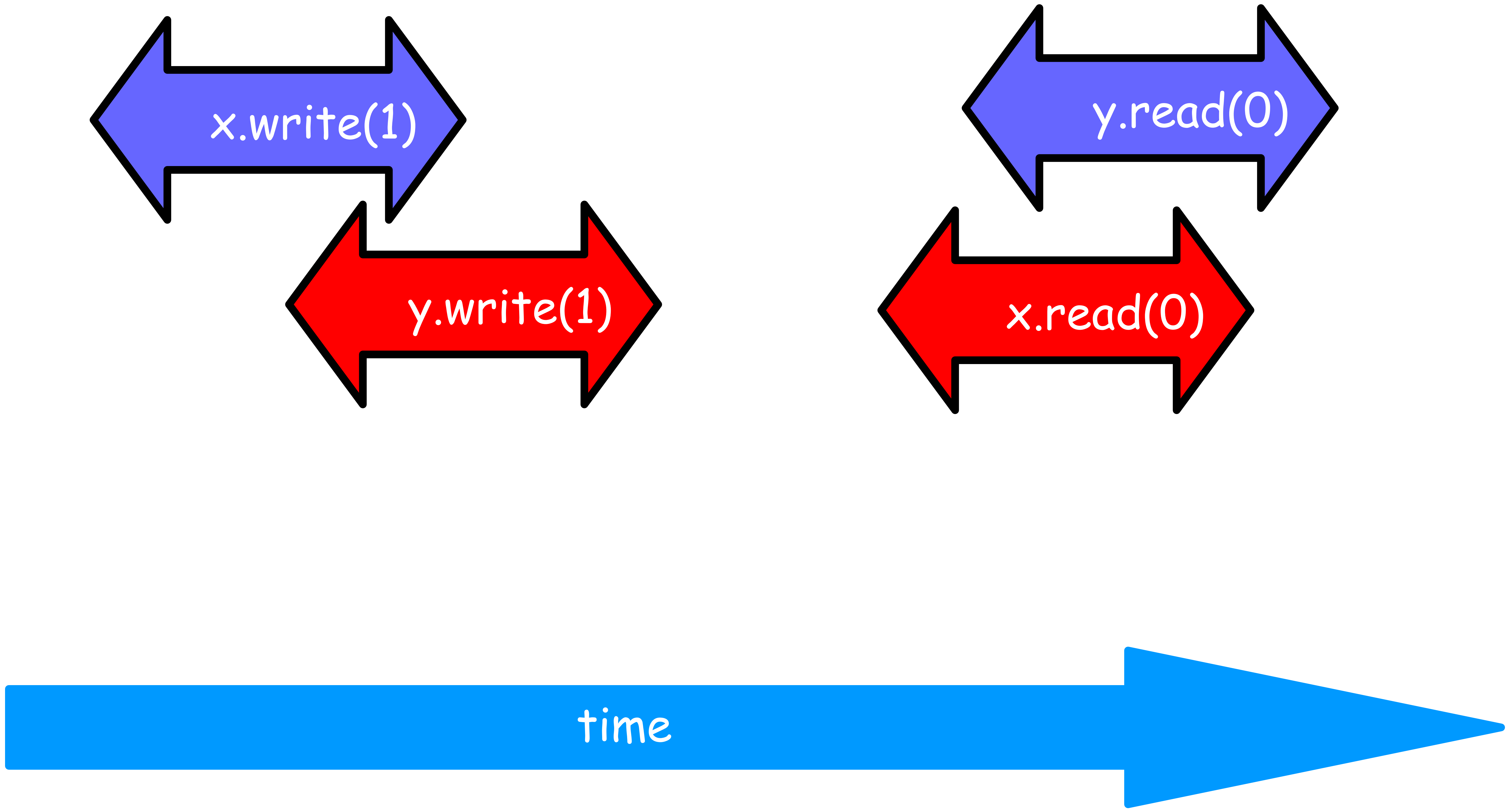
Sequential Consistency vs Linearizability

- Linearizability can be composed:
  - If p's execution and q's execution are both linearizable, then the combination must also be linearizable
- Sequential consistency can not be composed:
  - If p's execution and q's execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)
- Why use sequential consistency?
  - Does not require global clock

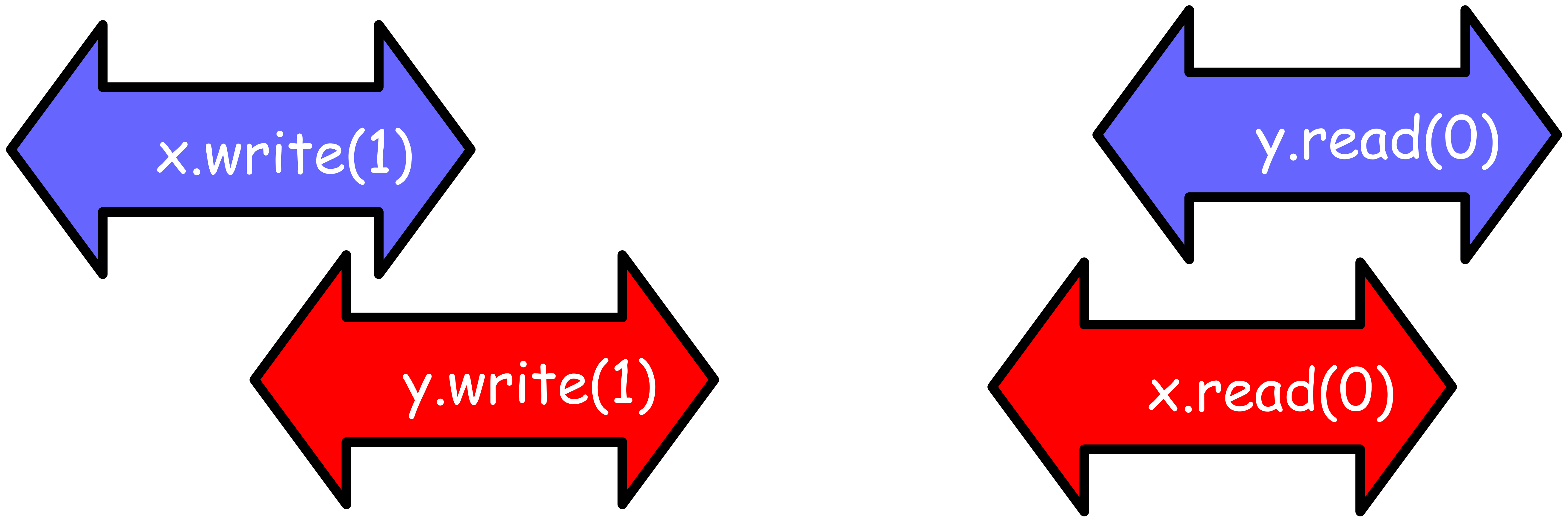
Sequential Consistency

- Even though it's easier to support than linearizability…
- Most hardware architectures don't support sequential consistency
- Because they think it's too strong
- Here's another story …

The Flag Example

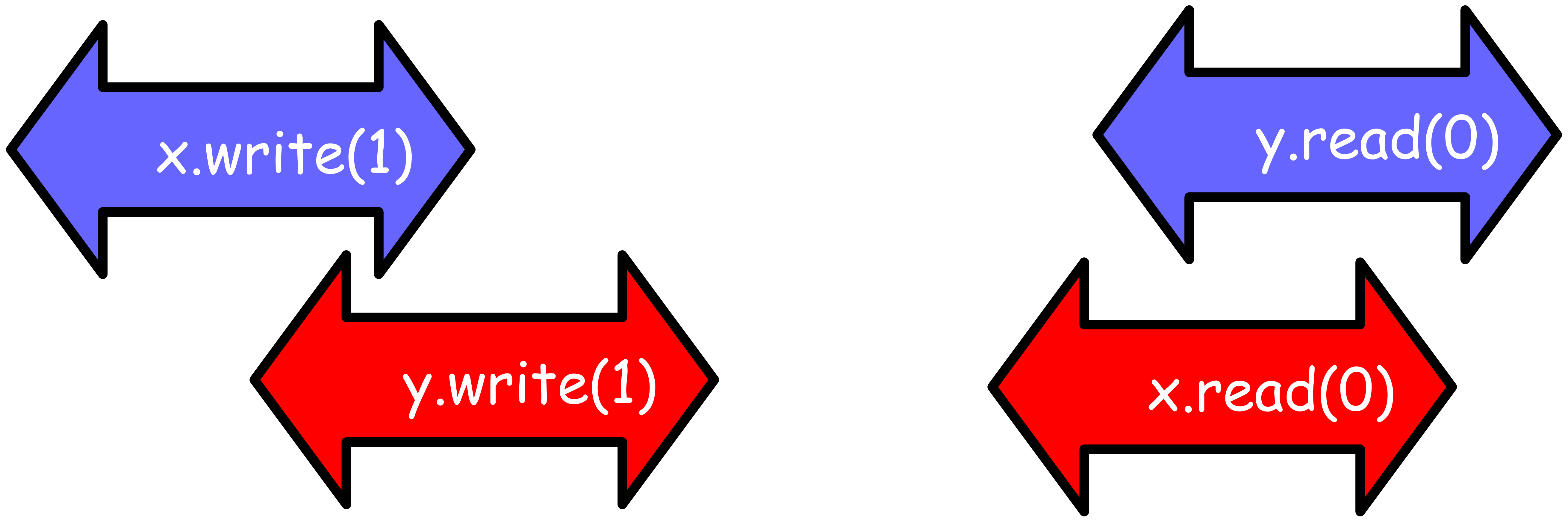


The Flag Example



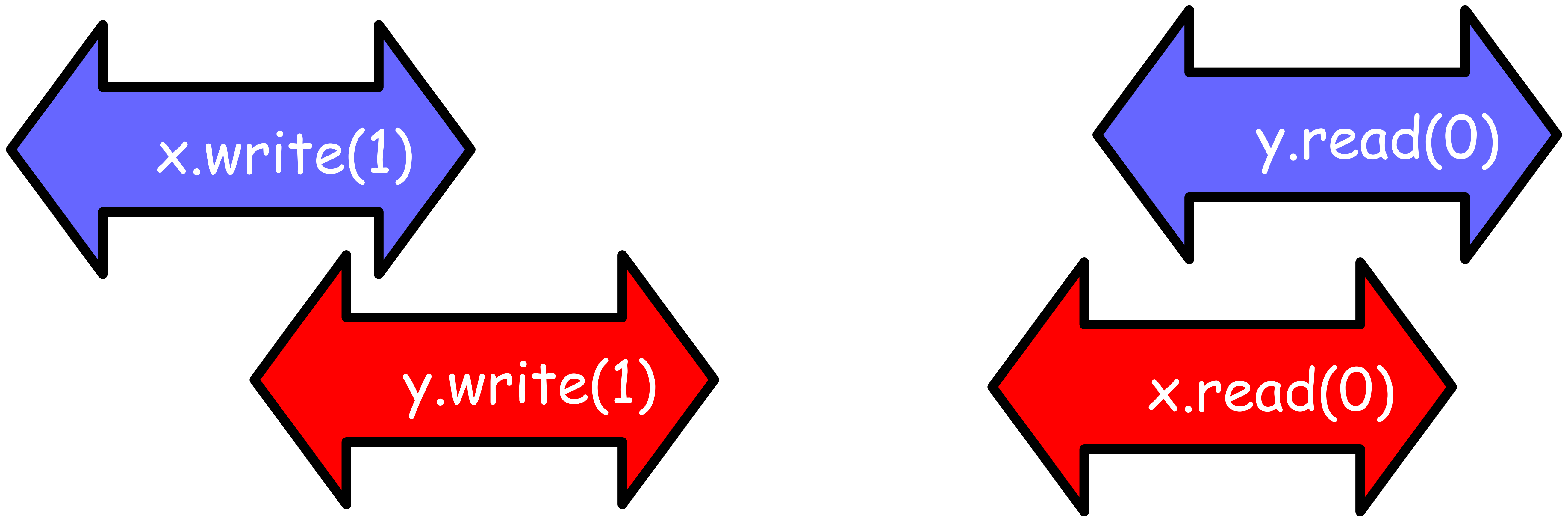
- Each thread's view is sequentially consistent
  – It went first

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 …

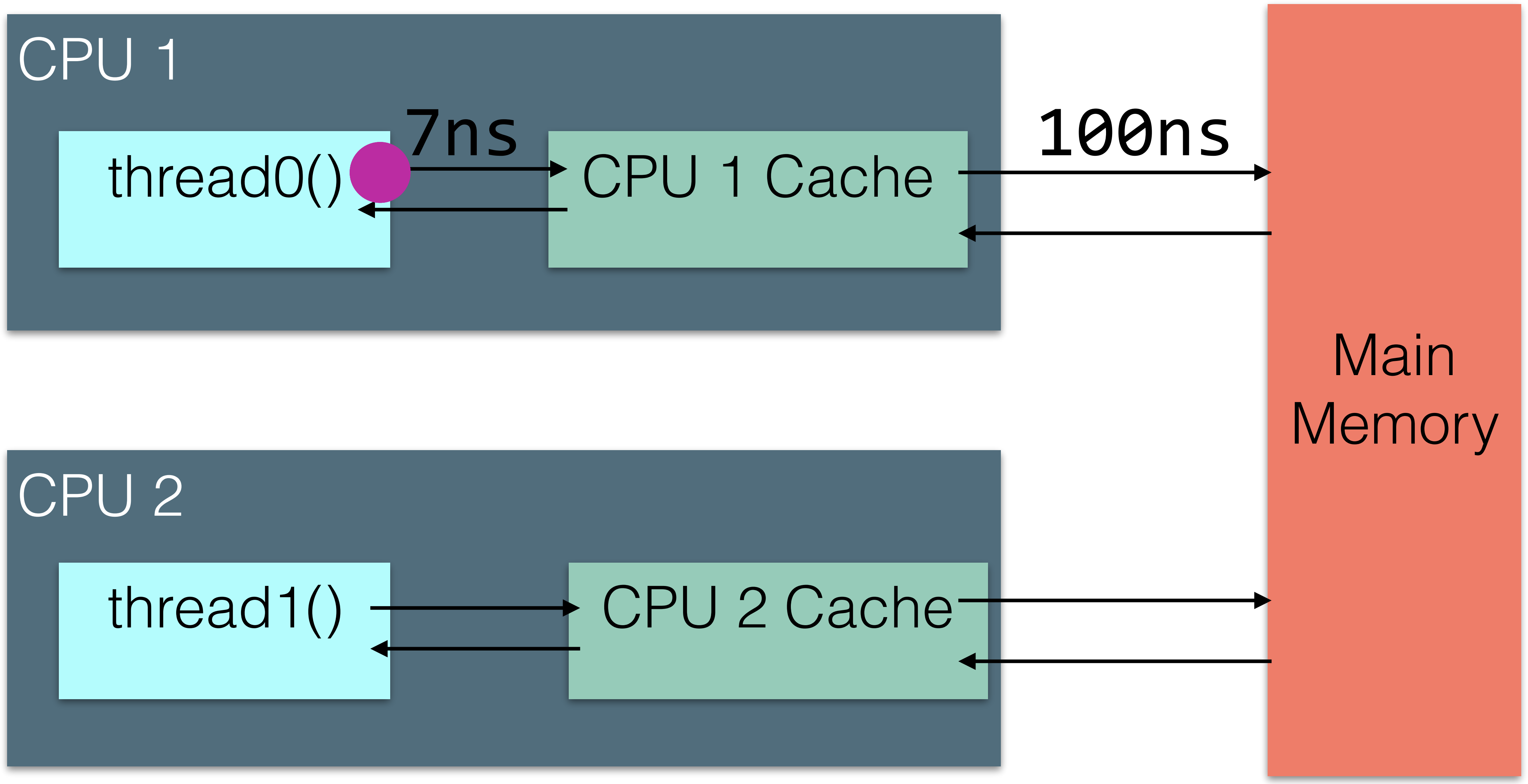
Opinion 1: It’s Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - We used this to implement our lock…
- It’s non-negotiable!

Opinion 2: But It Feels So Right …

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

Memory Operations are Slow



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
- Foreshadowing: Writing to network = mailing a letter to the moon

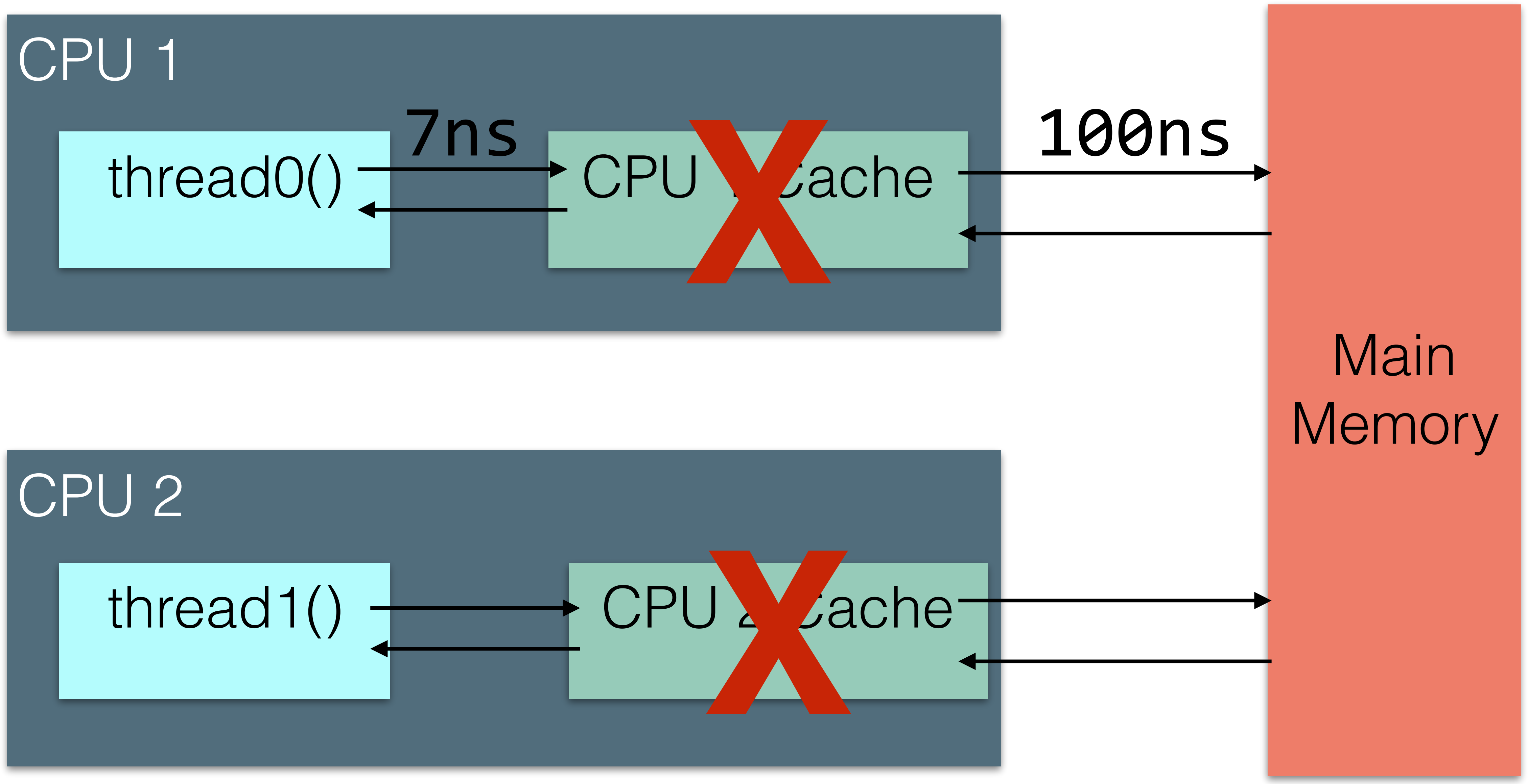
Explicit Synchronization

- 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”

Volatile Keyword



Linearizability, Sequential Consistency, Java, and Us

- What guarantees can we get, and how?
  - Different threads’ access to the same variable is sequential -> volatile
  - Different threads’ access to different variables is linear -> a lock around *every* read/write to those variables

Revisionist History

```
class Bouncer {
   public static final int DOWN = 0;
   public static final int RIGHT = 1;
   public static final int STOP = 2;
   private boolean goRight = false;
   private ThreadLocal<Integer> myIndex;
   private int last = -1;

   int visit() {
      int i = myIndex.get();
      last = i;
      if (goRight)
         return RIGHT;
      goRight = true;
      if (last == i)
         return STOP;
      else
         return DOWN;
   }
}
```

Without volatile, it would actually be possible for ALL threads to see STOP

Revisionist History

```
class Bouncer {
   public static final int DOWN = 0;
   public static final int RIGHT = 1;
   public static final int STOP = 2;
   private volatile boolean goRight = false;
   private ThreadLocal<Integer> myIndex;
   private volatile int last = -1;

   int visit() {
      int i = myIndex.get();
      last = i;
      if (goRight)
         return RIGHT;
      goRight = true;
      if (last == i)
         return STOP;
      else
         return DOWN;
   }
}
```

Without volatile, it would actually be possible for ALL threads to see STOP

Synchronization beyond locks and flags

Return to the FIFO queue

- What if we want our lock based queue to be *bounded*
- Recall: what we did was use a lock around the enqueue and dequeue methods

```
mutex.lock()
try{
 queue.enq(x);
} finally{
 mutex.unlock();
}
```

- What happens if the queue is full?
  - Throw an exception?
  - Can’t we make it wait until the queue has space?

Conditions

- When a thread is waiting for something to happen, it might want to release the lock and be notified when that thing has happened
  - Ex: while queue is full, release the lock to let someone else empty it
- This is what a *condition* does
- Key methods: *await*, *signal*

```
Condition condition = lock.newCondition();
lock.lock();
try{
   while(!property)
      condition.await();
}catch(InterruptedException e){
   //Application-dependent response, property may still be false
}
//at this point, property must be true and we have the lock again
```

```
condition.signal(); //wake up one thread await'ing
condition.signalAll(); //wake up all threads await'ing
```

Conditions

- An awakened thread must:
  - try to reclaim the lock;
  - when this has happened, retest the property it is waiting for;
  - if the property doesn't hold, release the lock by calling await().
- wrong: `if "boolean expression" condition.await()`
- correct: `while "boolean expression" condition.await()`

Bounded Queue with Conditions

```
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();
public void enq(Object x) throws InterruptedException {
   lock.lock();
   try {
      while (count == items.length)
         notFull.await();
      items[tail] = x;
      if (++tail == items.length)
         tail = 0;
      ++count;
      notEmpty.signal();
   } finally {
      lock.unlock();
   }
}
```

```
public Object deq() {
   lock.lock();
   try {
      while (count == 0)
         notEmpty.await();
      Object y = items[head];
      if (++head == items.length)
         head = 0;
      --count;
      notFull.signal();
      return y;
   } finally {
      lock.unlock();
}}
```