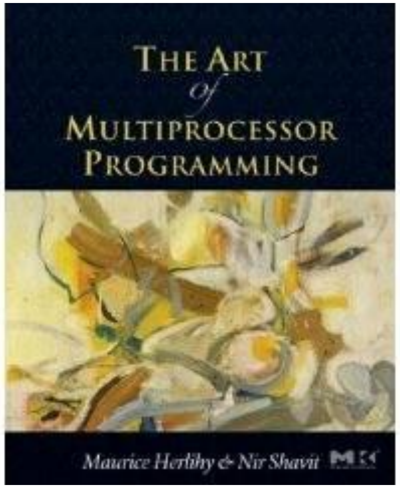
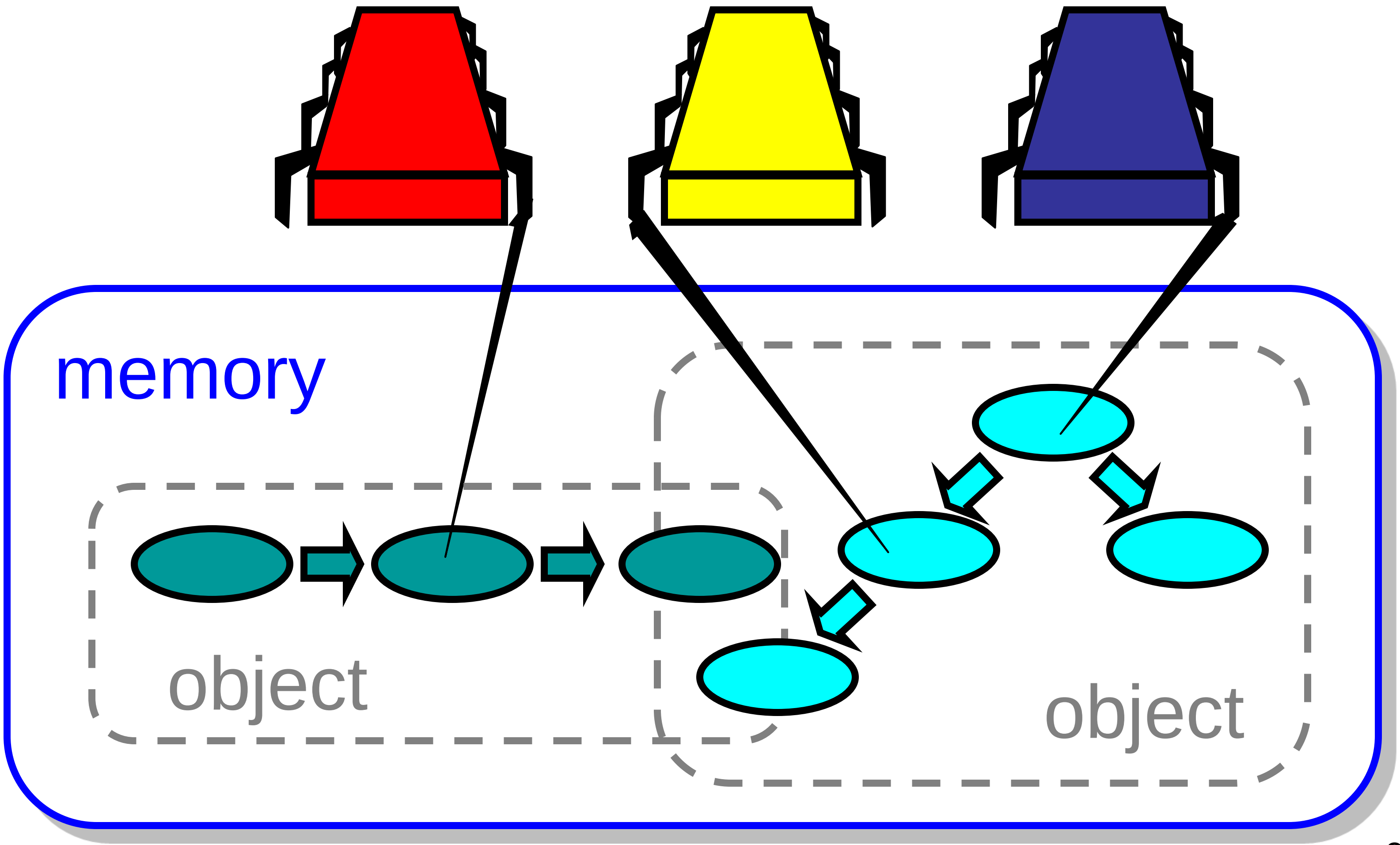
Programming Paradigms for Concurrency
Lecture 3 – Concurrent Objects



Based on companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Modified by
Thomas Wies
New York University

Concurrent Computation



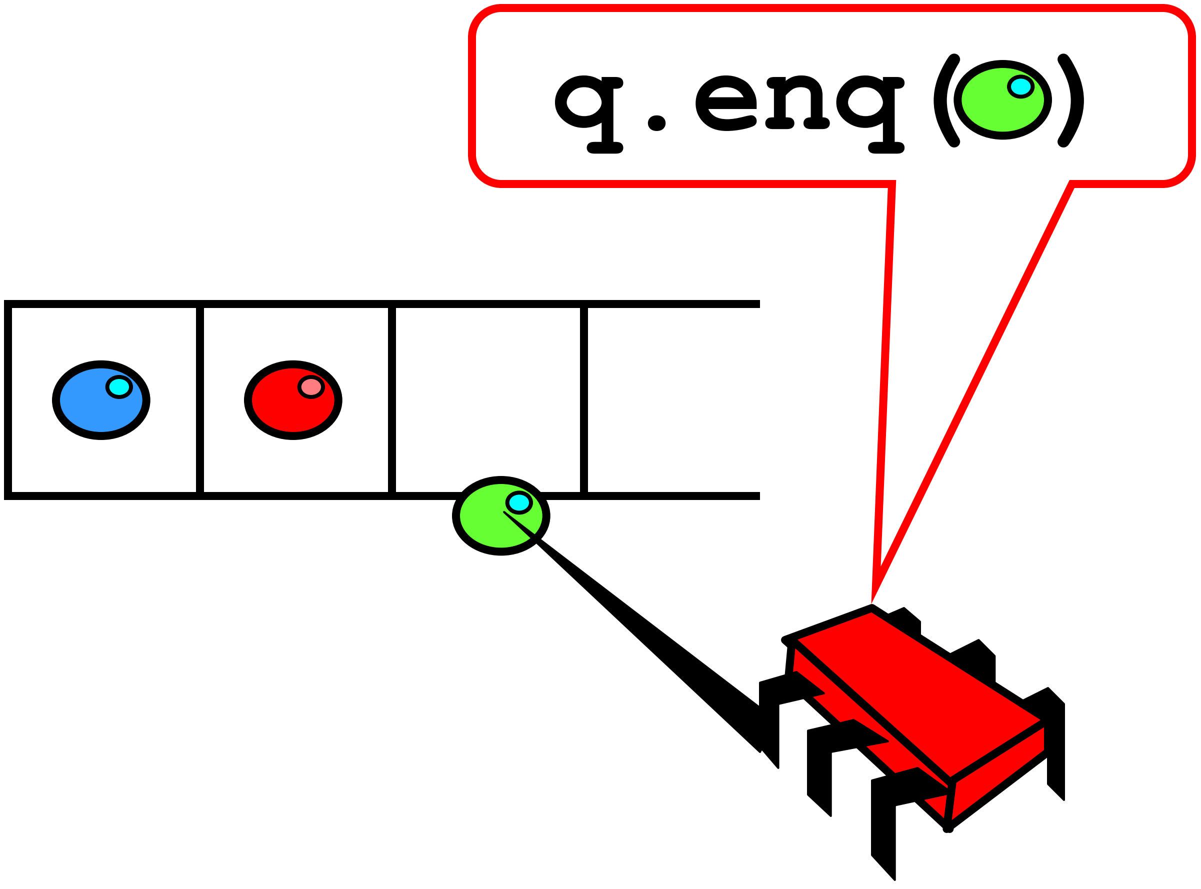
Objectivism

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

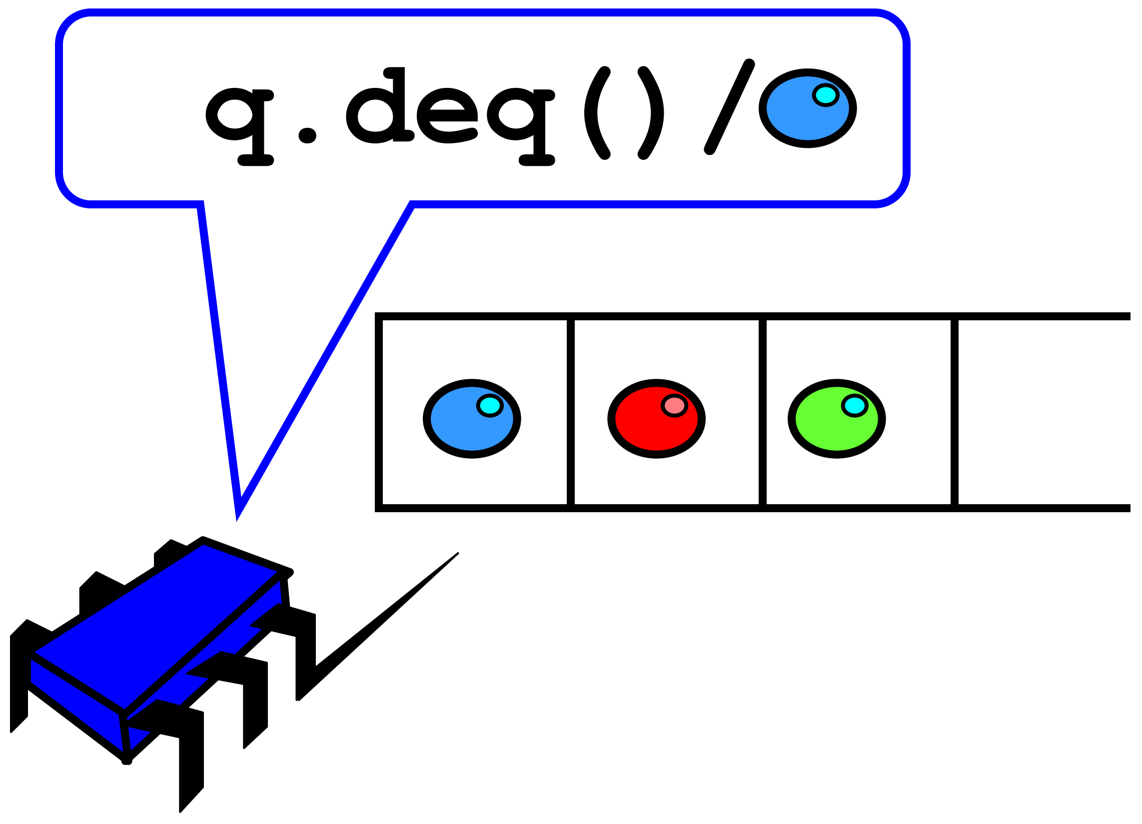
Objectivism

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

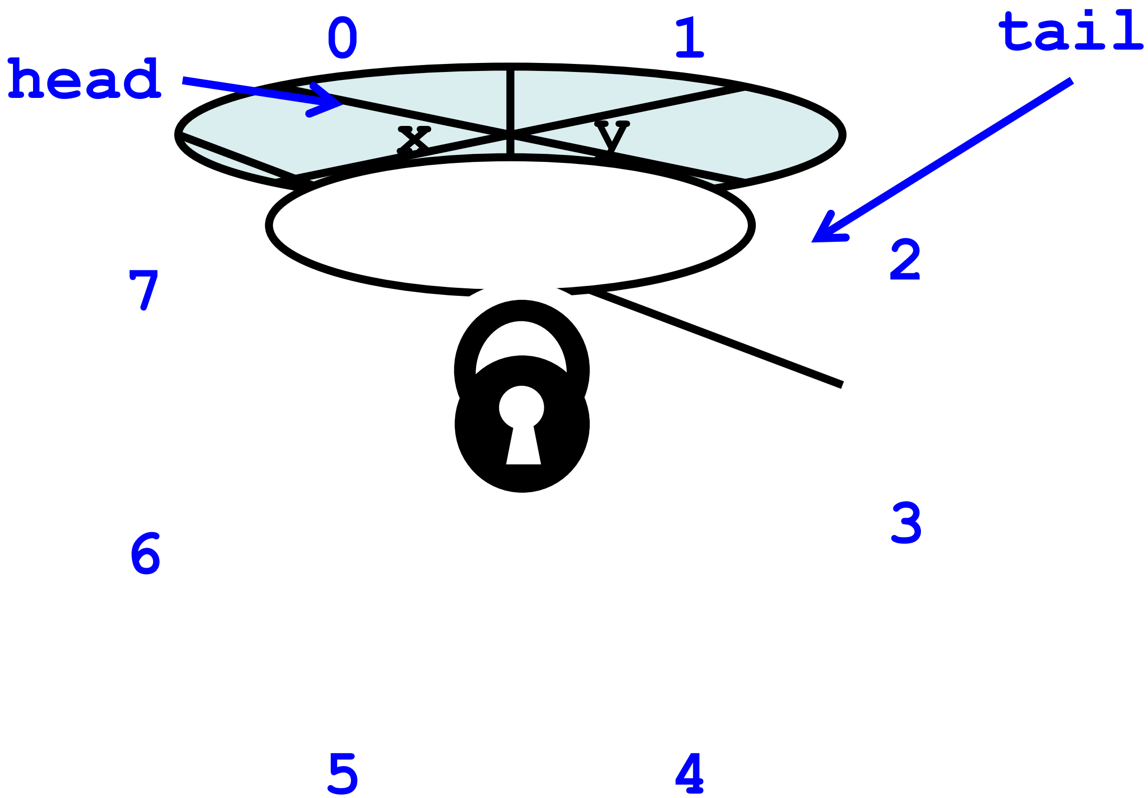
FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method

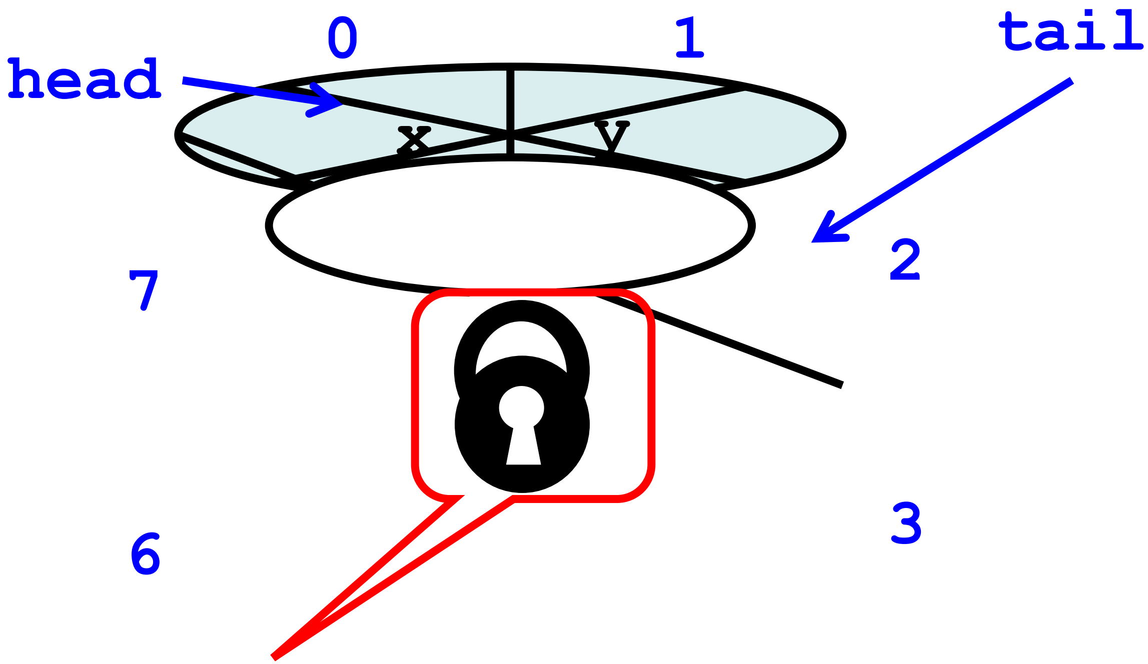


Lock-Based Queue



capacity = 8

Lock-Based Queue

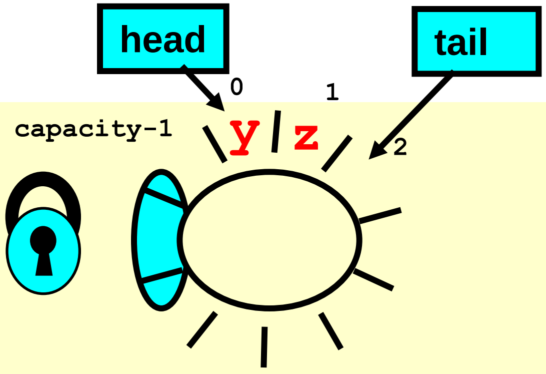


Fields protected by single shared lock

capacity = 8

A Lock-Based Queue

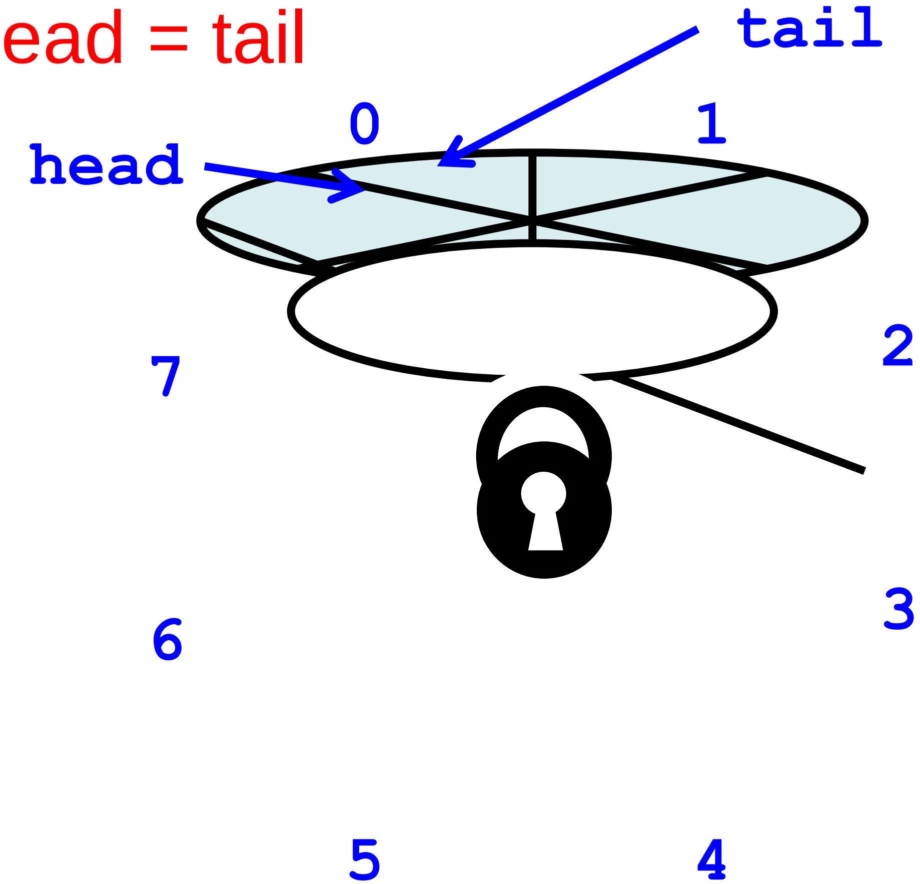
```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```



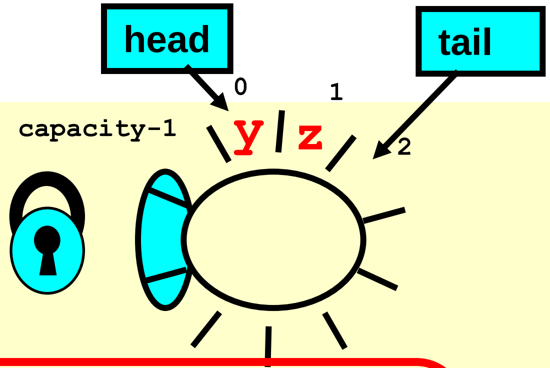
Fields protected by single shared lock

Lock-Based Queue

Initially head = tail



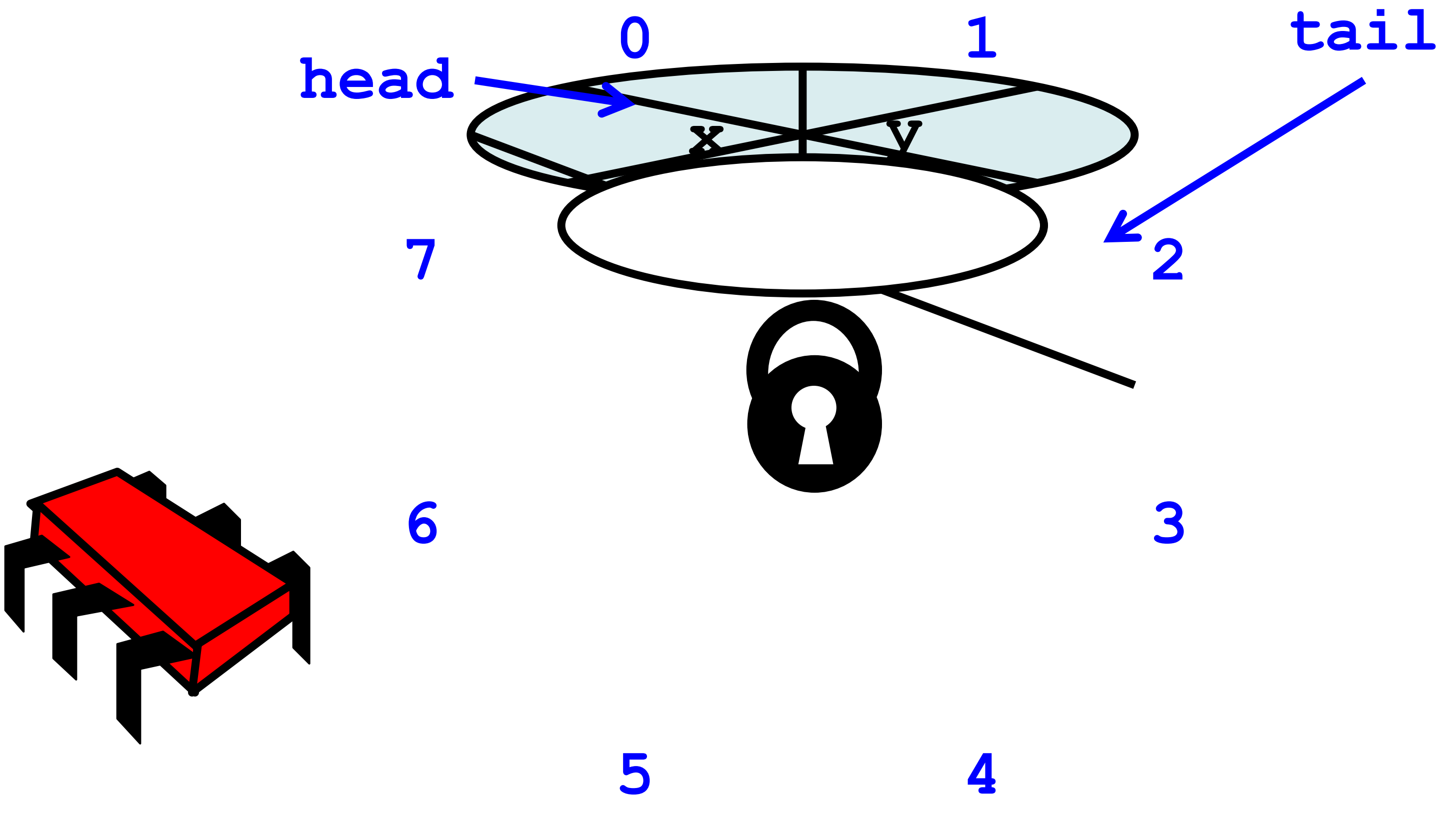
A Lock-Based Queue



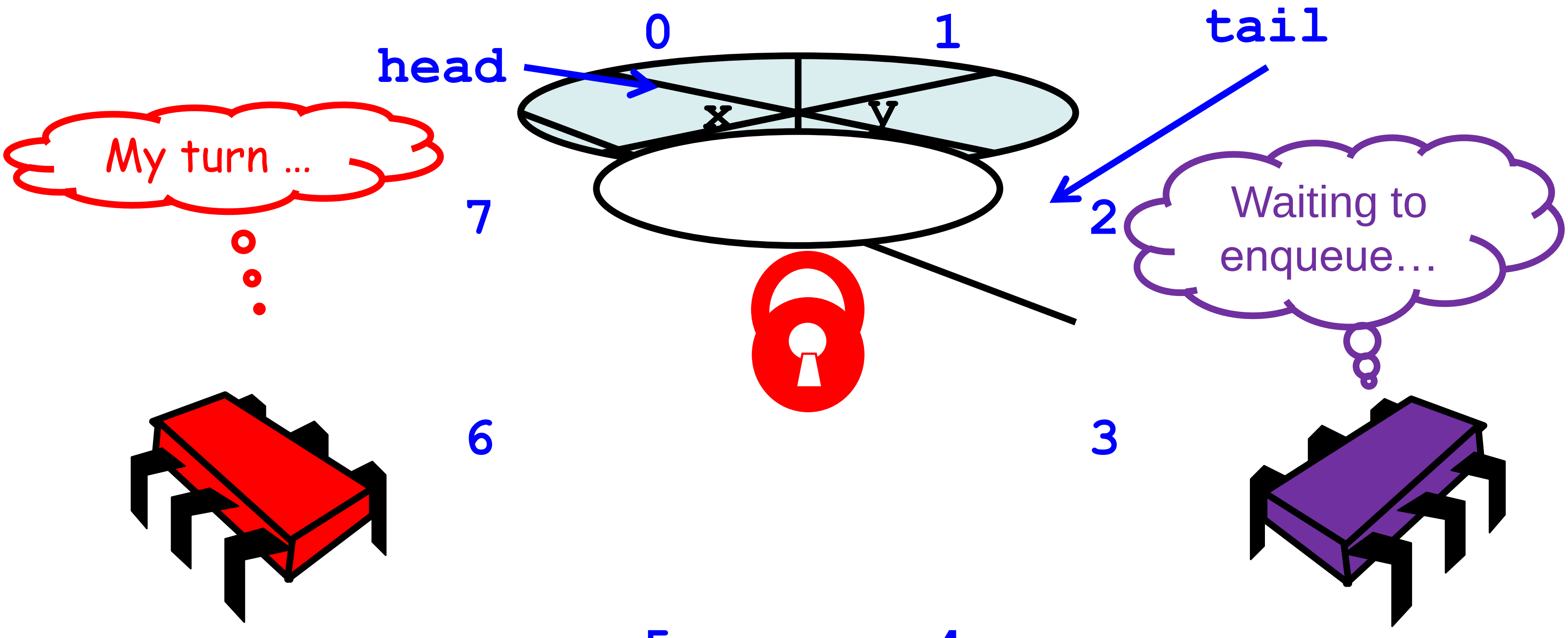

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Initially head = tail

Lock-Based deq()



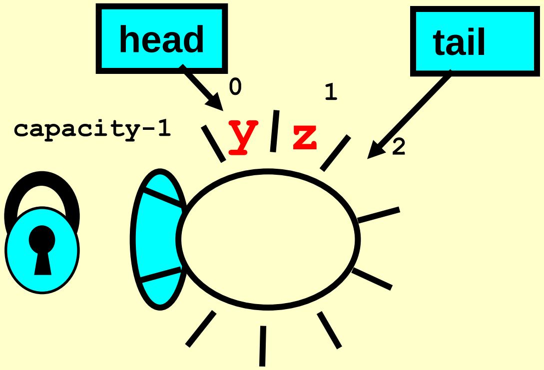
Acquire Lock



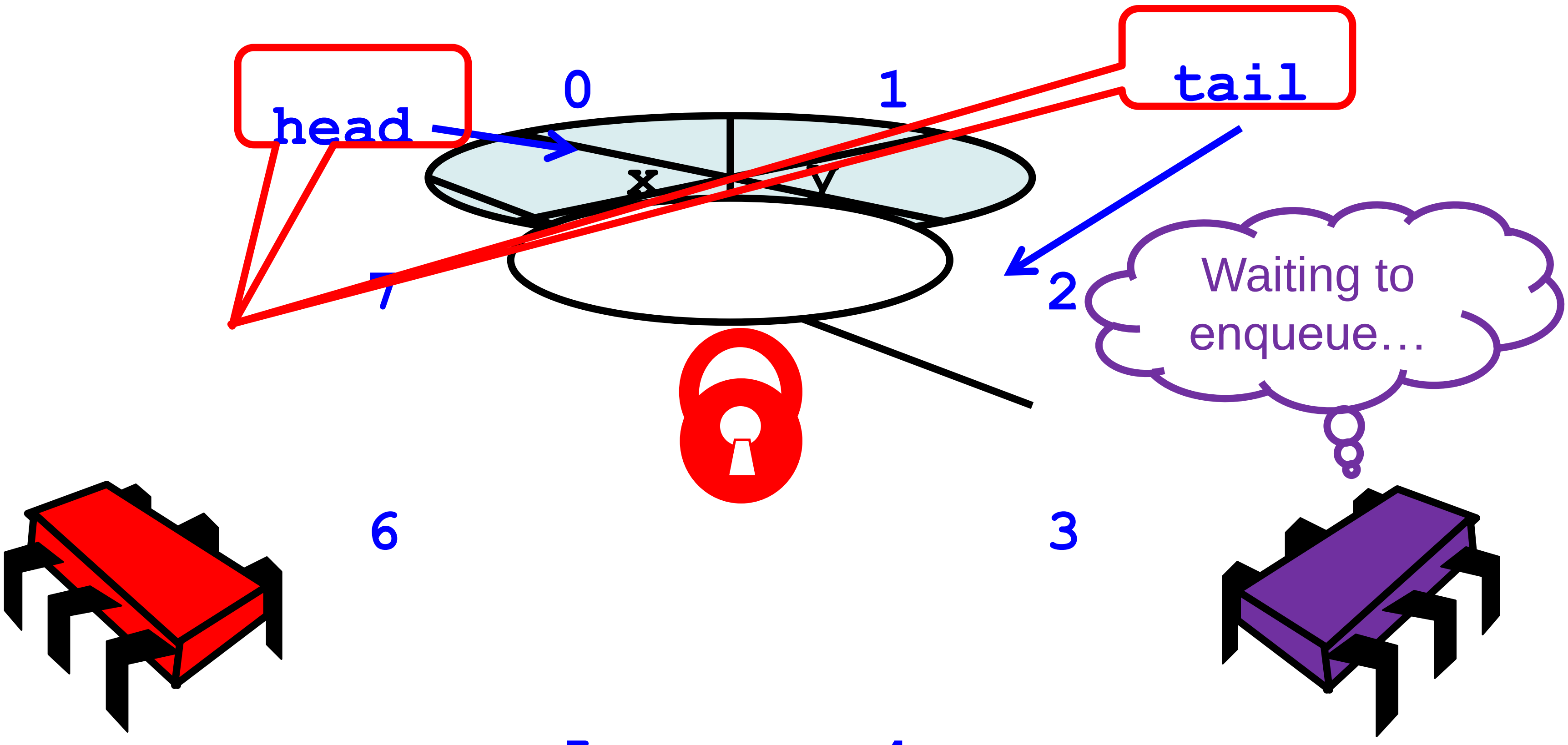
Implementation: deq()

```
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();
  }
}
```

Acquire lock at method start

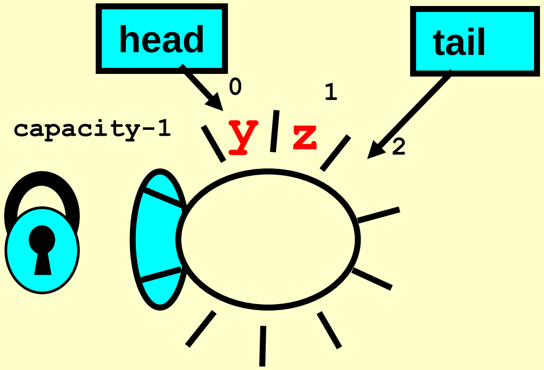


Check if Non-Empty



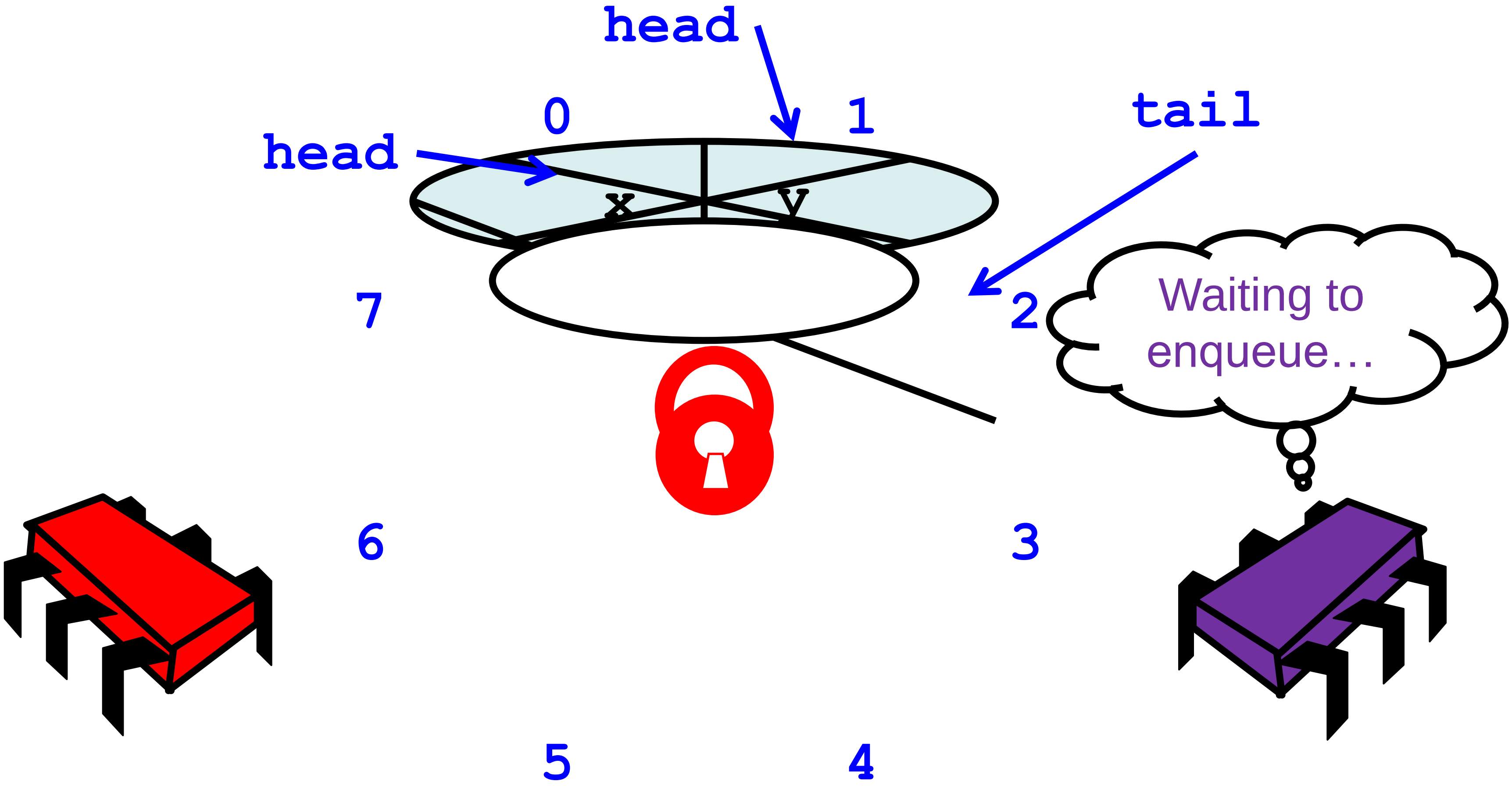
Implementation: deq()

```
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();
  }
}
```



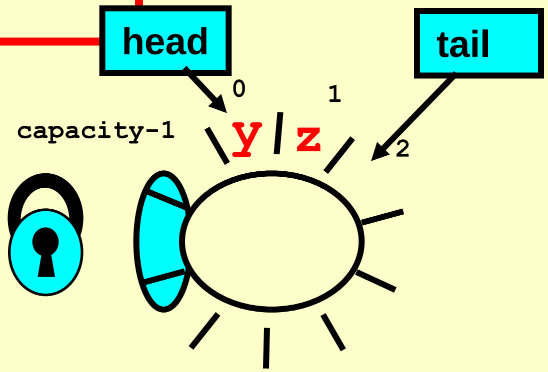
If queue empty throw exception

Modify the Queue



Implementation: deq()

```
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();
  }
}
```



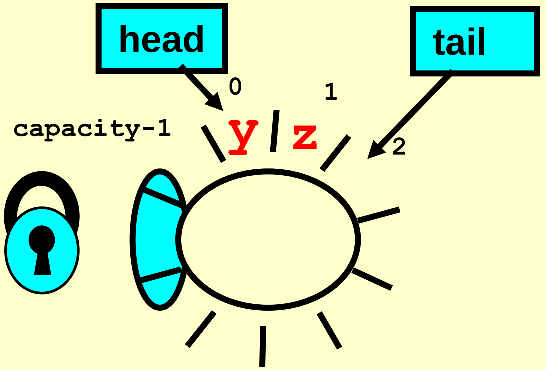
Queue not empty?

Remove item and update head

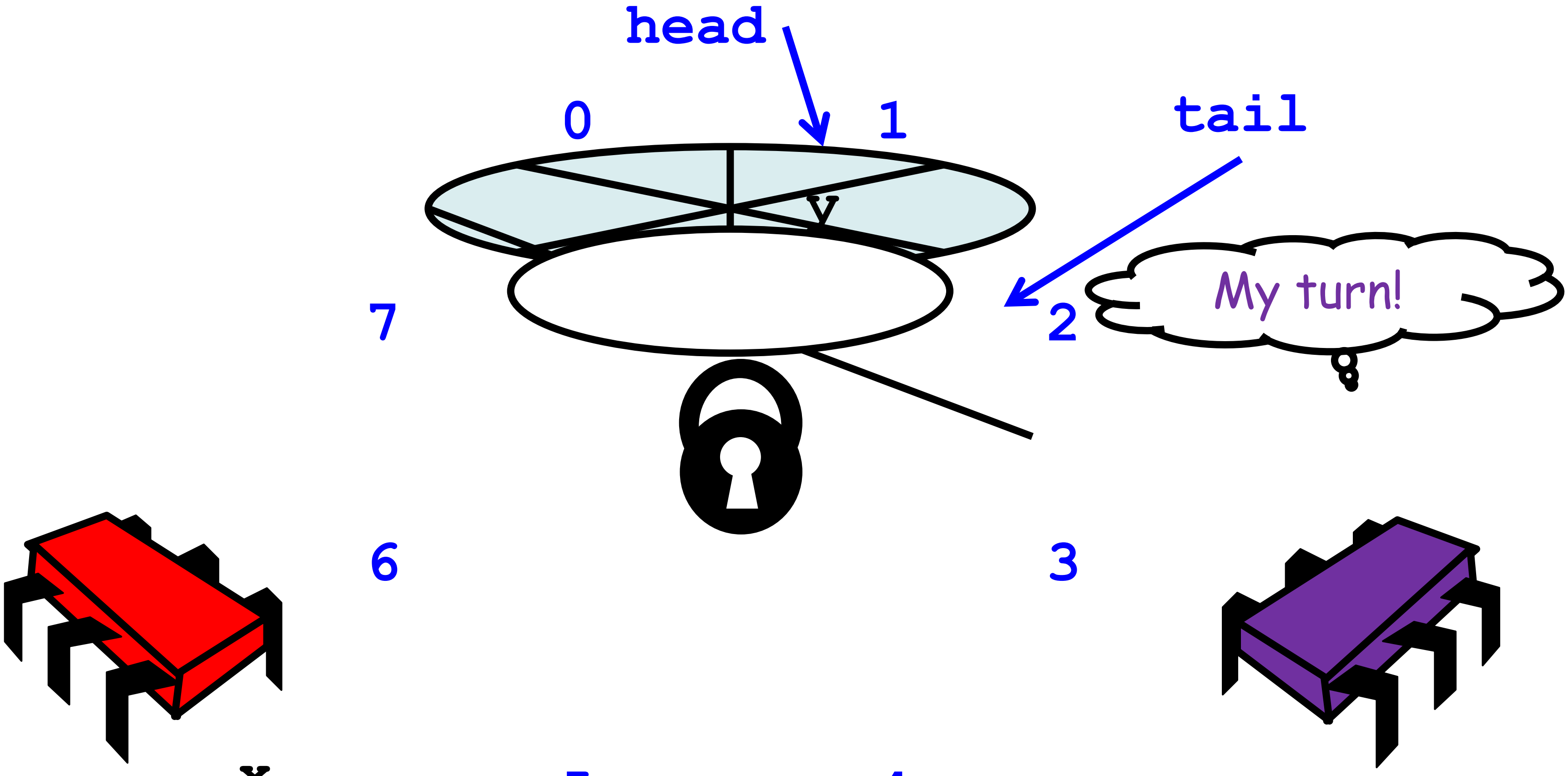
Implementation: deq()

```
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();
  }
}
```

Return result



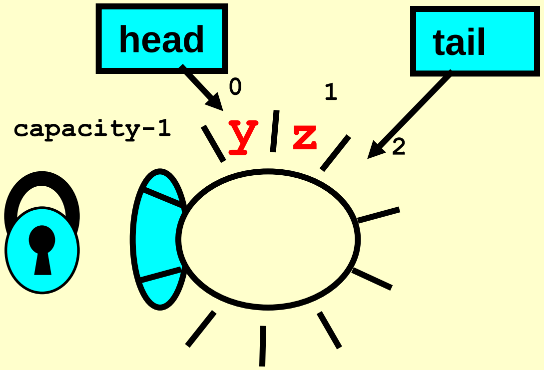
Release the Lock



Implementation: deq()

```
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();
  }
}
```

Release lock no matter what!



Implementation: deq()

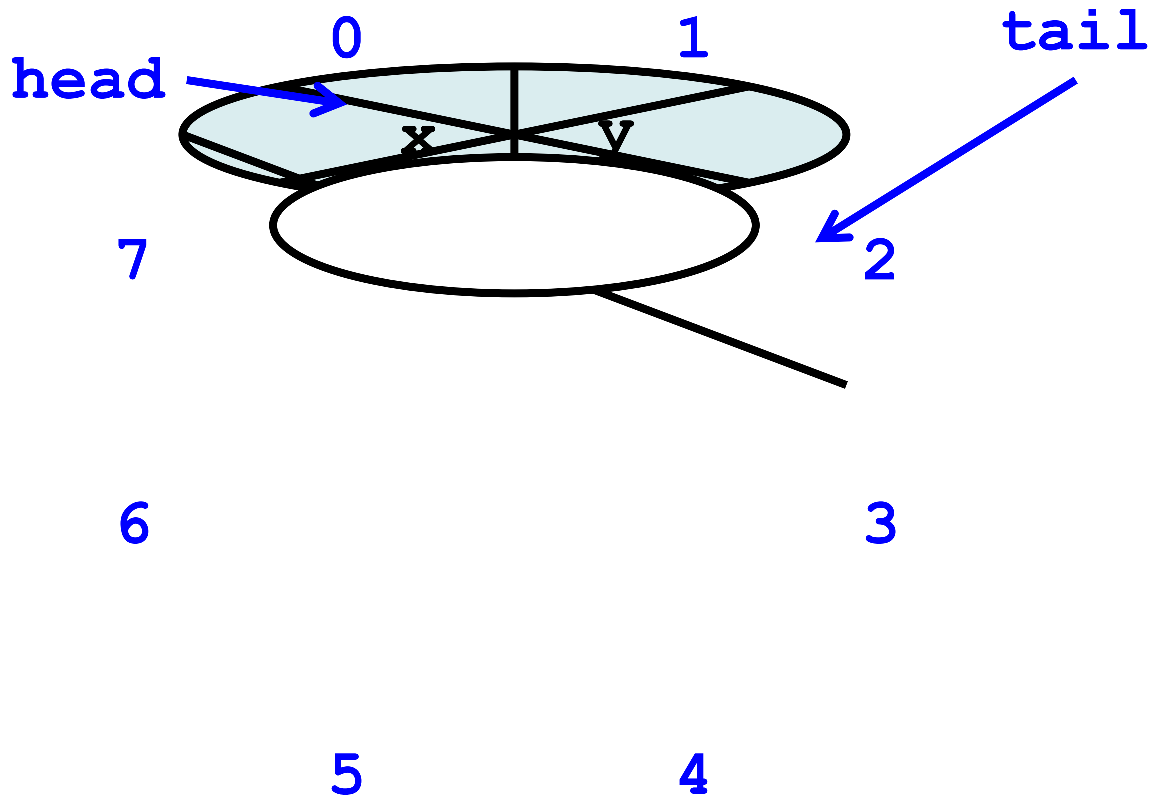
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Should be correct because modifications are mutually exclusive…

Now consider the following implementation

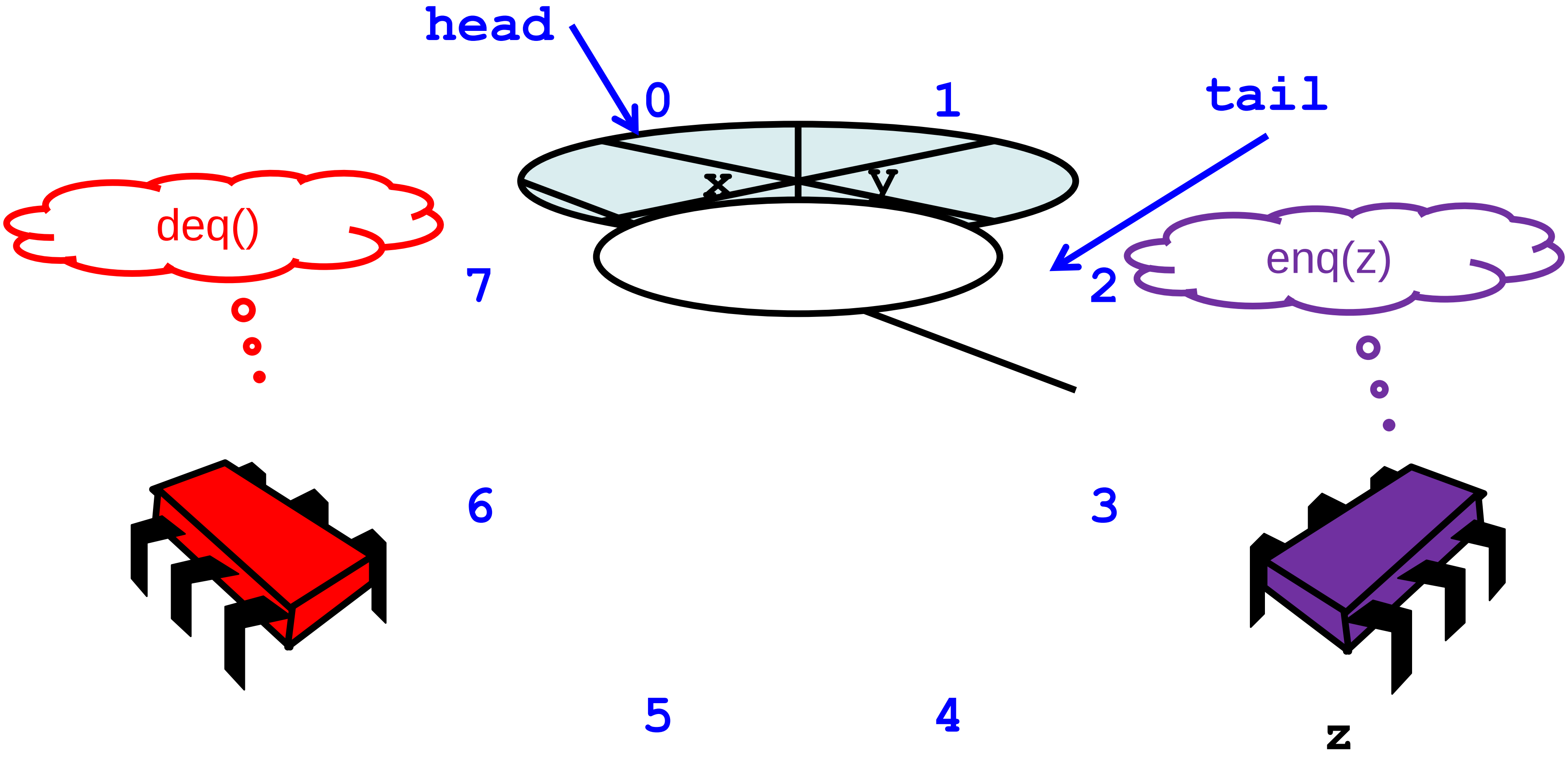
- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

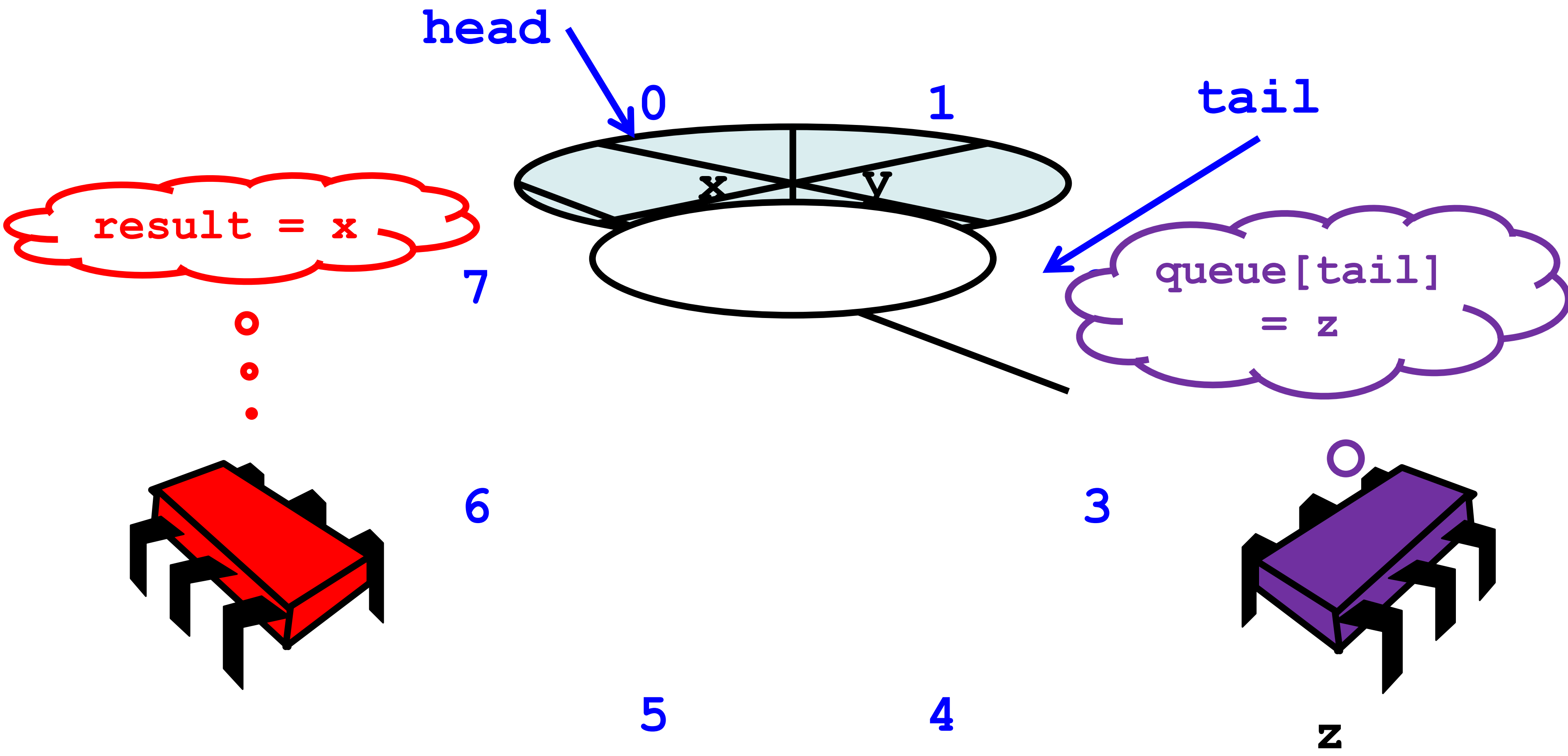


capacity = 8

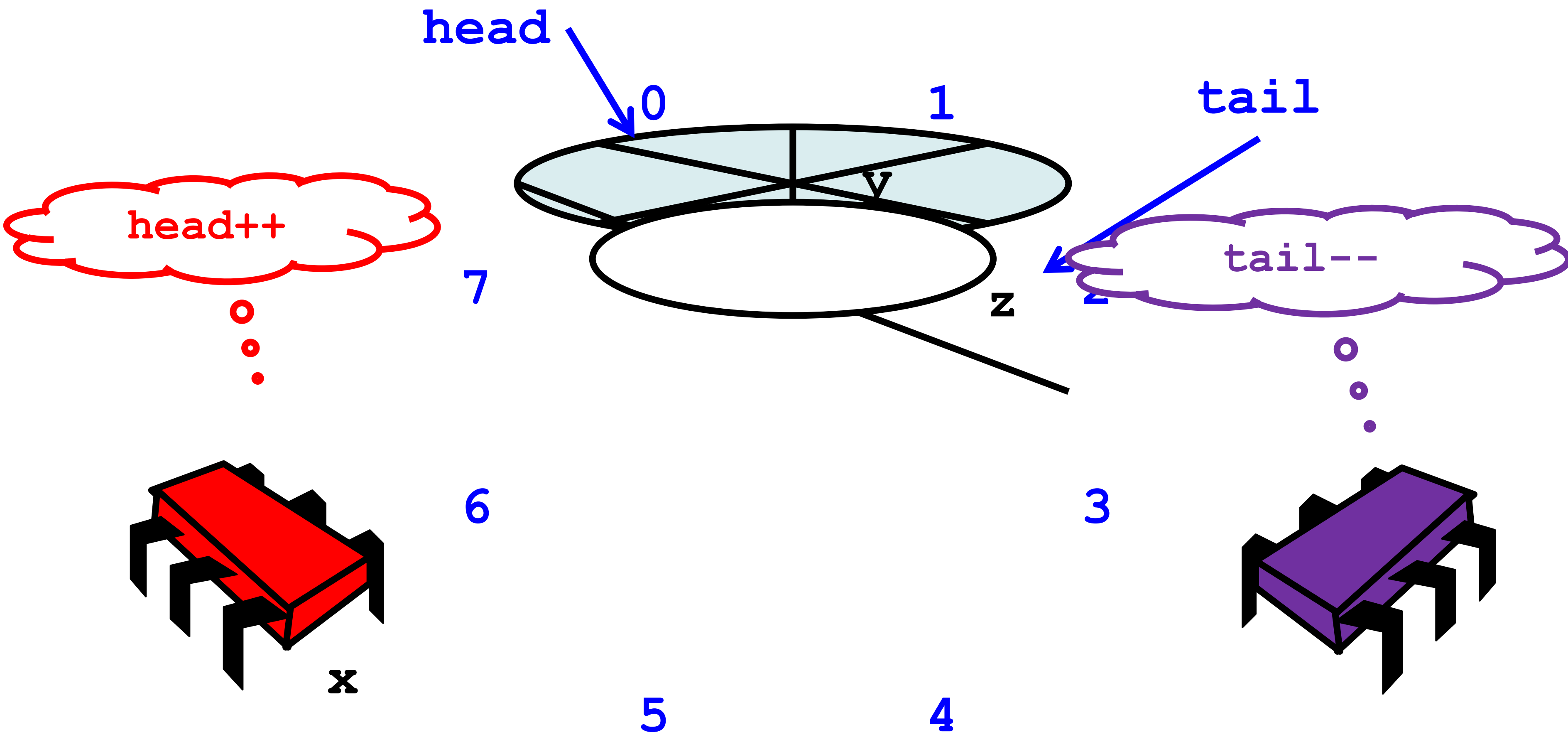
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



Wait-free 2-Thread Queue

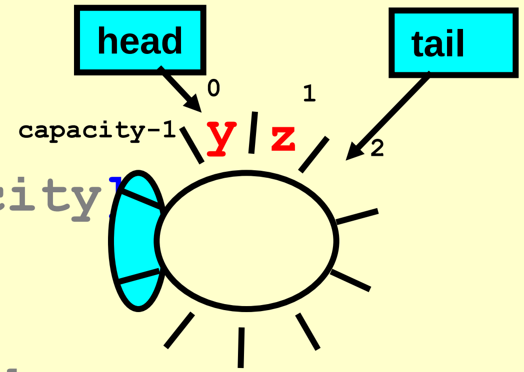


Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```



No lock needed

Wait-free 2-Thread Queue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyExc
    T x = items[hea
    head++;
    returr
  } finall
    lock.ur
  }
}
```

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

What *is* a Concurrent Queue?

- 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 …

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness

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

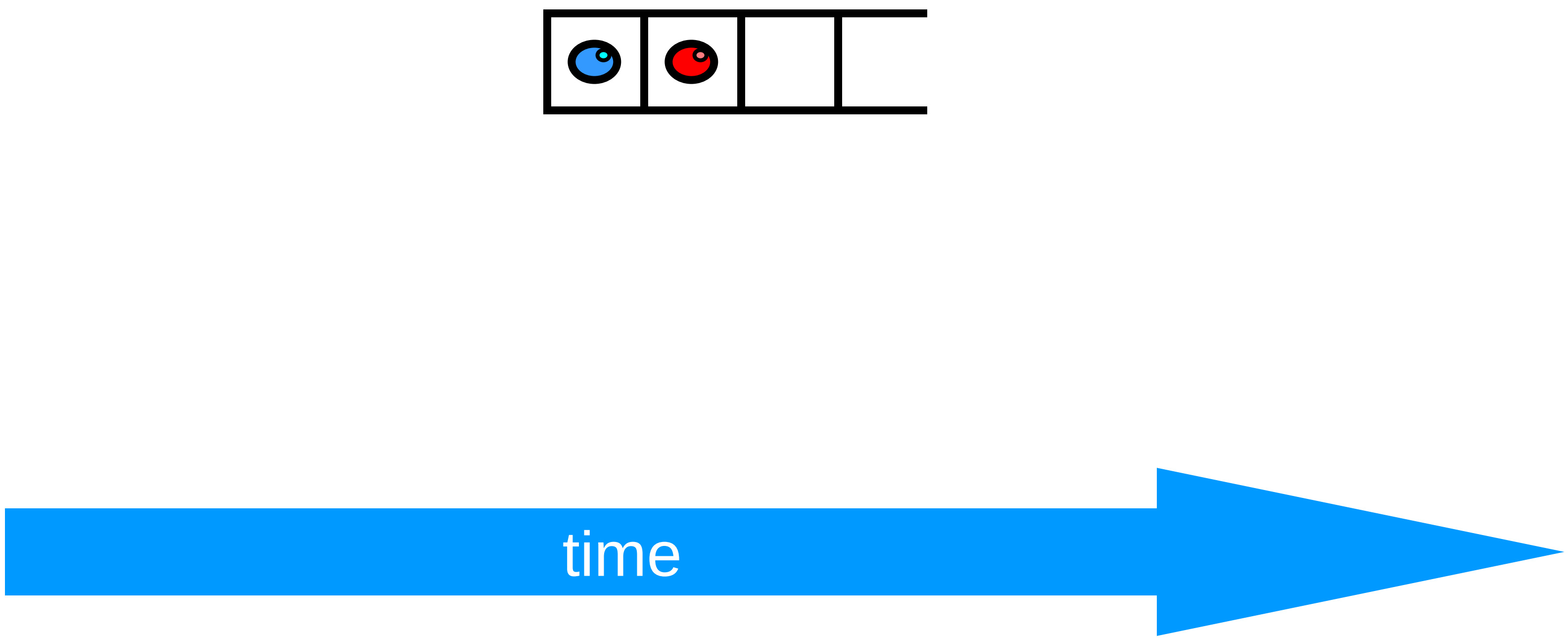
Why Sequential Specifications Totally Rock

- 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

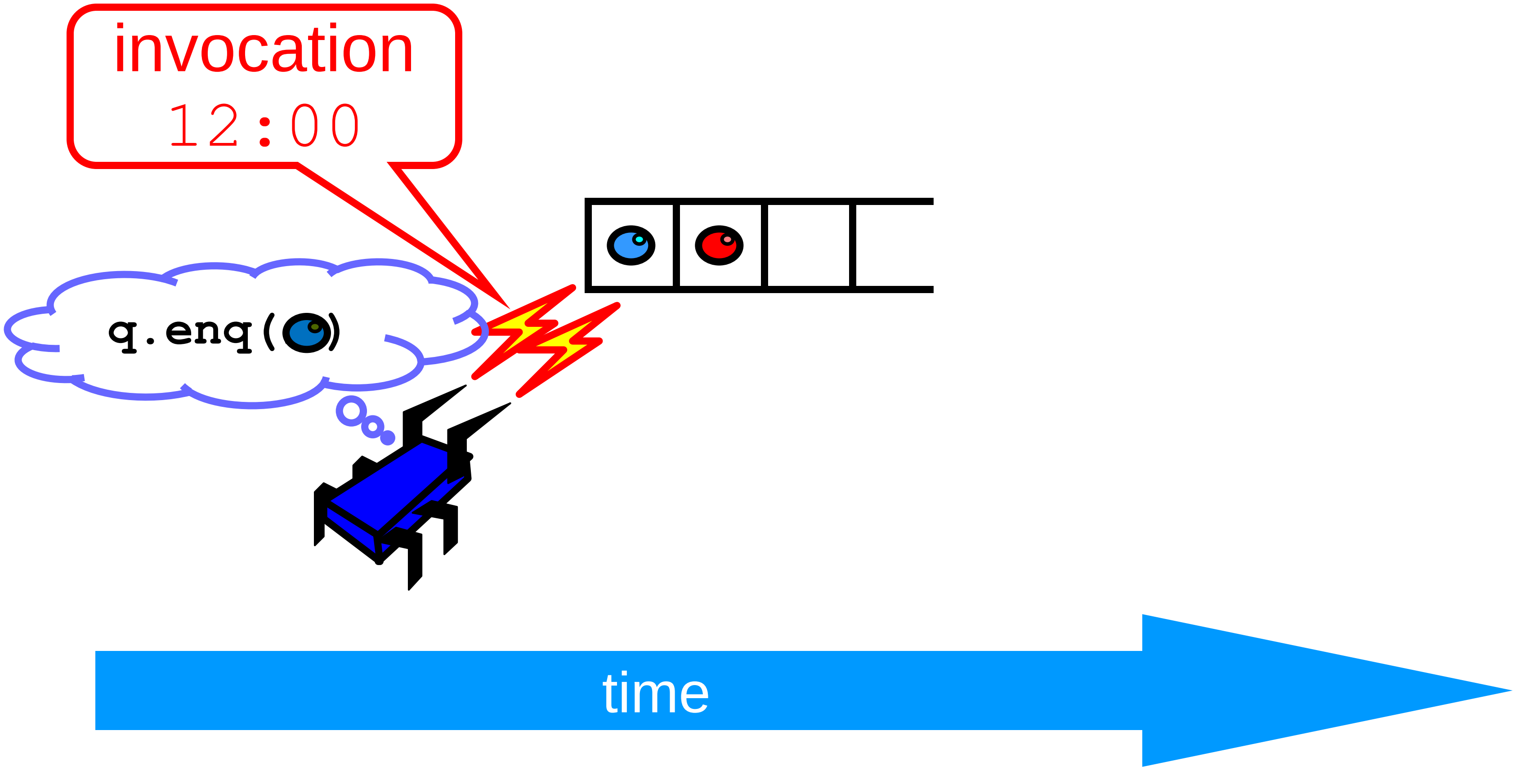
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

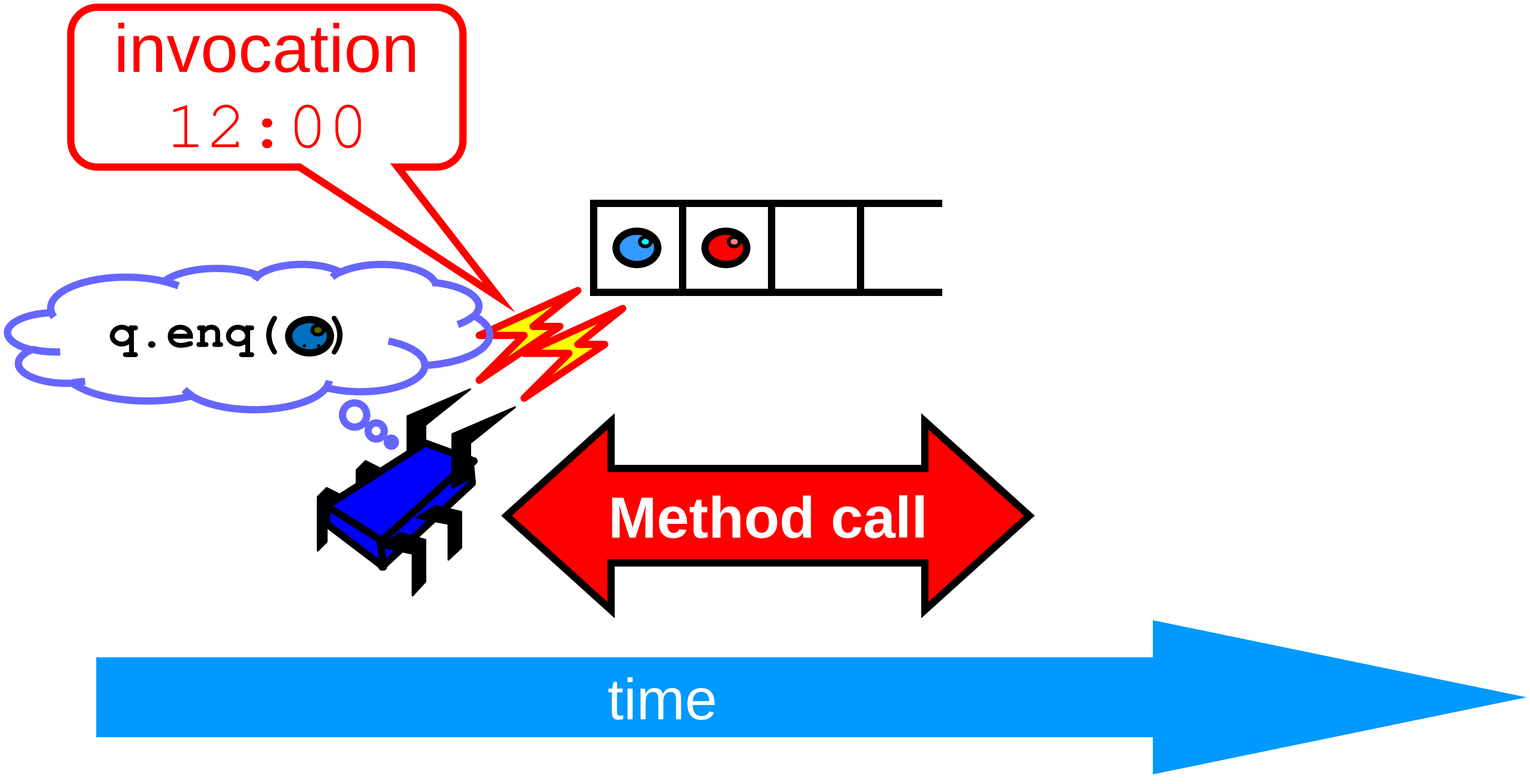
Methods Take Time



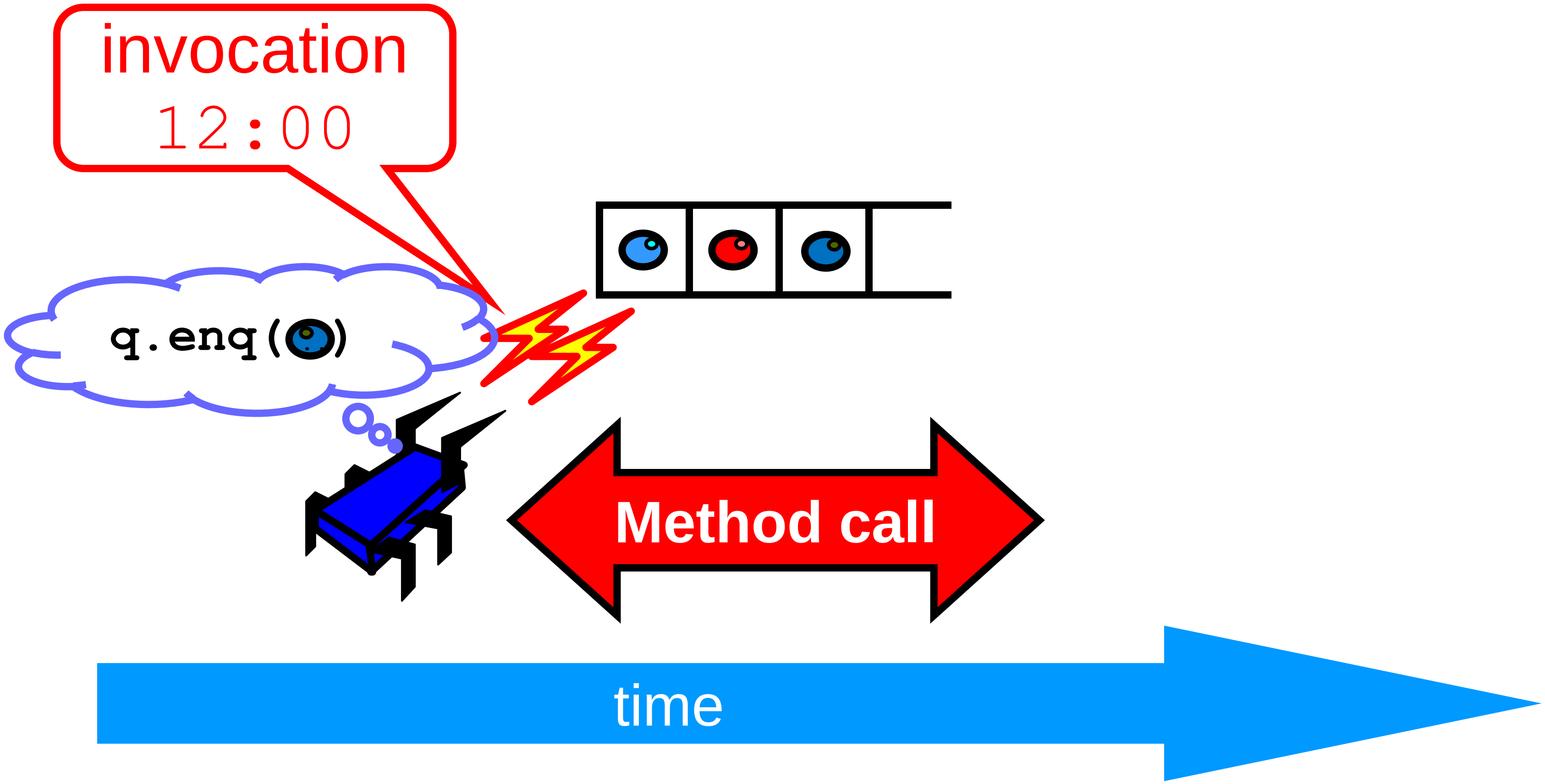
Methods Take Time



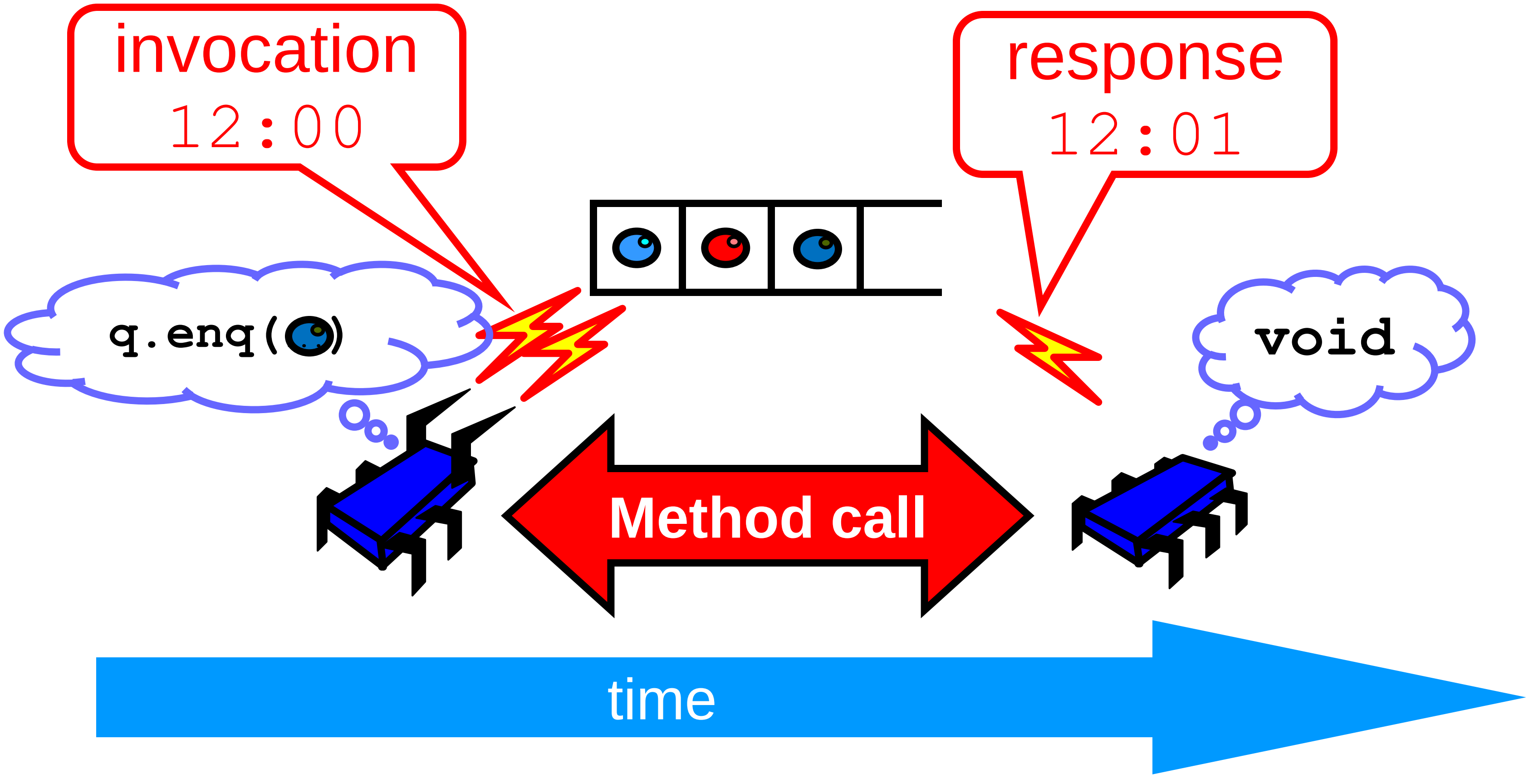
Methods Take Time



Methods Take Time



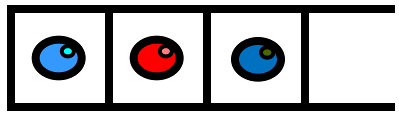
Methods Take Time



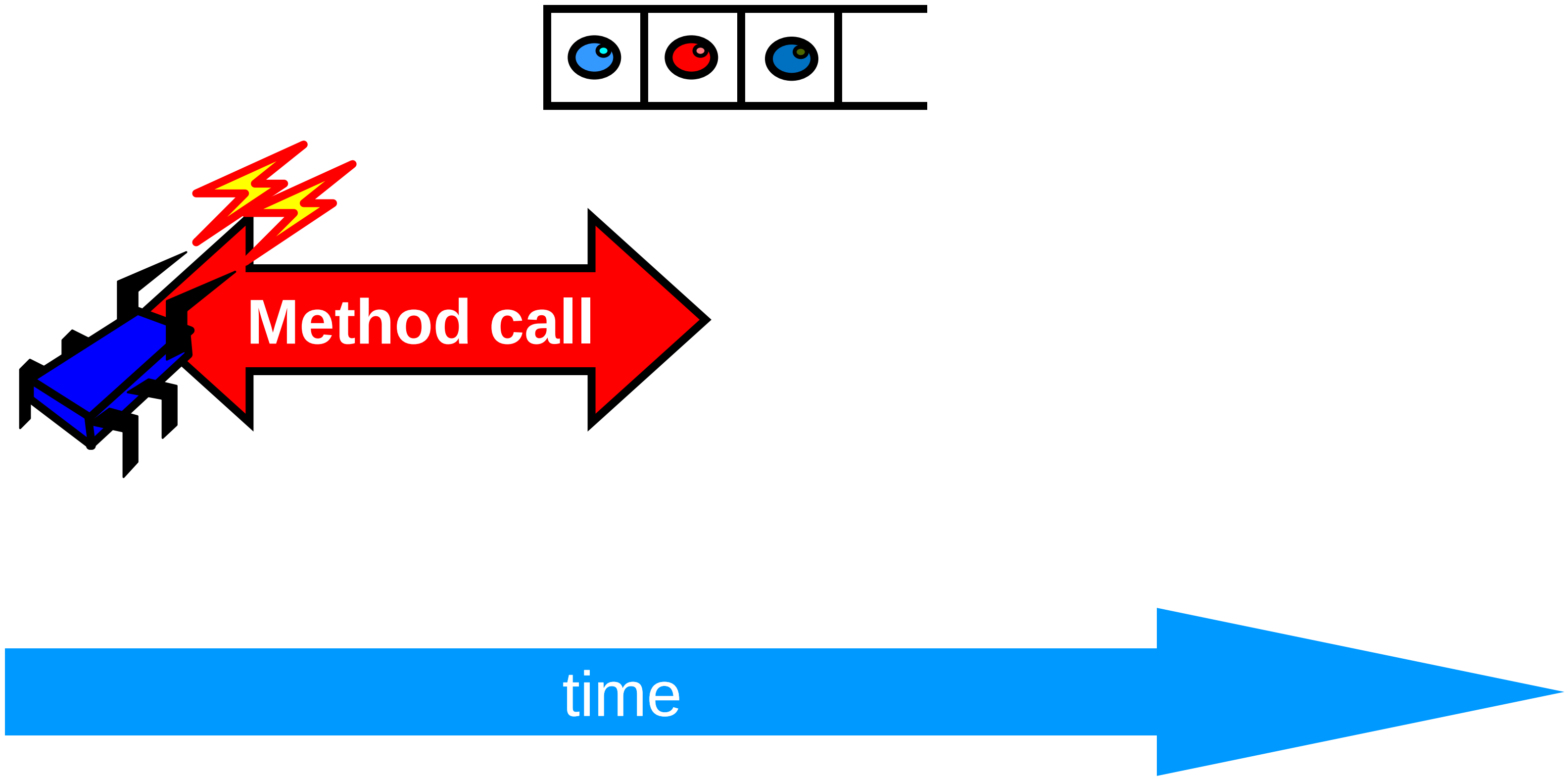
Sequential vs Concurrent

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

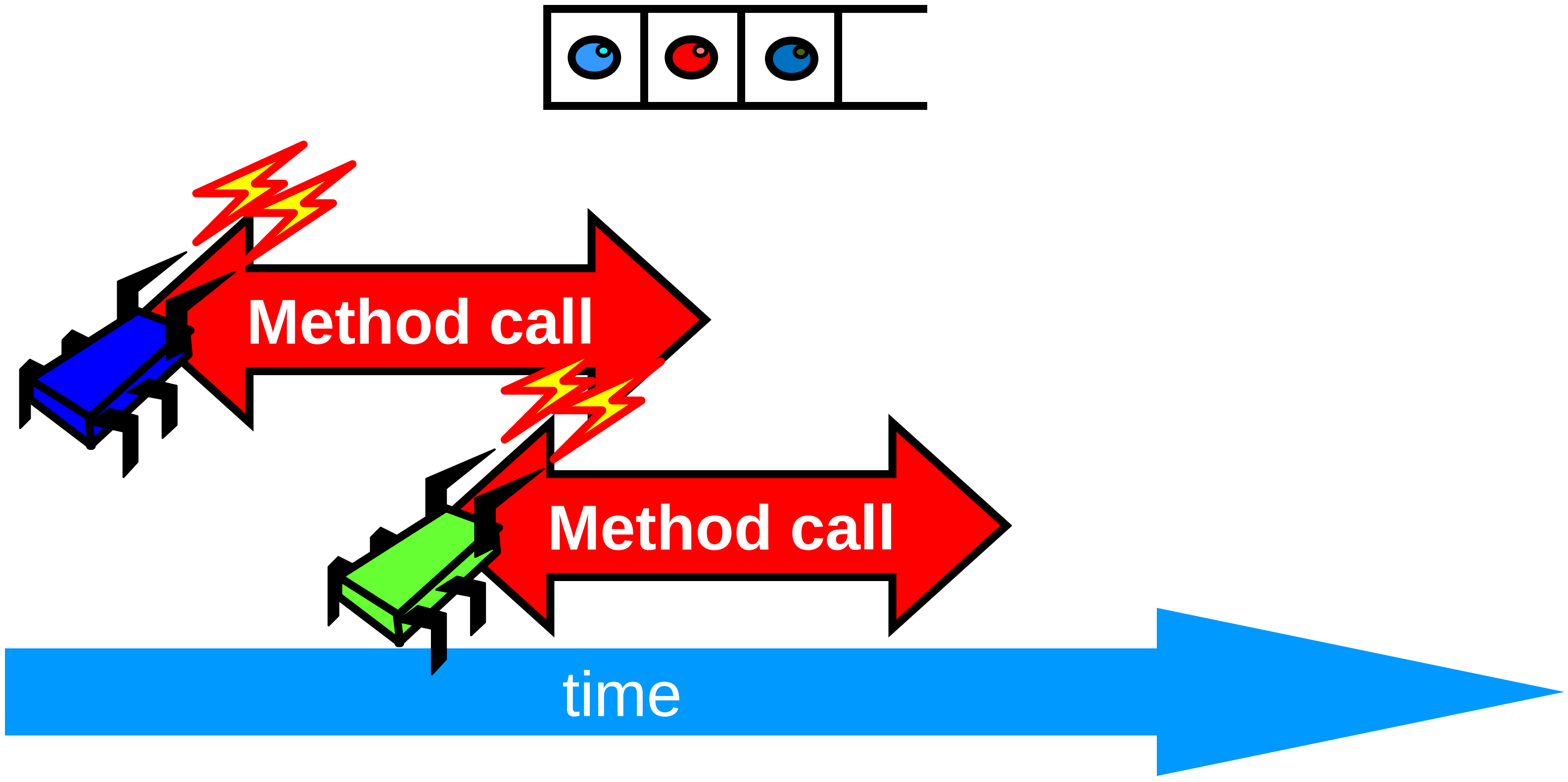
Concurrent Methods Take Overlapping Time



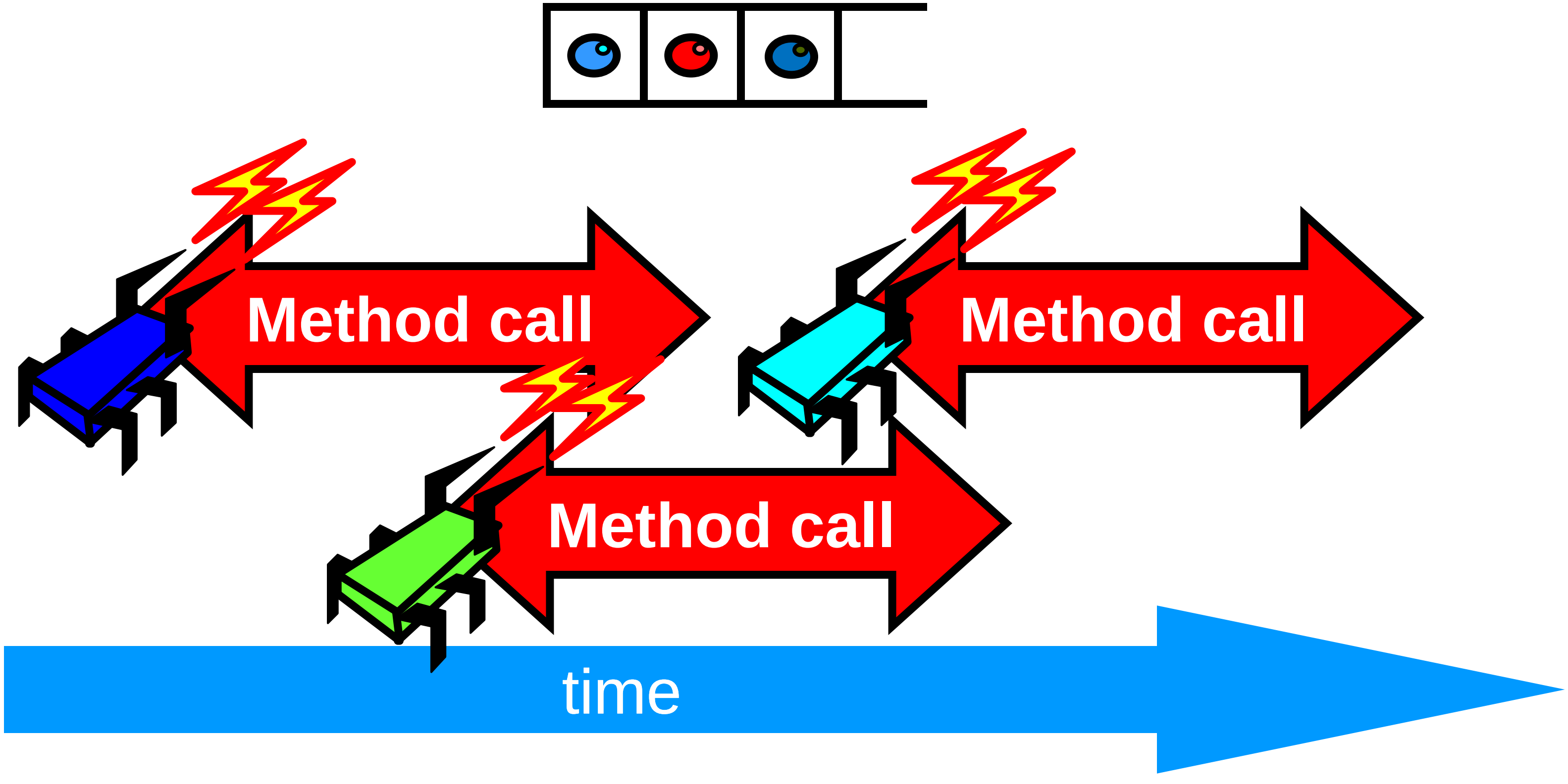
Concurrent Methods Take Overlapping Time



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 `enq`s overlap?
    - Two `deq`s? `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 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();
  }
}
```

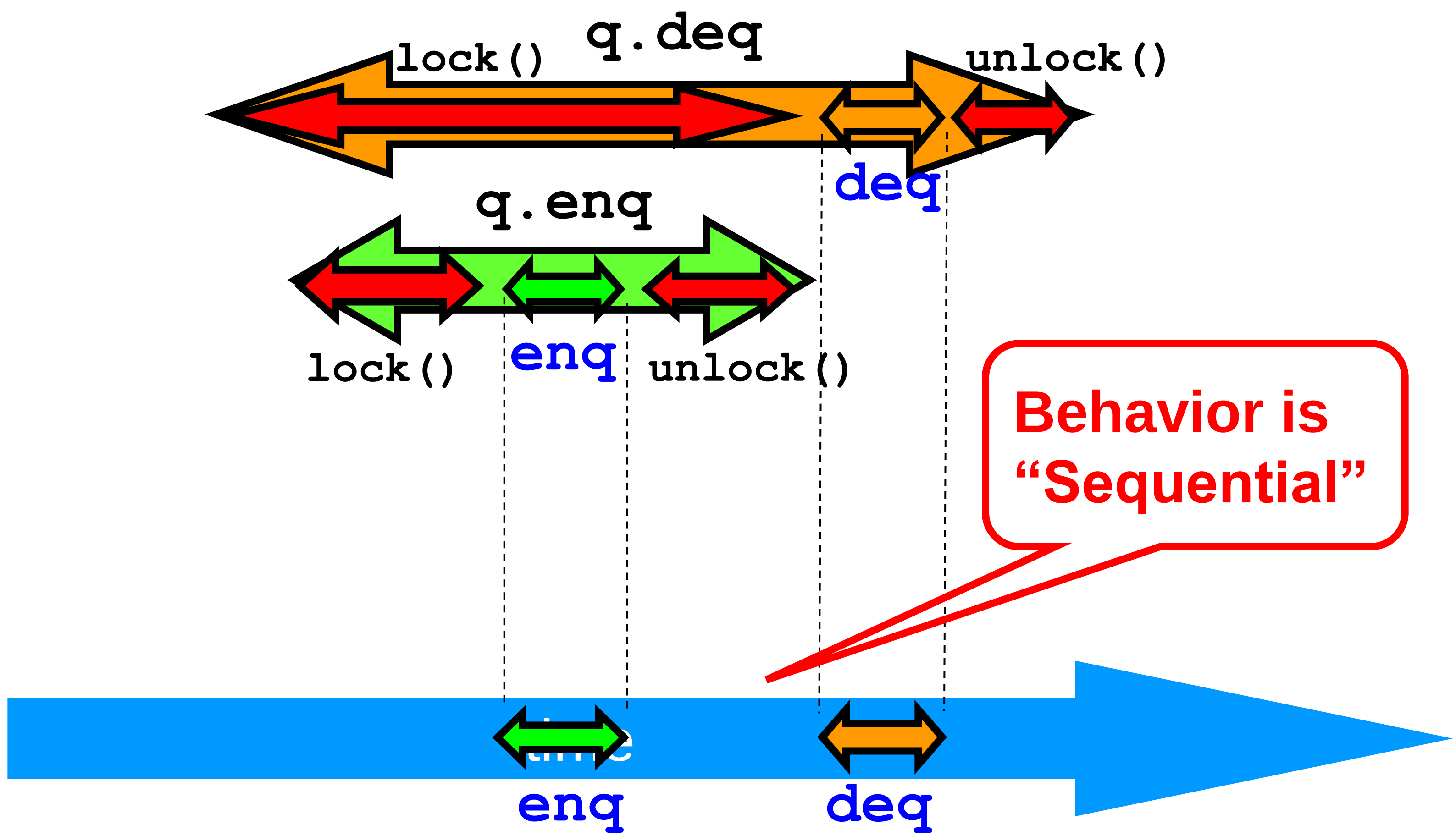
Intuitively…

```
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 queue modifications are mutually exclusive

Intuitively

Lets capture the idea of describing the concurrent via the sequential



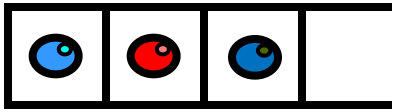
Linearizability

- 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™**

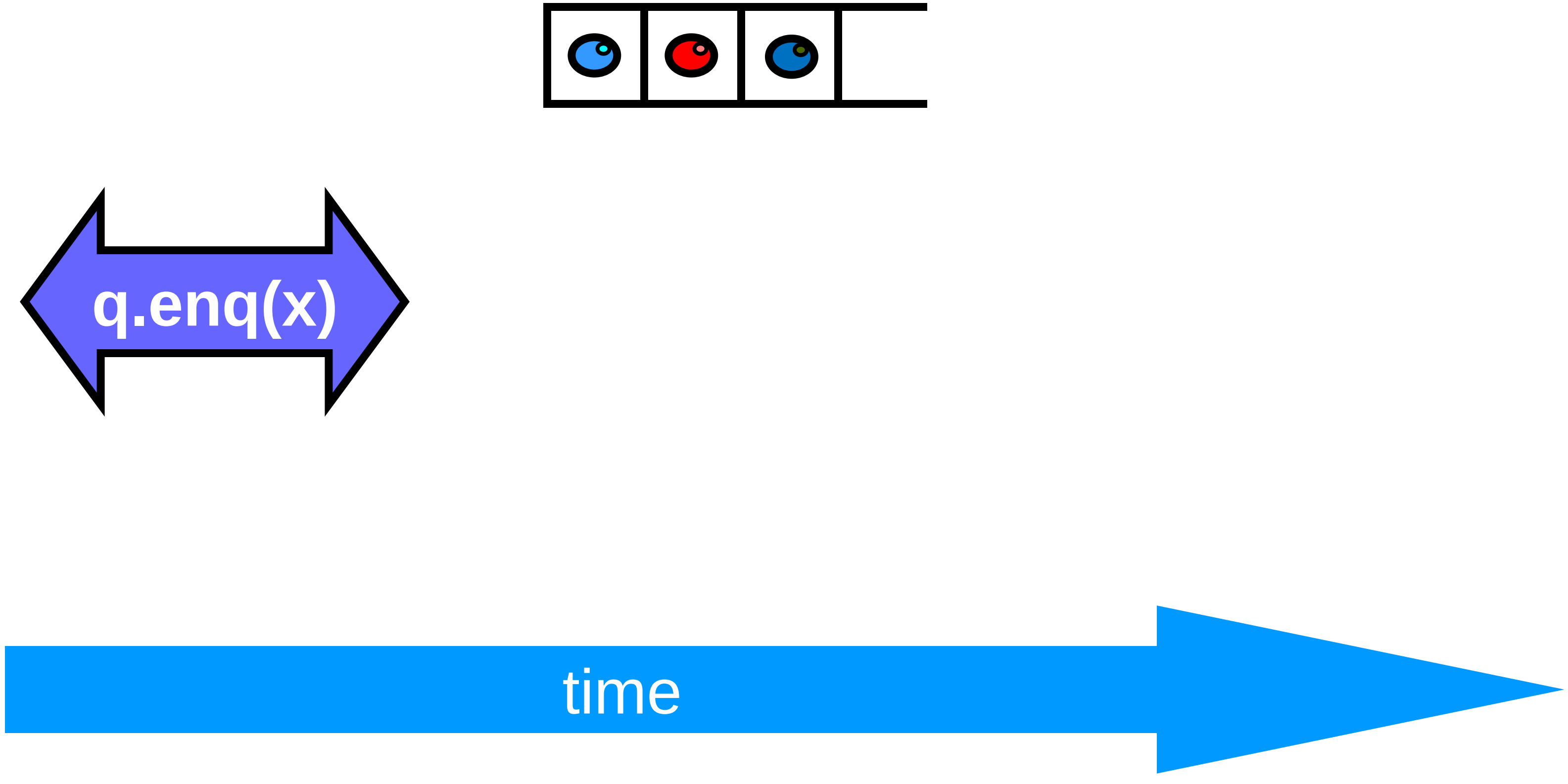
Is it really about the object?

- 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

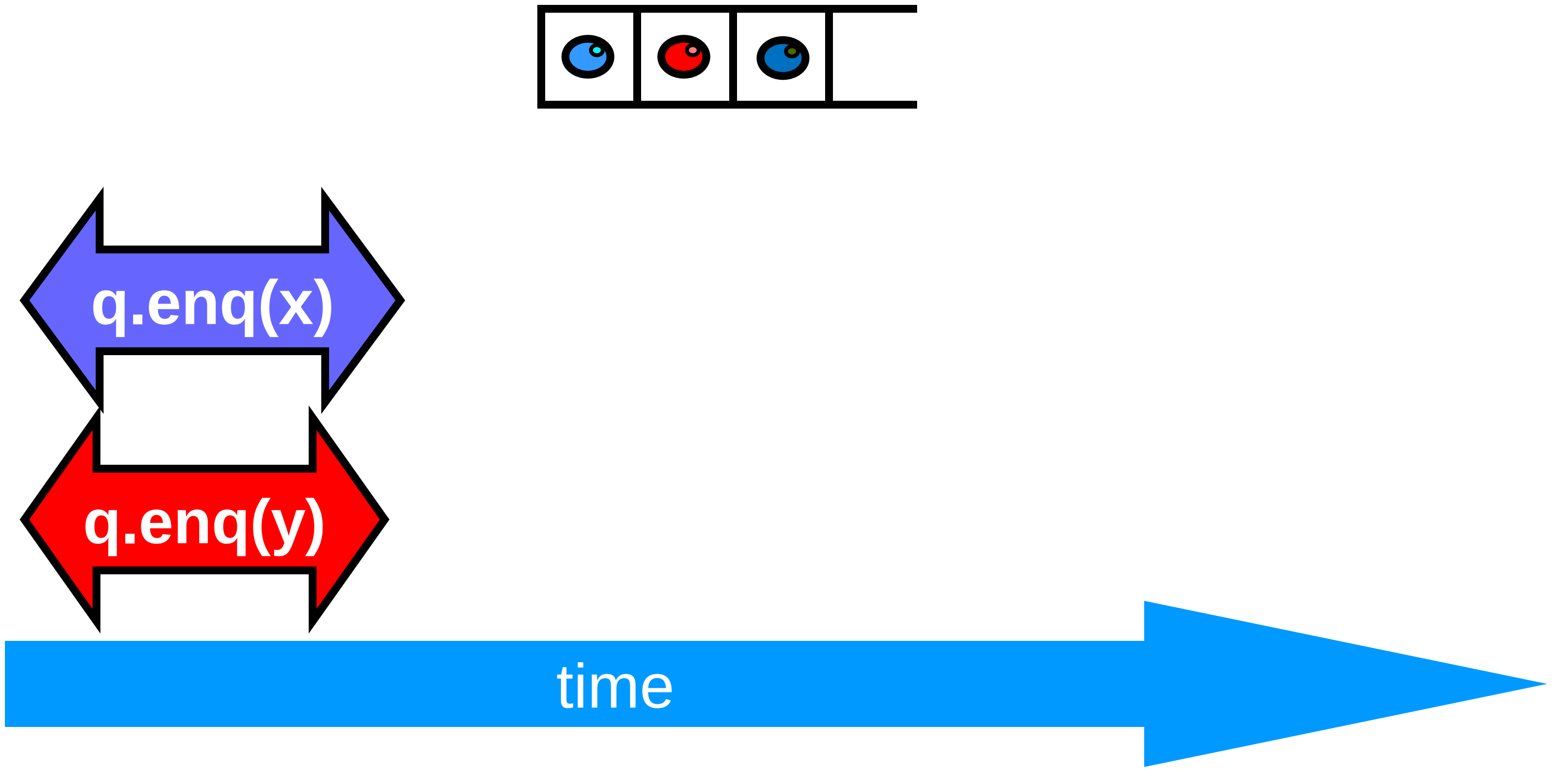
Example



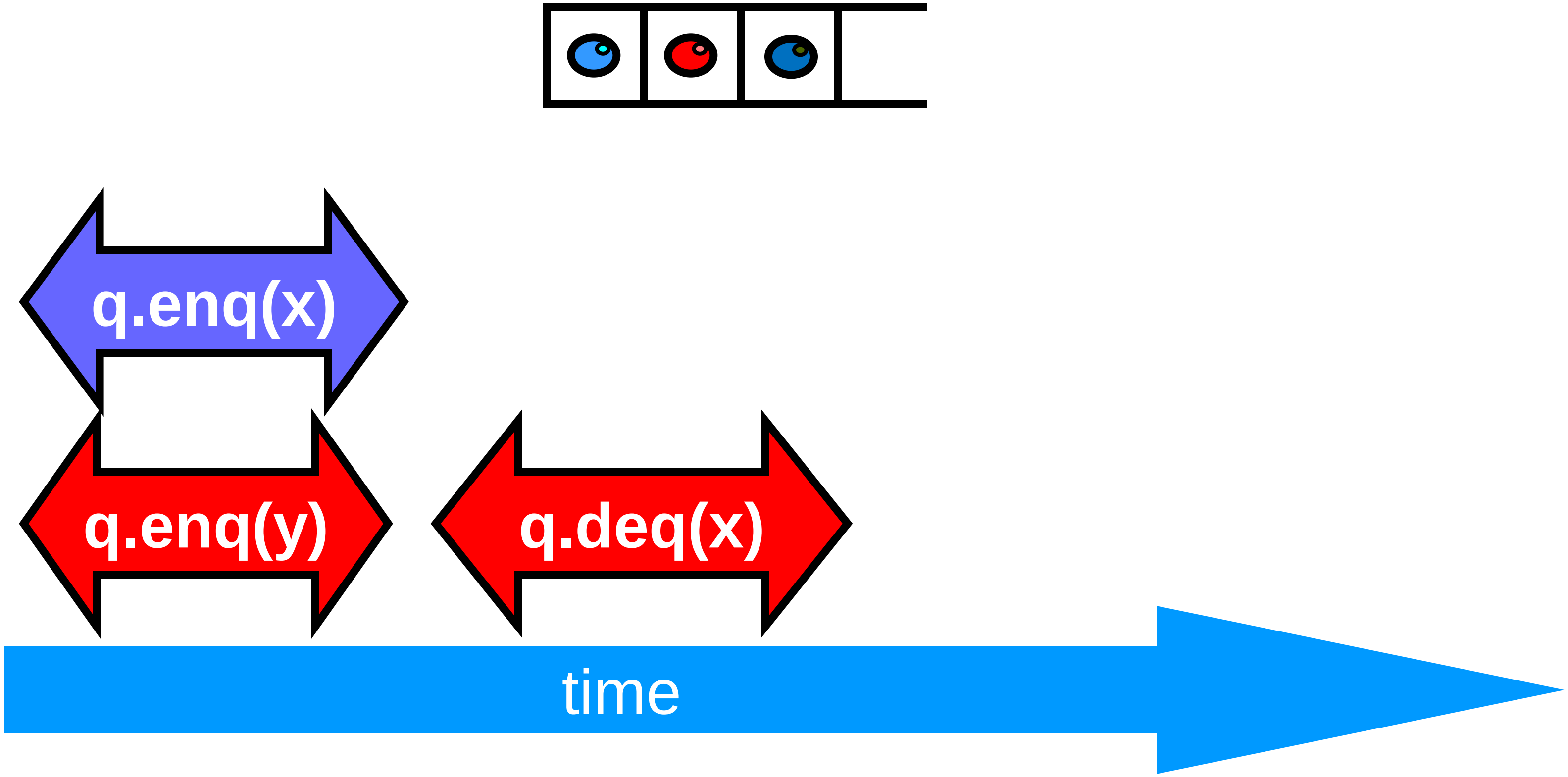
Example



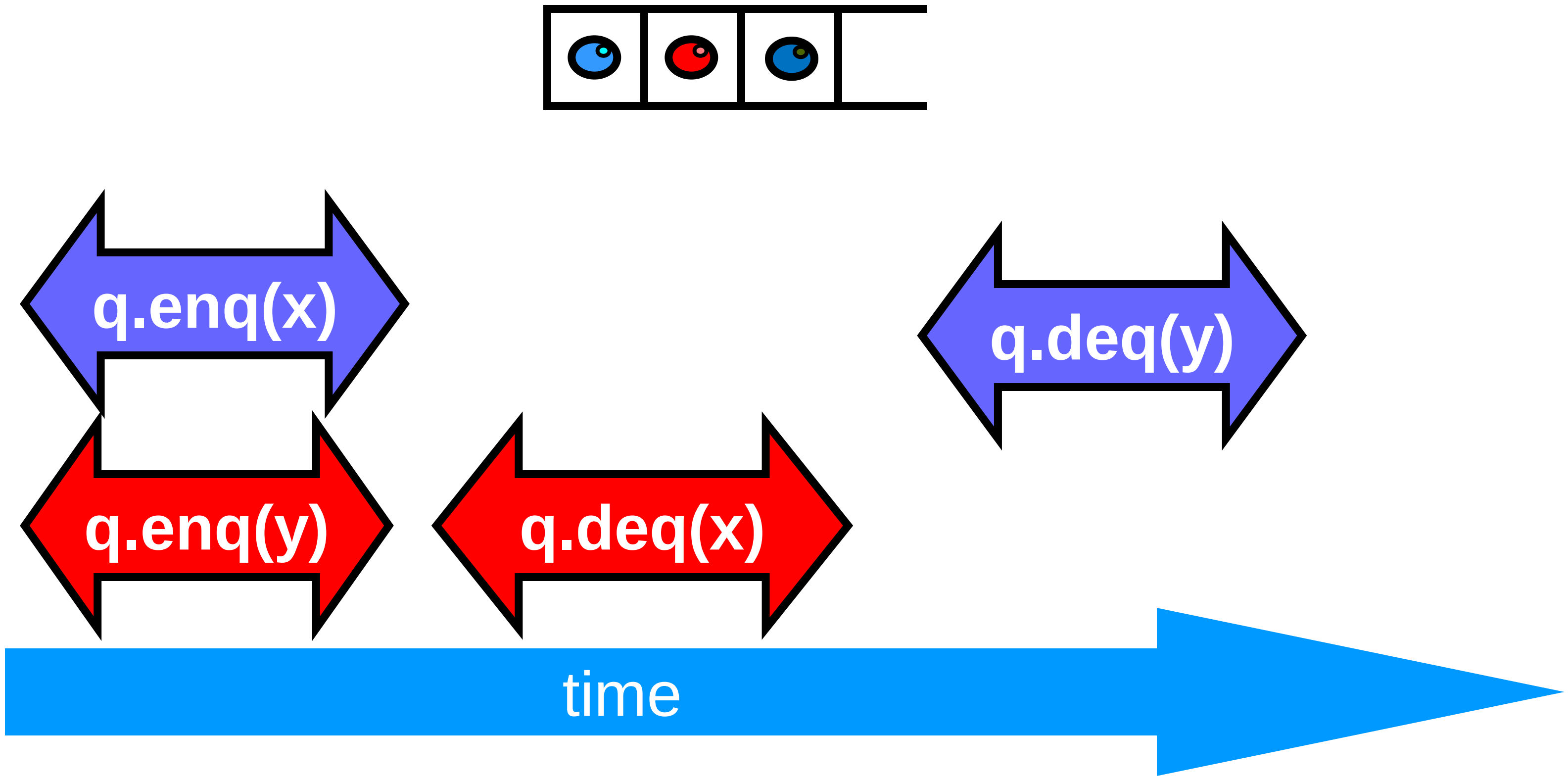
Example



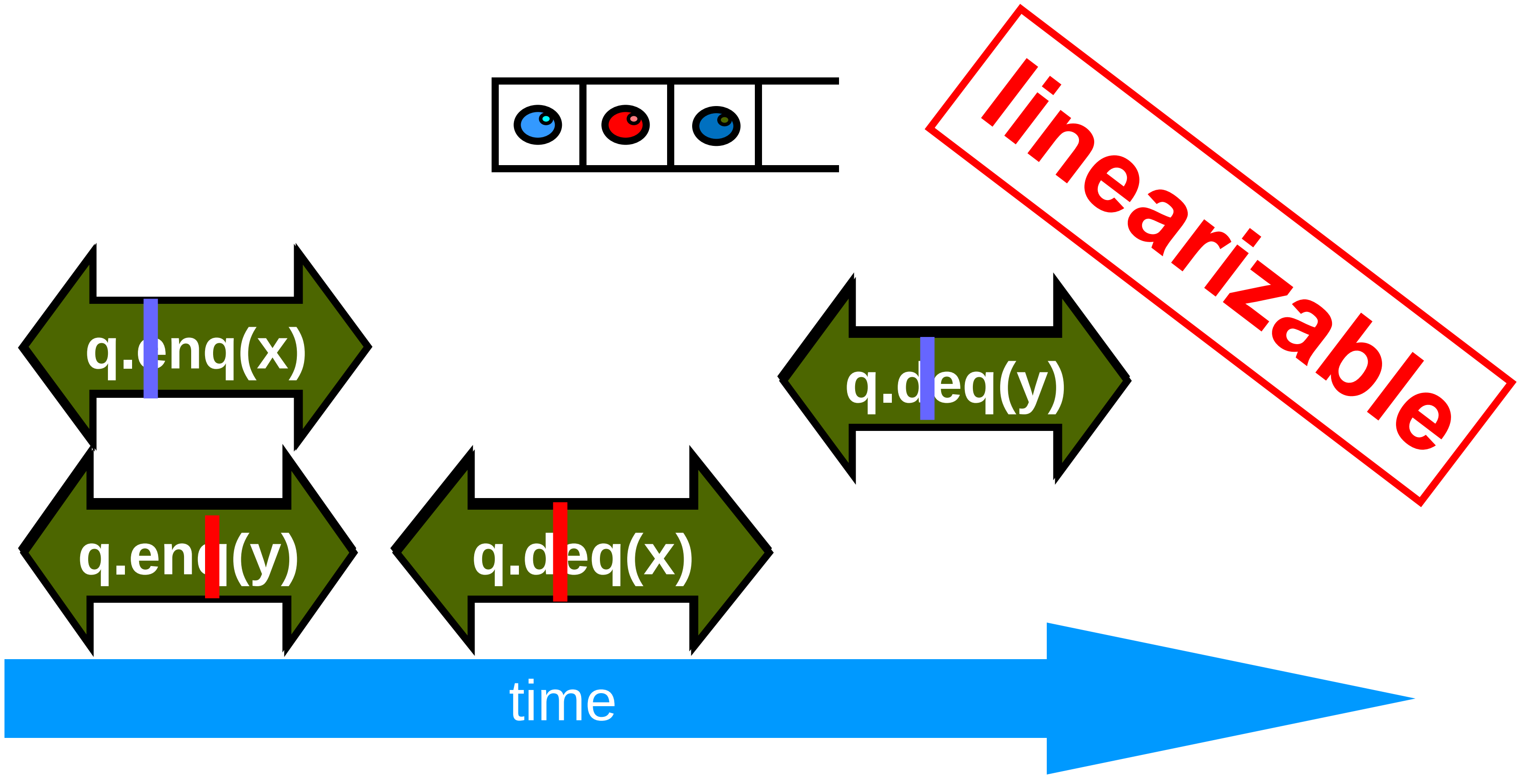
Example



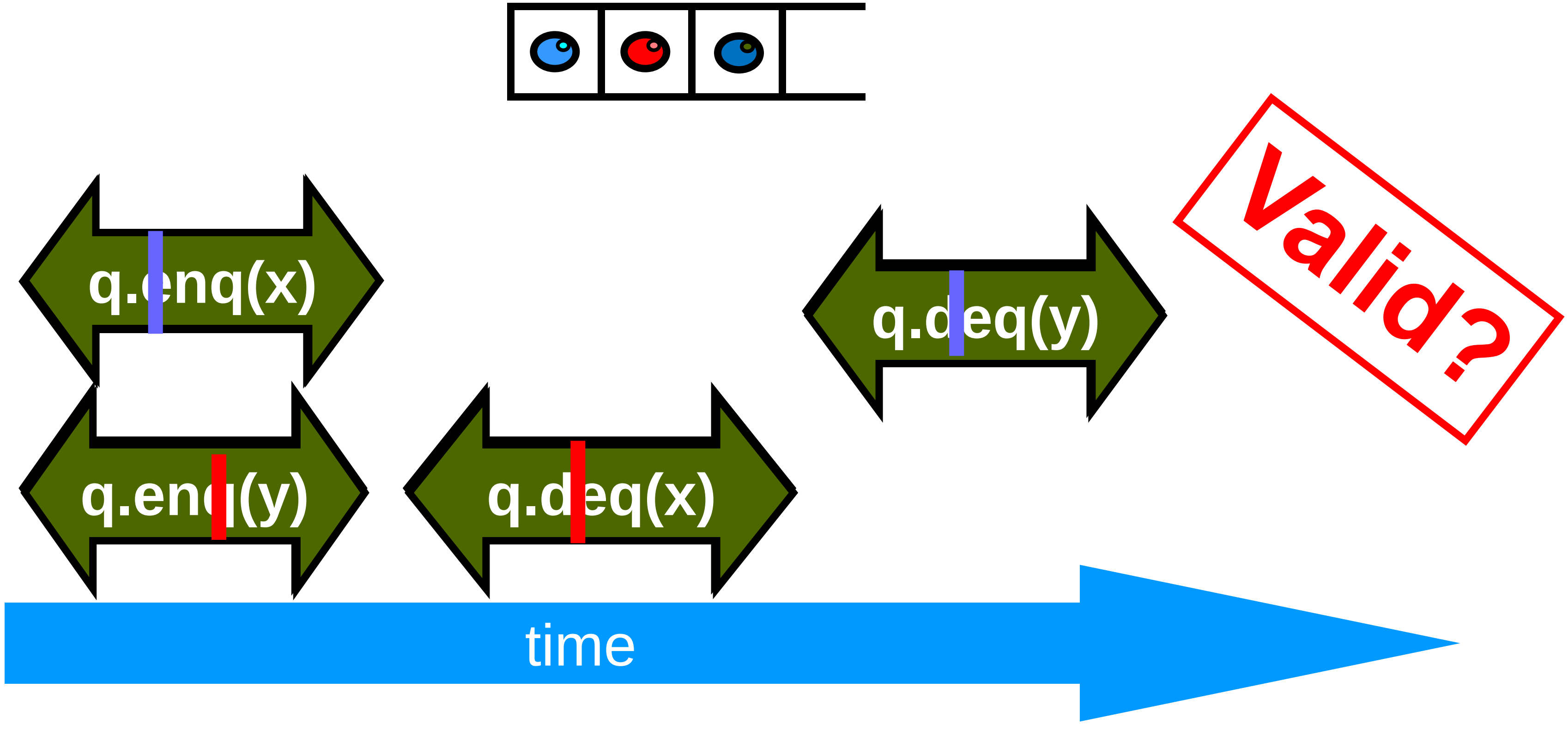
Example



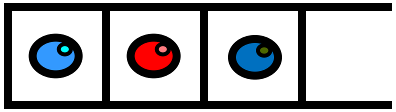
Example



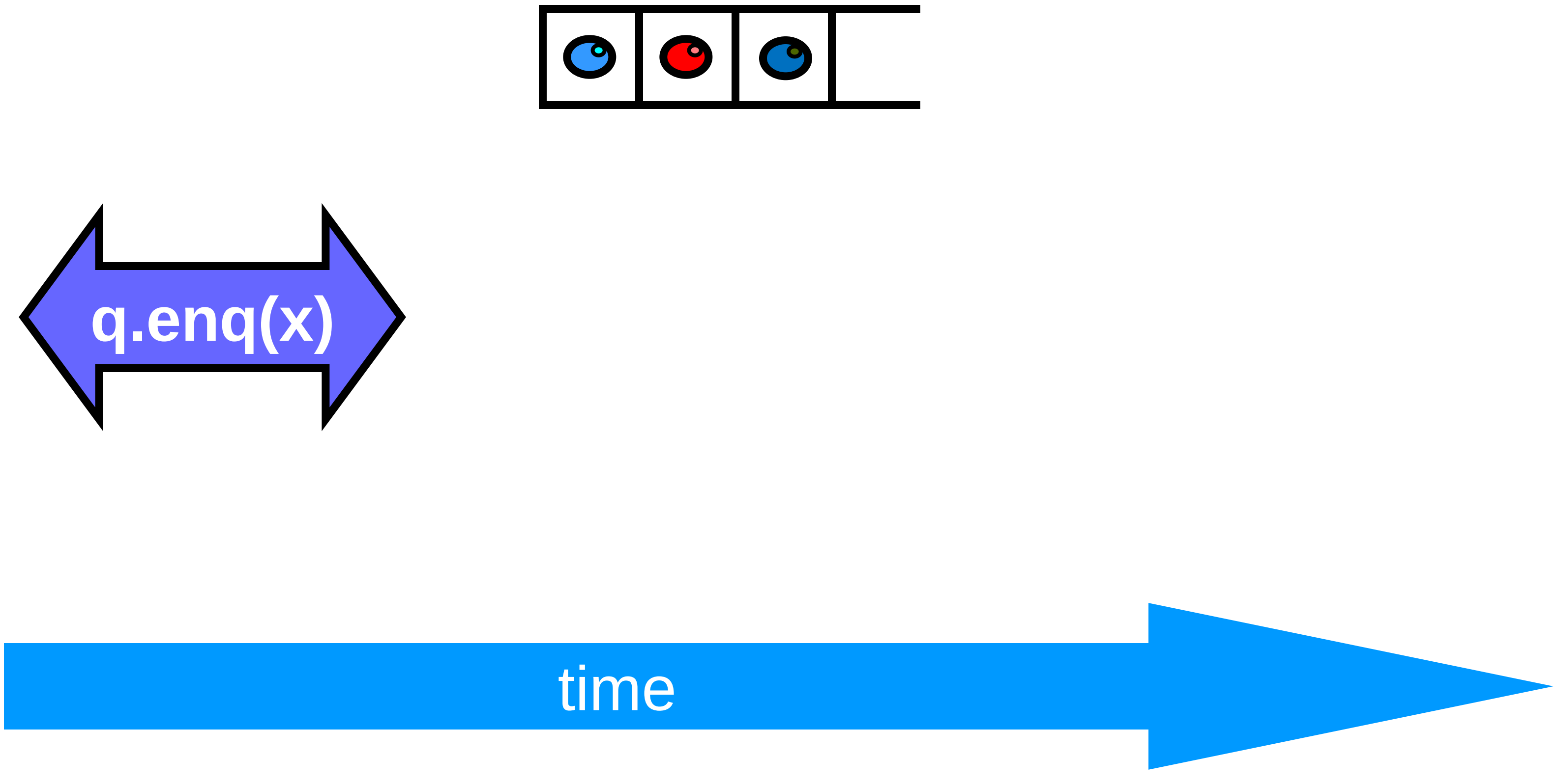
Example



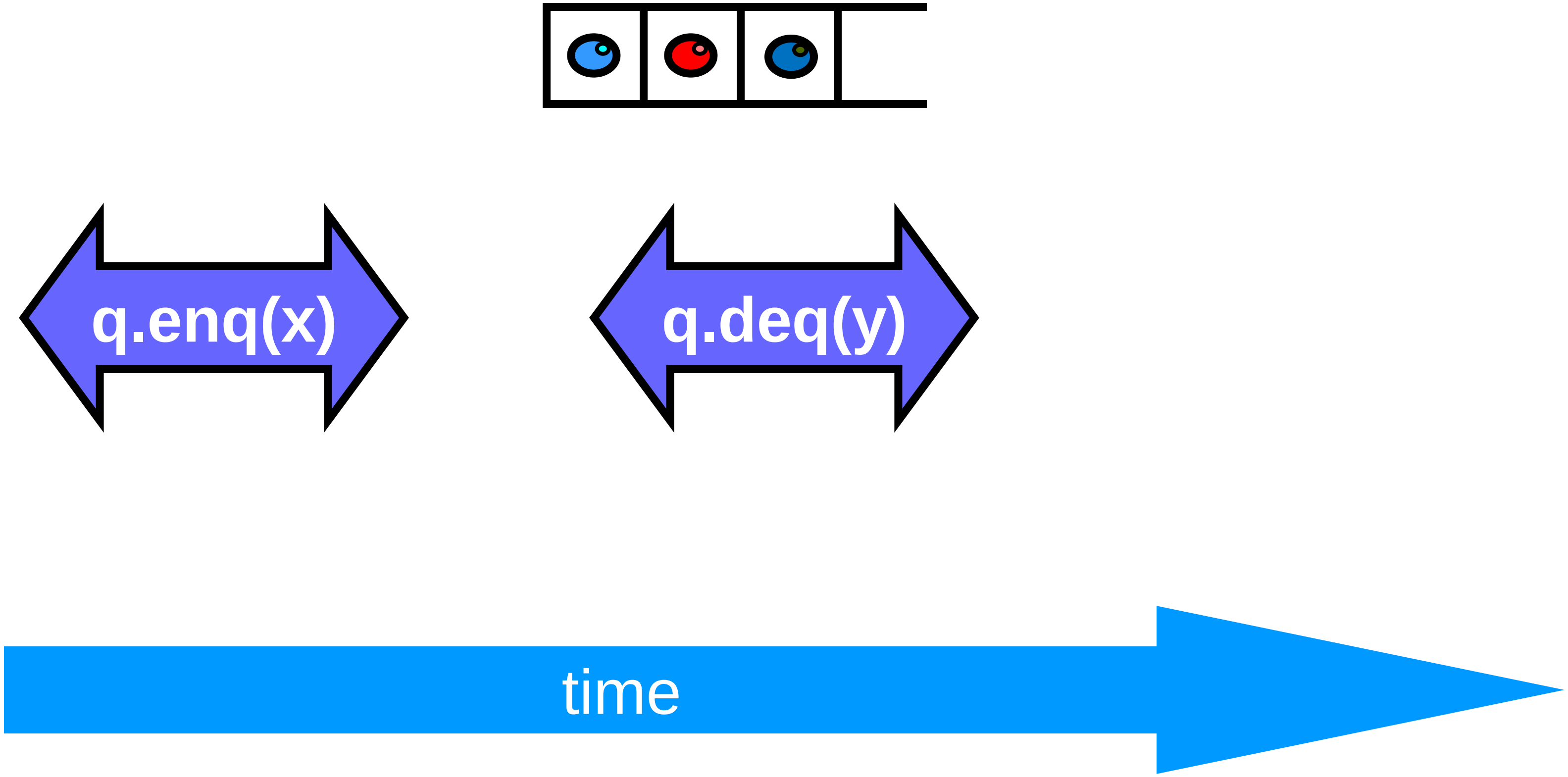
Example



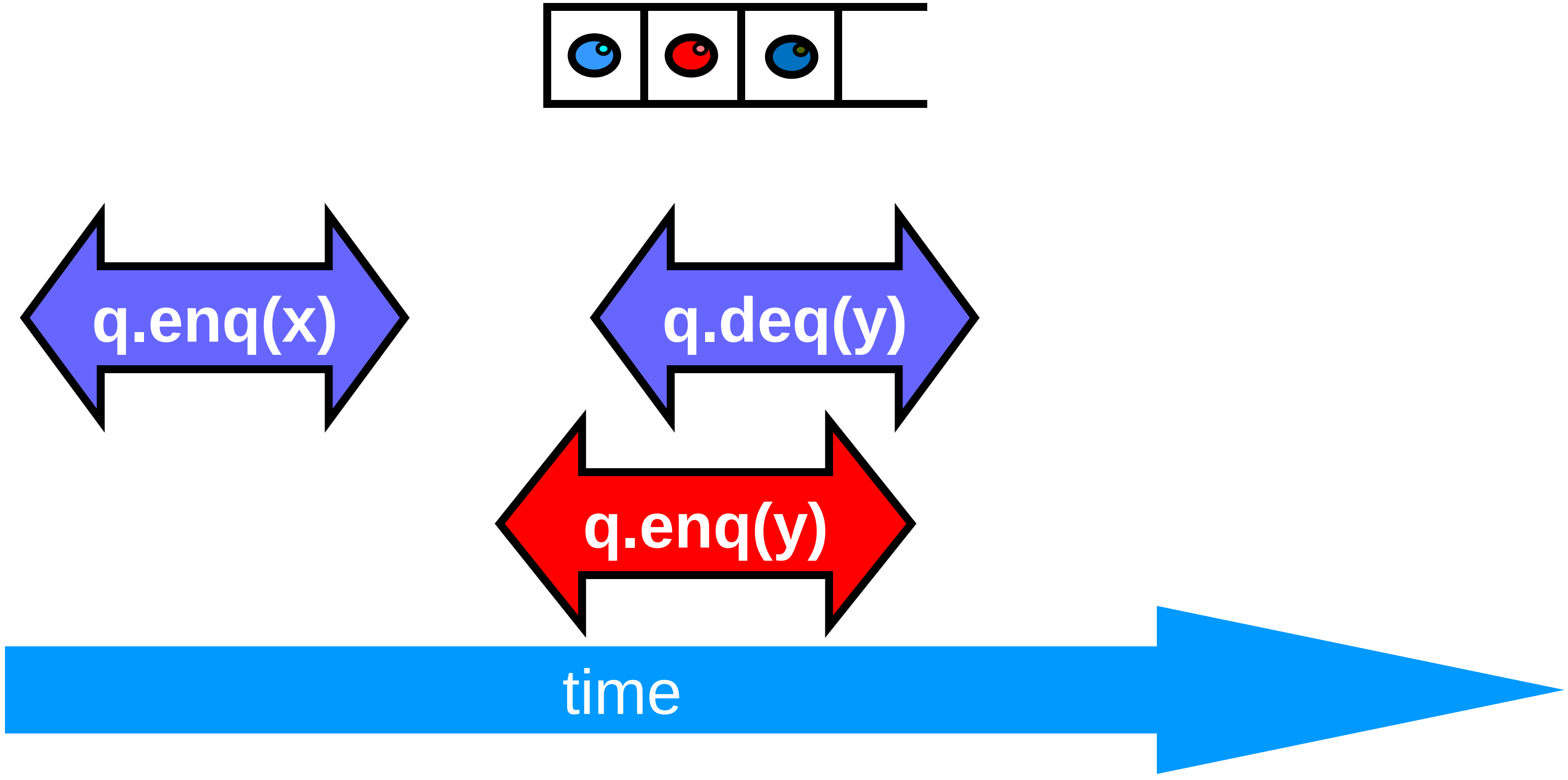
Example



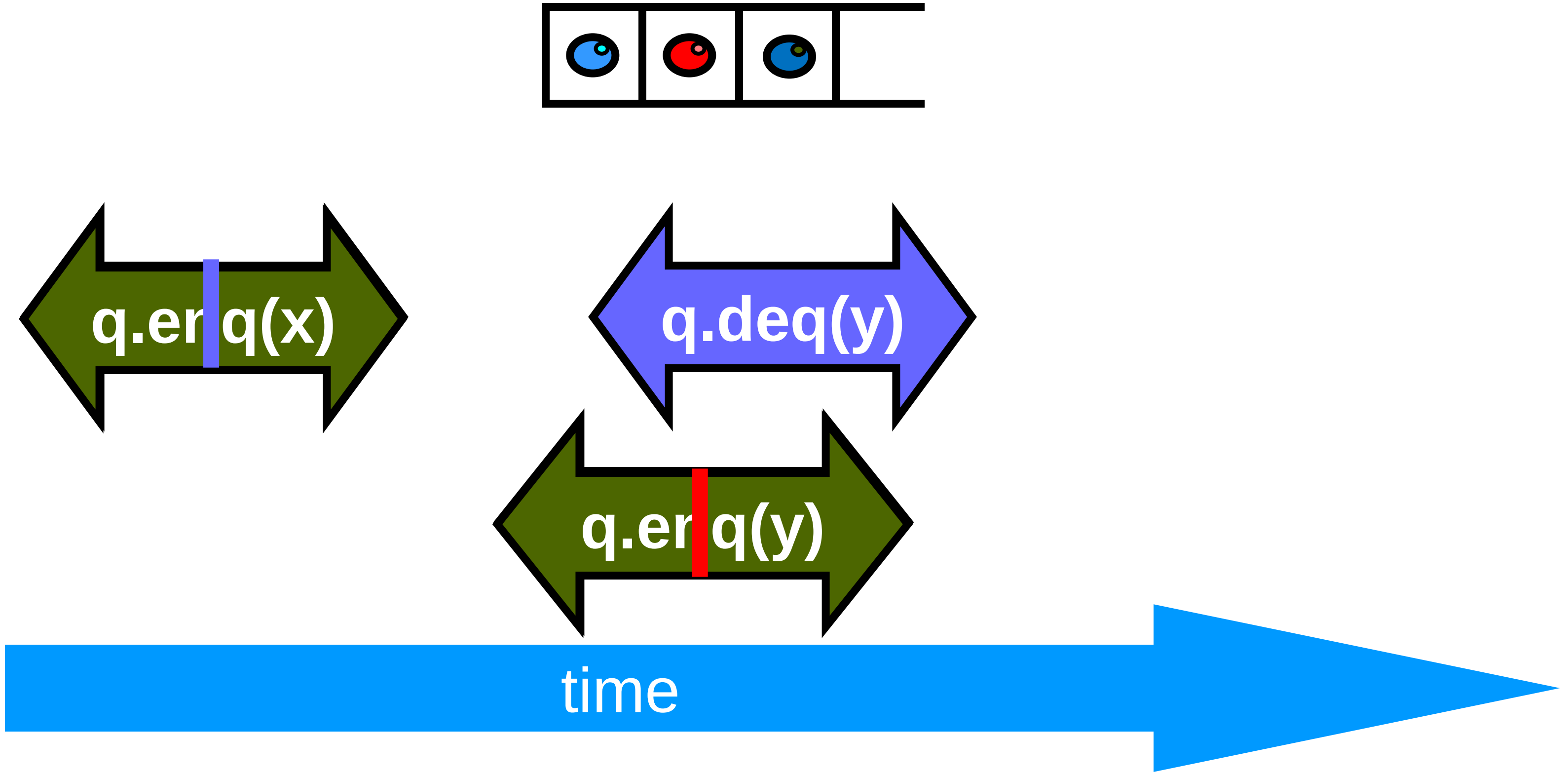
Example



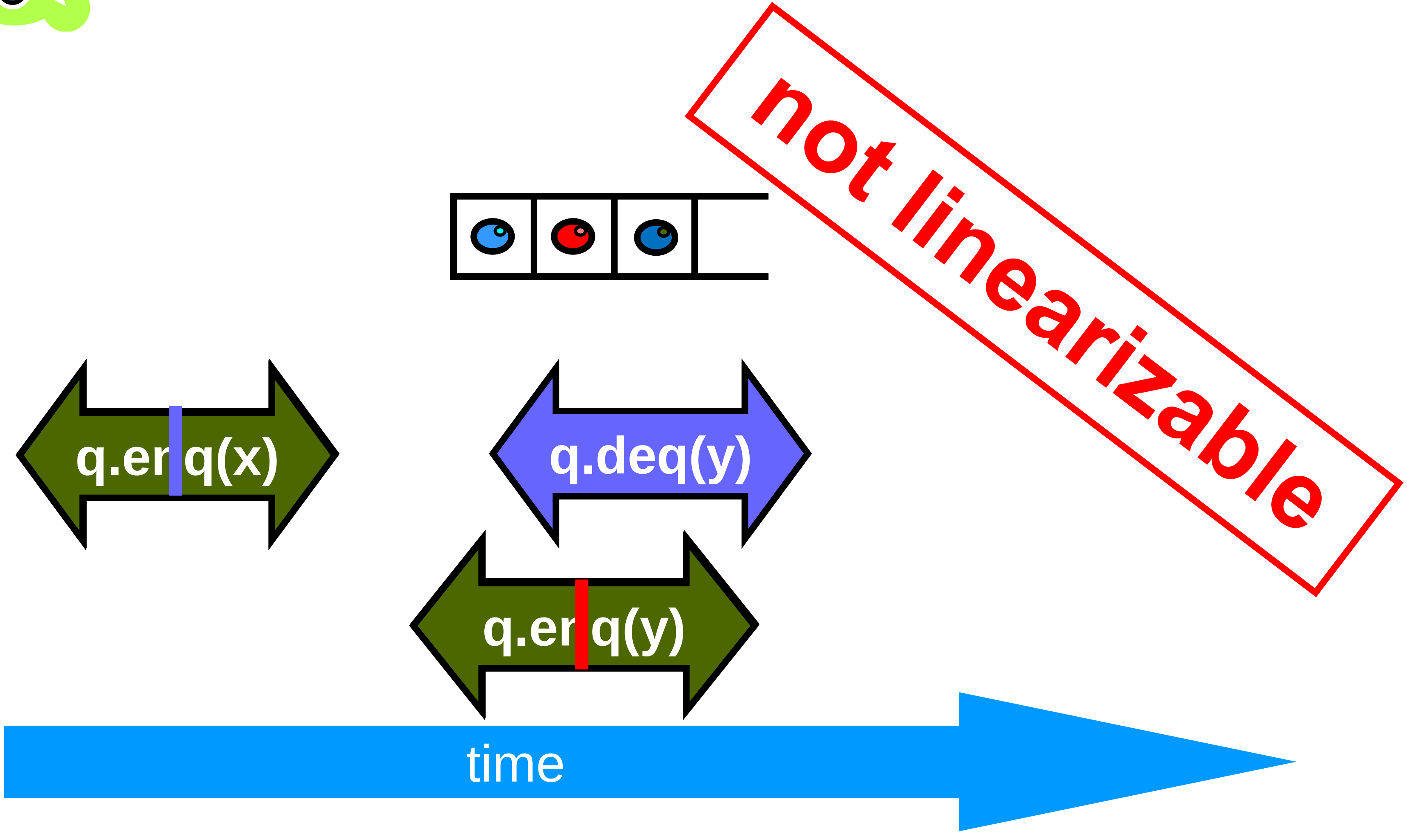
Example



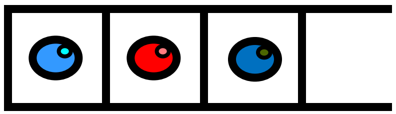
Example



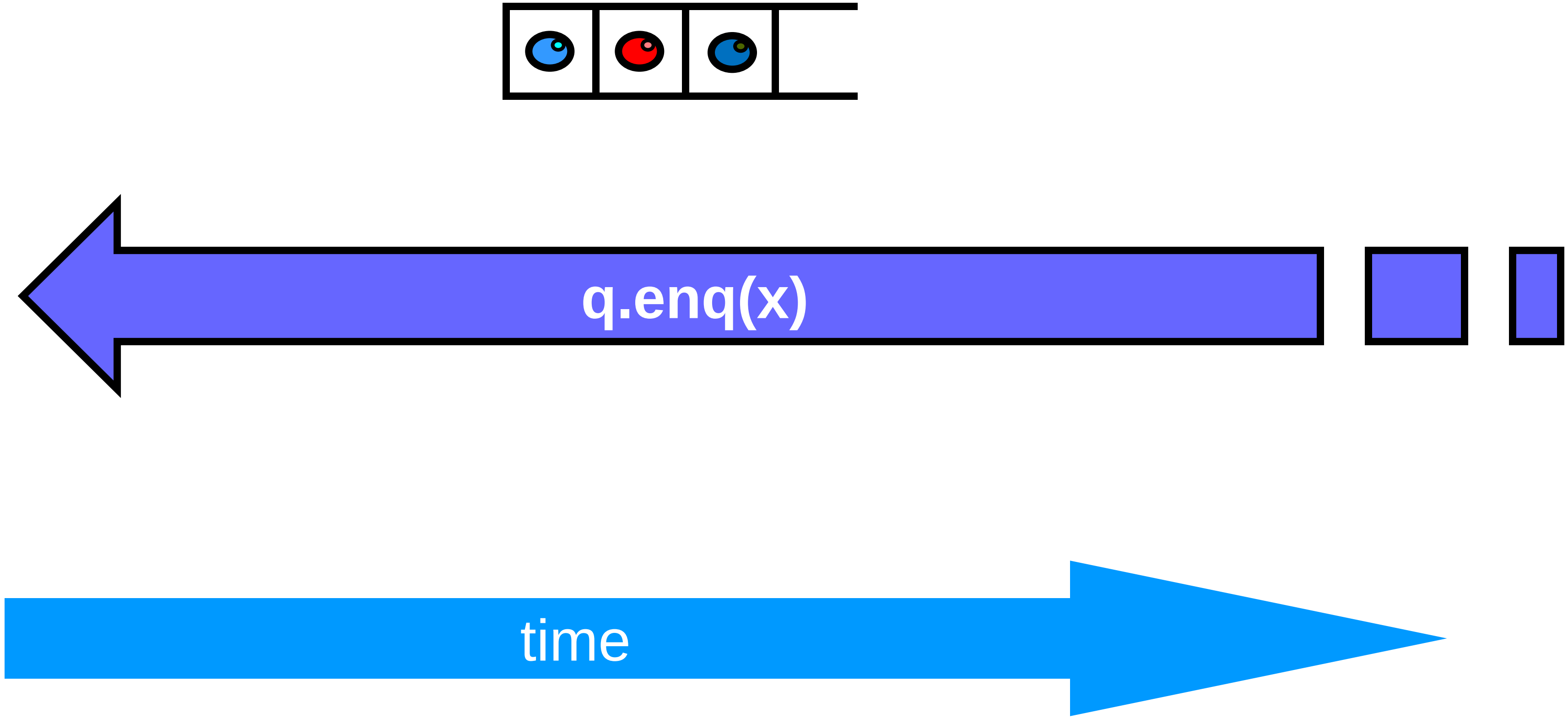
Example



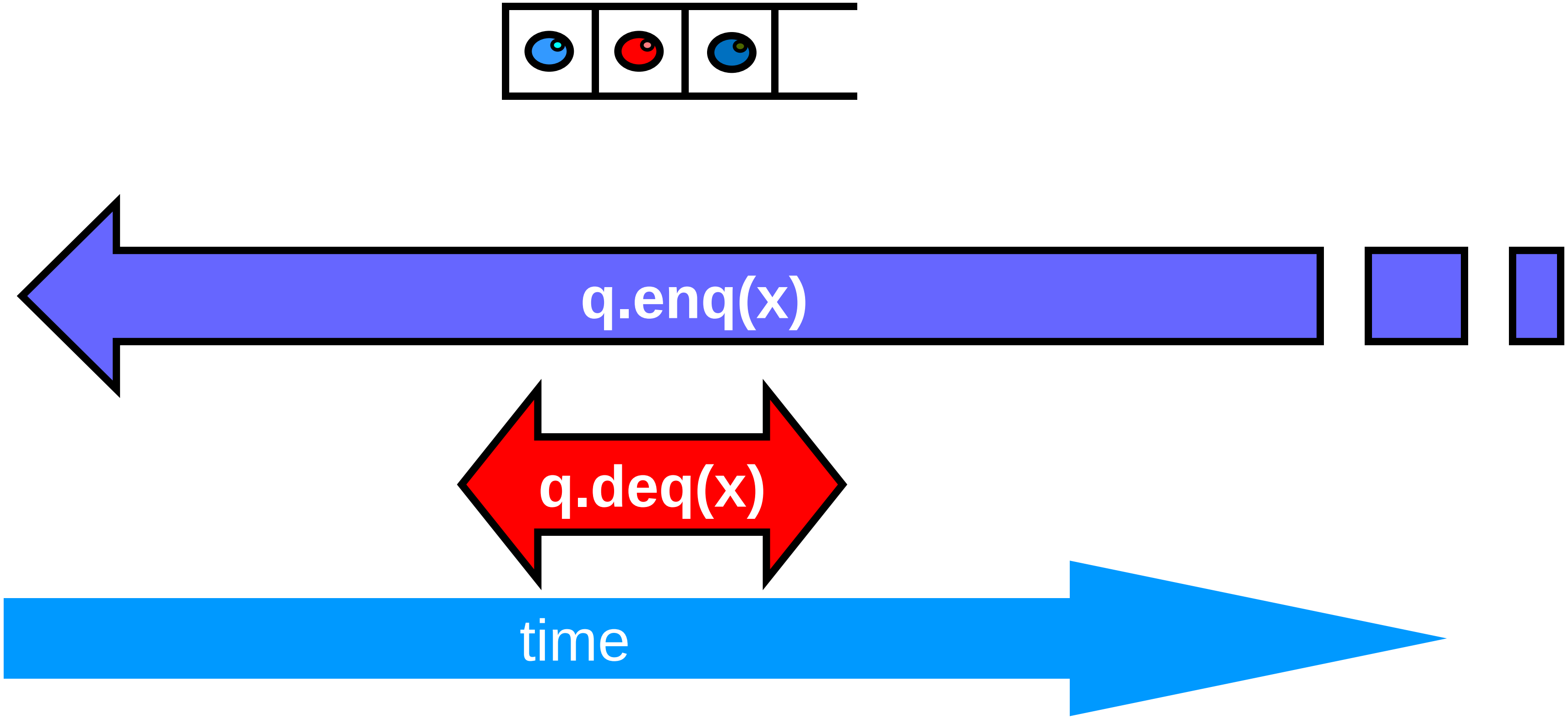
Example



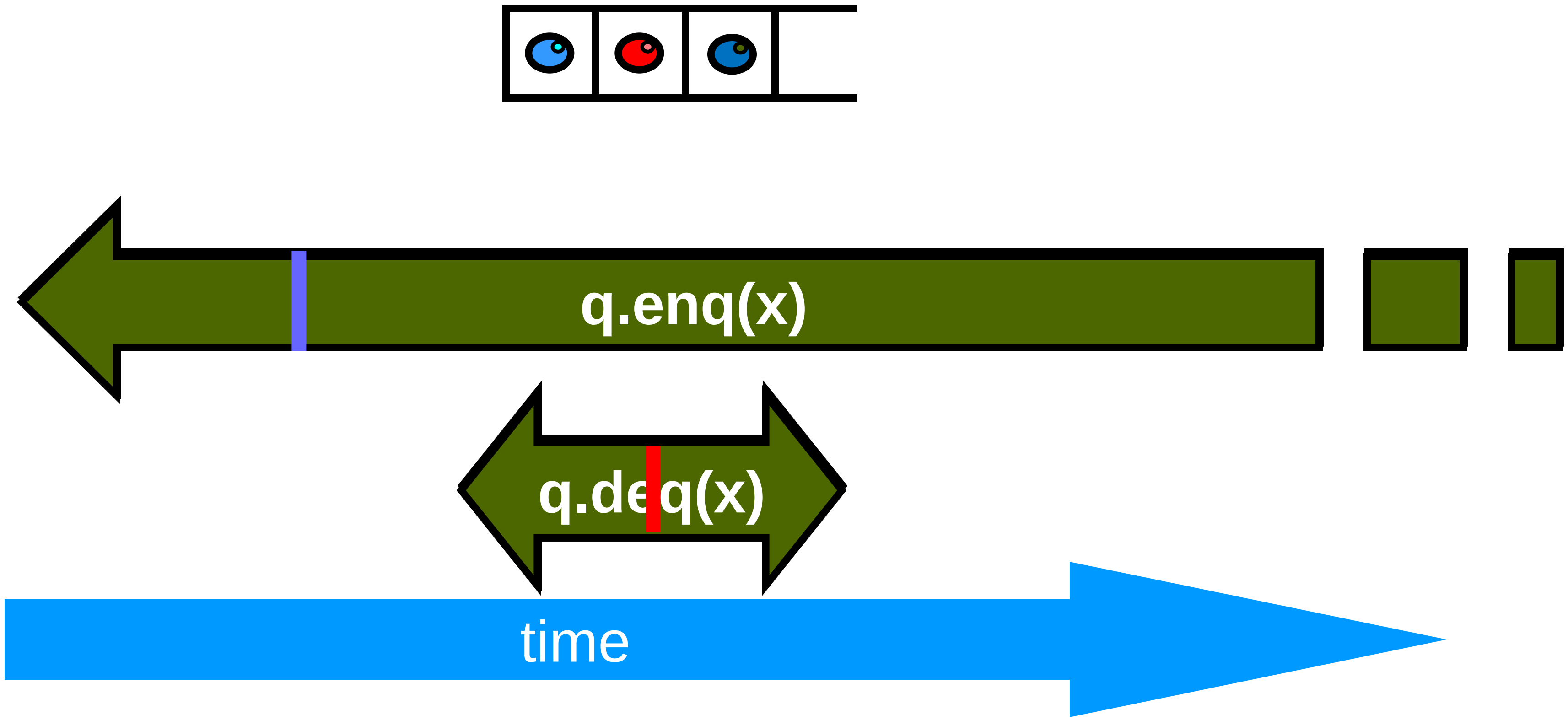
Example



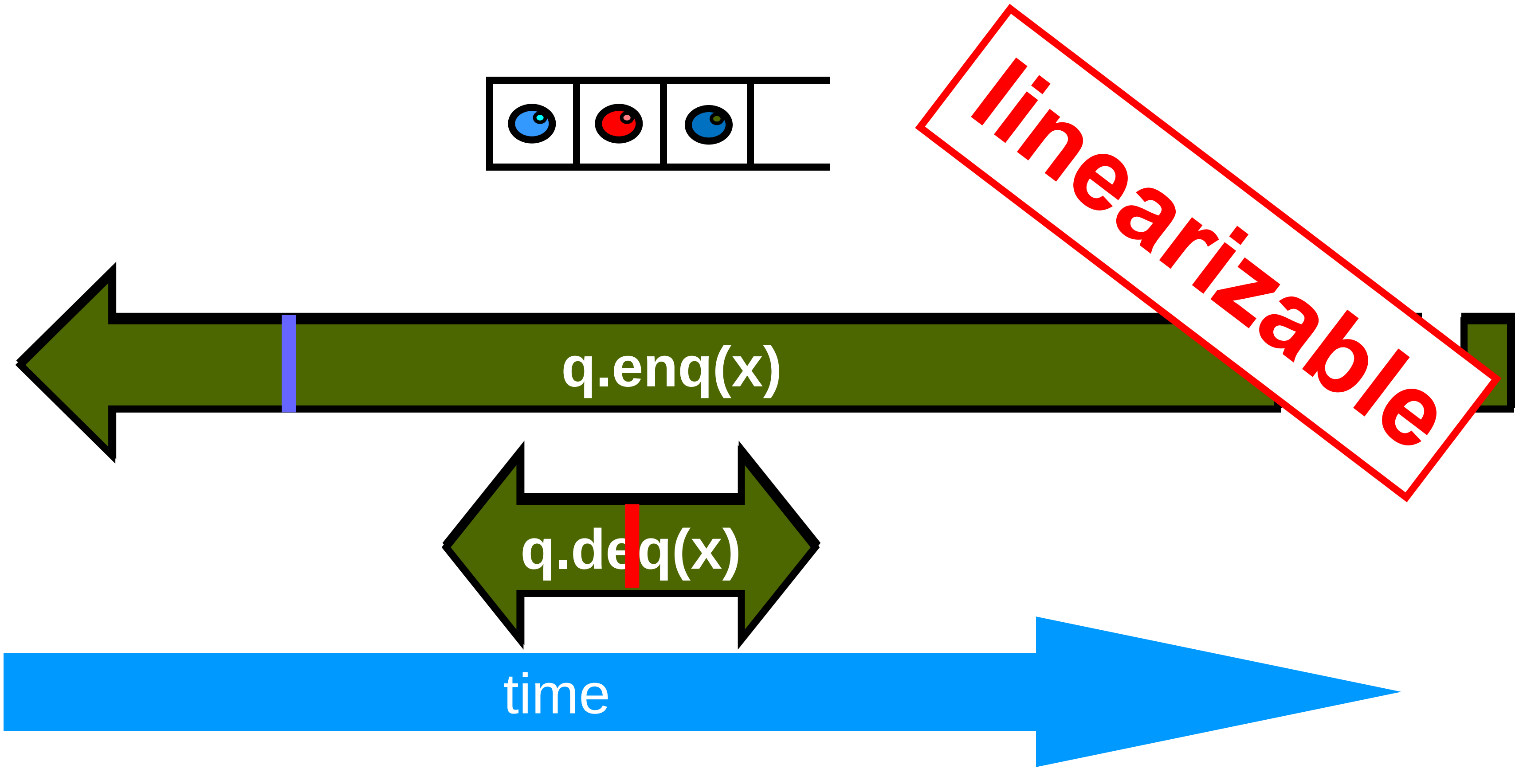
Example



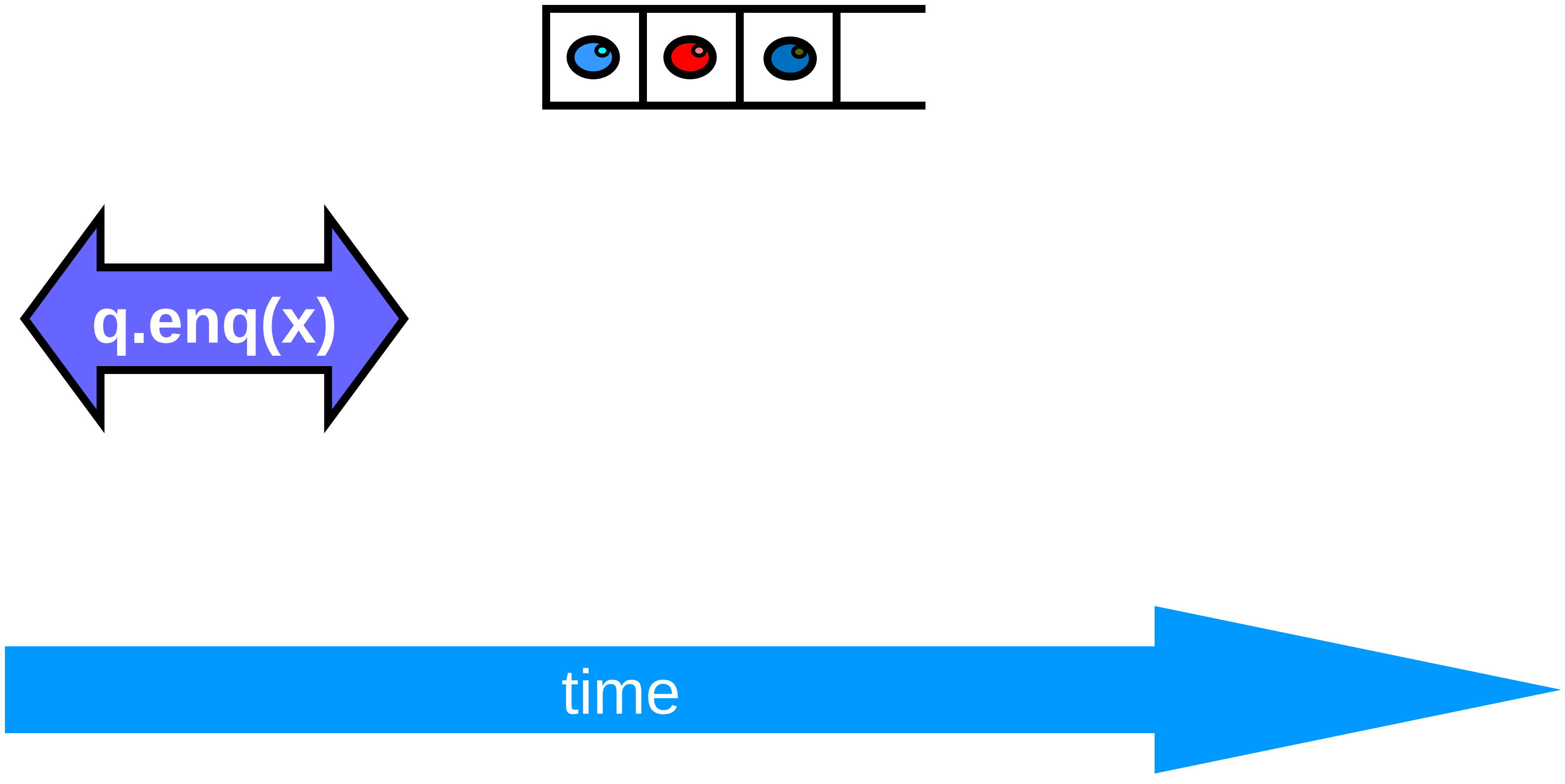
Example



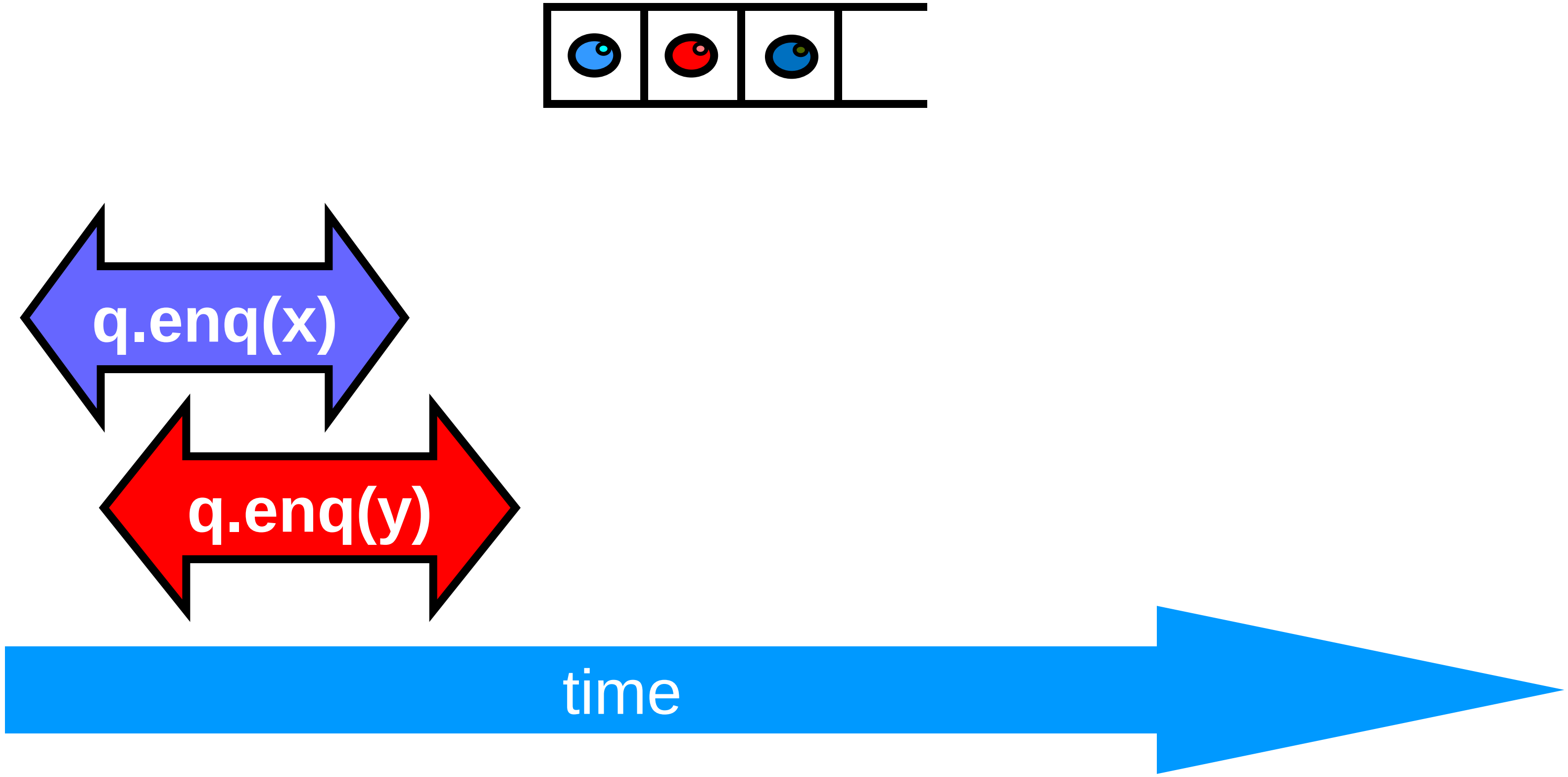
Example



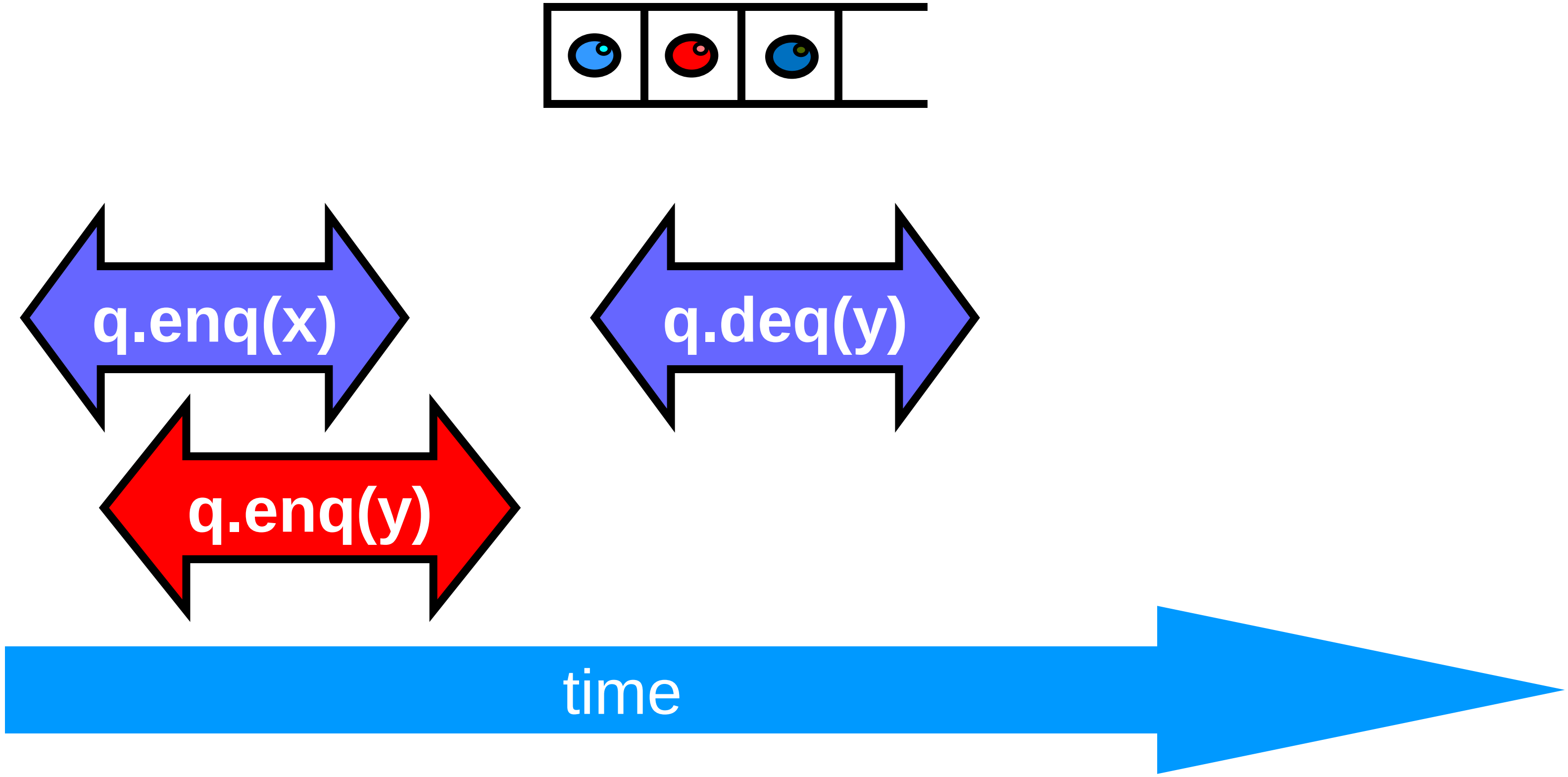
Example



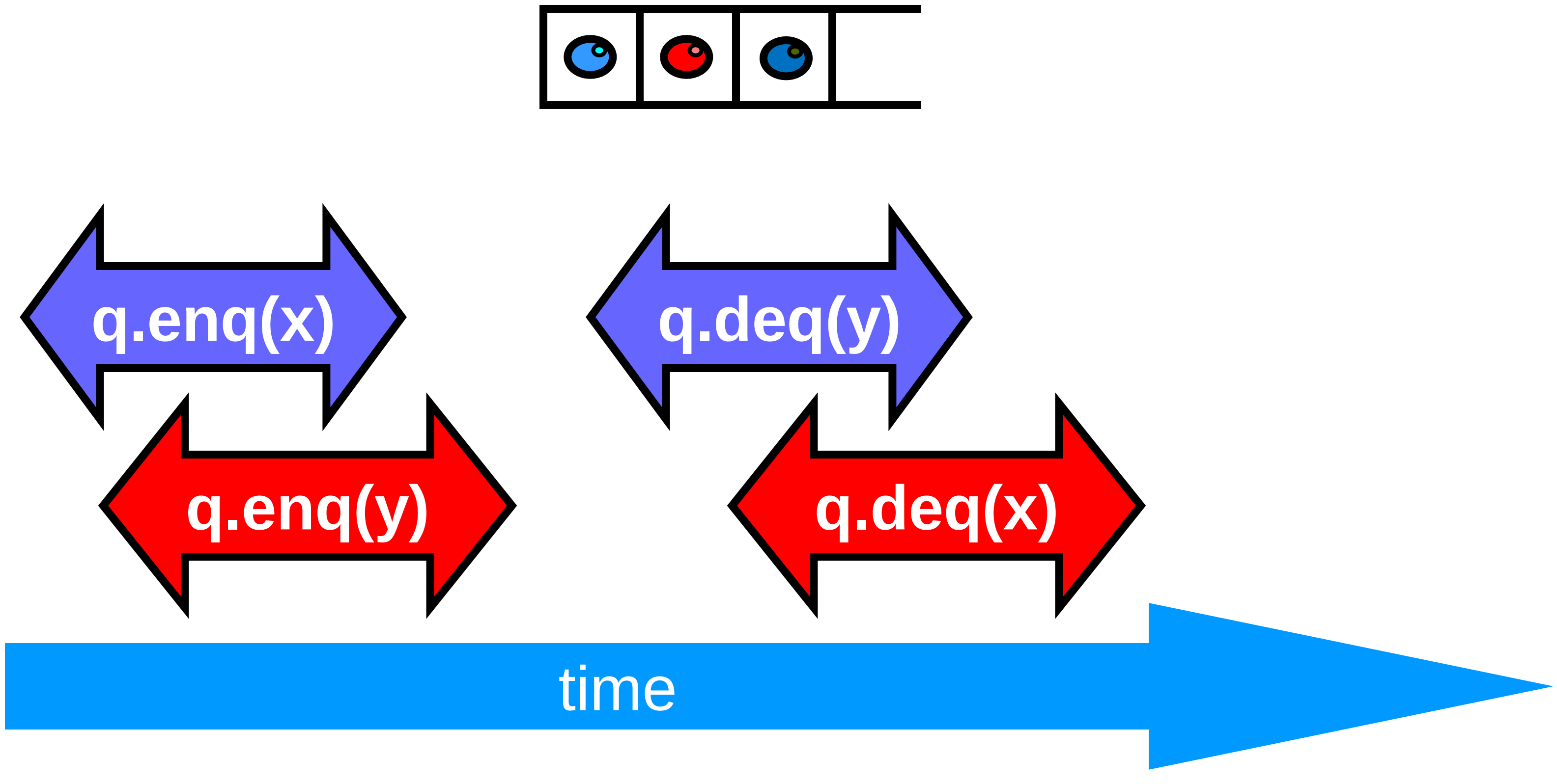
Example



Example

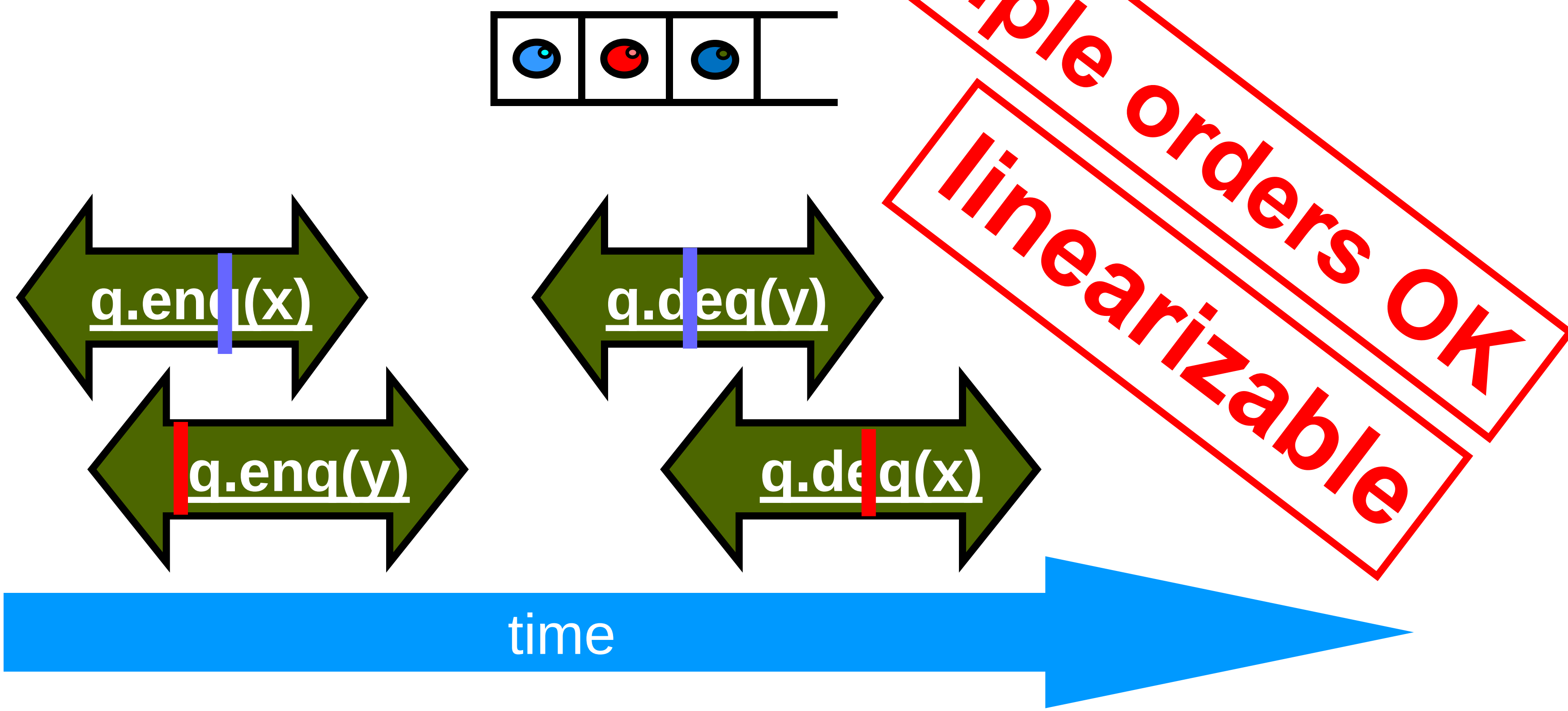


Example

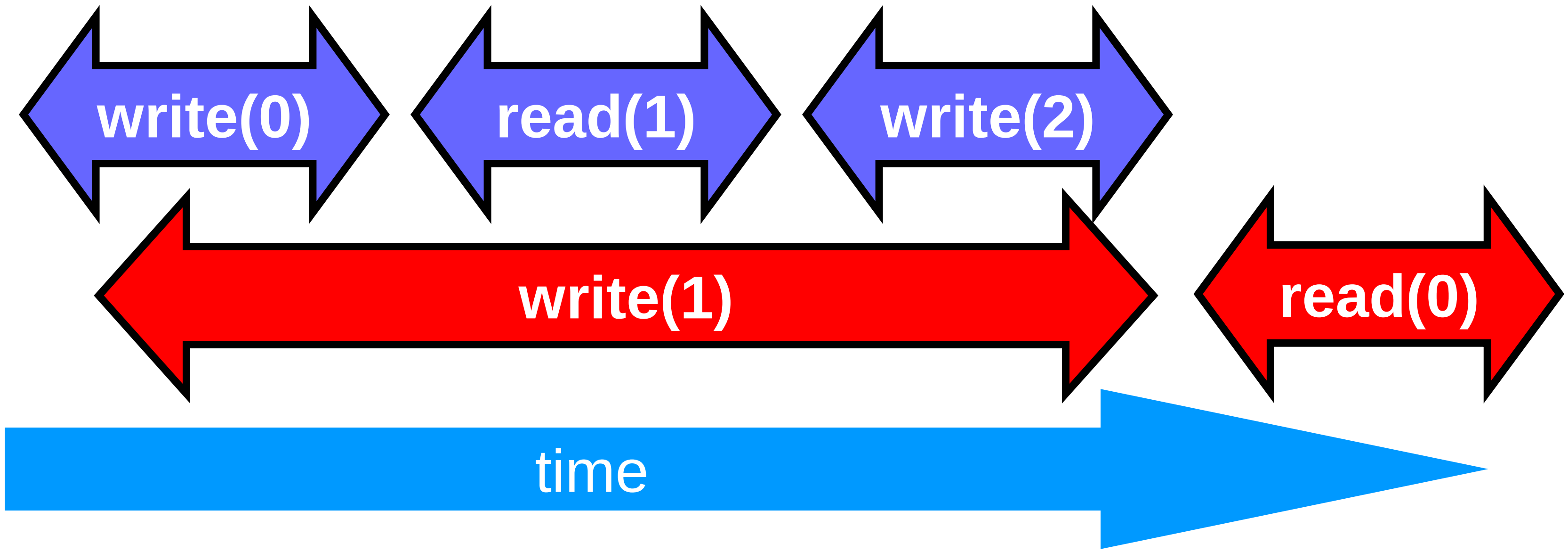


Example

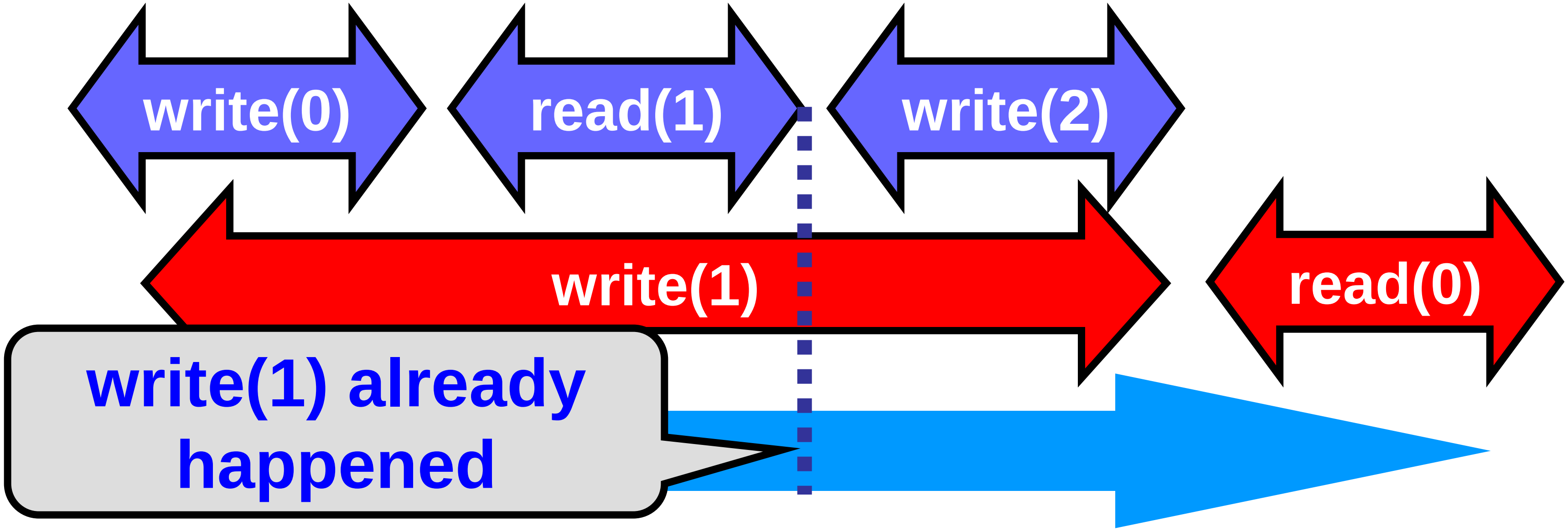
Comme ci
Comme ça



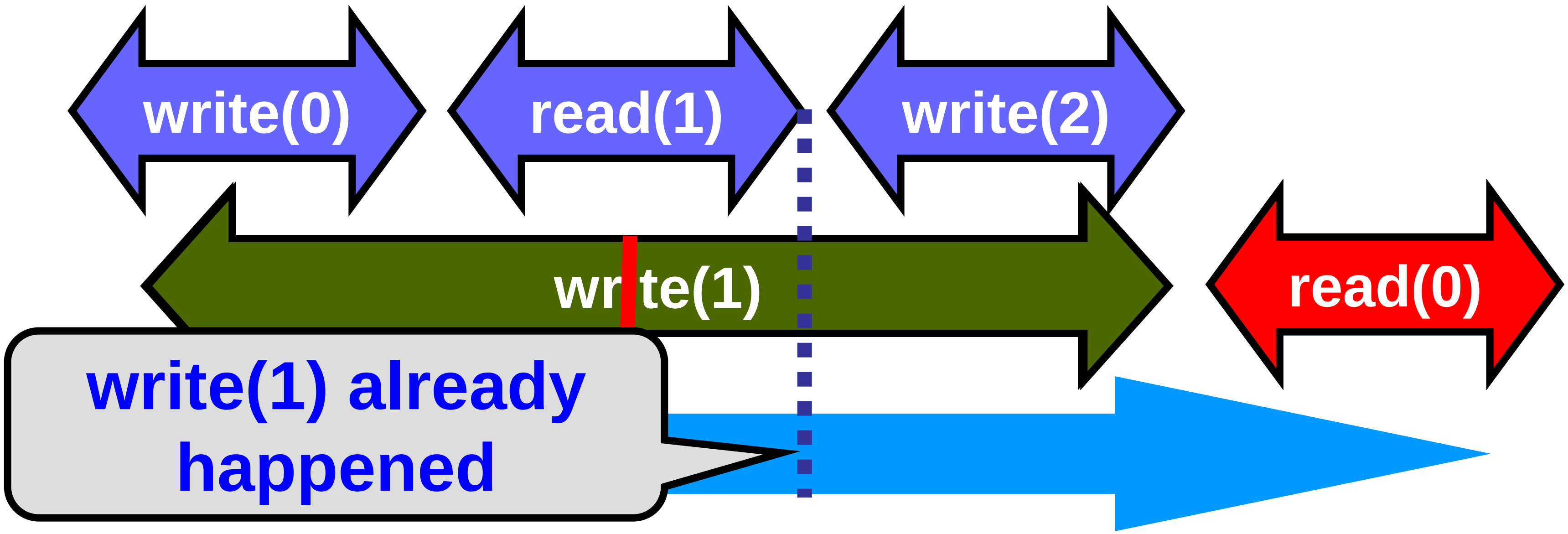
Read/Write Register Example



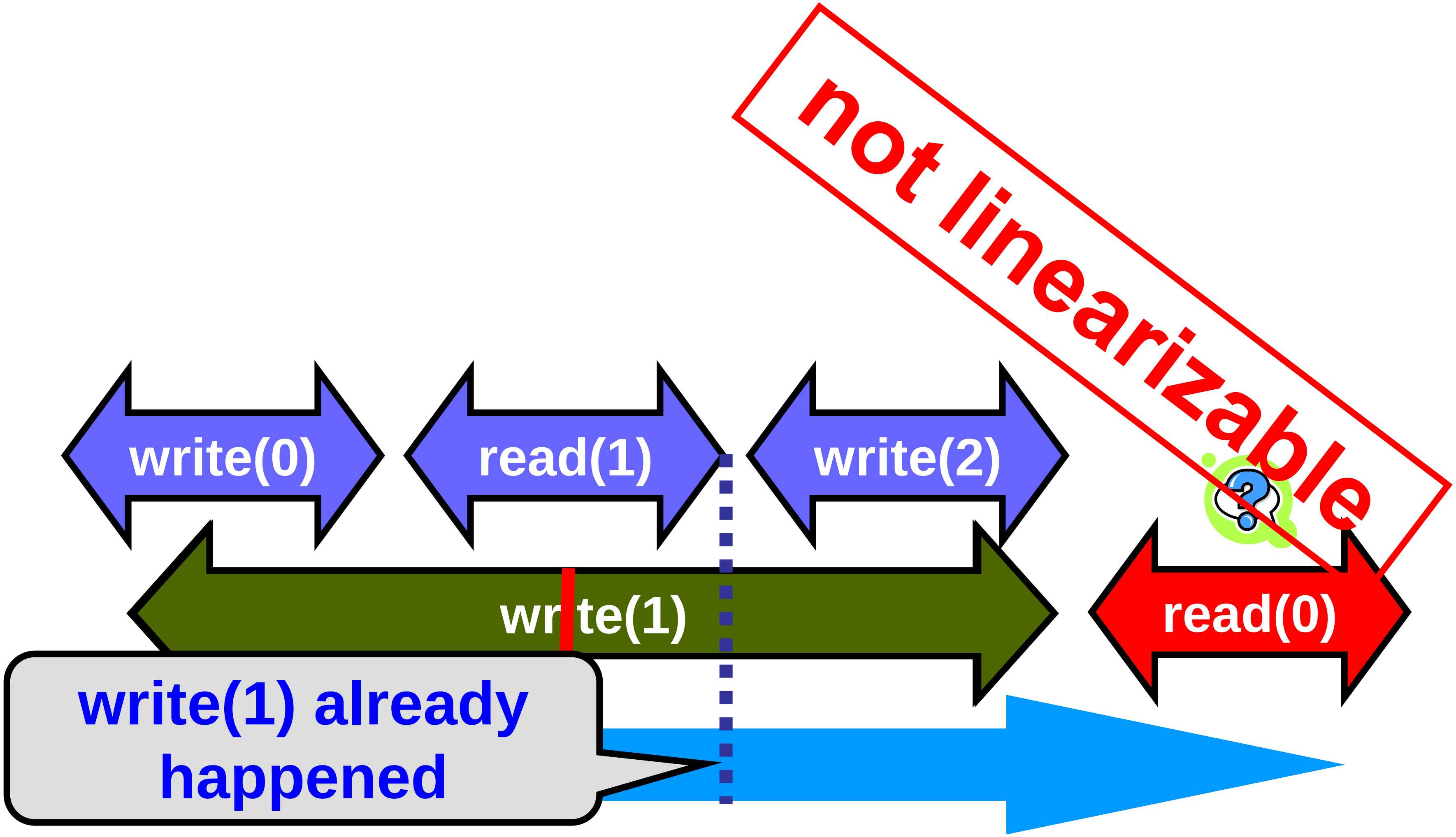
Read/Write Register Example



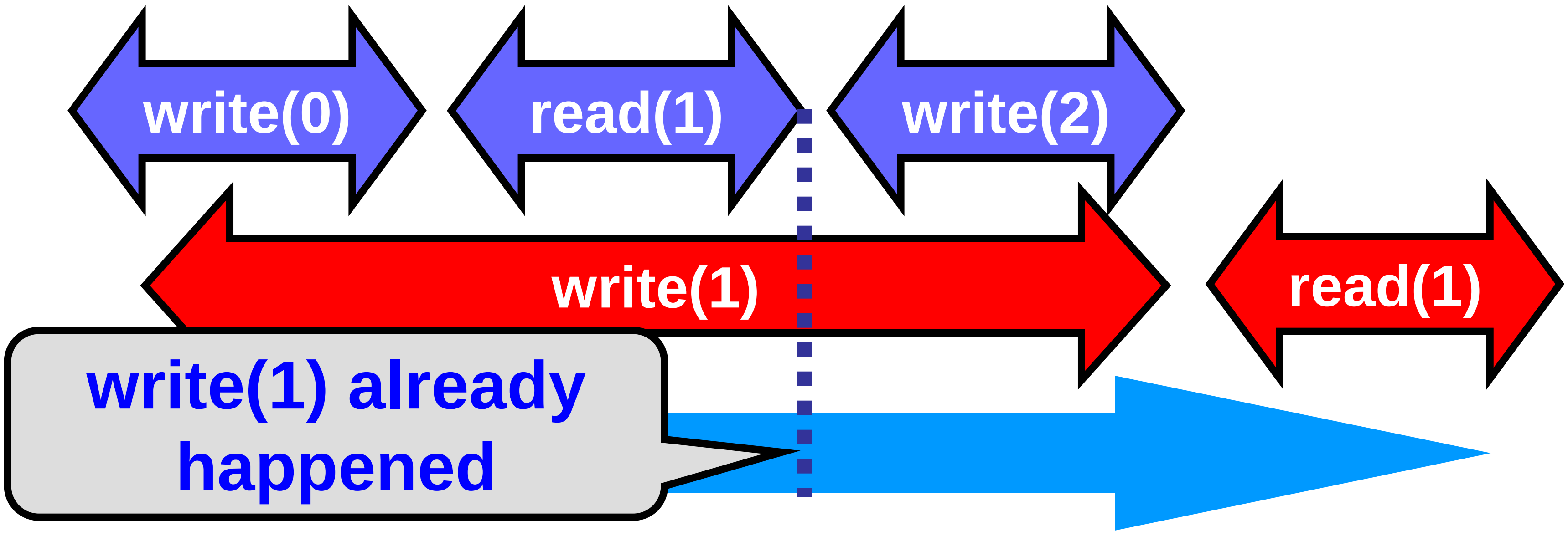
Read/Write Register Example



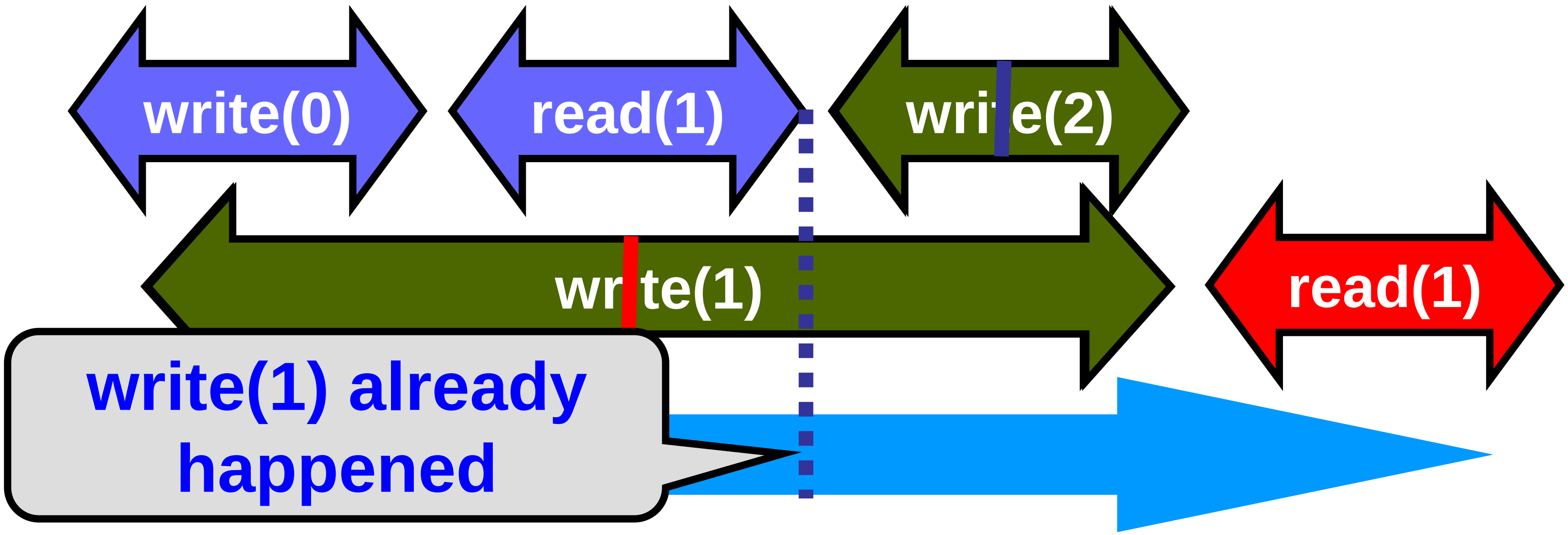
Read/Write Register Example



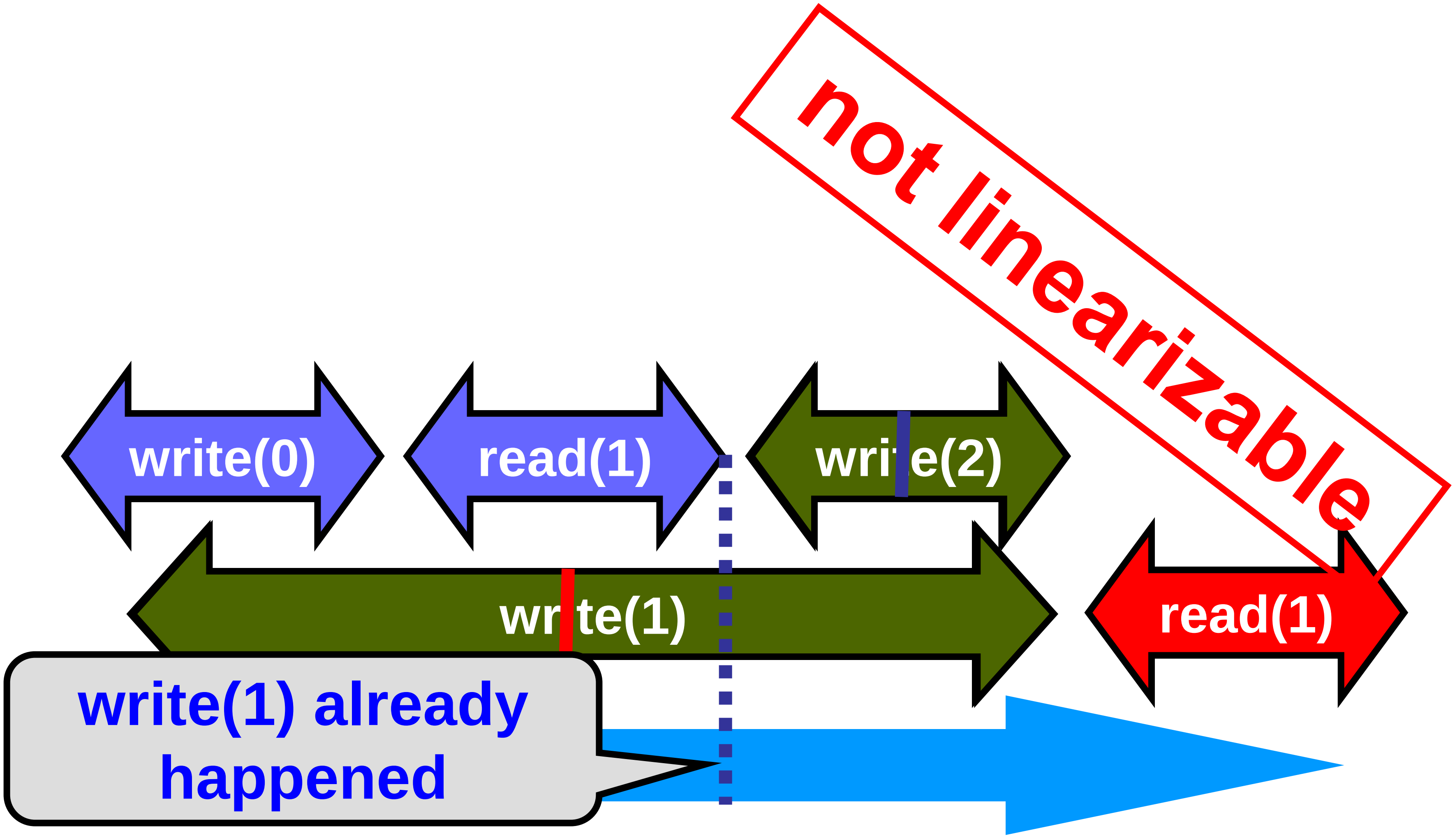
Read/Write Register Example



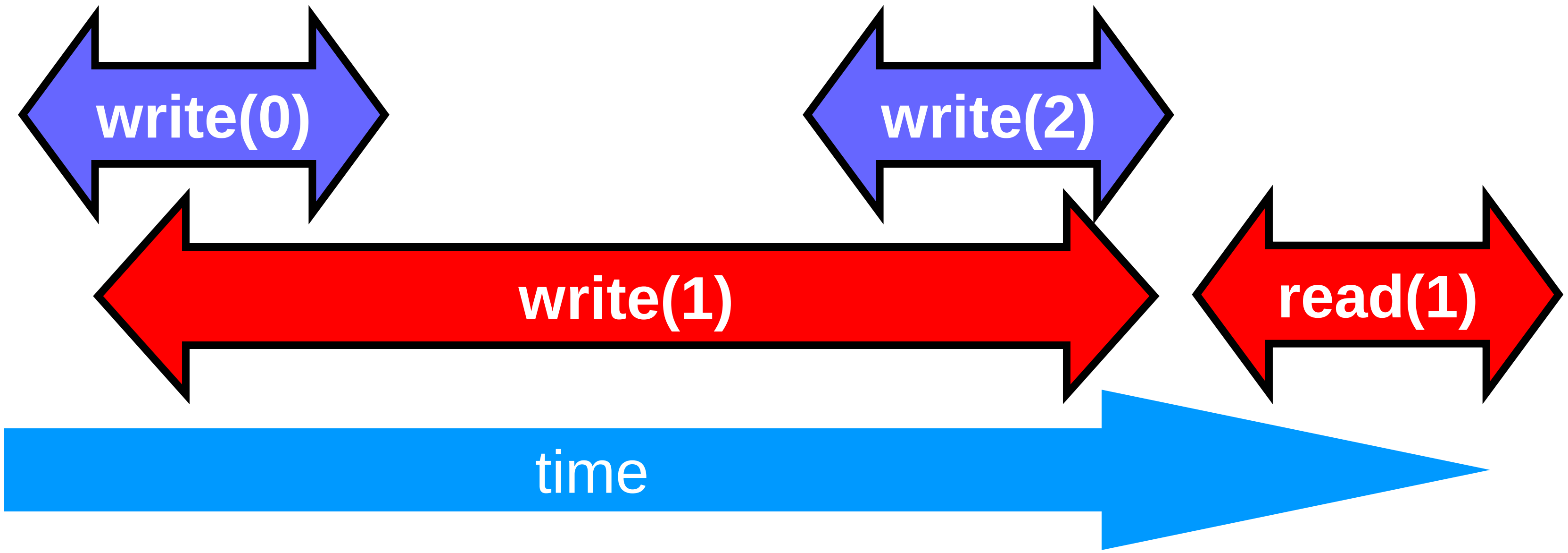
Read/Write Register Example



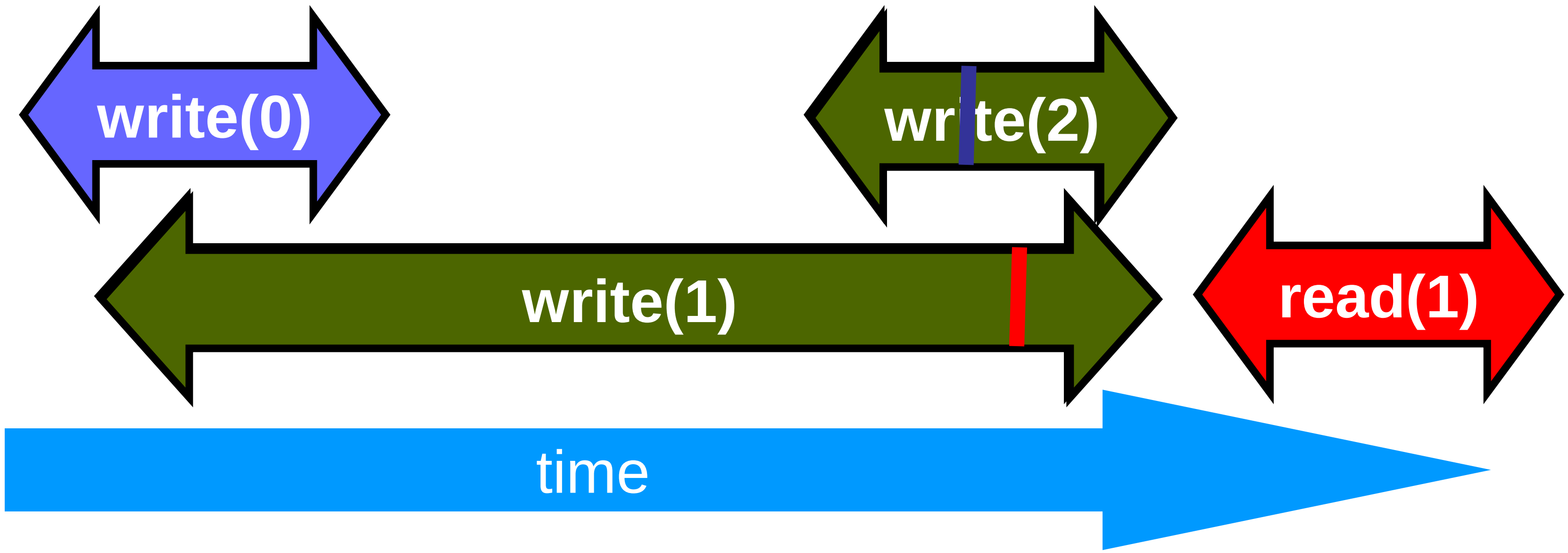
Read/Write Register Example



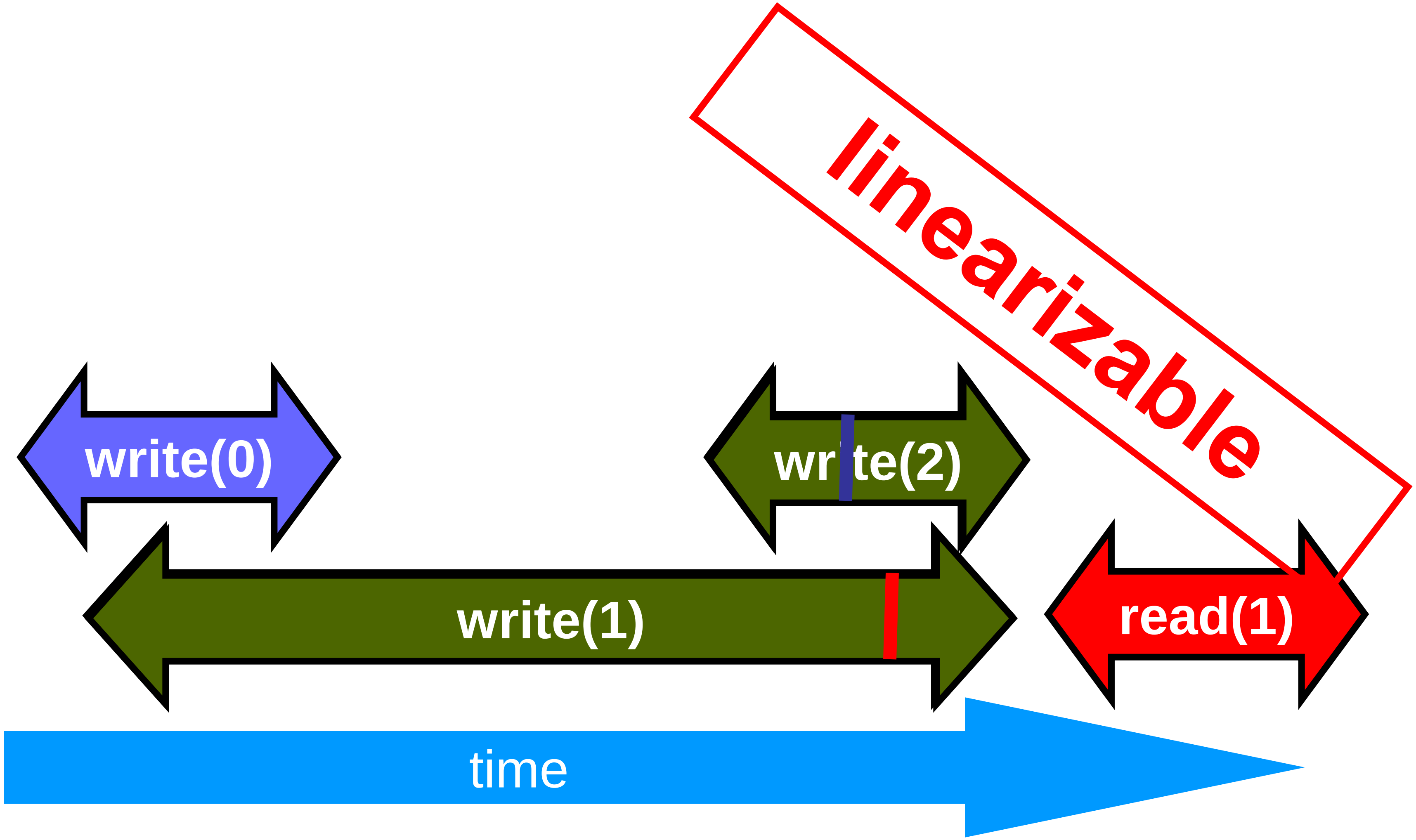
Read/Write Register Example



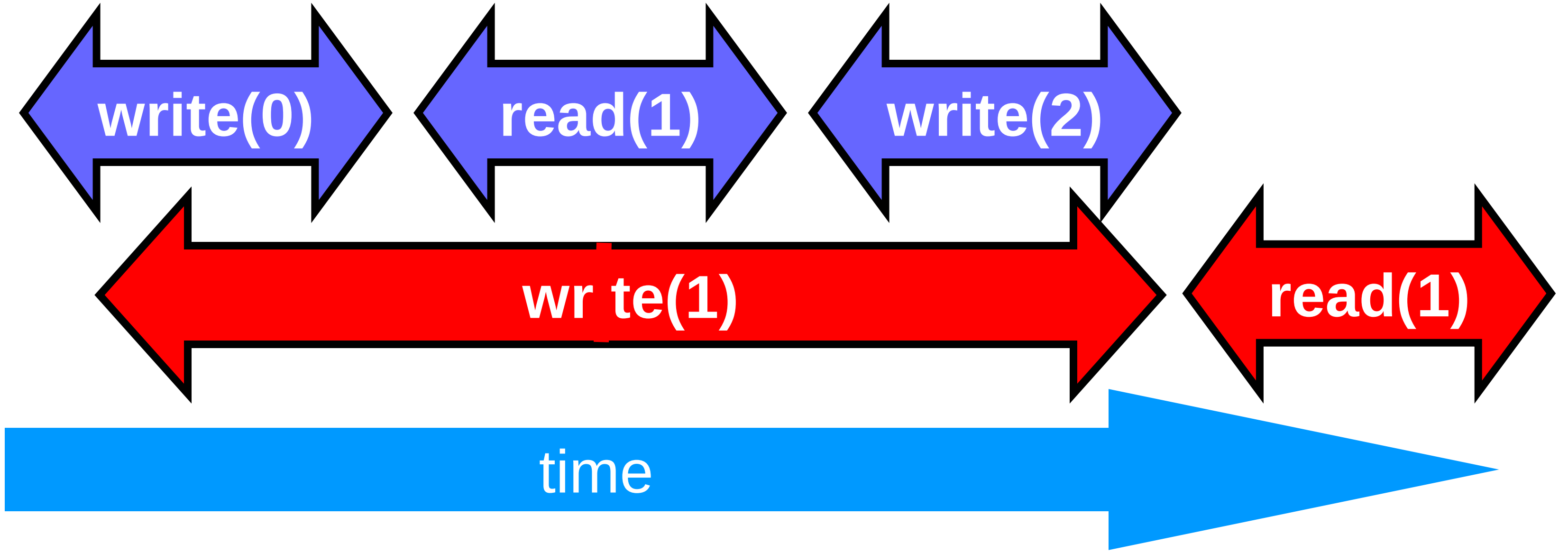
Read/Write Register Example



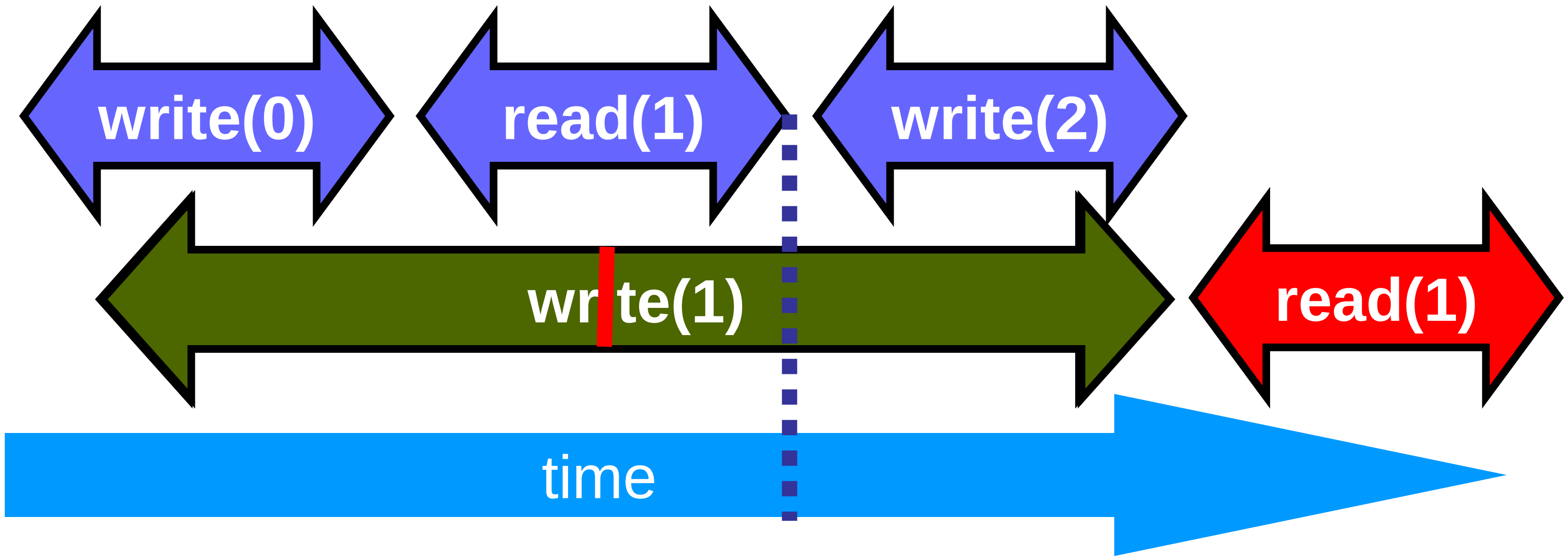
Read/Write Register Example



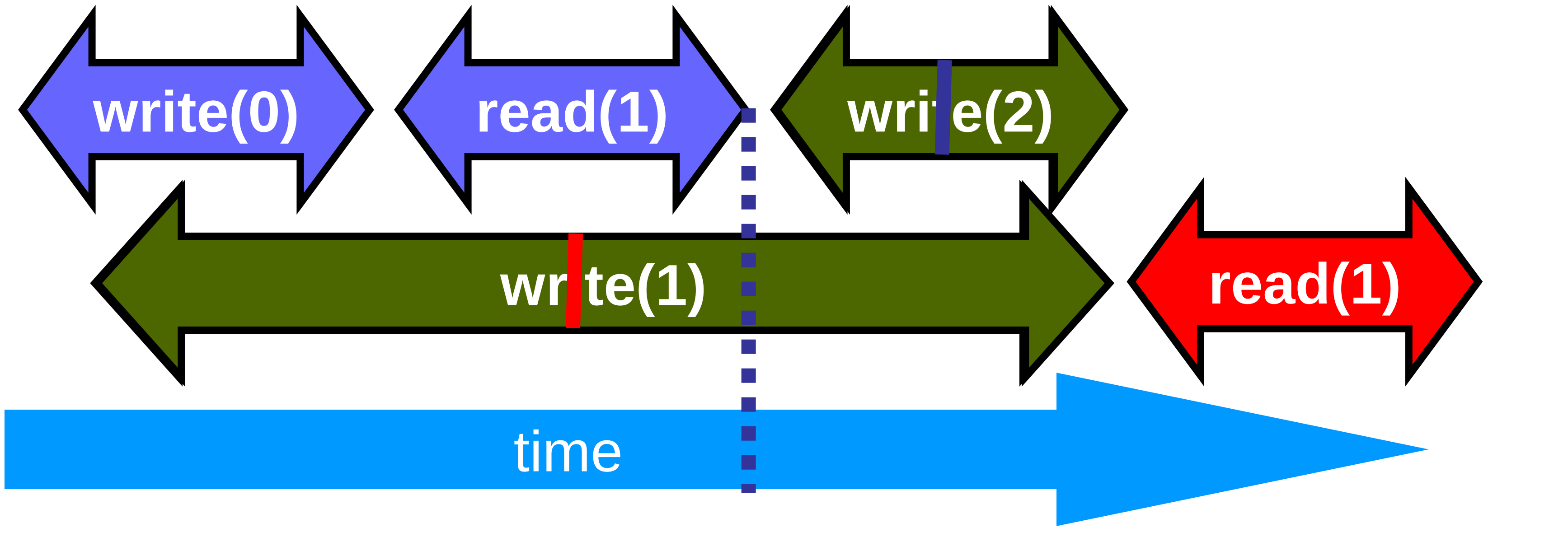
Read/Write Register Example



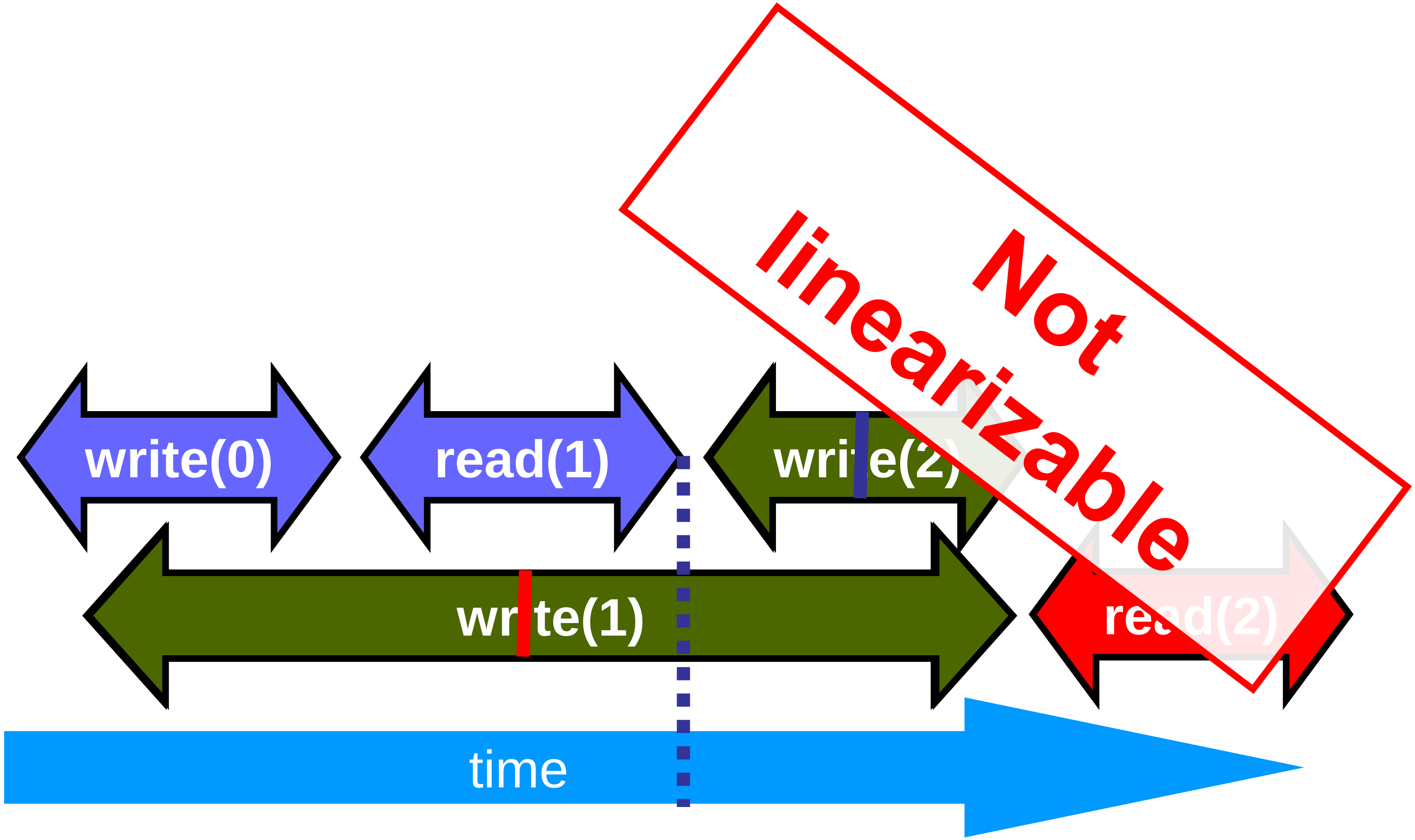
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Talking About Executions

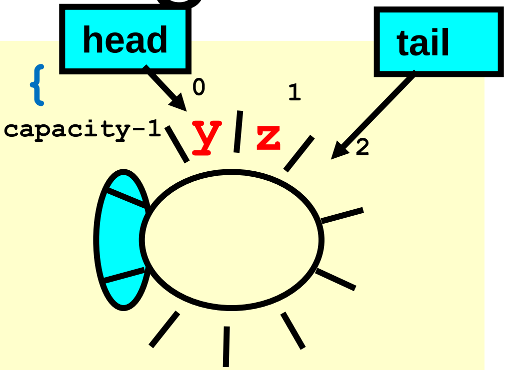
- Why?
  - Can't we specify the linearization point of each operation without describing an execution?
- Not Always
  - In some cases, linearization point depends on the execution

Linearizable Objects are Composable

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

Reasoning About Linearizability: 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();
  }
}
```

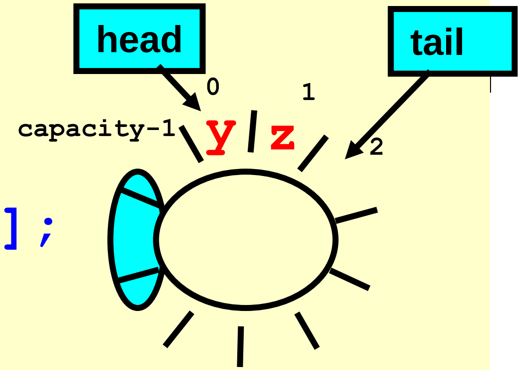


Reasoning About Linearizability: Locking

```
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 points are when locks are released

More Reasoning: Wait-free




```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

More Reasoning: Wait-free

```
public clas   itFreeQueue {

  int h       ll =
  it          new Ob

          id enq(Item x) {
        ail-head == capacity) throw
        new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
        new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Remember that there is only one enqueuer and only one dequeuer

Linearization order is order head and tail fields modified

Strategy

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

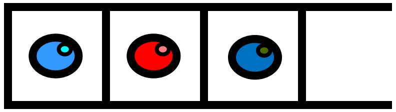
Linearizability: Summary

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

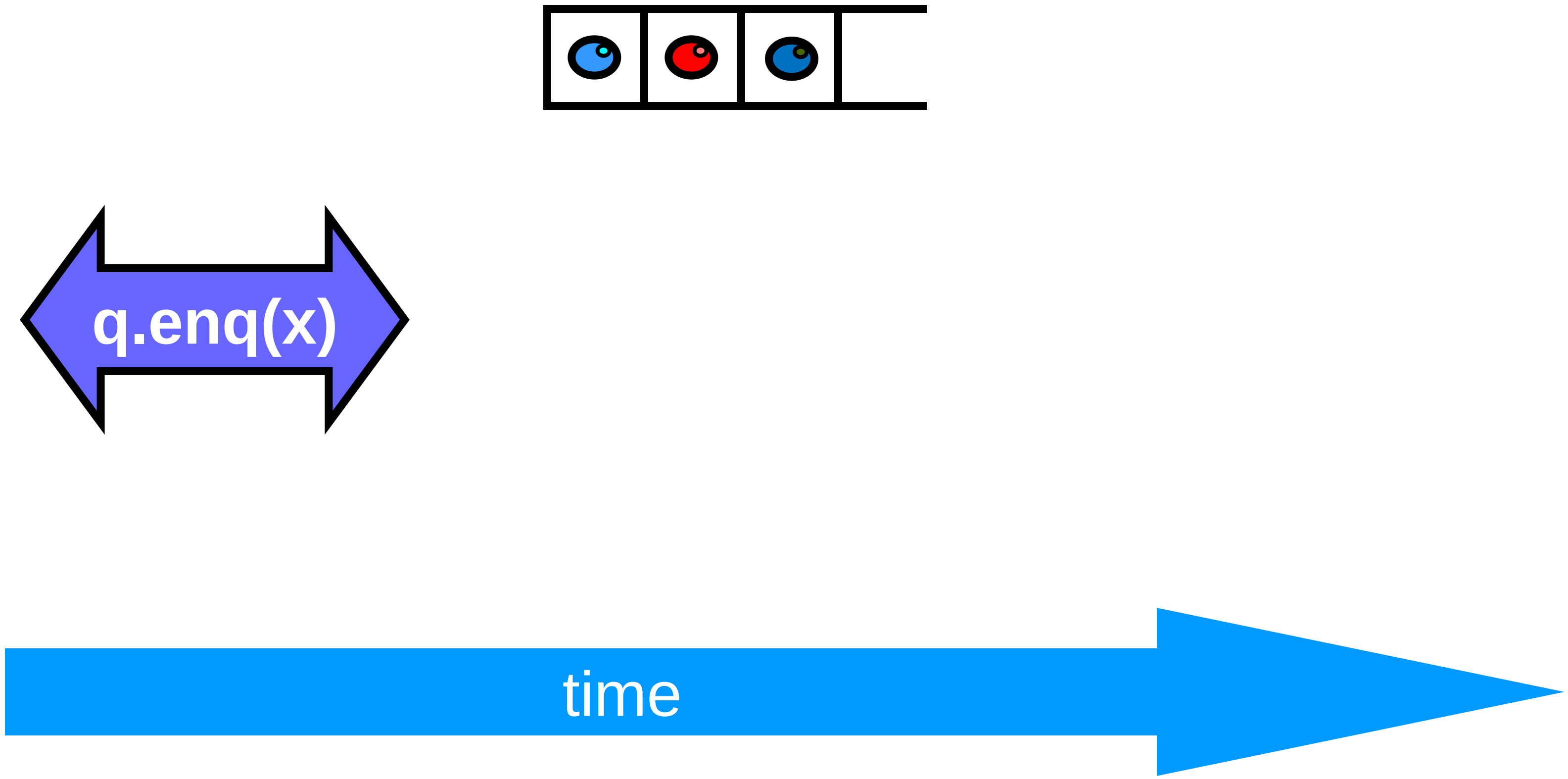
Sequential Consistency

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

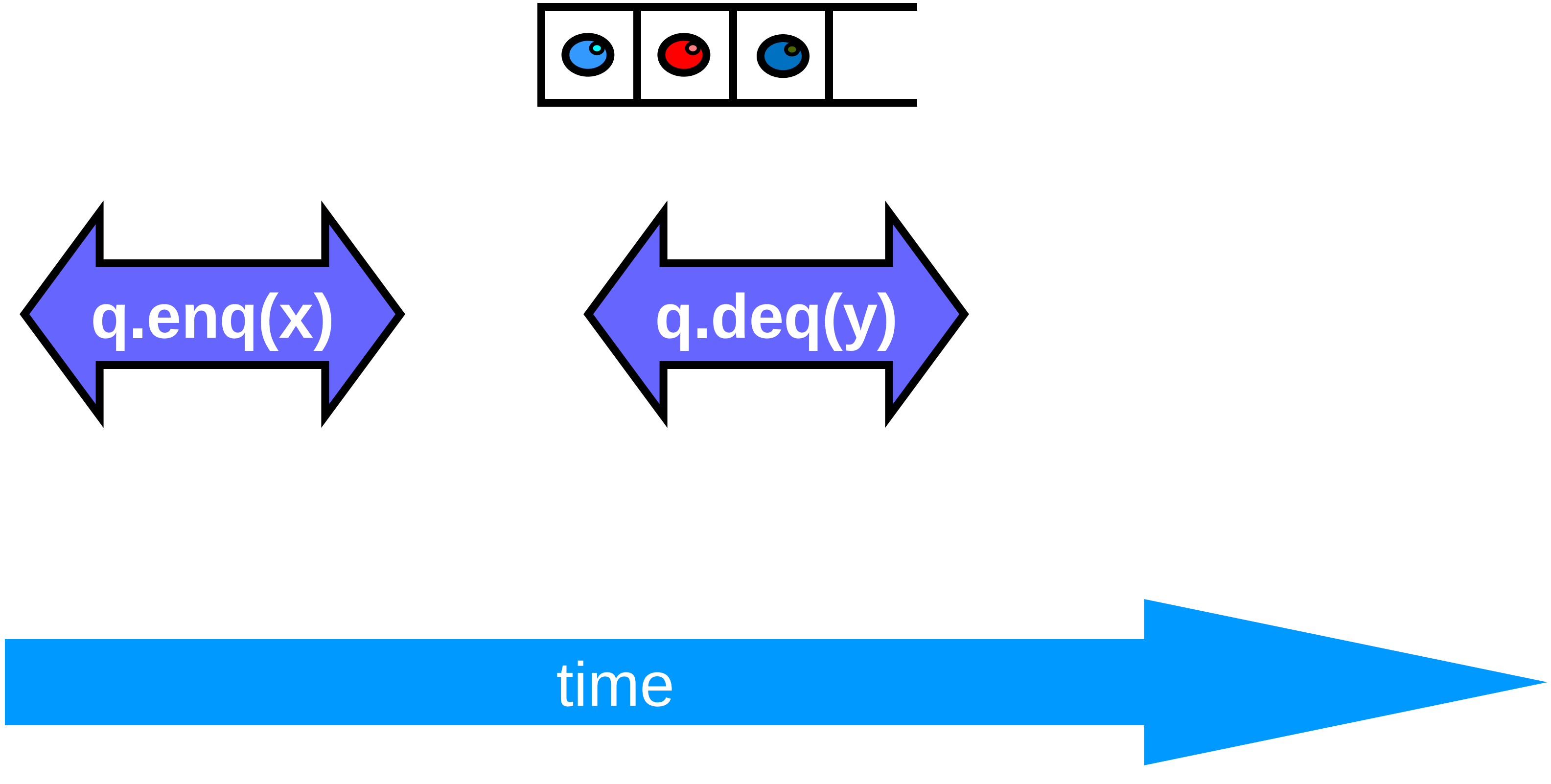
Example



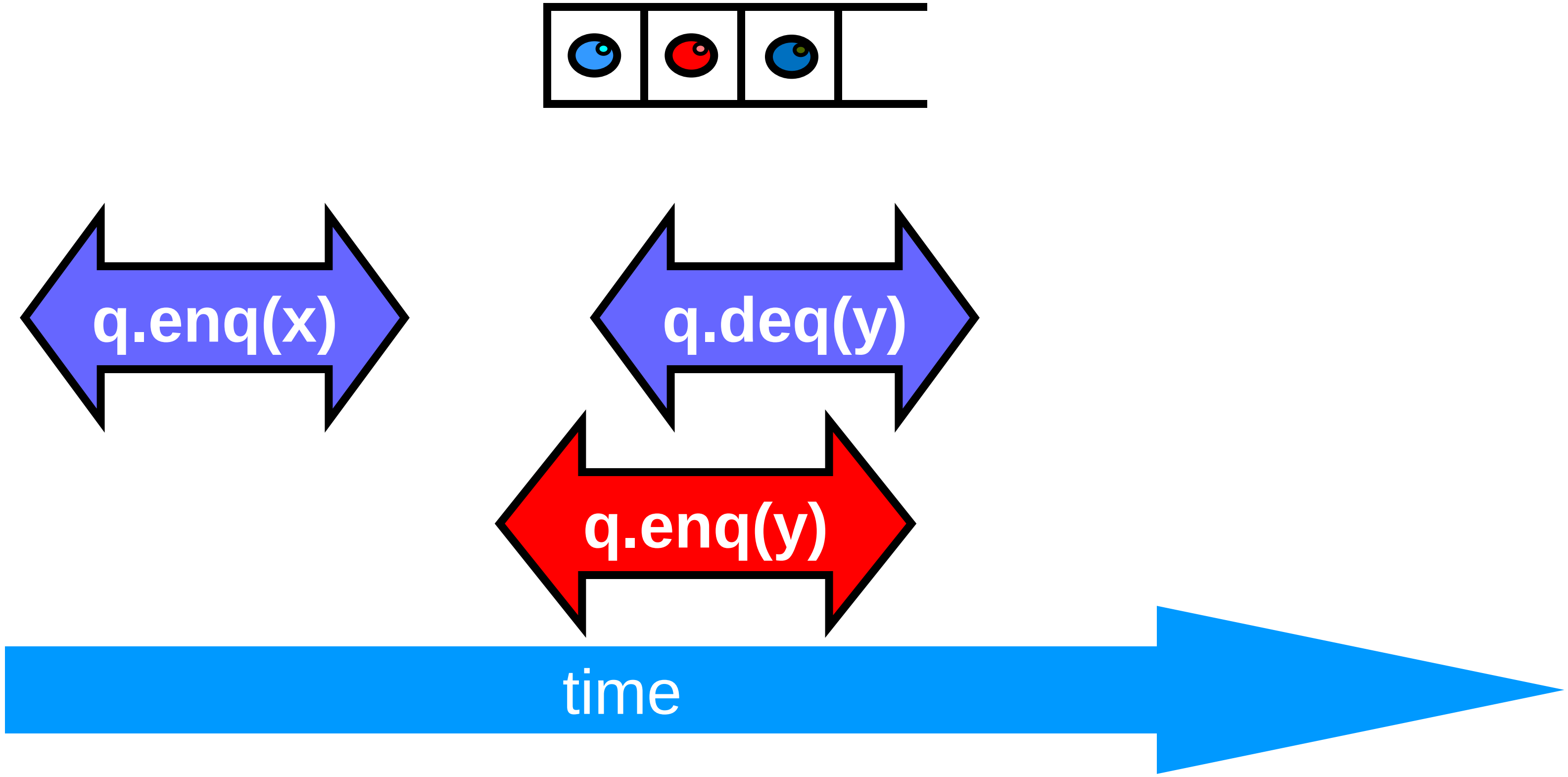
Example



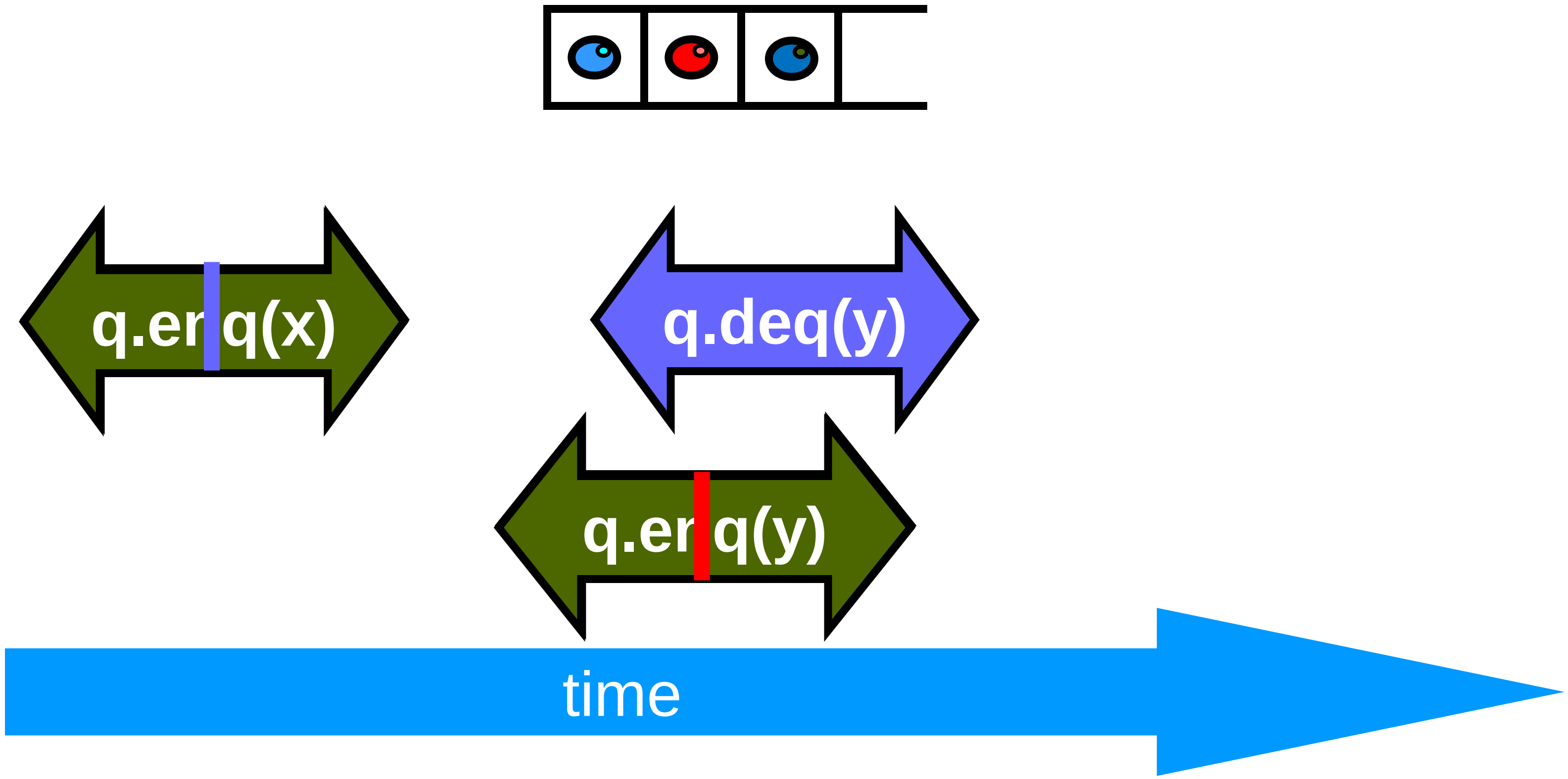
Example



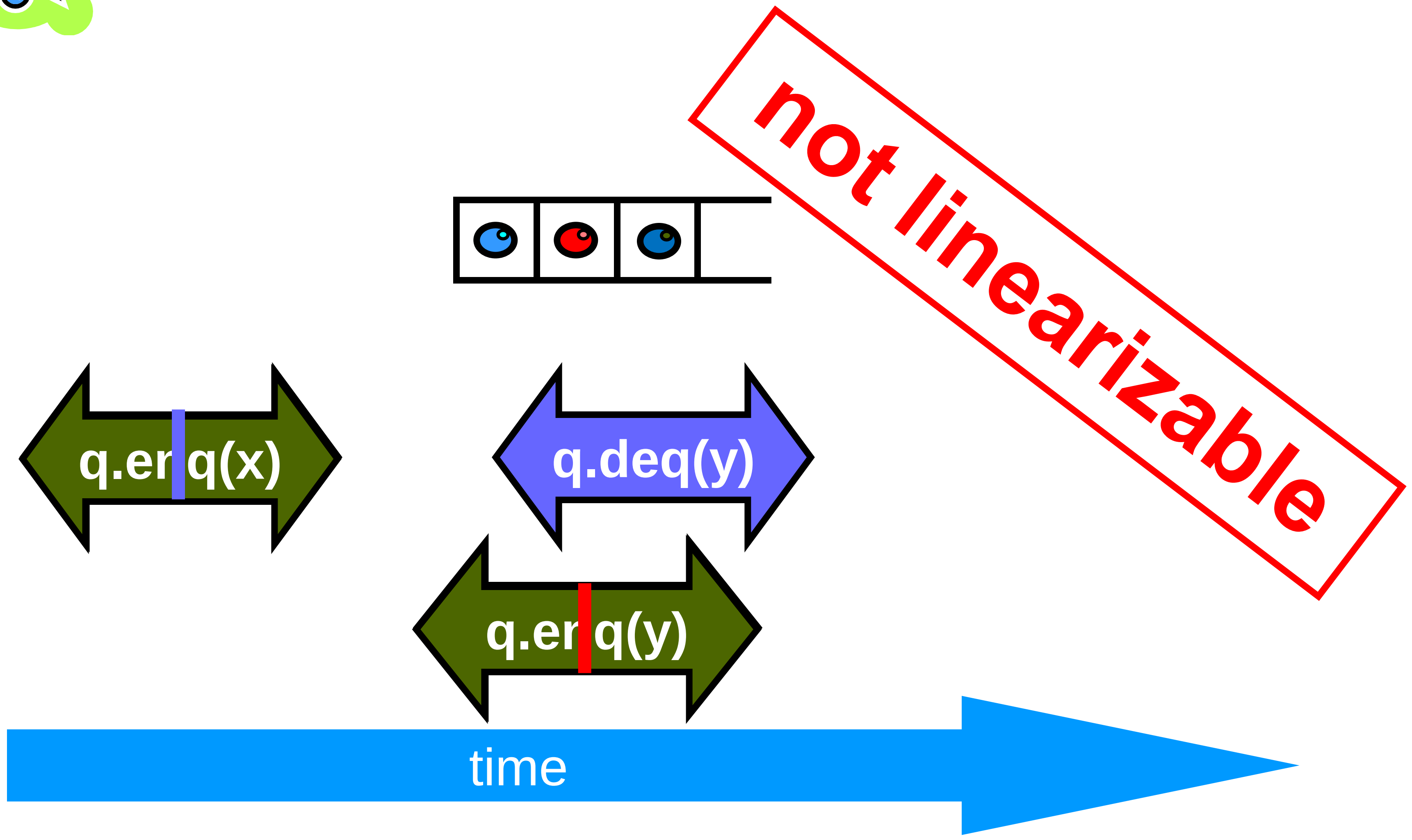
Example



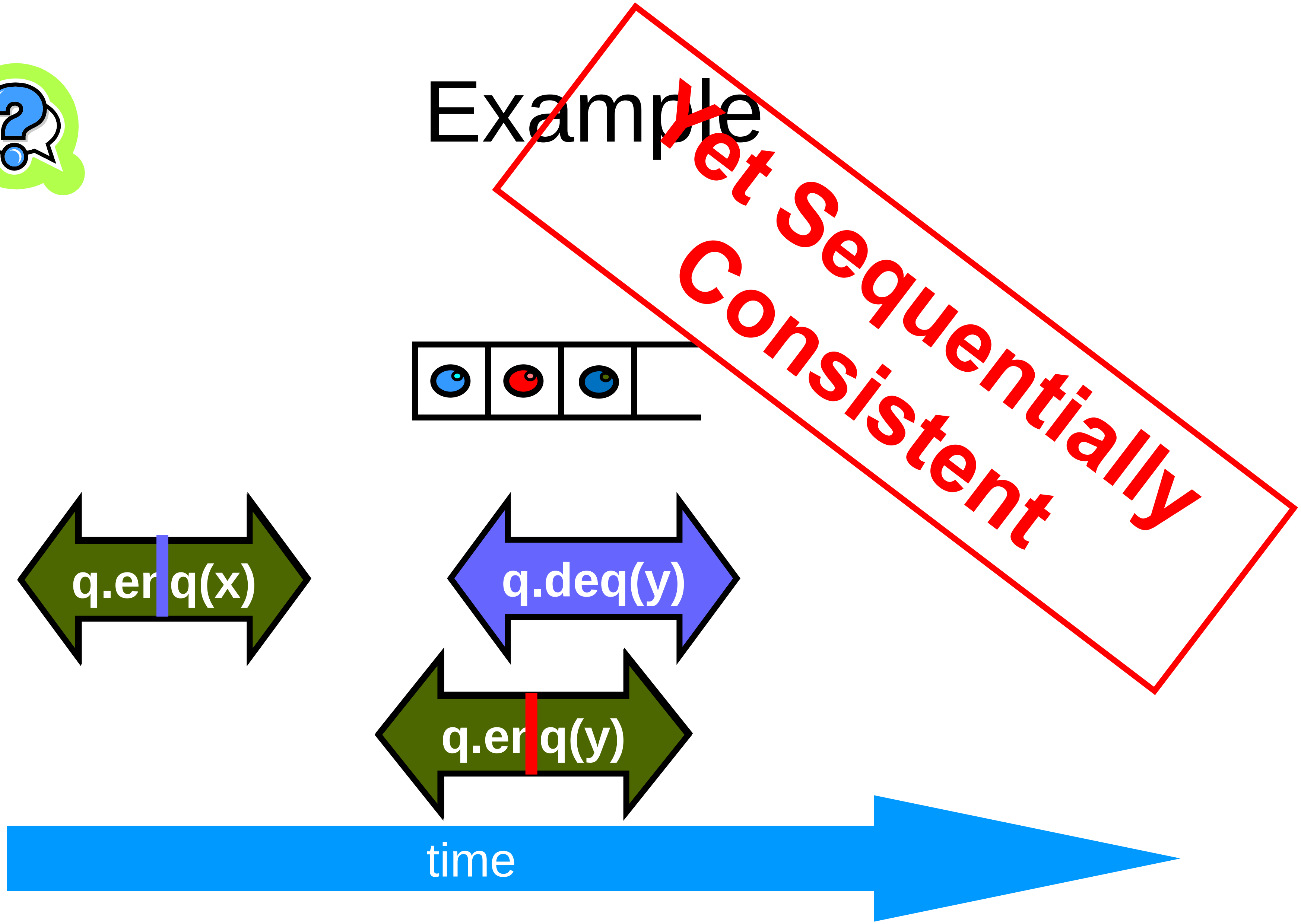
Example



Example



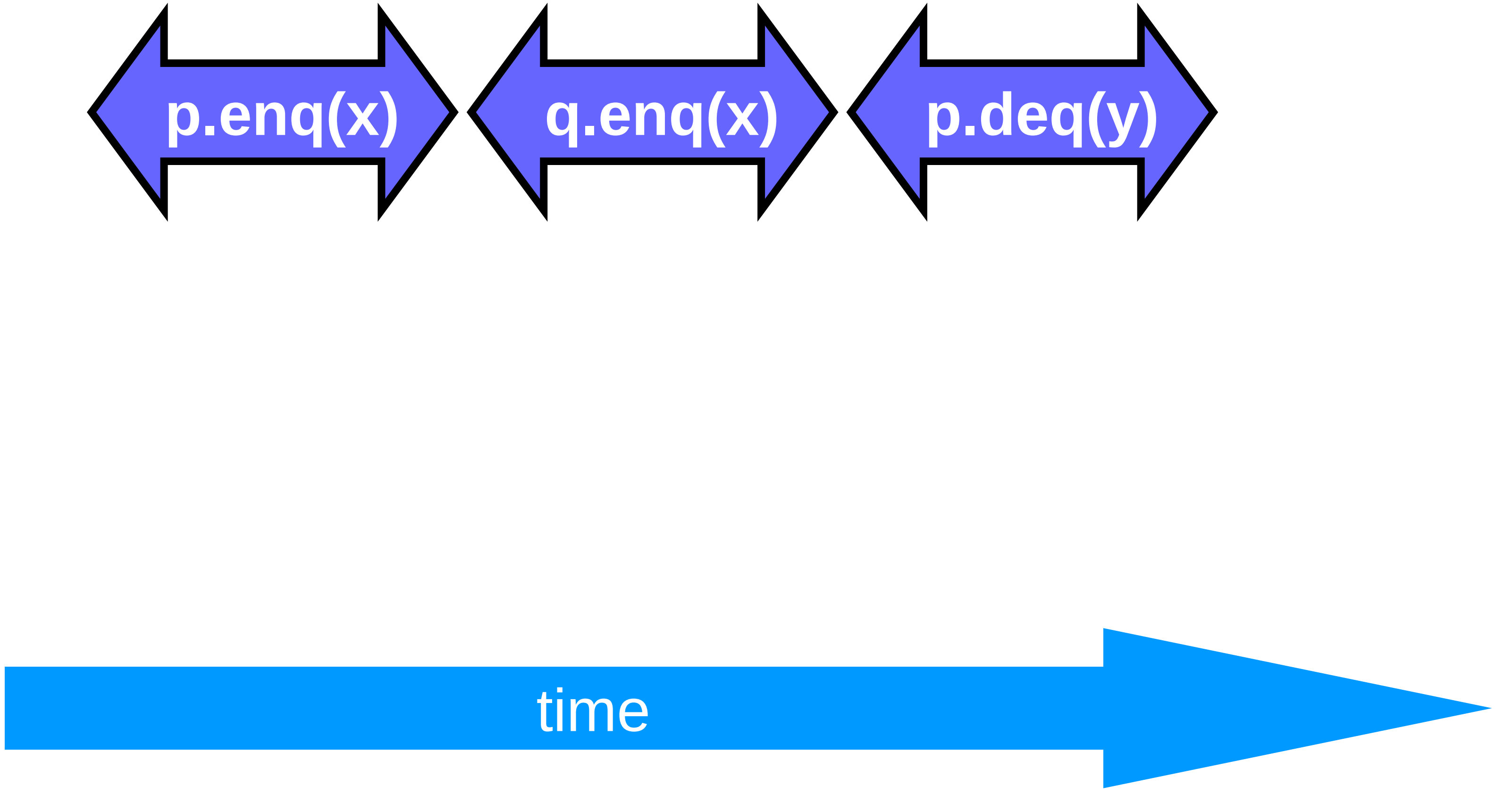
Example



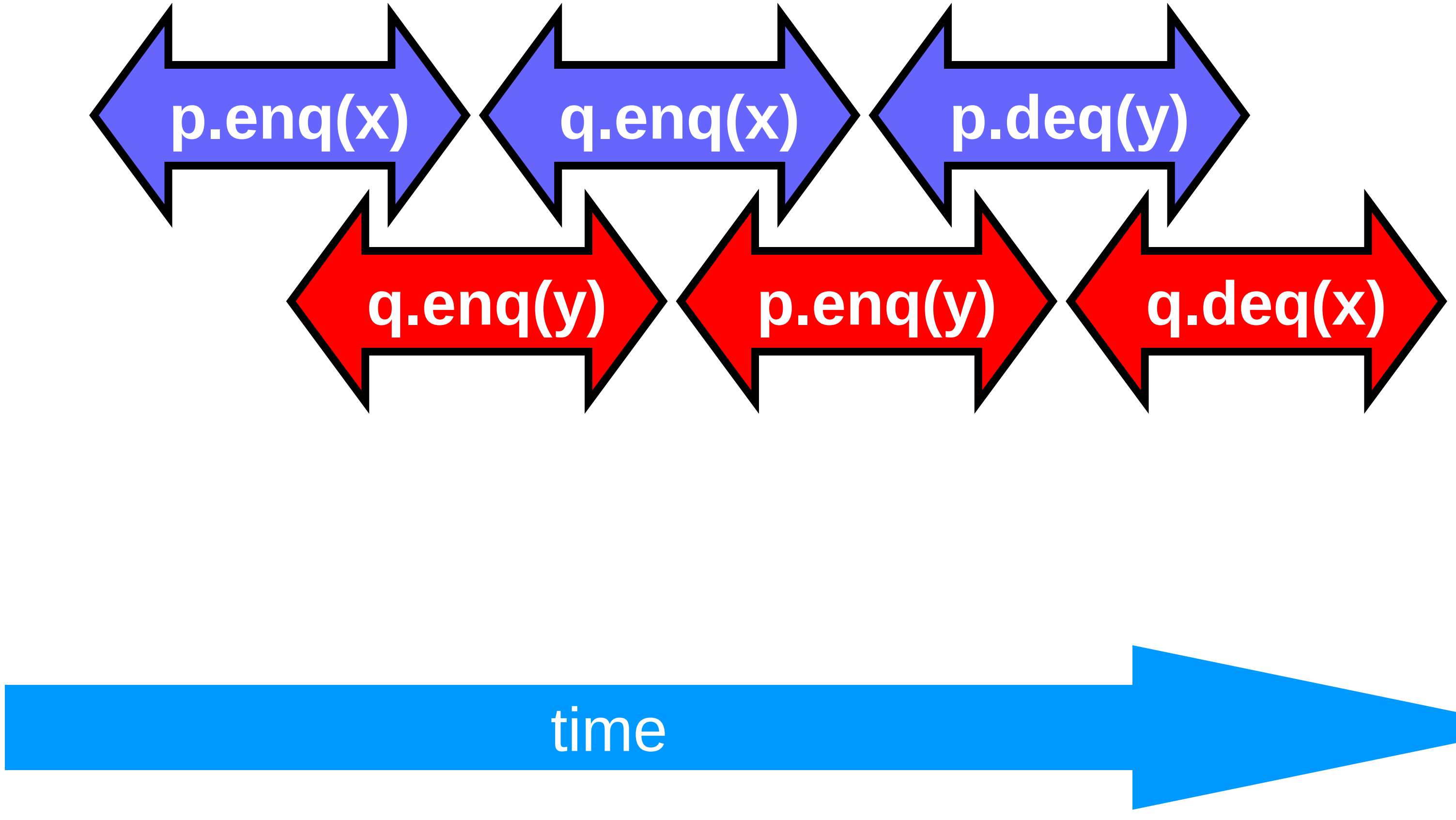
Theorem

Sequential Consistency is not composable

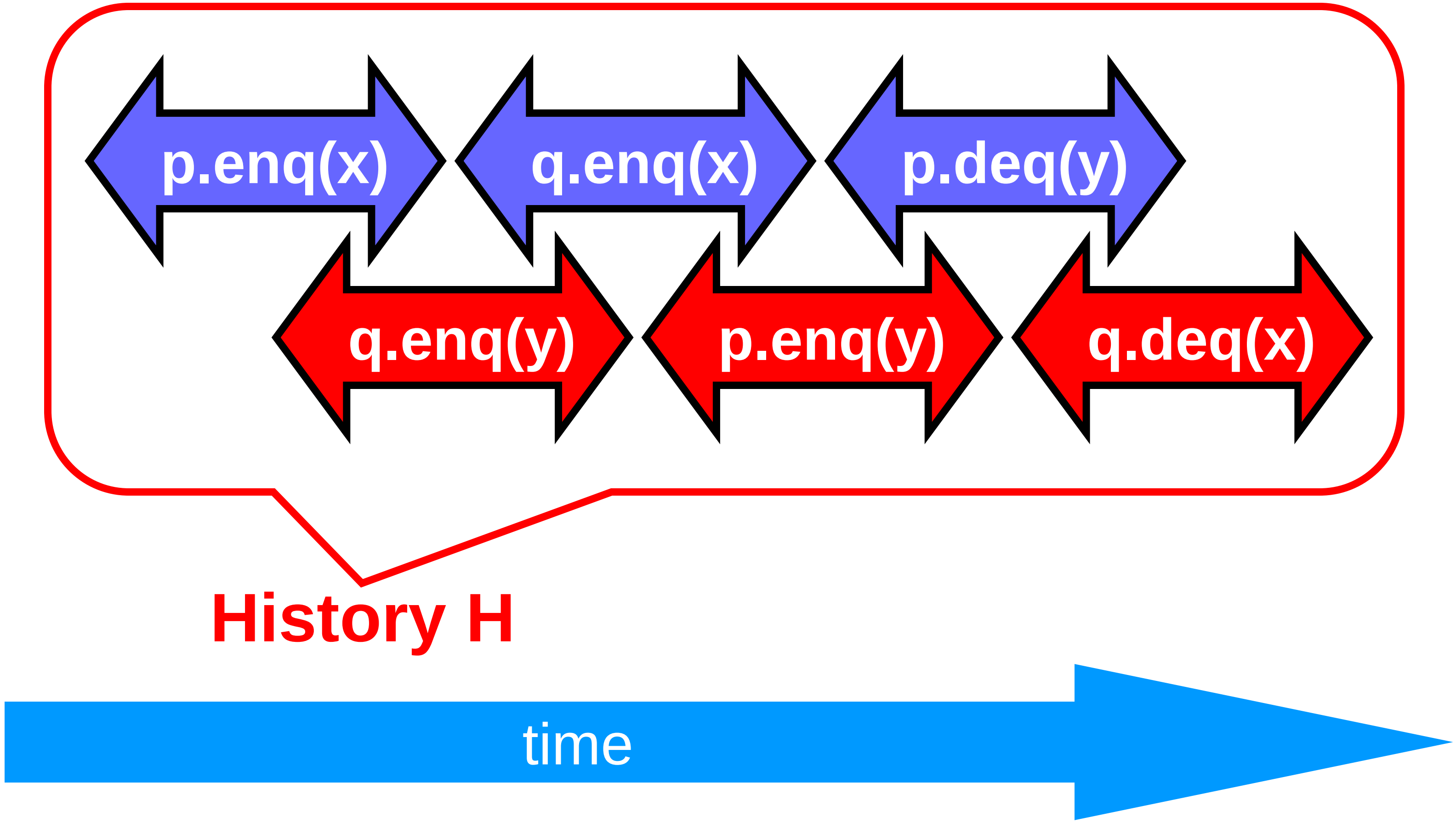
FIFO Queue Example



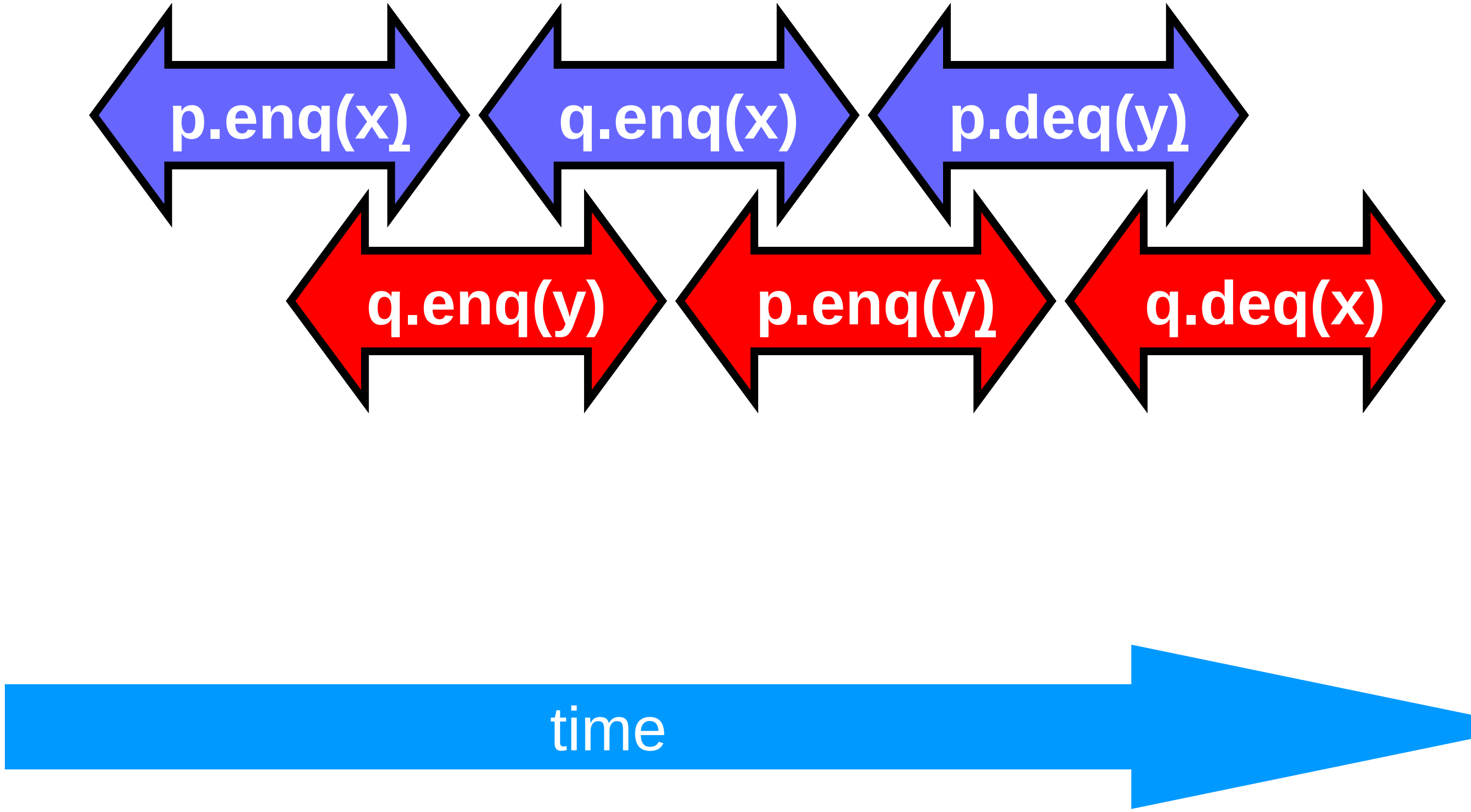
FIFO Queue Example



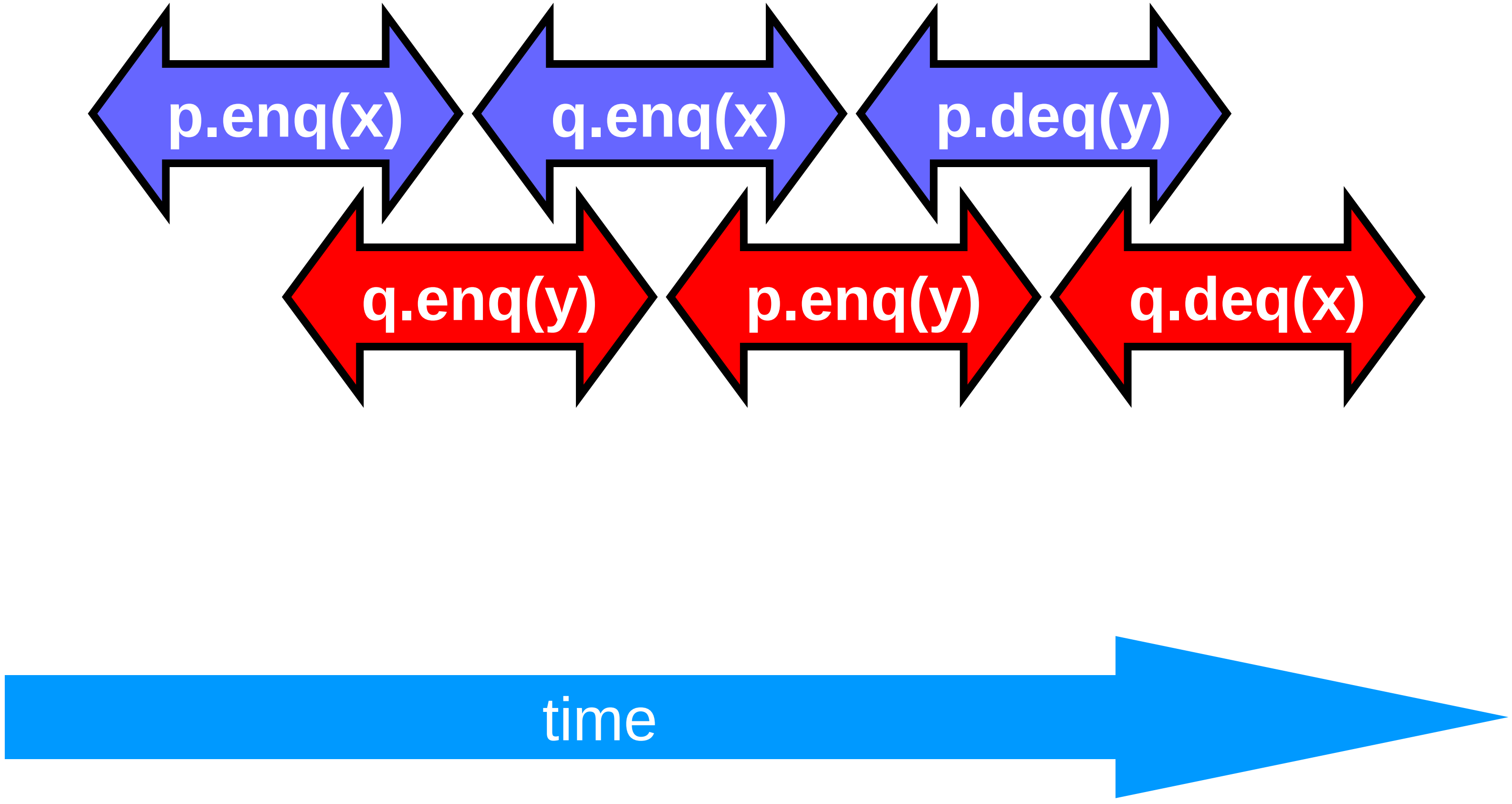
FIFO Queue Example



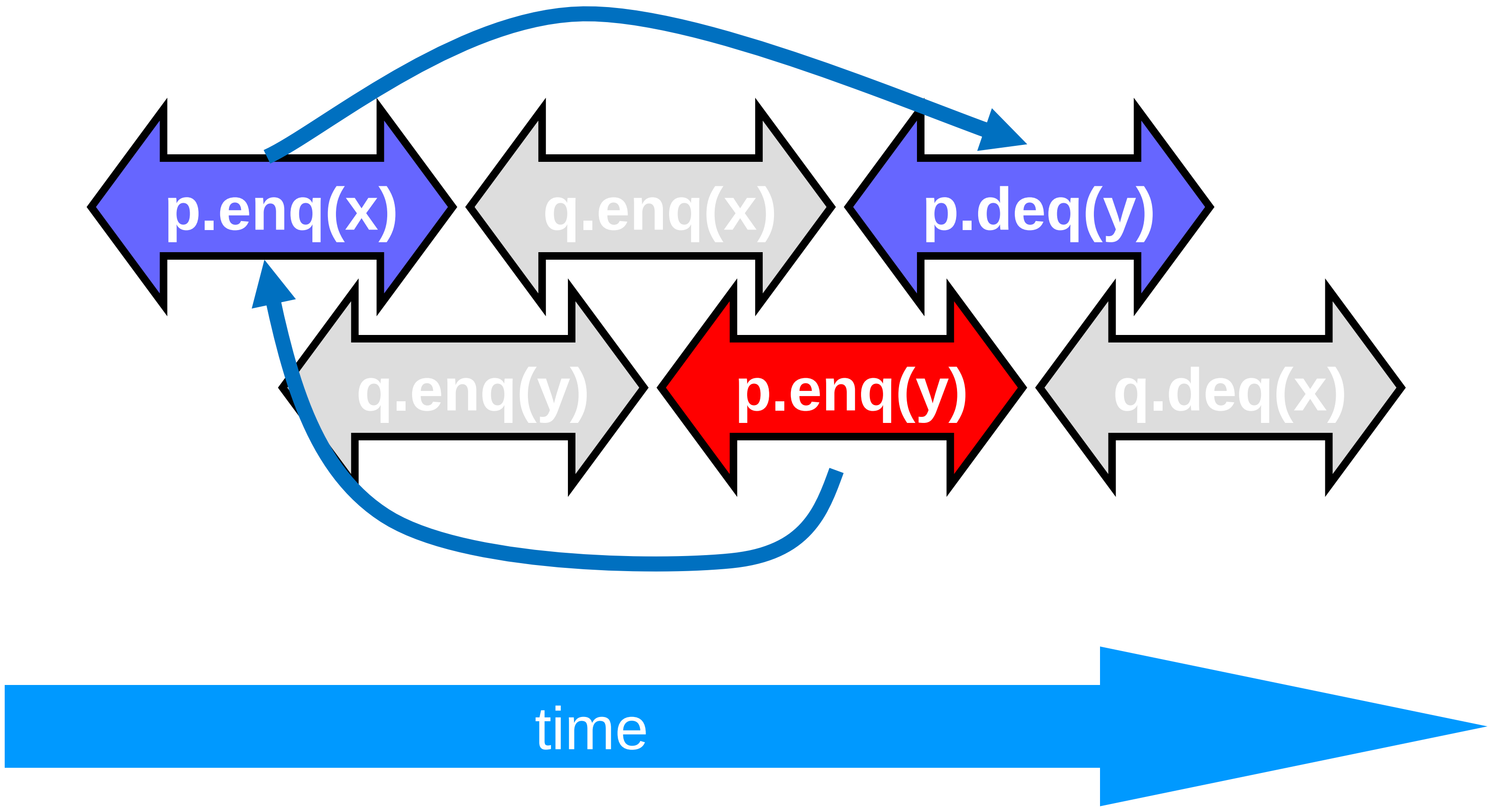
Sequentially Consistent



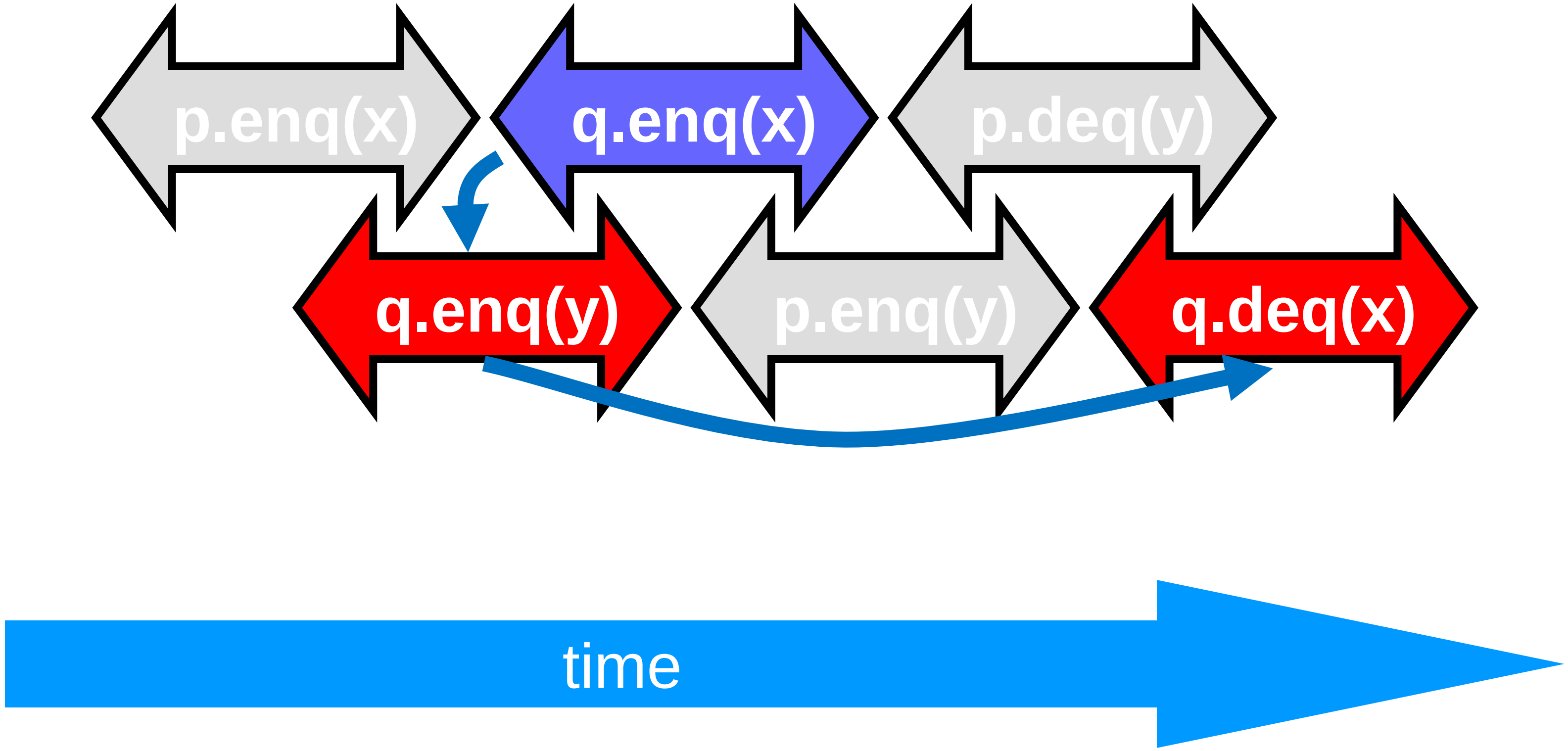
Sequentially Consistent



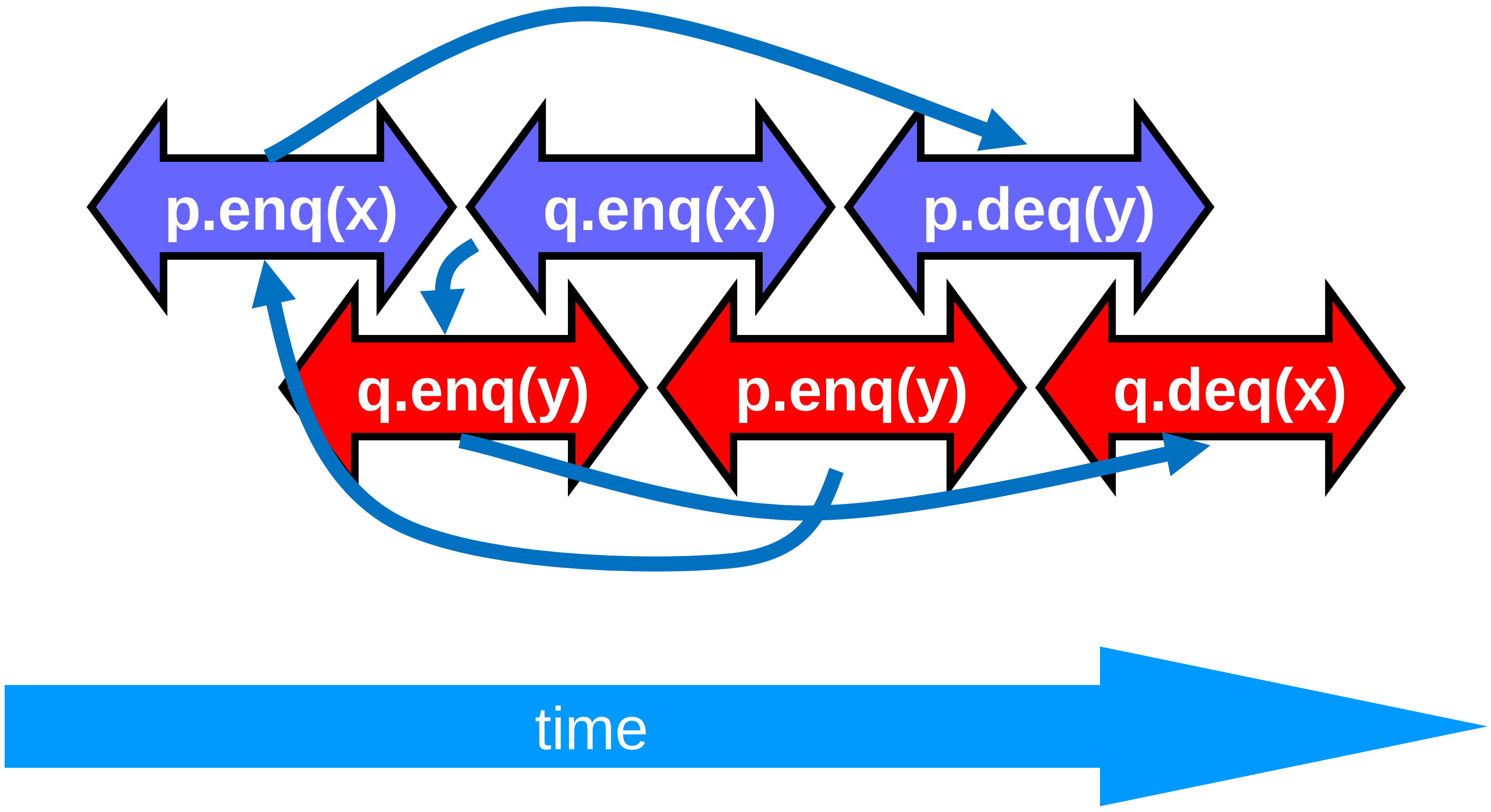
Ordering imposed by p



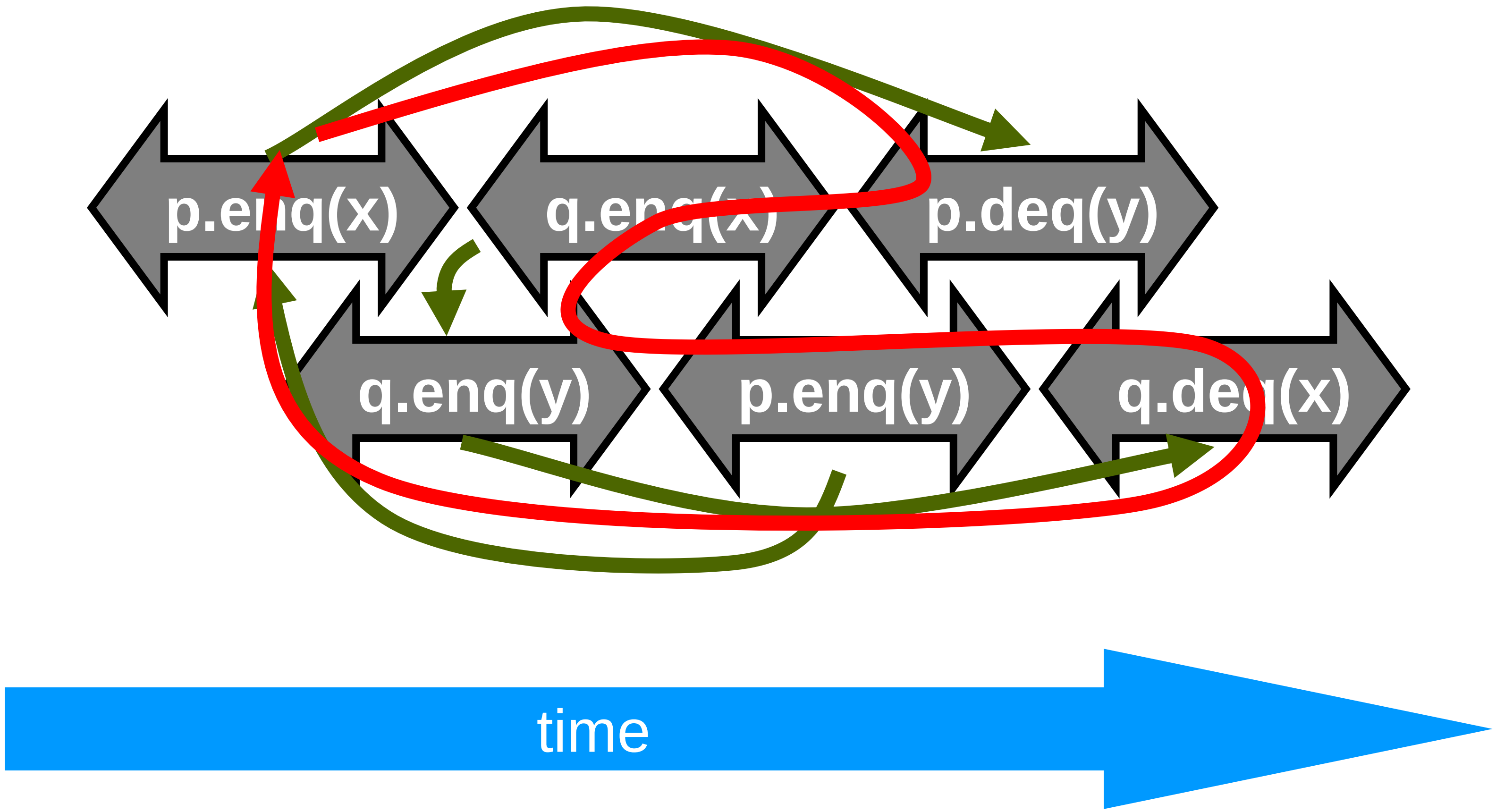
Ordering imposed by q



Ordering imposed by both



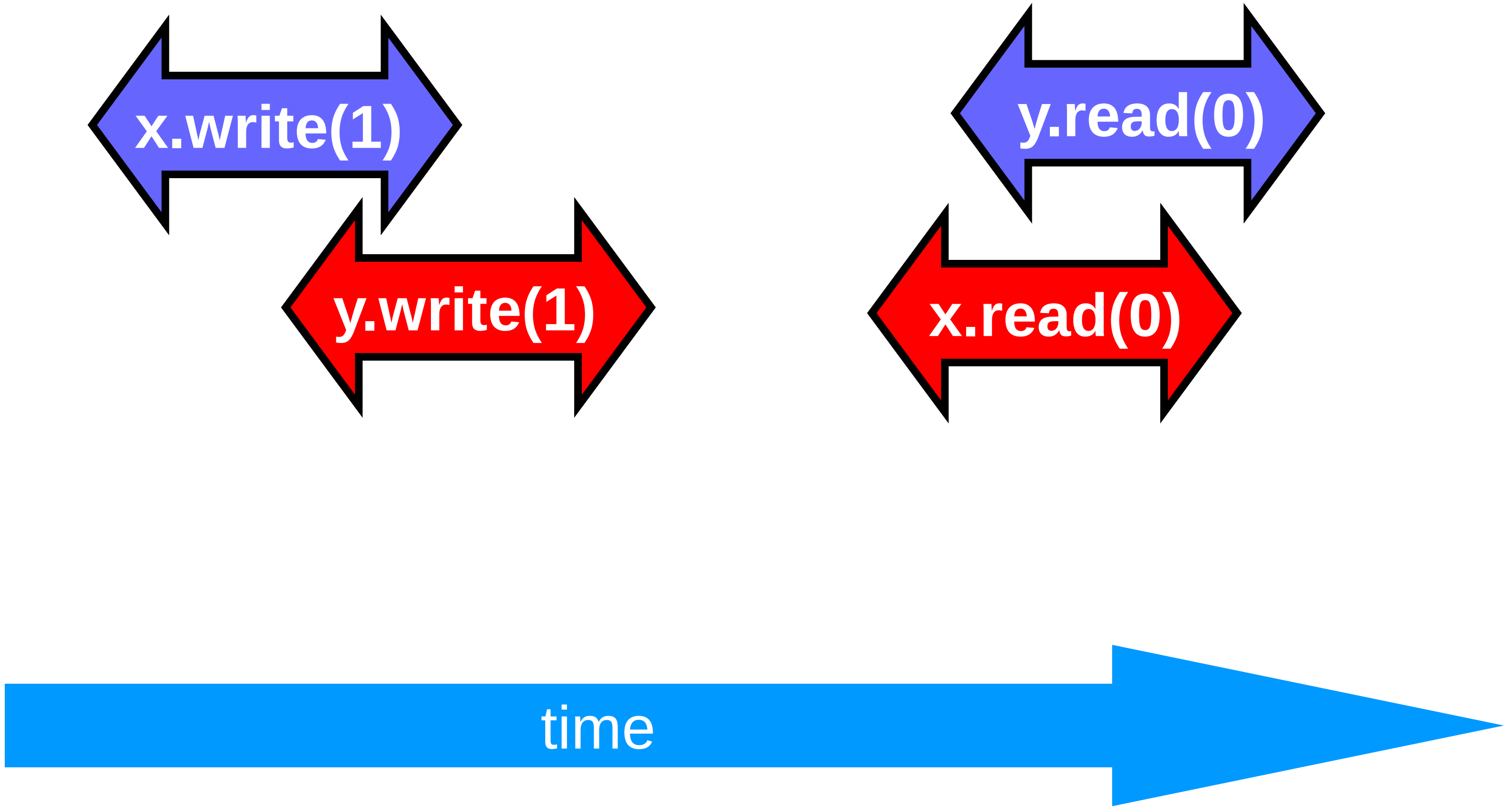
Combining orders



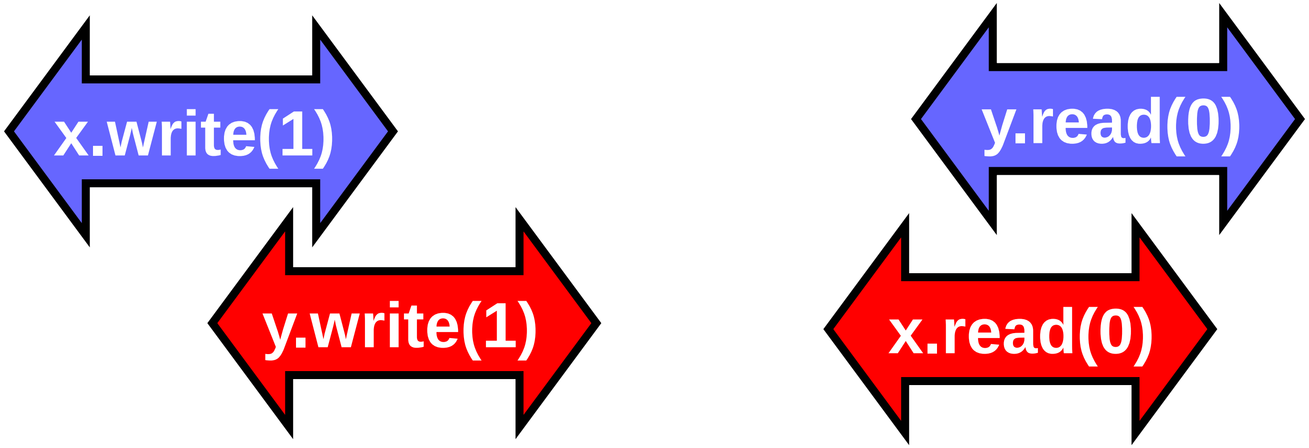
Fact

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

The Flag Example

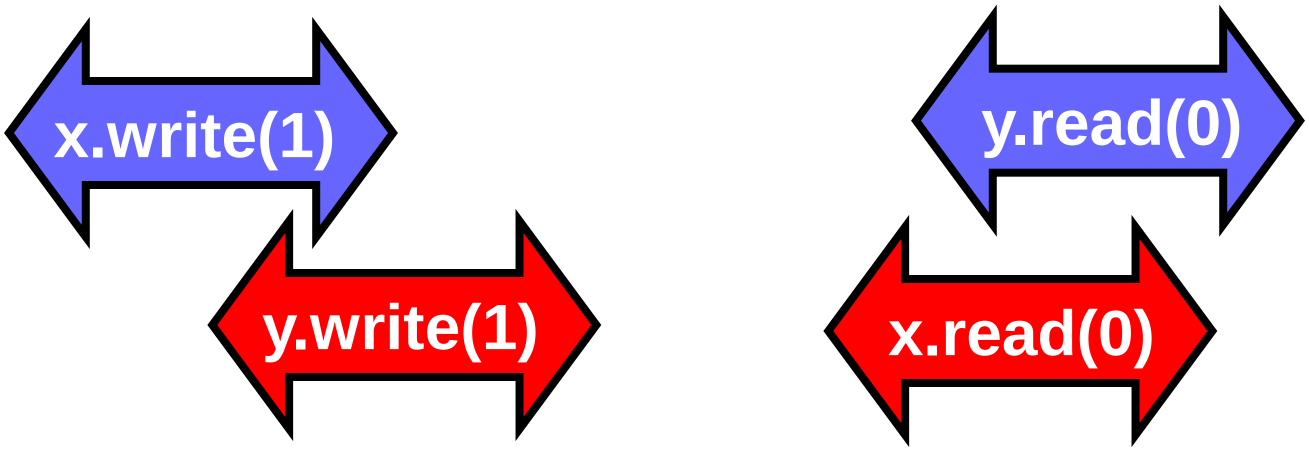


The Flag Example



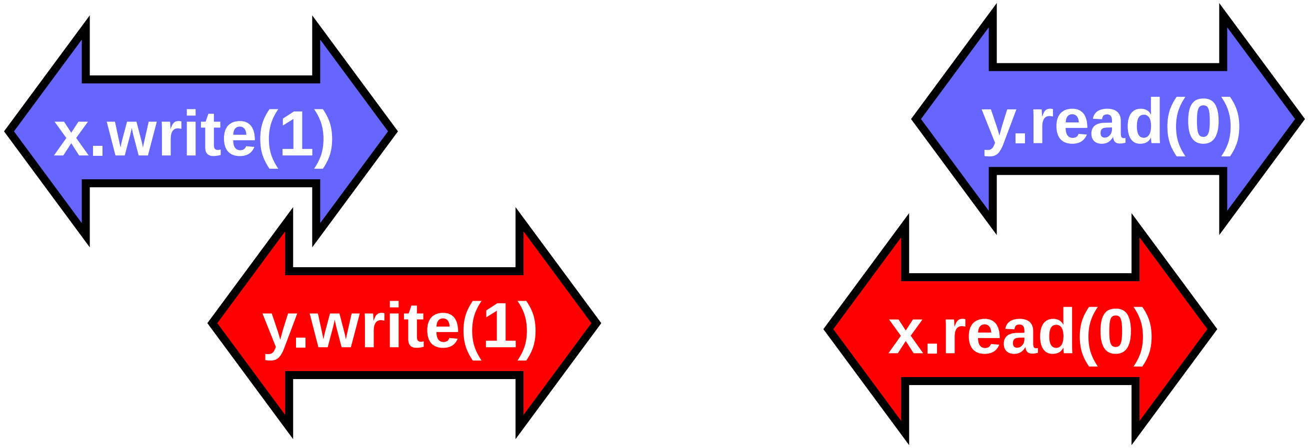
- Each thread's view is sequentially consistent
  - It went first

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

Double-Checked Locking

```
public class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized(Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

unsafe

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"

Bakery Algorithm revisited

```
class Bakery implements Lock {
   volatile boolean[] flag;
   volatile Label[] label;
  public Bakery (int n) {
    flag  = new boolean[n];
    label = new Label[n];
    for (int i = 0; i < n; i++) {
       flag[i] = false; label[i] = 0;
    }
  }
 …
```

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

Linearizability

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

Correctness: Linearizability

- 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

Progress

- We saw an implementation whose methods were lock-based (deadlock-free)
- We saw an implementation whose methods did not use locks (lock-free)
- How do they relate?

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.

Progress Conditions

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

Summary

- We will look at *linearizable blocking* and *non-blocking* implementations of objects.