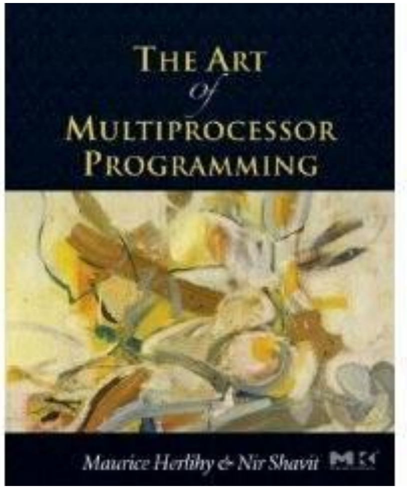
Concurrent Objects

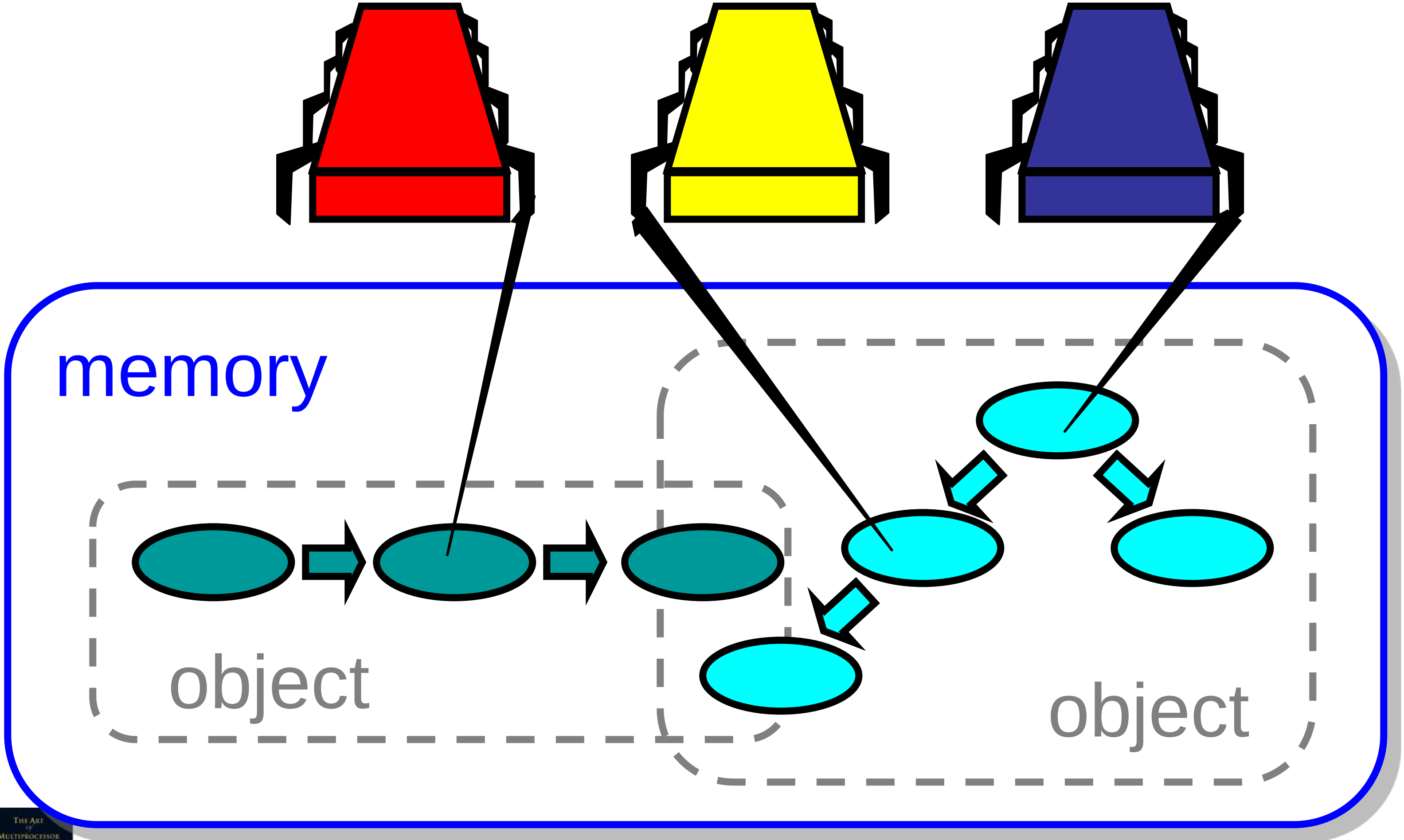


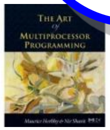
Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

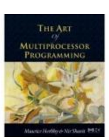
Concurrent Computation





Objectivism

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

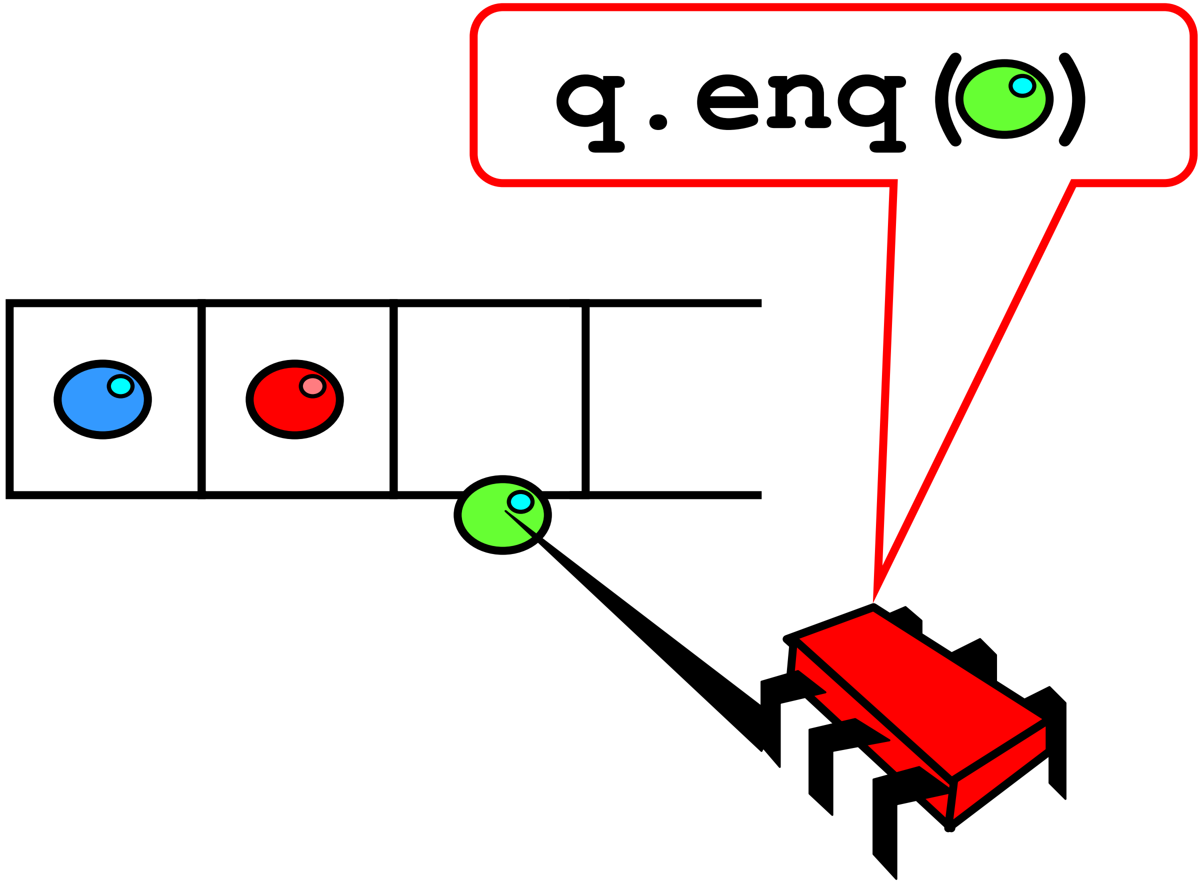


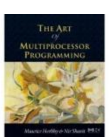
Objectivism

- What is a concurrent object?
  – How do we **describe** one?

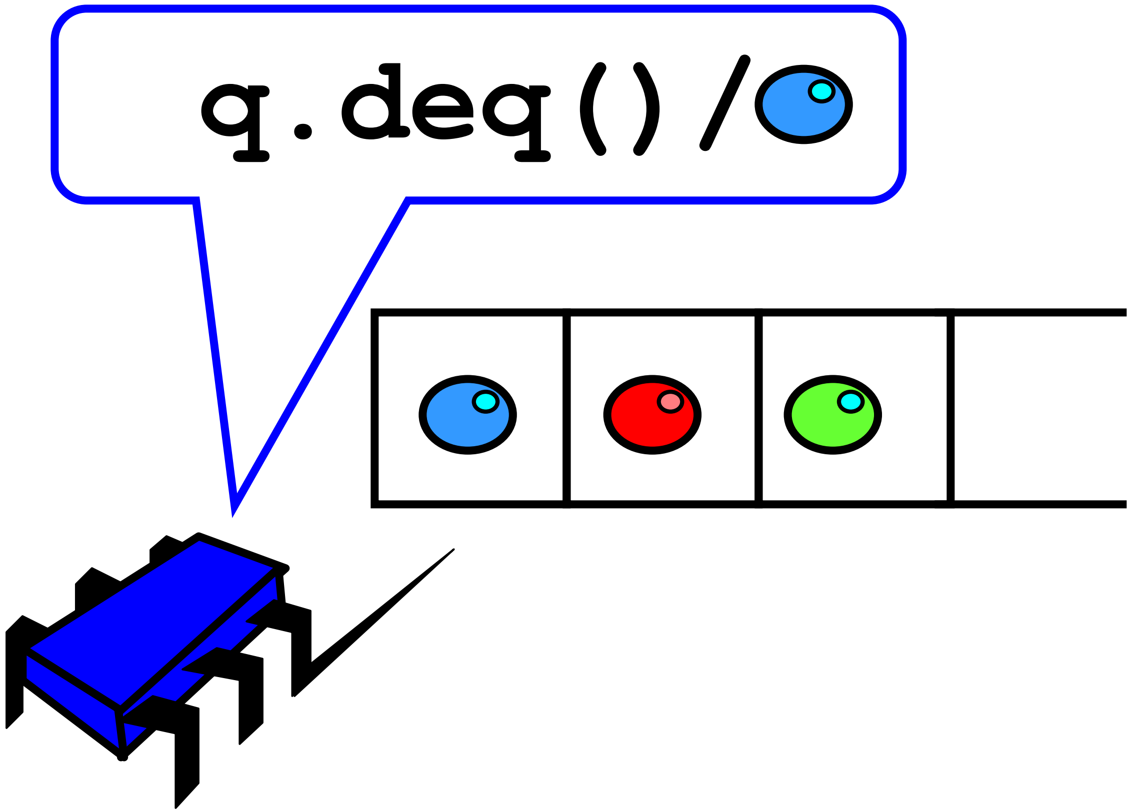
  – How do we **tell if we're right**?

FIFO Queue: Enqueue Method

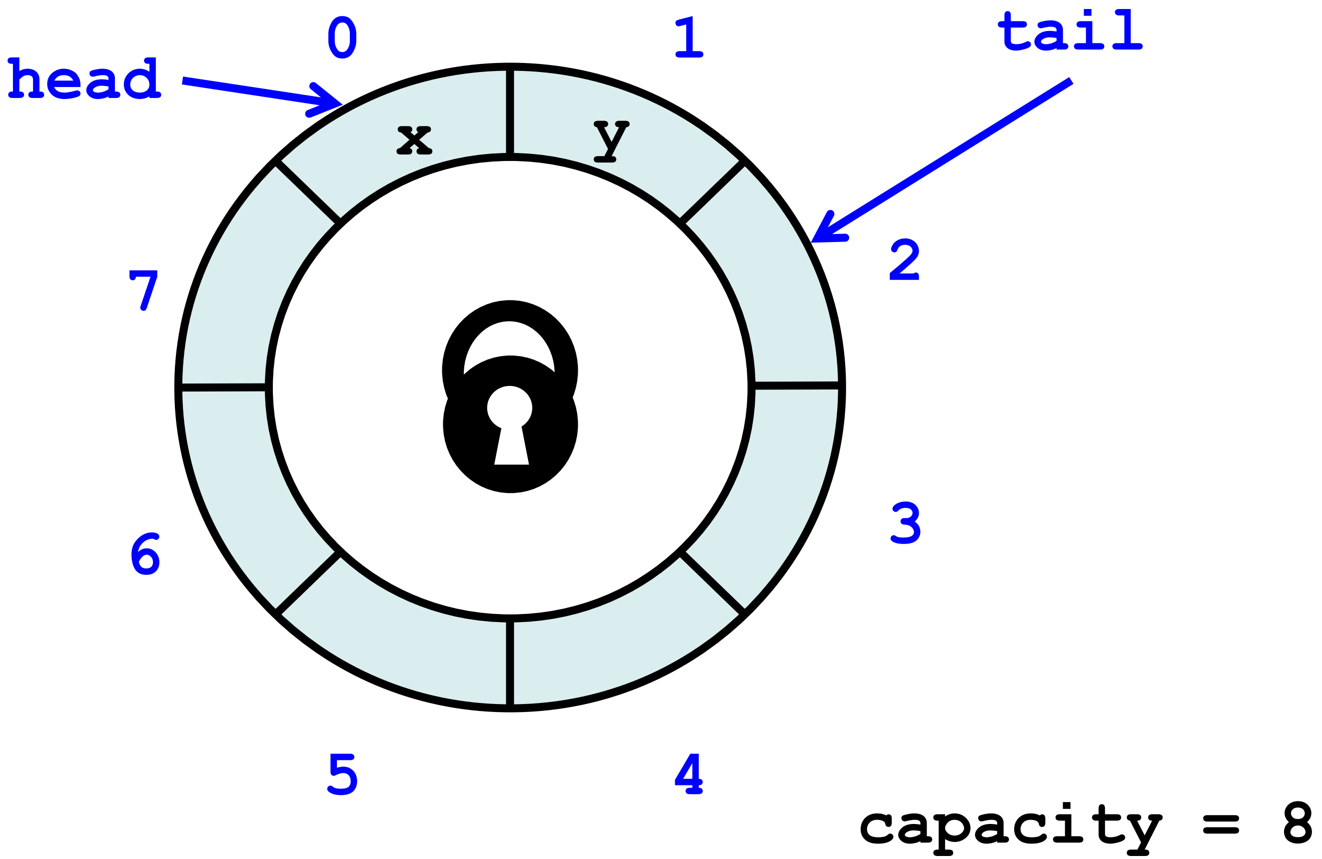


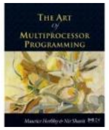


FIFO Queue: Dequeue Method

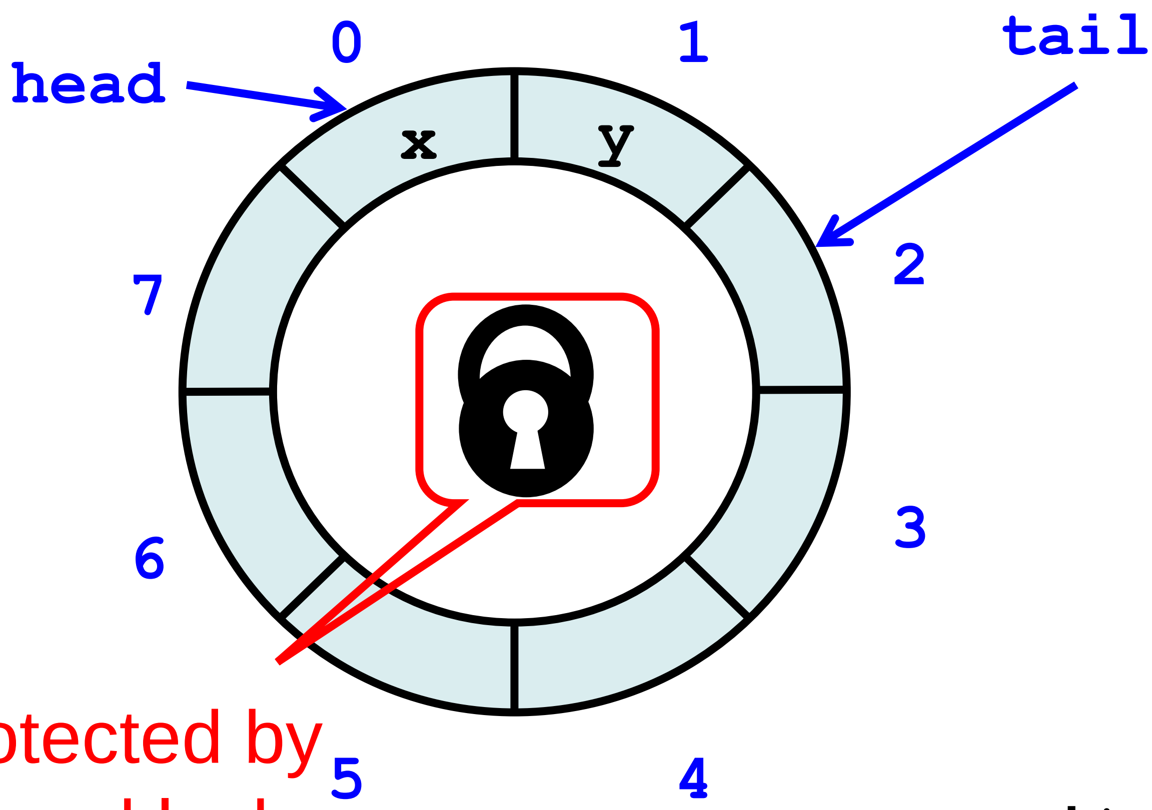


Lock-Based Queue





Lock-Based Queue

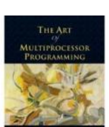


Fields protected by single shared lock

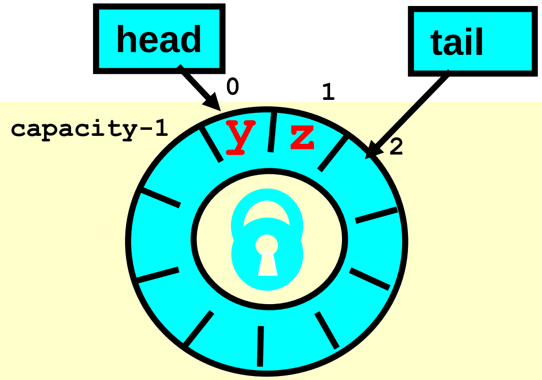
capacity = 8

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];
}
```

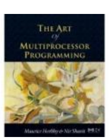


A Lock-Based Queue



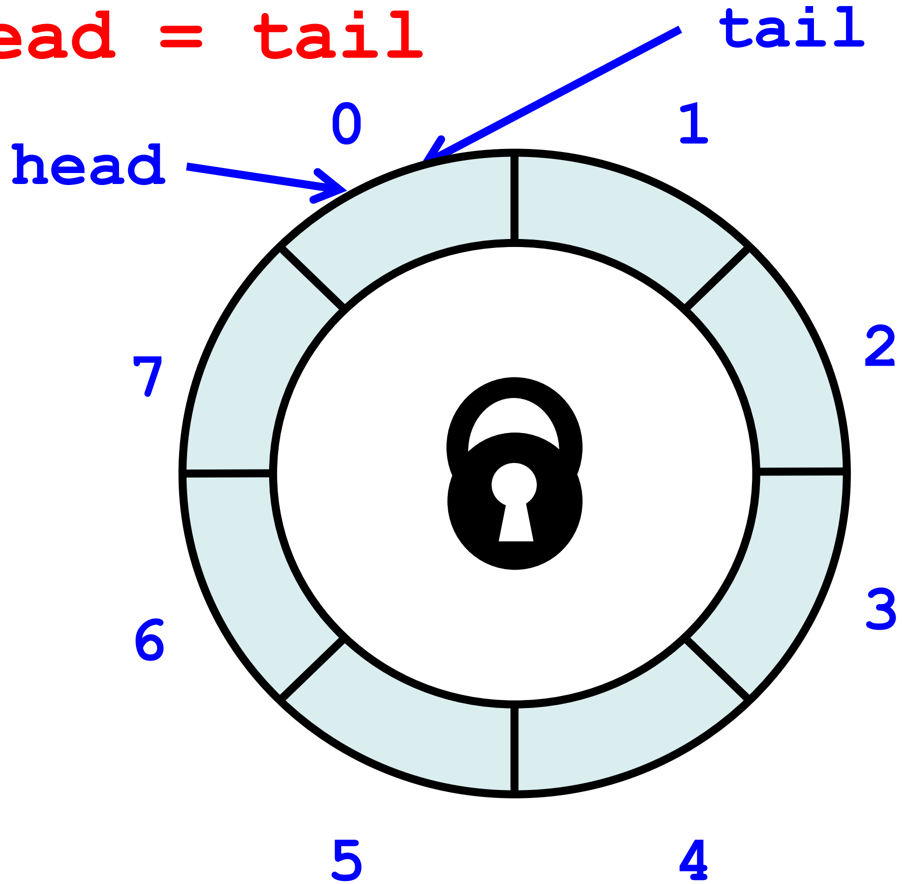

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Fields protected by single shared lock

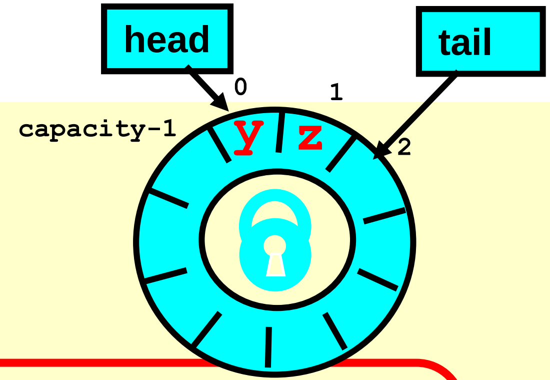


Lock-Based Queue

Initially: **`head = tail`**

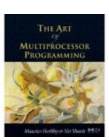


A Lock-Based Queue

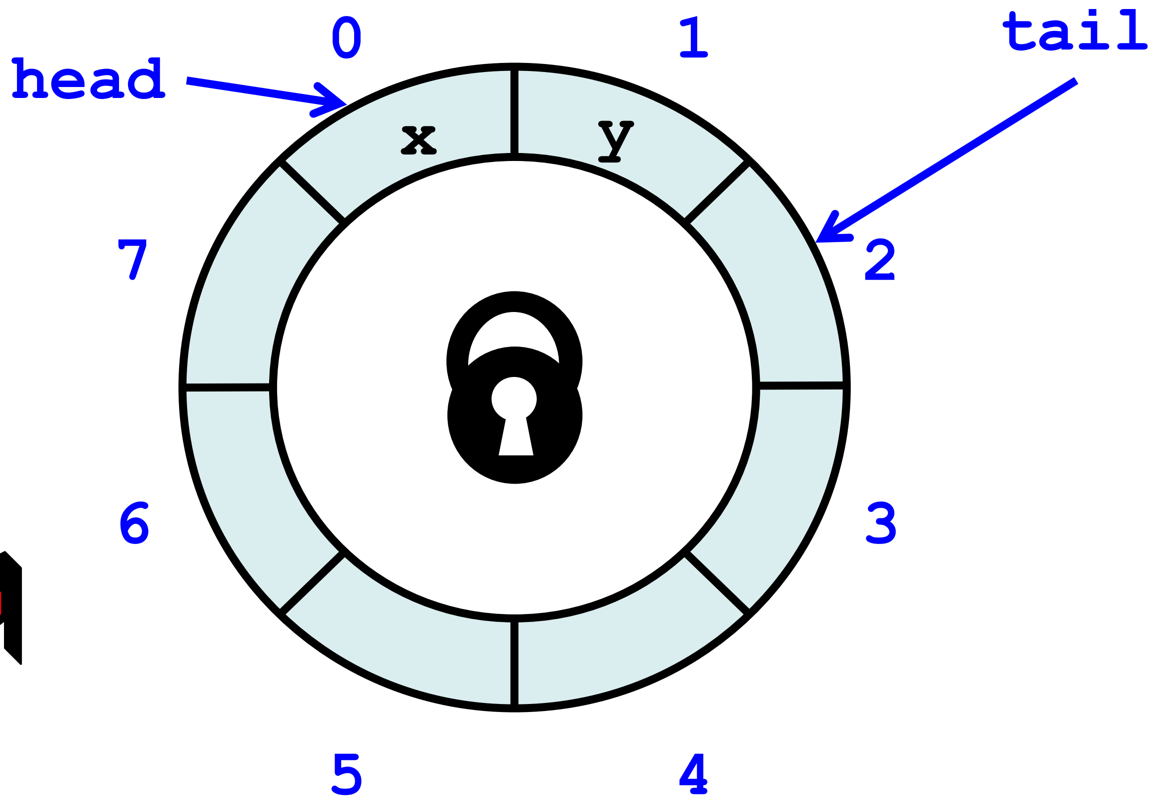


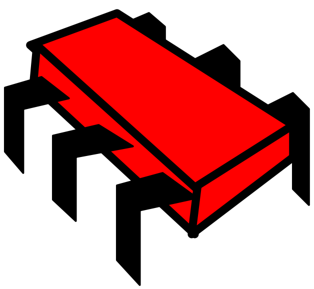

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

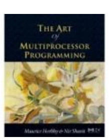
Initially **`head = tail`**



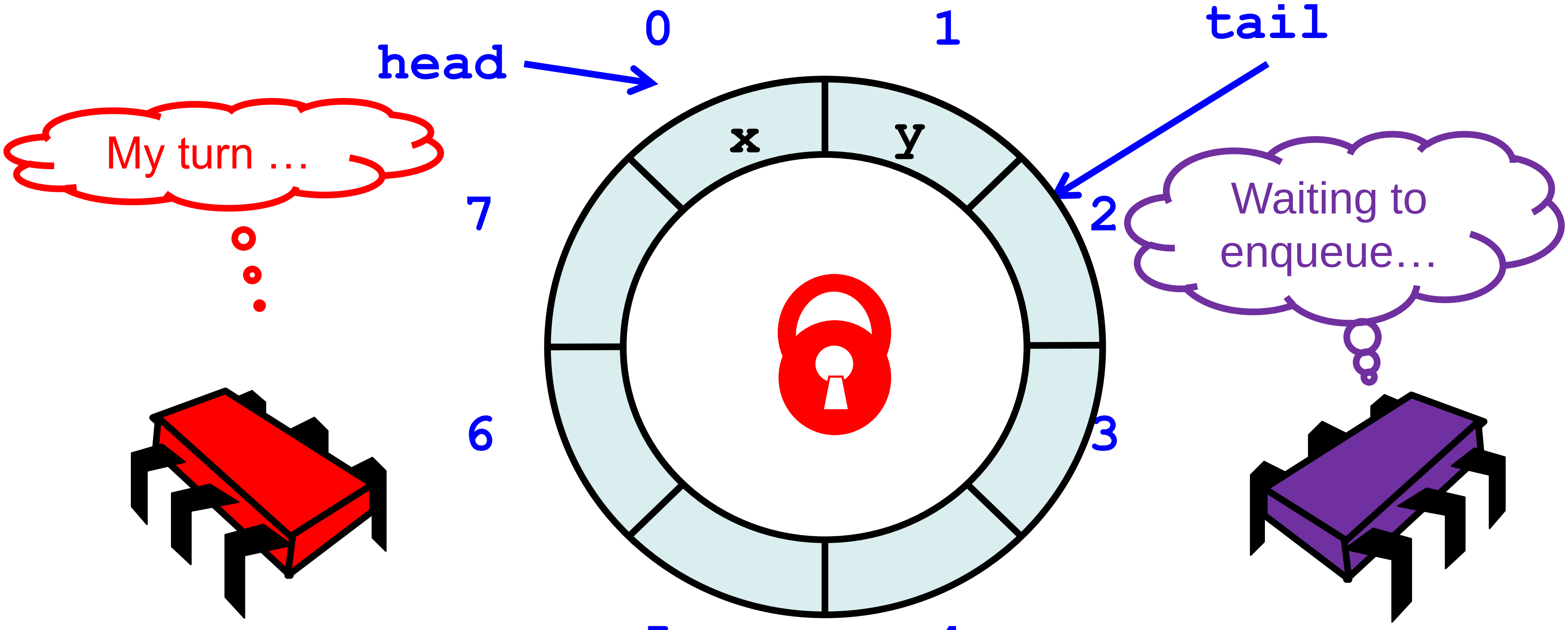
Lock-Based **deq()**

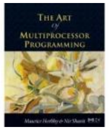






Acquire Lock

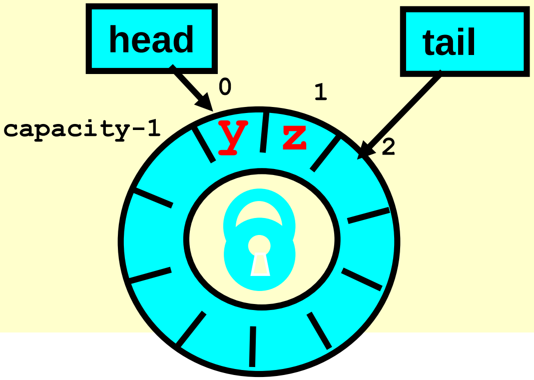




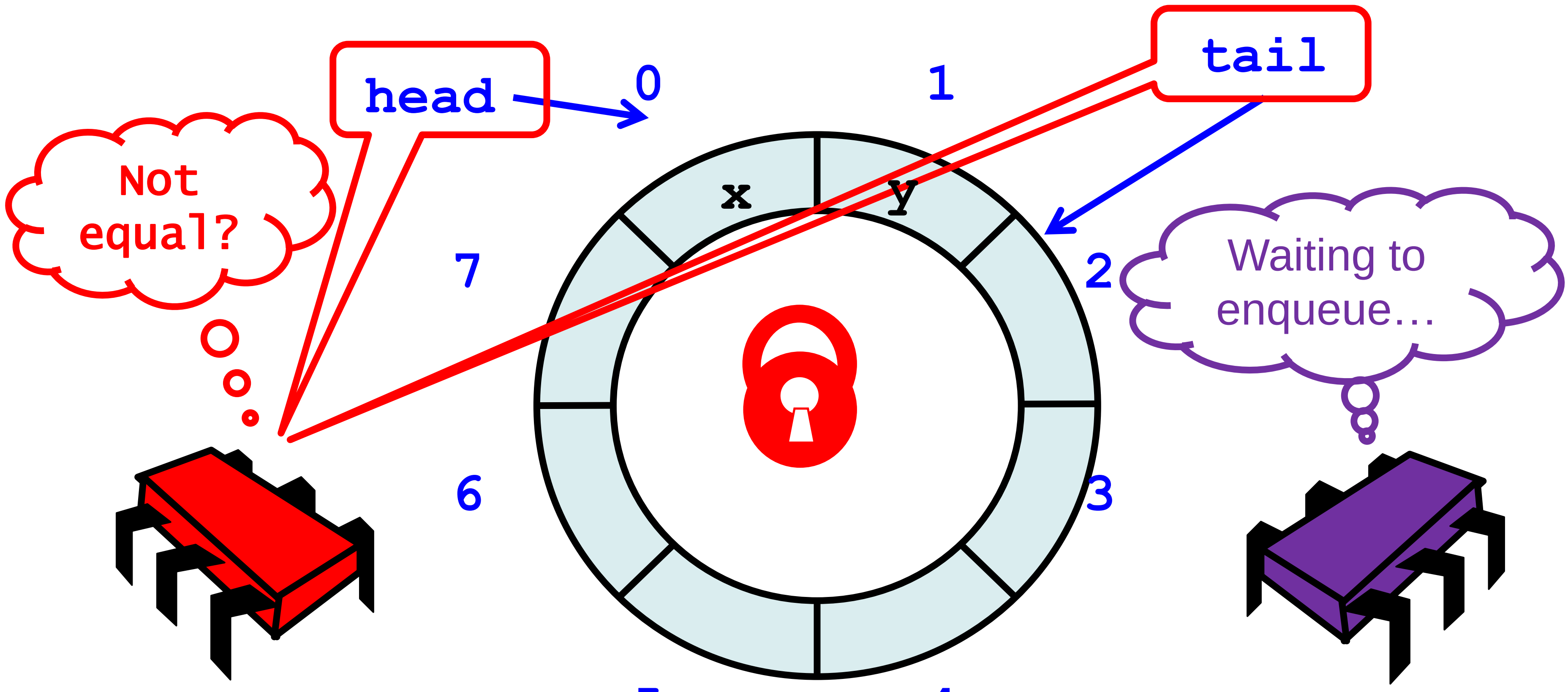
Implementation: **deq()**

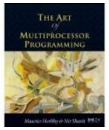
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Acquire lock at method start



Check if Non-Empty

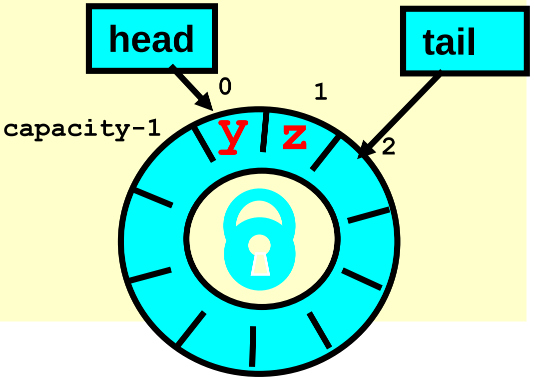


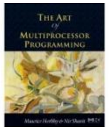


Implementation: **deq()**

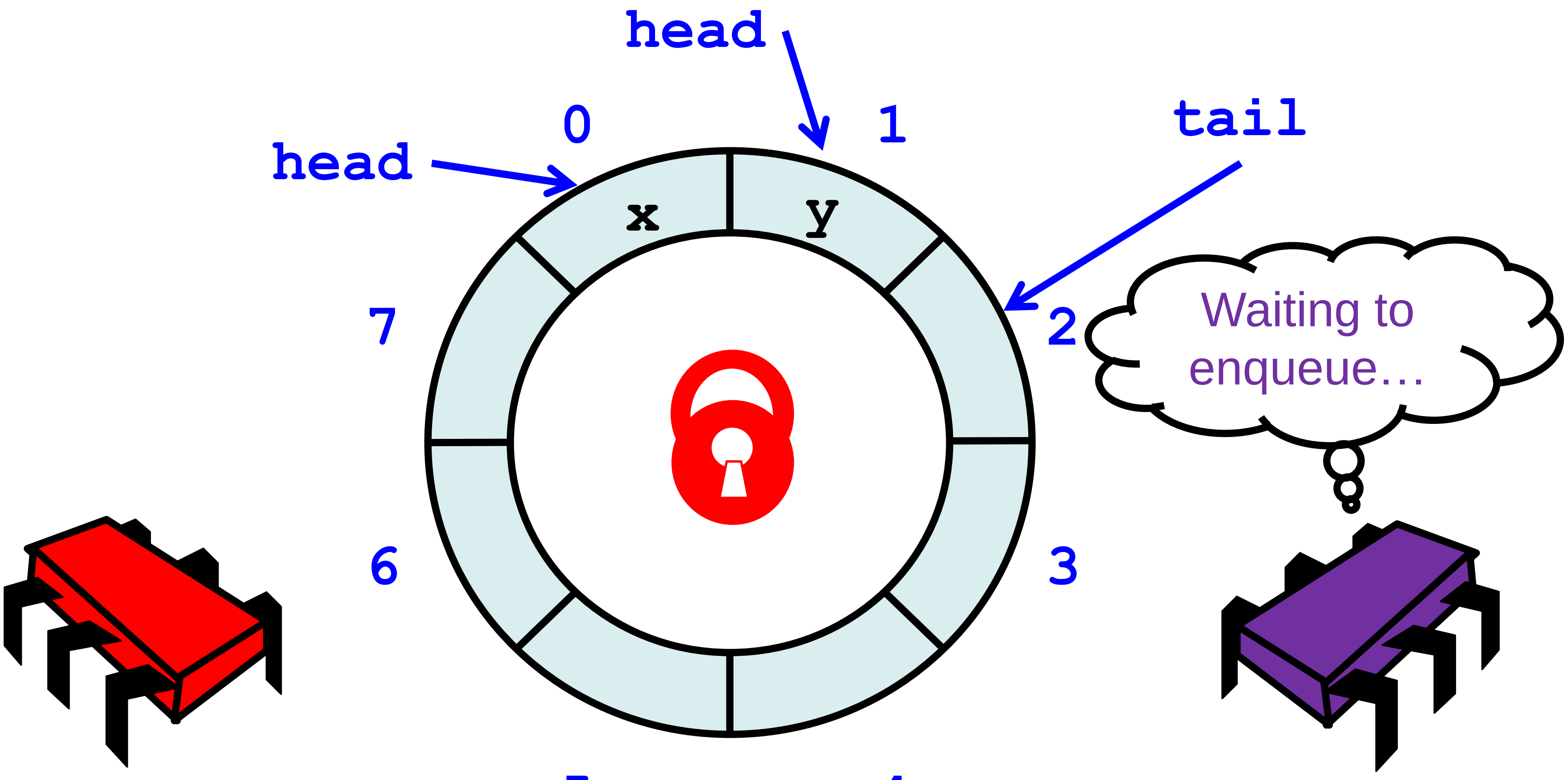
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

If queue empty throw exception



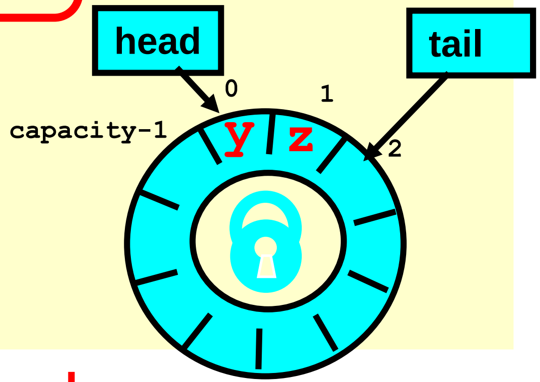


Modify the Queue



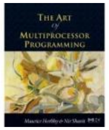
Implementation: **deq()**

```
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?

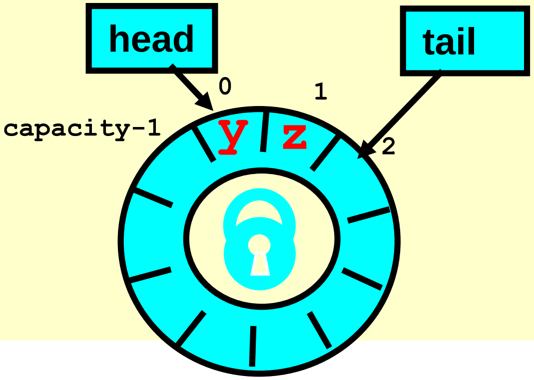
Remove item and update head

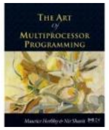


Implementation: **deq()**

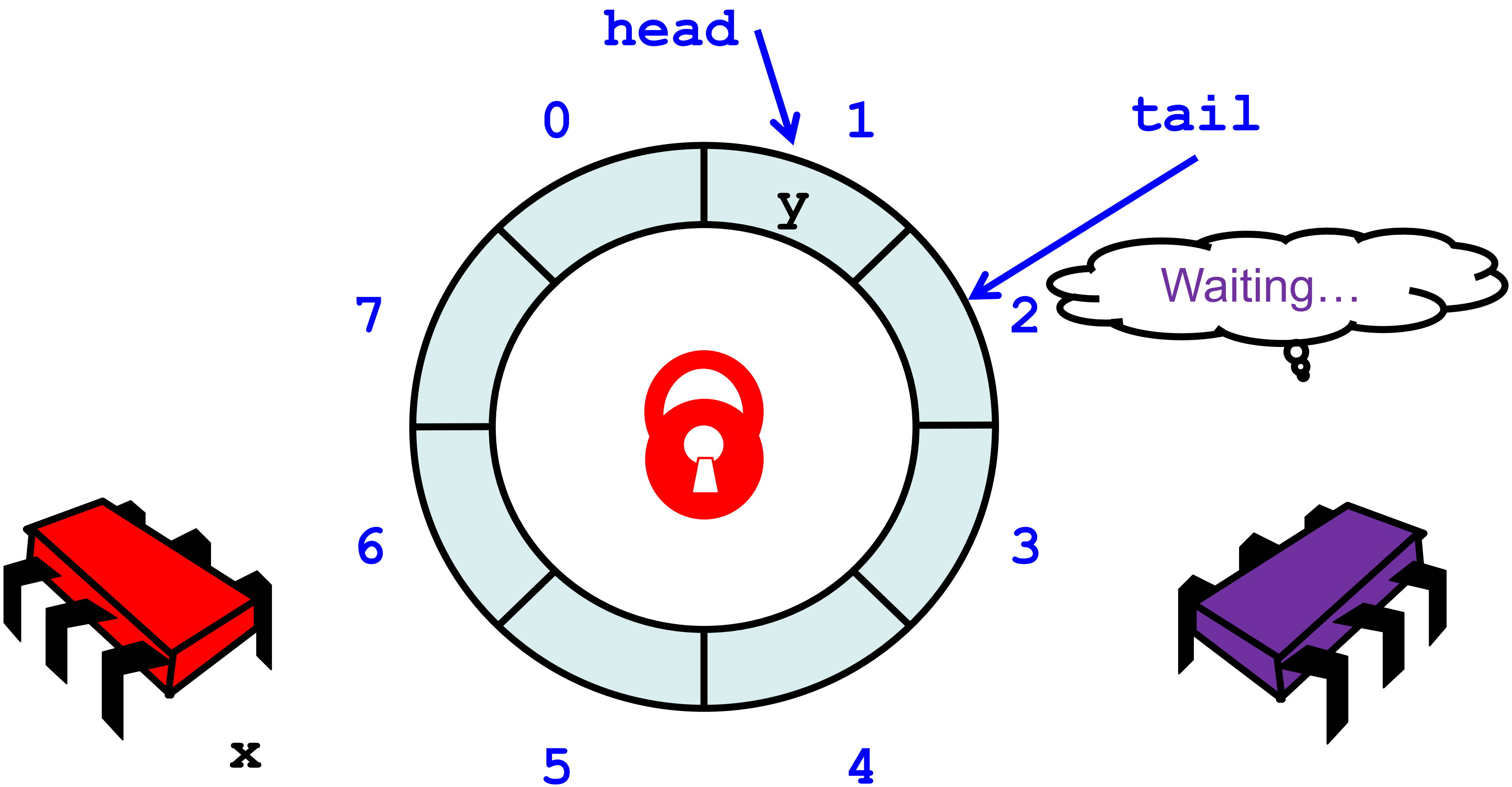
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Return result

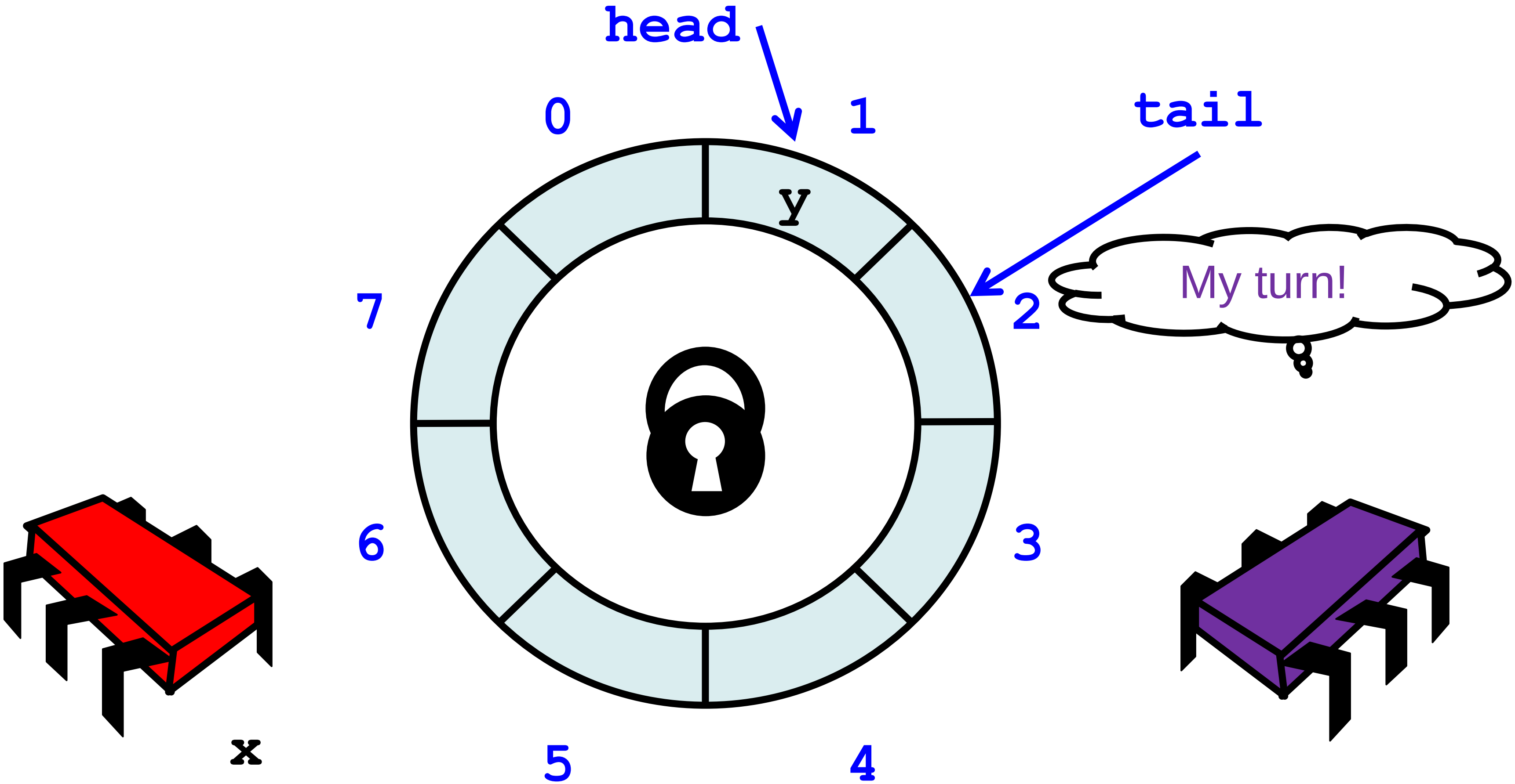


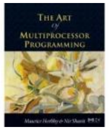


Release the Lock



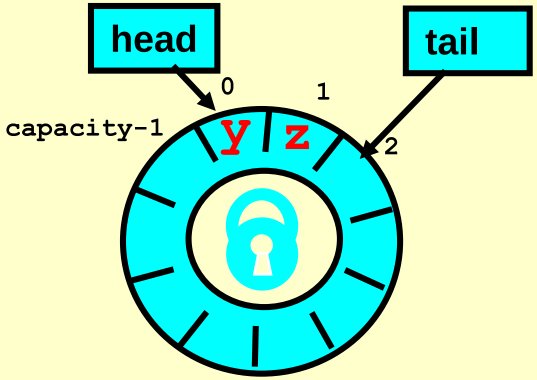
Release the Lock



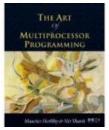


Implementation: `deq()`

```
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();
  }
}
```



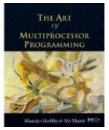
Release lock no matter what!



Implementation: **deq()**

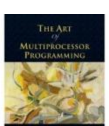
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Should be correct because modifications are mutually exclusive…

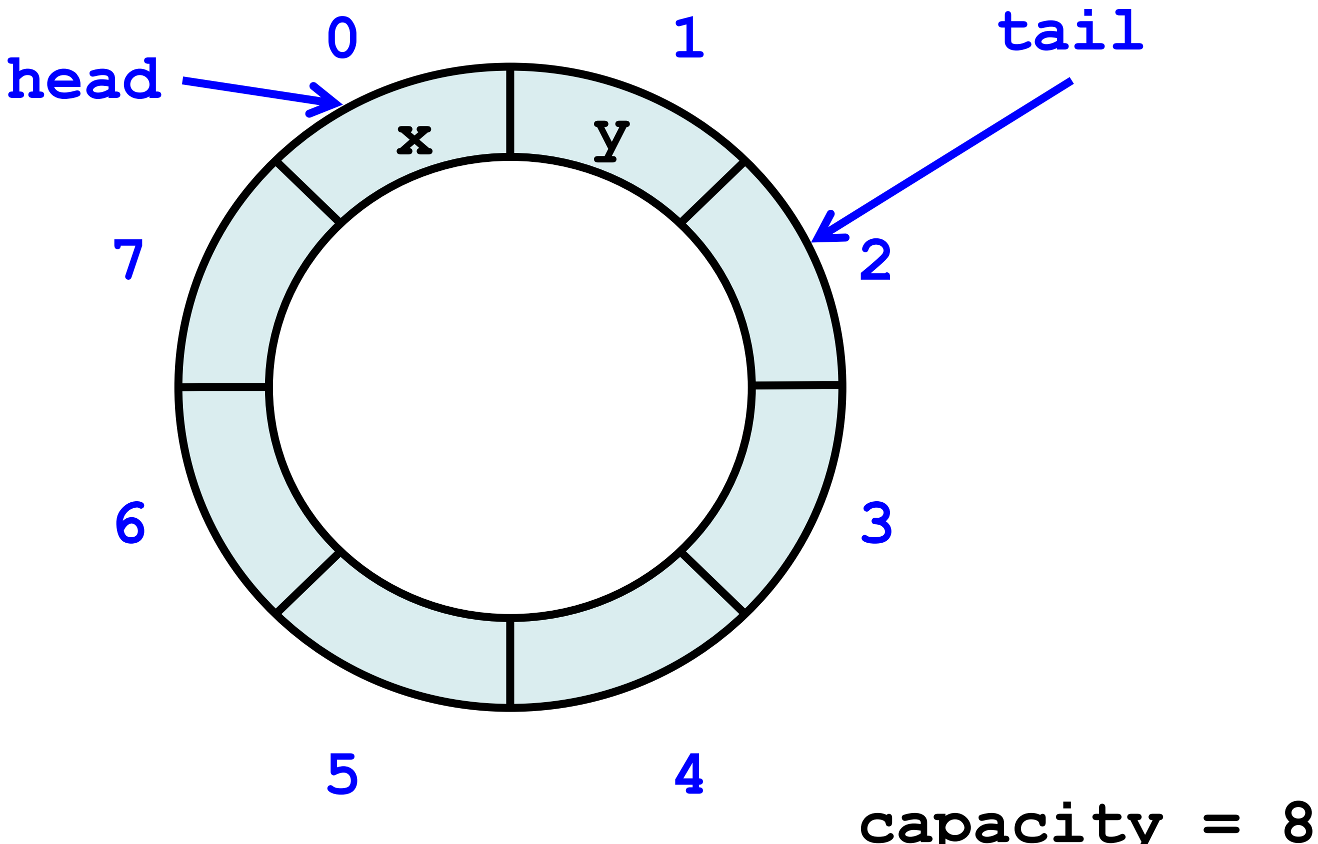


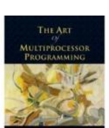
Now consider the following implementation

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

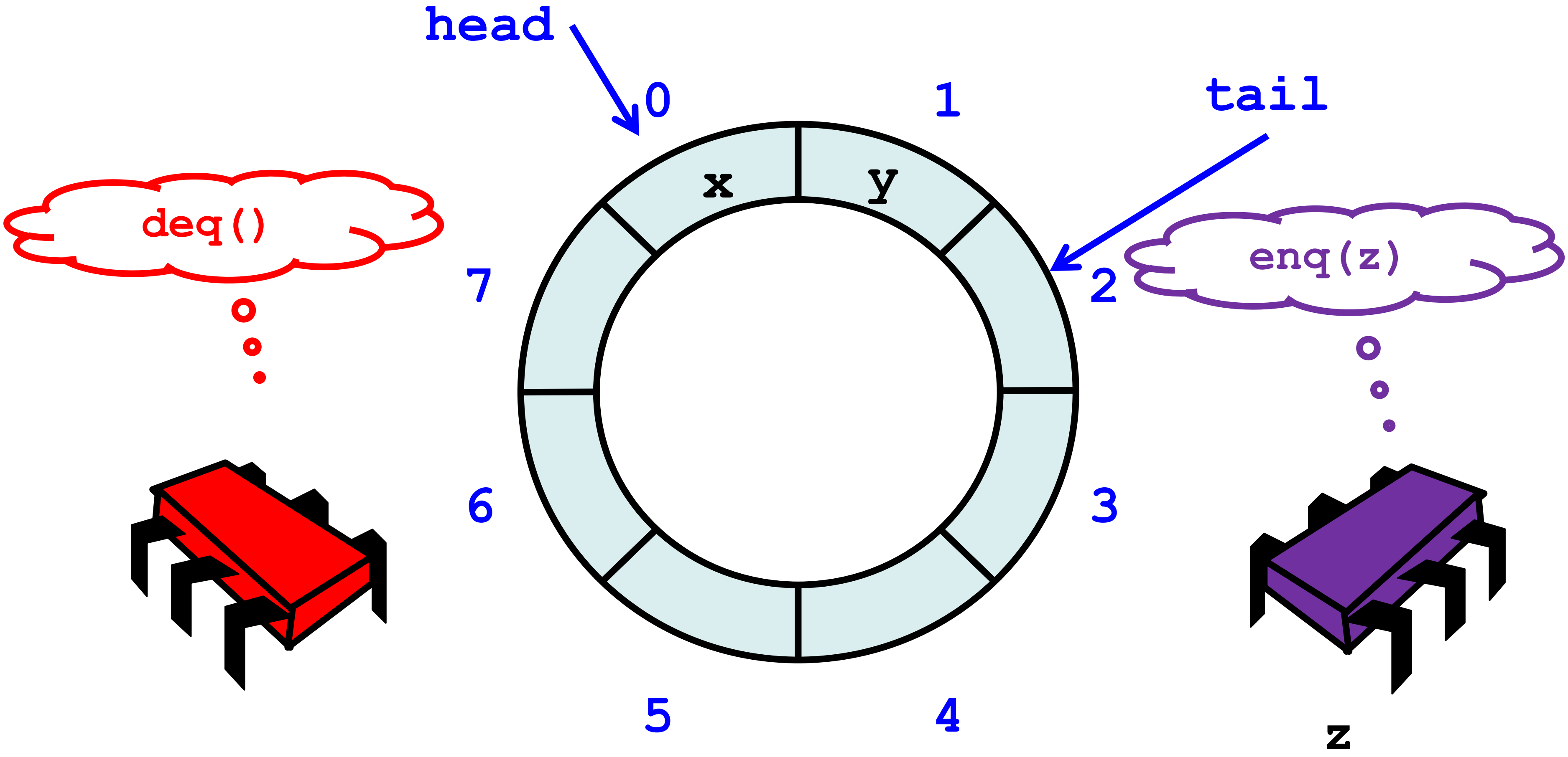


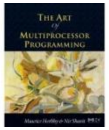
Wait-free 2-Thread Queue



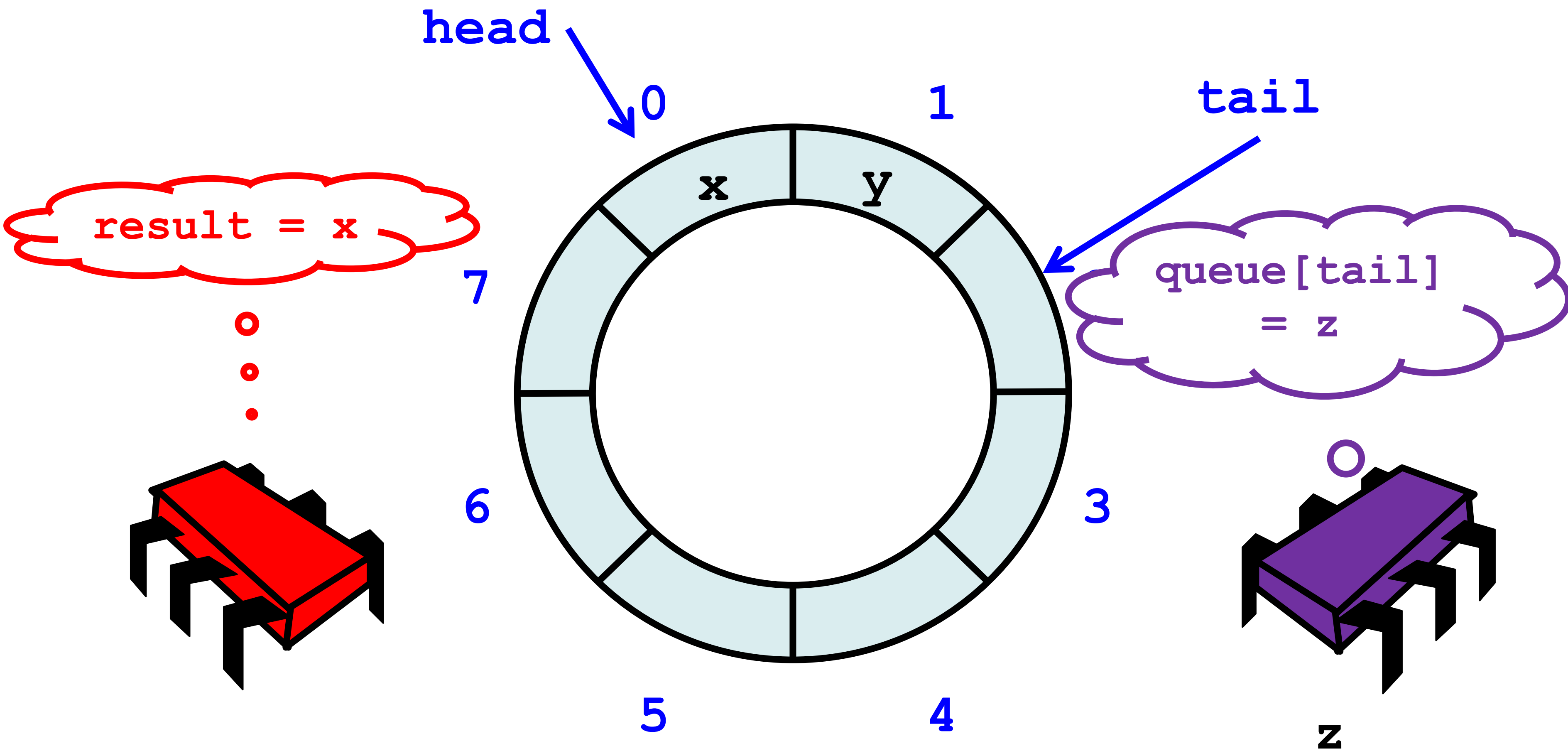


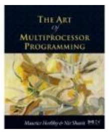
Wait-free 2-Thread Queue



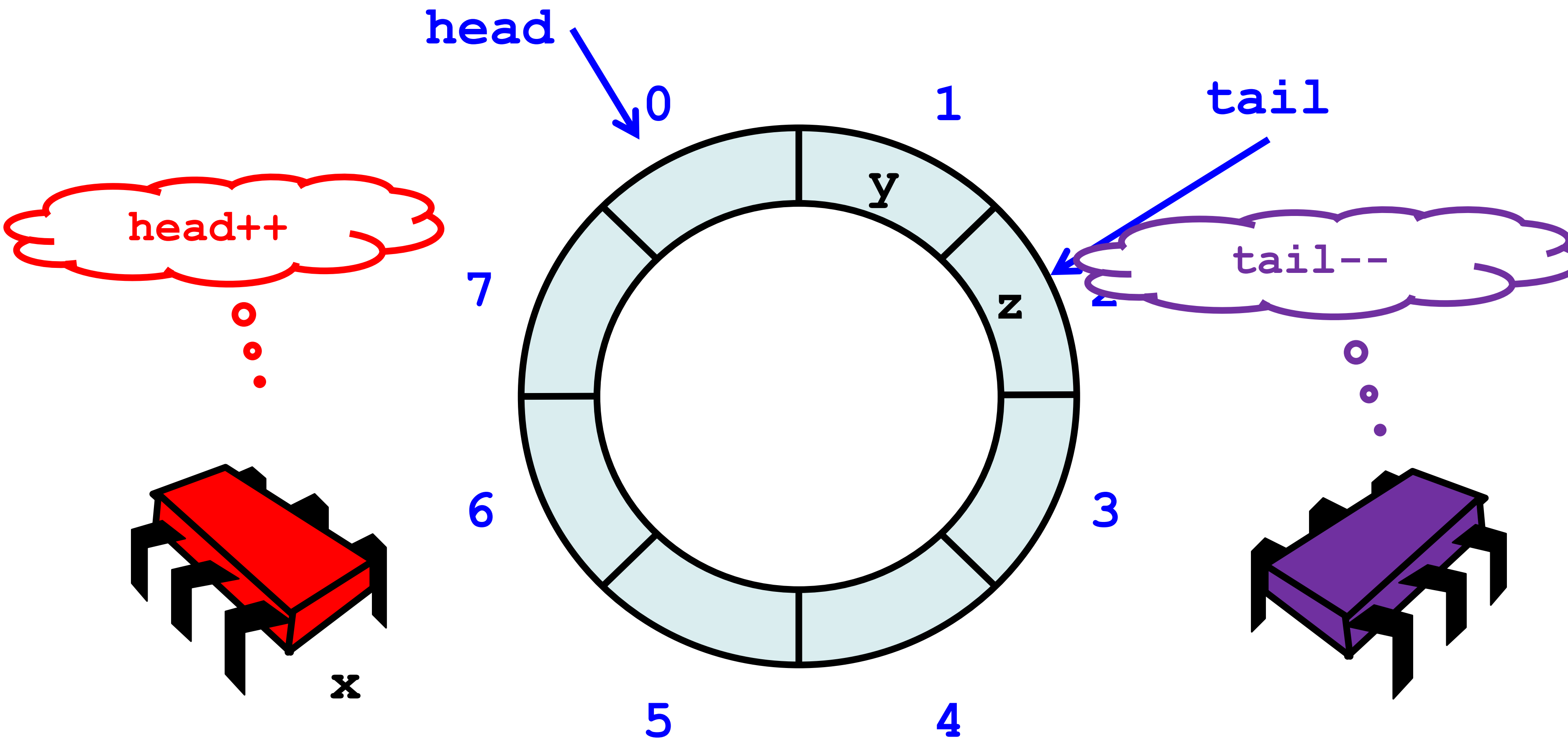


Wait-free 2-Thread Queue

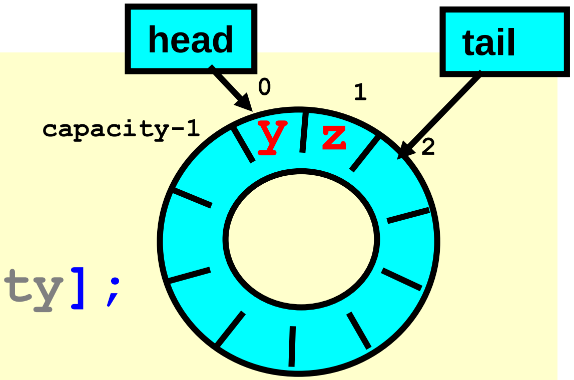




Wait-free 2-Thread Queue



Wait-free 2-Thread Queue



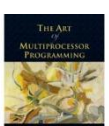

```
public class WaitFreeQueue {

  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

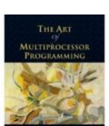
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyExc
    T x = items[hea
    head++;
    retur
  } finall
    lock.un
  }
}
```

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



What *is* a Concurrent Queue?

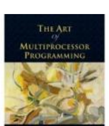
- Need a way to specify a concurrent queue object
- Need a way to prove that an algorithm implements the object's specification
- Lets talk about object specifications …



Correctness and Progress

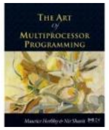
- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness



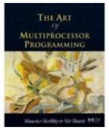
Sequential Objects

- 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



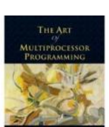
Sequential Specifications

- 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

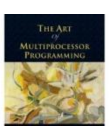


Pre and PostConditions for Dequeue

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

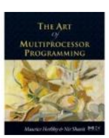
Why Sequential Specifications Totally Rock

- 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

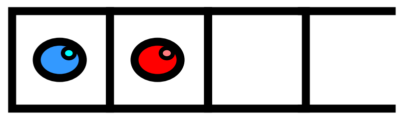


What About Concurrent Specifications ?

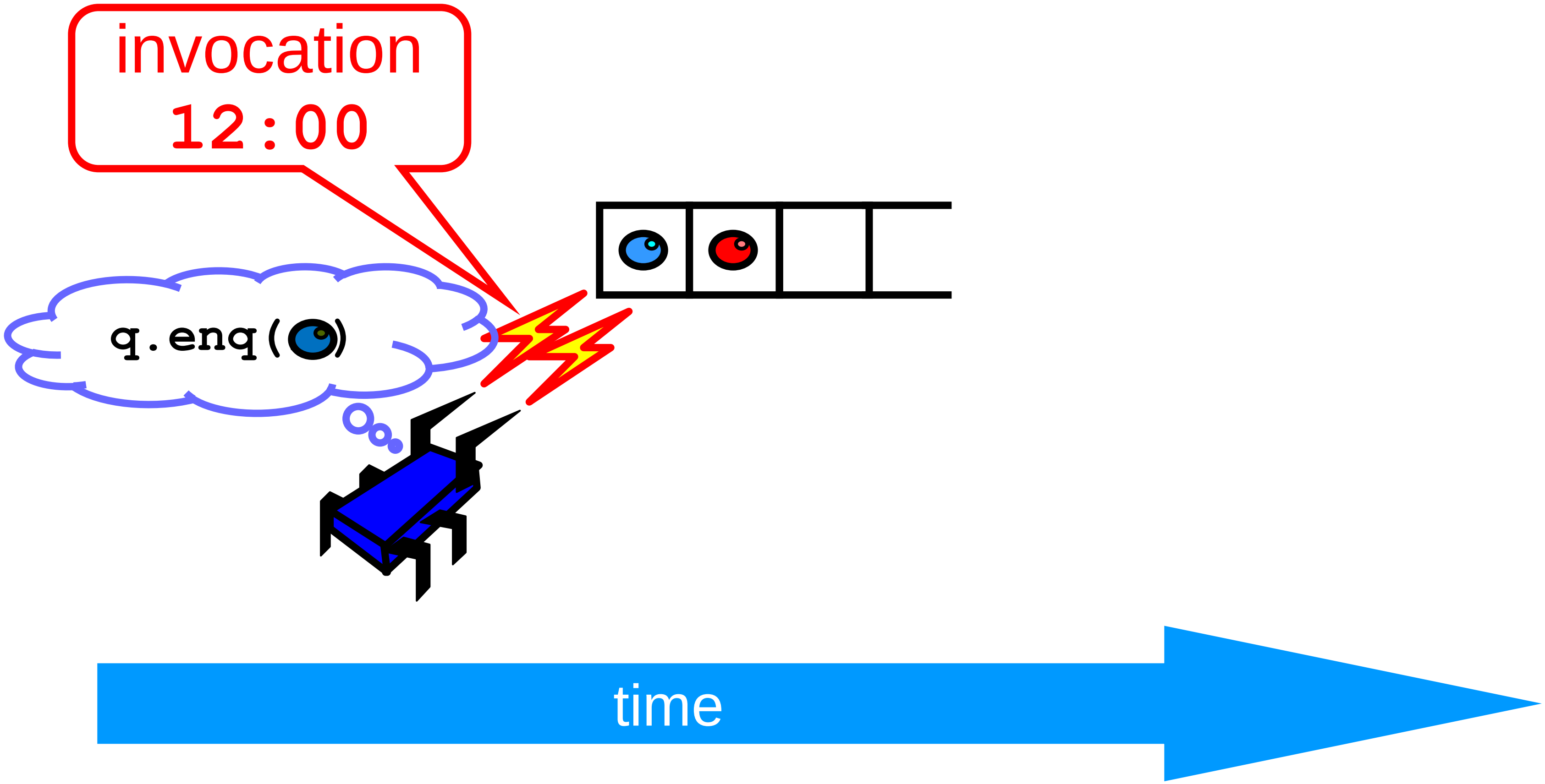
- Methods?
- Documentation?
- Adding new methods?

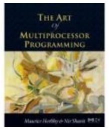


Methods Take Time

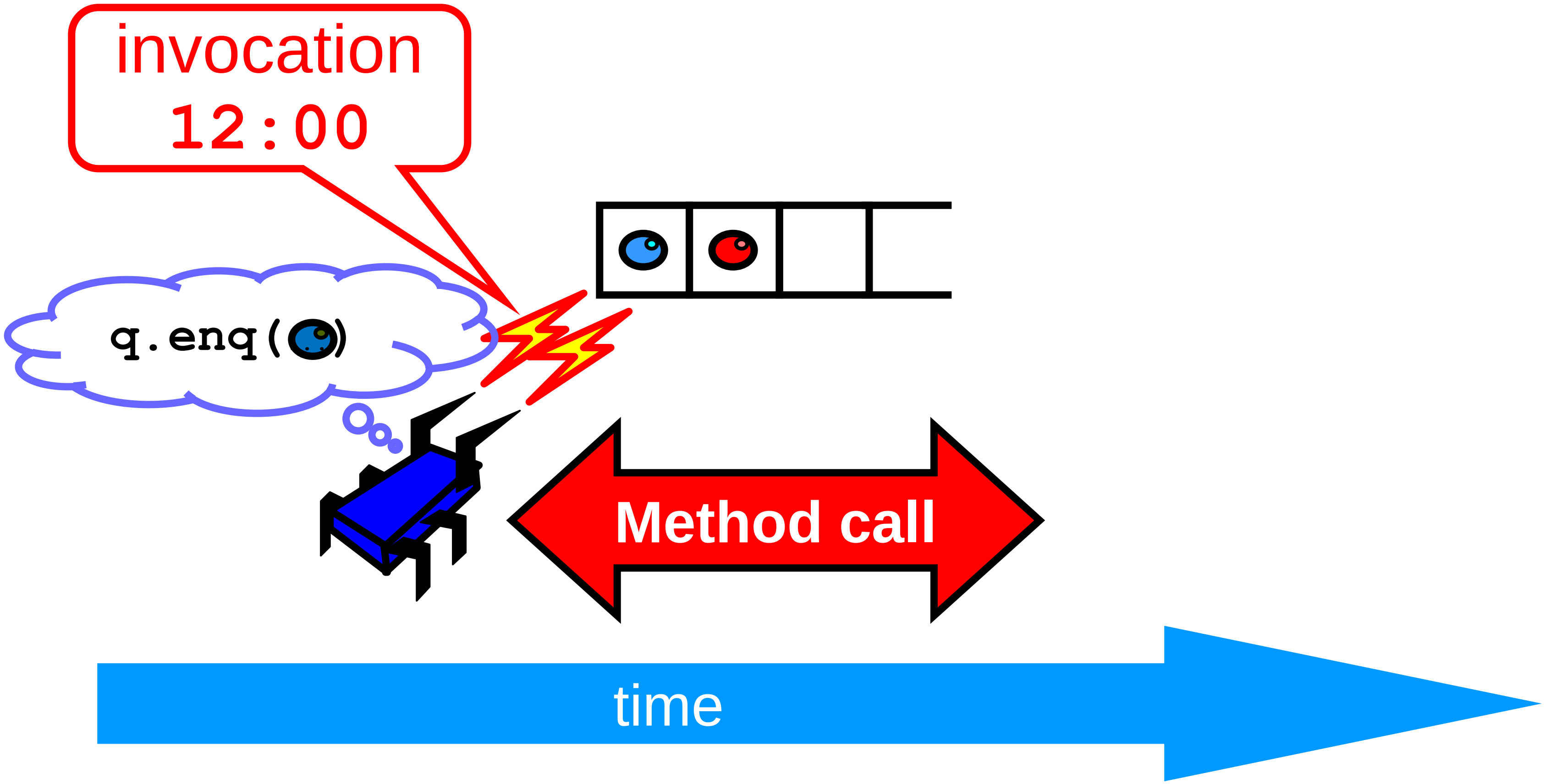


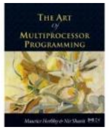
Methods Take Time



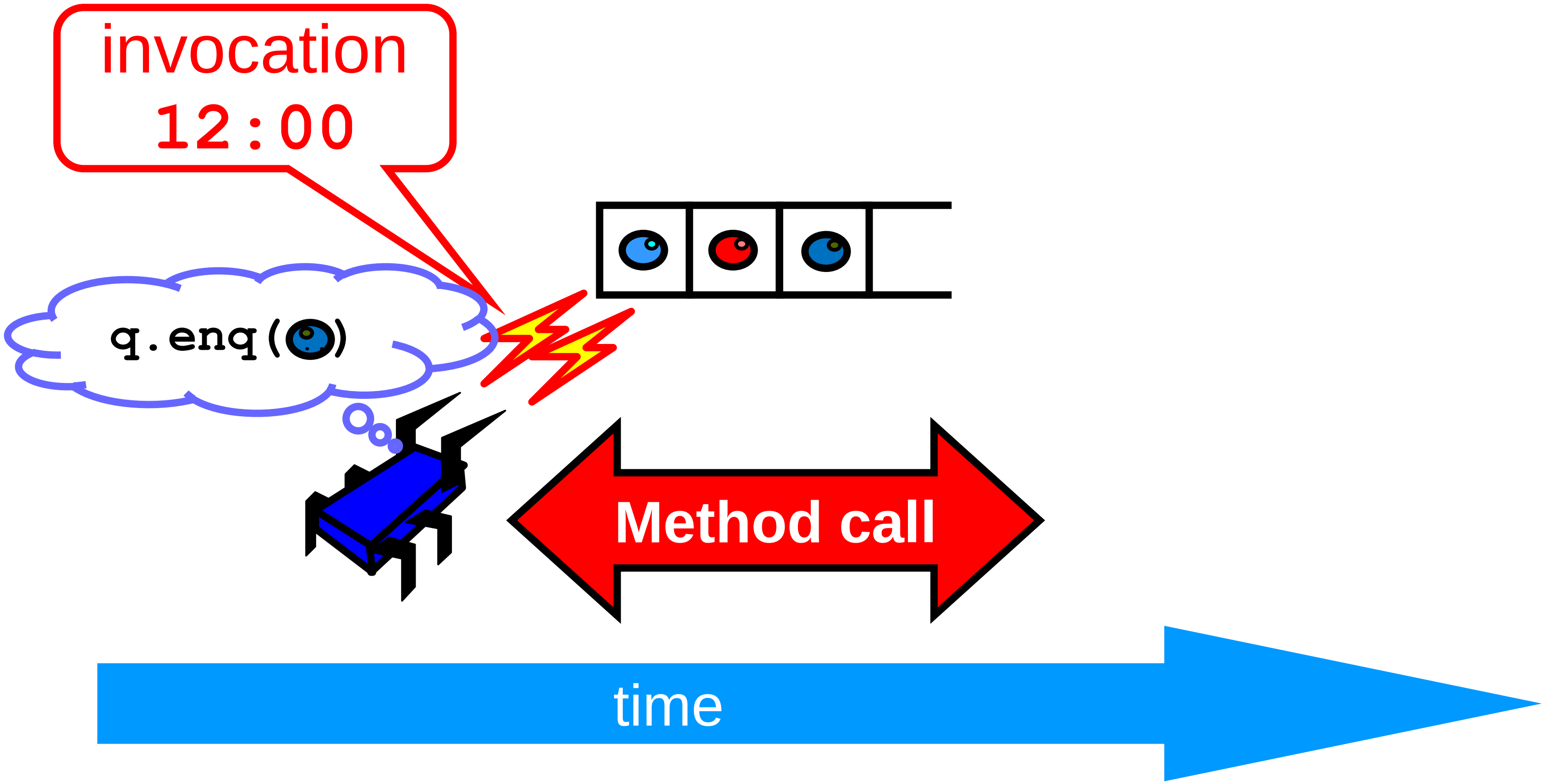


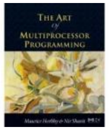
Methods Take Time



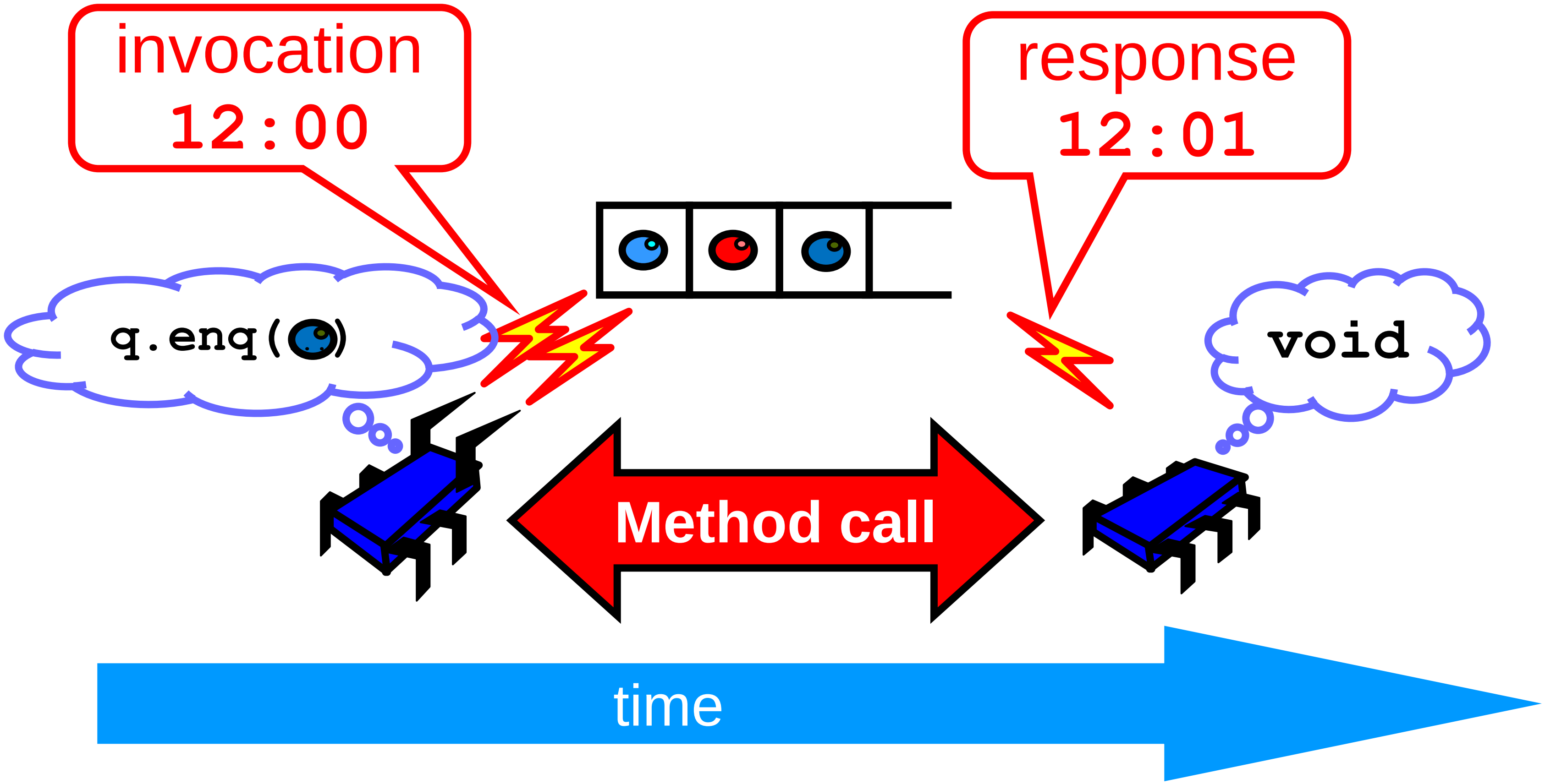


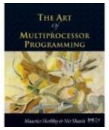
Methods Take Time





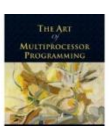
Methods Take Time



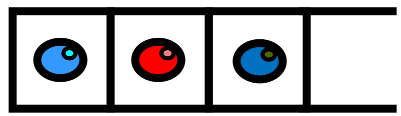


Sequential vs Concurrent

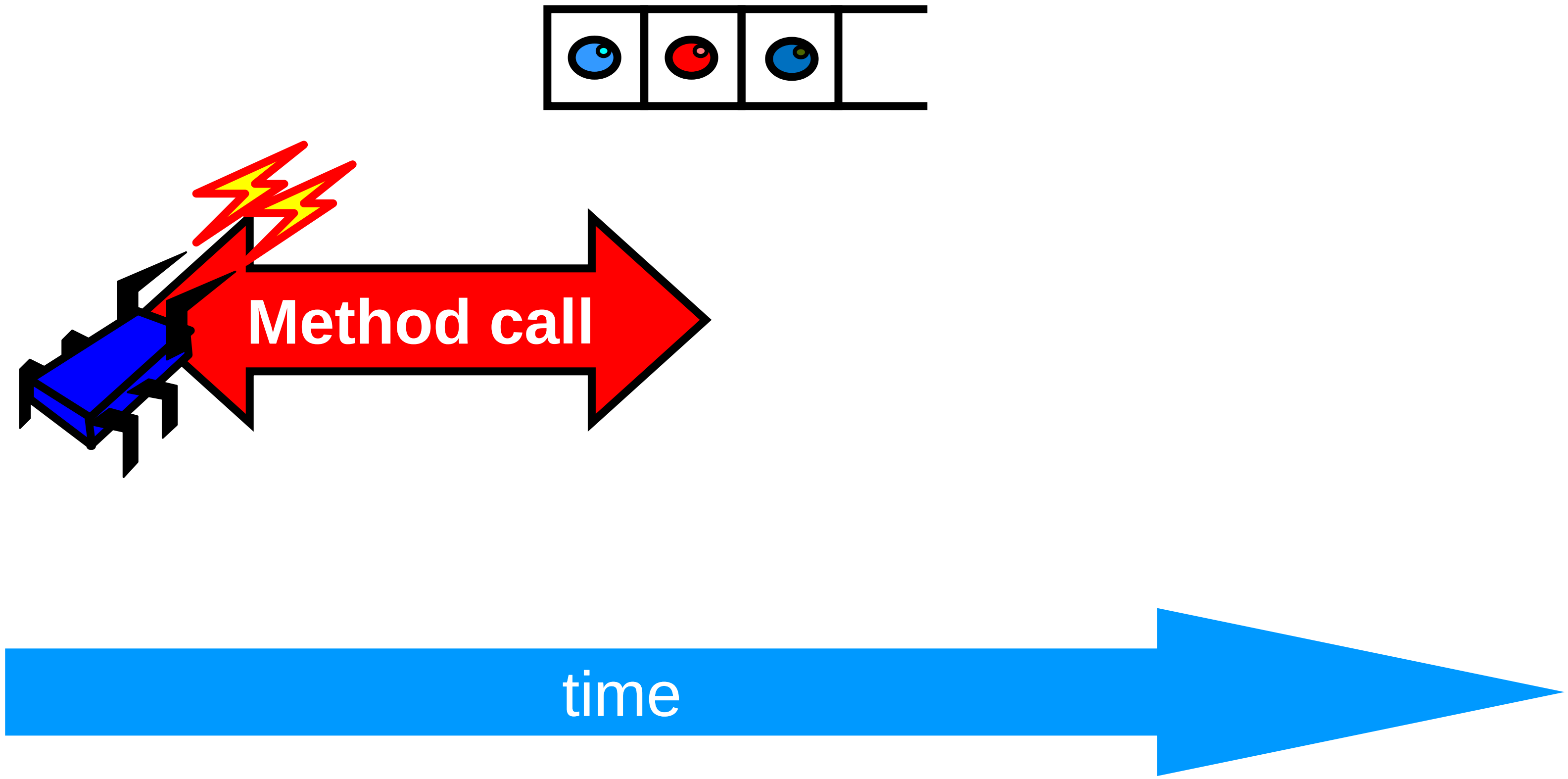
- Sequential
  - – Methods take time? Who knew?
- Concurrent
  - – Method call is not an event
  - – Method call is an interval.



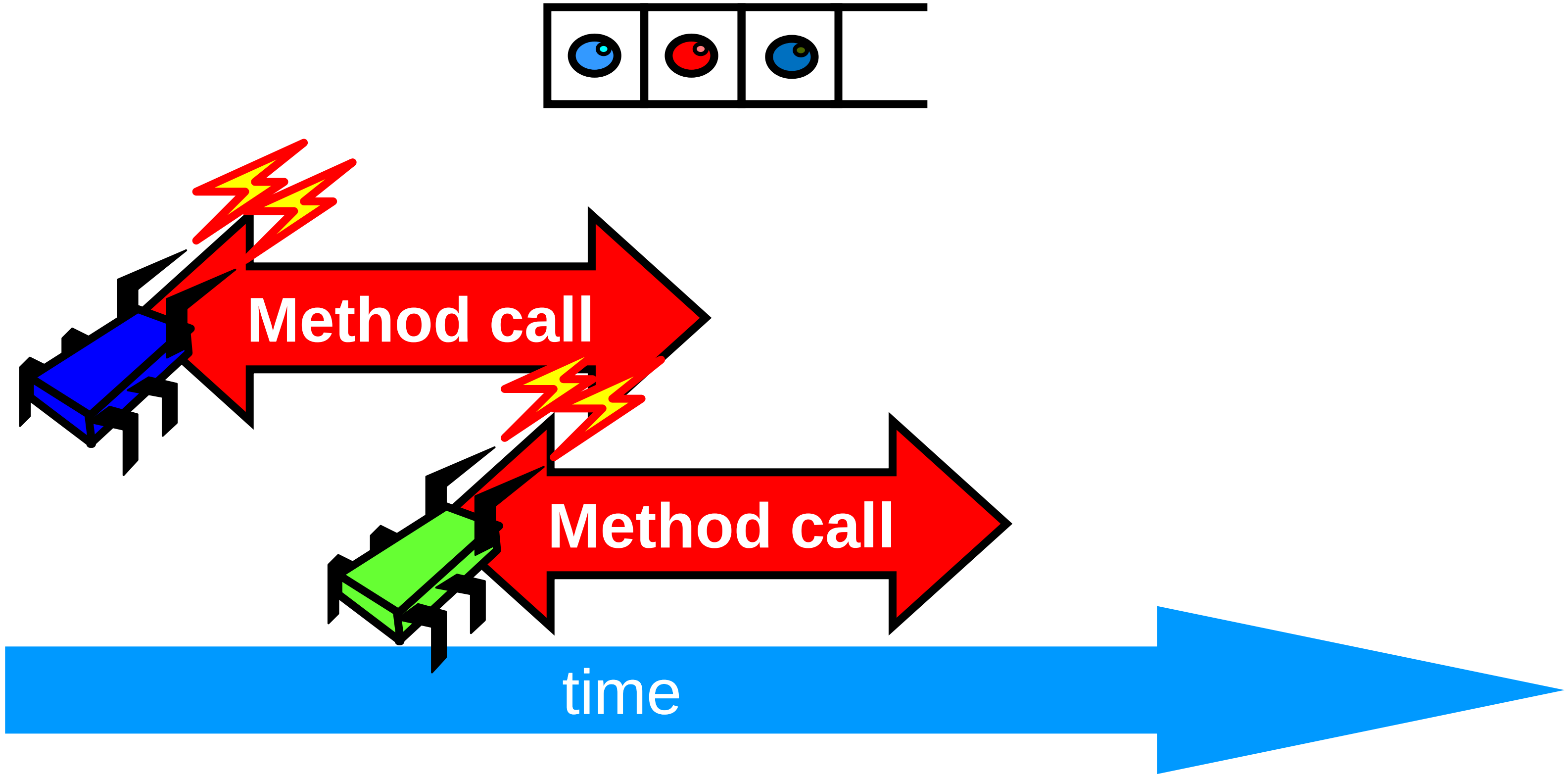
Concurrent Methods Take Overlapping Time



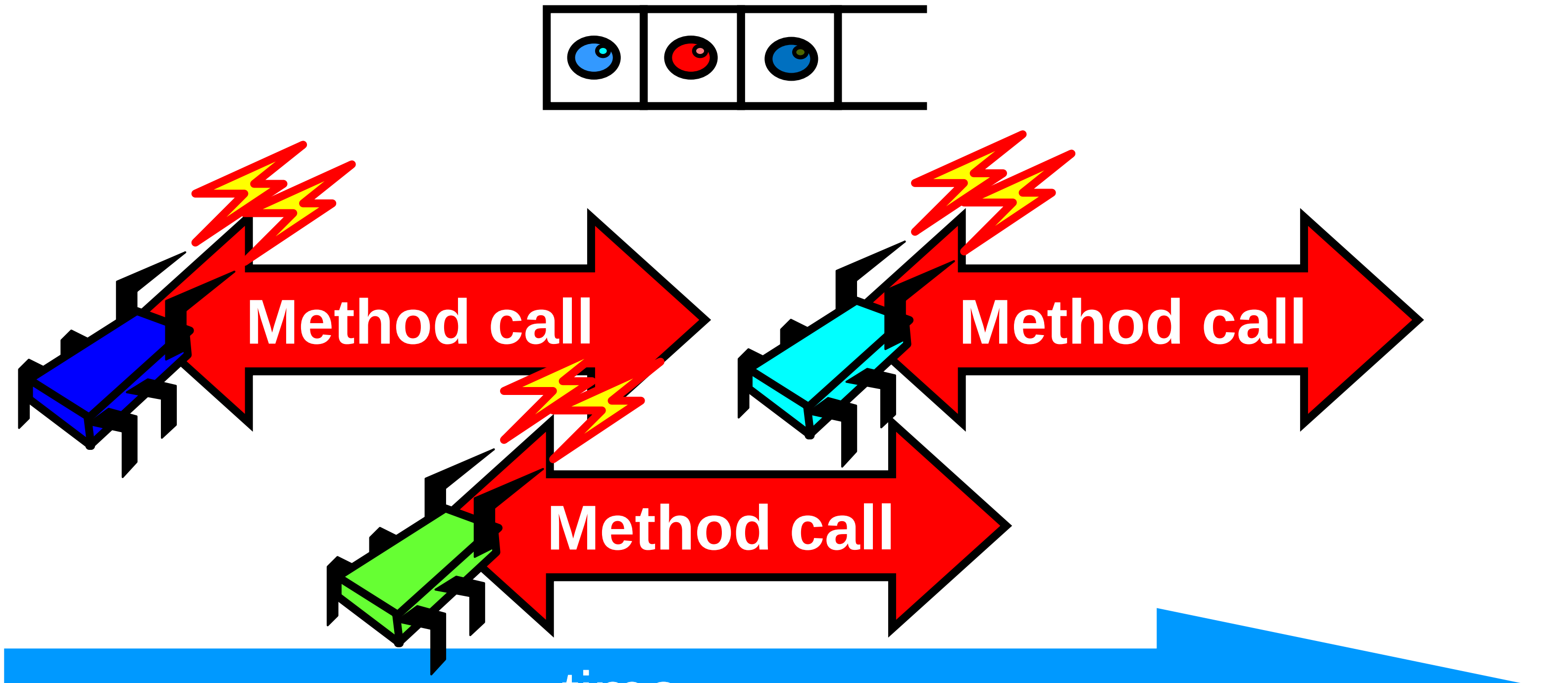
Concurrent Methods Take Overlapping Time

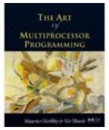


Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



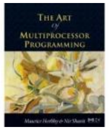


Sequential vs Concurrent

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

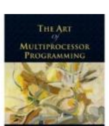
Sequential vs Concurrent

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



Sequential vs Concurrent

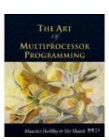
- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else



Sequential vs Concurrent

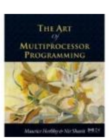
- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Panic!



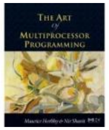
The Big Question

- 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…

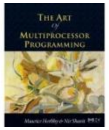
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Intuitively…

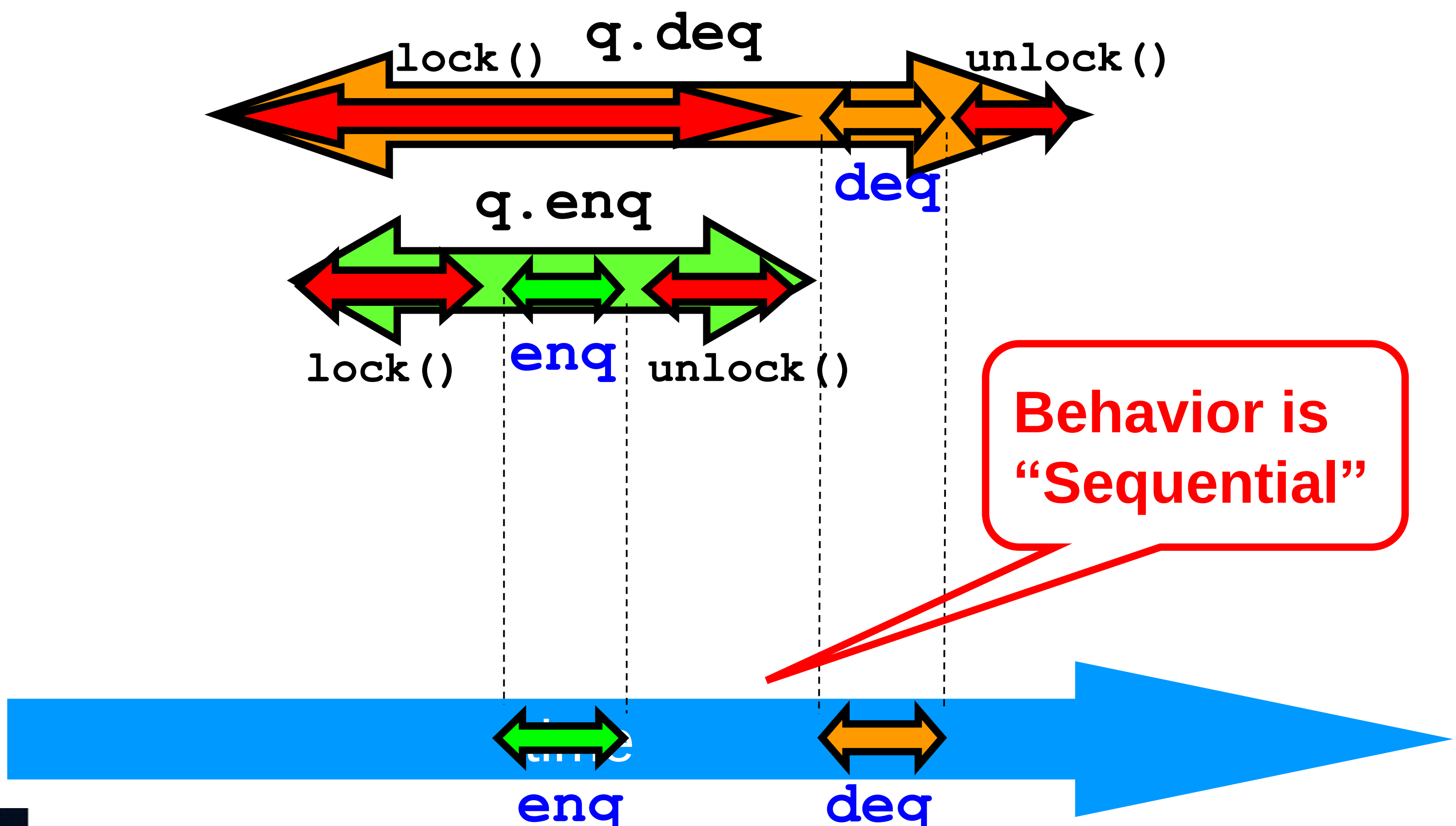
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

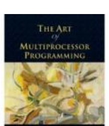
All queue modifications are mutually exclusive



Intuitively

Lets capture the idea of describing the concurrent via the sequential



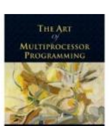


Linearizability

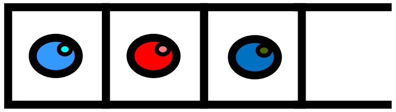
- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is
  - **Linearizable™**

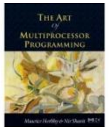
Is it really about the object?

- 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

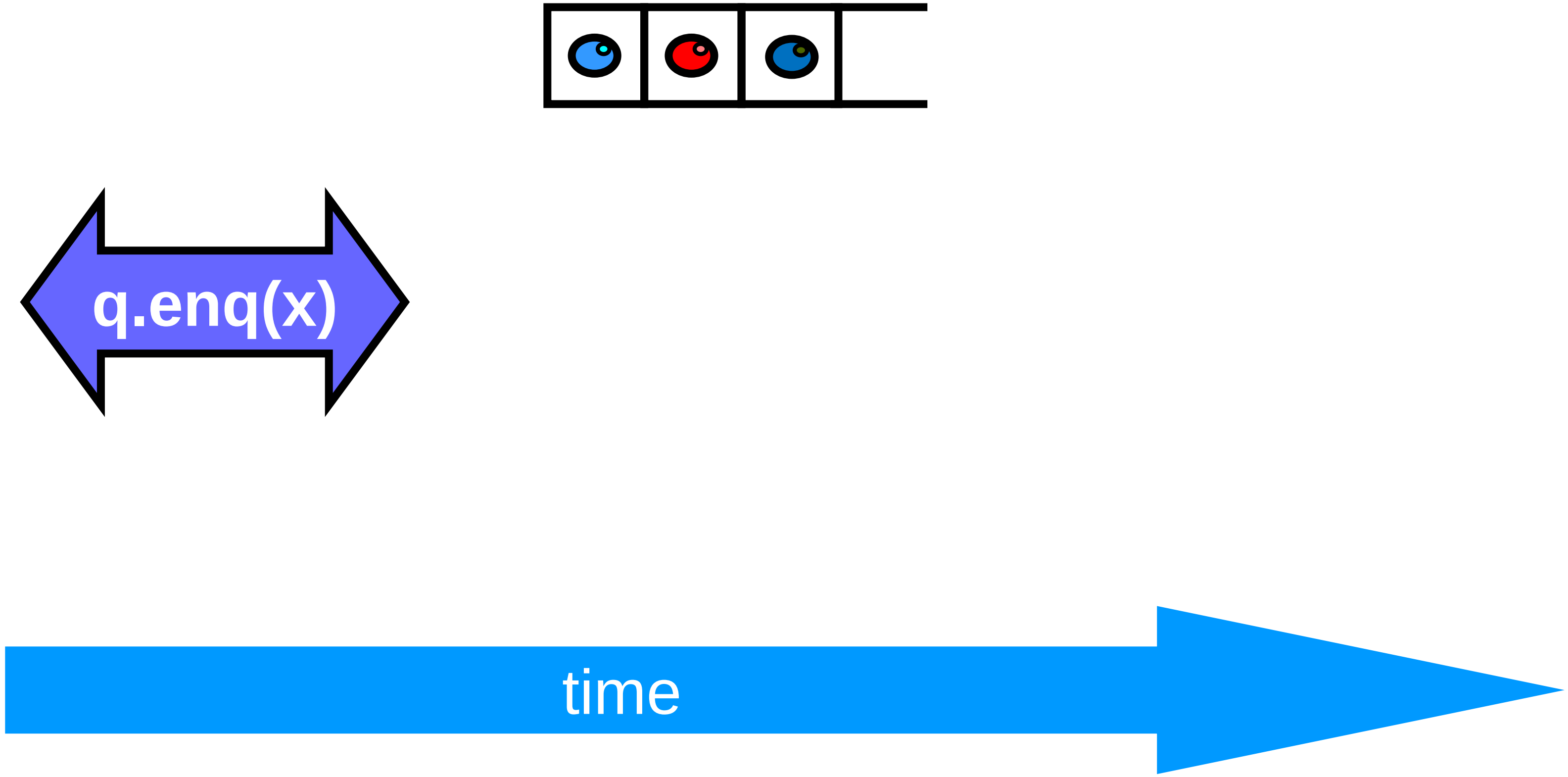


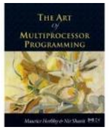
Example



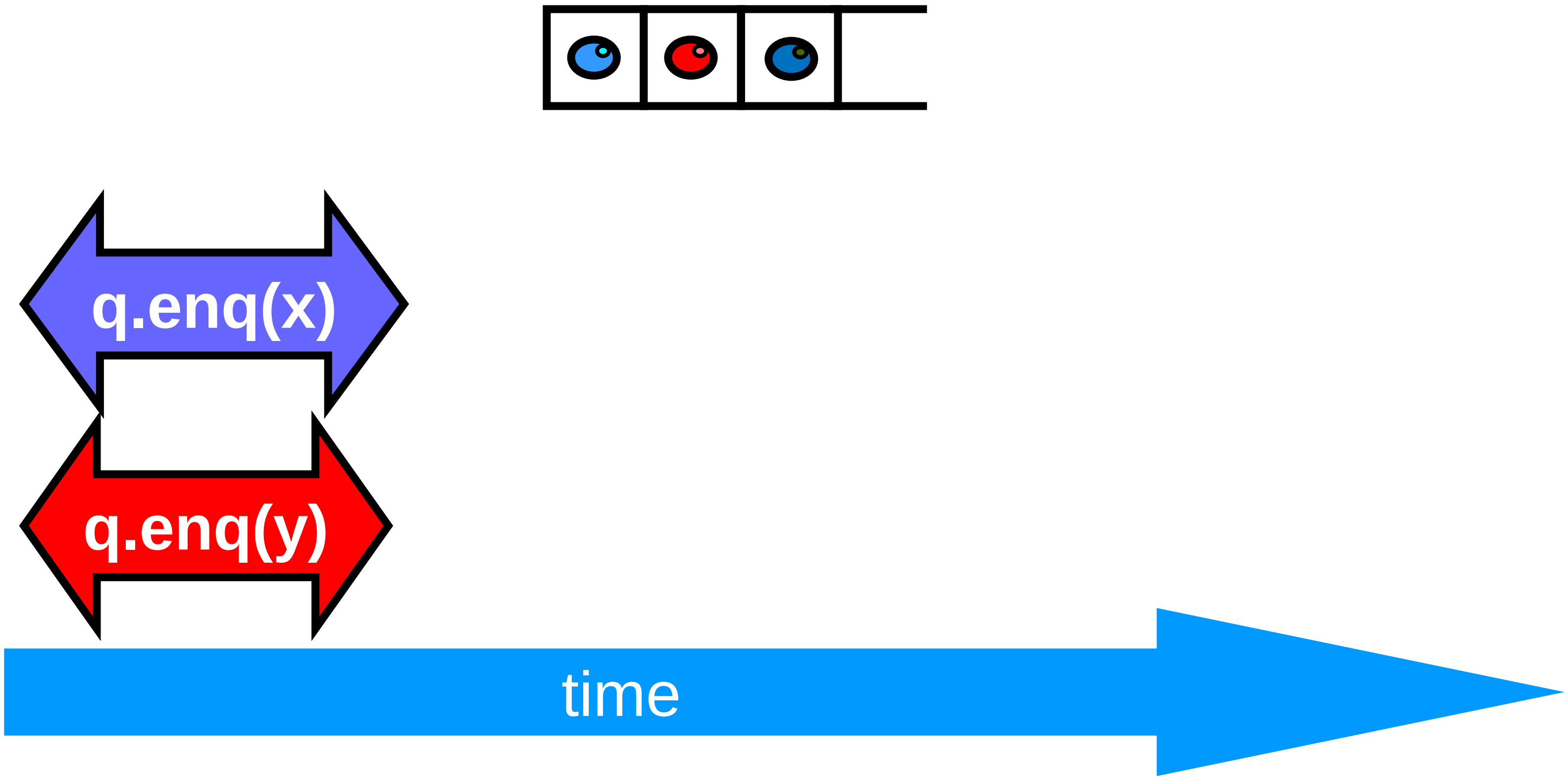


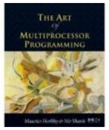
Example



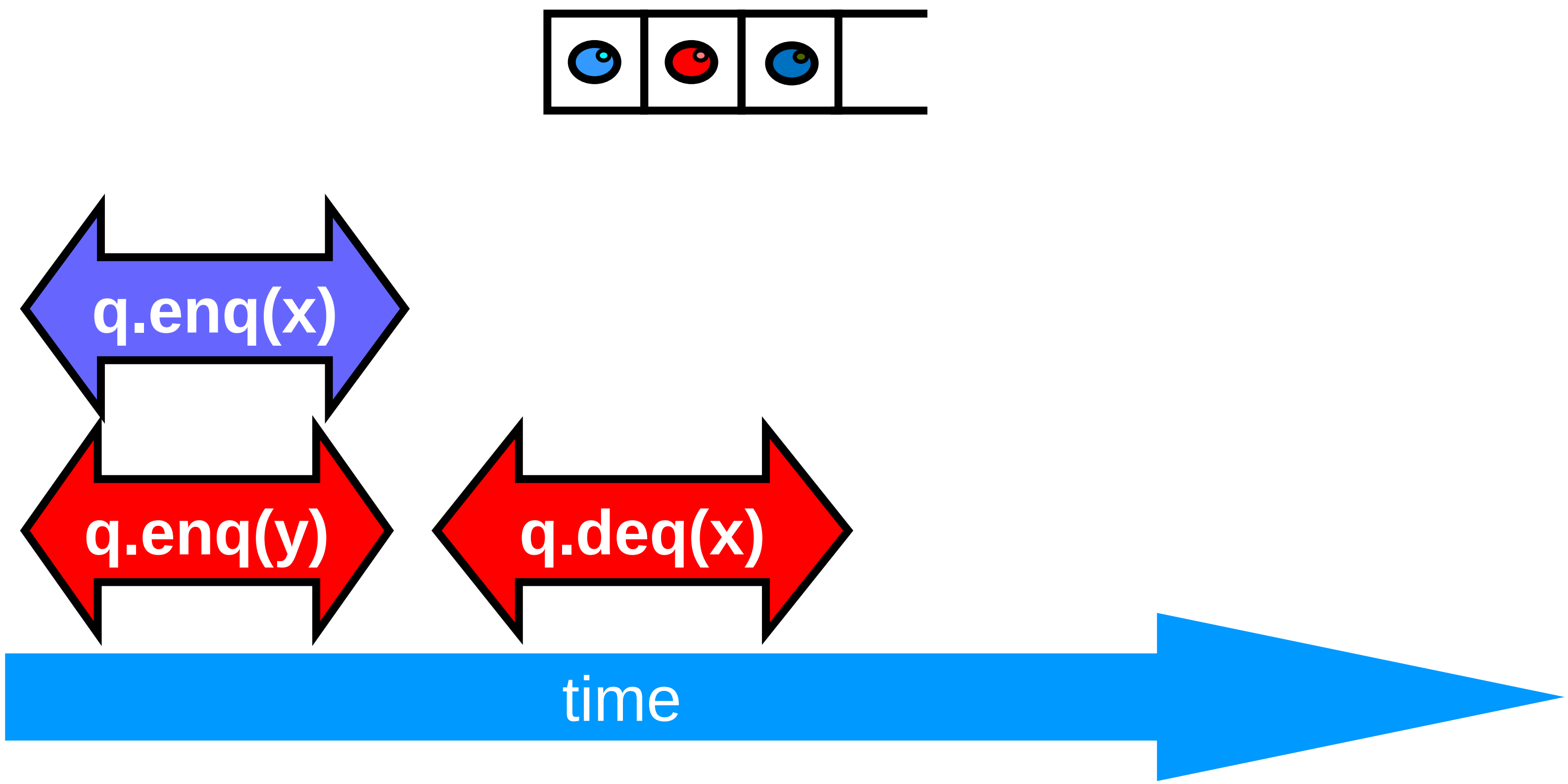


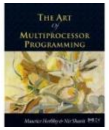
Example



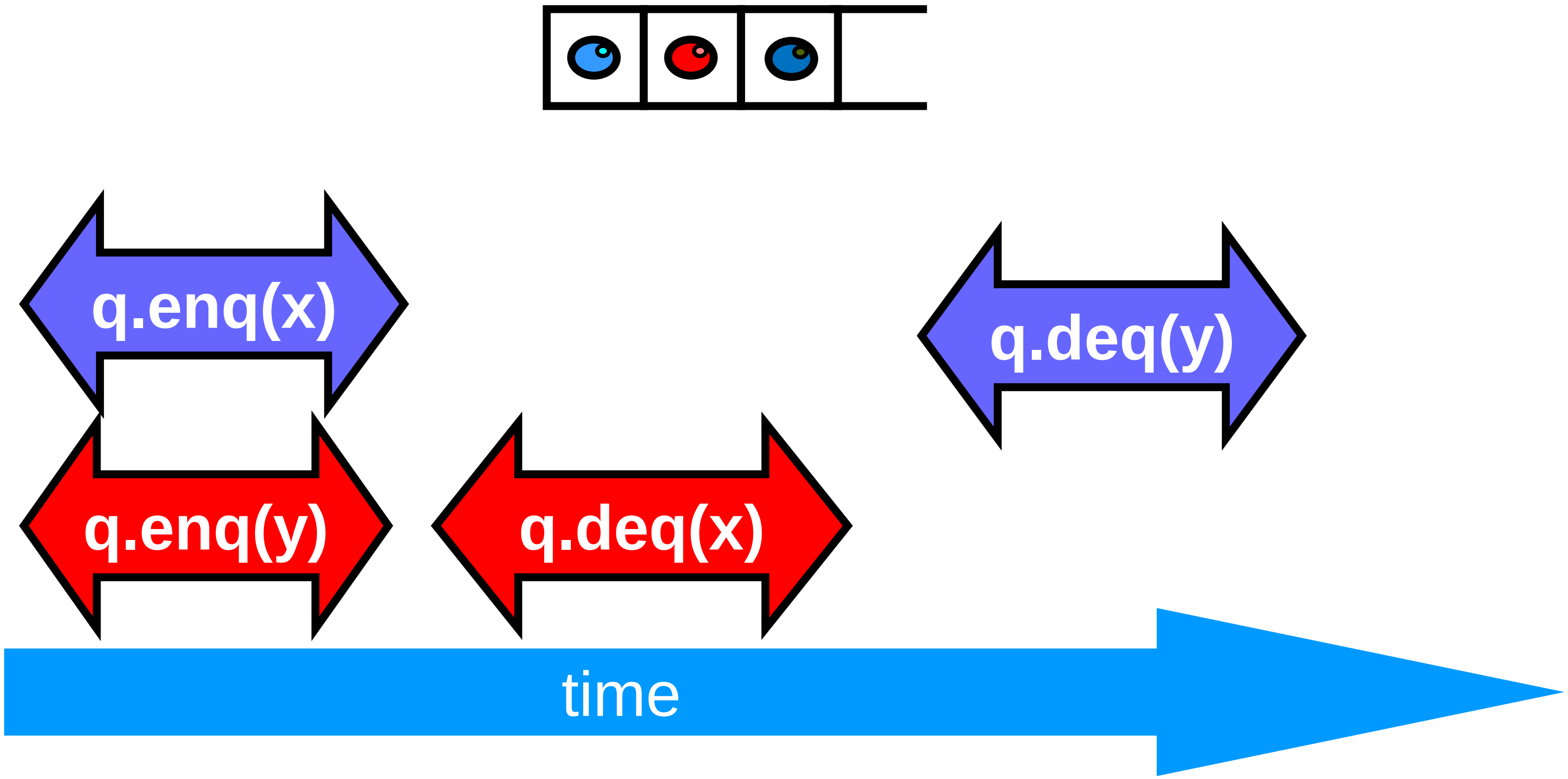


Example

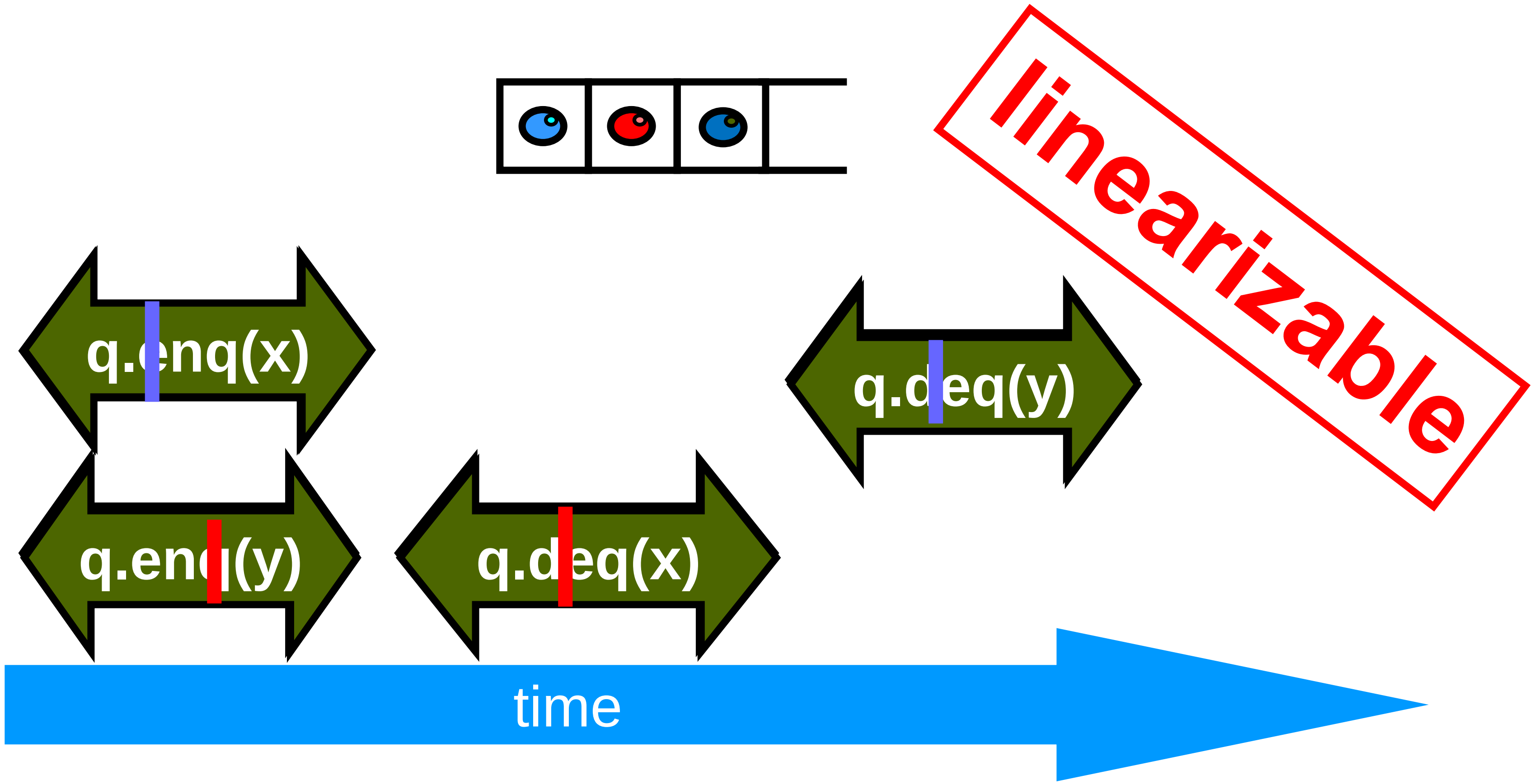


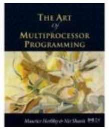


Example

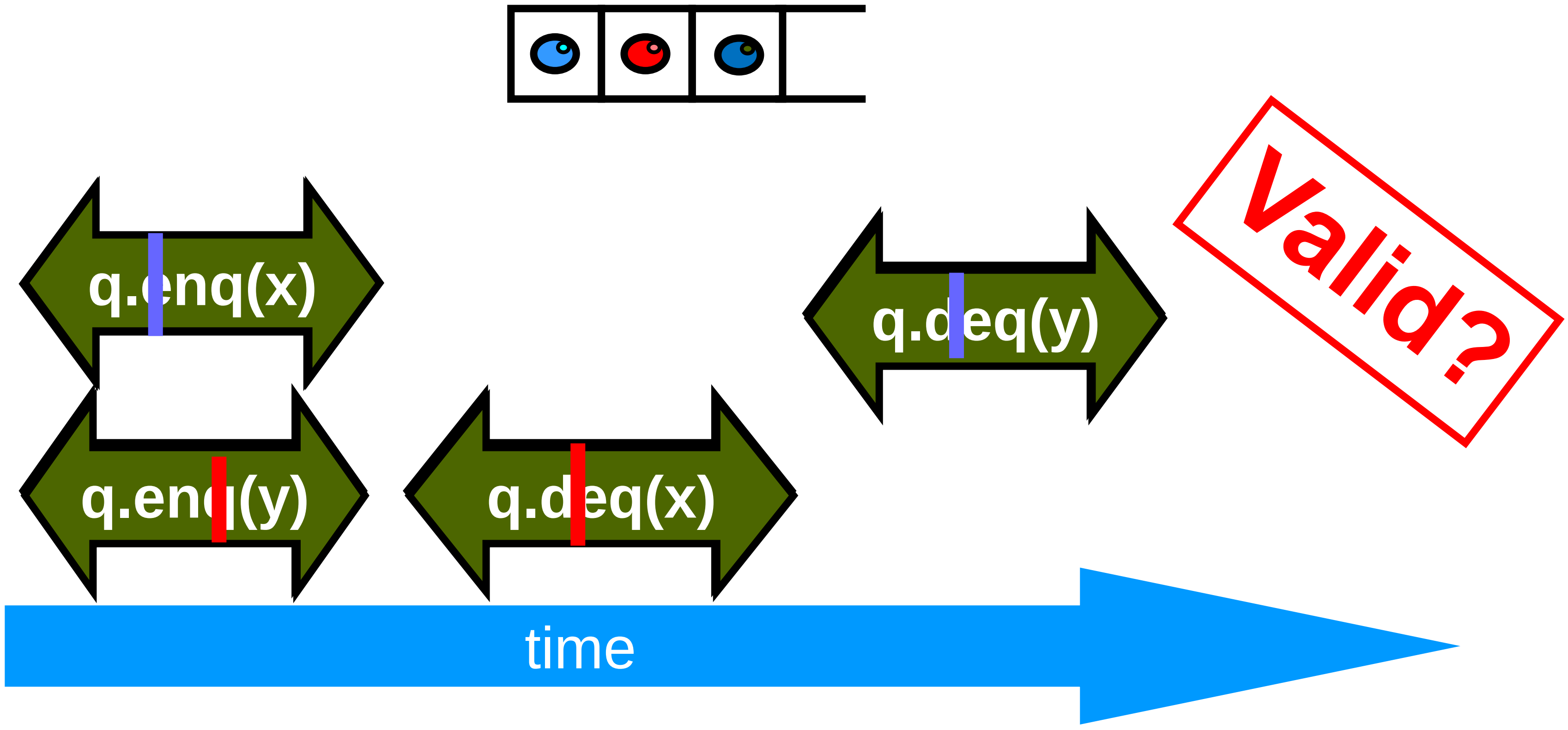


Example

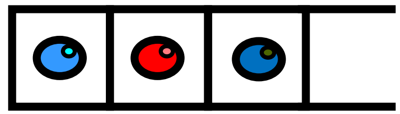




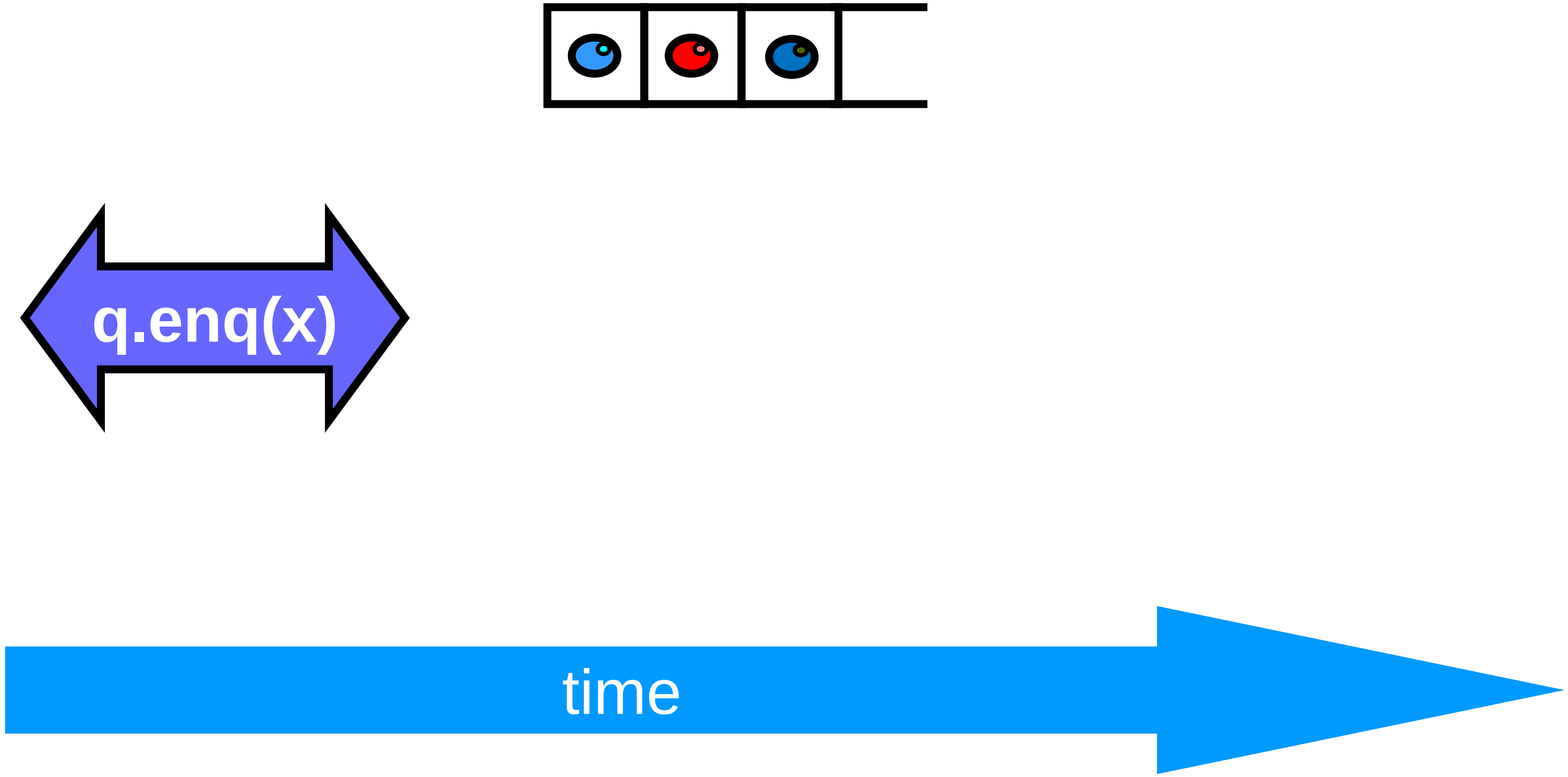
Example

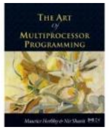


Example

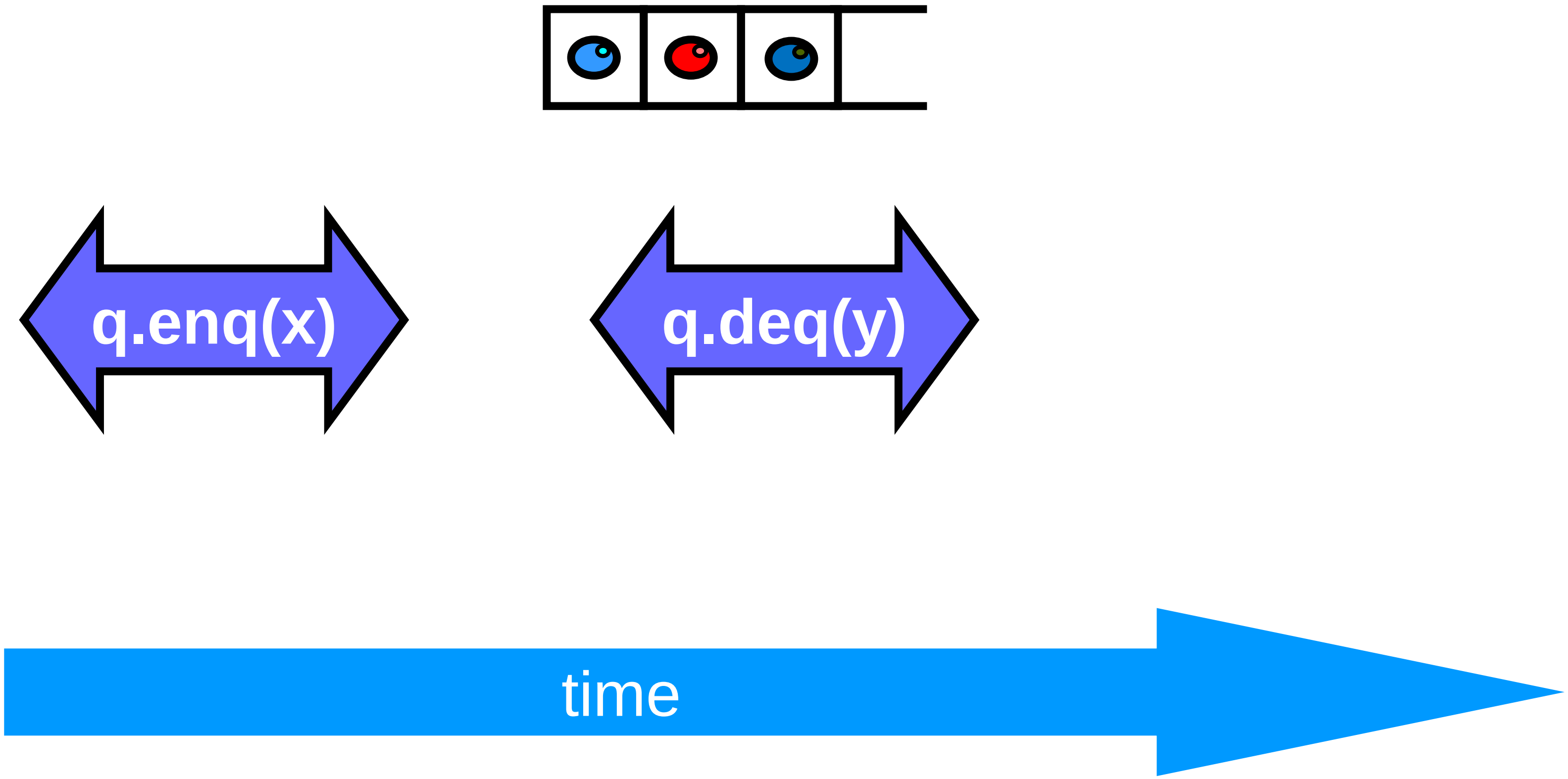


Example

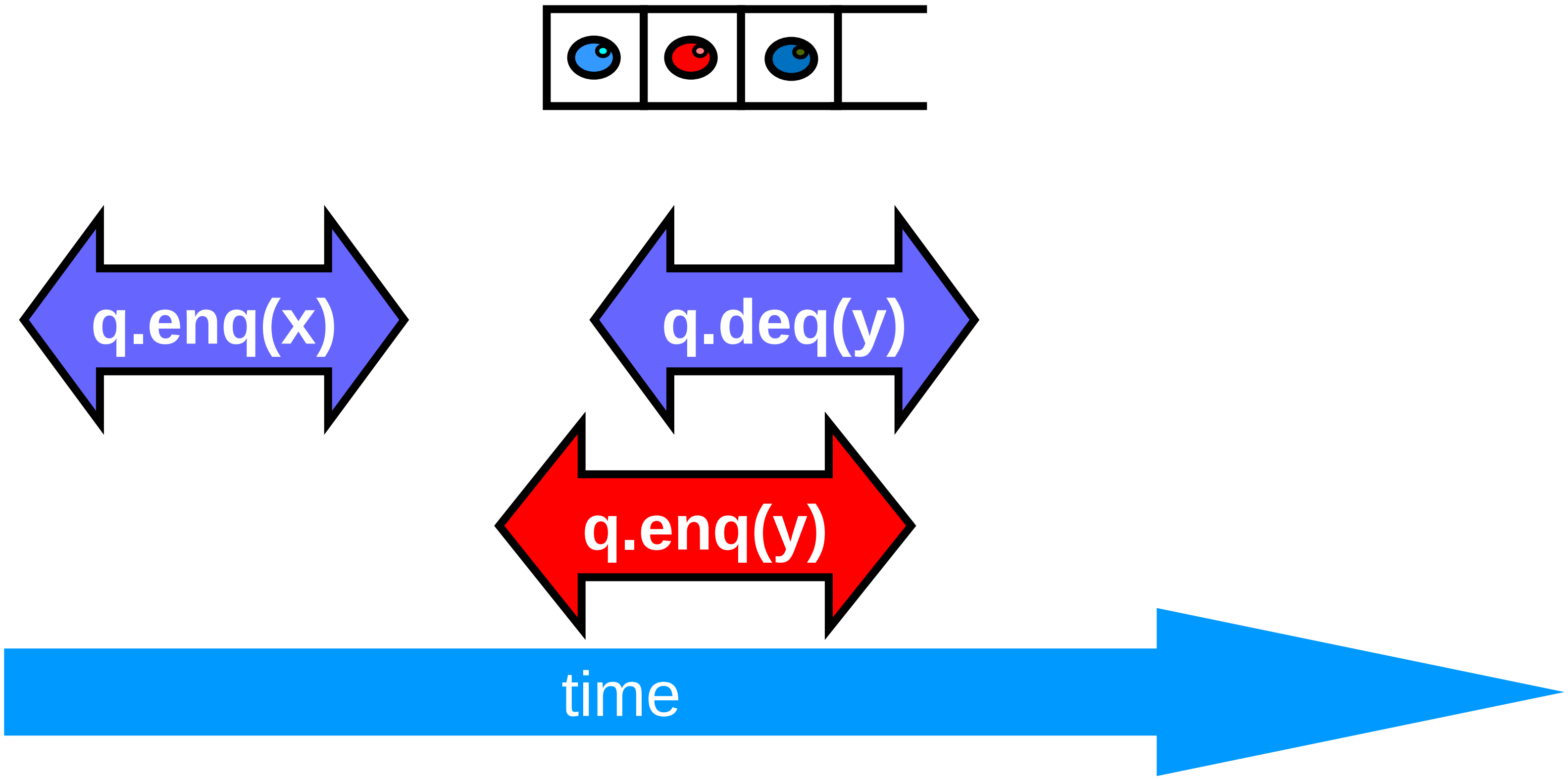




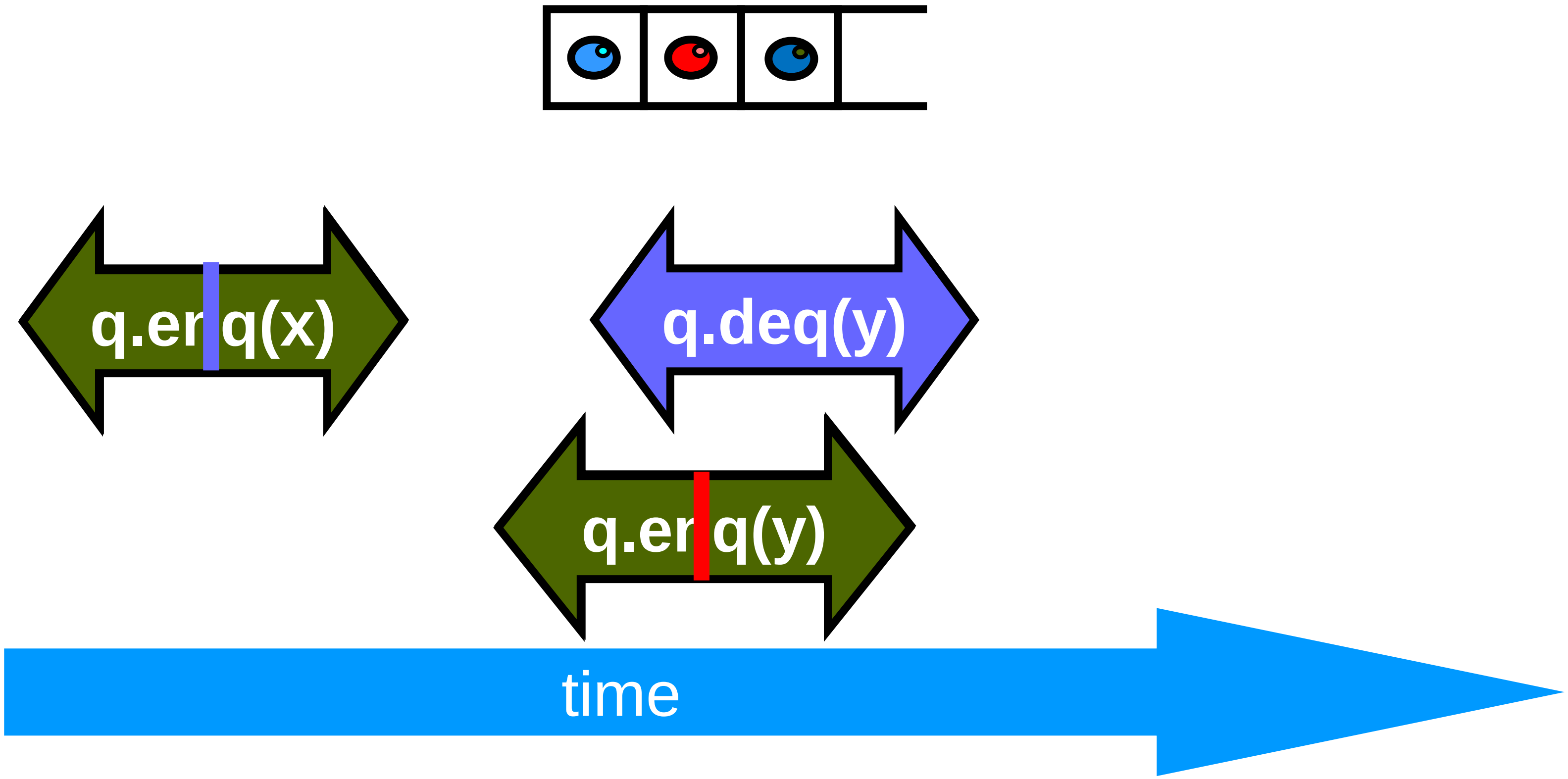
Example



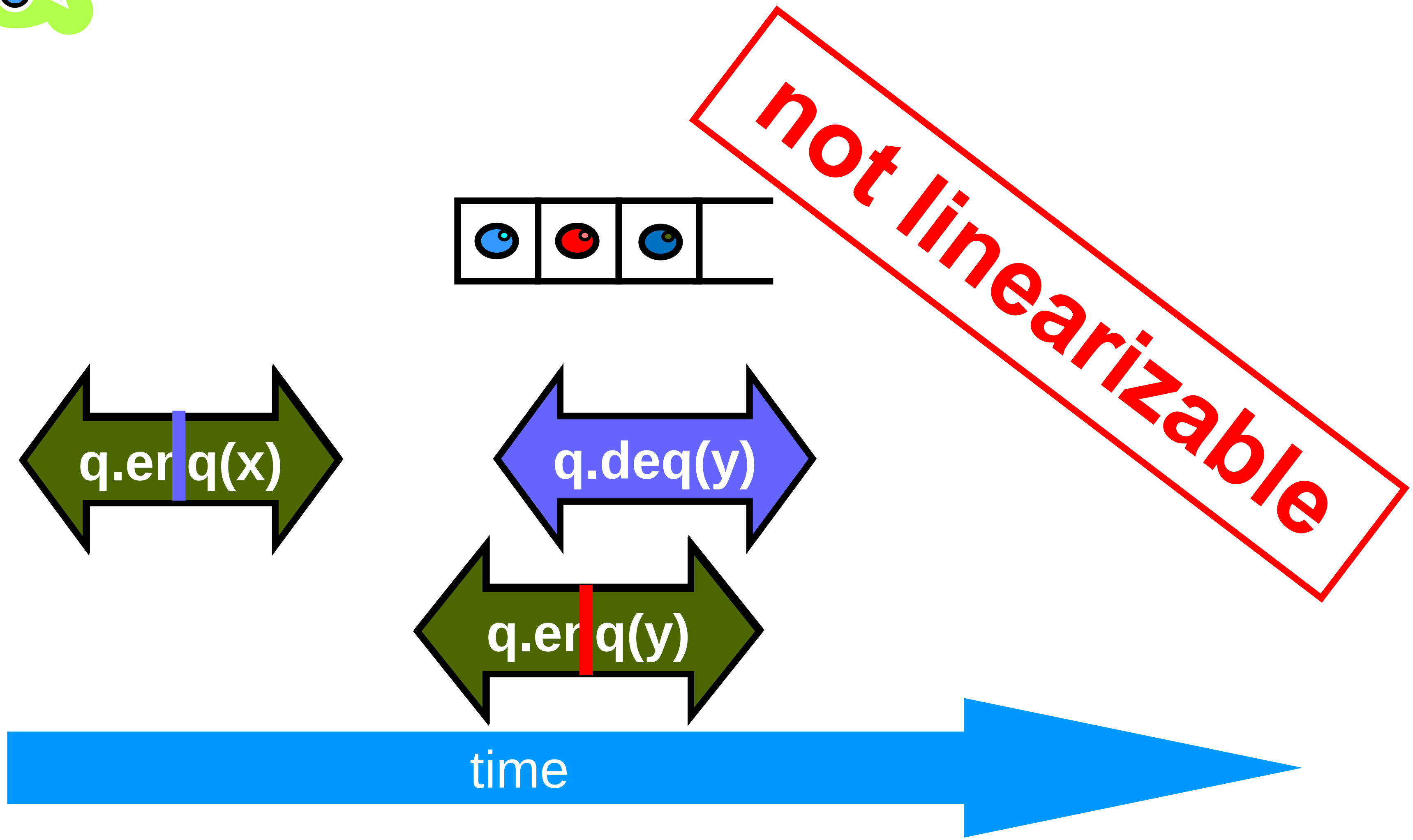
Example



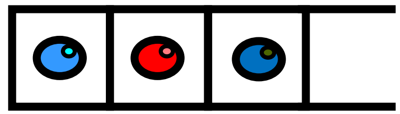
Example



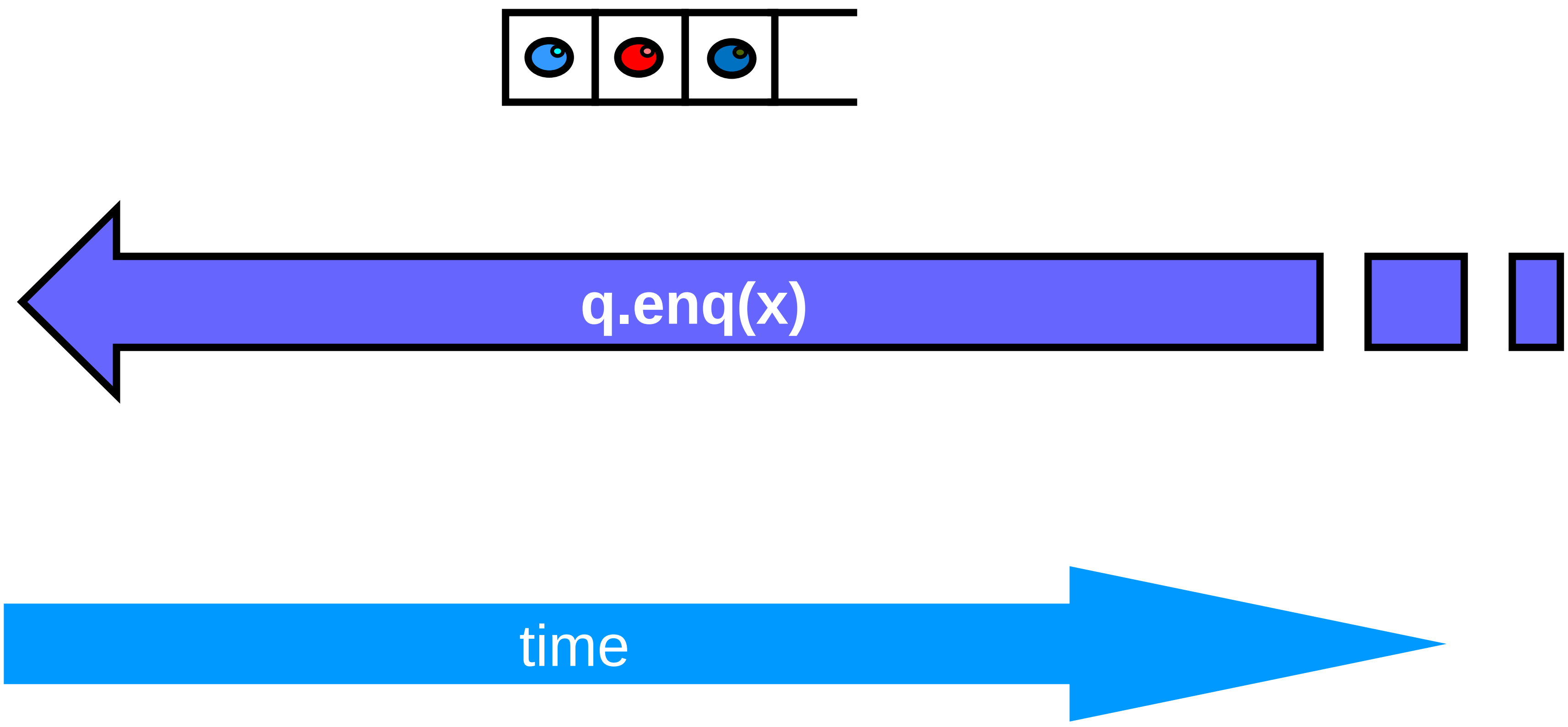
Example

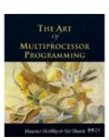


Example

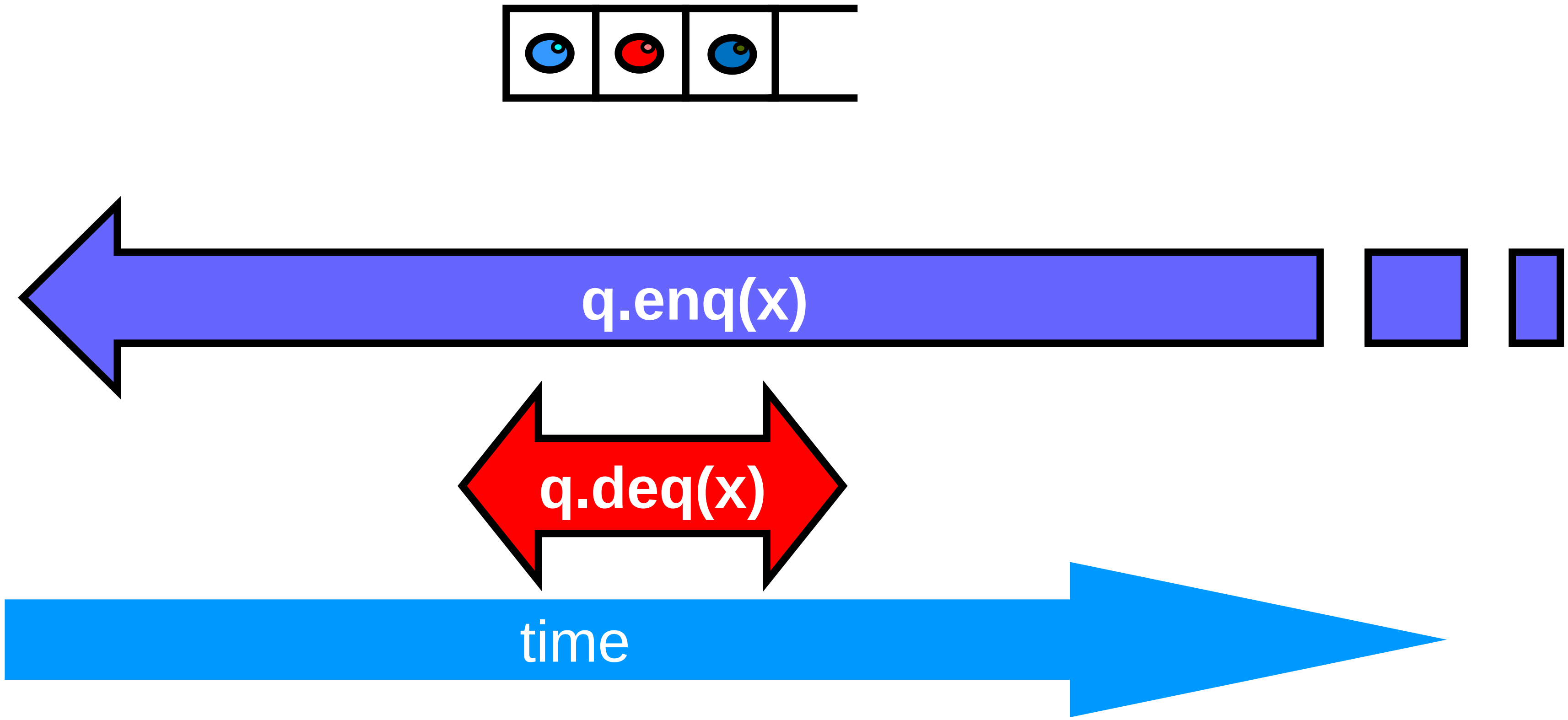


Example

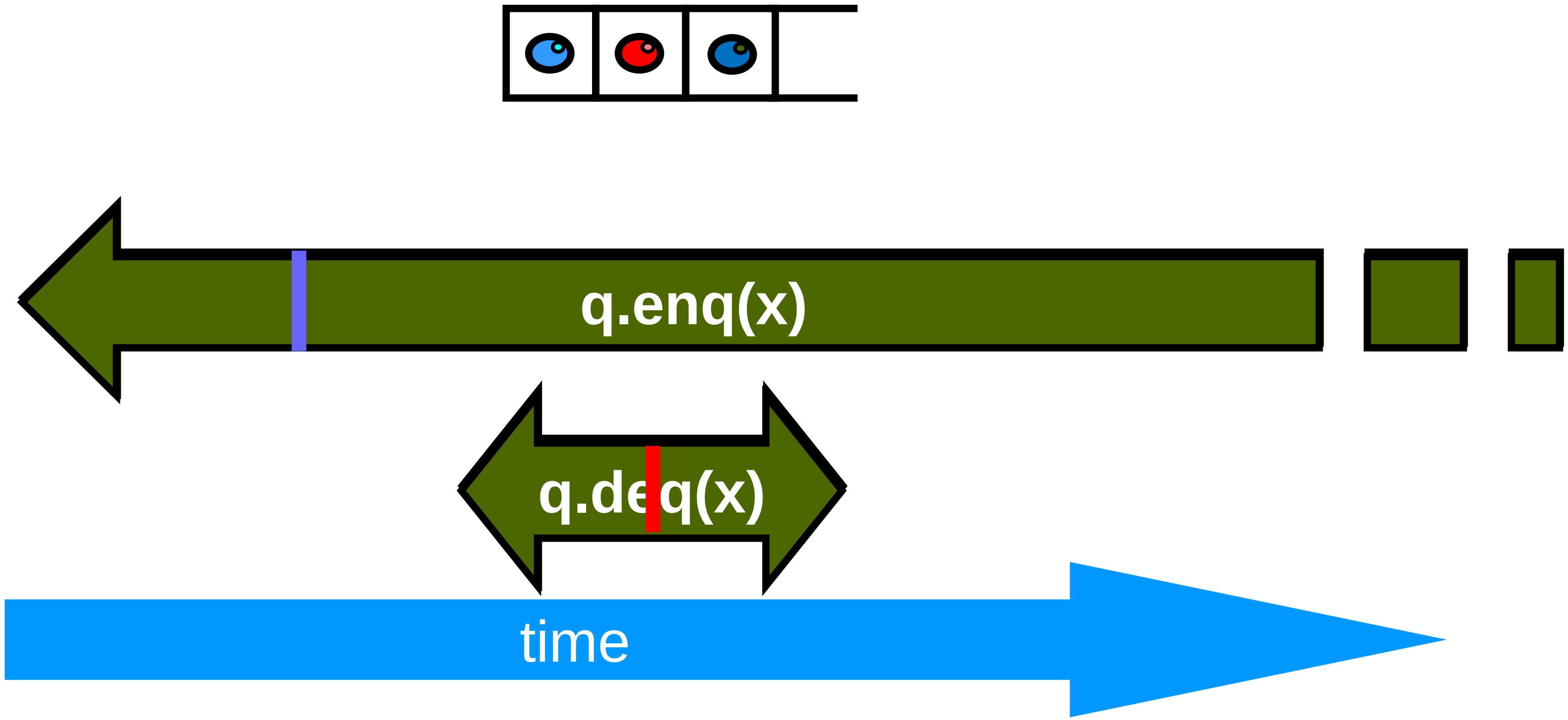




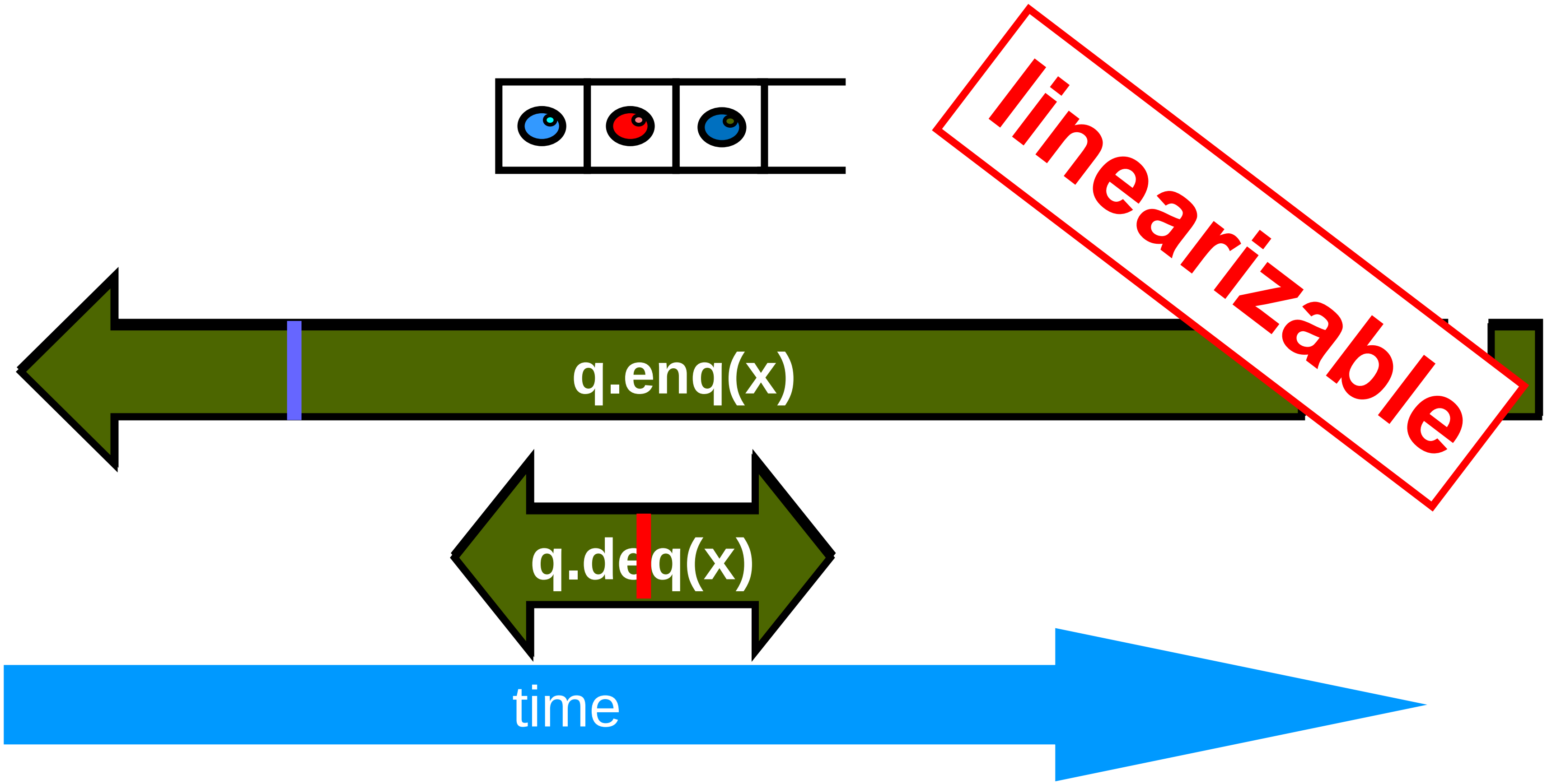
Example



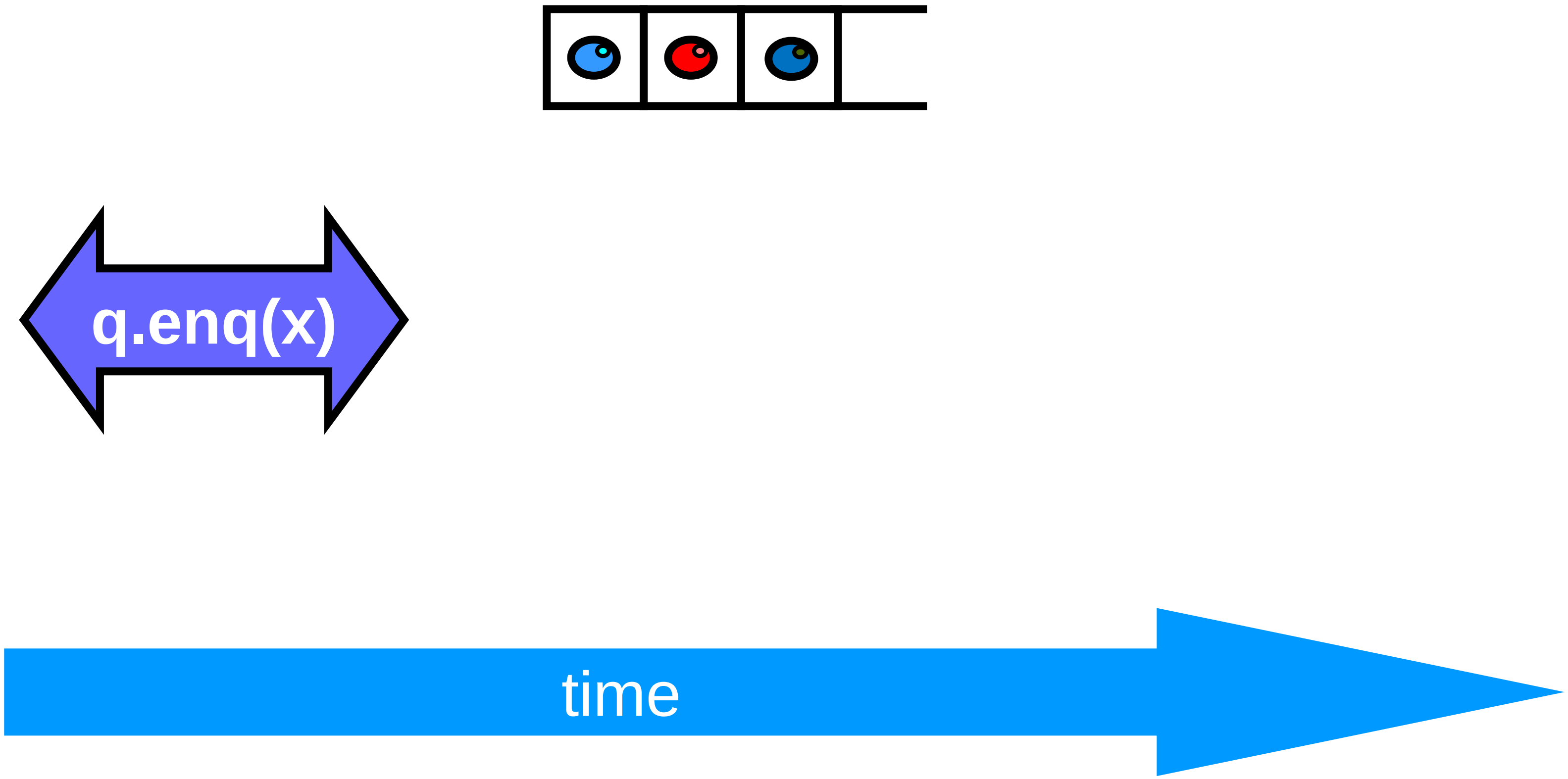
Example



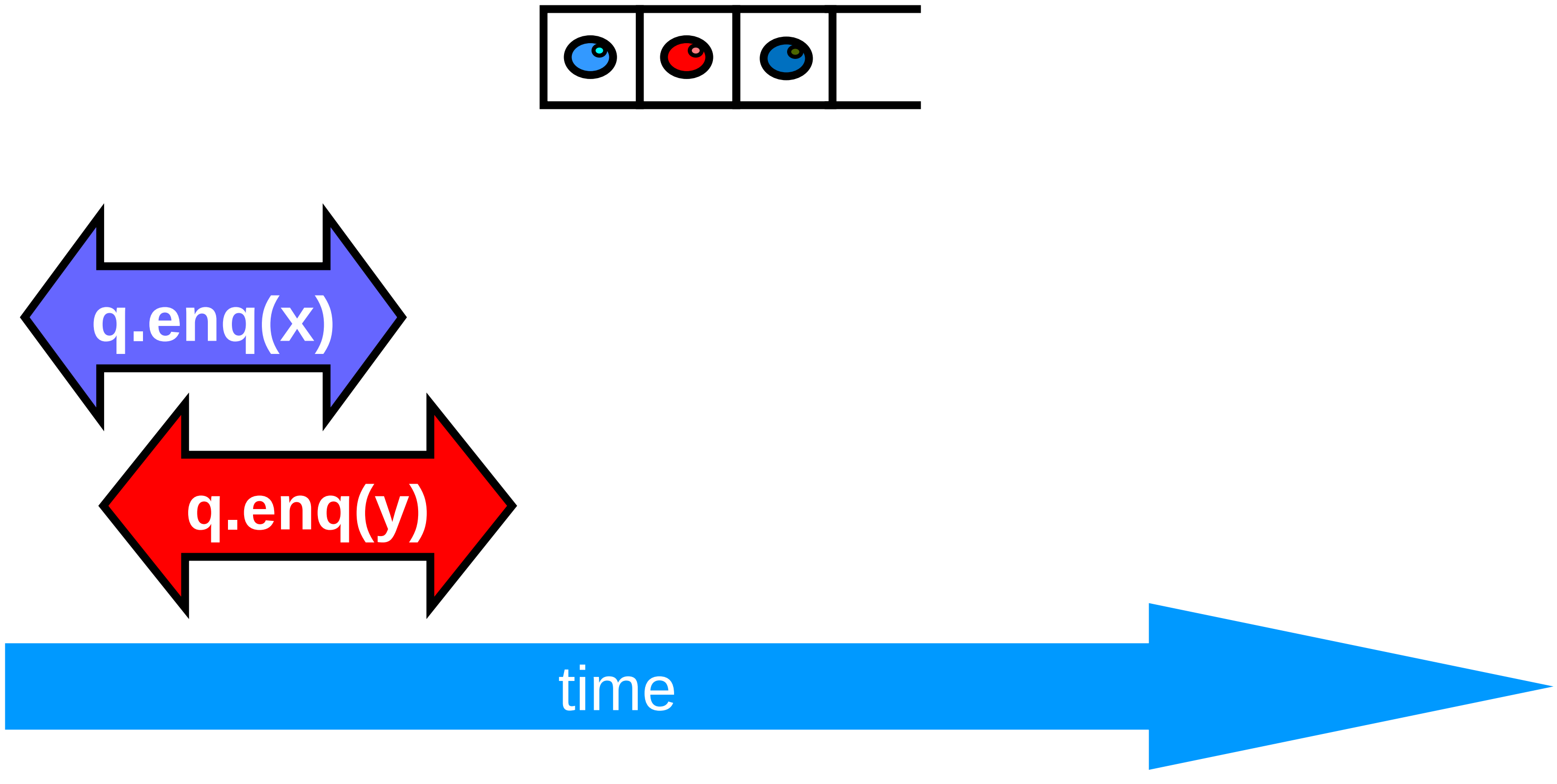
Example



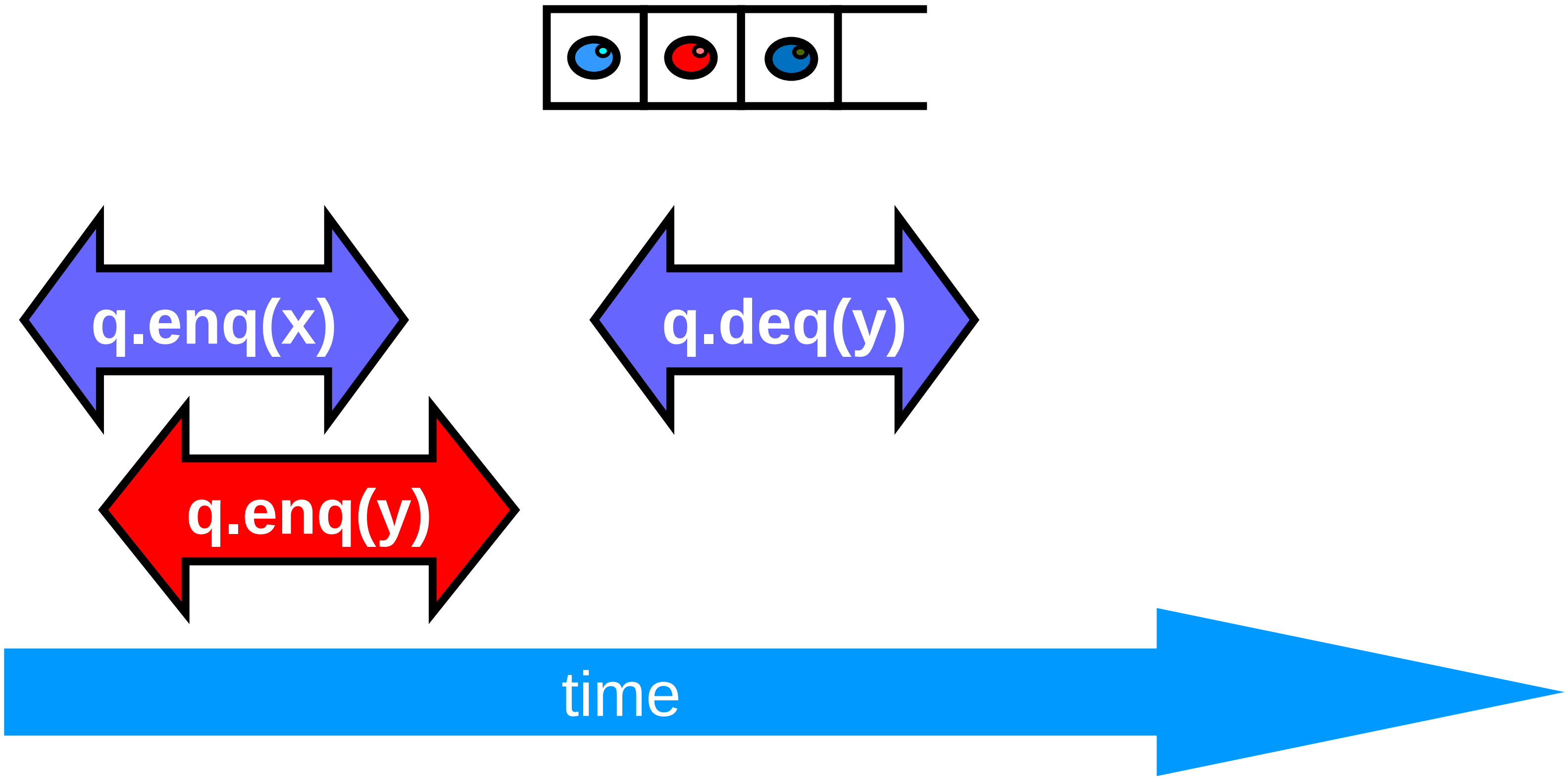
Example



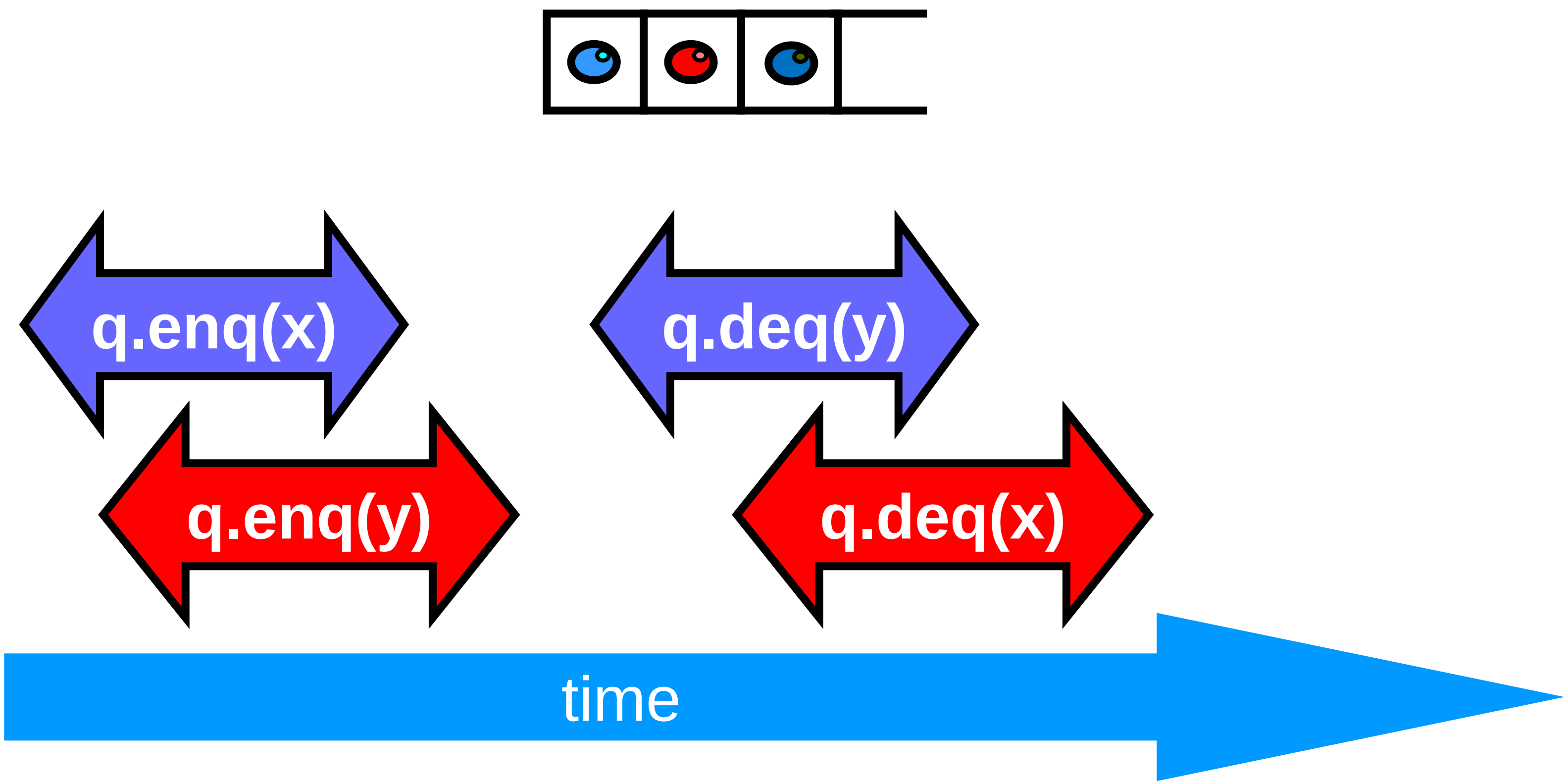
Example



Example



Example

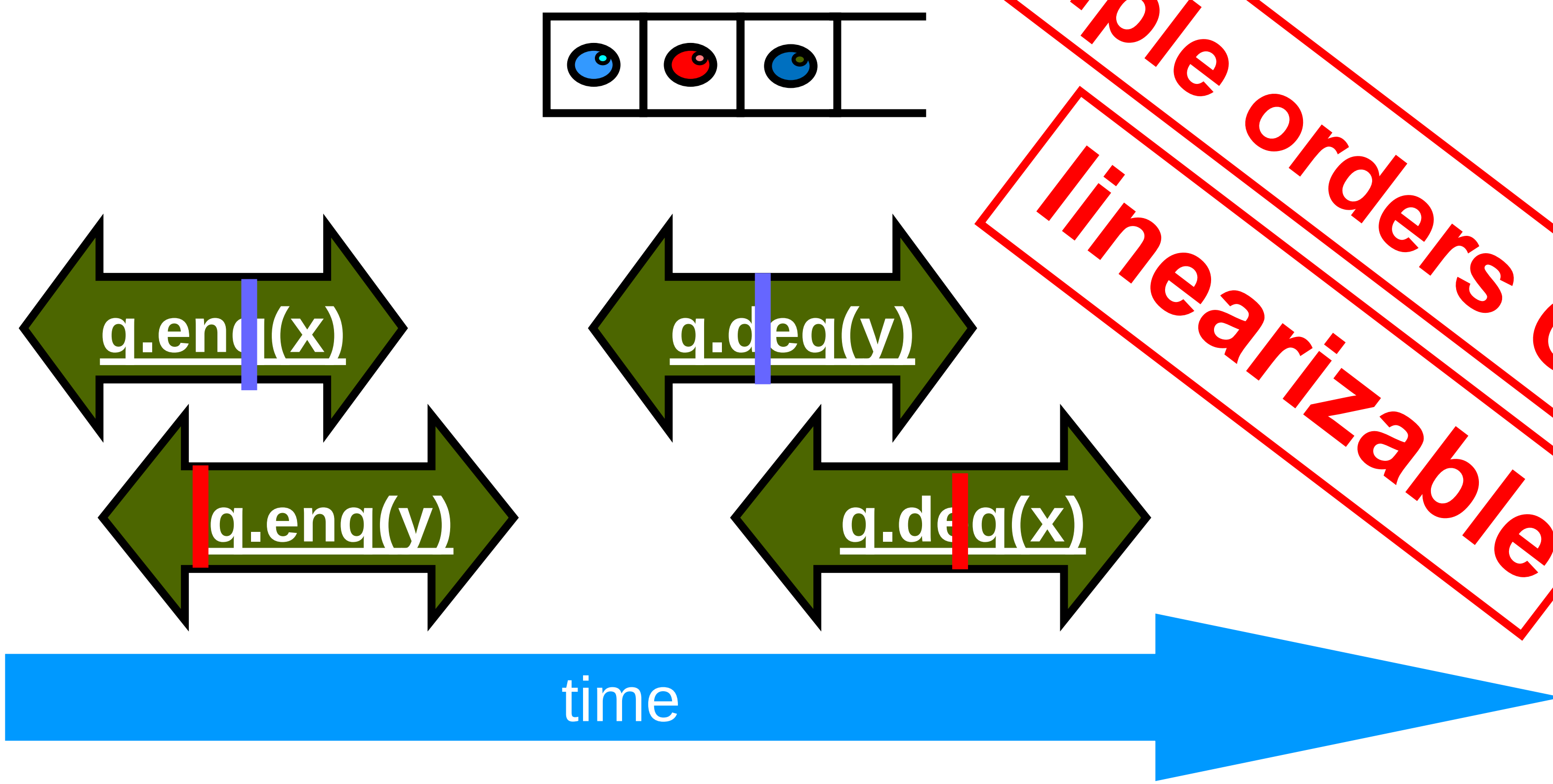


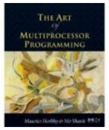
Comme ci
Comme ça

Example

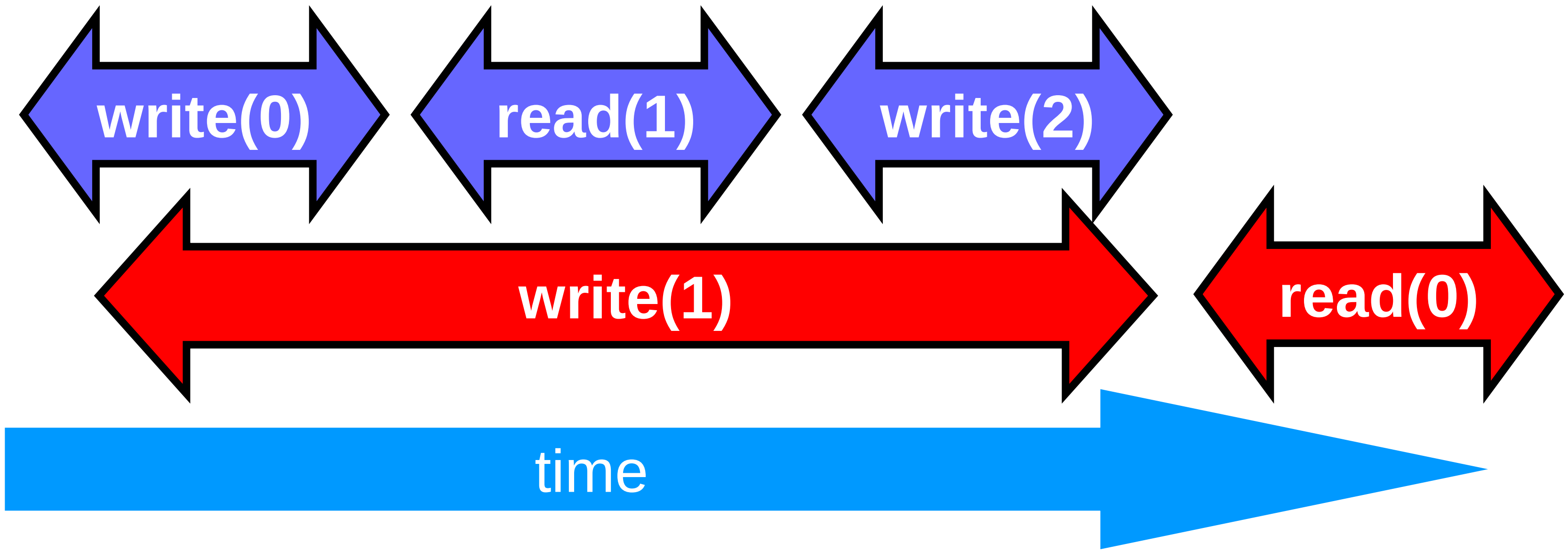
multiple orders OK

linearizable

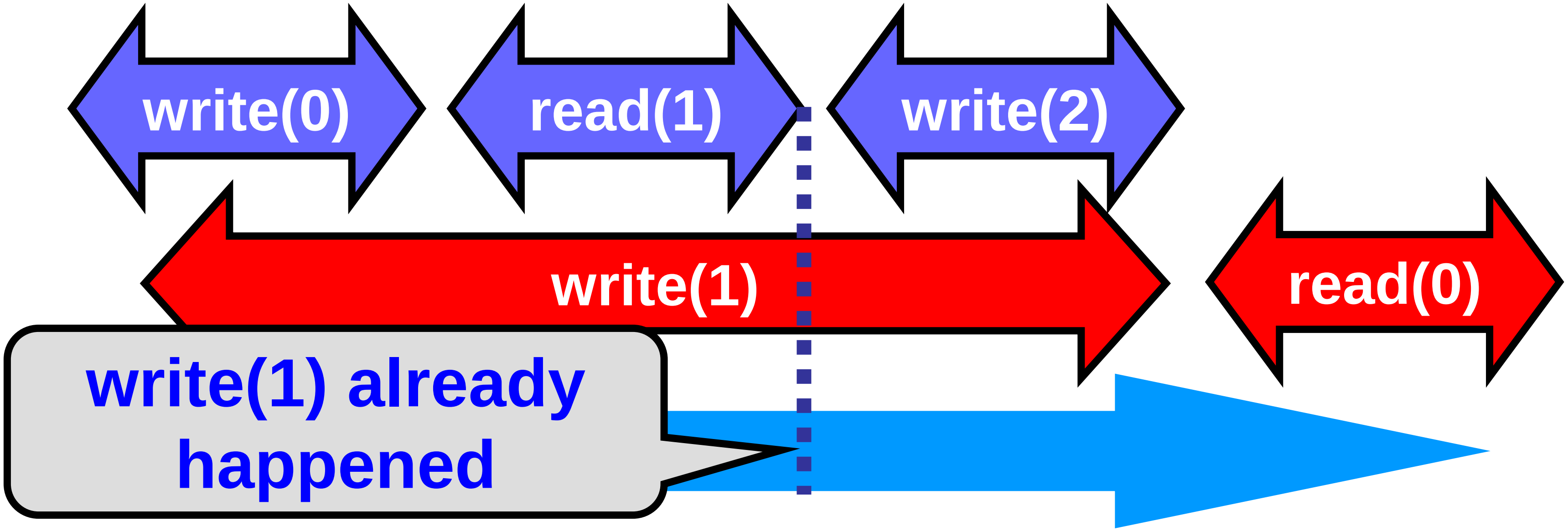


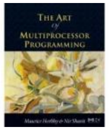


Read/Write Register Example

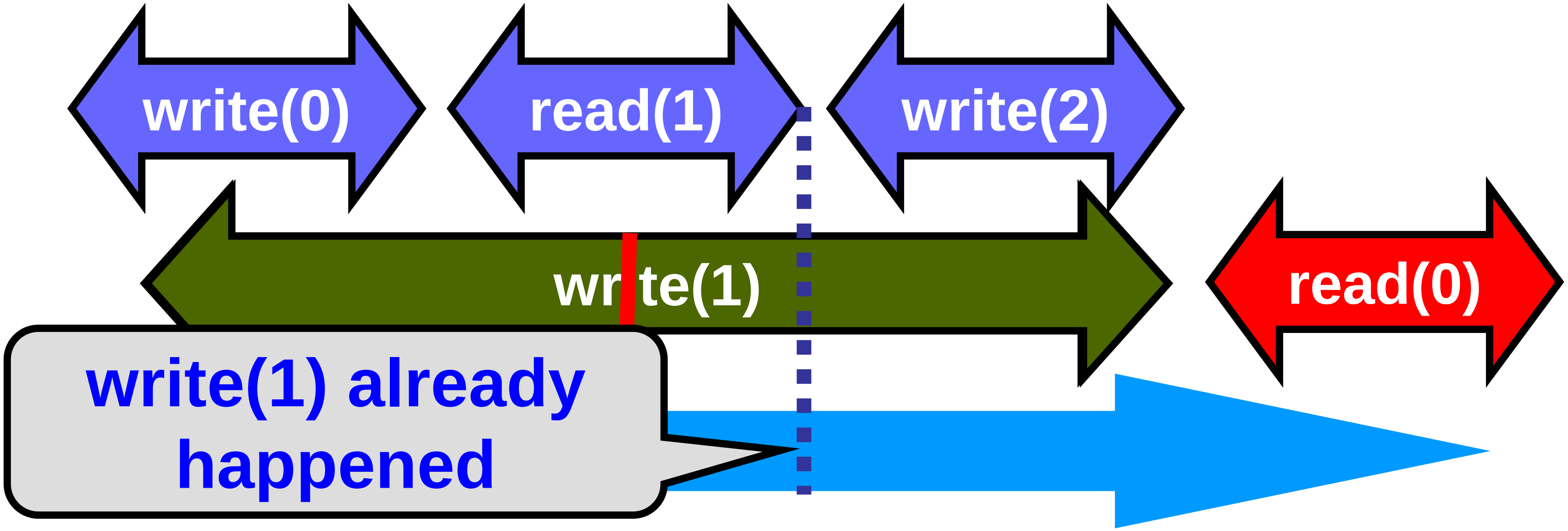


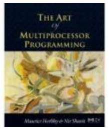
Read/Write Register Example



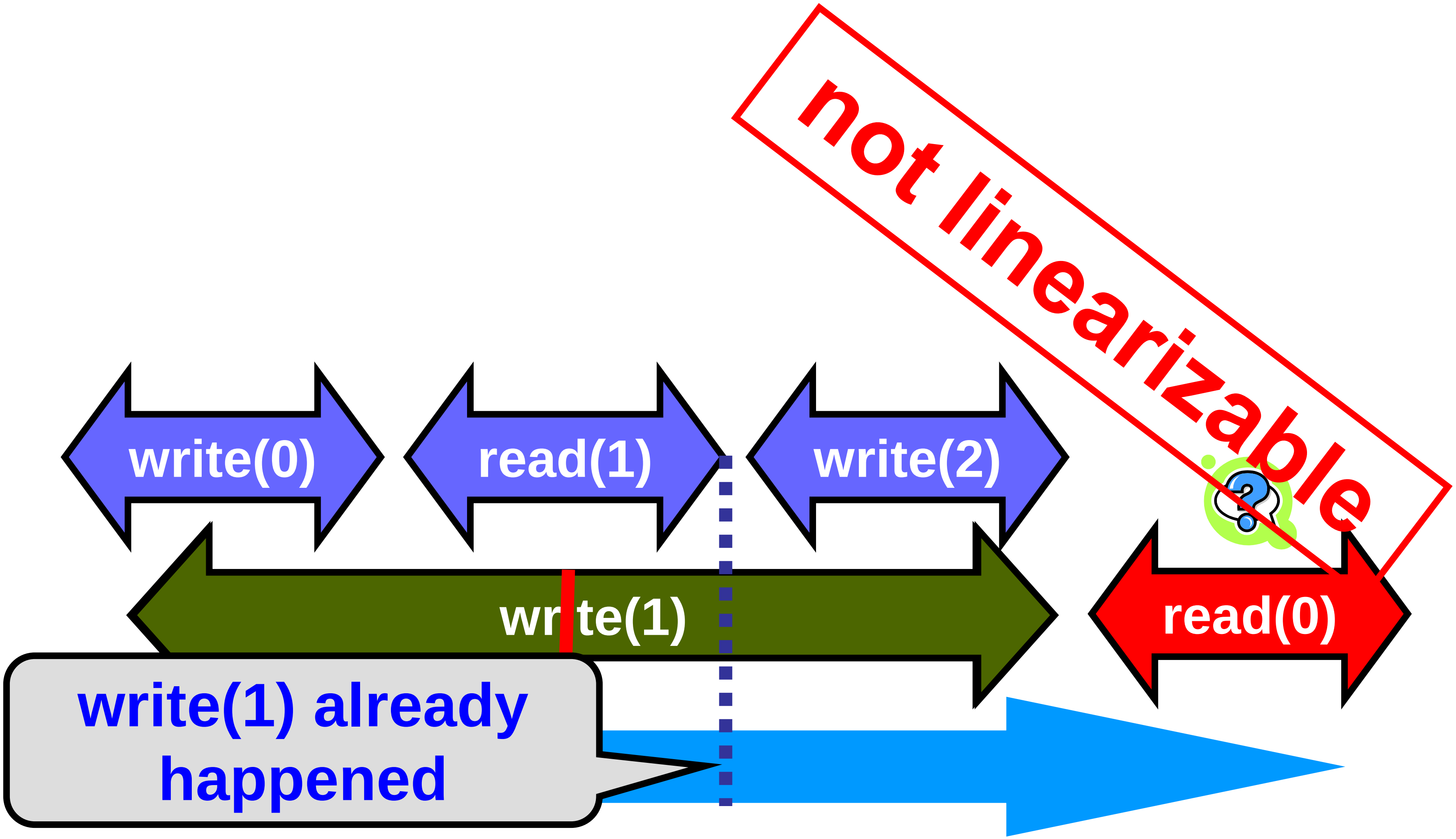


Read/Write Register Example

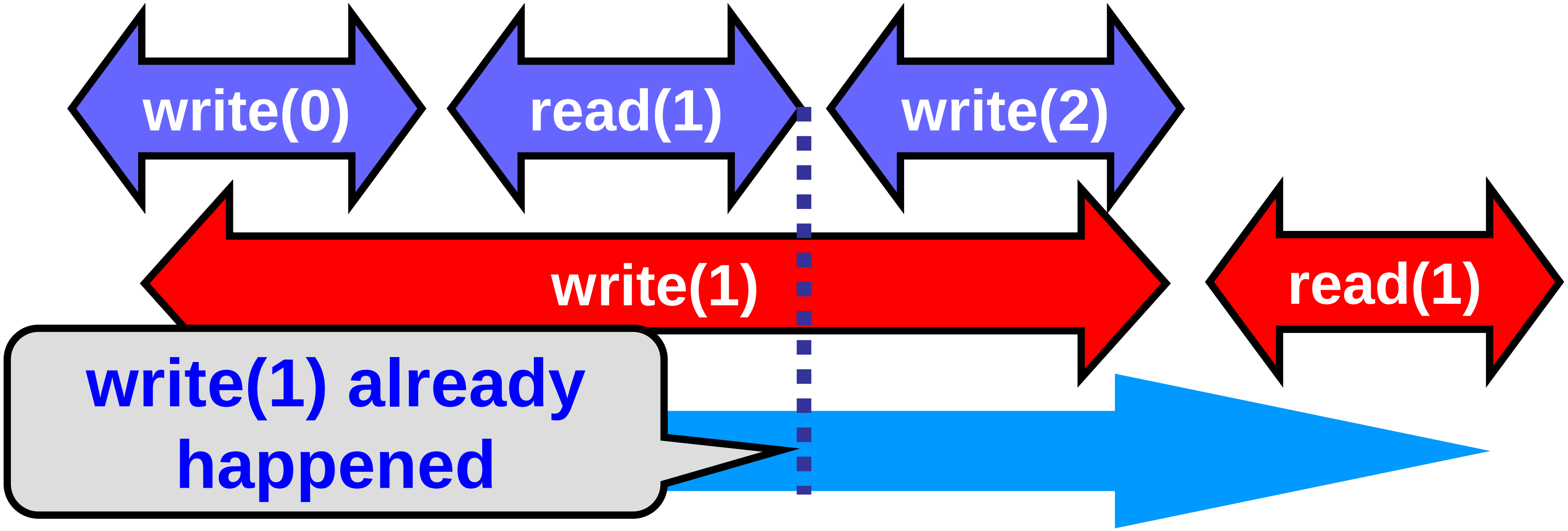


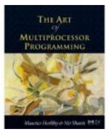


Read/Write Register Example

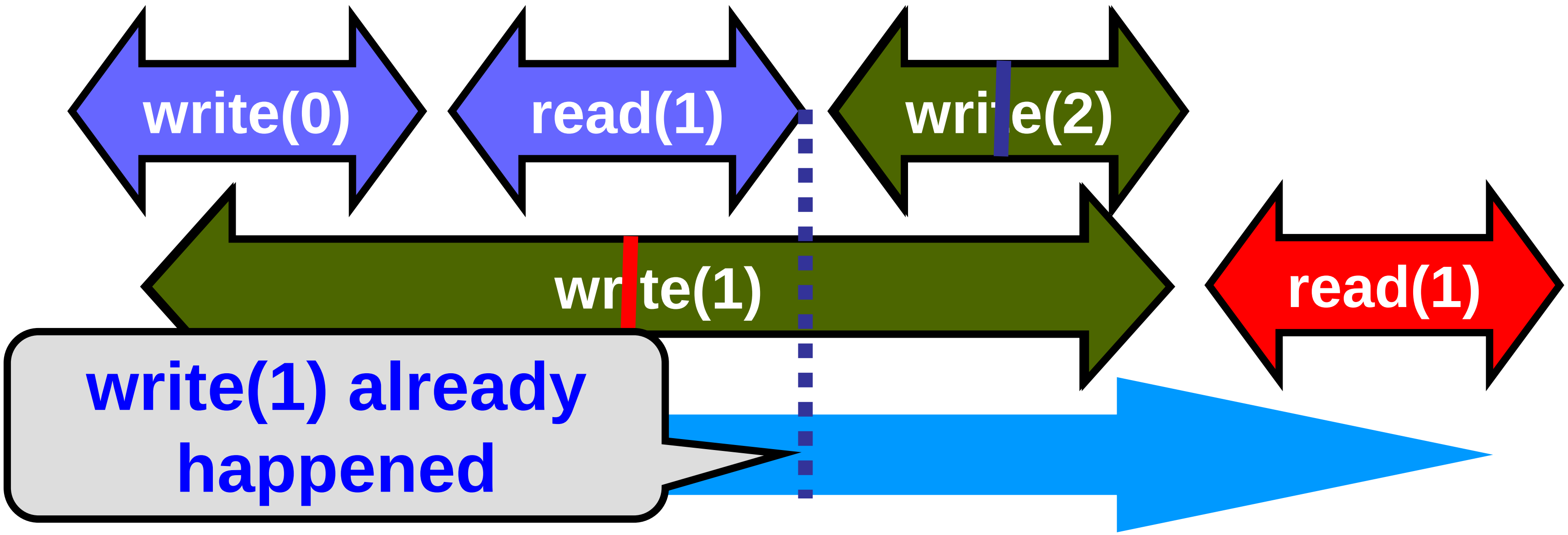


Read/Write Register Example

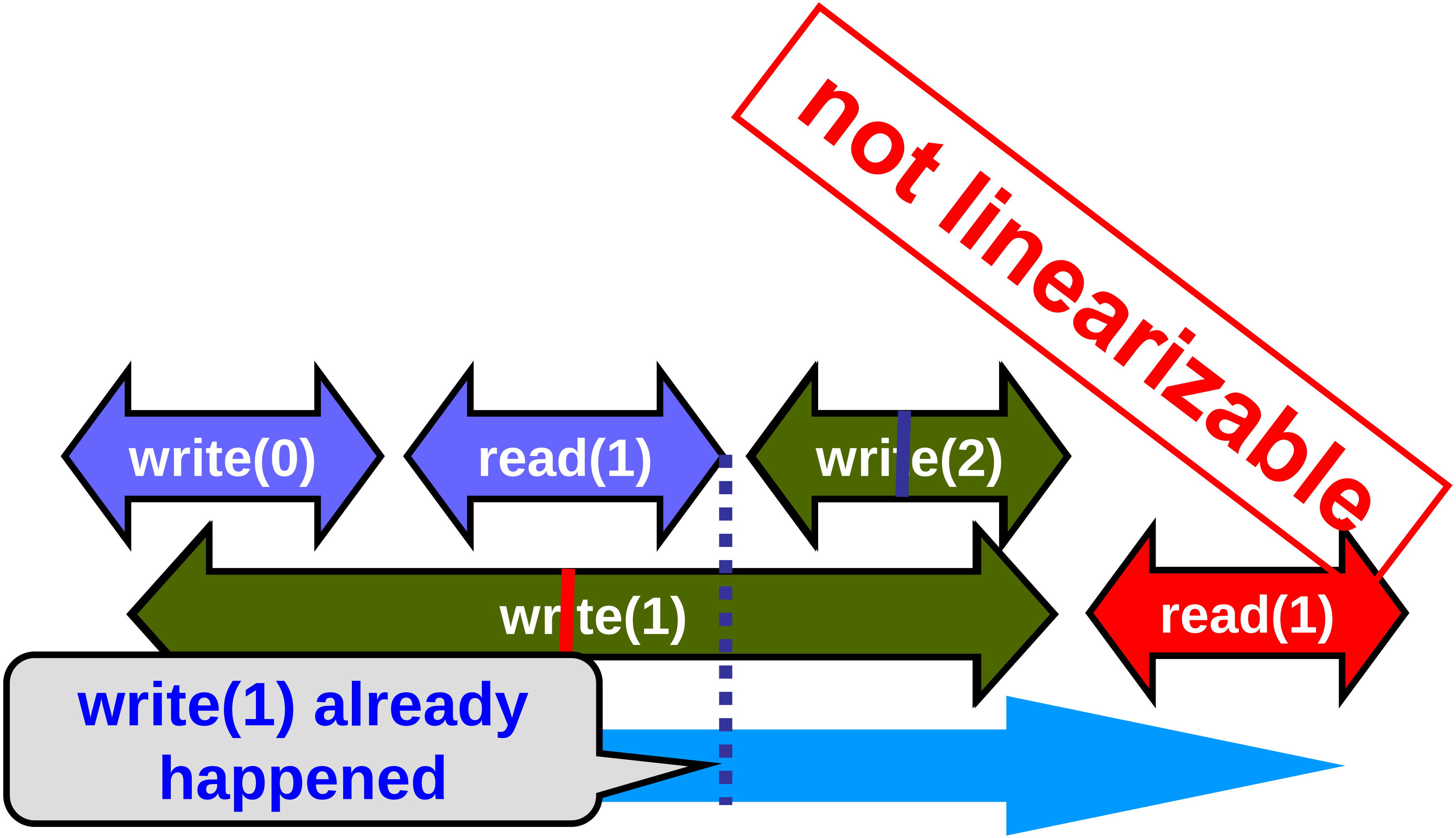


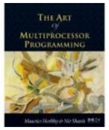


Read/Write Register Example

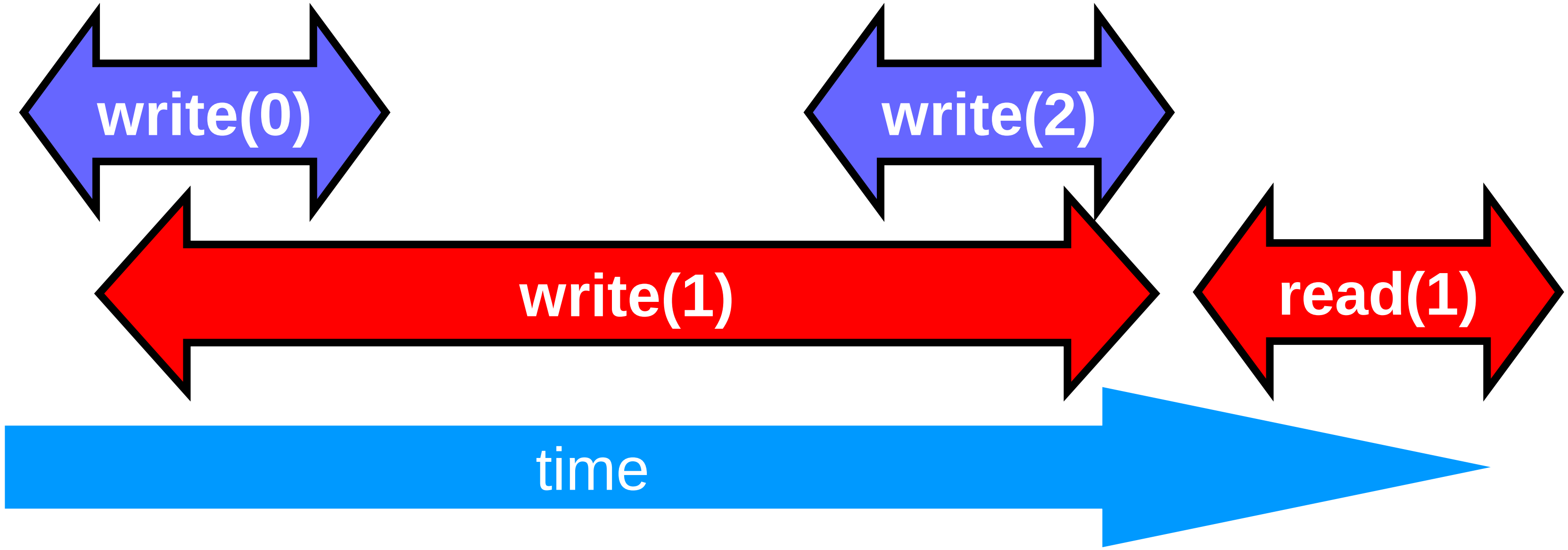


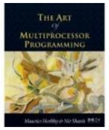
Read/Write Register Example



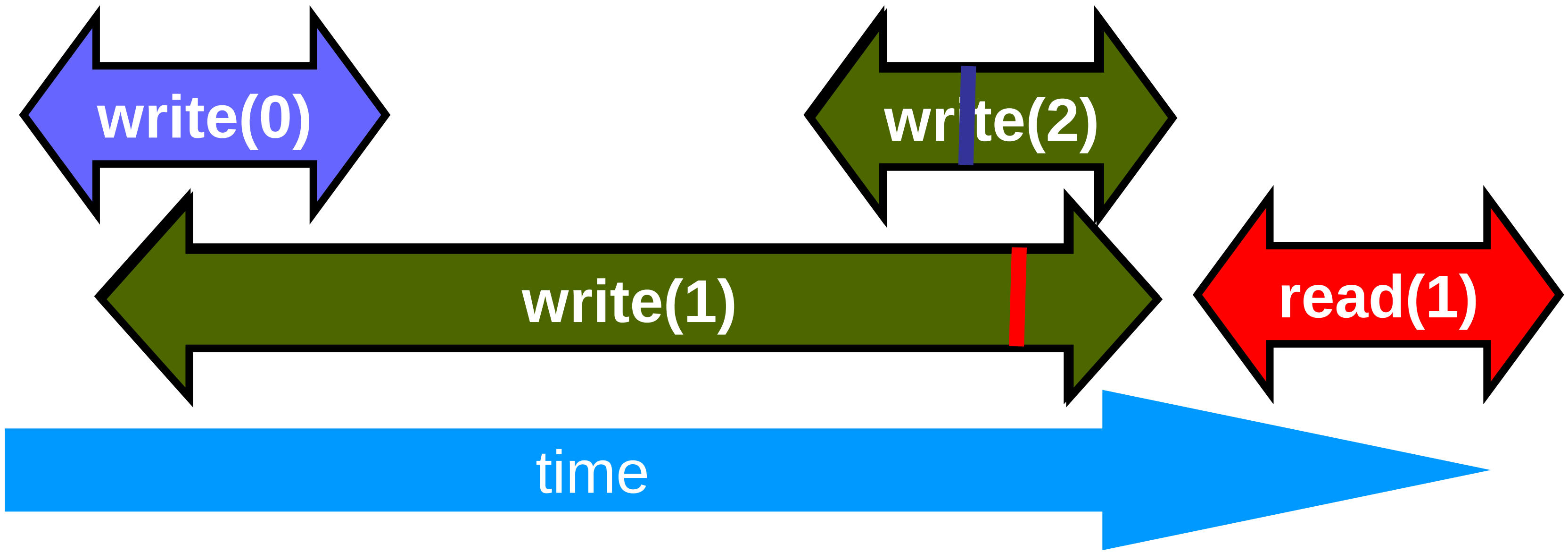


Read/Write Register Example

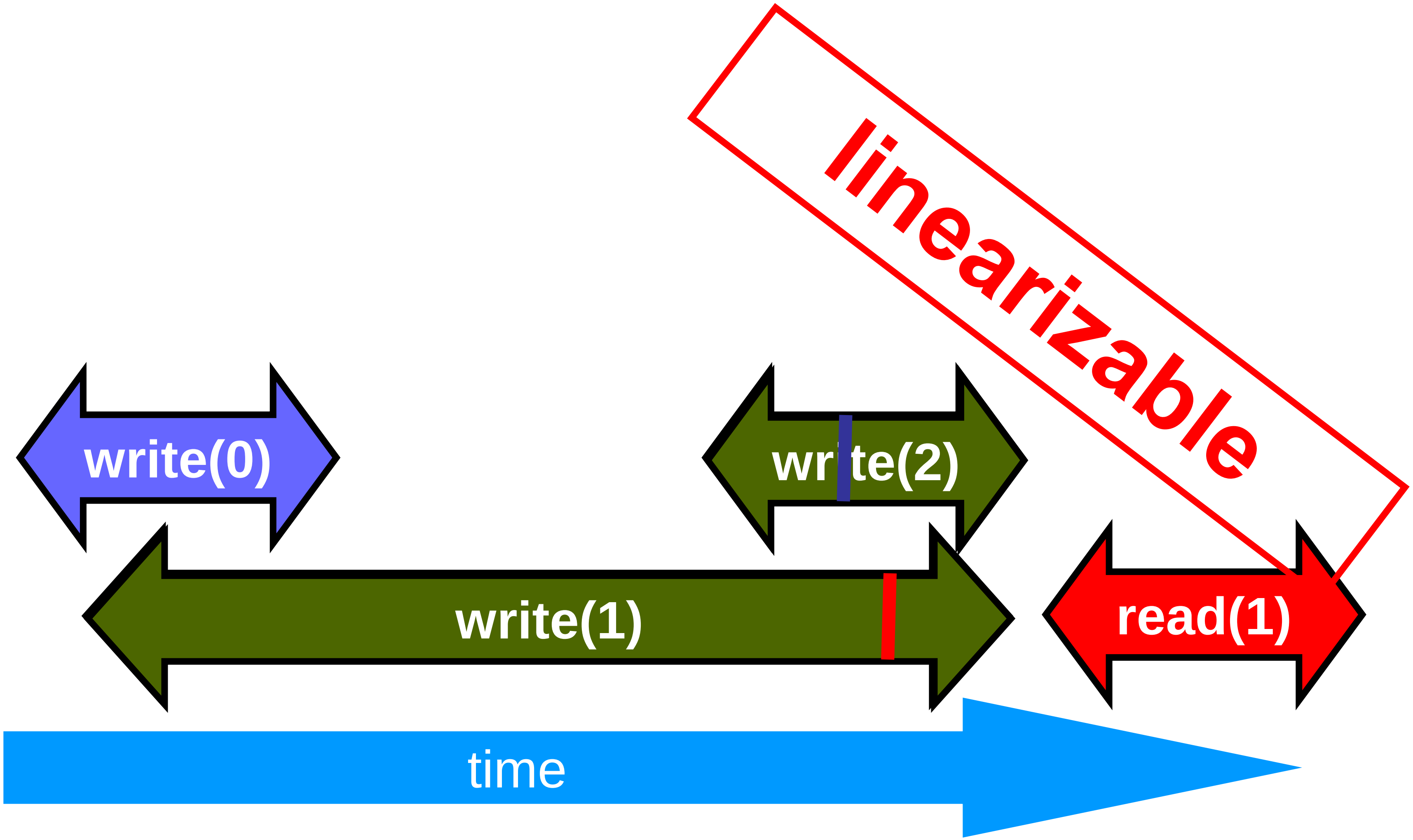




Read/Write Register Example

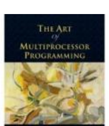


Read/Write Register Example



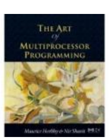
Talking About Executions

- 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***



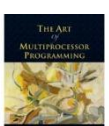
Formal Model of Executions

- 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!



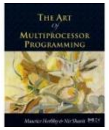
Split Method Calls into Two Events

- 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

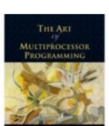
A q.enq(x)

thread

method

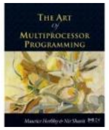
object

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

result

object

Response Notation

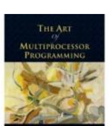
Method is implicit

A q: void

thread

result

object



Response Notation

Method is implicit

A q: empty()

thread

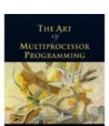
exception

object

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

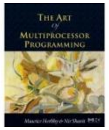
- Invocation & response *match* if

Thread names agree

Object names agree

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

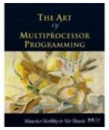
Method call



Object Projections

H =

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

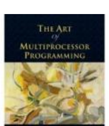


Object Projections

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

H|q =

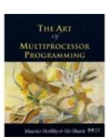
```
B q.deq()
B q:3
```



Thread Projections

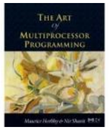
H =

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



Thread Projections

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



Complete Subhistory

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

H =

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

Complete Subhistory

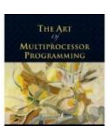
```
      A q.enq(3)
      A q:void
      A q.enq(5)
H =   B p.enq(4)
      B p:void
      B q.deq()
      B q:3
```

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

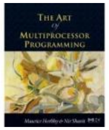


Complete Subhistory

```
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

match

```
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

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
```

match

```
B p.enq(4)
B p:void
```

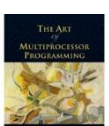
match

```
B q.deq()
B q:3
```

match

```
A q:enq(5)
```

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)
```

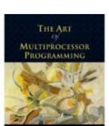
match

match

match

Final pending invocation OK

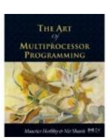
Method calls of different threads do not interleave



Well-Formed Histories

H=

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

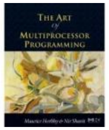
Well-Formed Histories

Per-thread projections sequential

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

```
     B p.enq(4)
H|B= 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

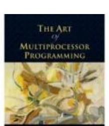
$$\left\{ \begin{array}{l} H|A = G|A \\ H|B = G|B \end{array} \right.$$

H=

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

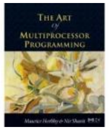
G=

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



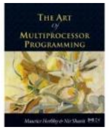
Sequential Specifications

- 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

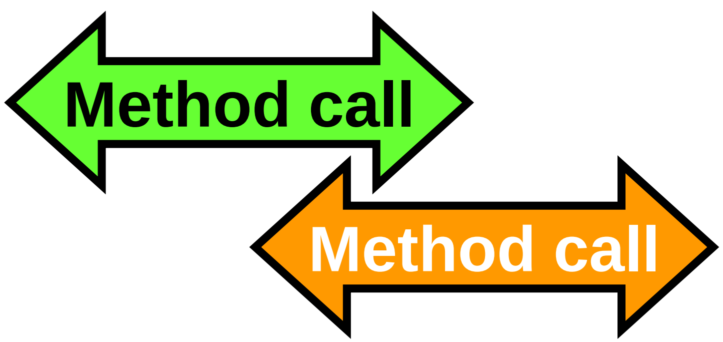
Method call

Method call

Non-Precedence

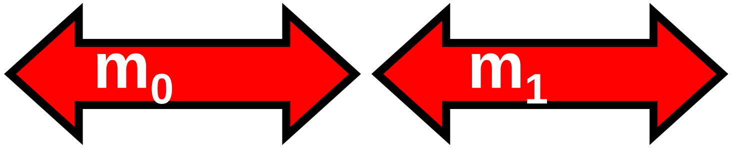
```
A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3
```

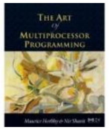
Some method calls overlap one another



Notation

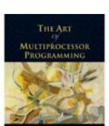
- Given
  - History **H**
  - method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**
- We say $m_0 \rightarrow_{\mathbf{H}} m_1$, if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$
- Relation $\mathbf{m_0 \rightarrow_H m_1}$ is a
  - Partial order
  - Total order if **H** is sequential





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 $\rightarrow_{\mathbf{G}} \subset \rightarrow_{\mathbf{S}}$



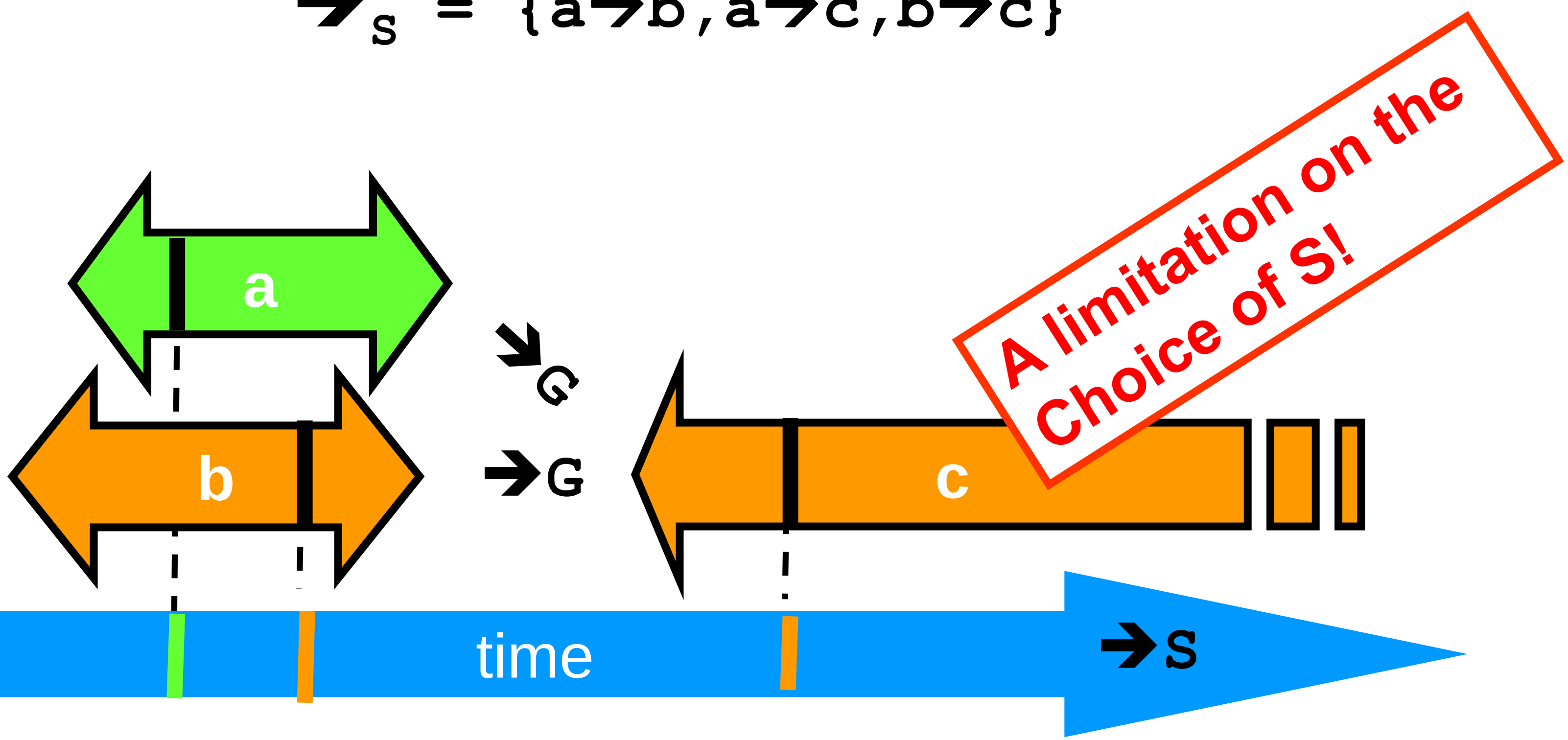
Remarks

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

Ensuring $\rightarrow_G \subset \rightarrow_S$

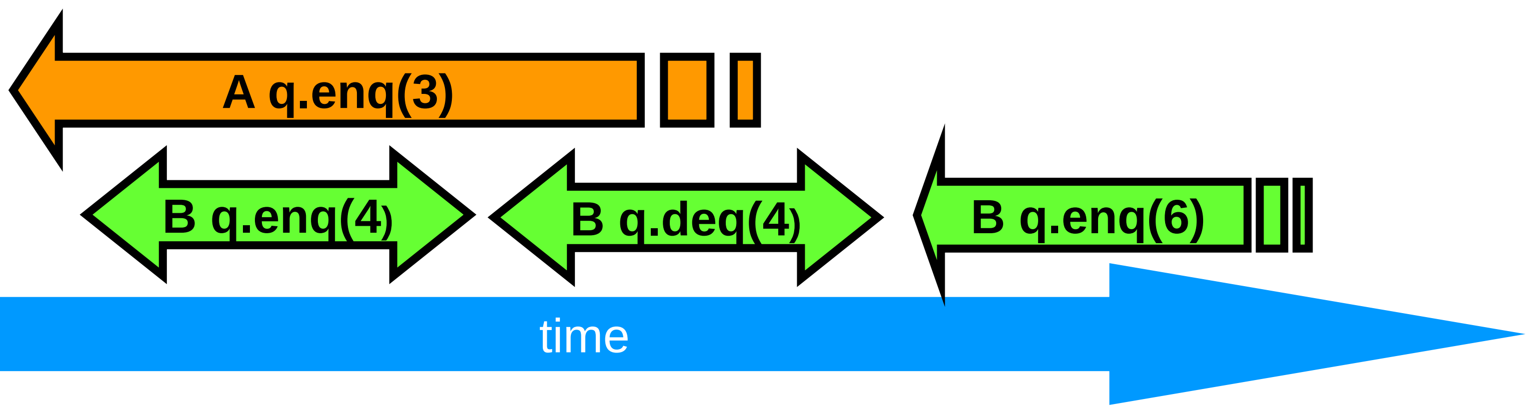
$\rightarrow_G$ = {a$\rightarrow$c,b$\rightarrow$c}

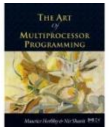
$\rightarrow_S$ = {a$\rightarrow$b,a$\rightarrow$c,b$\rightarrow$c}



Example

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

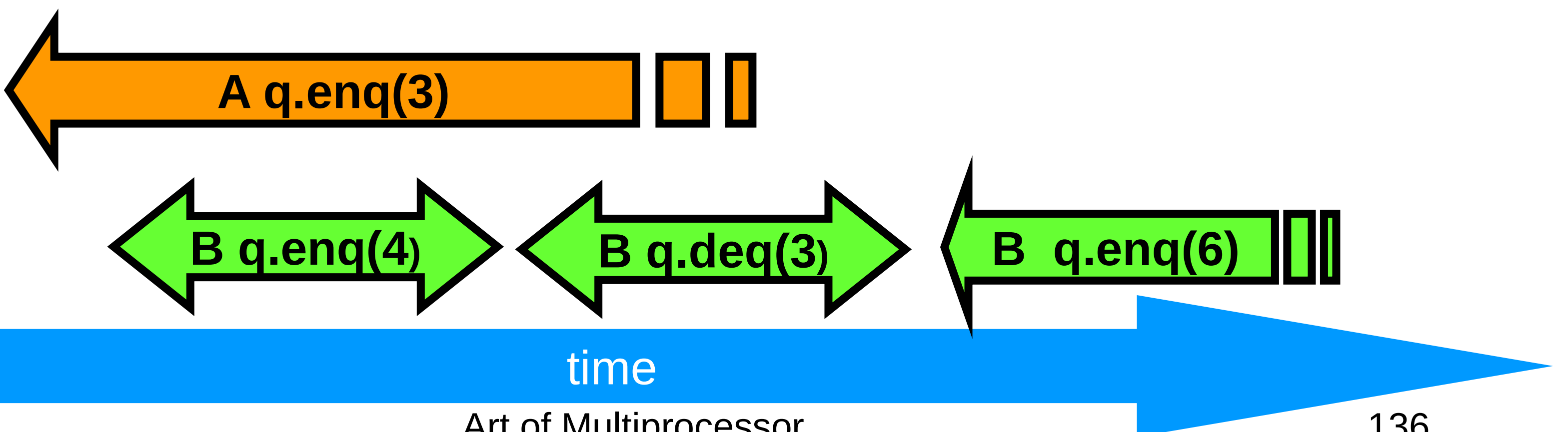




Example

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

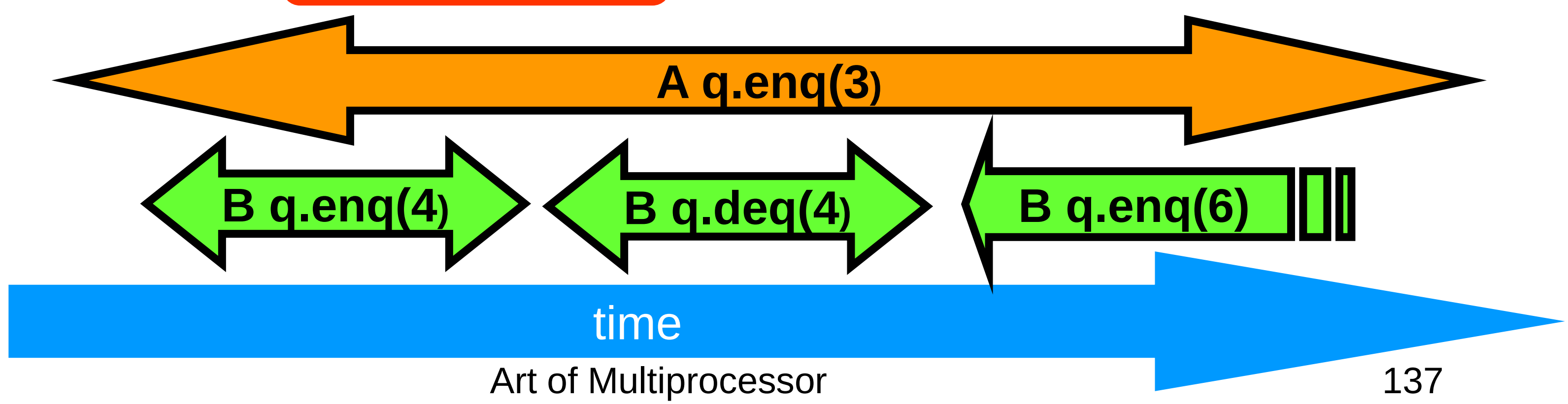
Complete this pending invocation



Example

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

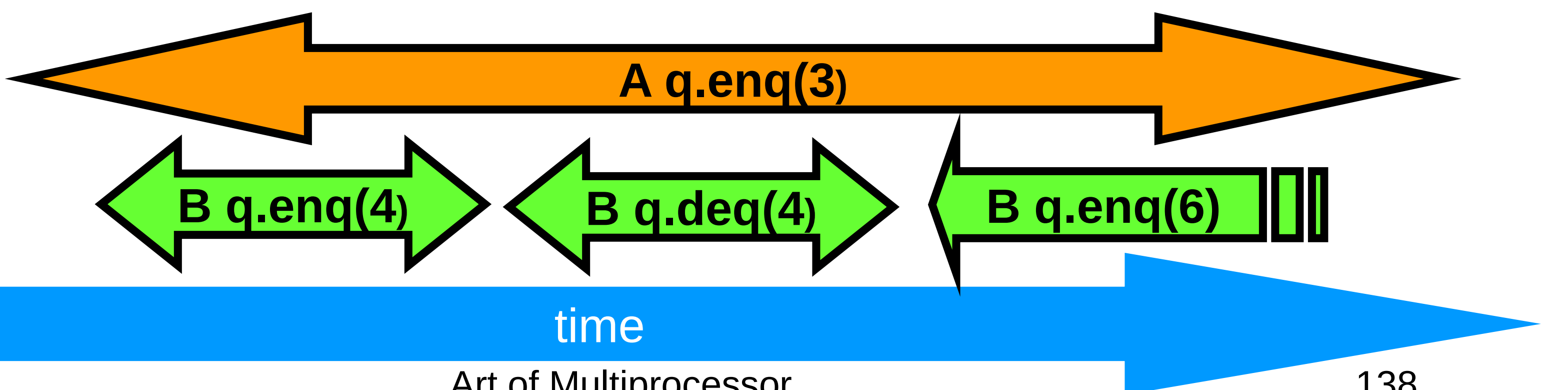
Complete this pending invocation

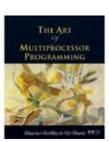


Example

```
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



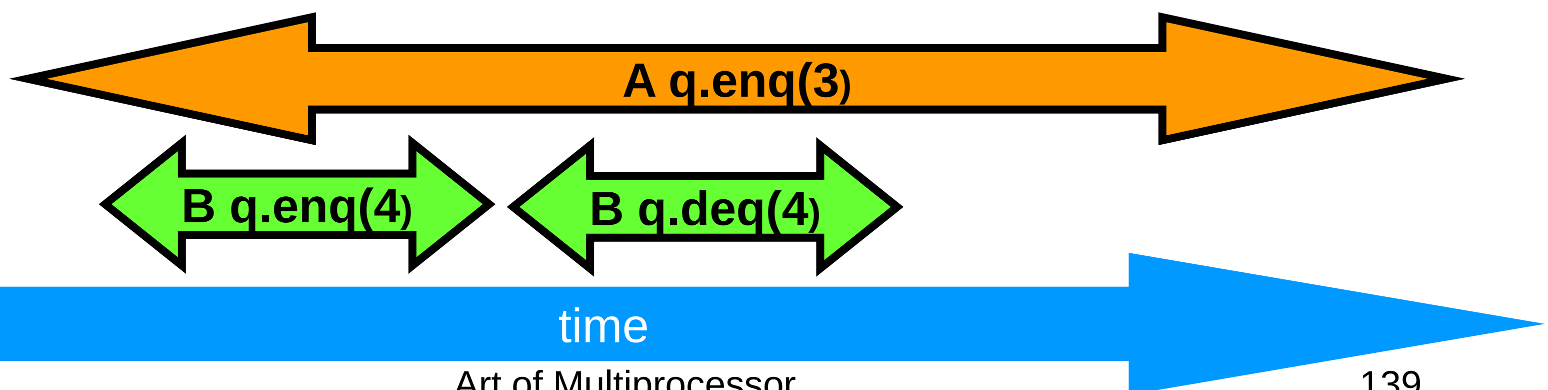


Example

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

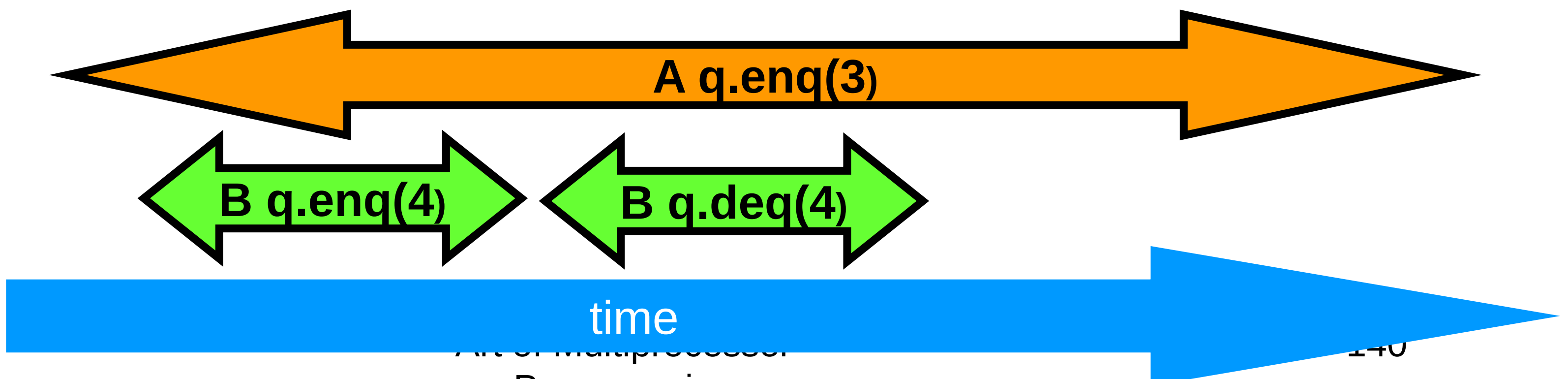
discard this one

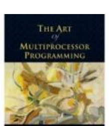
```
A q:void
```



Example

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

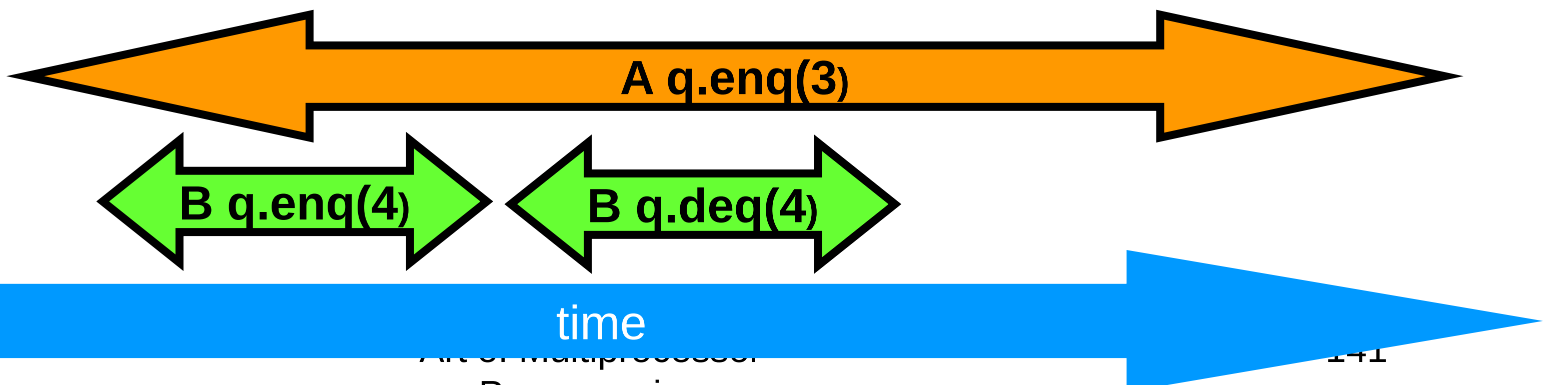




Example

```
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
```

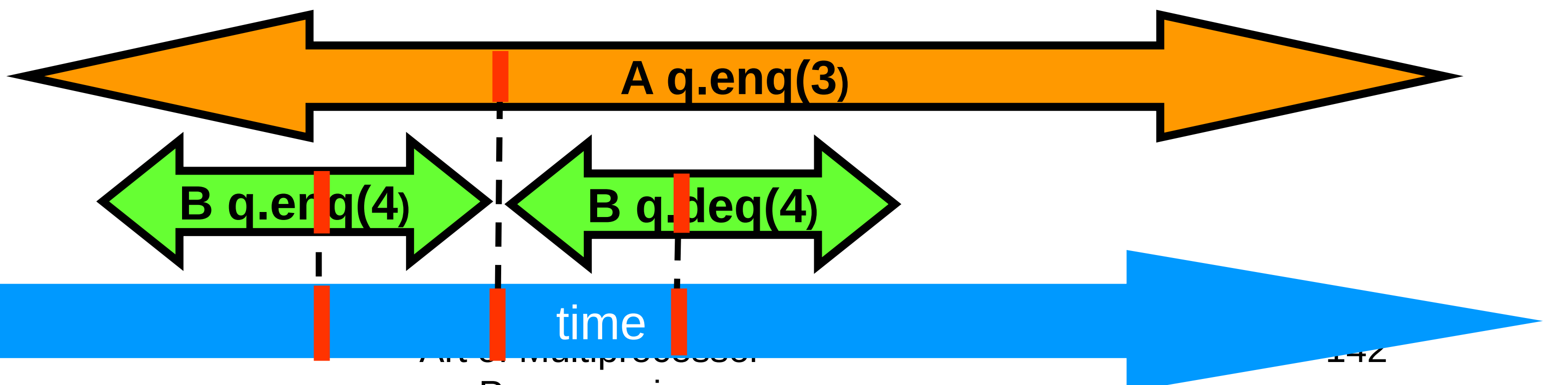


Example

Equivalent sequential history

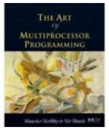
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```

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



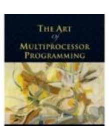
Concurrency

- How much concurrency does linearizability allow?
- When must a method invocation block?



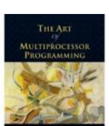
Concurrency

- Focus on ***total*** methods
  - Defined in every state
- Example:
  - **`deq()`** that throws **`Empty`** exception
  - Versus **`deq()`** that waits …
- Why?
  - Otherwise, blocking unrelated to synchronization



Concurrency

- Question: When does linearizability require a method invocation to block?
- Answer: never.
- Linearizability is *non-blocking*



Non-Blocking Theorem

If method invocation

`A q.inv(…)`

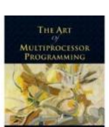
is pending in history H, then there exists a response

`A q:res(…)`

such that

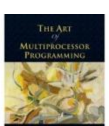
`H + A q:res(…)`

is linearizable



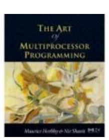
Proof

- Pick linearization S of H
- If S already contains
  - Invocation **`A q.inv(…)`** and response,
  - Then we are done.
- Otherwise, pick a response such that
  - **`S + A q.inv(…) + A q:res(…)`**
  - Possible because object is ***total***.



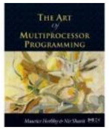
Composability Theorem

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



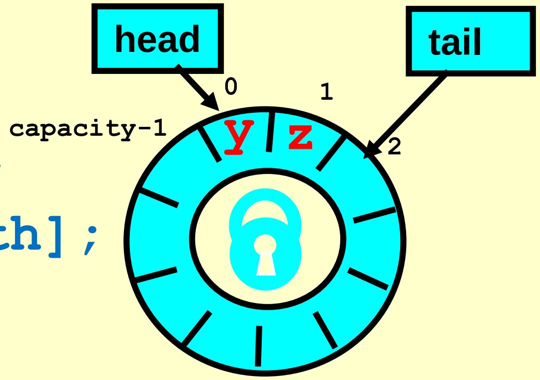
Why Does Composability Matter?

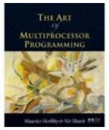
- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects



Reasoning About Linearizability: Locking

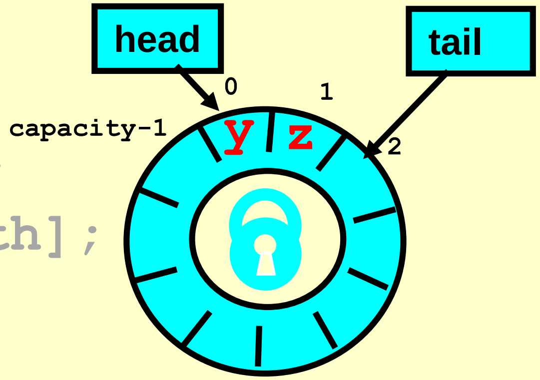
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



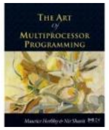


Reasoning About Linearizability: Locking

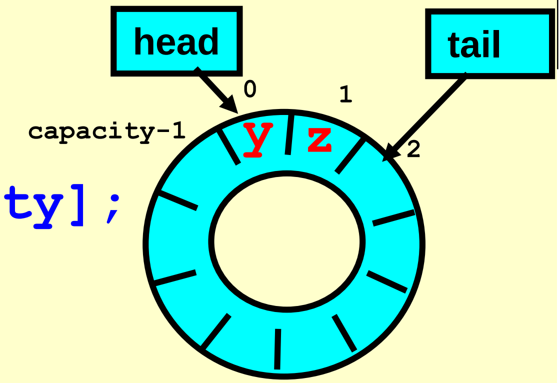
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



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   tFreeQueue {

  int         l =
  it      new Ob

        d 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

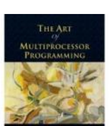
Linearization order is order head and tail fields modified

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

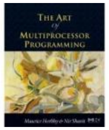
- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don’t leave home without it



Alternative: Sequential Consistency

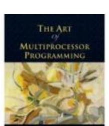
- History H is ***Sequentially Consistent*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to a
  - Legal sequential history **S**
  - ~~Where **→G** ⊂ **→S**~~

Differs from linearizability

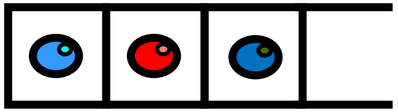


Sequential Consistency

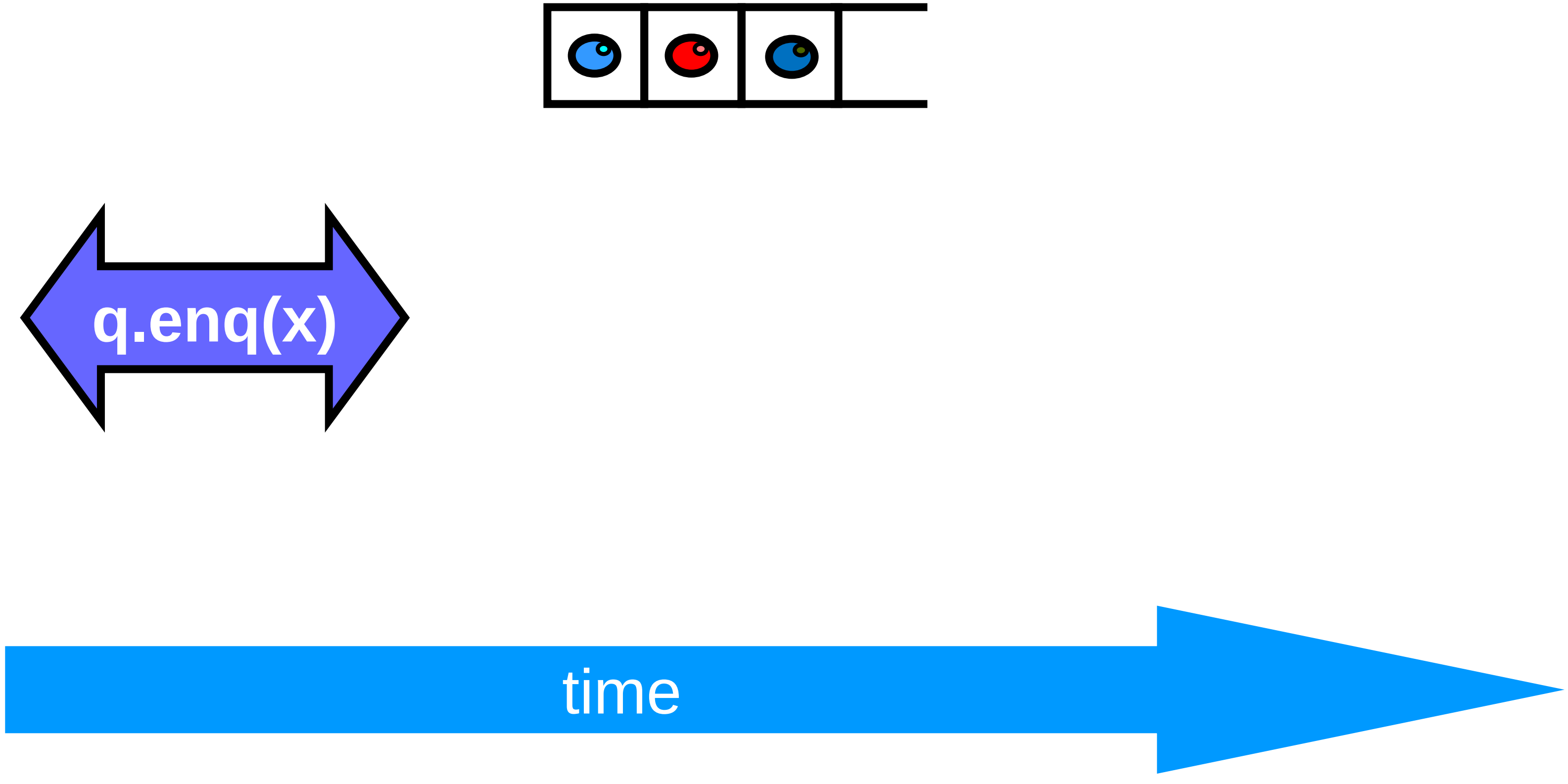
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures



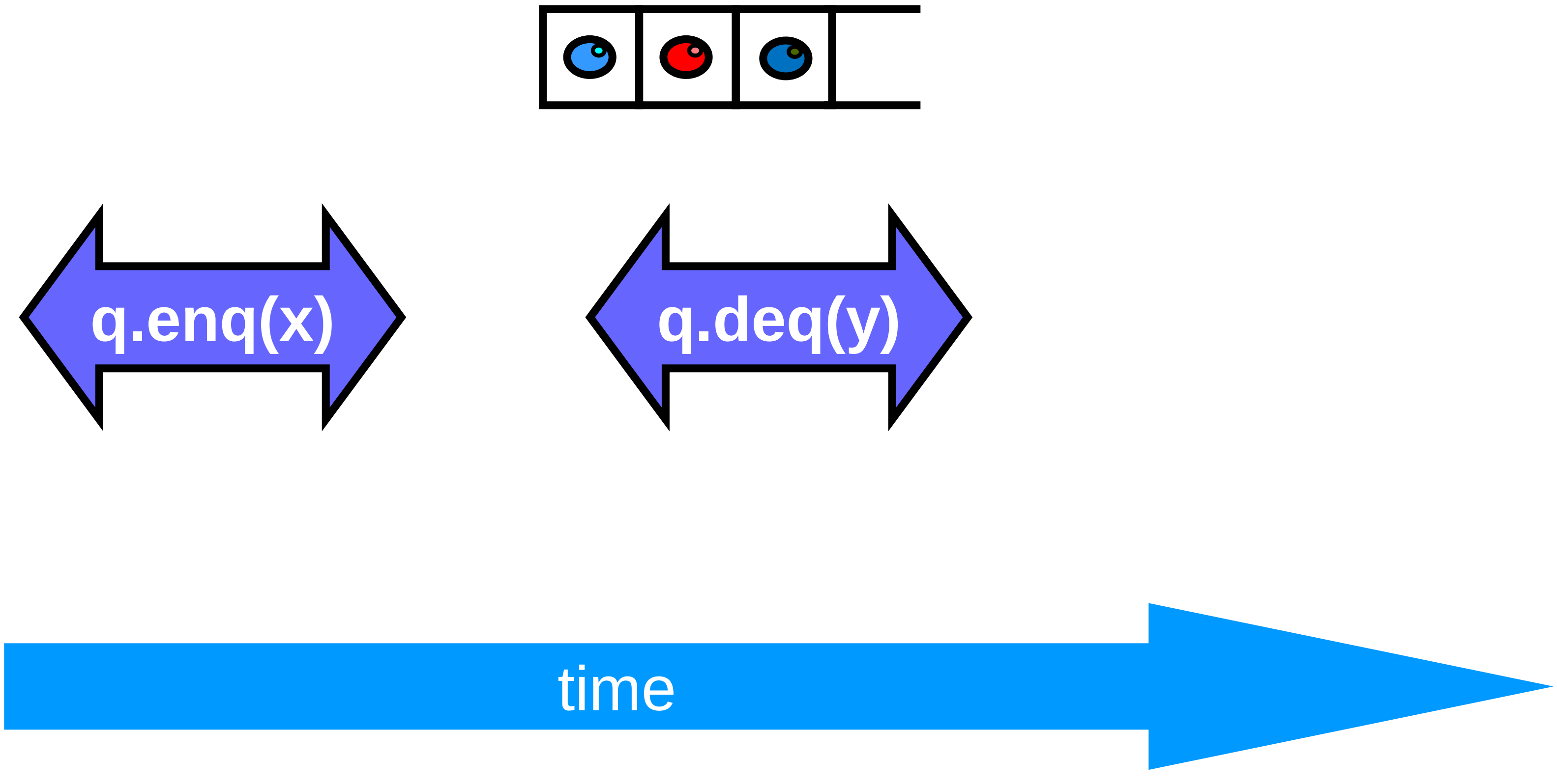
Example



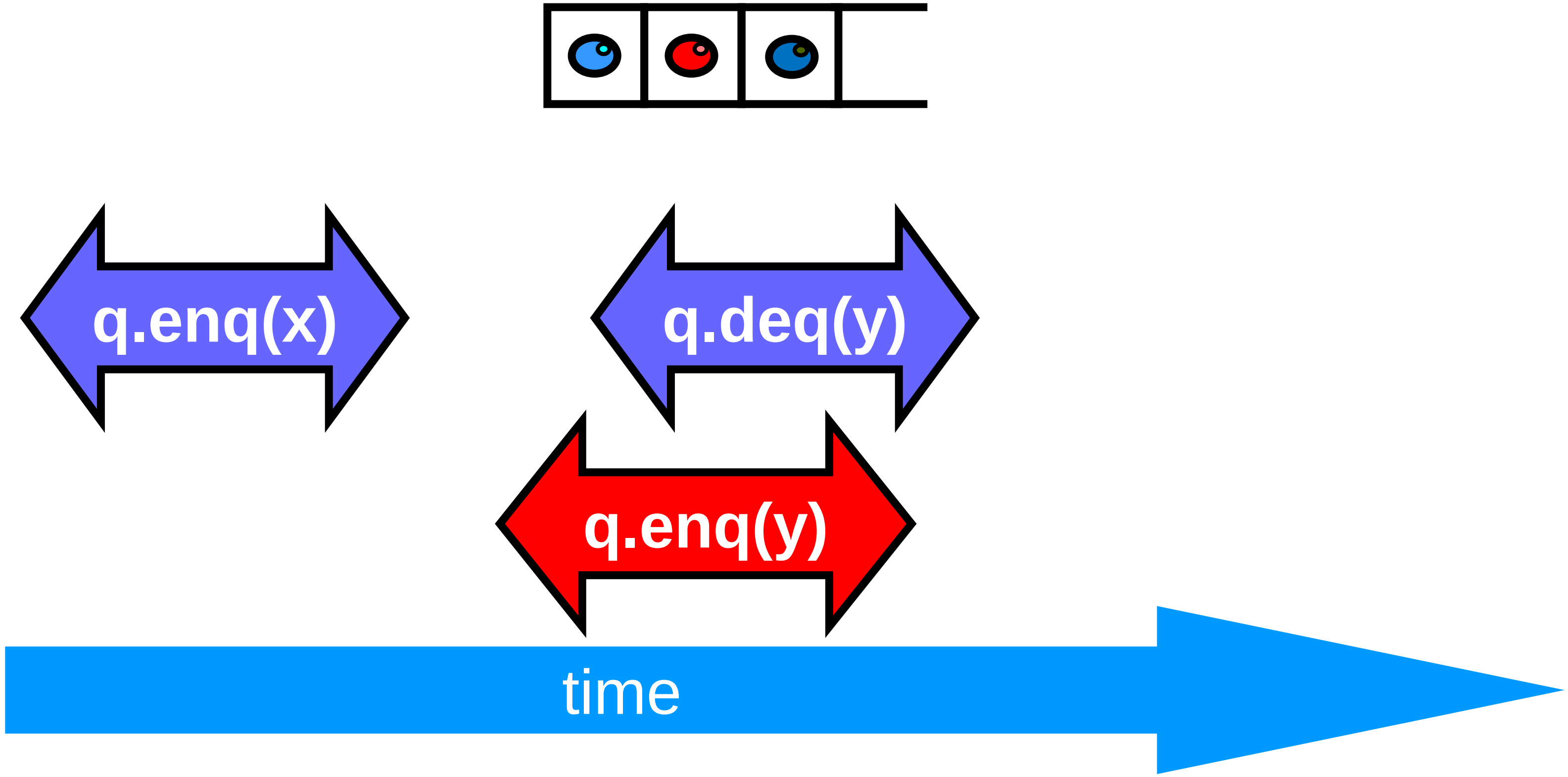
Example



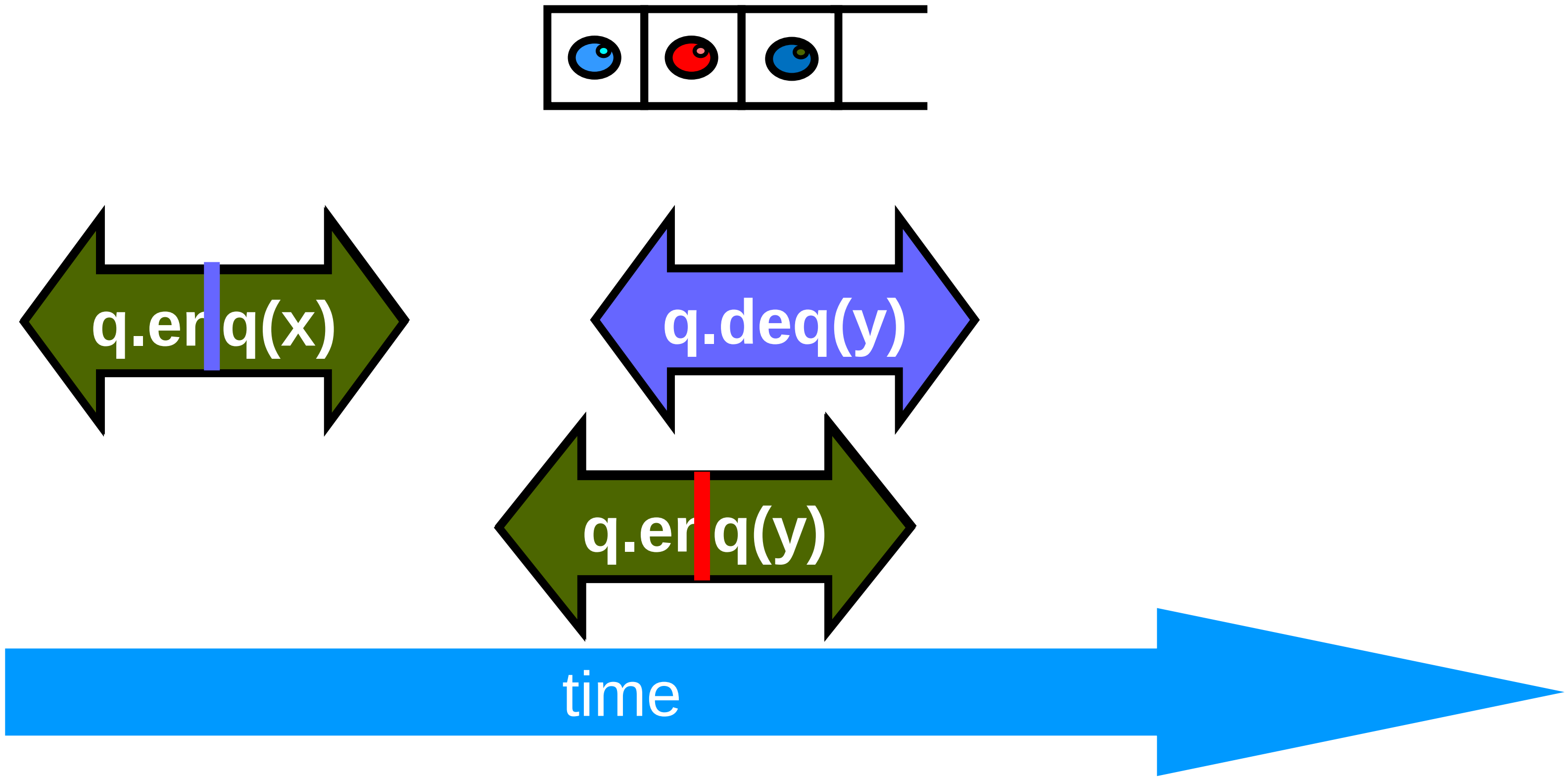
Example



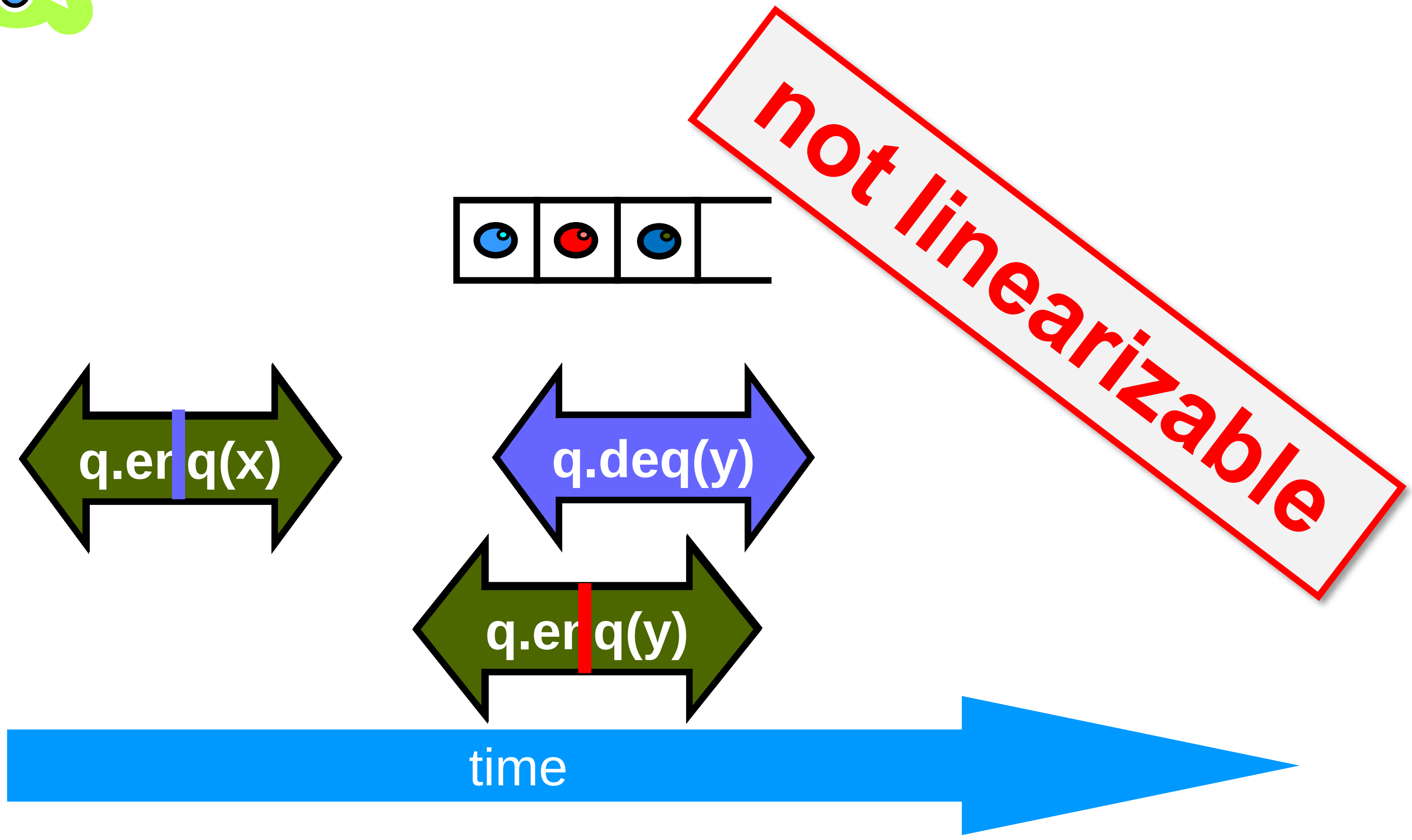
Example



Example

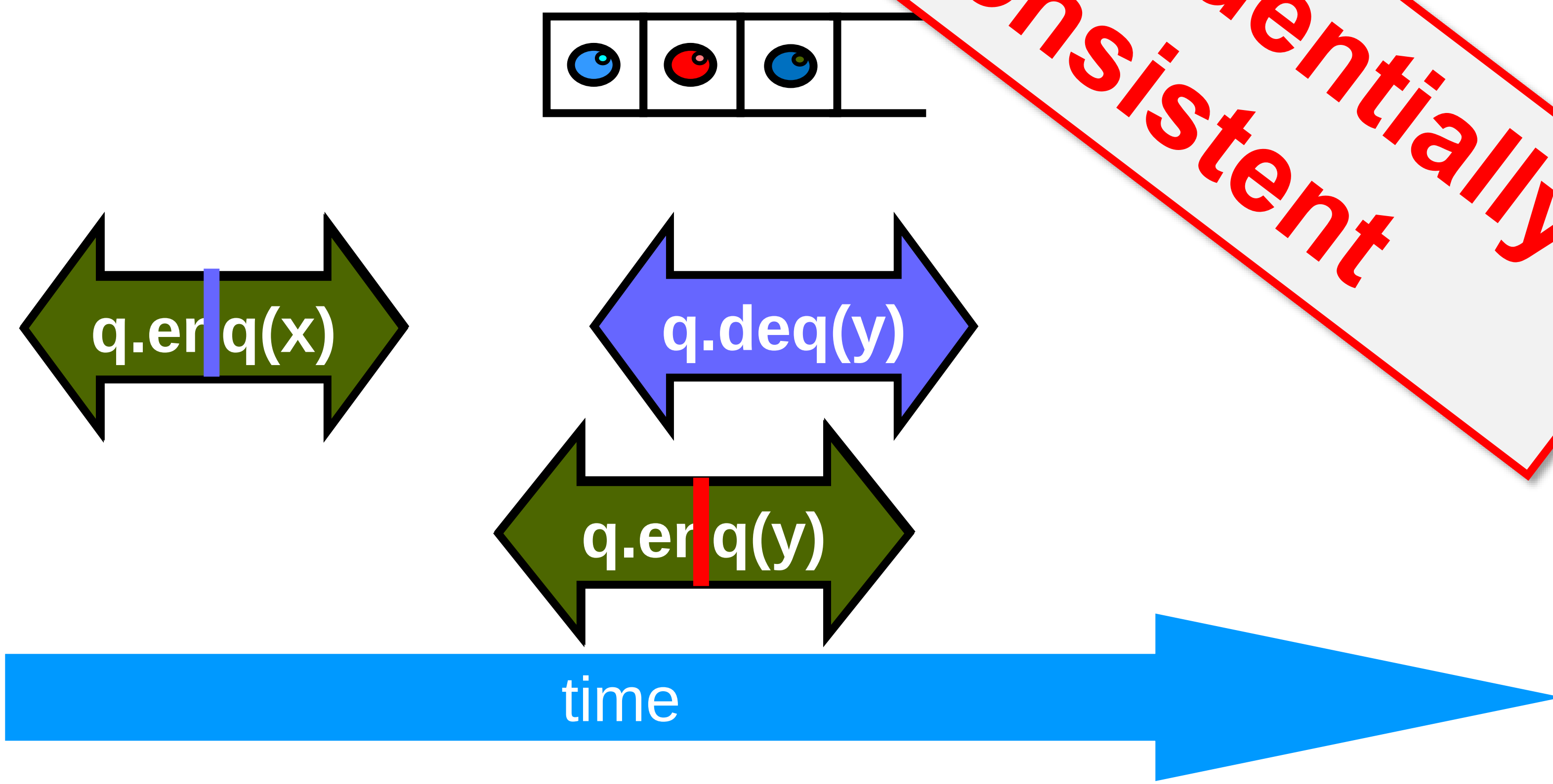


Example



Exa

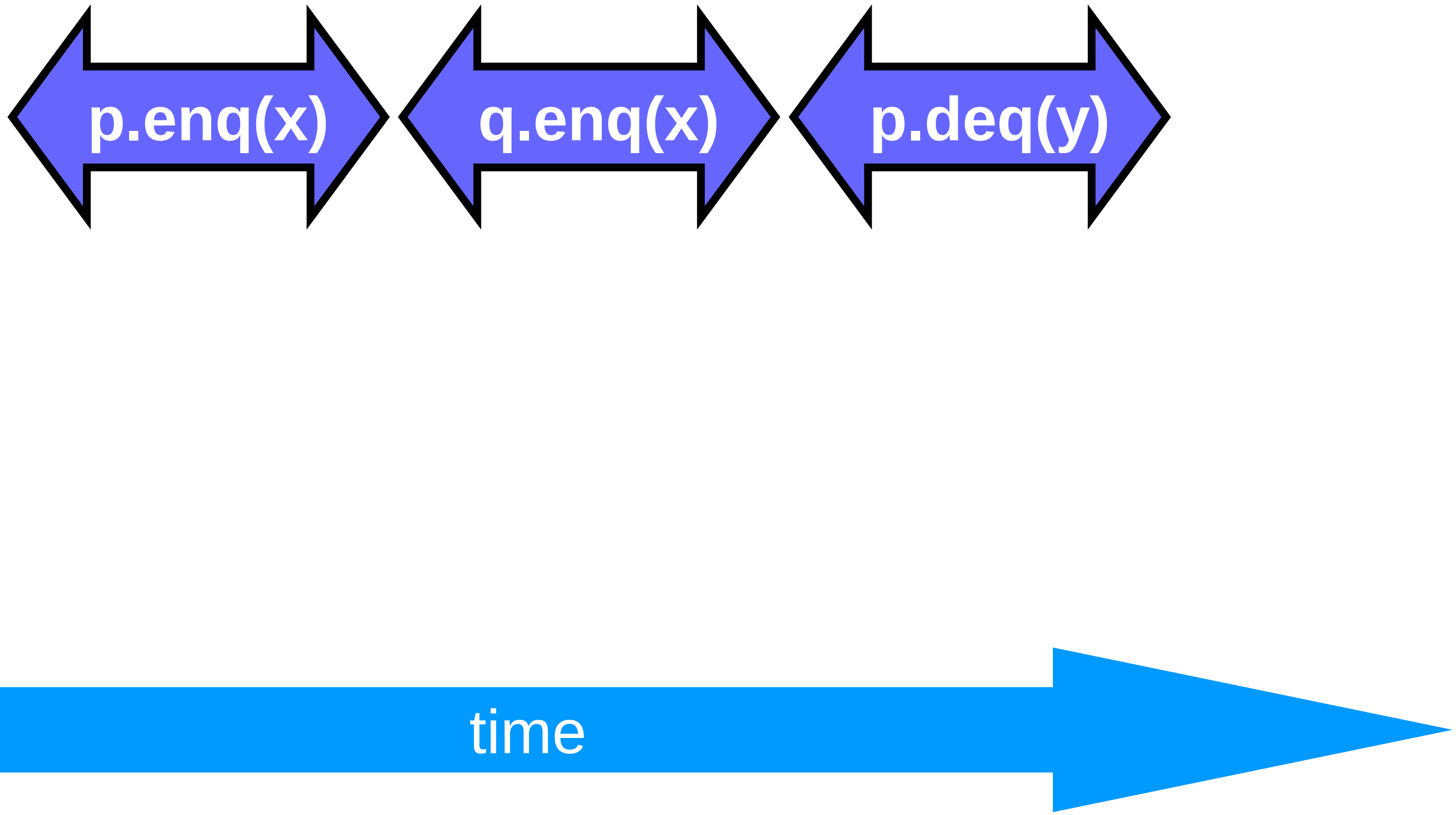
Yet Sequentially Consistent



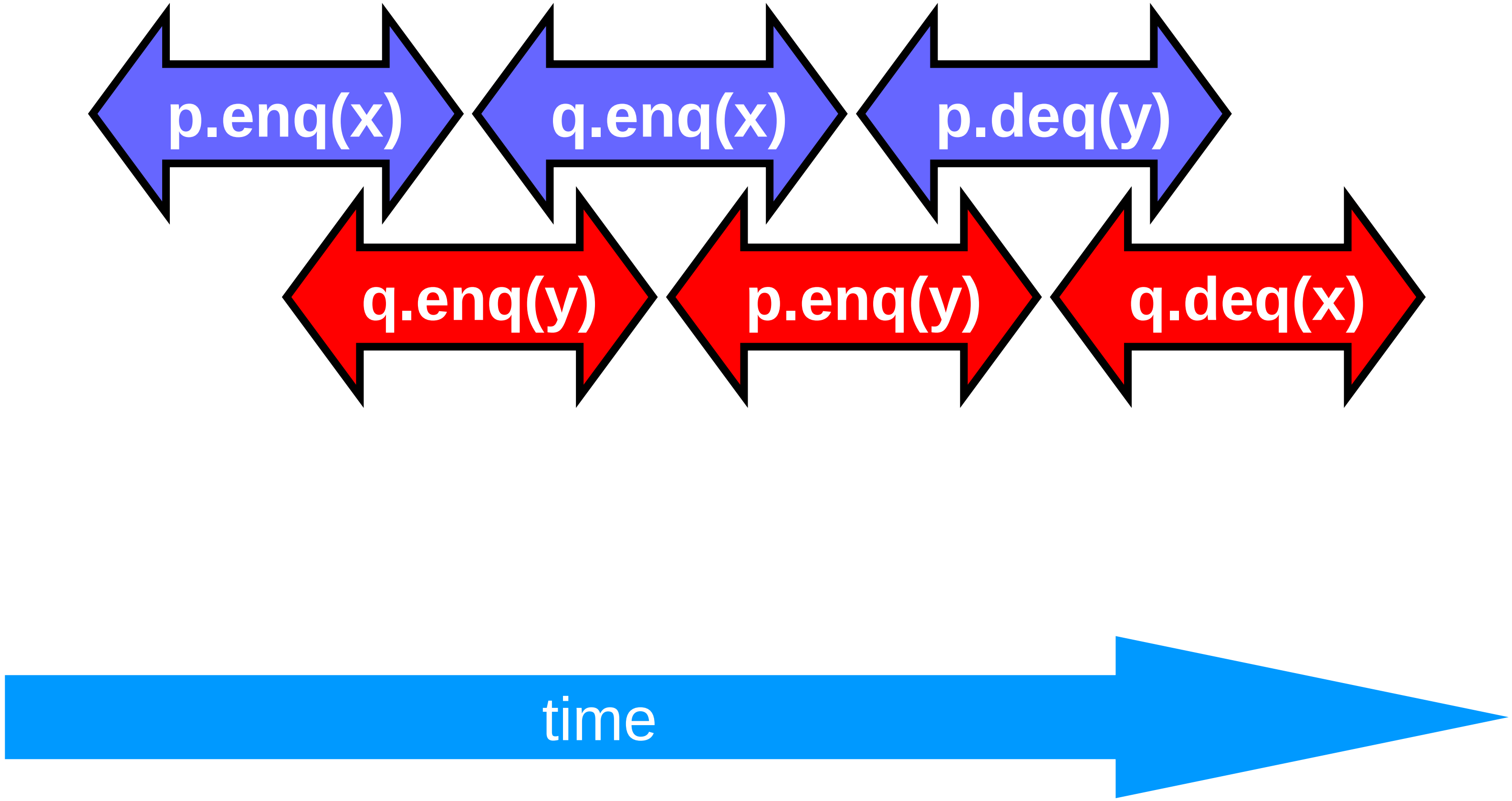
Theorem

Sequential Consistency is not composable

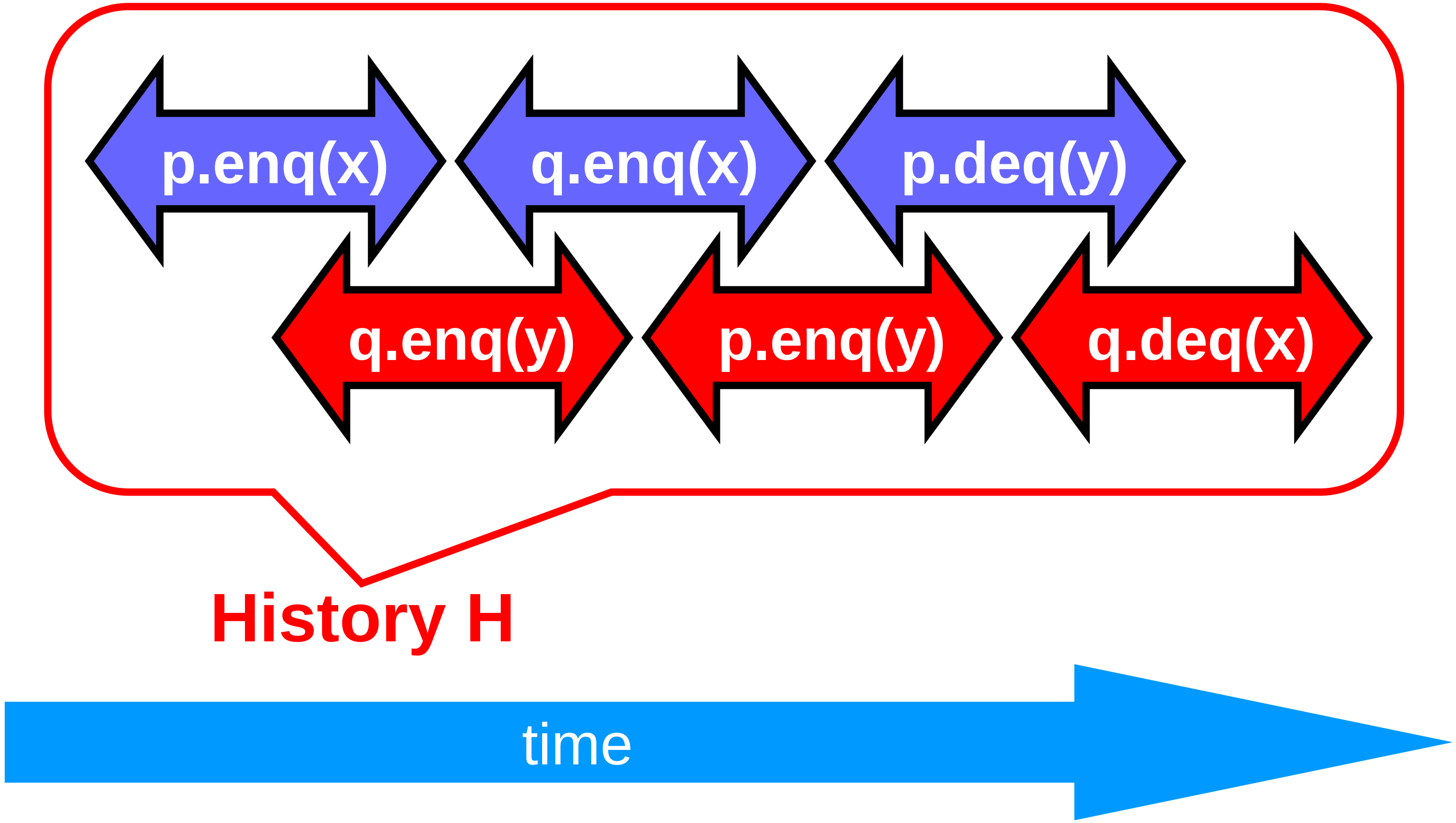
FIFO Queue Example



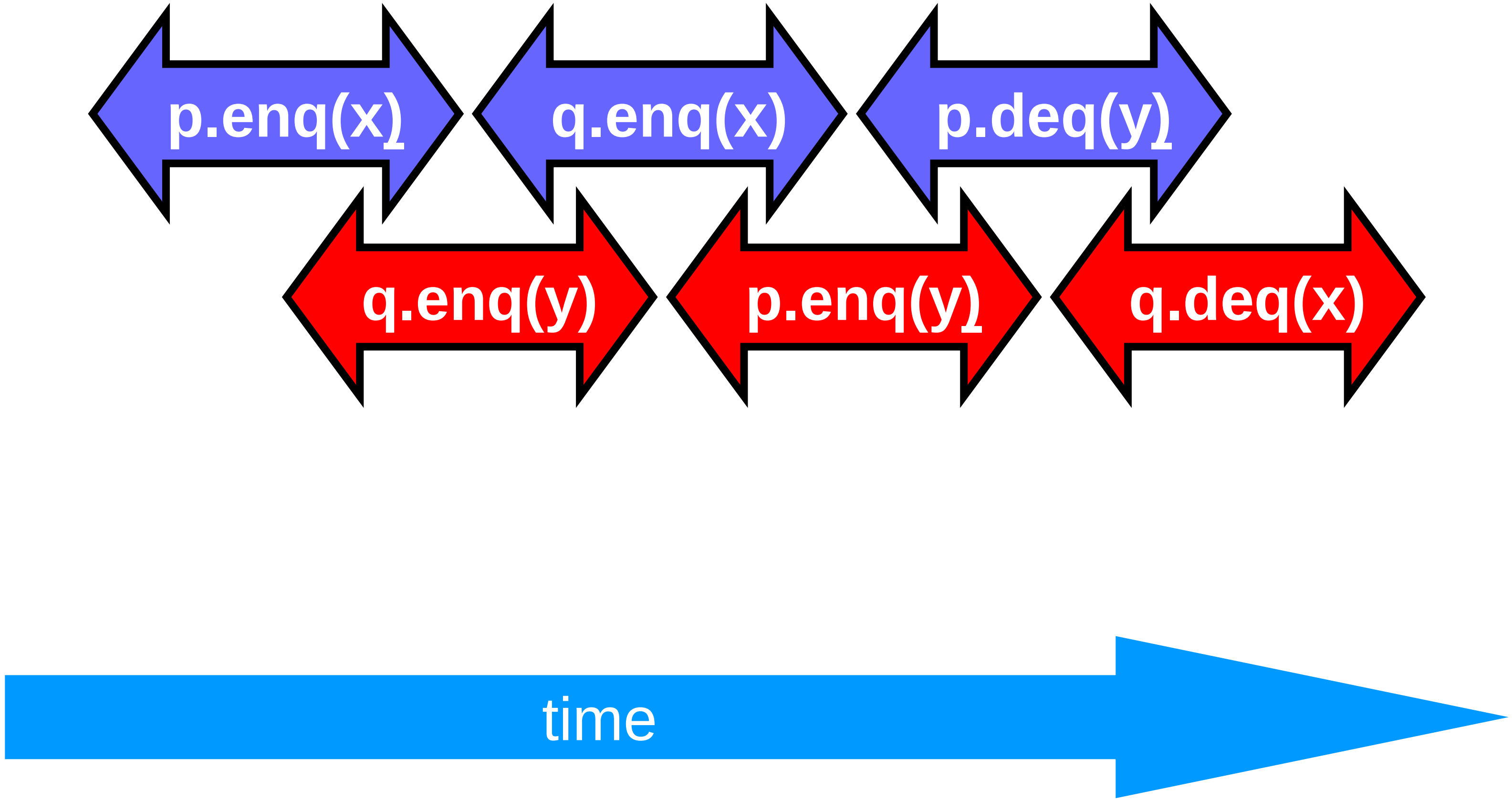
FIFO Queue Example

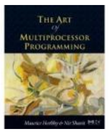


FIFO Queue Example

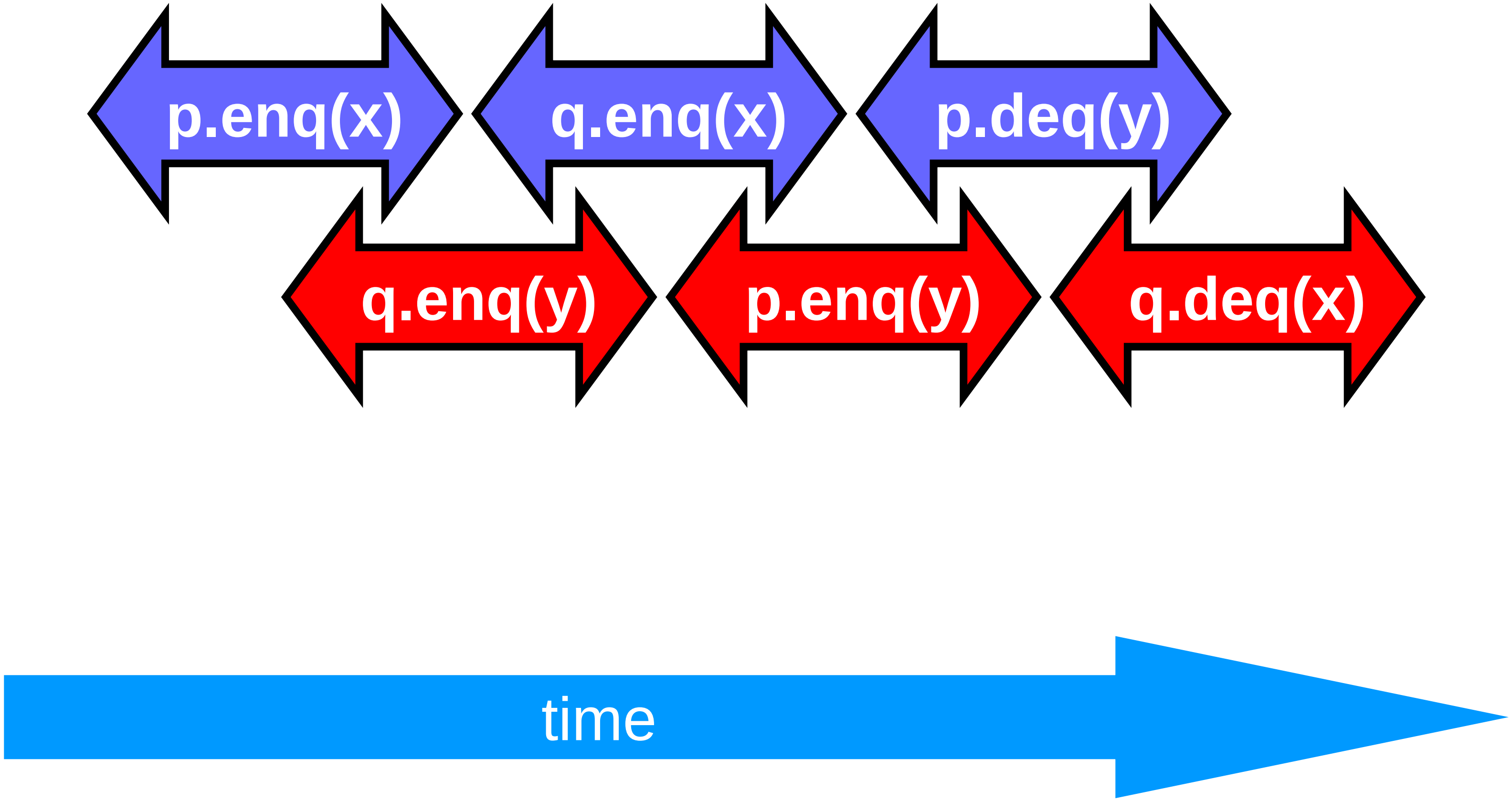


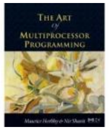
H|p Sequentially Consistent



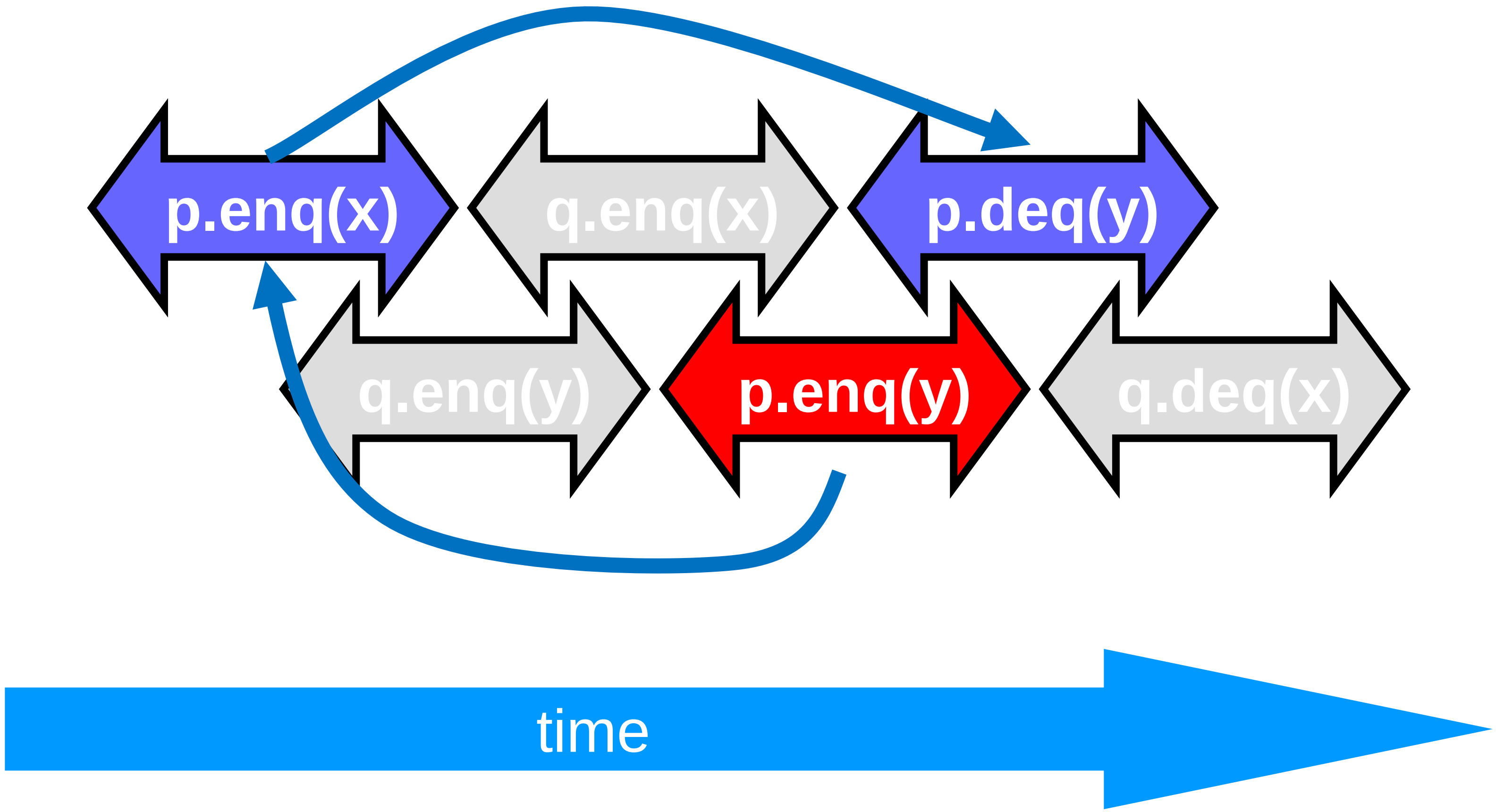


H|q Sequentially Consistent

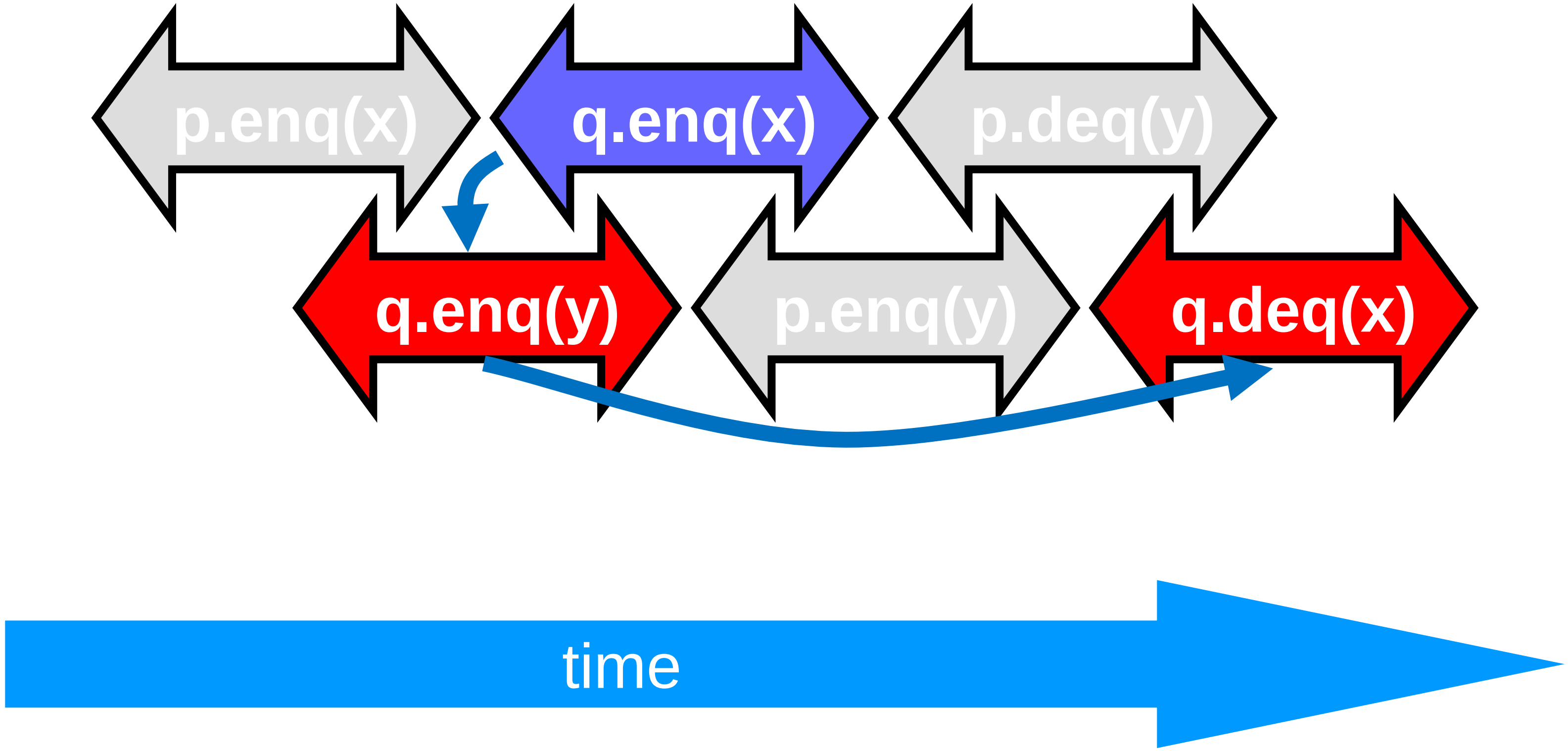


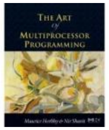


Ordering imposed by p

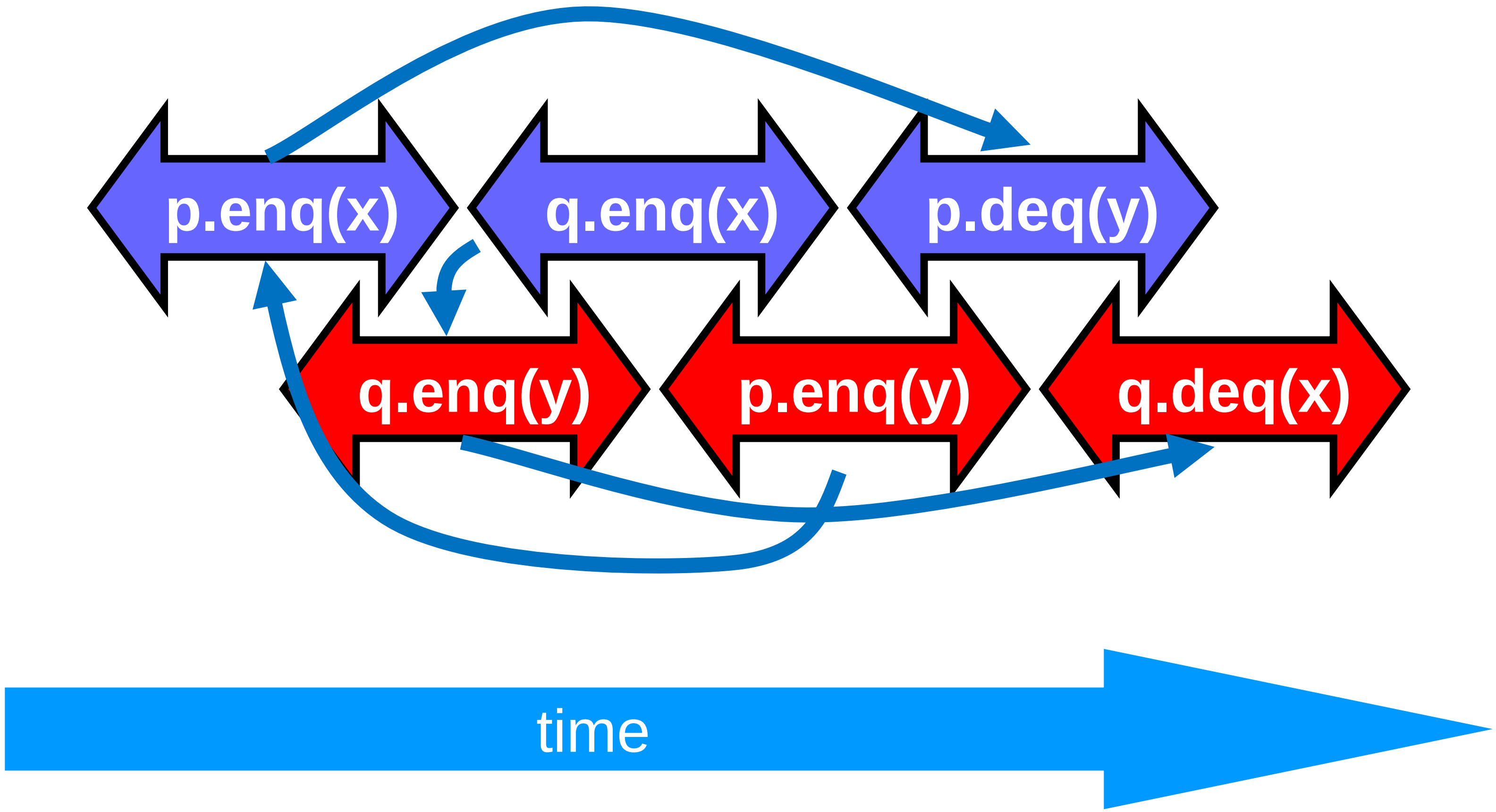


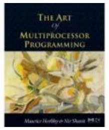
Ordering imposed by q



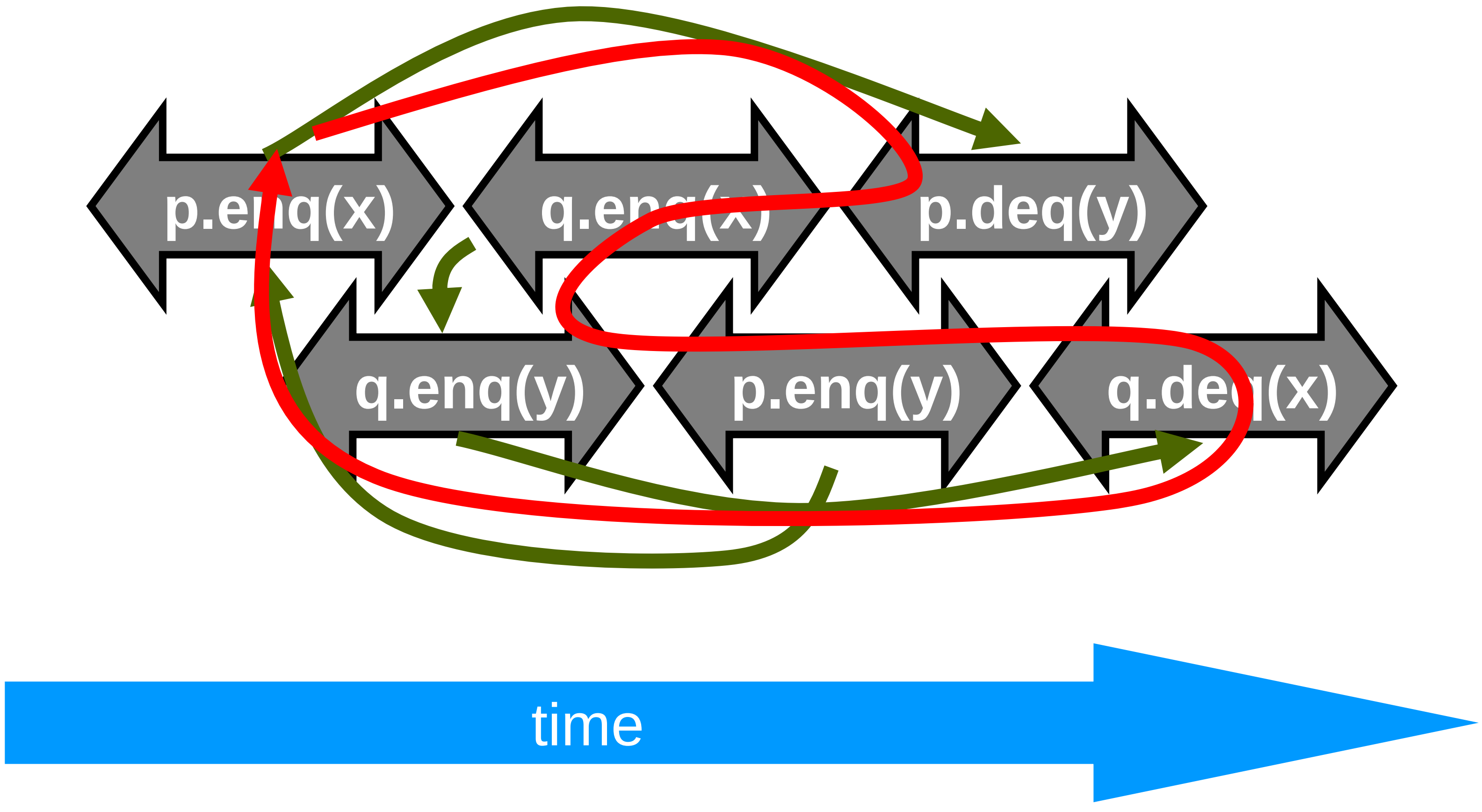


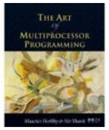
Ordering imposed by both





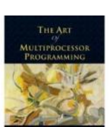
Combining orders



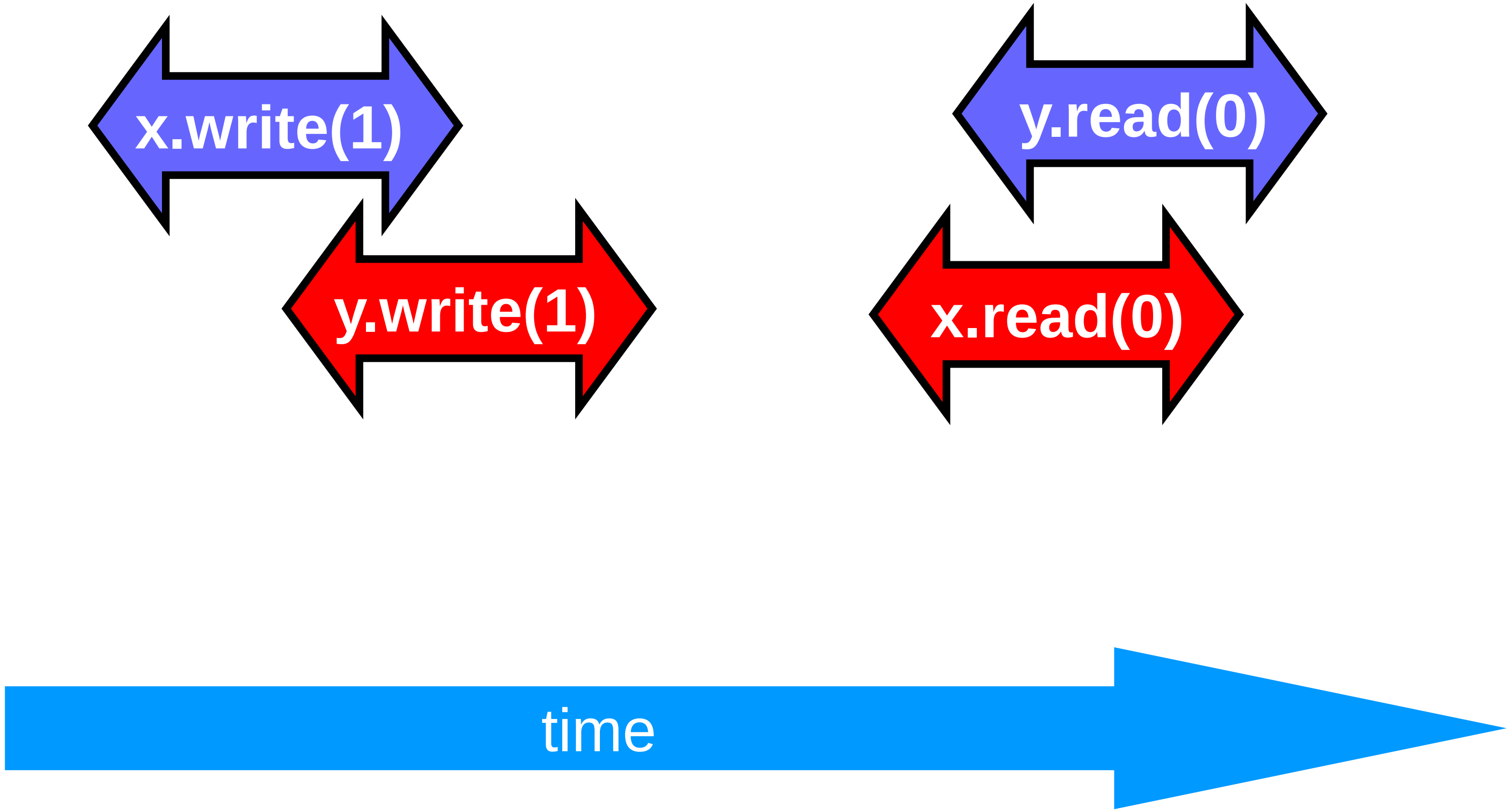


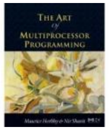
Fact

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

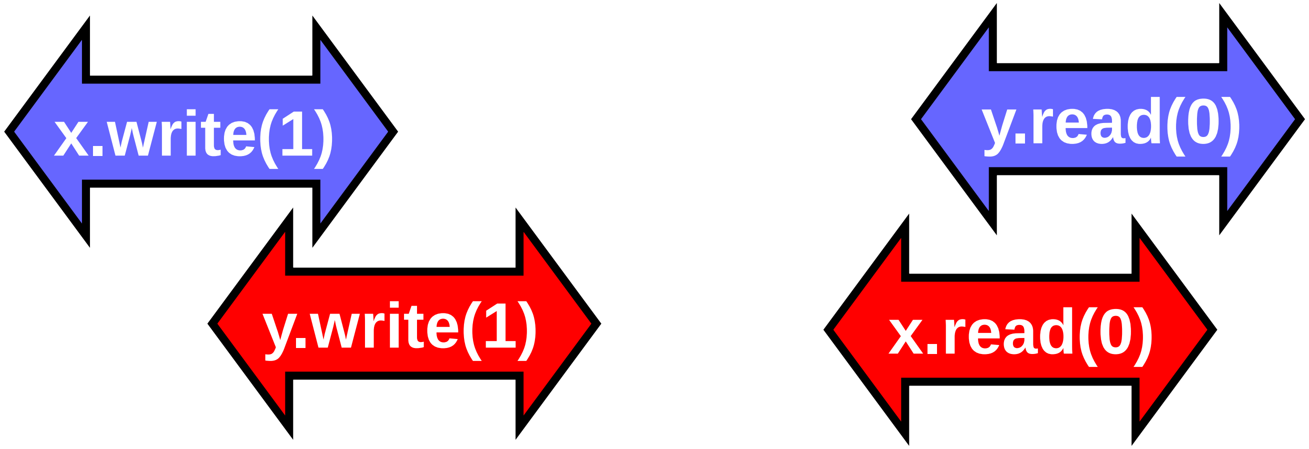


The Flag Example



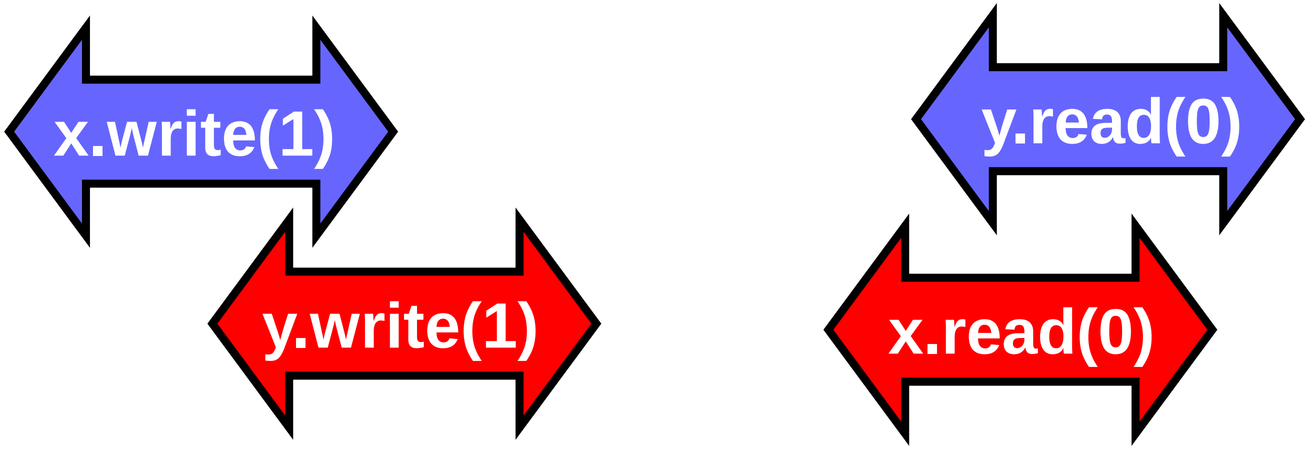


The Flag Example

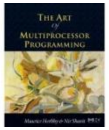


- Each thread's view is sequentially consistent
  - It went first

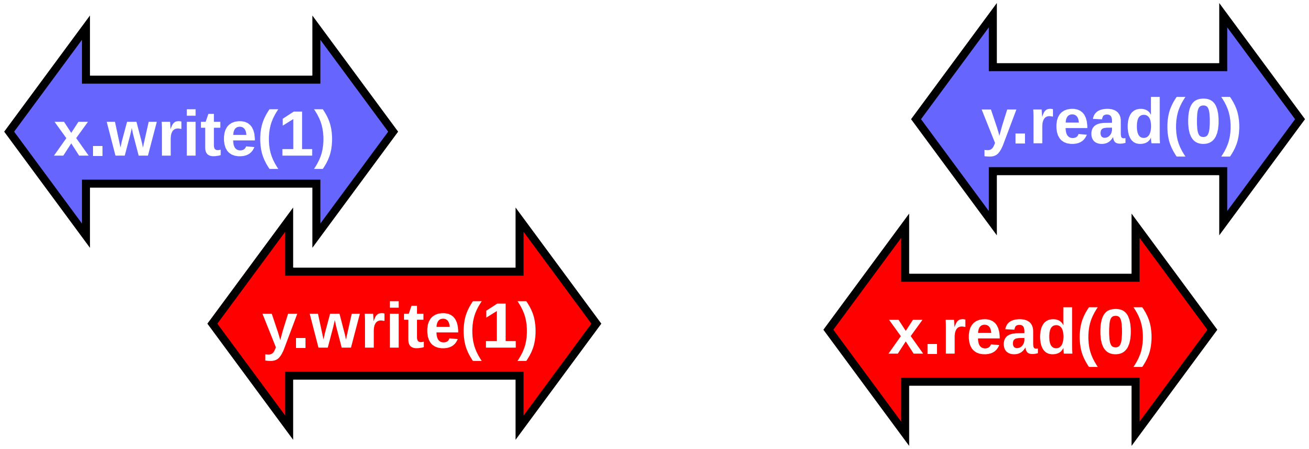
The Flag Example



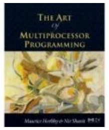
- Entire history isn't sequentially consistent
  - Can't both go first



The Flag Example

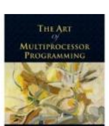


- Is this behavior really so wrong?
  - We can argue either way …



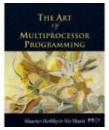
Opinion: It’s Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It’s non-negotiable!



Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

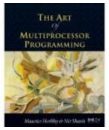


Crux of Peterson Proof

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

(2) $\text{write}_A(\text{victim=A}) \rightarrow \text{read}_A(\text{flag[B]})$
$\rightarrow \text{read}_A(\text{victim})$



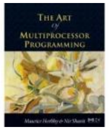
Crux of Peterson Proof

(1) $\boxed{\text{write}_B(\text{flag[B]=true})} \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

(2) $\text{write}_A(\text{victim=A}) \rightarrow \boxed{\text{read}_A(\text{flag[B]})}$
$\rightarrow \text{read}_A(\text{victim})$

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



Opinion: But It Feels So Right …

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

Hardware Consistancy

Initially, a = b = 0.

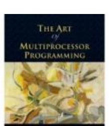
Processor 0

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

Processor 1

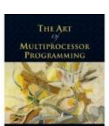
```
mov 1, b      ;Store
mov a, %eax   ;Load
```

- What are the final possible values of %eax and %ebx after both processors have executed?
- Sequential consistency implies that no execution ends with %eax= %ebx = 0



Hardware Consistency

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



Instruction Reordering

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

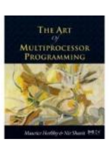
Program Order

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

Execution Order

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

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



Slide used with permission of
Charles E. Leiserson

Instruction Reordering

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

Program Order

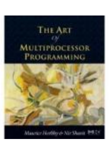
```
mov b, %ebx   ;Load
mov 1, a      ;Store
```

Execution Order

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

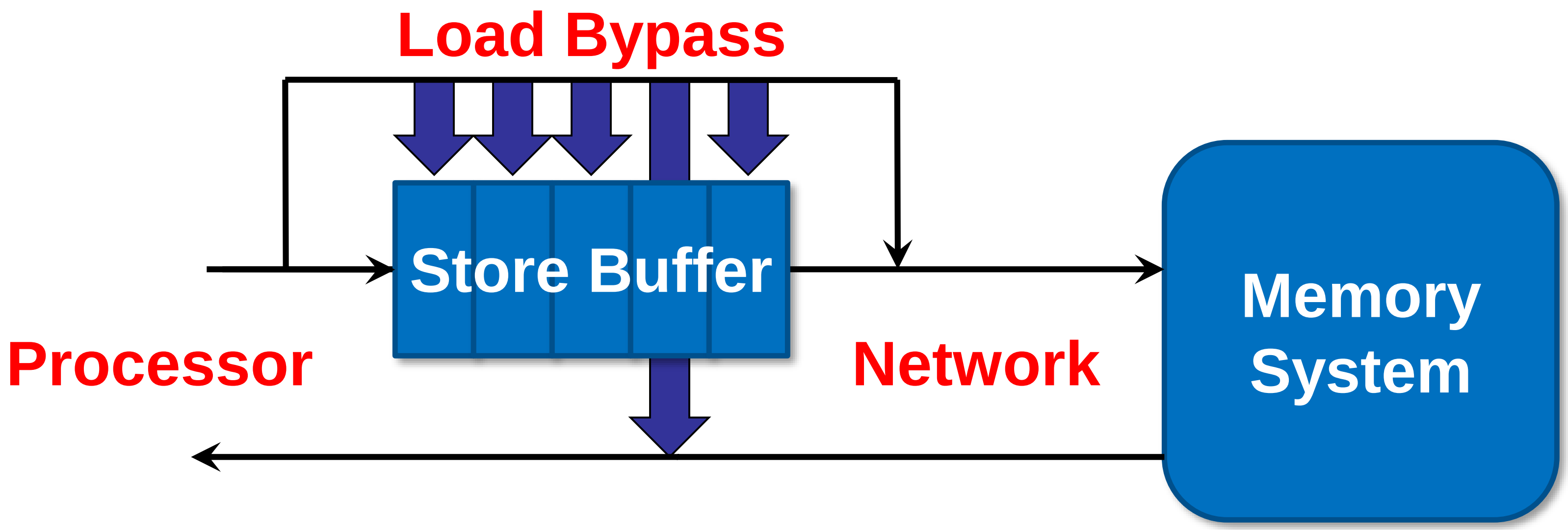
A. When a ≠ b.

A′. And there's no concurrency.

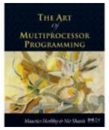


Slide used with permission of
Charles E. Leiserson

Hardware Reordering



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



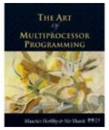
Slide used with permission of
Charles E. Leiserson

X86: Memory Consistency

Thread's Code

~~Store1~~
~~Store2~~
Load1
~~Load2~~
~~Store3~~
Store4
Load3
~~Load4~~
~~Load5~~

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



X86: Memory Consistency

Thread's Code

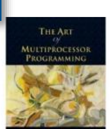
```
Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5
```

LOADS

1. Loads are *not* reordered with loads.
2. Stores are *not* reordered with stores.
3. ... or
4. ... prior ... *not* with a prior store to the same location.
5. Stores to the same location respect a global total order.

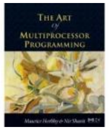
Total Store Ordering (TSO)…weaker than sequential consistency

OK!



Memory Barriers (Fences)

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



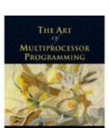
X86: Memory Consistency

Thread's Code

```
Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5
```

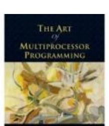
1. Loads **are *not* reordered with** loads.
2. Sto
3. Sto
   loa
4. A lo
   store to … location but not
   with a prior store to the same
   location.
5. Stores to the same location respect a global total order.

Total Store Ordering + properly placed memory barriers = sequential consistency



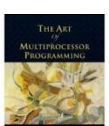
Memory Barriers

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



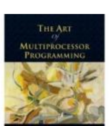
Volatile Variables

- In Java, can ask compiler to keep a variable up-to-date by declaring it volatile
- Adds a memory barrier after each store
- Inhibits reordering, removing from loops, & other “compiler optimizations”
- Will talk about it in detail in later lectures



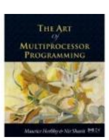
Summary: Real-World

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



Linearizability

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

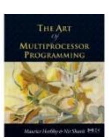


Summary: Correctness

- Sequential Consistency
  – Not composable
  – Harder to work with
  – Good way to think about hardware models
- We will use *linearizability* as our consistency condition in the remainder of this course unless stated otherwise

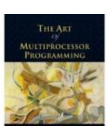
Progress

- 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?



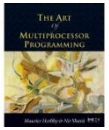
Progress Conditions

- *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.