CSE 511: Operating Systems Design

Lectures 20

Concurrent Objects

The slides are partially based on Maurice Herlihy's slides from "The Art of Multiprocessing"

Linearizability

Each method call should

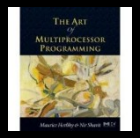
"take effect"

Instantaneously

Between invocation and response events

Object is correct if "sequential" behavior is correct

Any such concurrent object is *Linearizable*™



Linearizability

Each method call should

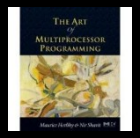
"take effect"

Instantaneously

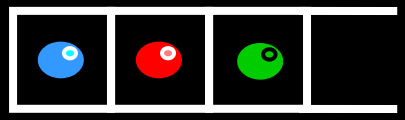
Between invocation and response events

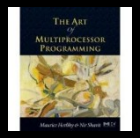
Actually a property of an execution

A *linearizable* object: one all of whose possible *executions* are linearizable

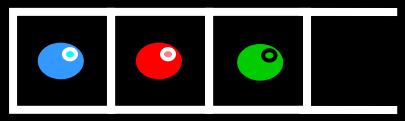


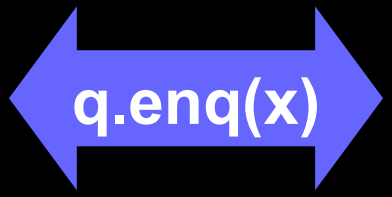
Example

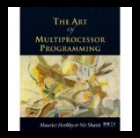




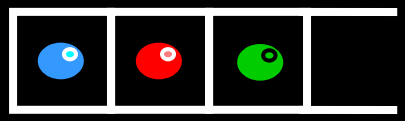
Example

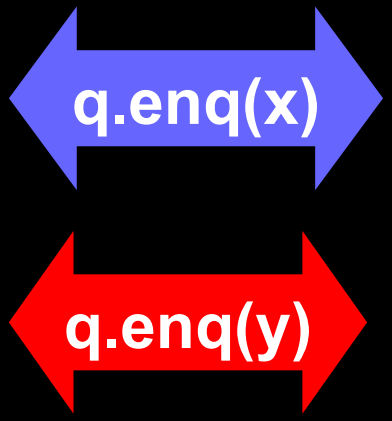


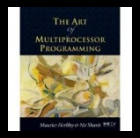




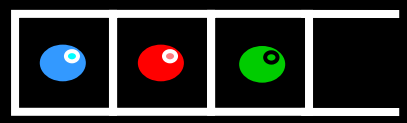
Example

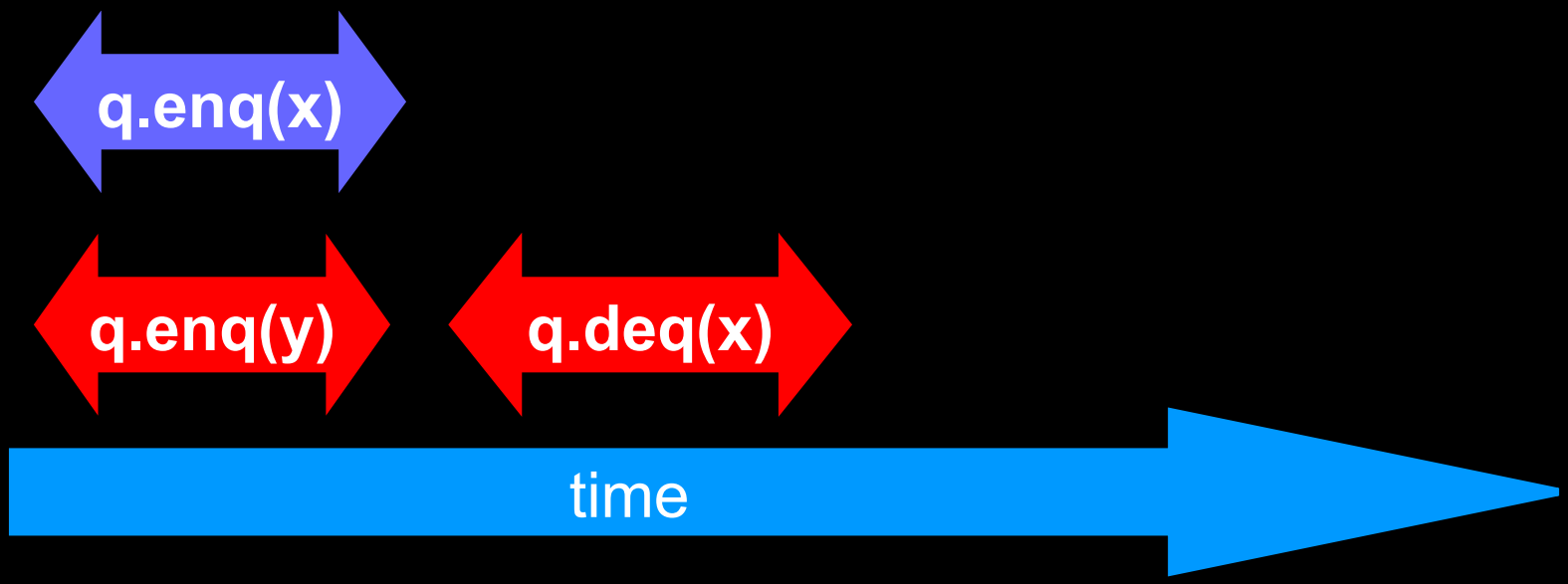


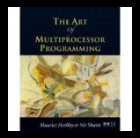




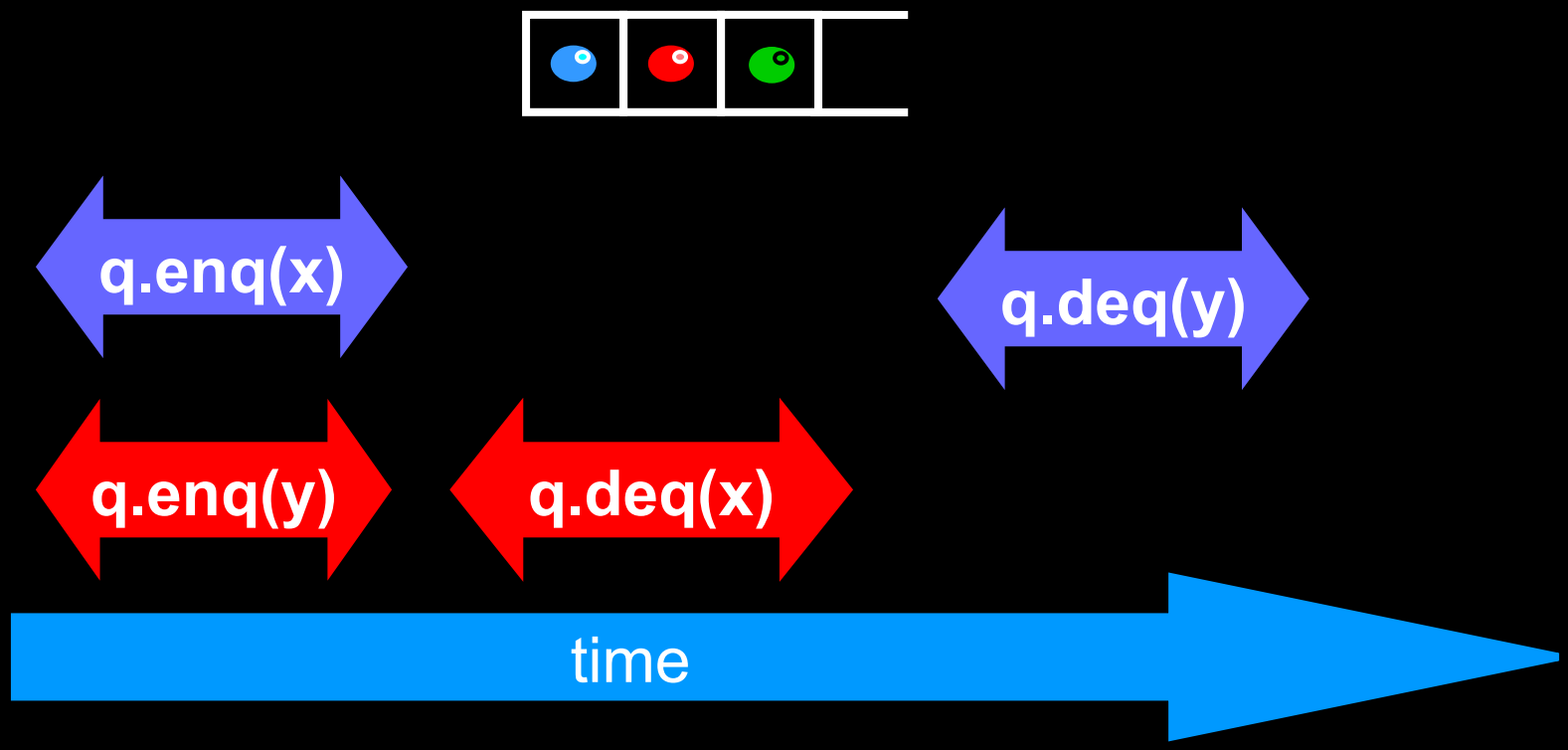
Example

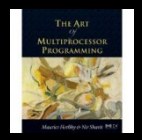



q.enq(x)
q.enq(y)
q.deq(x)
time

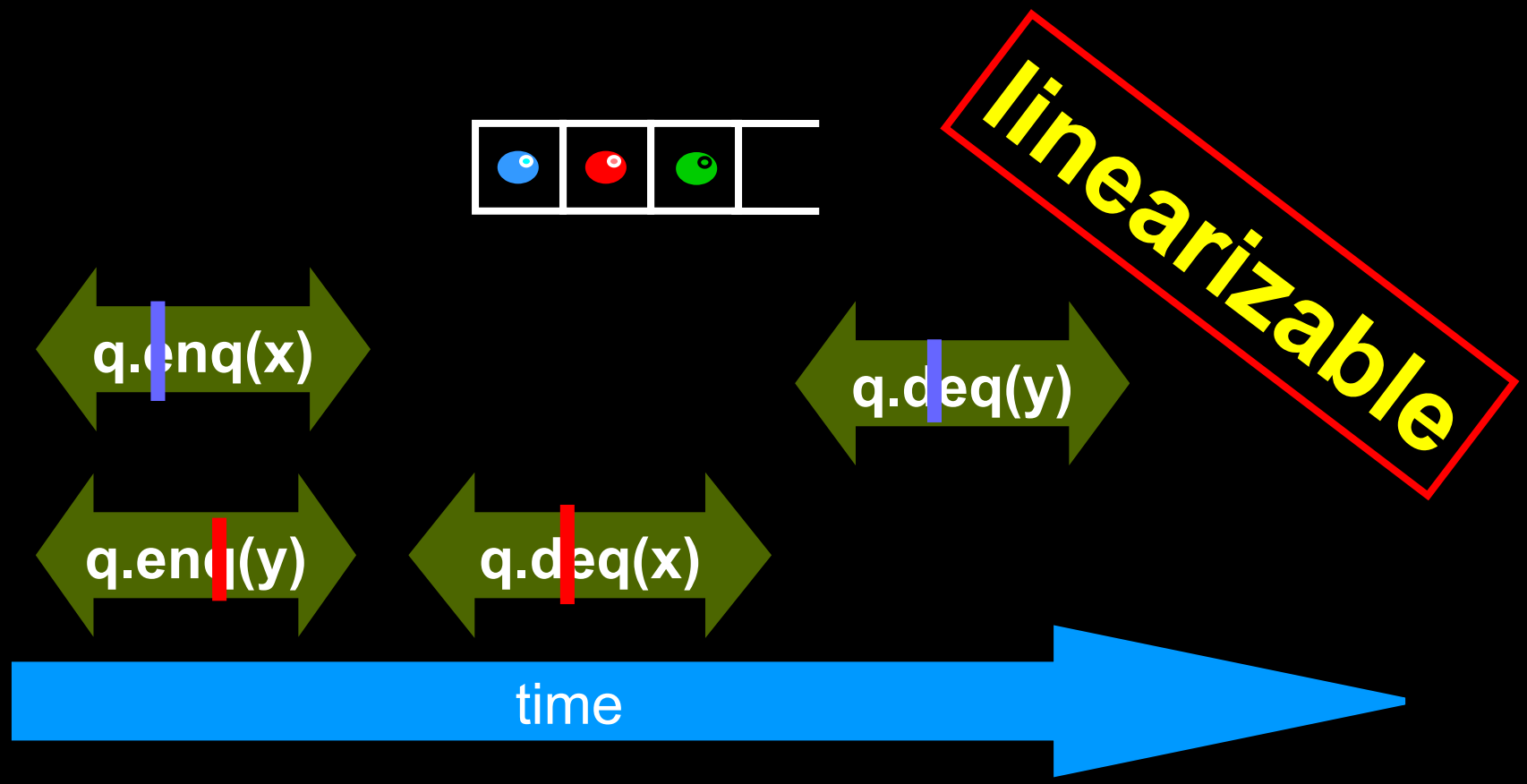

The Art of Multiprocessor Programming

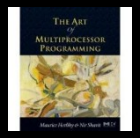
Example


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

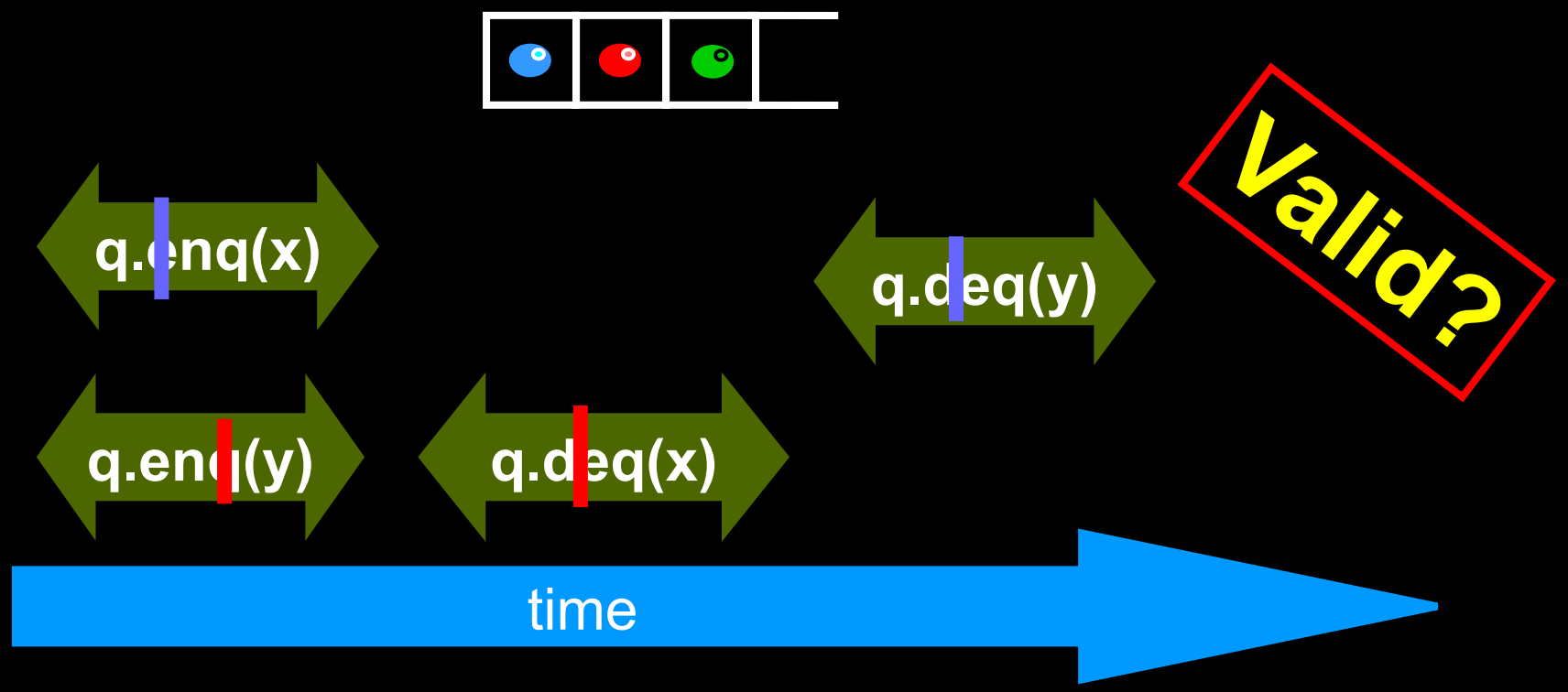

The Art of Multiprocessor Programming

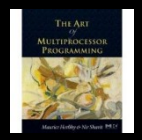
Example


linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time

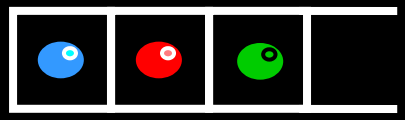

The Art of Multiprocessor Programming

Example

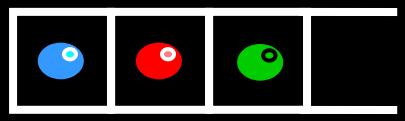


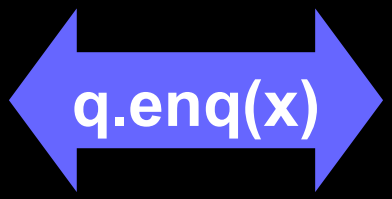


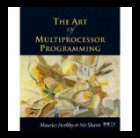
Example



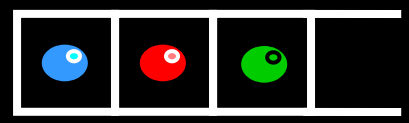
Example

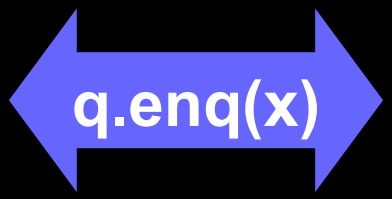


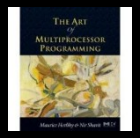




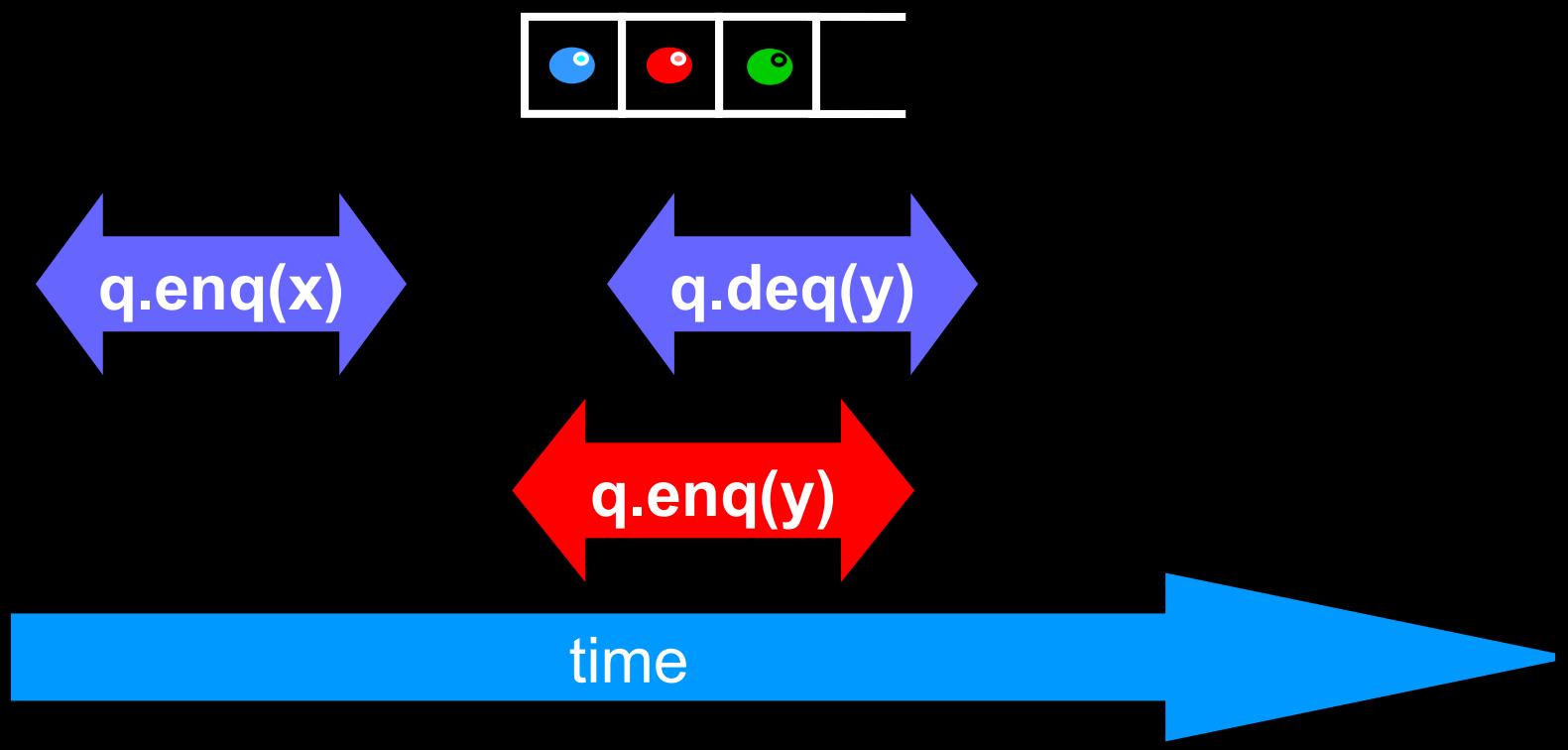
Example



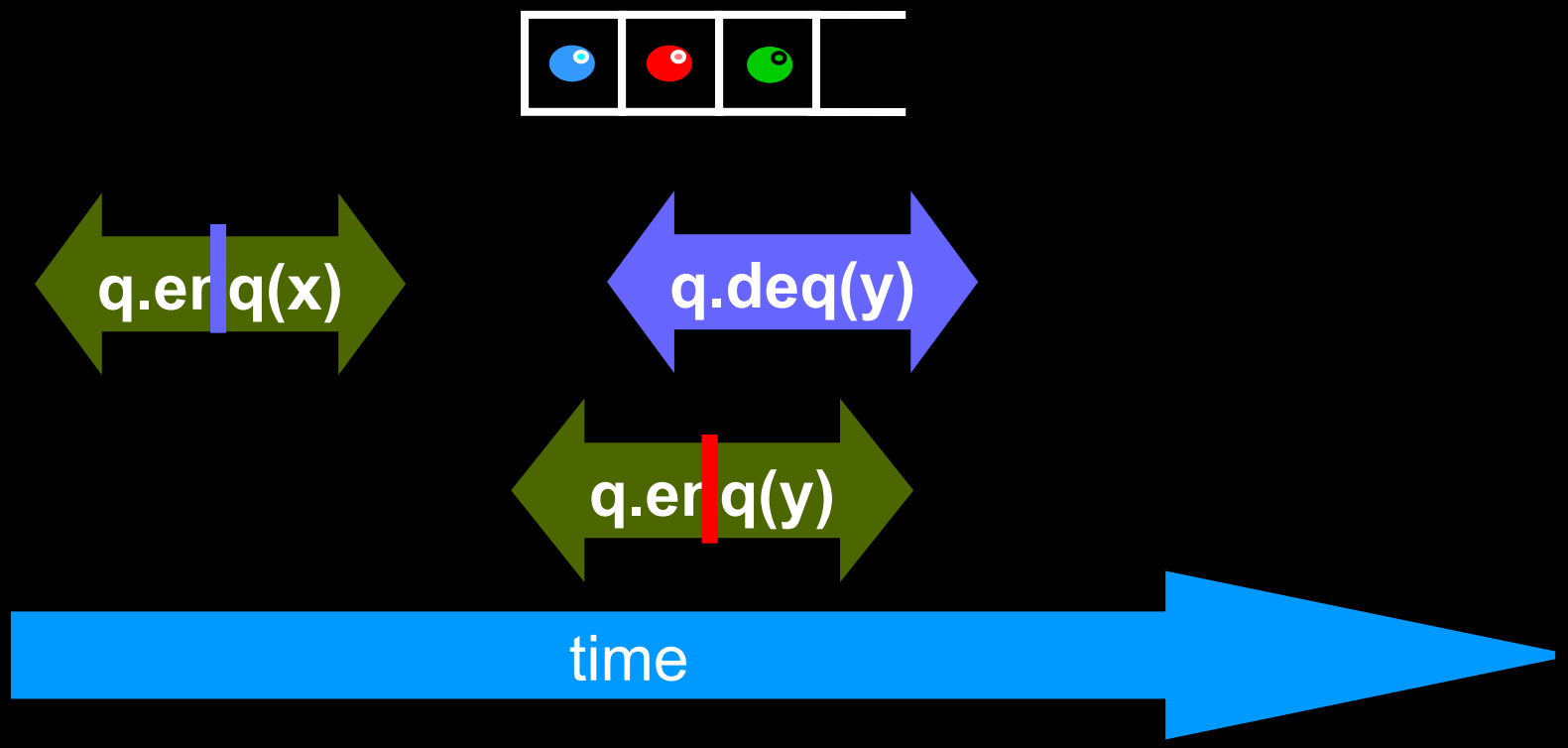




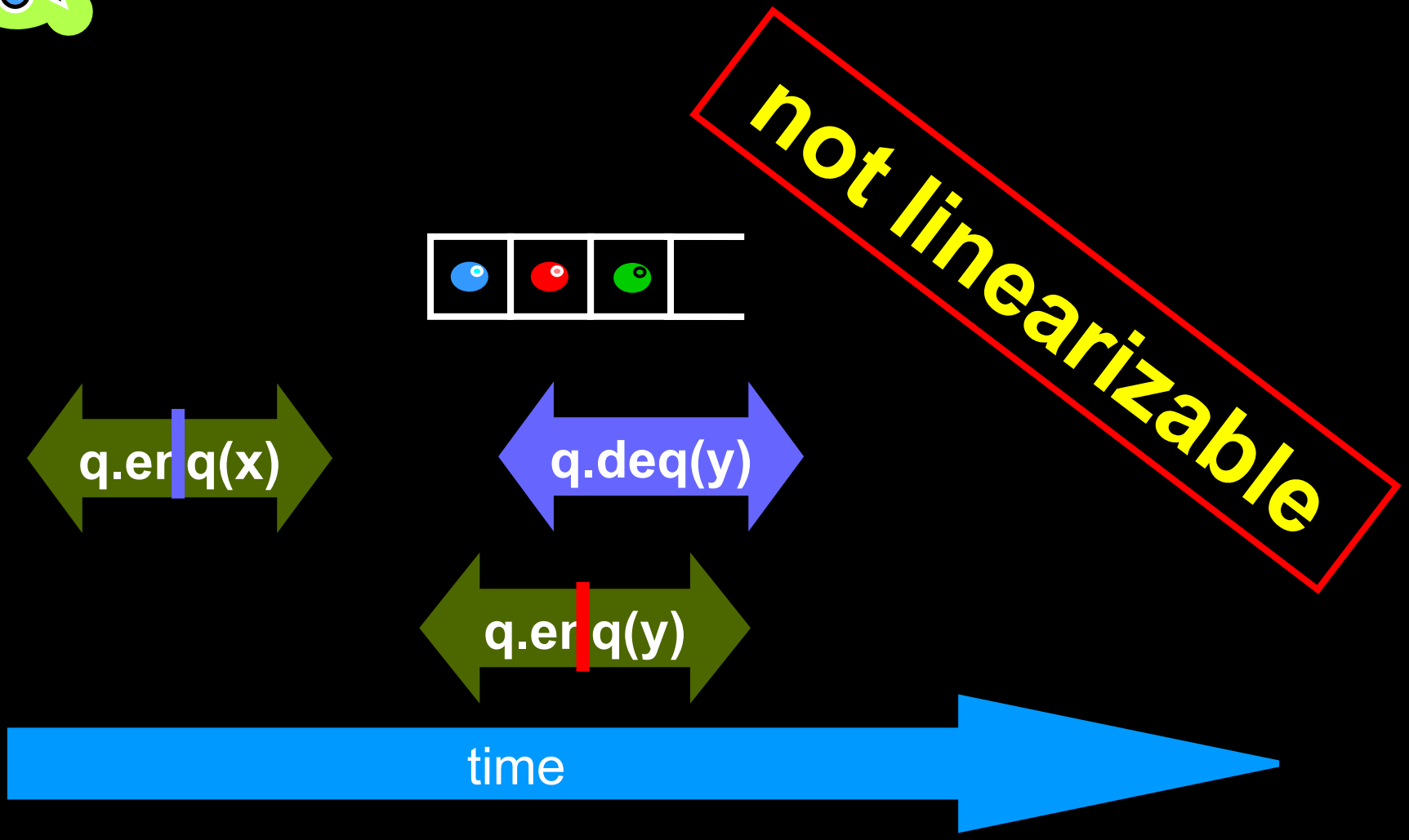
Example



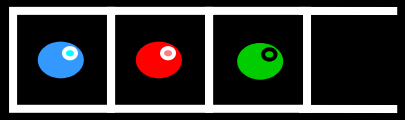
Example

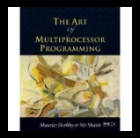


Example

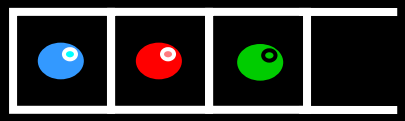


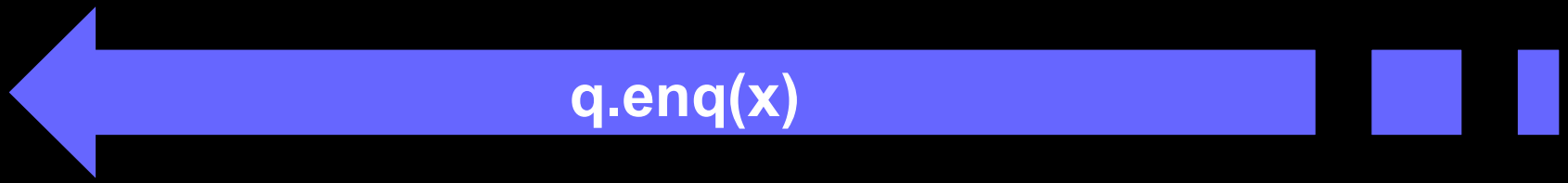
Example



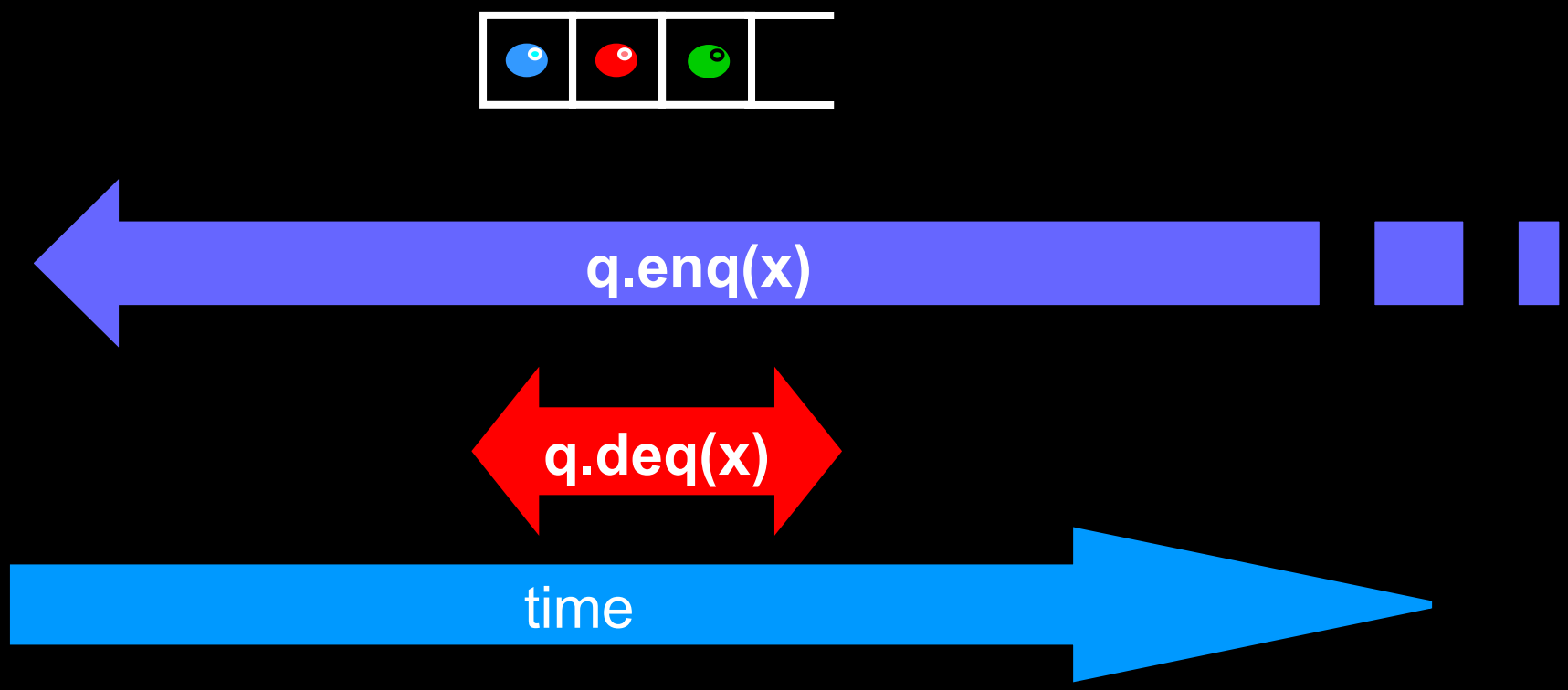


Example

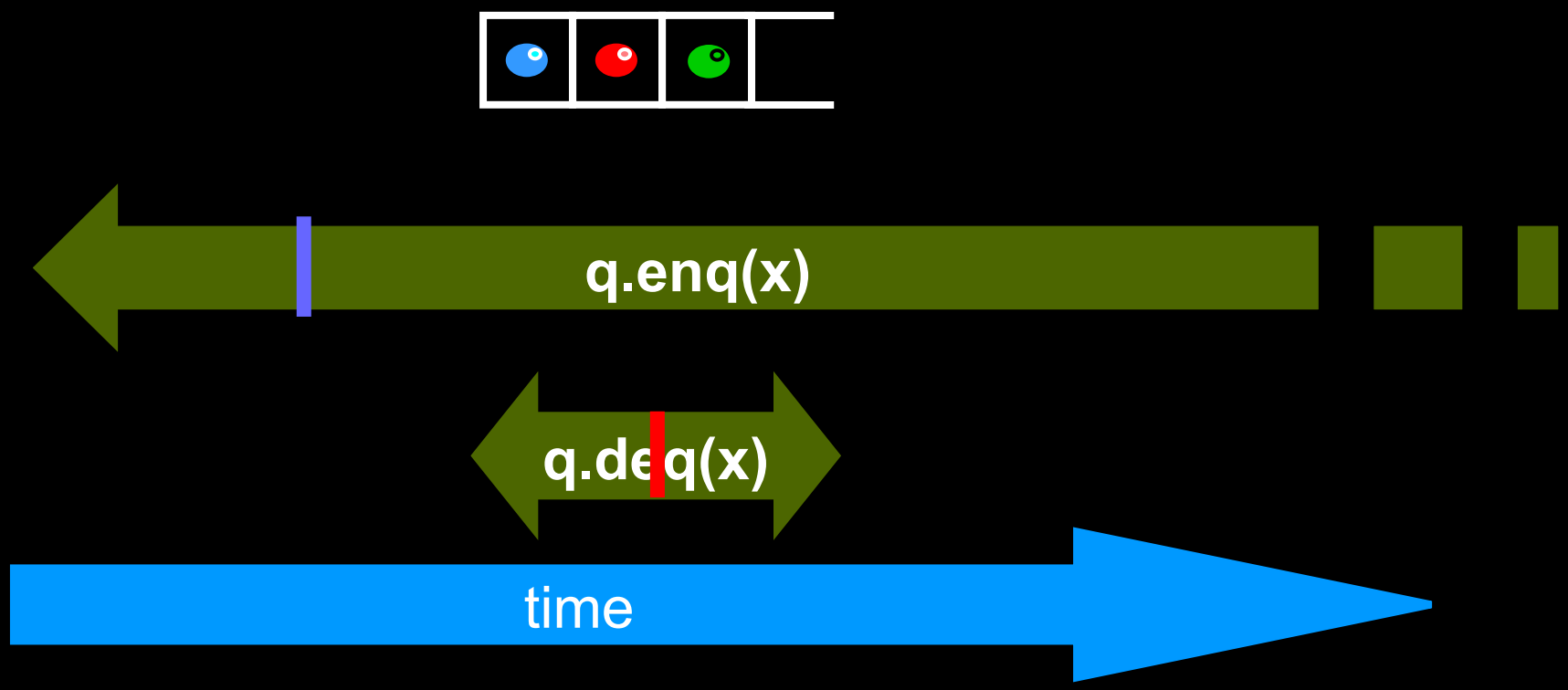




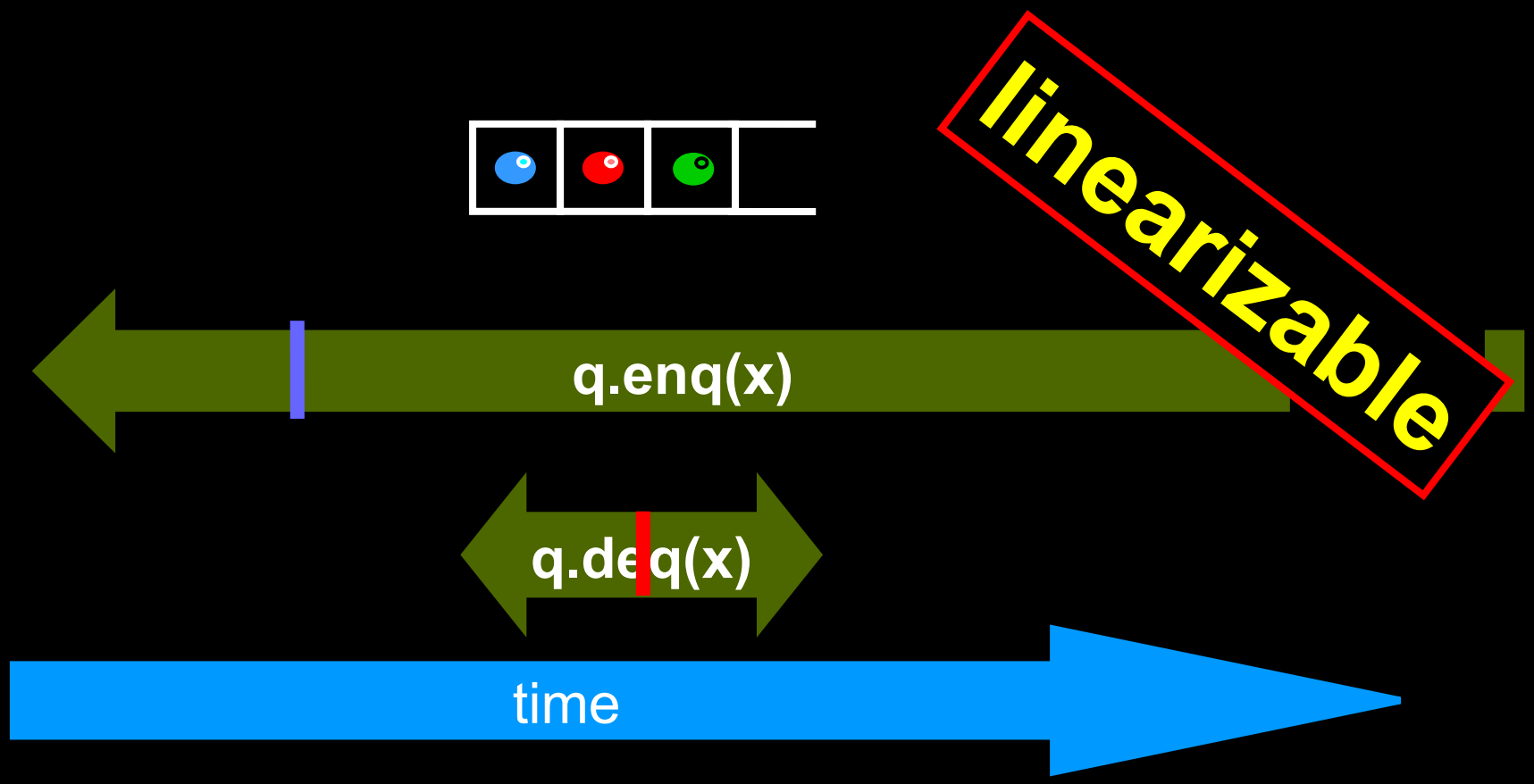
Example



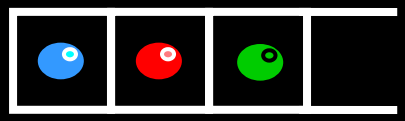
Example

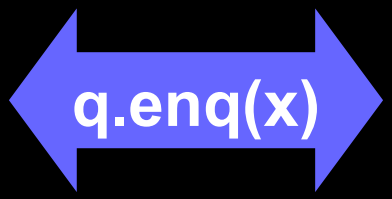


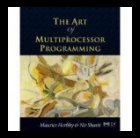
Example



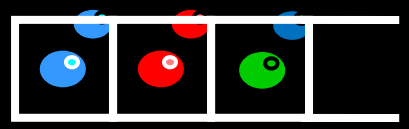
Example

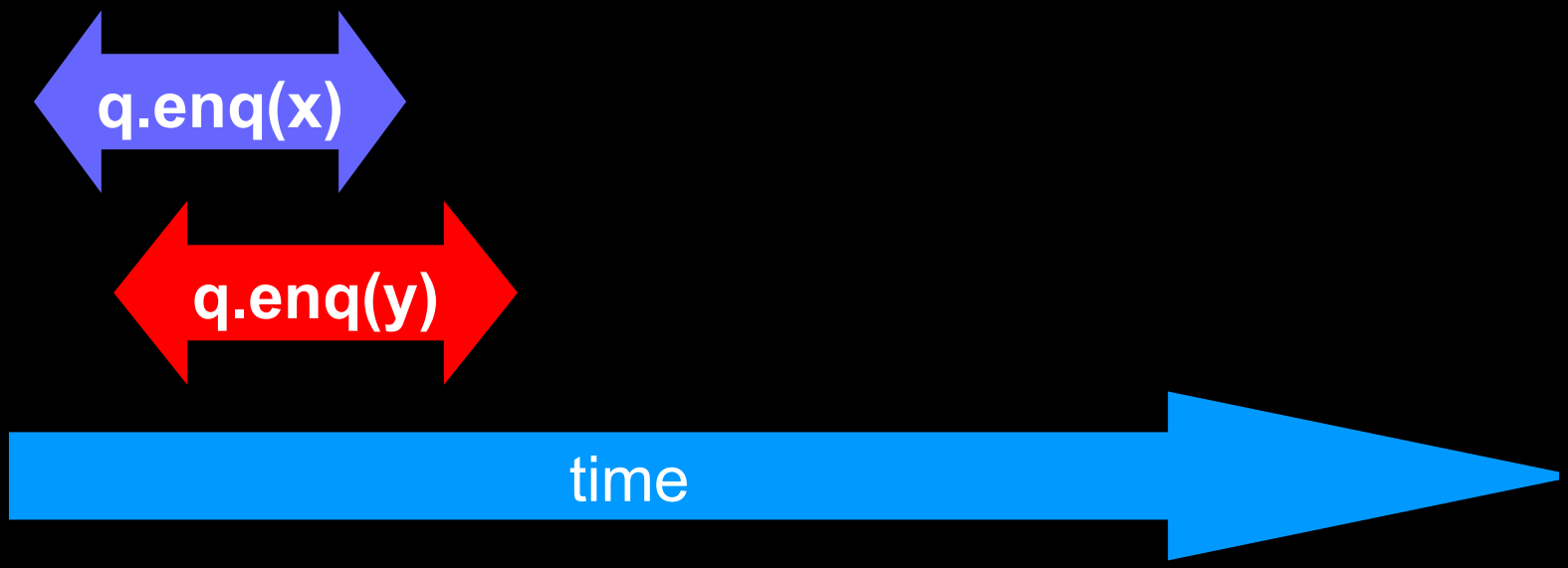


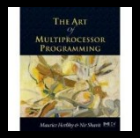




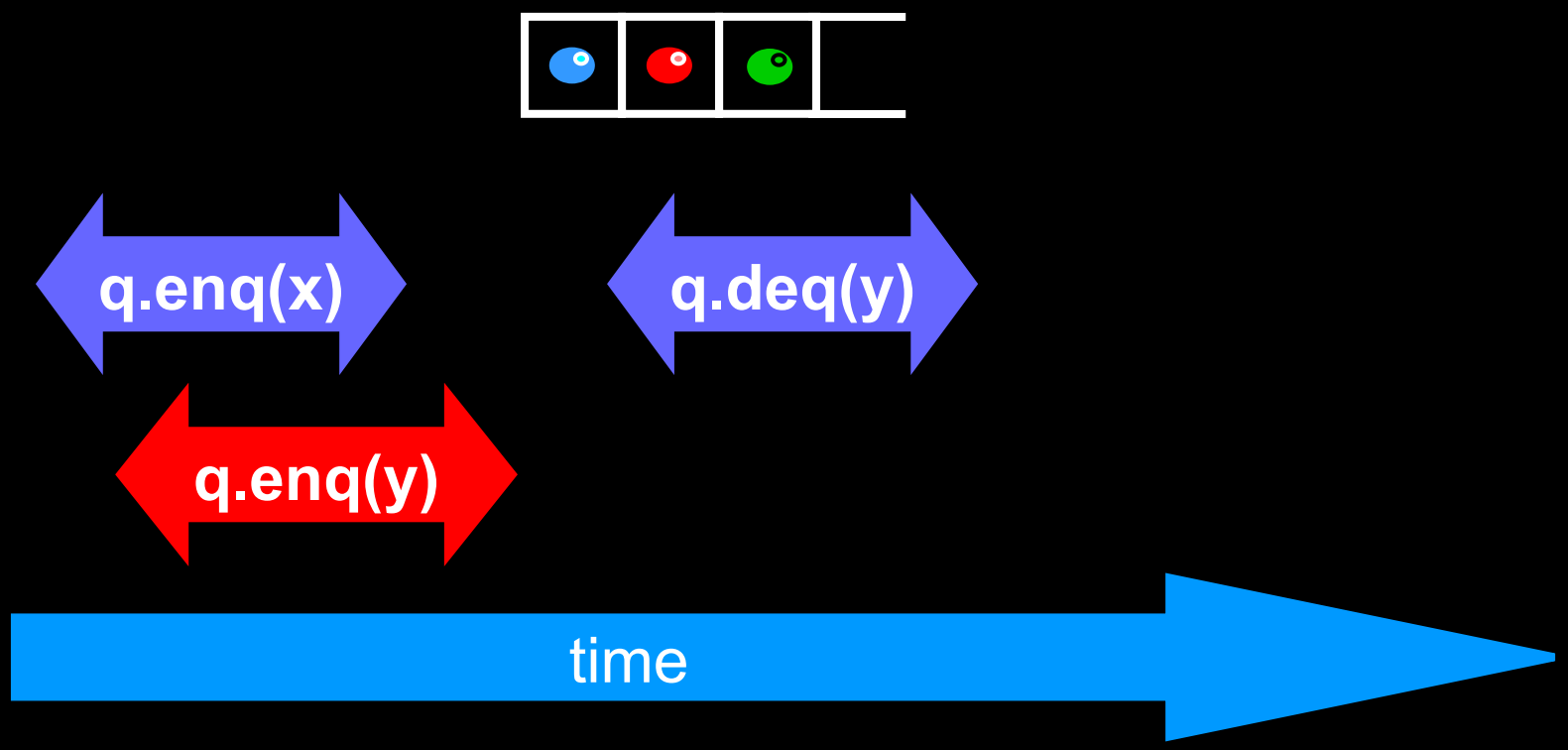
Example



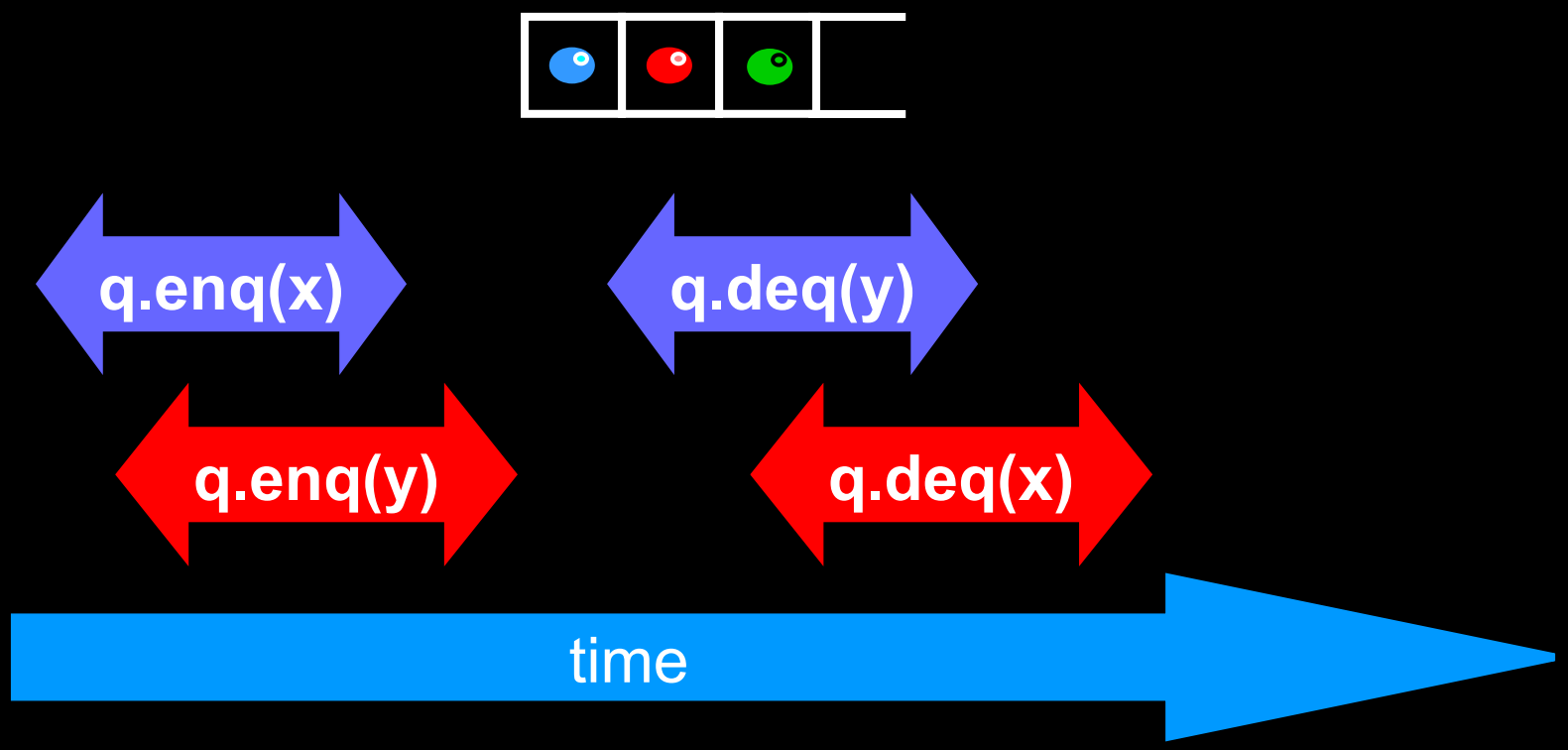


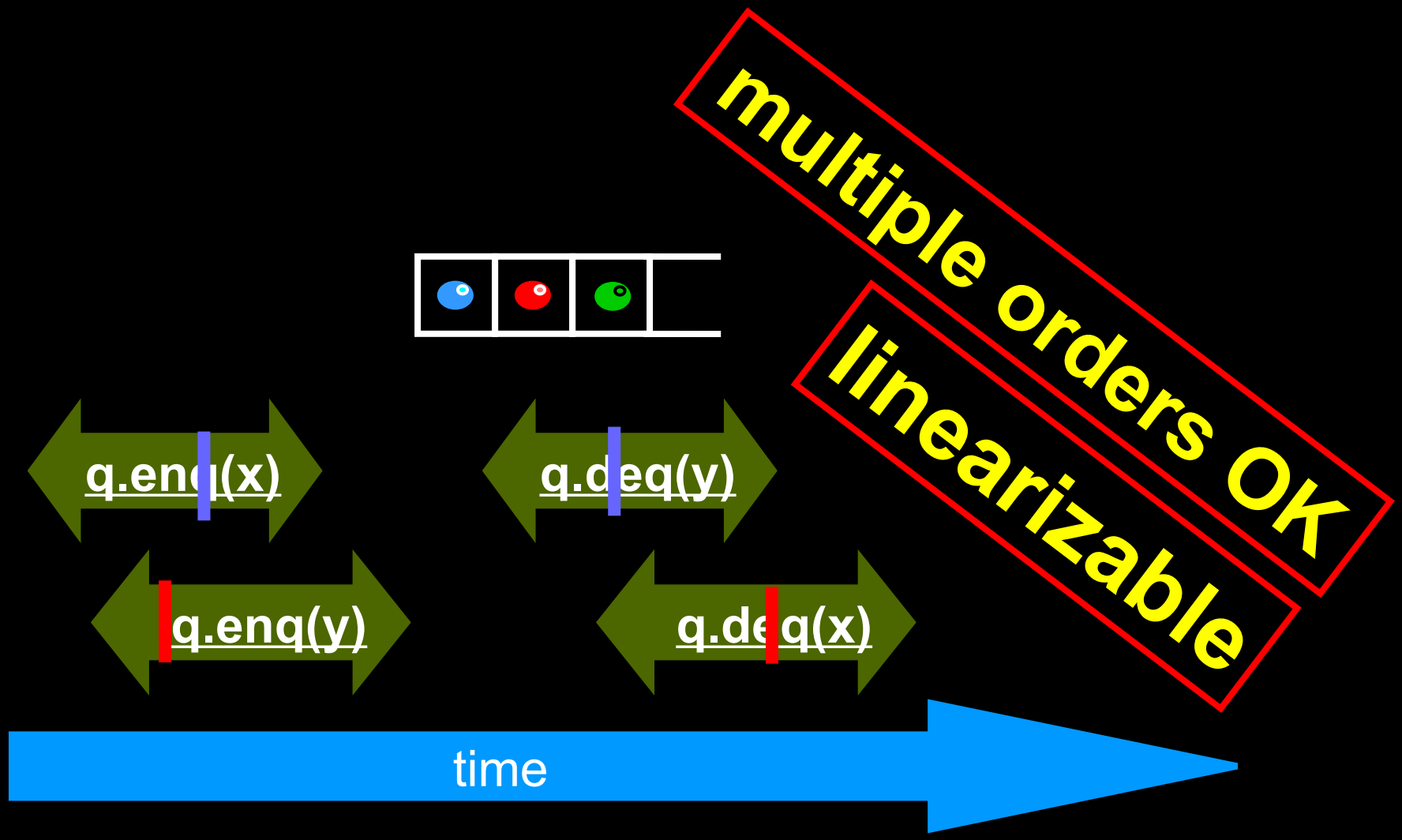


Example

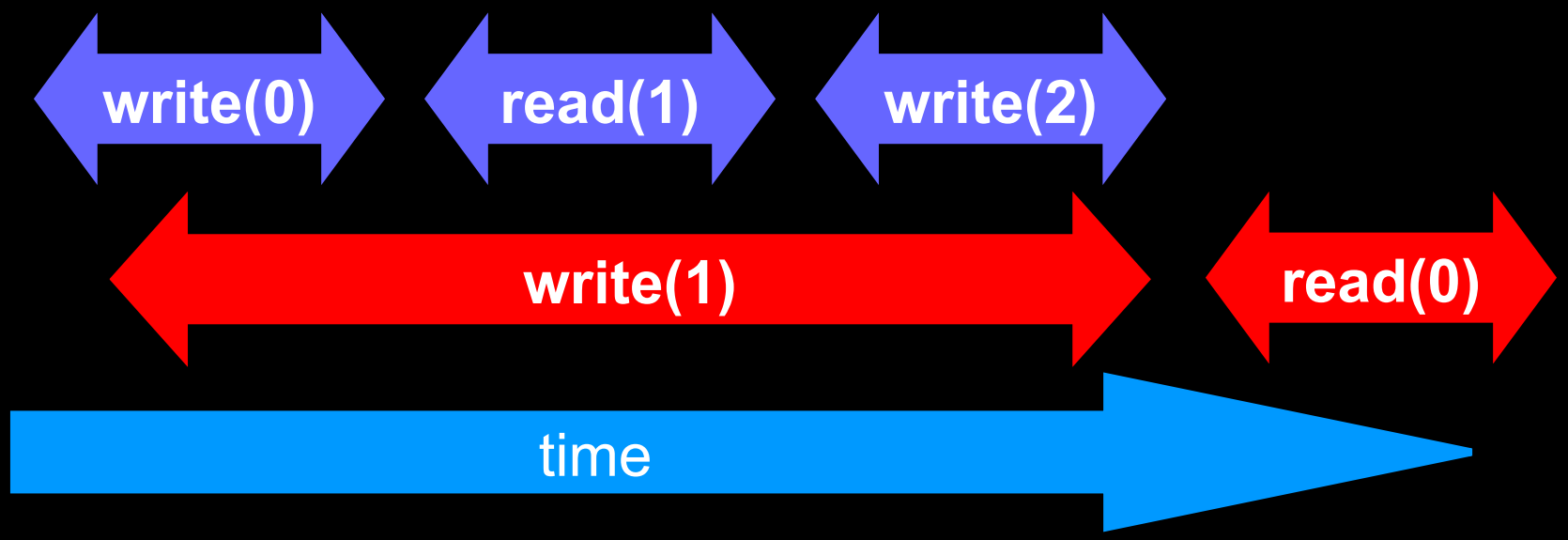


Example

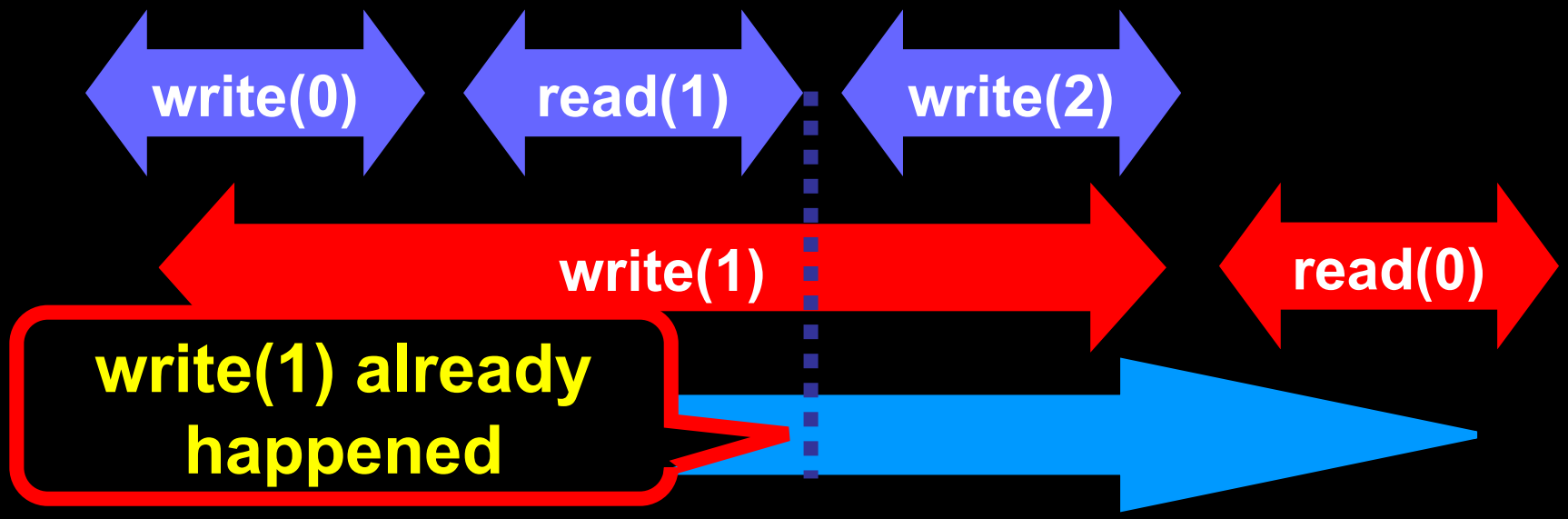



multiple orders OK
linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time

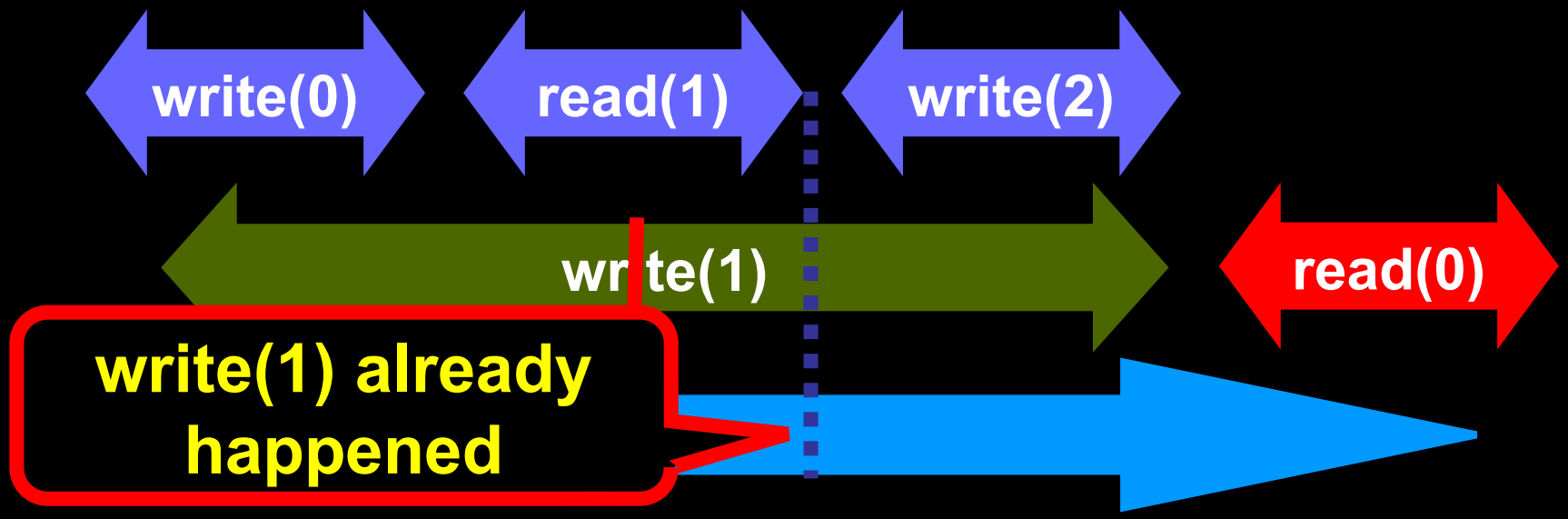
Read/Write Register Example



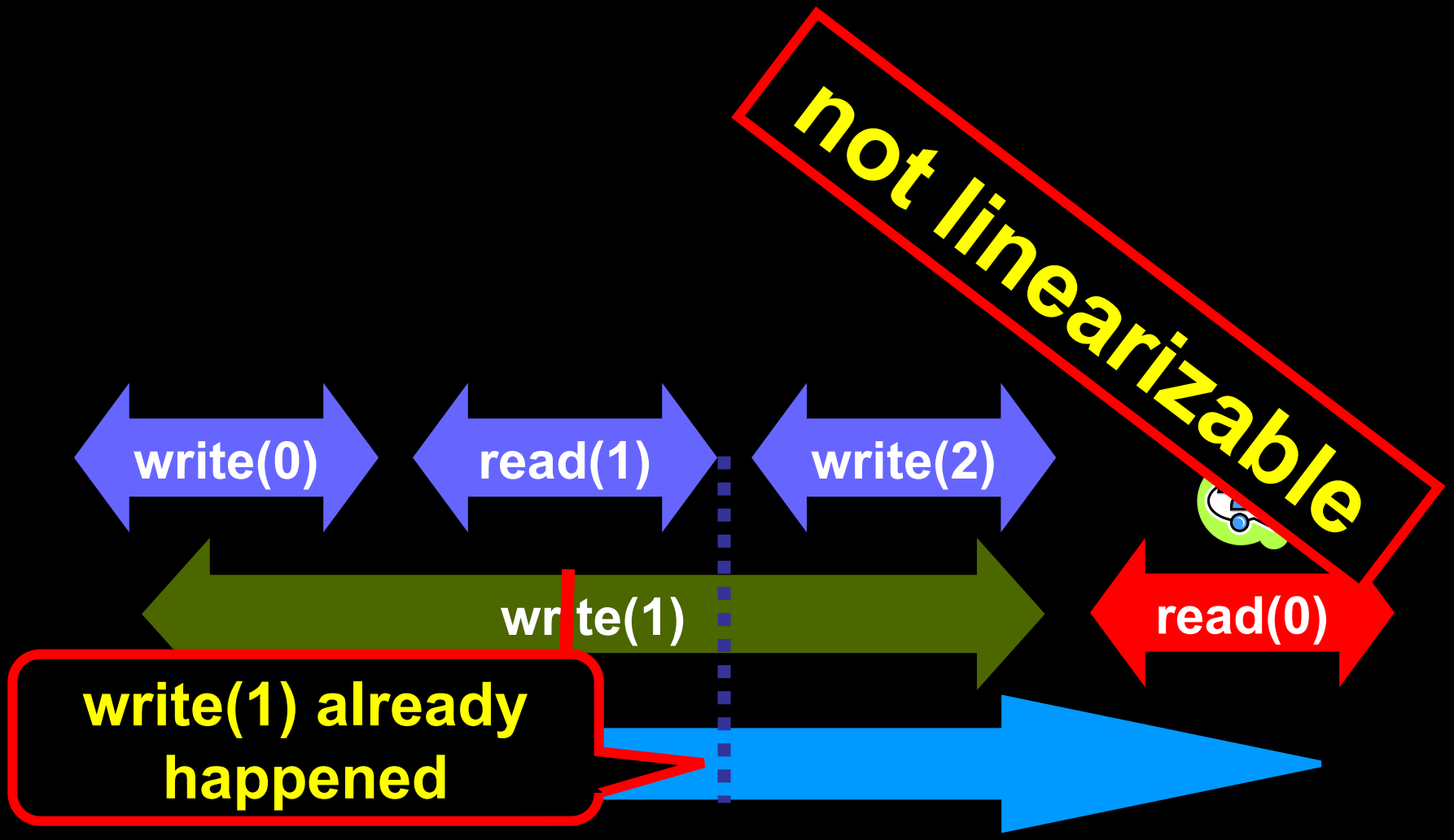
Read/Write Register Example



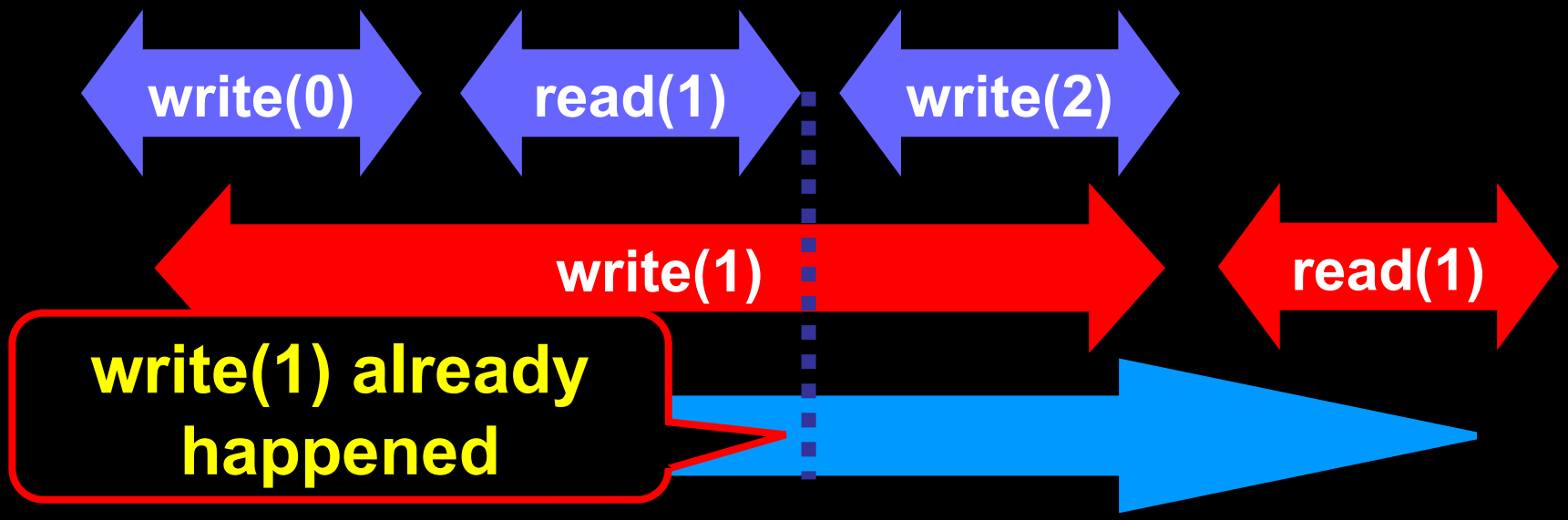
Read/Write Register Example



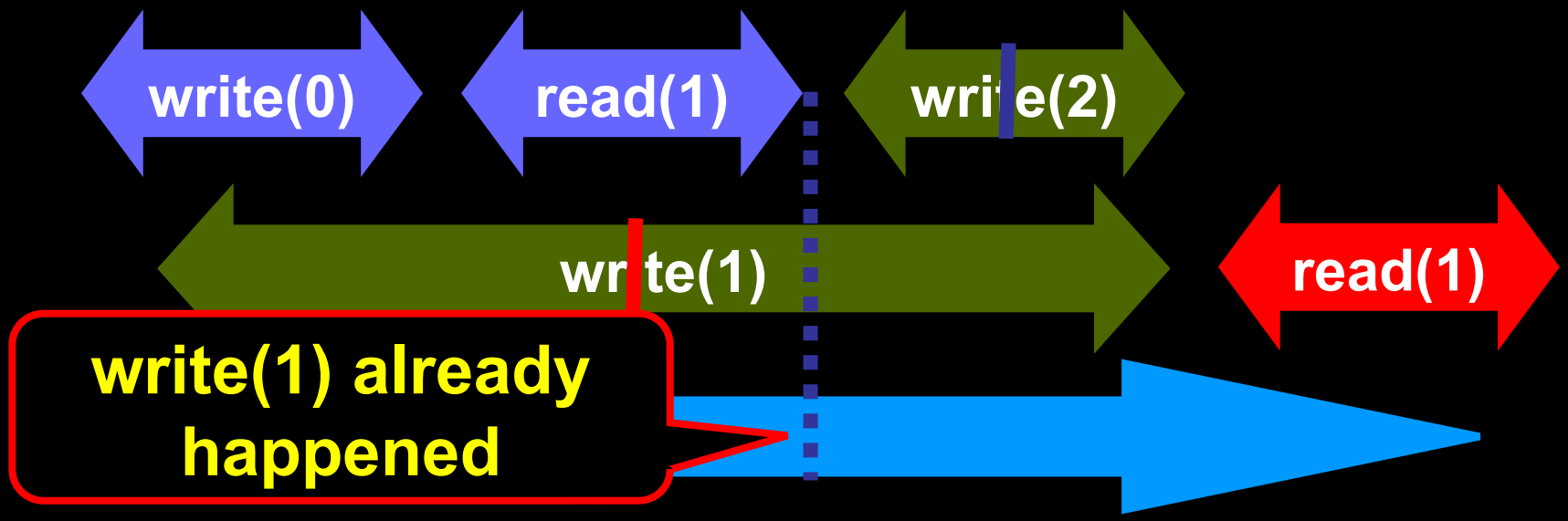
Read/Write Register Example

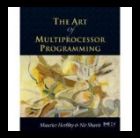


Read/Write Register Example

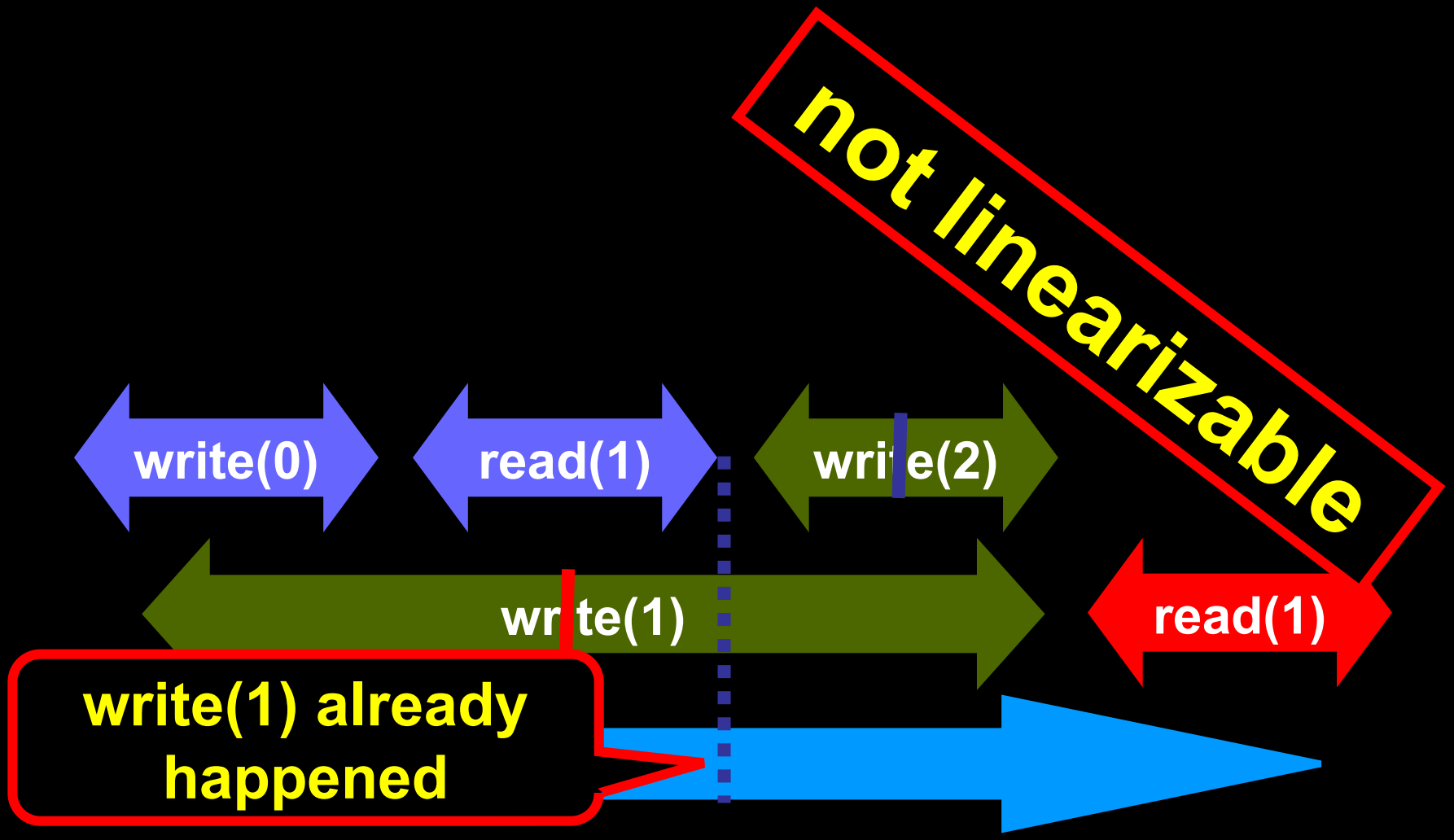


Read/Write Register Example

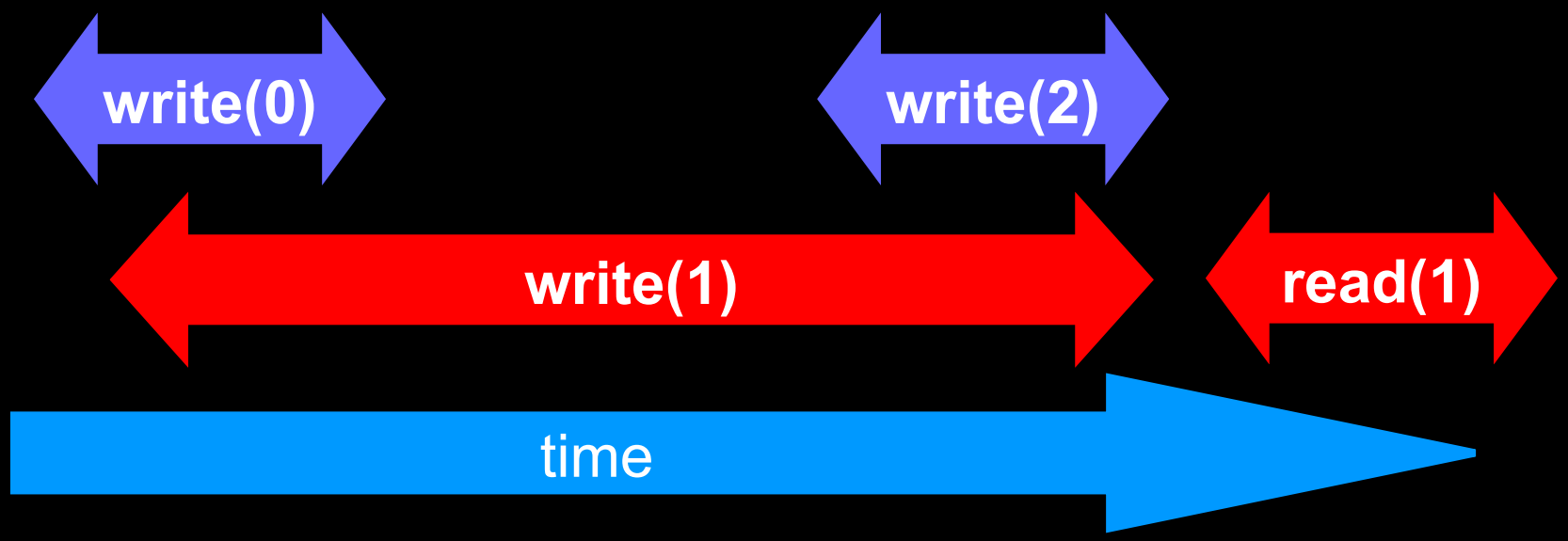




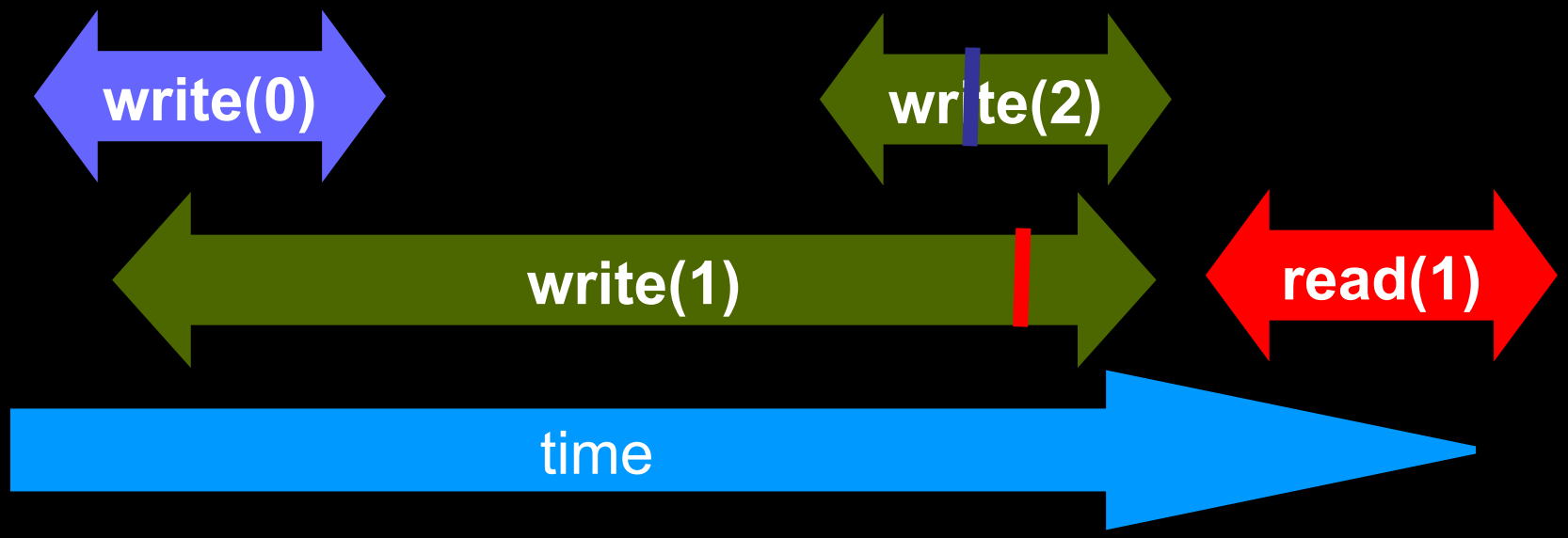
Read/Write Register Example

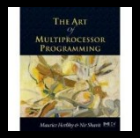


Read/Write Register Example

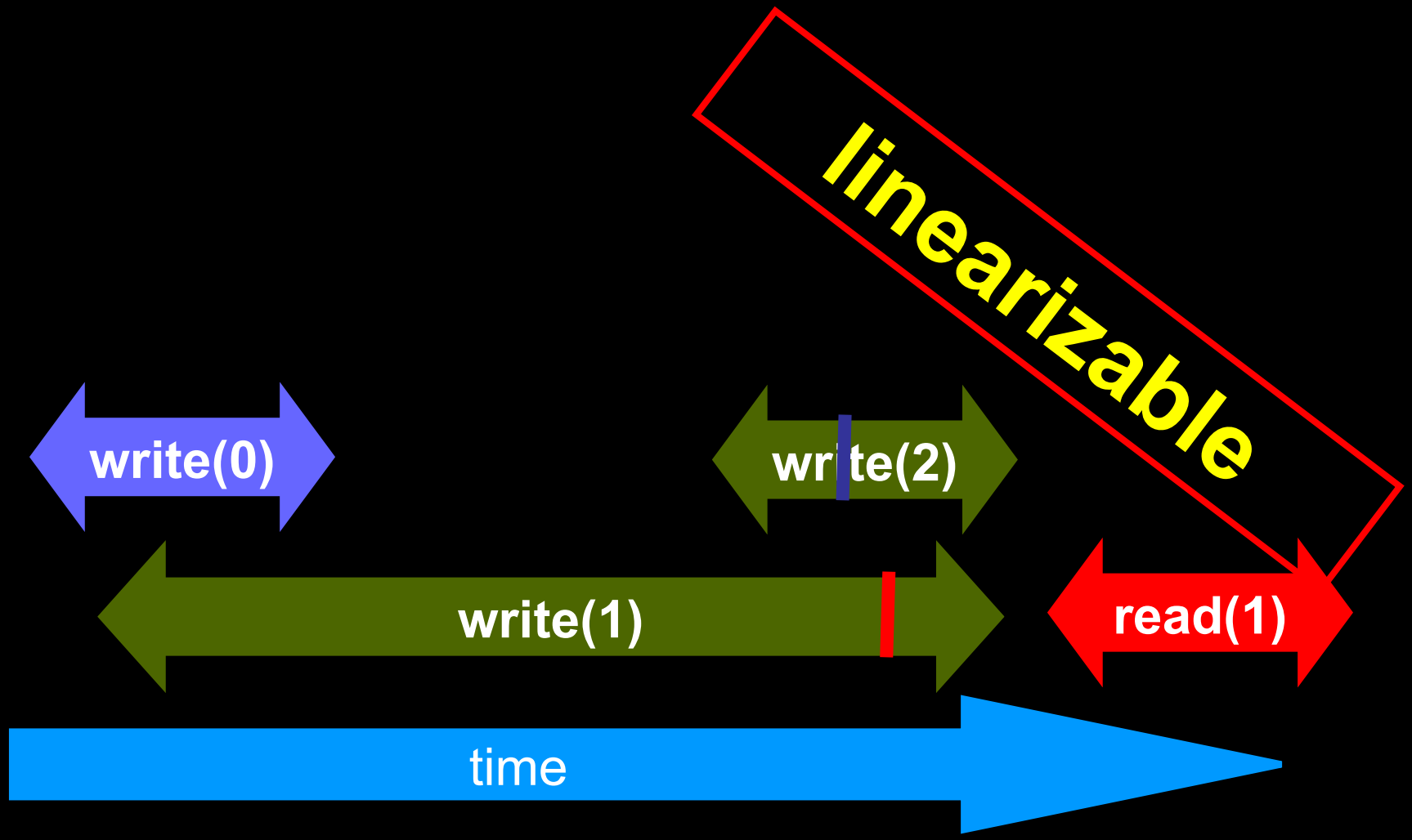


Read/Write Register Example





Read/Write Register Example

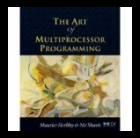


Linearization Points

Can we specify an operation's linearization point without describing an execution?

Usually, but not always

In some cases, linearization point *depends on the execution*



Formal Model of Executions

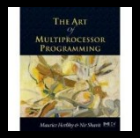
Why?

Indicate precisely what you mean to say …

We want to *reason* about objects

Sometimes formally …

Usually informally



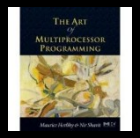
Split Method Calls into Two Events

Invocation

- **method name & args**
- `q.enq(x)`

Response

- **result or error**
- `q.enq(x)` **returns** `void`
- `q.deq()` **returns** `x`
- `q.deq()` **returns an error**



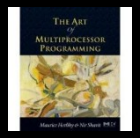
Invocation Notation

A q.enq(x)

Invocation Notation

A q.enq(x)

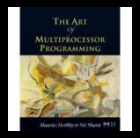
thread



Invocation Notation

A q.enq(x)

thread **method**



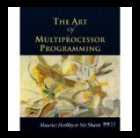
Invocation Notation

A q.enq(x)

thread

method

object



Invocation Notation

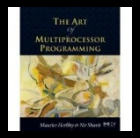
A q.enq(x)

thread

object

method

arguments



Response Notation

A q: void

Response Notation

A q: void

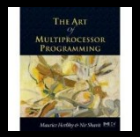
thread

Response Notation

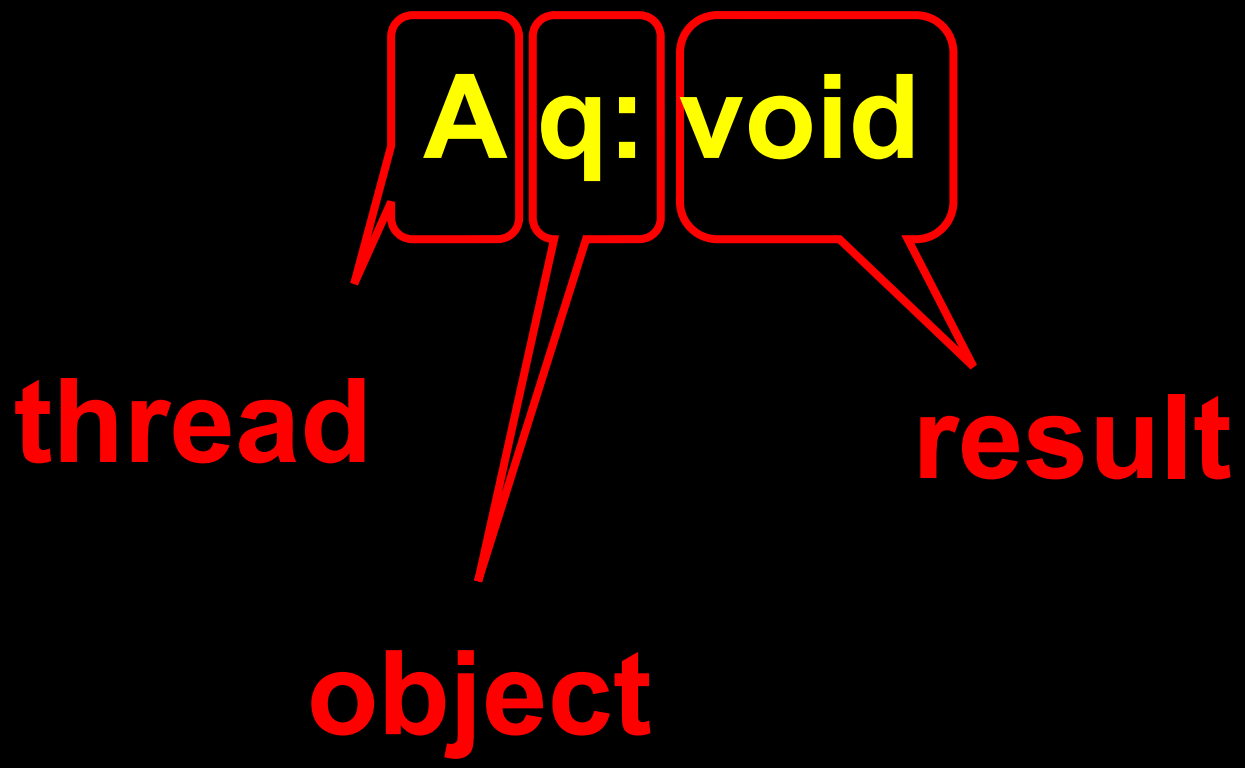
A q: void

thread

result



Response Notation



Response Notation

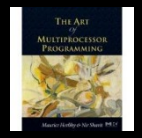
Method is implicit

A q: void

thread

object

result



Response Notation

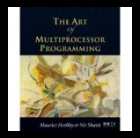
Method is implicit

A q: empty()

thread

object

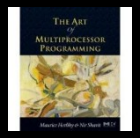
error



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

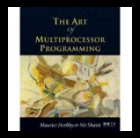
Invocation & response *match* if

Thread names agree

Object names agree

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

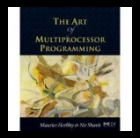
Method call



Object Projections

H =

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

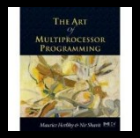


Object Projections

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

H|q =

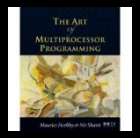
```
B q.deq()
B q:3
```



Thread Projections

H =

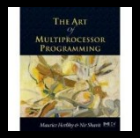
```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```



Thread Projections

H|B =

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

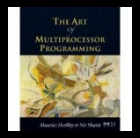


Complete Subhistory

H =

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

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

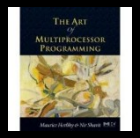


Complete Subhistory

H =

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

May or may not have taken effect

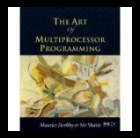


Complete Subhistory

H =

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

discard pending invocations

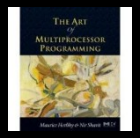


Complete Subhistory

Complete(H) =

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

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)
```

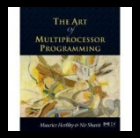
match

match

match

Final pending invocation OK

Method calls of different threads do not interleave



Well-Formed Histories

Per-thread projections sequential

H=

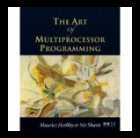
```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

H|B=

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

H|A=

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



Equivalent Histories

Threads see the same thing in both

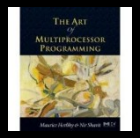
$$\begin{cases} H|A = G|A \\ H|B = G|B \end{cases}$$

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

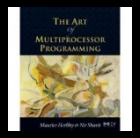
A single-thread, single-object history

Is legal

For example:

Pre and post-conditions

But any style will do …

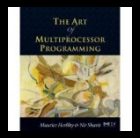


Legal Histories

A sequential (multi-object) history H is *legal* if

For every object *x*

H|x is in the sequential spec for *x*

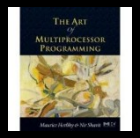


Precedence

```
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**



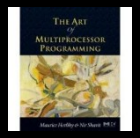
Concurrency

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

Some method calls overlap one another

Method call

Method call



Linearizability

History H is *linearizable* if it can be extended to G by

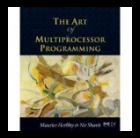
Appending zero or more responses to pending invocations

Discarding other pending invocations

So that G is equivalent to

Legal sequential history S

where S respects “real-time order” of G

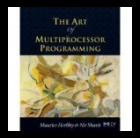


Remarks

Some pending invocations

Took effect, so keep them

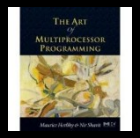
Discard the rest



Linearizability: Summary

Standard notion of correctness

Captures the notion of objects being “atomic”



Alternative: Sequential Consistency

History H is *Sequentially Consistent* if it can be extended to G by

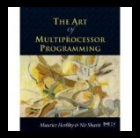
Appending zero or more responses to pending invocations

Discarding other pending invocations

So that G is equivalent to a

Legal sequential history S

Does *not* require that S respects “real-time order” of G



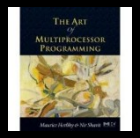
Sequential Consistency

Does *not* preserve real-time order

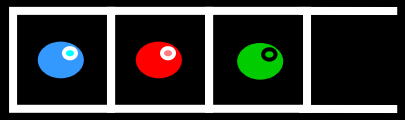
Not allowed to re-order same-thread operations

OK to re-order operations by different threads

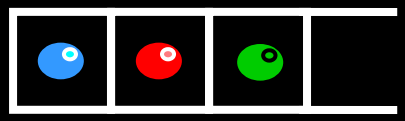
Often used for multiprocessor memories

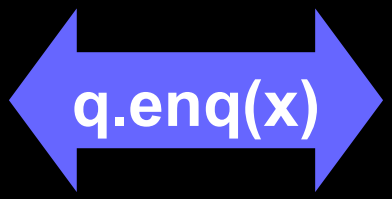


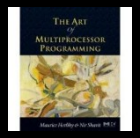
Example



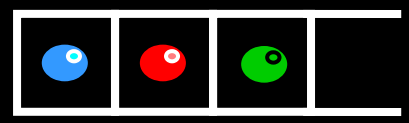
Example

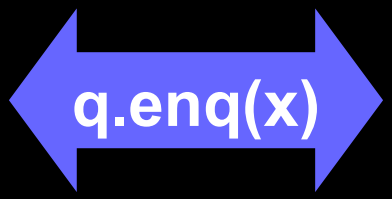




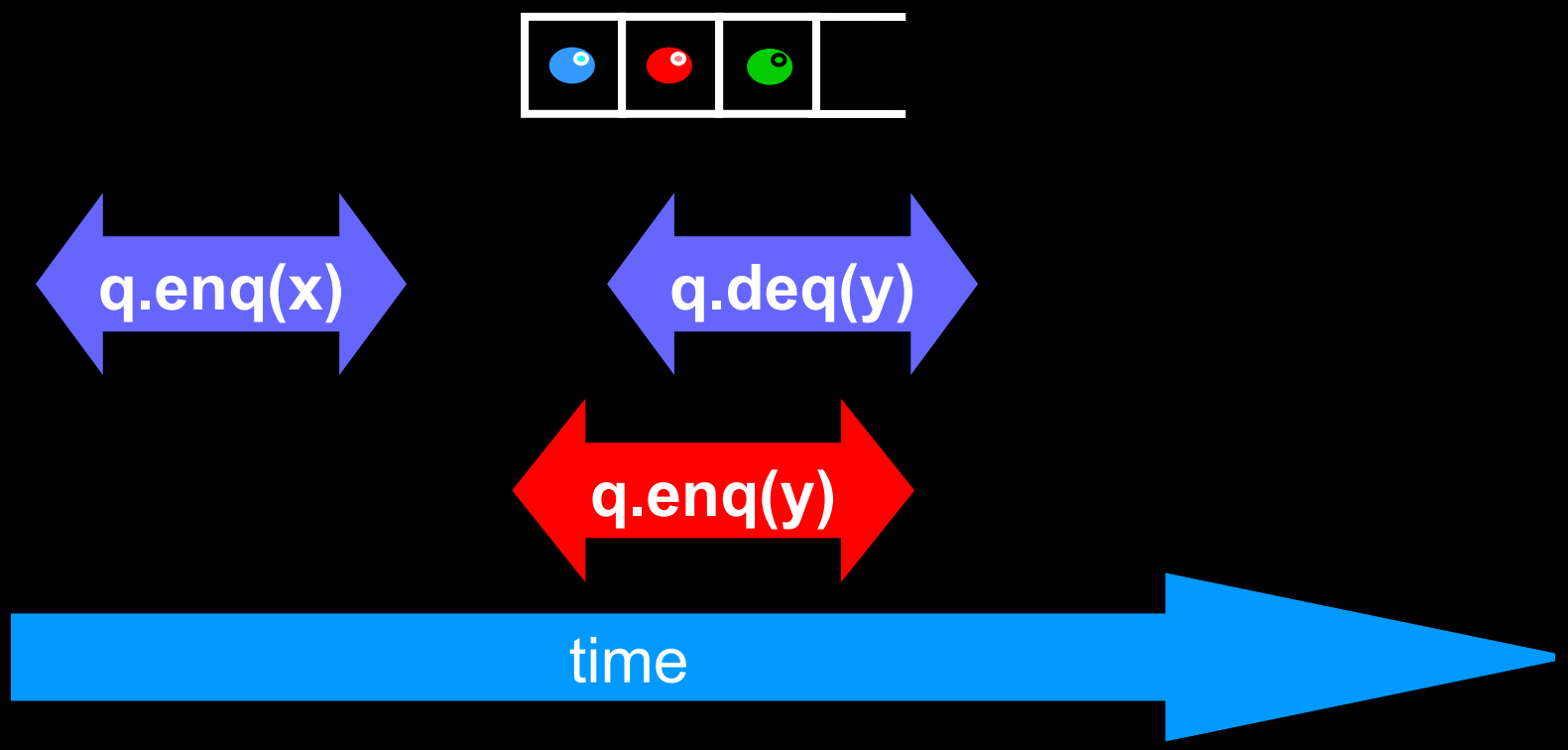


Example

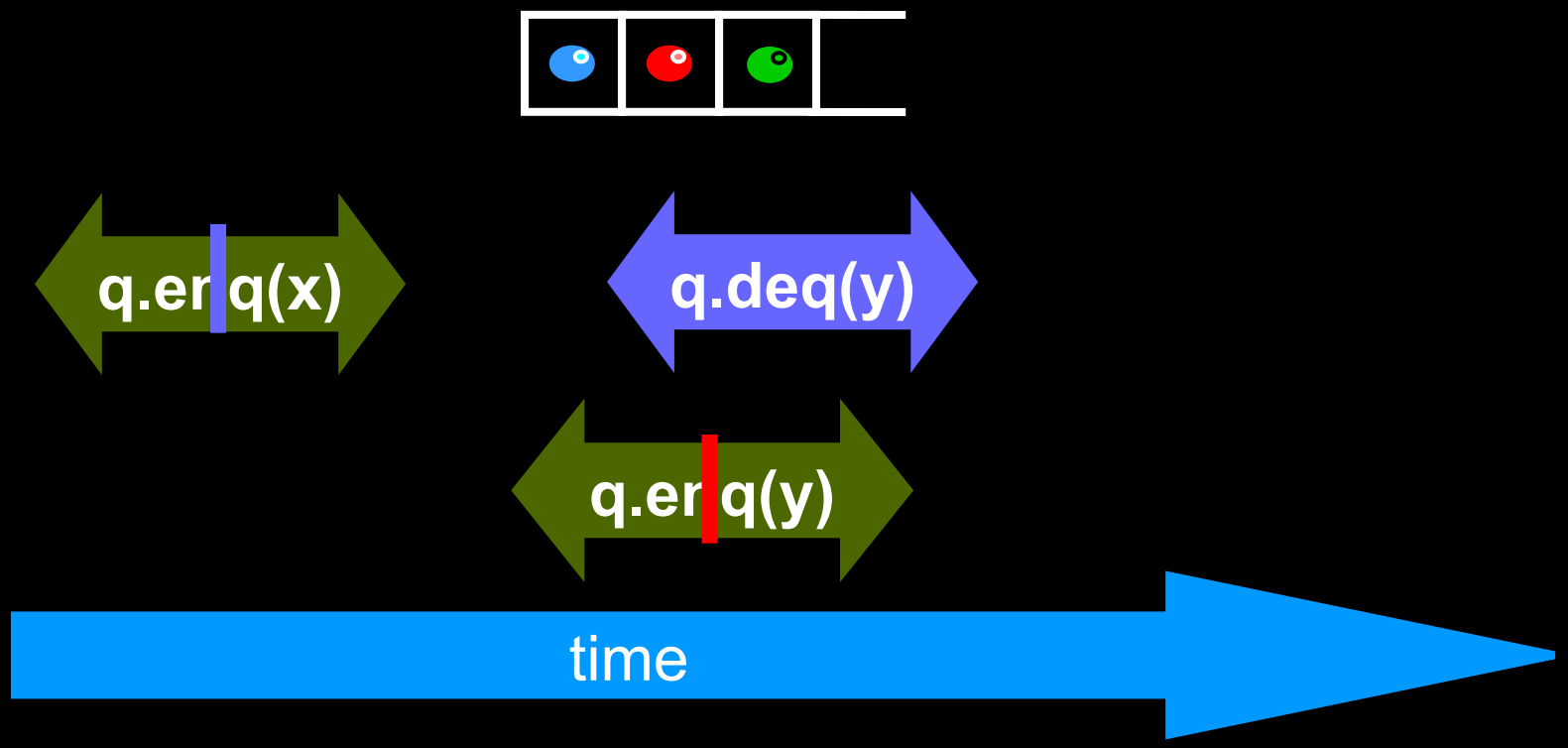




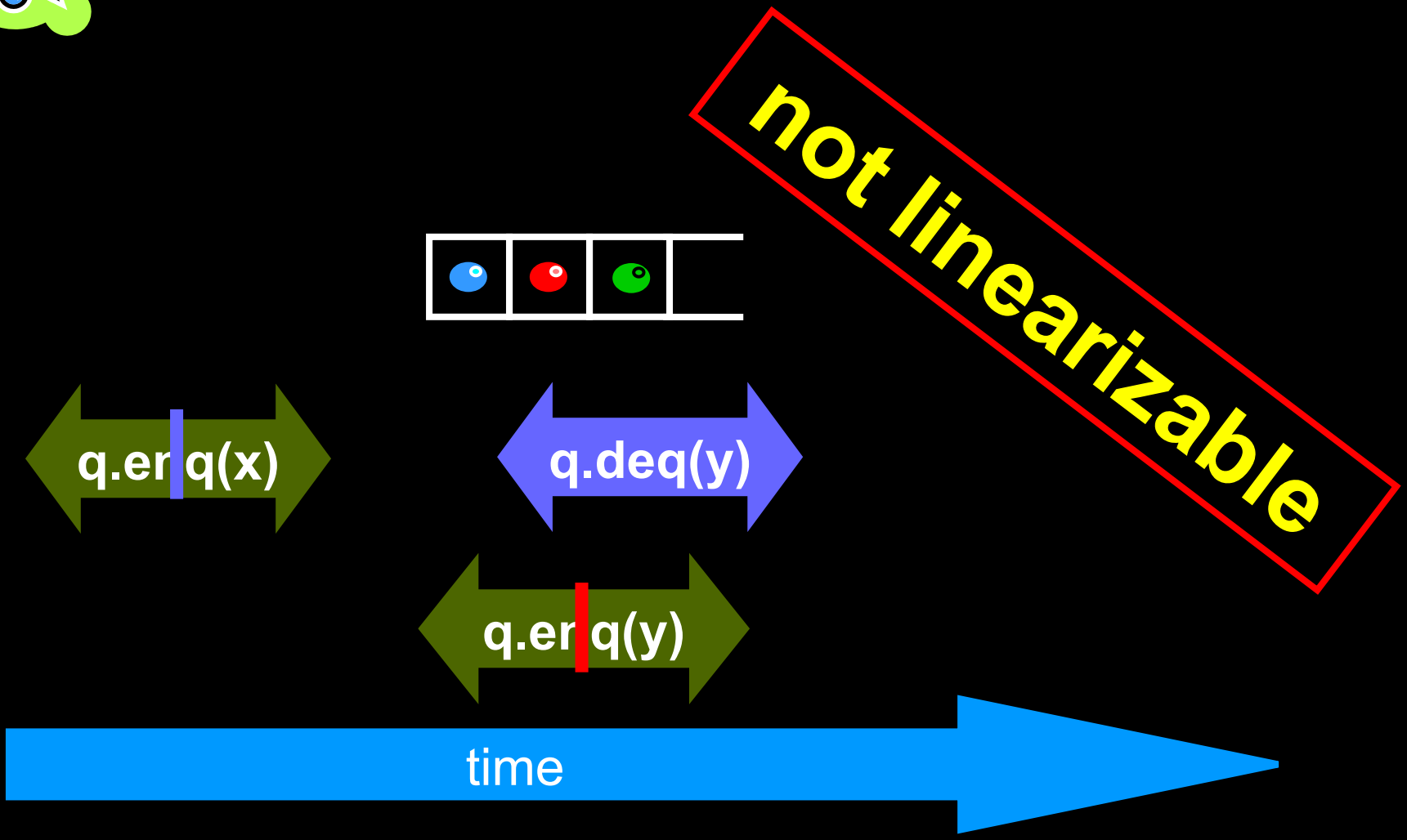
Example



Example

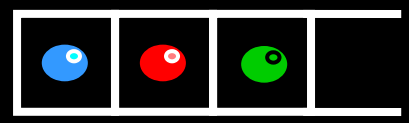


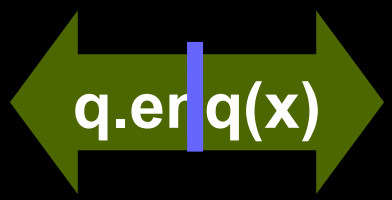
Example

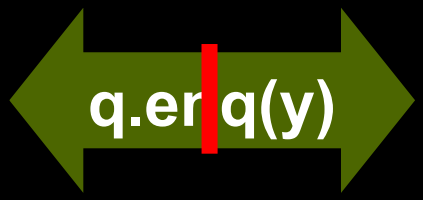


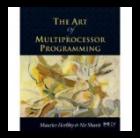
Exa

Yet Sequentially Consistent









Sequential consistency

Most hardware does not even support that, architects think that sequential consistency is *too strong*

Too expensive for modern hardware

OK if violated by *default*

Honored by *explicit request*

Hardware Consistancy

Initially, a = b = 0.

Core 0

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

Core 1

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

What are the possible values of %eax and %ebx registers 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 too

Because performance is dominated by single-thread unsynchronized execution

Instruction Reordering

```
mov 1, a      ;write
mov b, %ebx   ;read
```

```
mov b, %ebx   ;read
mov 1, a      ;write
```

Program Order

Execution Order

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

A. Higher performance by covering load latency — *instruction-level parallelism*.

Slide used with permission of
Charles E. Leiserson

Instruction Reordering

```
mov 1, a      ;write
mov b, %ebx   ;read
```

```
mov b, %ebx   ;read
mov 1, a      ;write
```

Program Order

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

X86: Memory Consistency – TSO (Total Store Ordering)

Thread Code

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

Loads are *not* reordered with loads.

Stores are *not* reordered with stores.

Stores are *not* reordered with prior loads.

A load *may* be reordered with a prior store to a different location but *not* with a prior store to the same location.

Stores to the same location respect a global total order.

Art of Multiprocessor Programming

Memory Barriers (Fences)

A *memory barrier* (or *fence*) is a hardware operation that enforces ordering between the instructions before and after the fence

A memory barrier can be an explicit 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
```

Loads are *not* reordered with loads.

Sto

Sto
load

A lo
store to
with a prior store to the same location.

Stores to the same location respect a global total order.

Total Store Ordering + properly placed memory barriers = sequential consistency

Memory Barriers

Memory barrier instruction will

Flush write buffer

Bring caches up to date

Compilers often do this for you

Entering and leaving critical sections