YSC4231: Parallel, Concurrent and Distributed Programming

Linearizability (c'd) and Wait-Free Implementations

Last Week: Linearizability

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 $\rightarrow_G \subset \rightarrow_S$

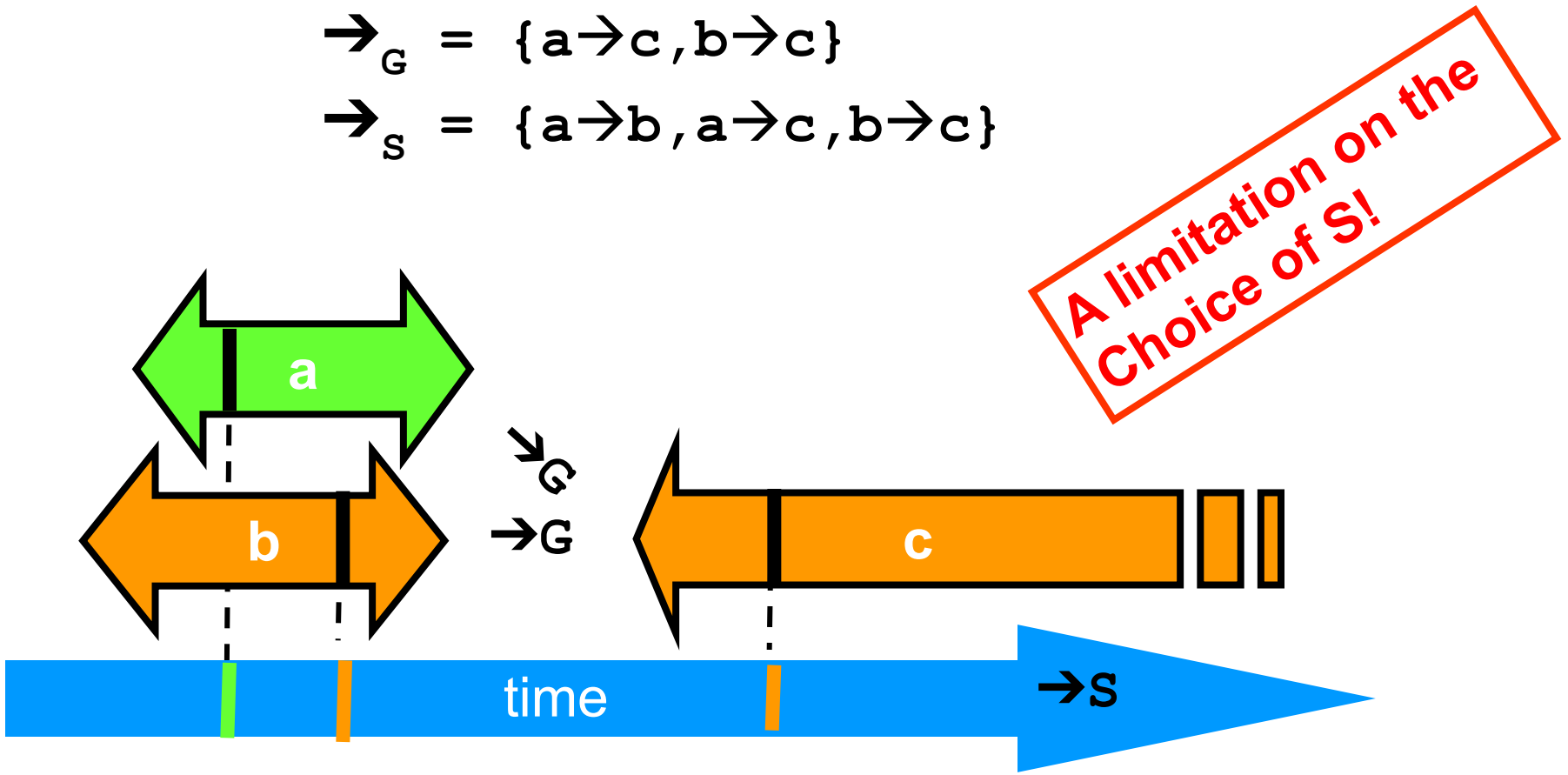
Remarks

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

Ensuring $\rightarrow_G \subset \rightarrow_S$

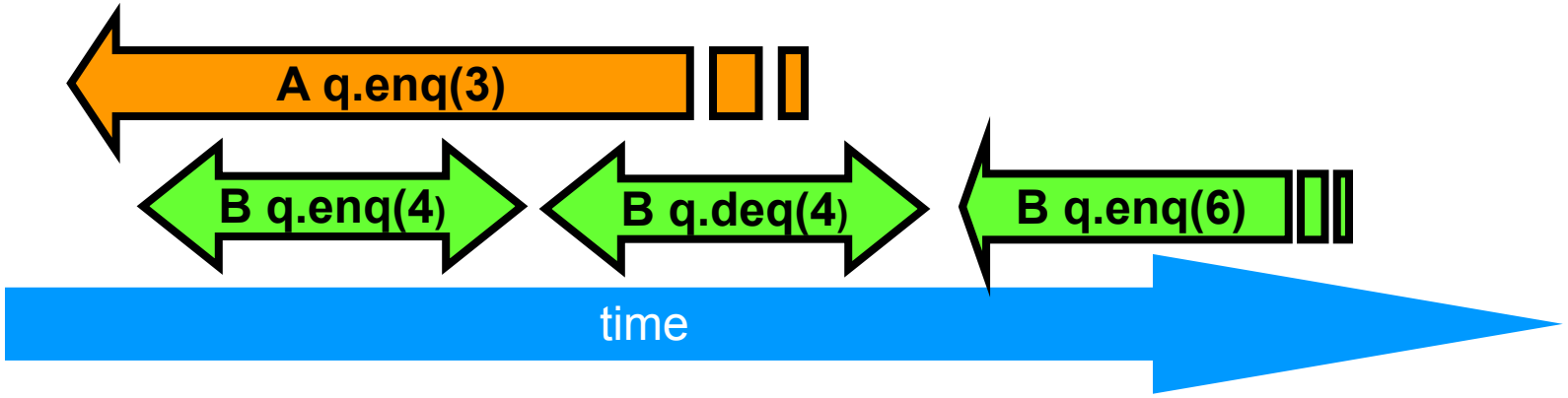
$\rightarrow_G$ = {a→c,b→c}

$\rightarrow_S$ = {a→b,a→c,b→c}



Example

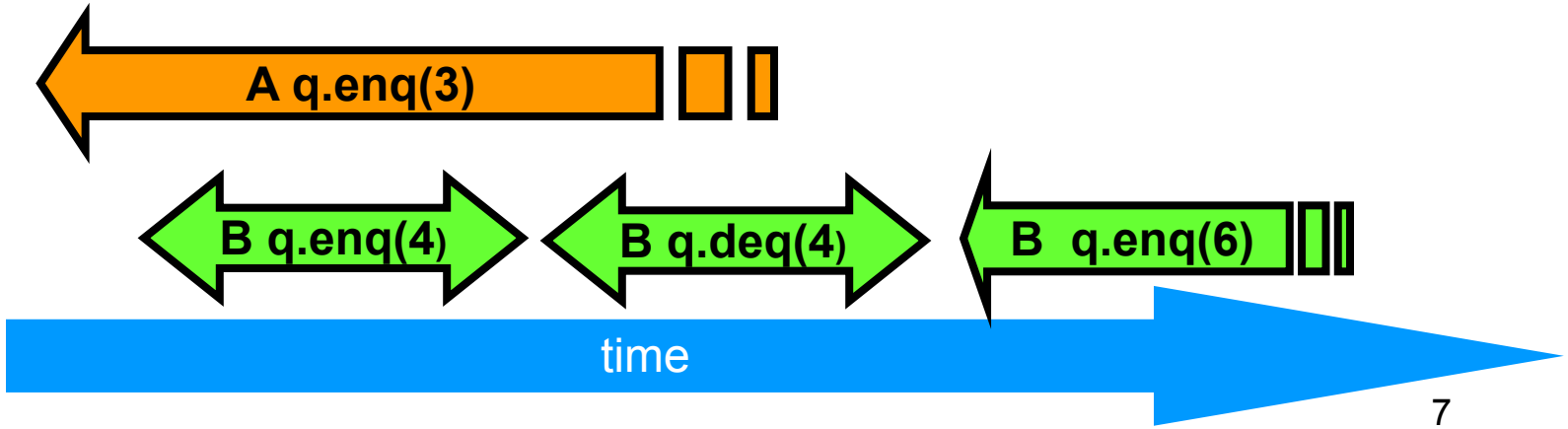
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
```



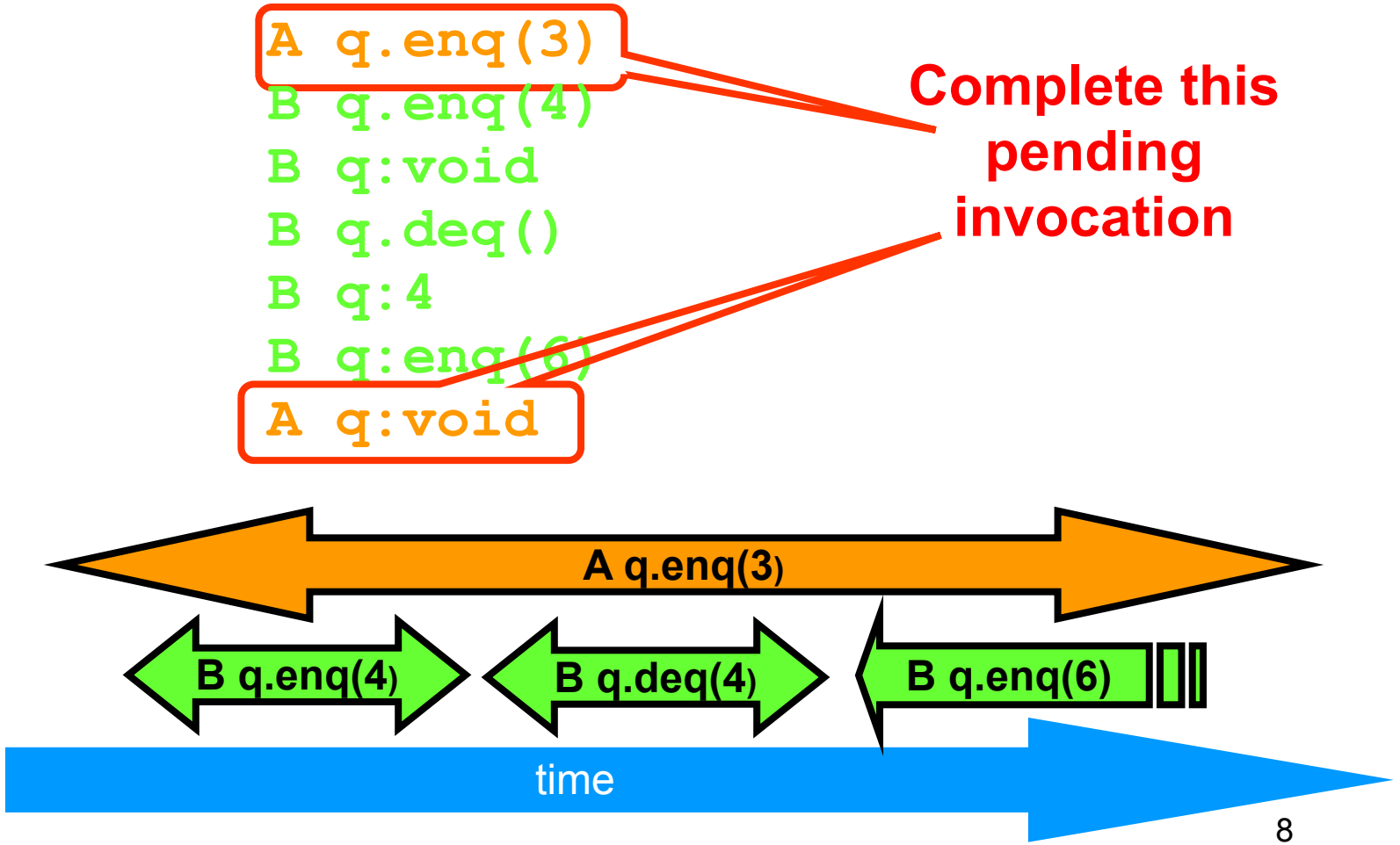
Example

```
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



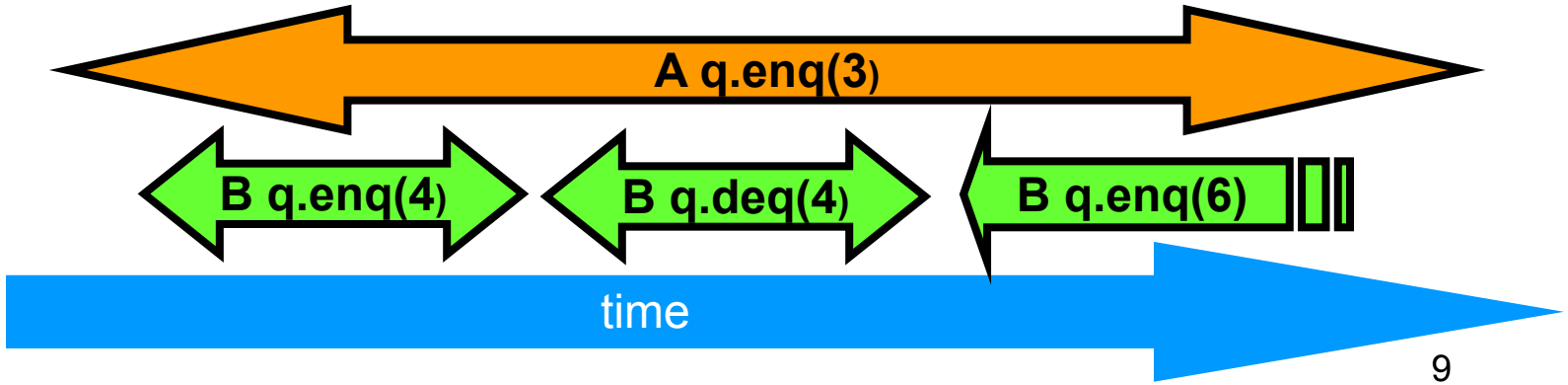
Example


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
A q.enq(3)
B q.enq(4)
B q.deq(4)
B q.enq(6)
time

Example

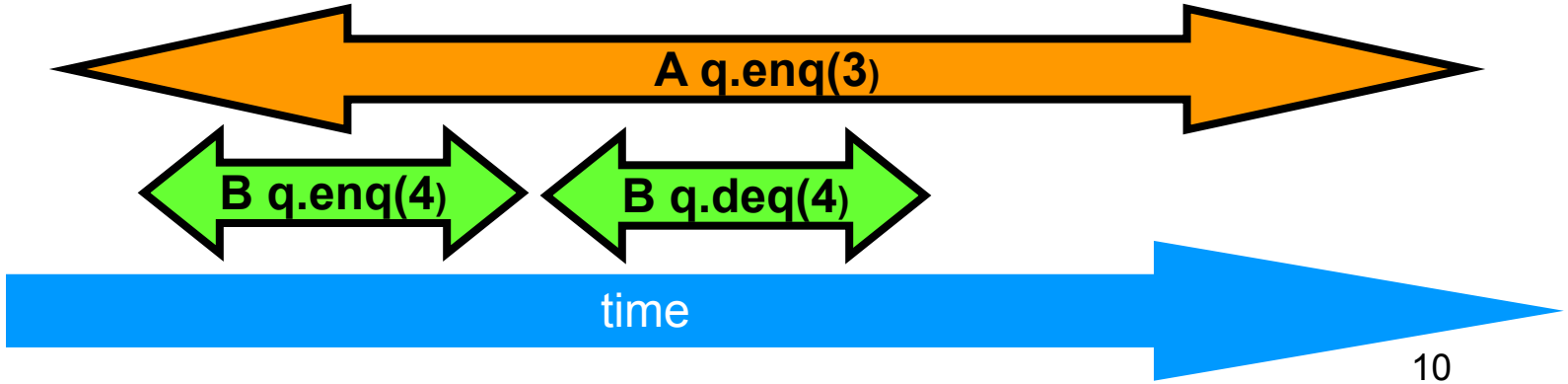
discard this one

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



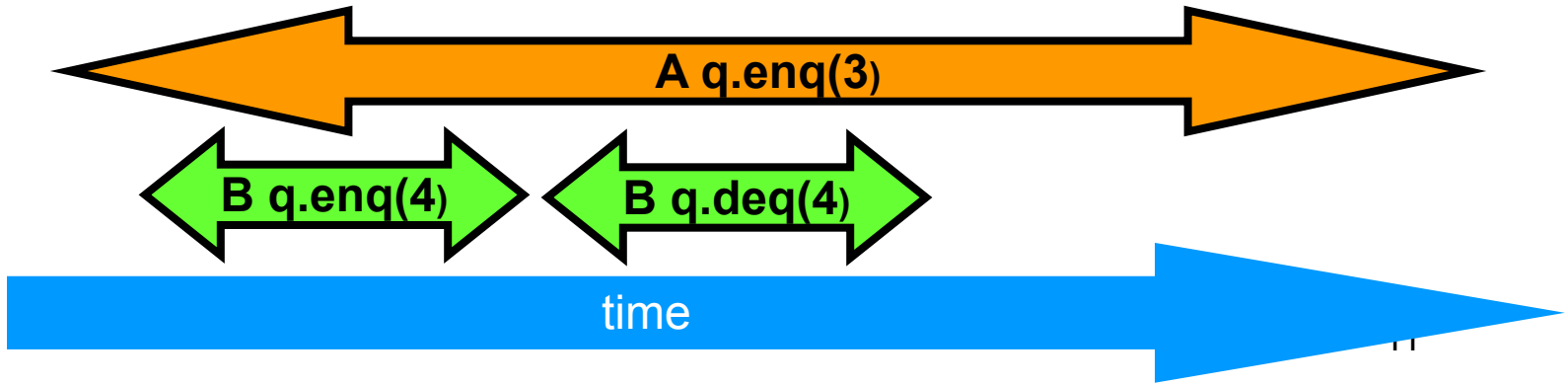
Example

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

A q:void
```



Example

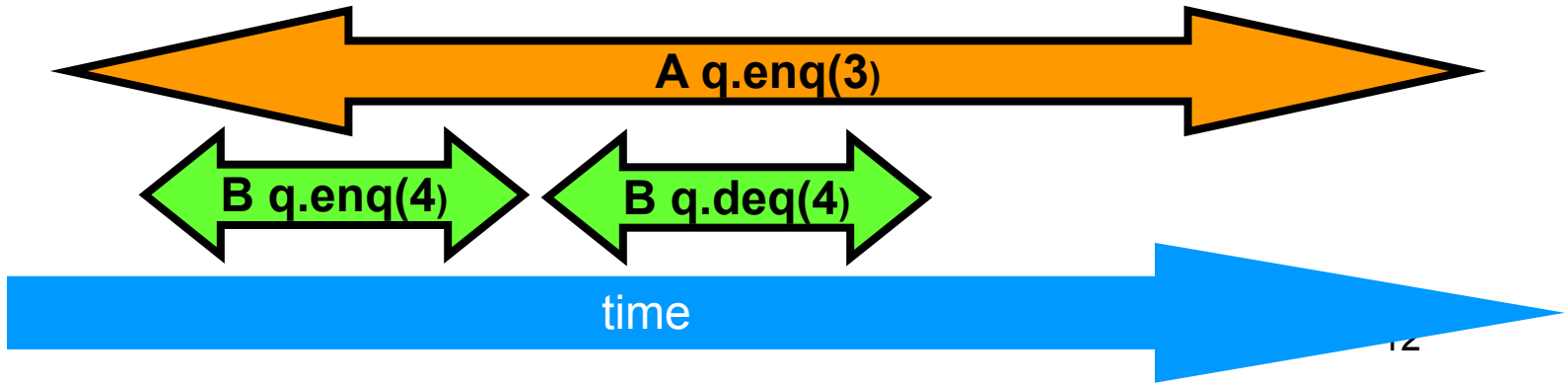
```
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void
```



Example

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

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

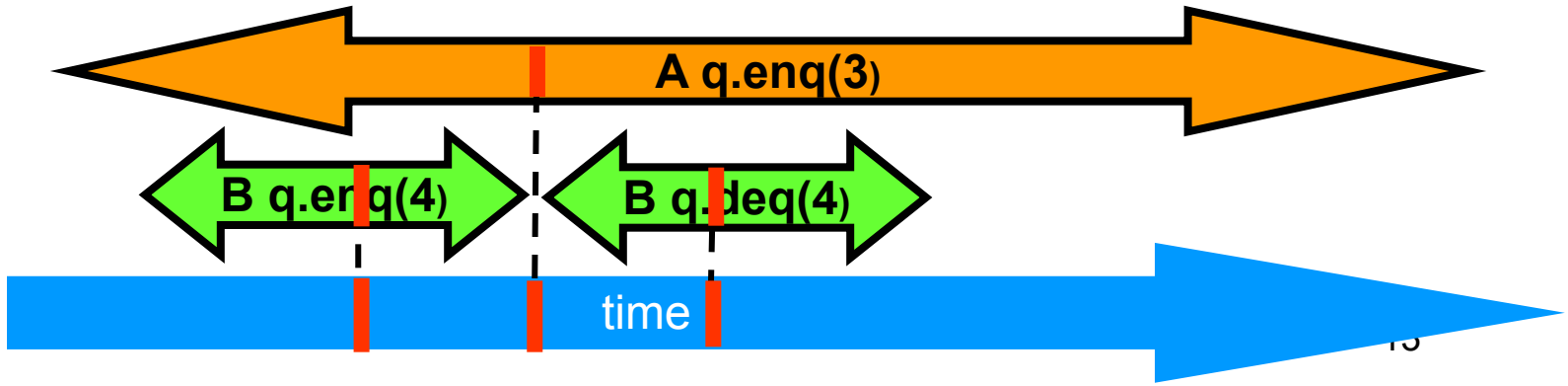


Example

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

Equivalent sequential history

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

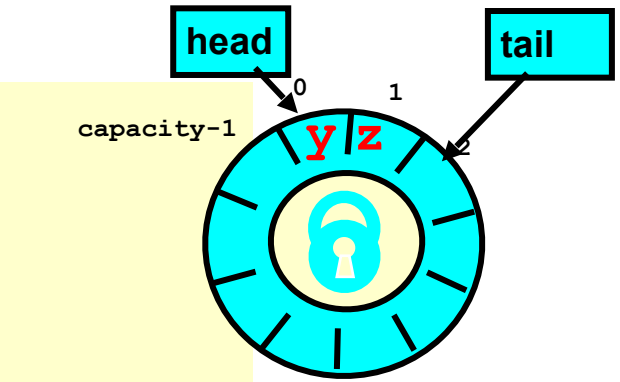


Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects
  - A history of two linearizable objects is linearizable

Reasoning About Linearizability: Locking

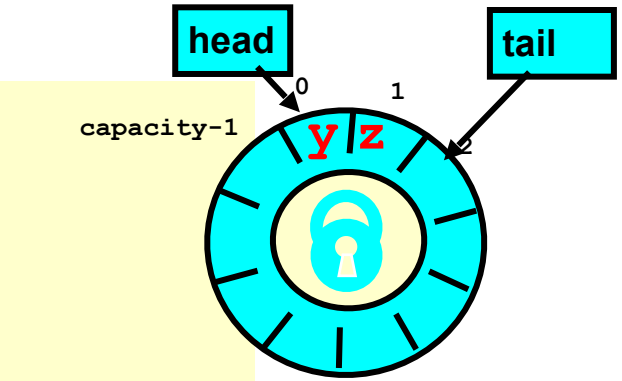
```
def deq() : T = {
  myLock.lock()
  try {
    if (tail == head) {
      throw EmptyException
    }
    val x = items(head % items.length)
    head = head + 1
    x
  } finally {
    myLock.unlock()
  }
}
```



Reasoning About Linearizability: Locking

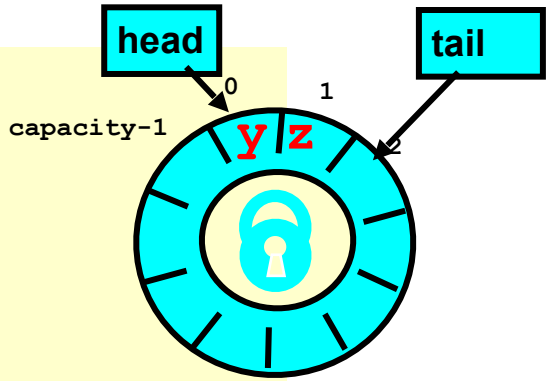
```
def deq() : T = {
  myLock.lock()
  try {
    if (tail == head) {
      throw EmptyException
    }
    val x = items(head % items.length)
    head = head + 1
    x
  } finally {
    myLock.unlock()
  }
}
```

Linearization points are when locks are released



More Reasoning: Wait-free

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {

  @volatile
  private var head, tail: Int = 0
  private val items = new Array[T](capacity)

  def enq(x: T): Unit = {
    if (tail - head == items.length) throw FullException
    items(tail % items.length) = x
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```



More Reasoning: Wait-free

```
class LockFr ueue[T: ClassTag](val capacity: Int) {


  d, tail: Int = 0
  s = new Array[T](capacity)

  Enq(x: T): Uni
    if (tail - head
    items(tail % ite
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```

Remember that there is only one enqueuer and only one dequeuer

Linearization in the case when operations succeed

More Reasoning: Wait-free

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {

  @volatile
  private var head, ta
  private val items =

  def enq(x: T): Unit
    if (tail - head == items.length) throw FullException
    items(tail % items.length) = x
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```

Linearization in the case when operations fail

Strategy

- Identify one atomic step where method “happens”
  - Critical section
  - Machine instruction
- Doesn’t always work
  - Might need to define several different scenarios for a given method
  - Example: if the method’s fails, its linearization point is A, if it succeeds its LP is B

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

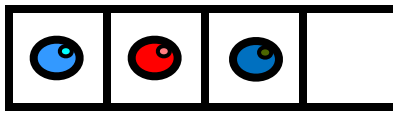
- History H is ***Sequentially Consistent*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to a
  - Legal sequential history **S**
  - Where $\rightarrow_G \subset \rightarrow_S$

Differs from linearizability

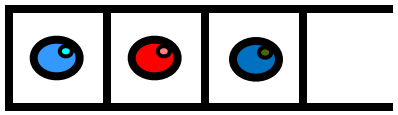
Sequential Consistency

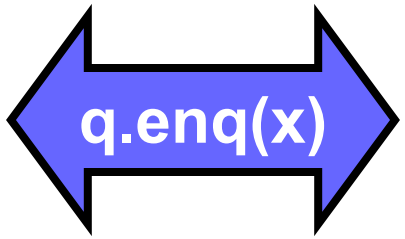
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - **Can** re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

Example

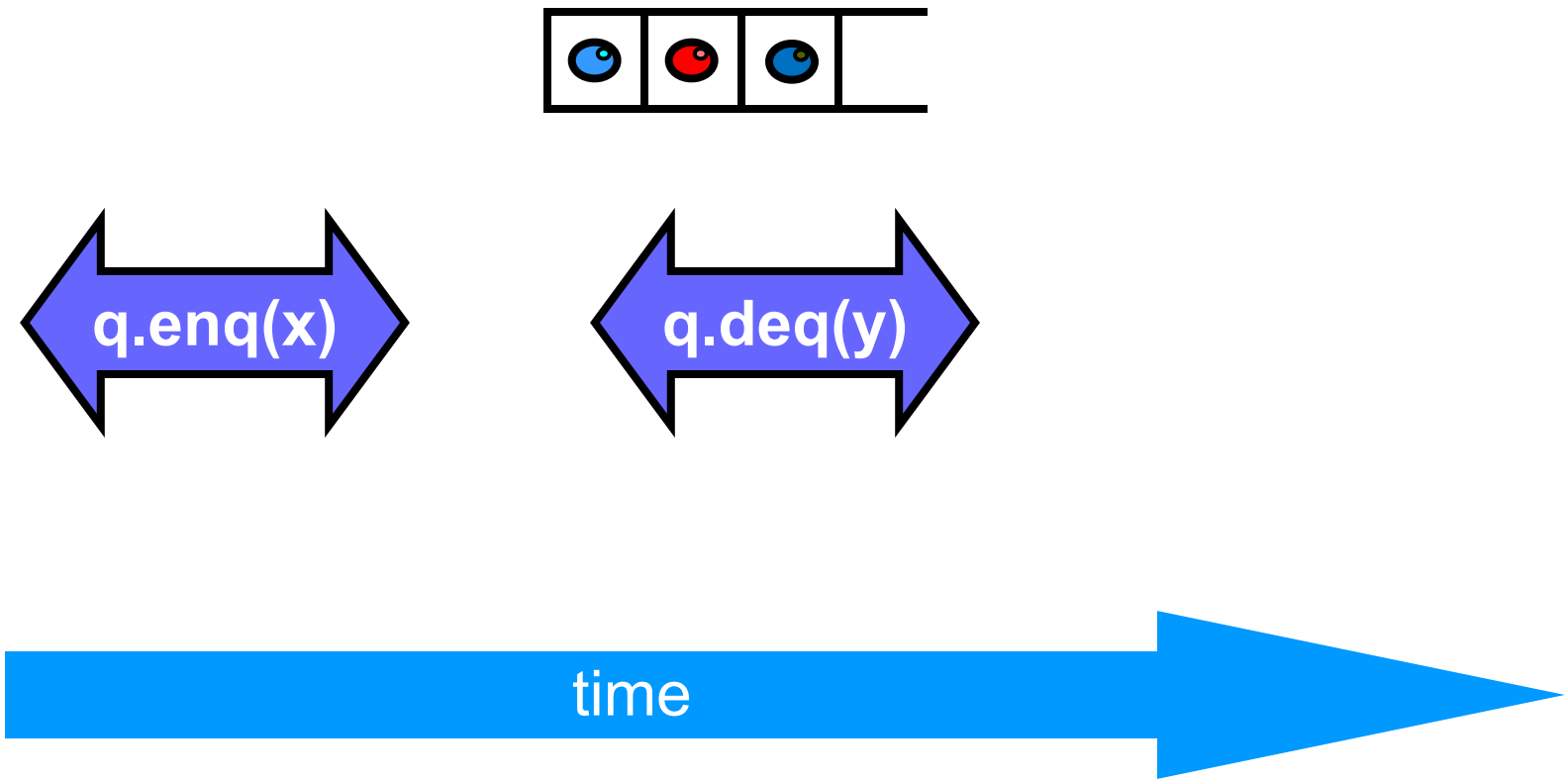


Example

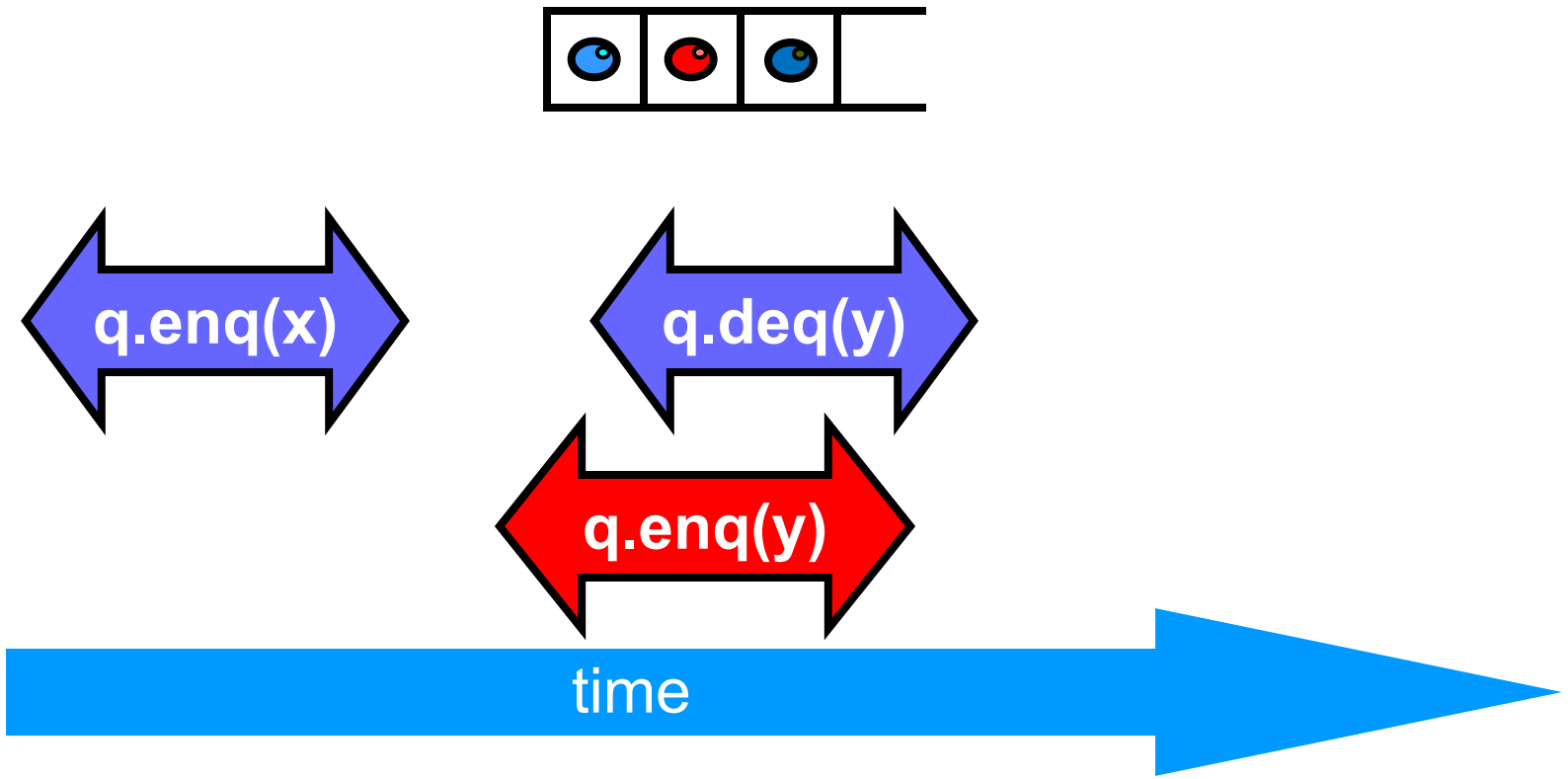




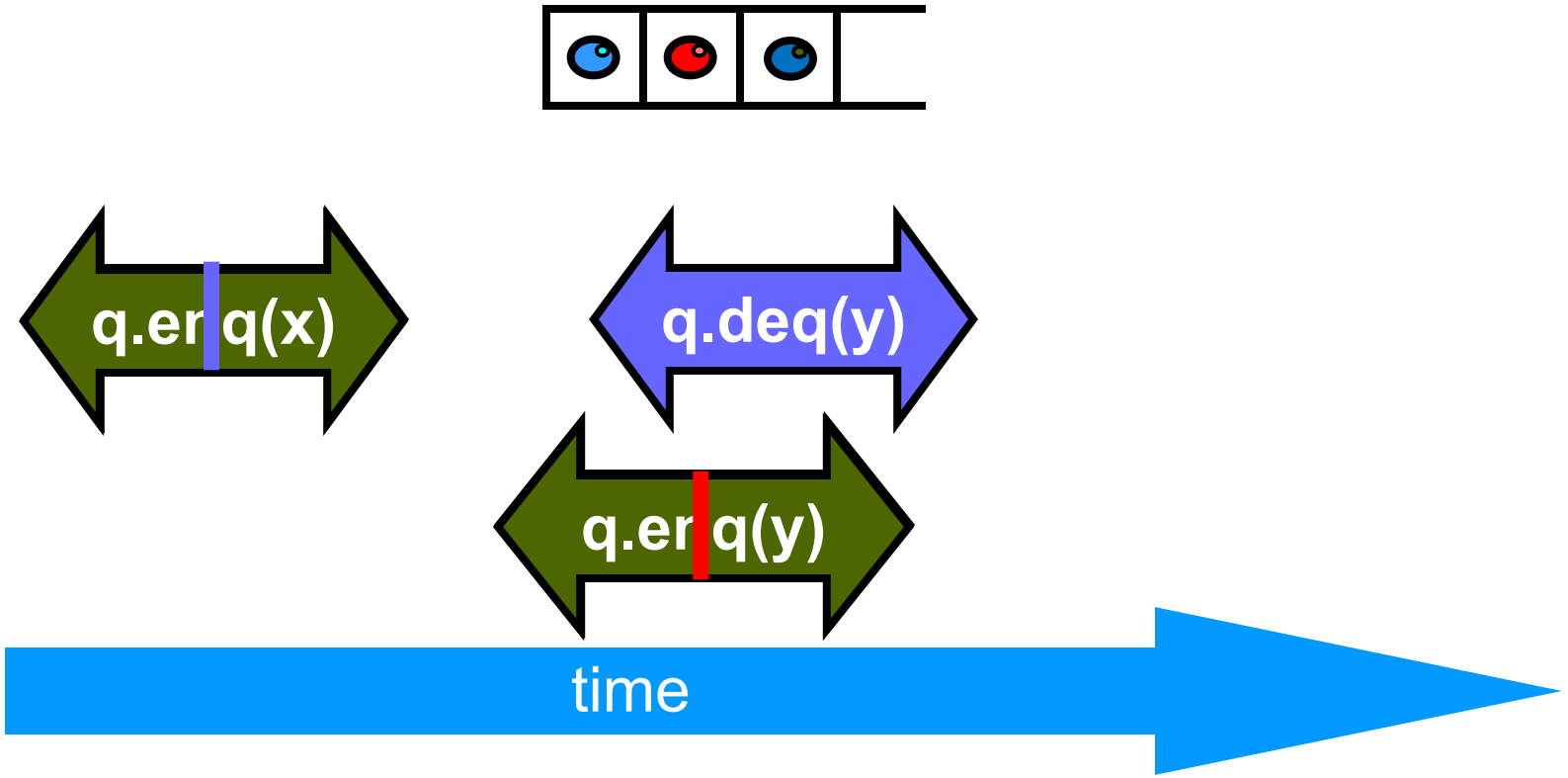
Example



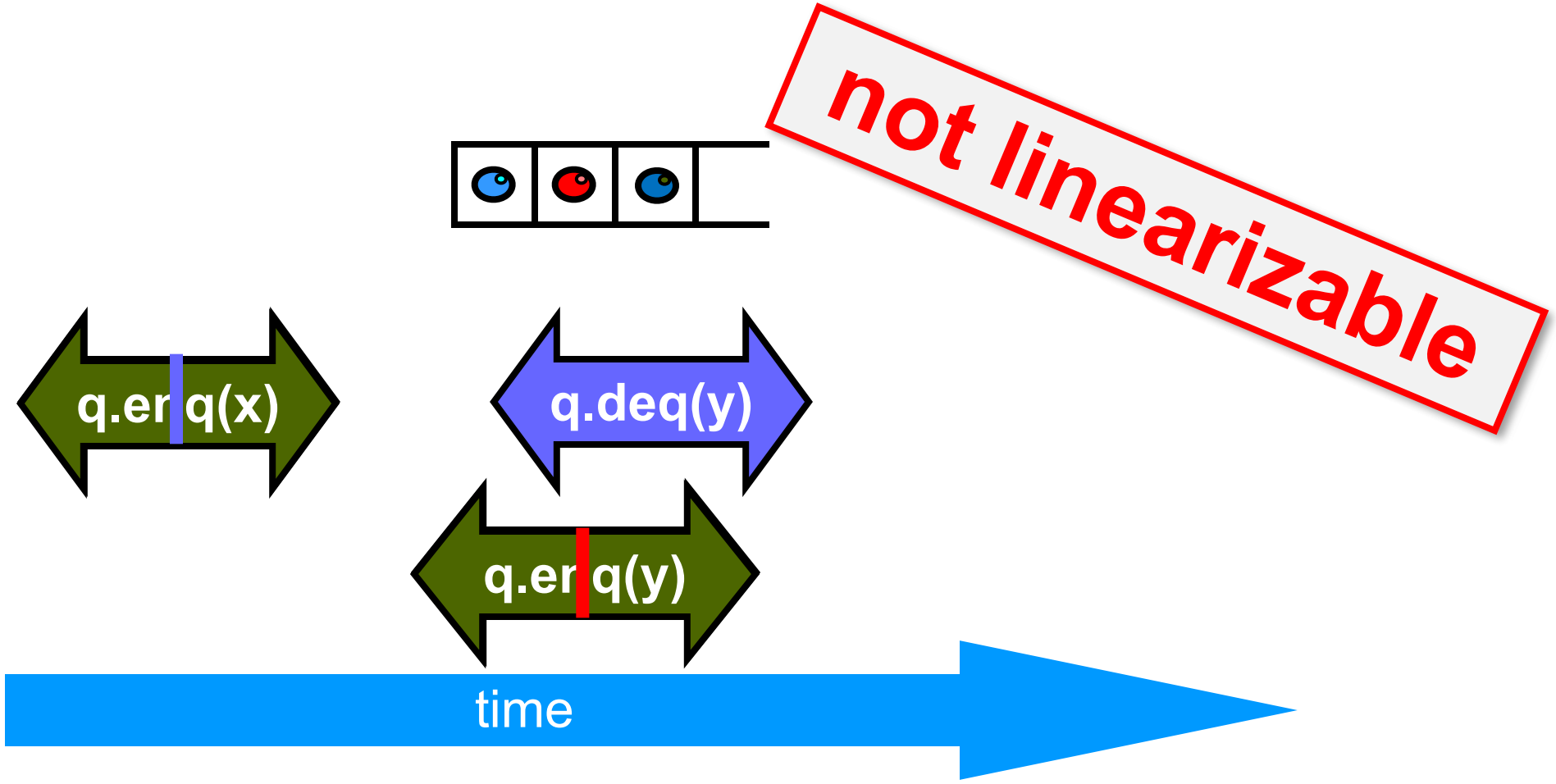
Example



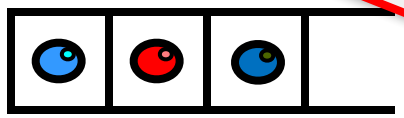
Example

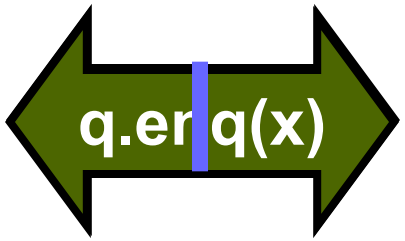


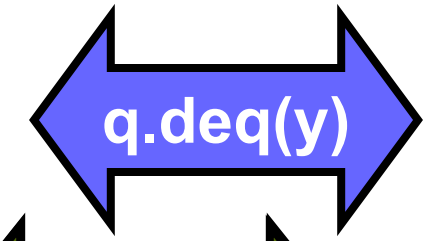
Example

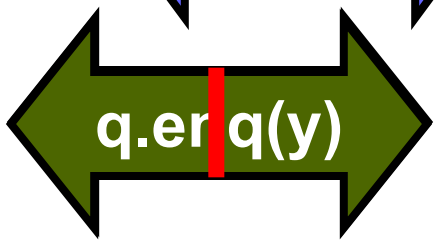


Example







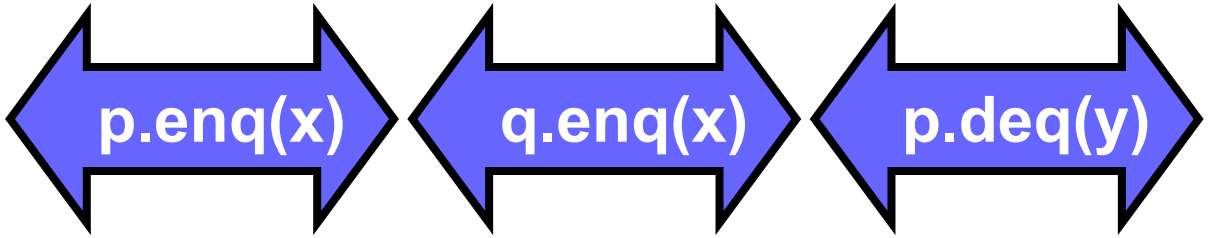


Yet Sequentially Consistent

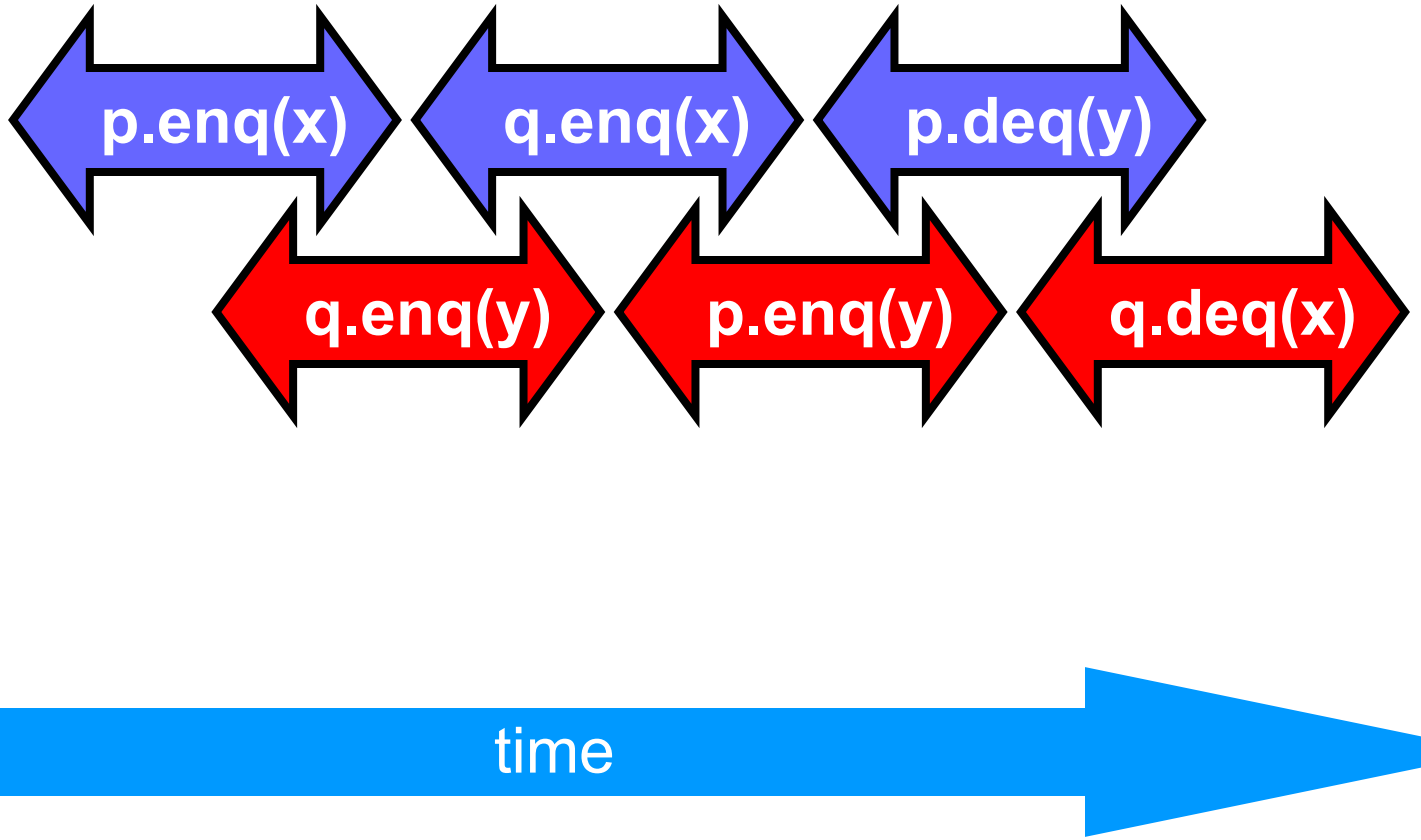
Theorem

Sequential Consistency is not composable

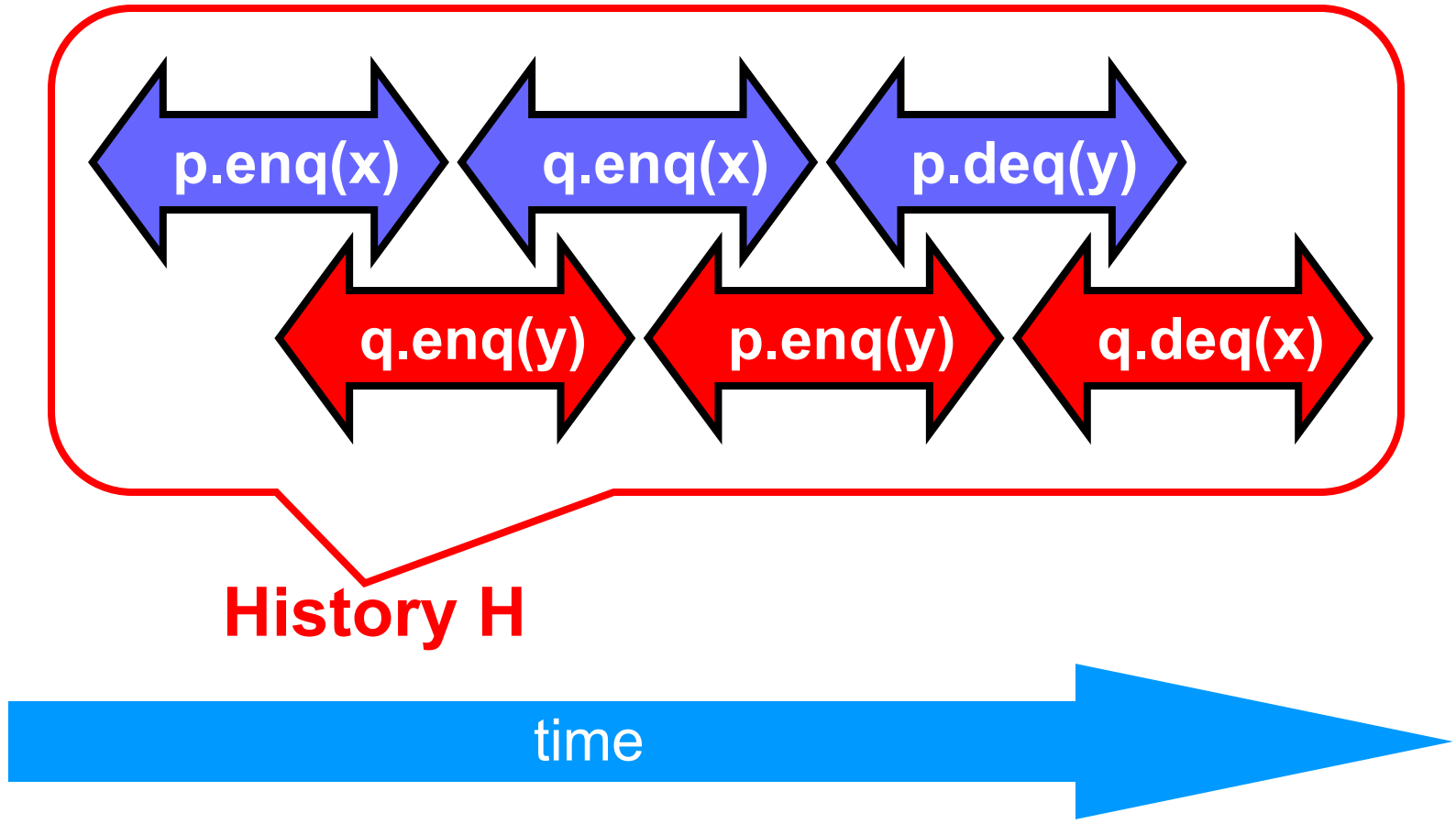
FIFO Queue Example



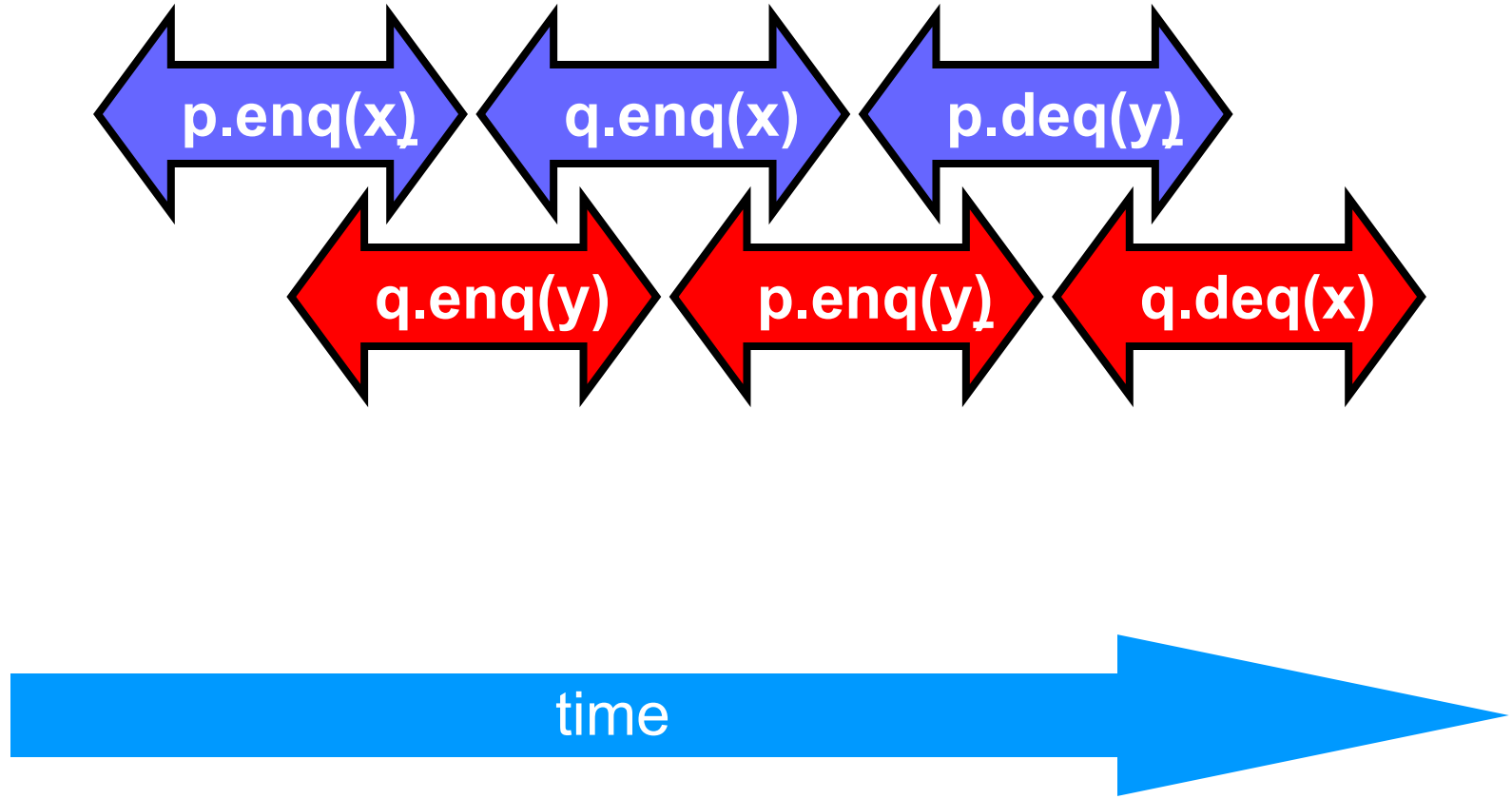
FIFO Queue Example



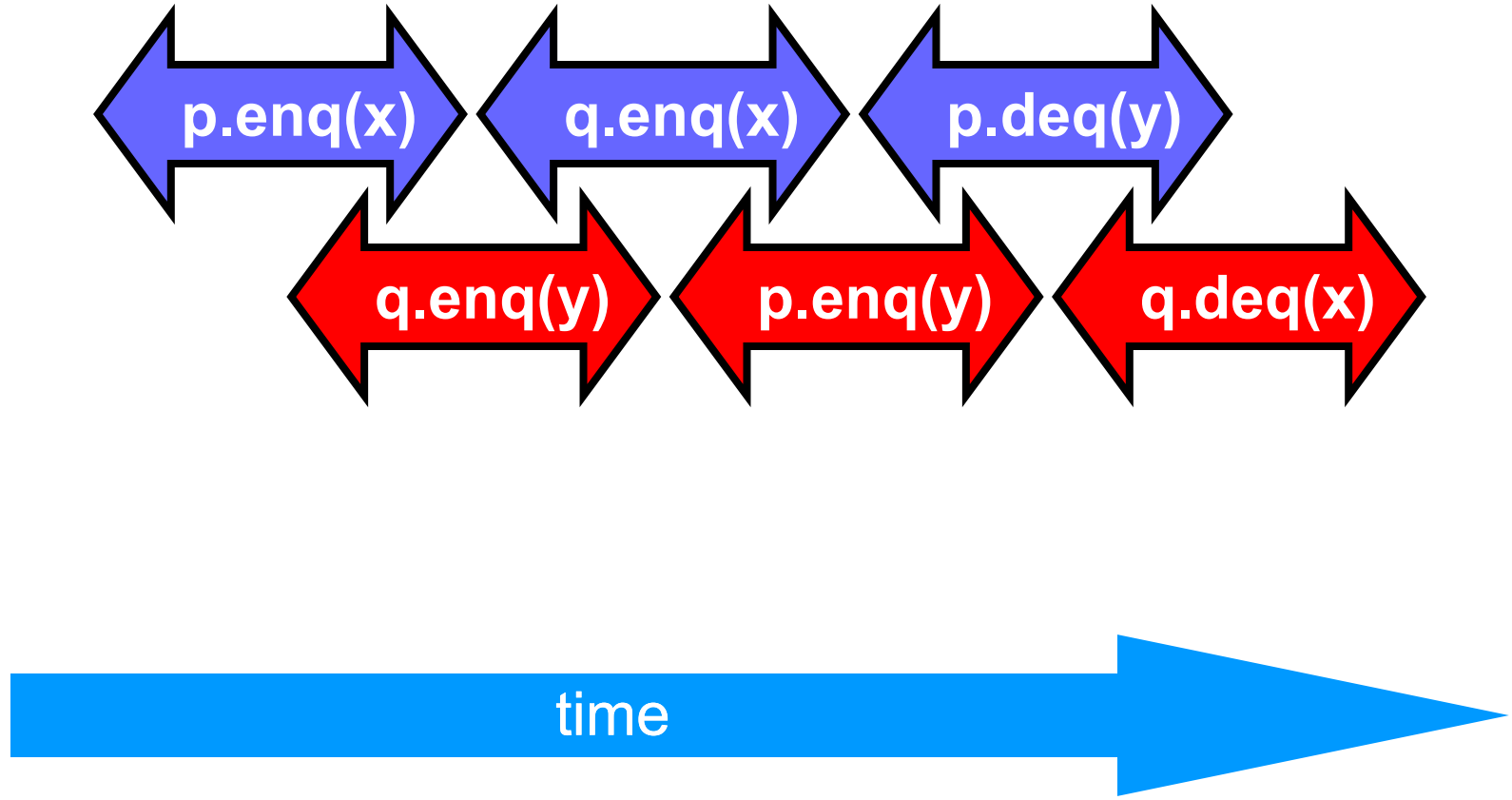
FIFO Queue Example



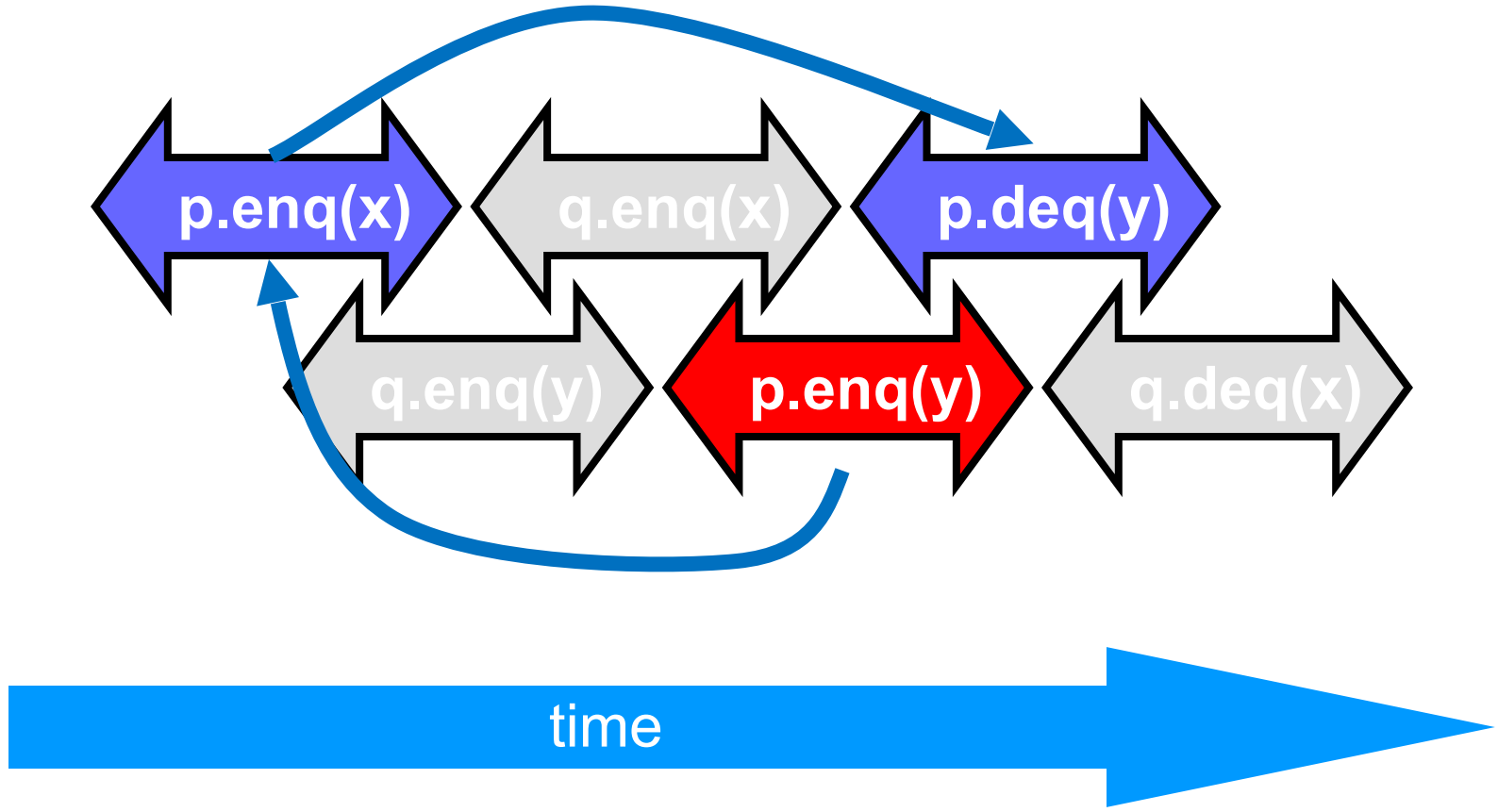
H|p Sequentially Consistent



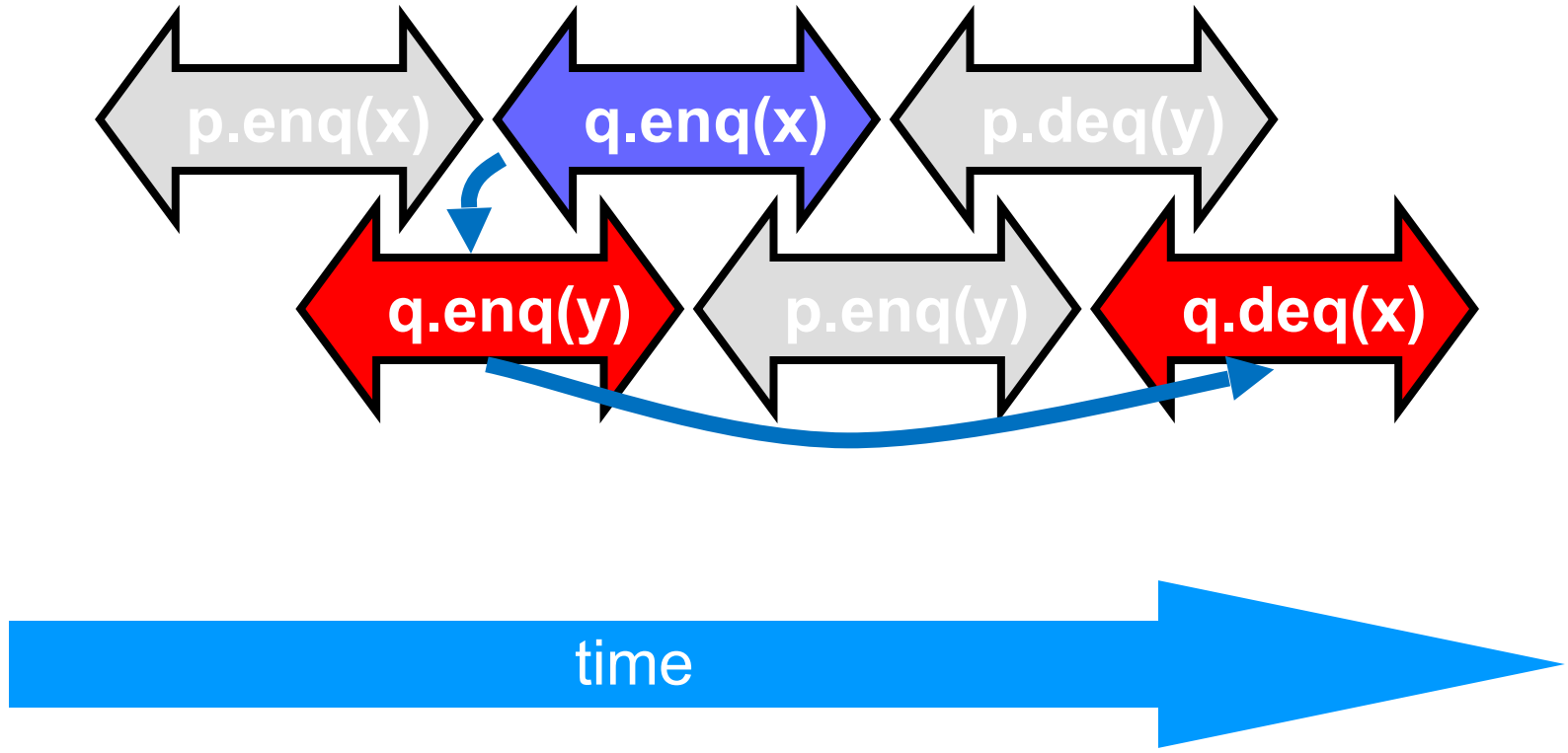
H|q Sequentially Consistent



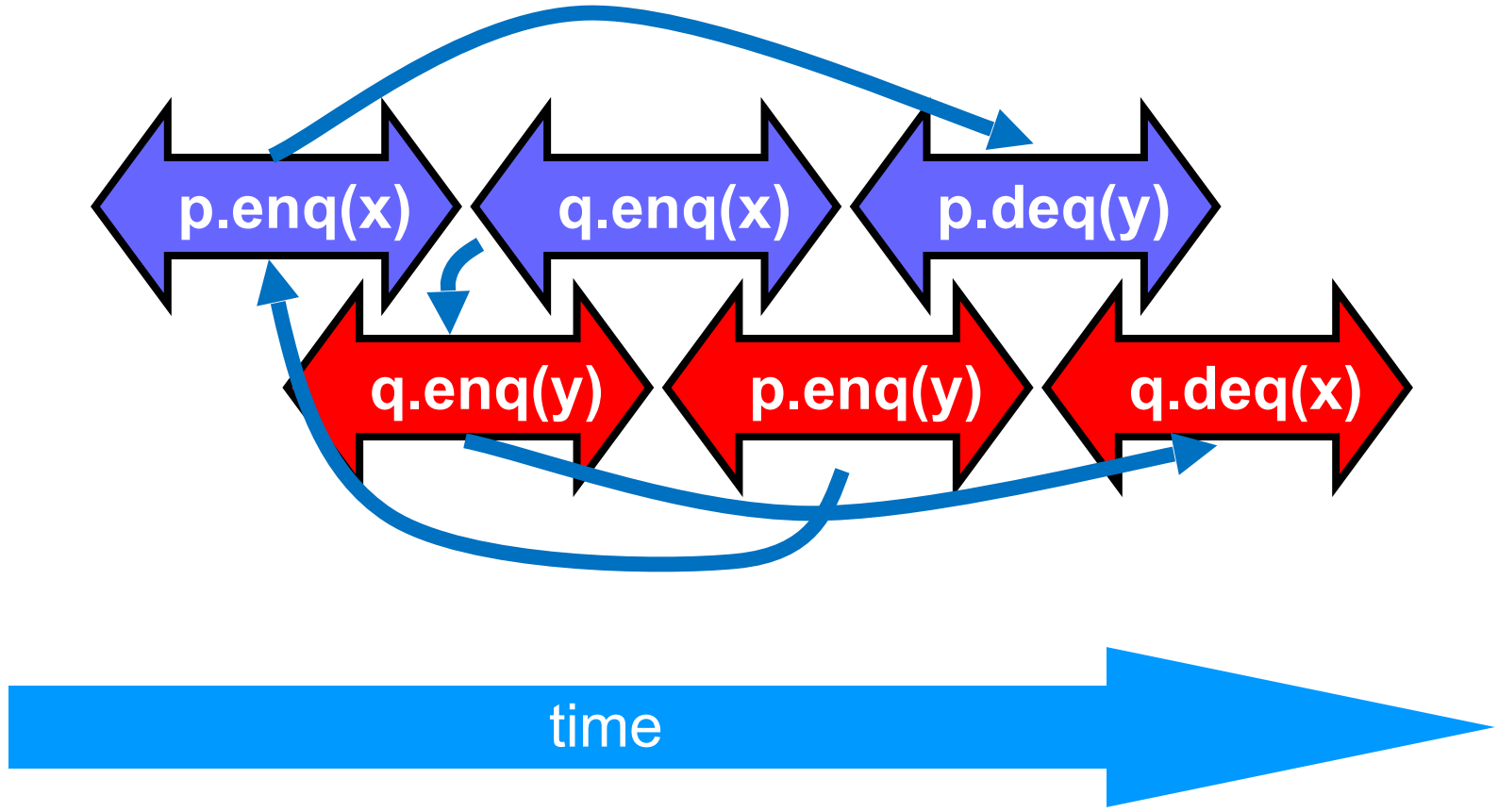
Ordering imposed by p



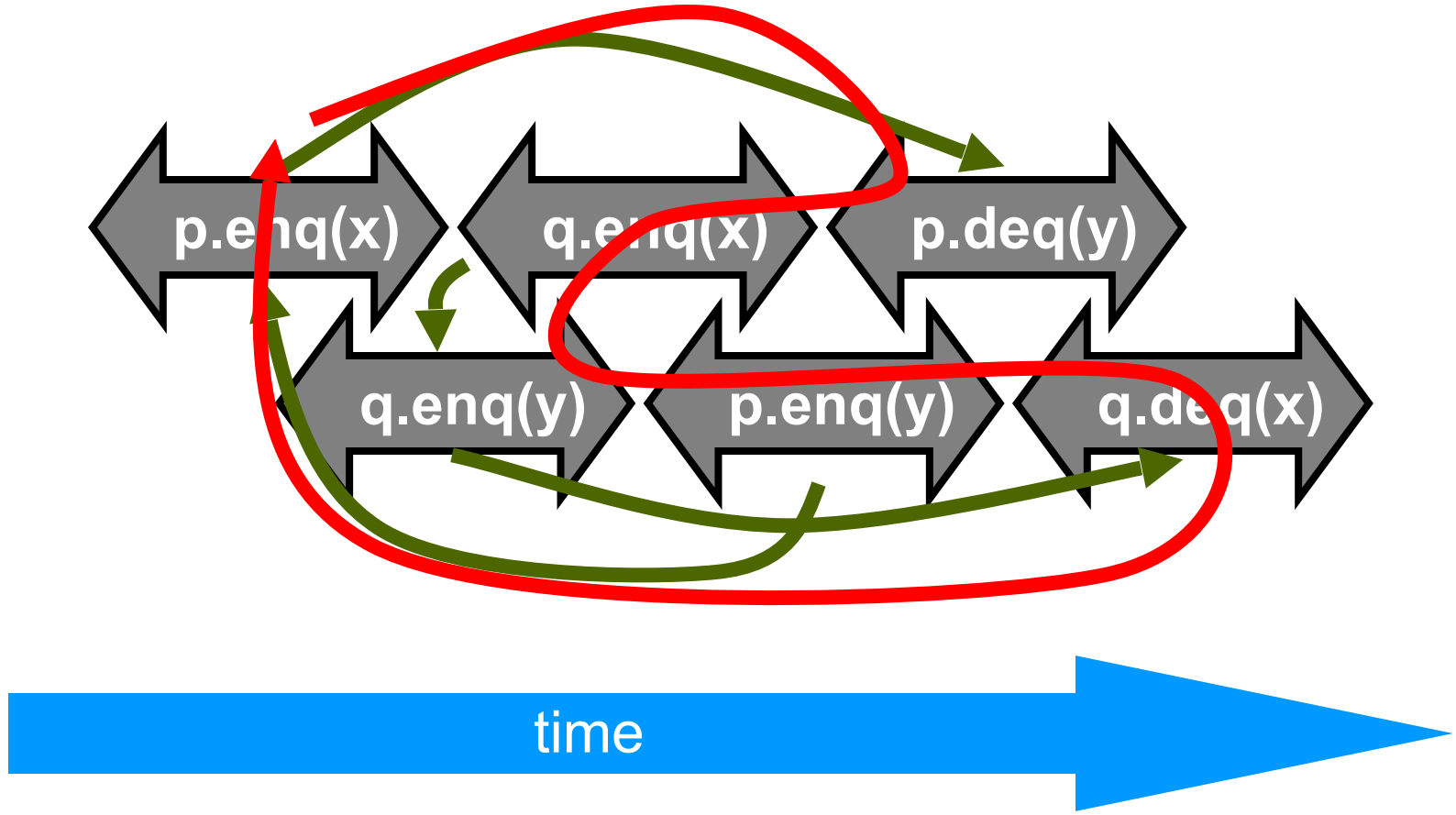
Ordering imposed by q



Ordering imposed by both



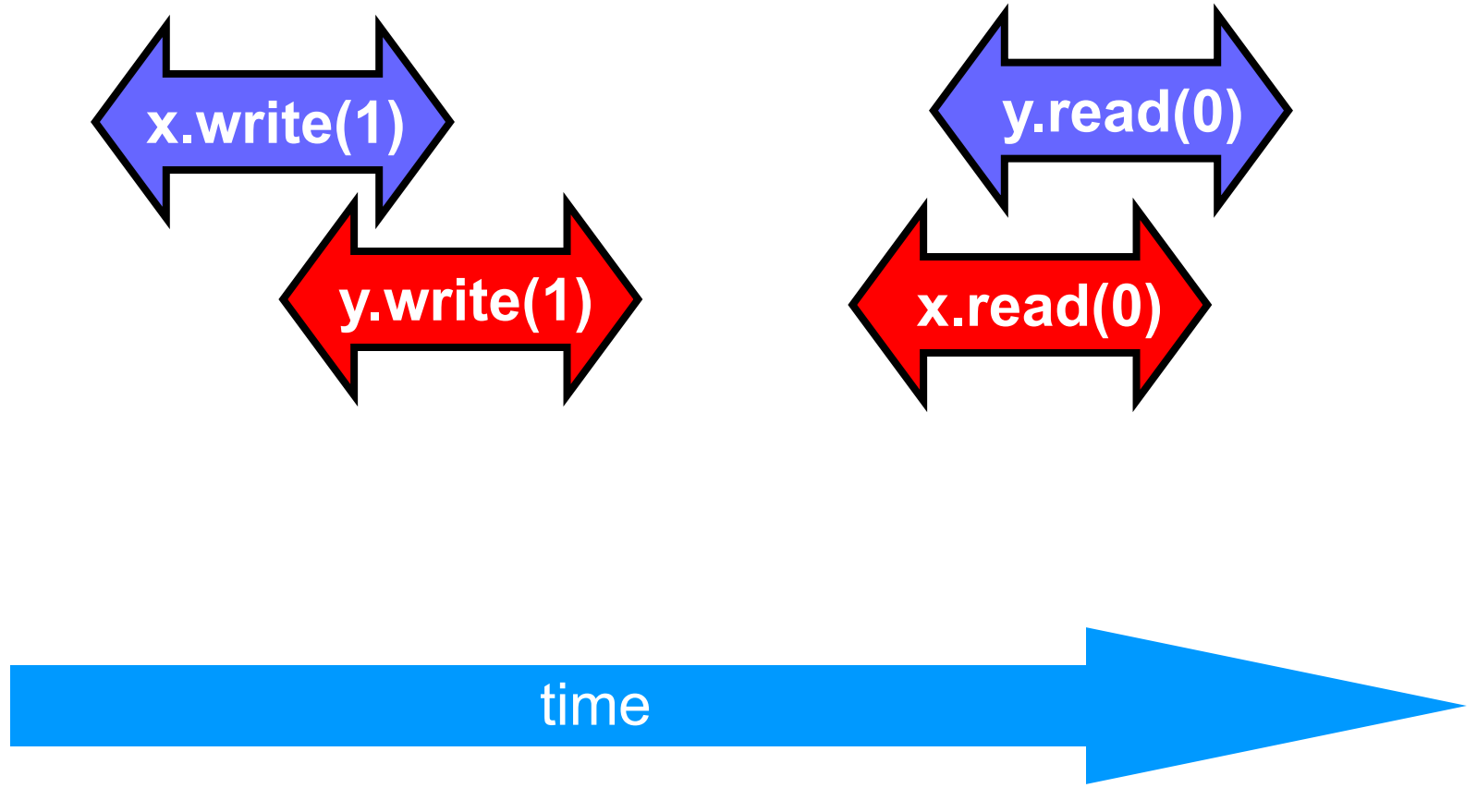
Combining orders



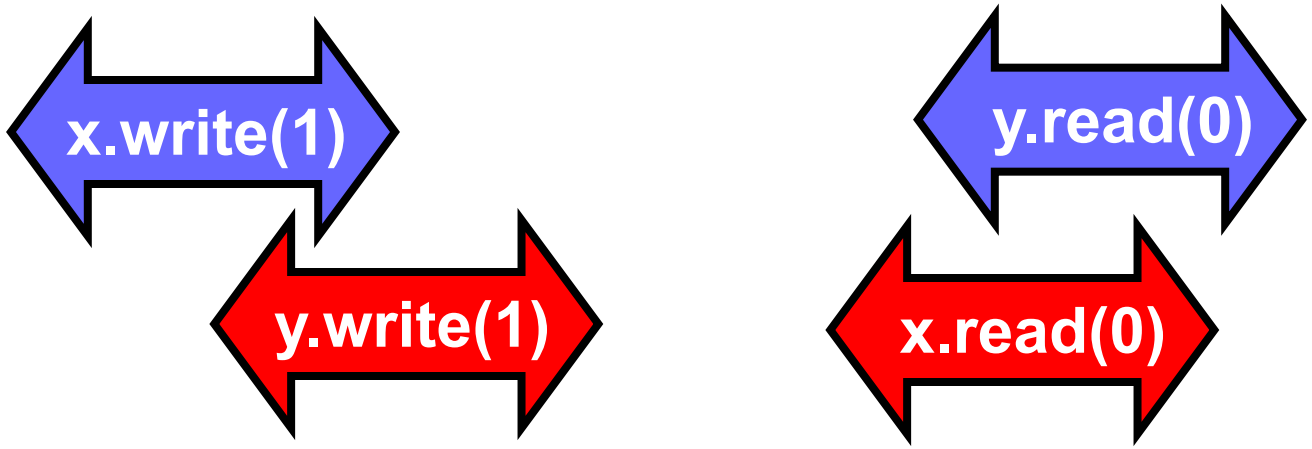
Fact

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

The Flag Example

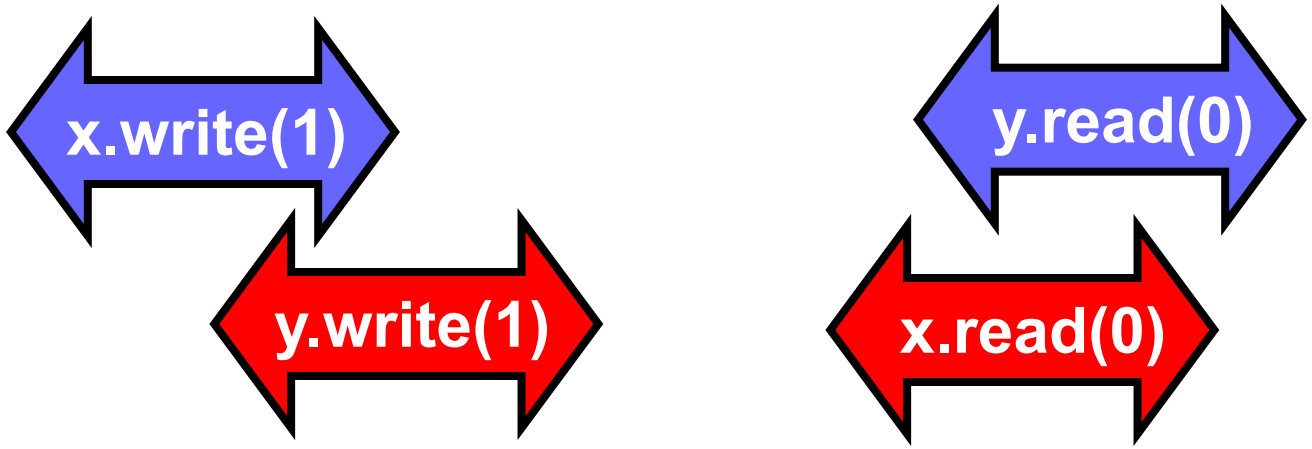


The Flag Example



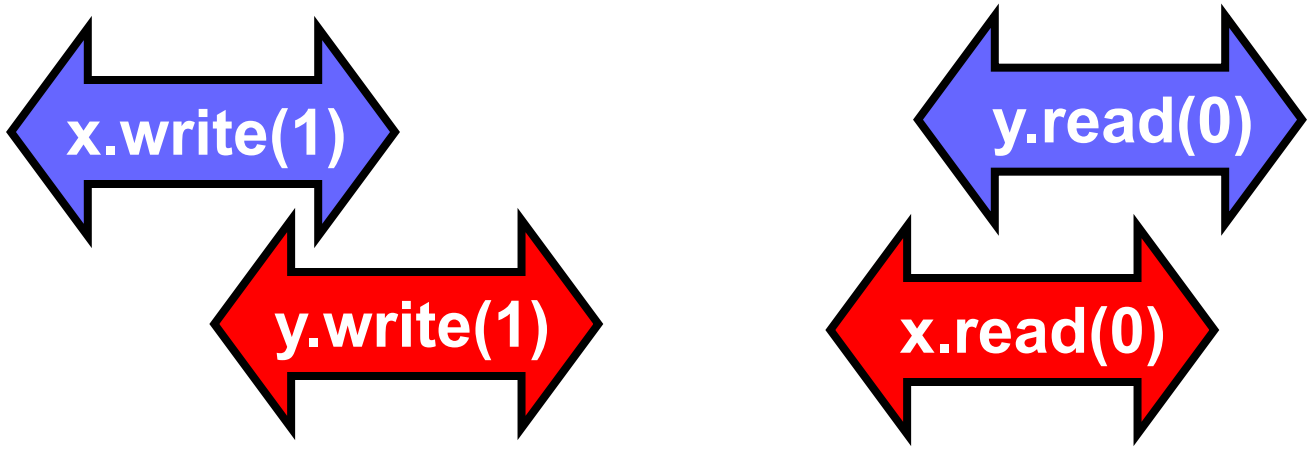
- Each thread's view is sequentially consistent
  - It went first

The Flag Example



- Entire history isn't sequentially consistent
  - Can't *both* go first
  - Petersen't lock now got a problem!

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

```
def lock(): Unit = {
  flag(i) = true
  victim = i
  while (flag(1 - i) && victim == i) {}
}

def unlock(): Unit = {
  val i = ThreadID.get
  flag(i) = false
}
```

Crux of Peterson Proof

(1) $write_B(flag[B]=true)$➔

(3) $write_B(victim=B)$➔

(2) $write_A(victim=A)$➔$read_A(flag[B])$

➔ $read_A(victim)$

Crux of Peterson Proof

(1) $write_B(flag[B]=true)$ →

(3) $write_B(victim=B)$ →

(2) $write_A(victim=A)$ → $read_A(flag[B])$

→ $read_A(victim)$

Observation: 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 Consistency

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?

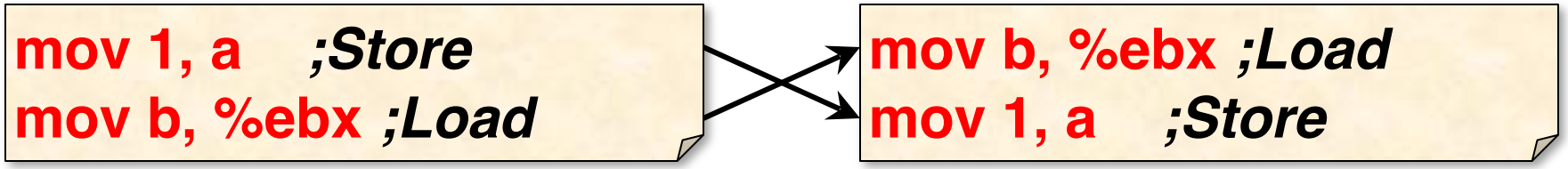
Sequential consistency implies that no execution ends with %eax= %ebx = 0

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!

This is known as **Weak (Relaxed) Memory Semantics**

Weak-Memory Instruction Reordering



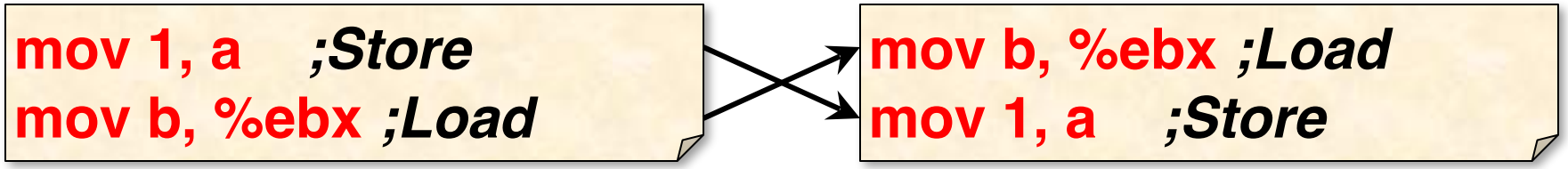
Program Order Execution Order

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

Weak-Memory Instruction Reordering



Program Order Execution Order

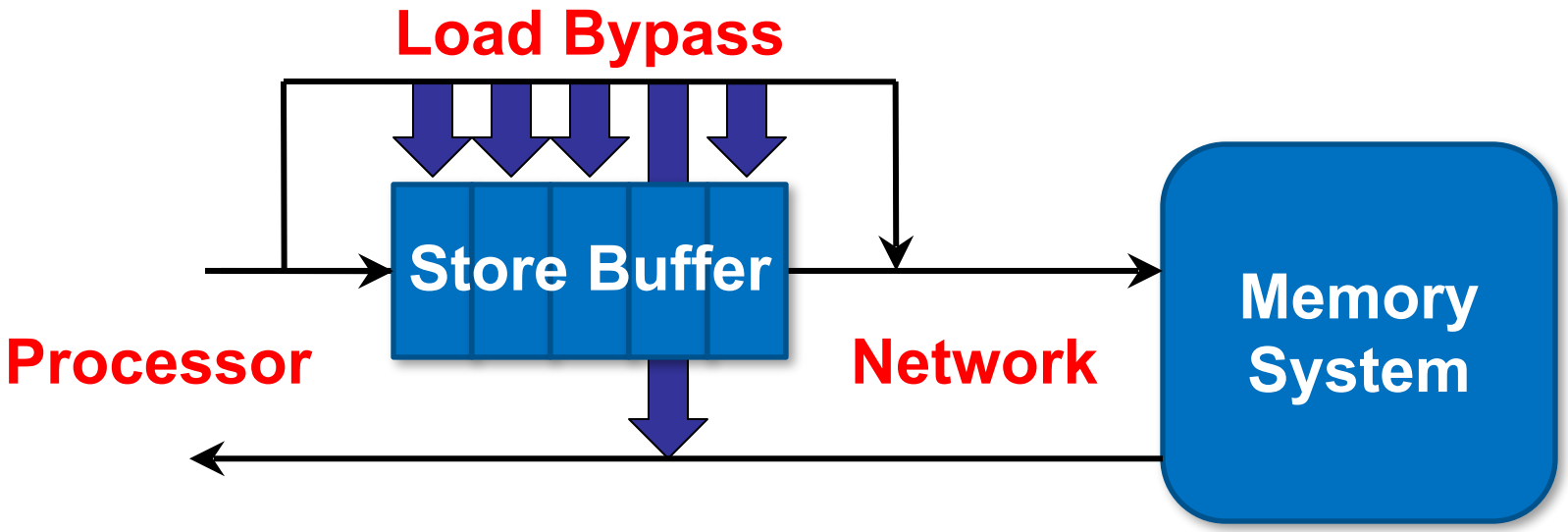
Q. When is it safe for the hardware or compiler to perform this reordering?

A. When a ≠ b.

A′. And there's no concurrency.

Slide used with permission of
Charles E. Leiserson

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.

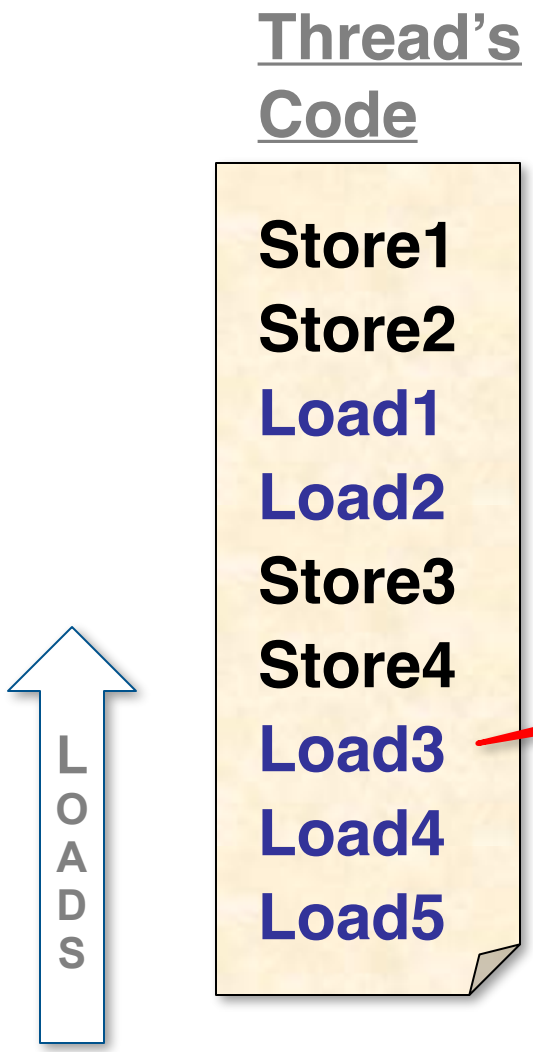
Slide used with permission of
Charles E. Leiserson

X86 Relaxed Memory Model

Thread's Code

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 Relaxed Memory Model



1. Loads are *not* reordered with loads.
2. Stores are *not* reordered with stores.
3. [illegible]or
4. [illegible]rior [illegible]*ot* 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 Relaxed Memory Model

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5

1. Loads
2. Store
3. Store
   loads
4. A loa
   store
   with a pri 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 via Java's synchronized
  - Also, enforced by library implementations of lock/unlock()

Java/Scala Volatile Variables

- In Java, can ask compiler to keep a variable up-to-date by declaring it **volatile**
- In Scala, use **@volatile** annotation
- Adds a memory barrier after each store
- Inhibits reordering, removing from loops, & other “compiler optimizations”

Summary: Real-World

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

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

<A good place for a break>

Checkpoint

- Defined concurrent objects using linearizability and sequential consistency
- Fact: implemented linearizable objects (Two thread FIFO Queue) in read-write memory without mutual exclusion
- Fact: hardware does not provide linearizable read-write memory

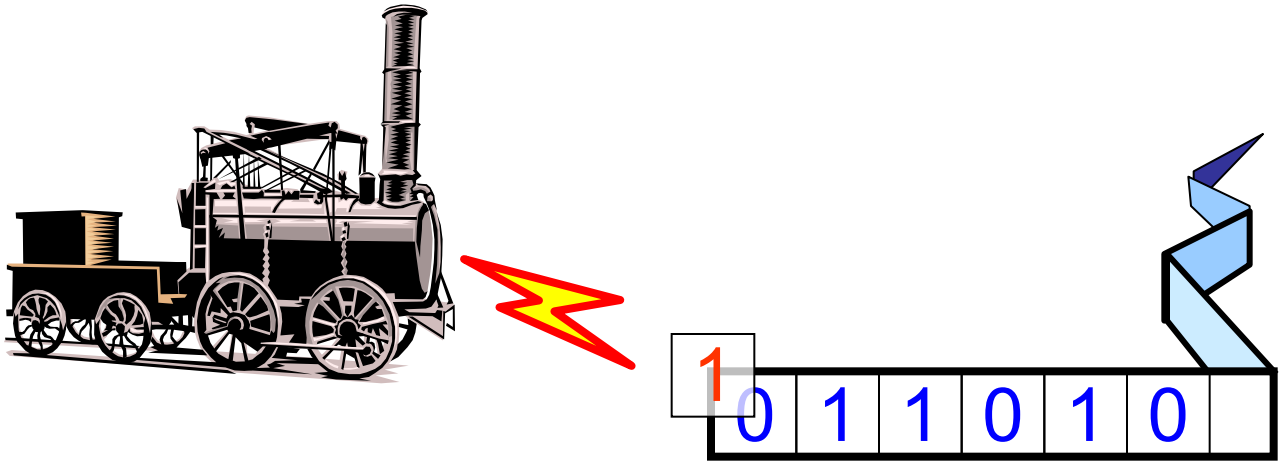
Fundamentals

- What is the weakest form of communication that supports mutual exclusion?
- What is the weakest shared object that allows shared-memory computation?

Alan Turing

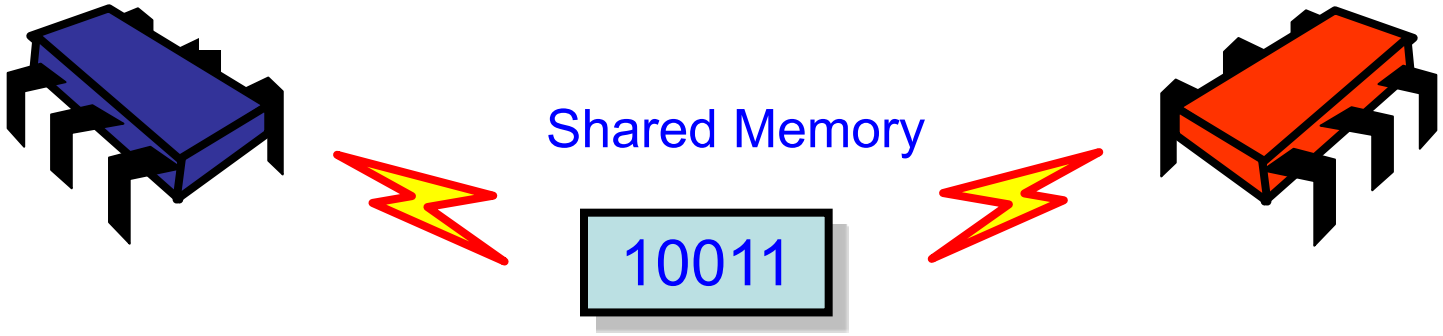
- Showed what is and is not computable on a sequential machine.
- Still best model there is.

Turing Computability



- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

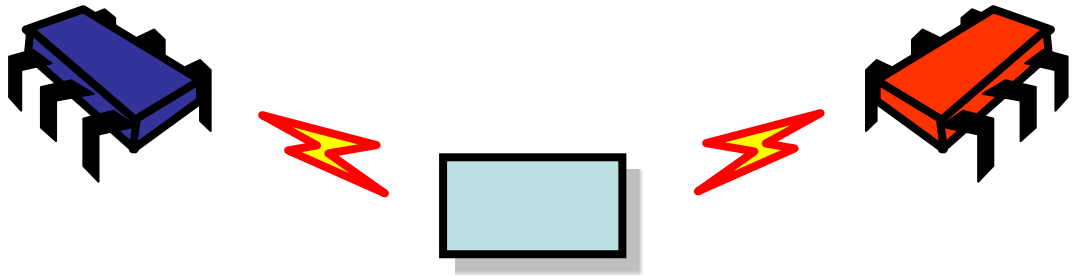
Shared-Memory Computability?



- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

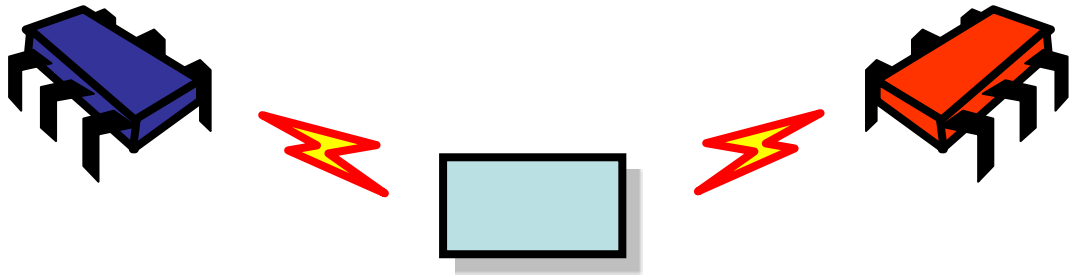
Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions …



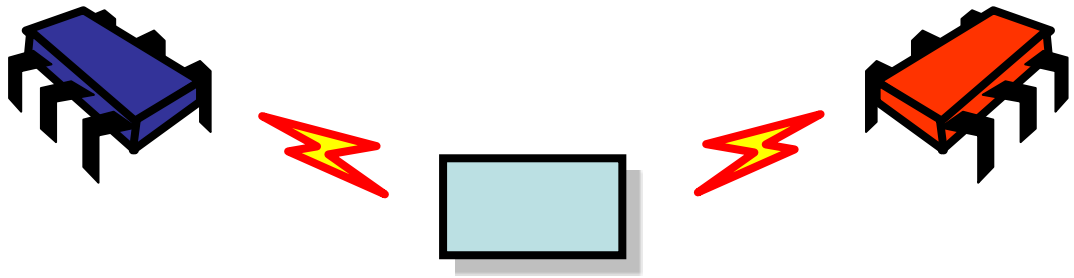
Foundations of Shared Memory

What is the weakest useful form of shared memory?

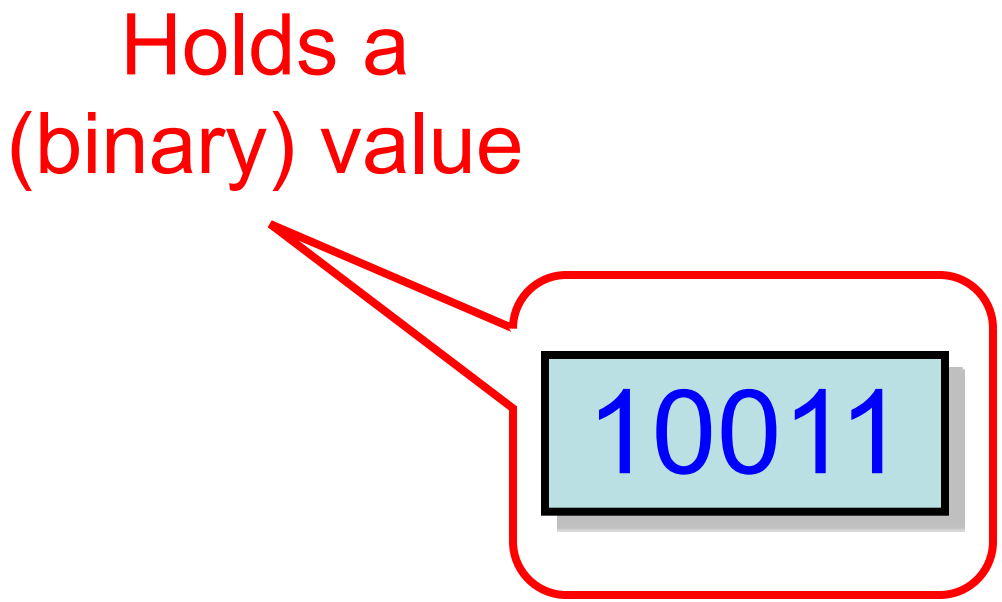


Foundations of Shared Memory

What can it do?

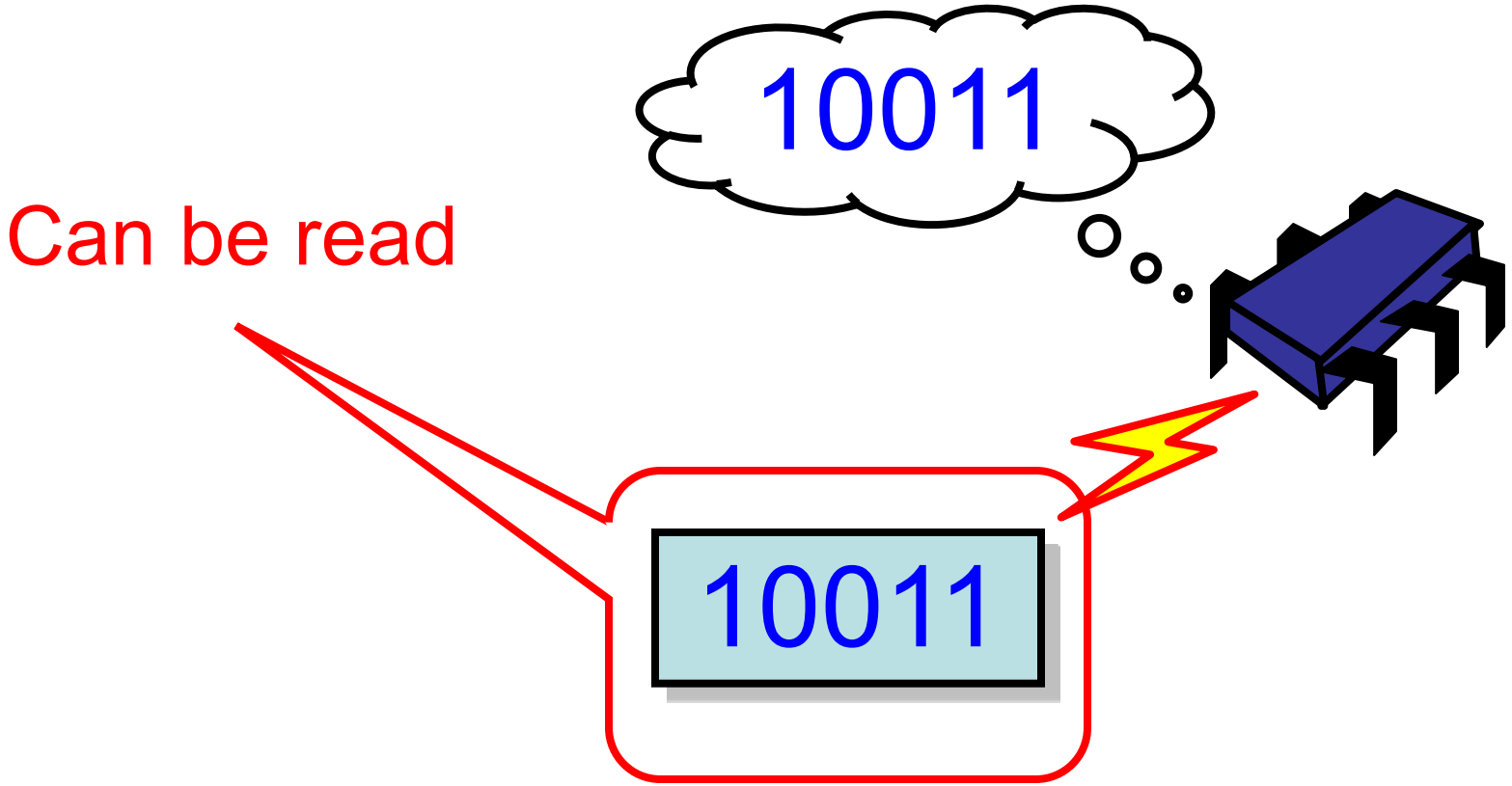


Register*

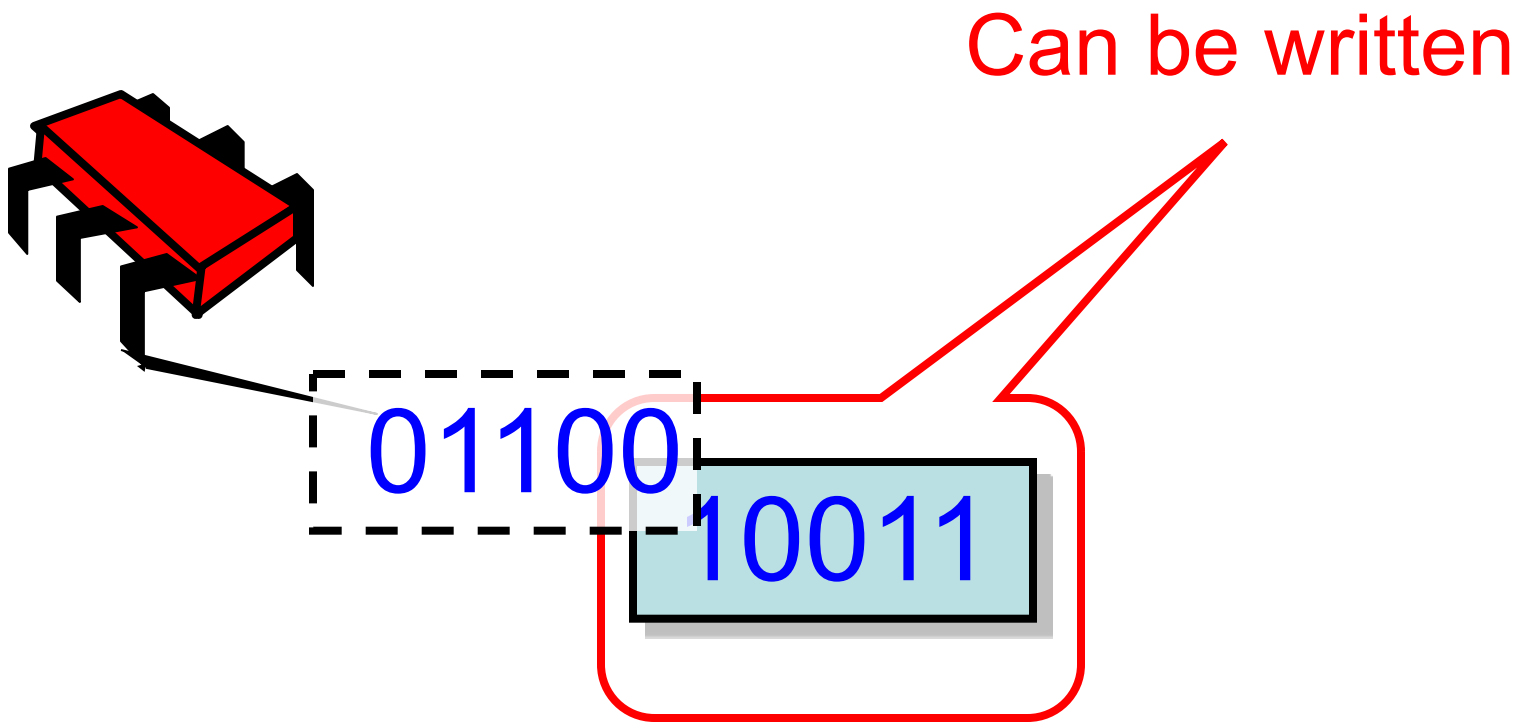


*** A memory location: name is historical**

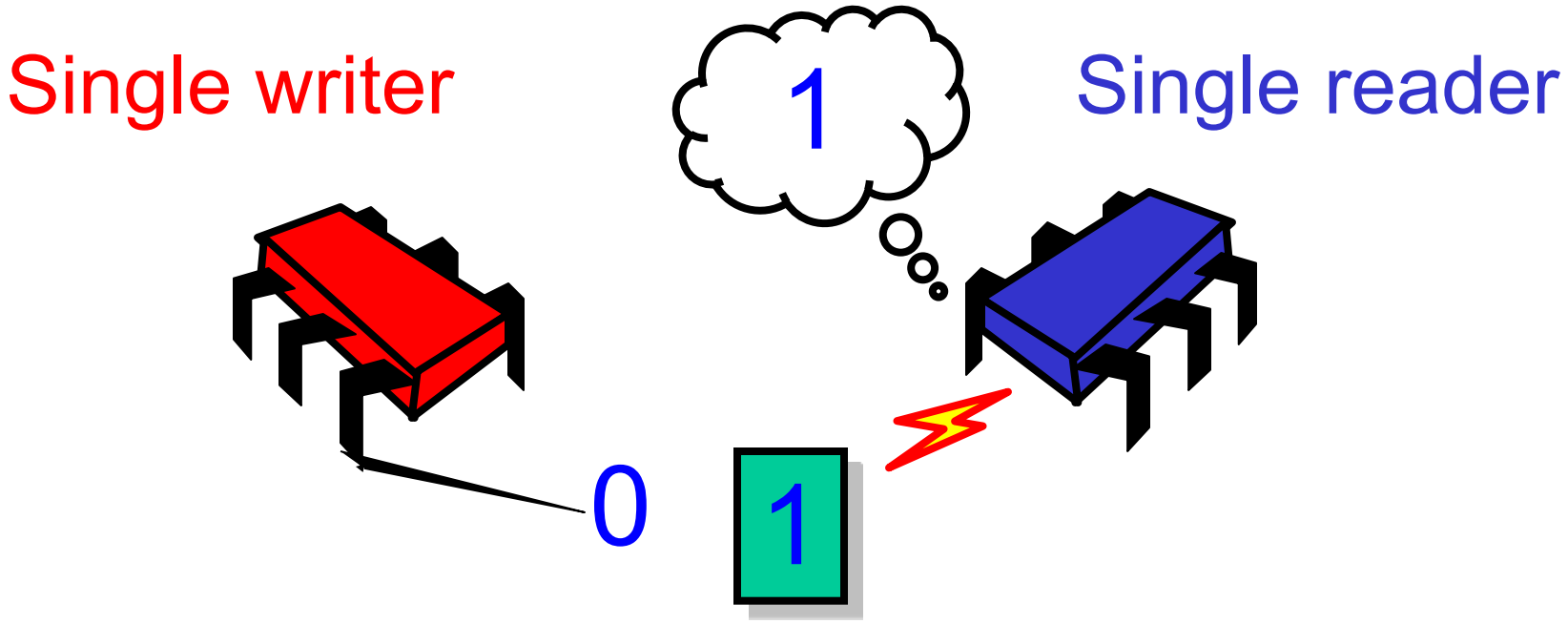
Register



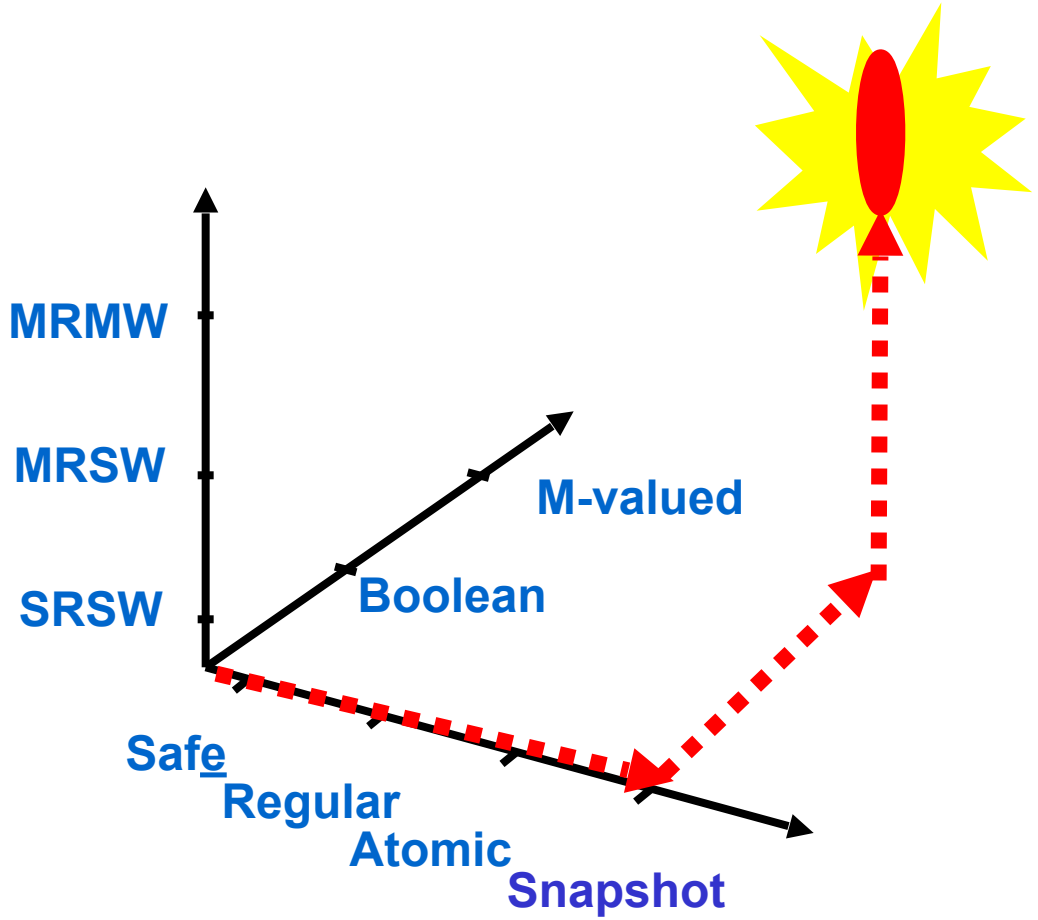
Register

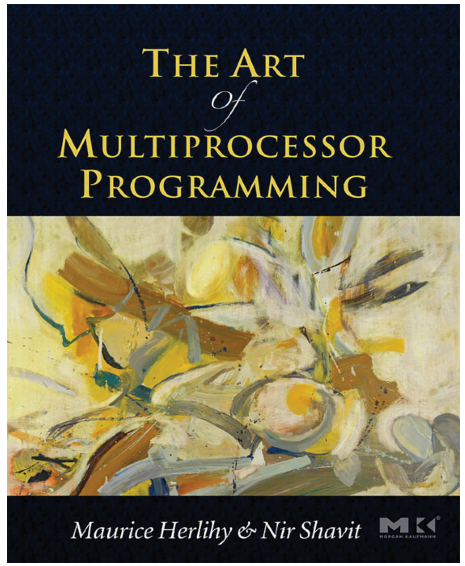


From Weakest Register



All the way to a Wait-free Implementation of Atomic Snapshots





Chapter 4

Crux of Concurrency: Wait-Free Synchronization

- Every method call completes in finite number of steps
- Implies no mutual exclusion

Rationale for wait-freedom

- We wanted atomic registers to implement mutual exclusion

Rationale for wait-freedom

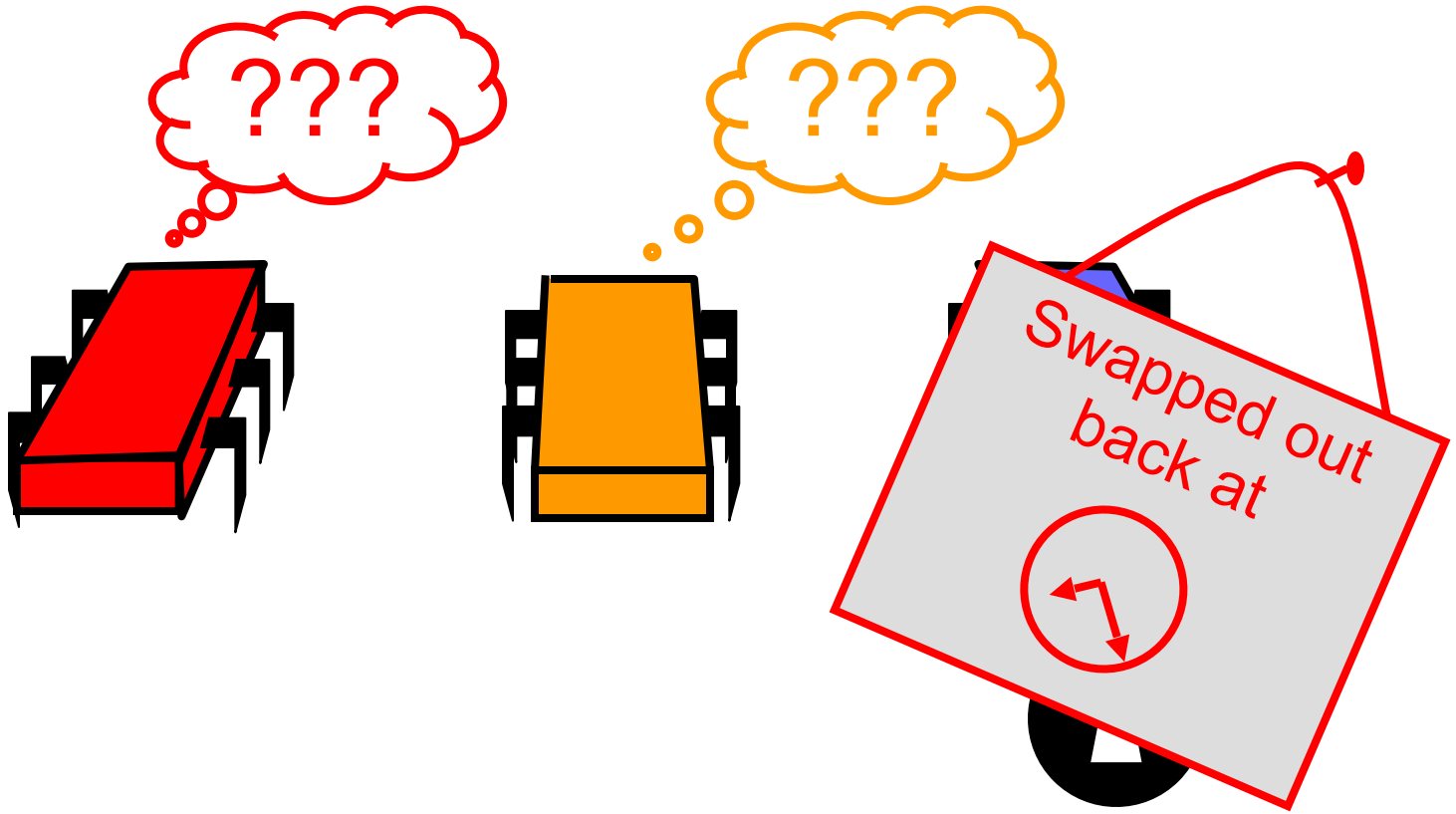
- We wanted atomic registers to implement mutual exclusion
- So we couldn't use mutual exclusion to implement atomic registers

Rationale for wait-freedom

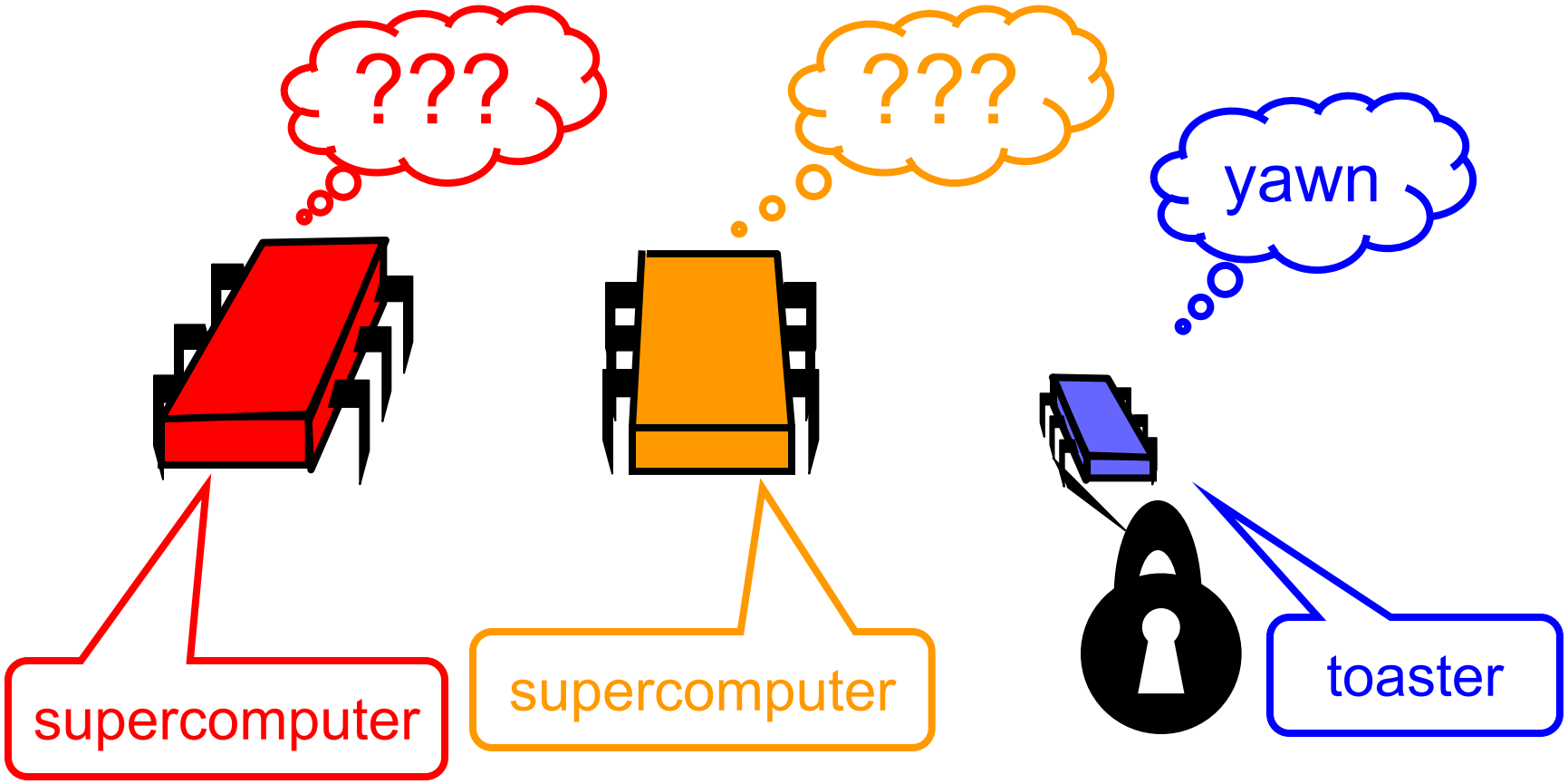
- We wanted atomic registers to implement mutual exclusion
- So we couldn't use mutual exclusion to implement atomic registers
- But wait, there's more!

What's the problem with Mutual Exclusion?

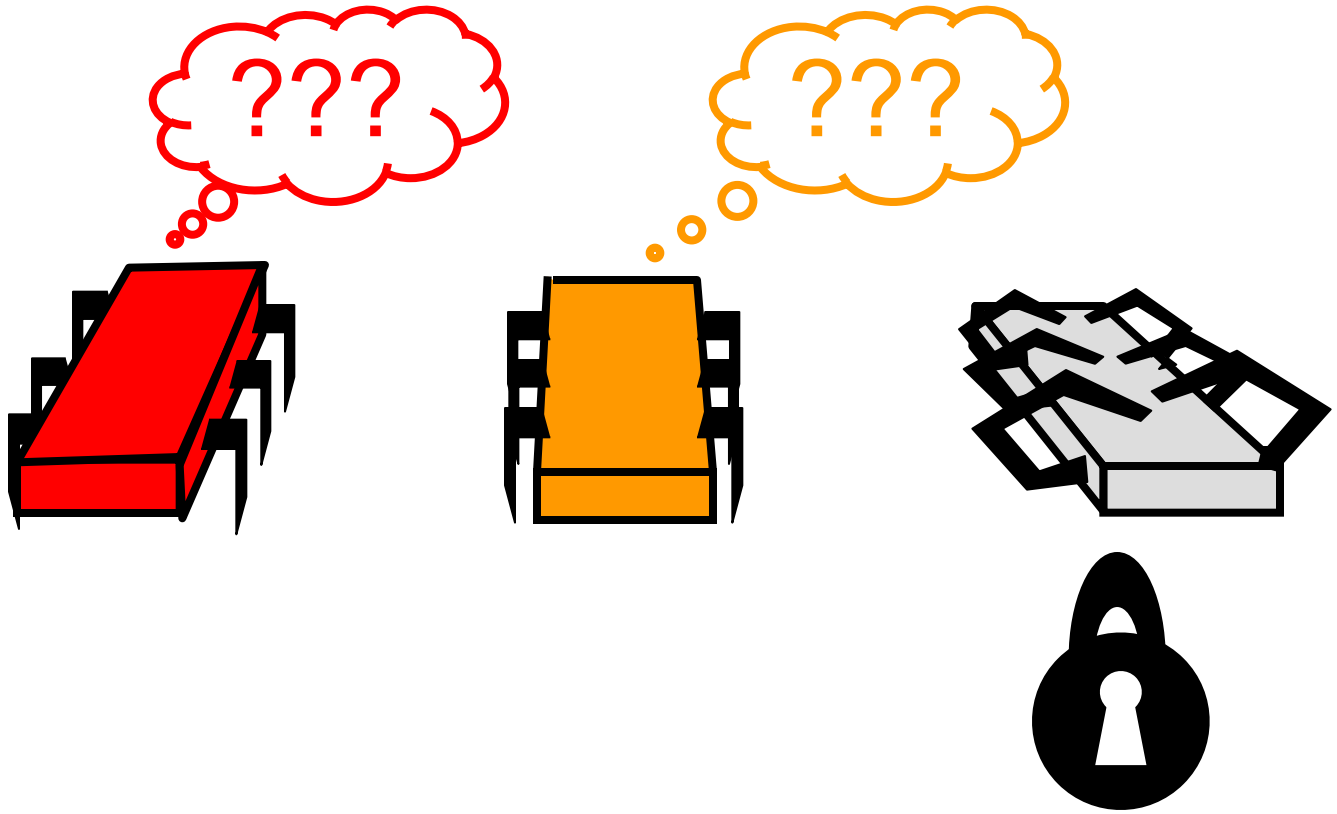
Asynchronous Interrupts



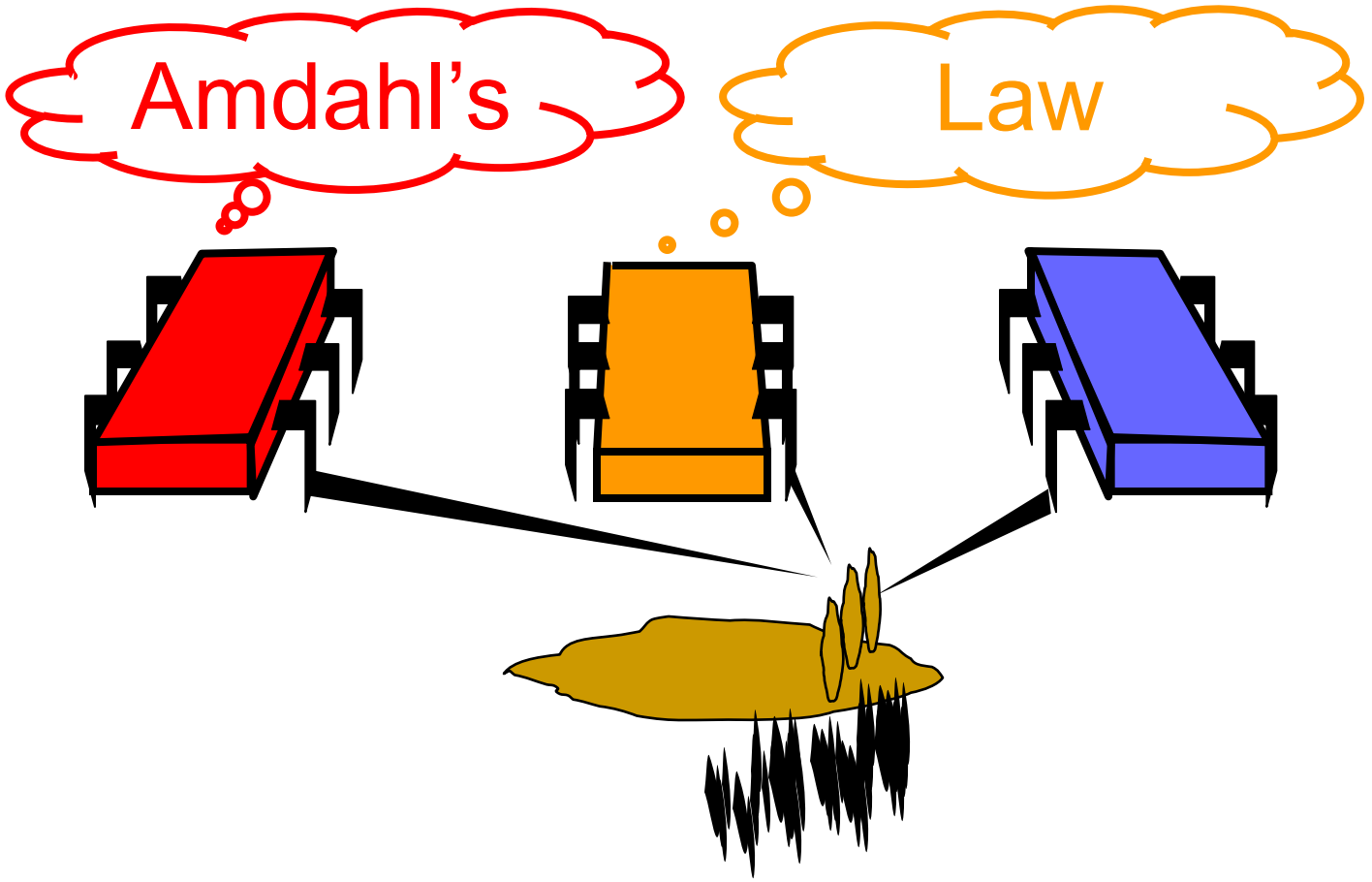
Heterogeneous Processors



Fault-tolerance



Machine Level Instruction Granularity



Basic Questions

- Can we syncrhonize without ME?

Basic Questions

- Can we syncrhonize threads without ME?
- Wait-Free synchronization might be a good idea in principle

Basic Questions

- Can we syncrhonize threads without ME?
- Wait-Free synchronization might be a good idea in principle
- But how do you do it …

Basic Questions

- Can we syncrhonize threads without ME?
- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?

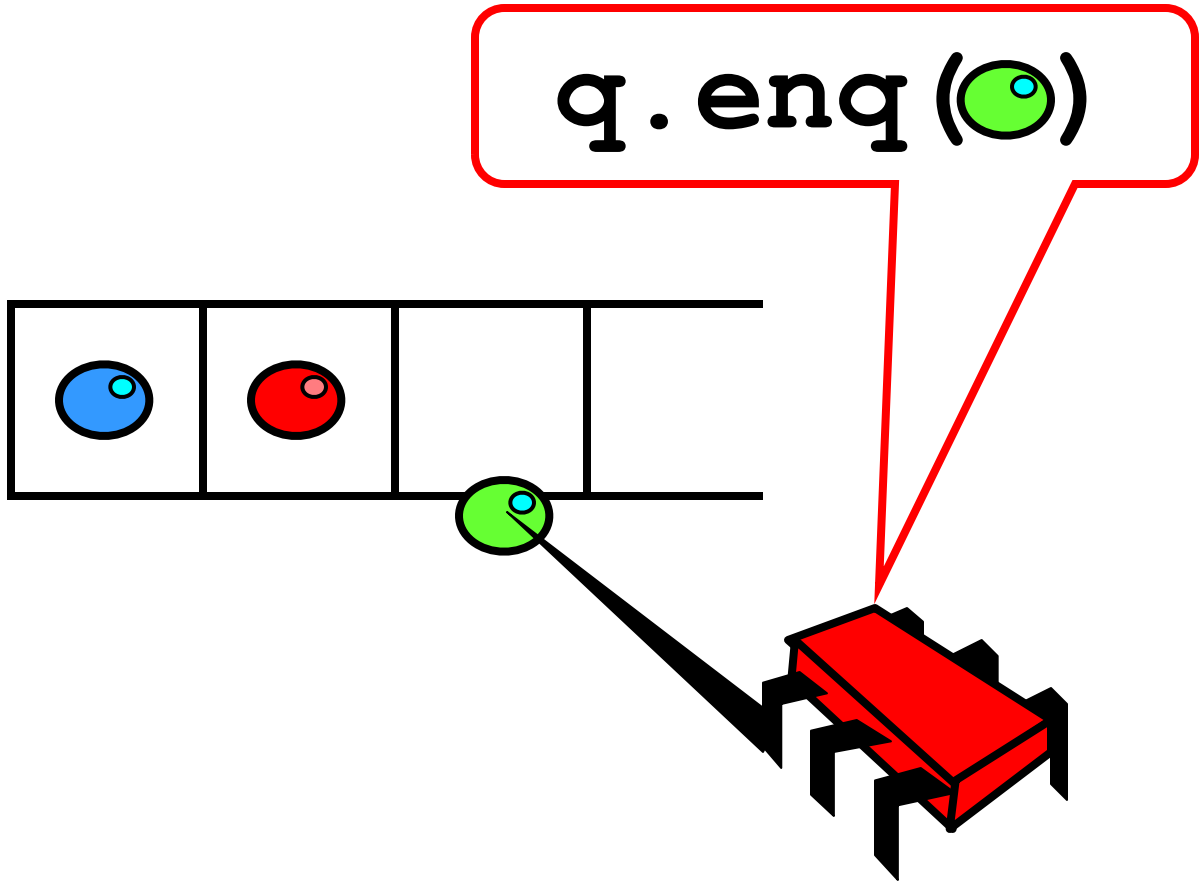
Basic Questions

- Can we syncrhonize threads without ME?
- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?

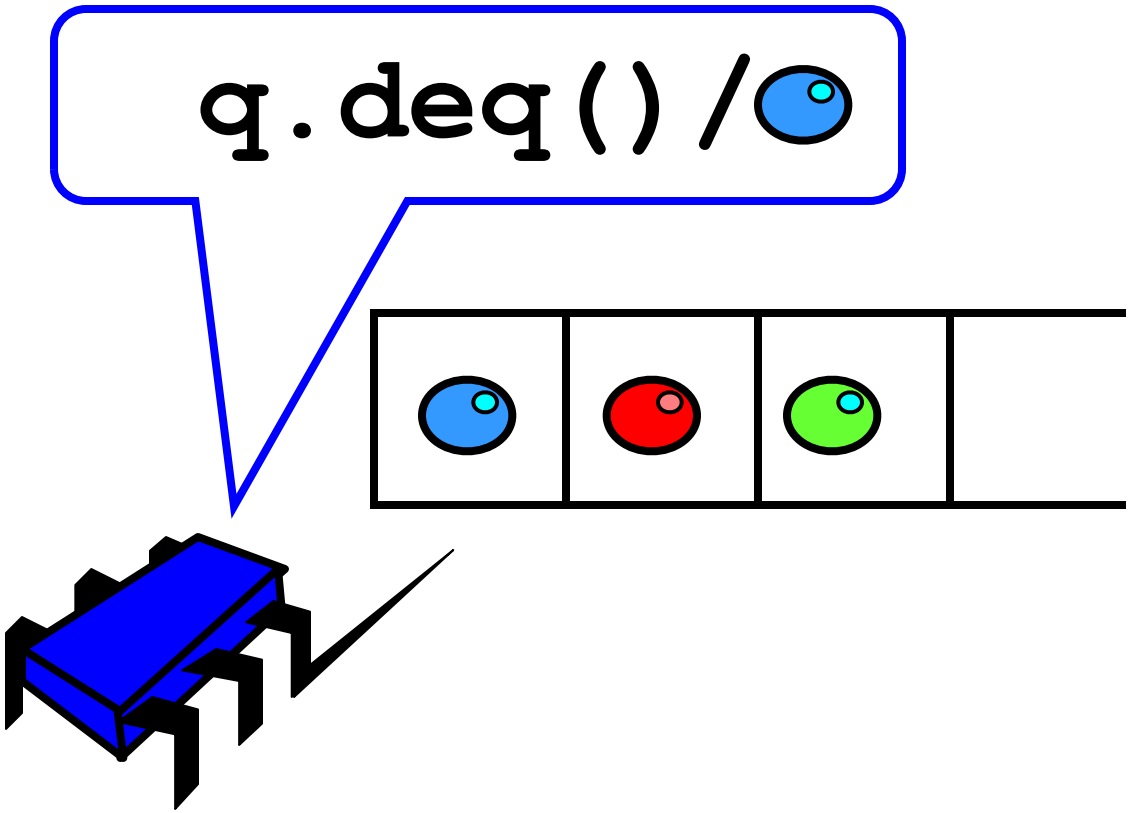
Basic Questions

- Can we synchronize threads without ME?
- Wait-Free synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?
  - Efficiently?

FIFO Queue: Enqueue Method

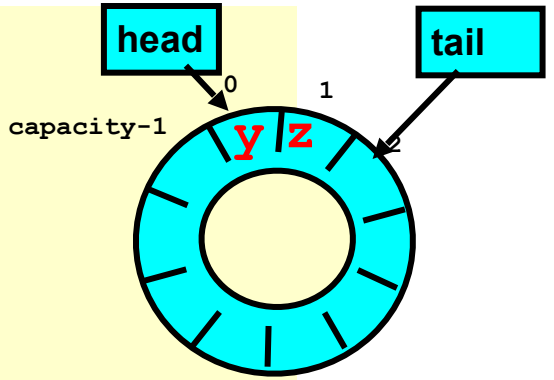


FIFO Queue: Dequeue Method

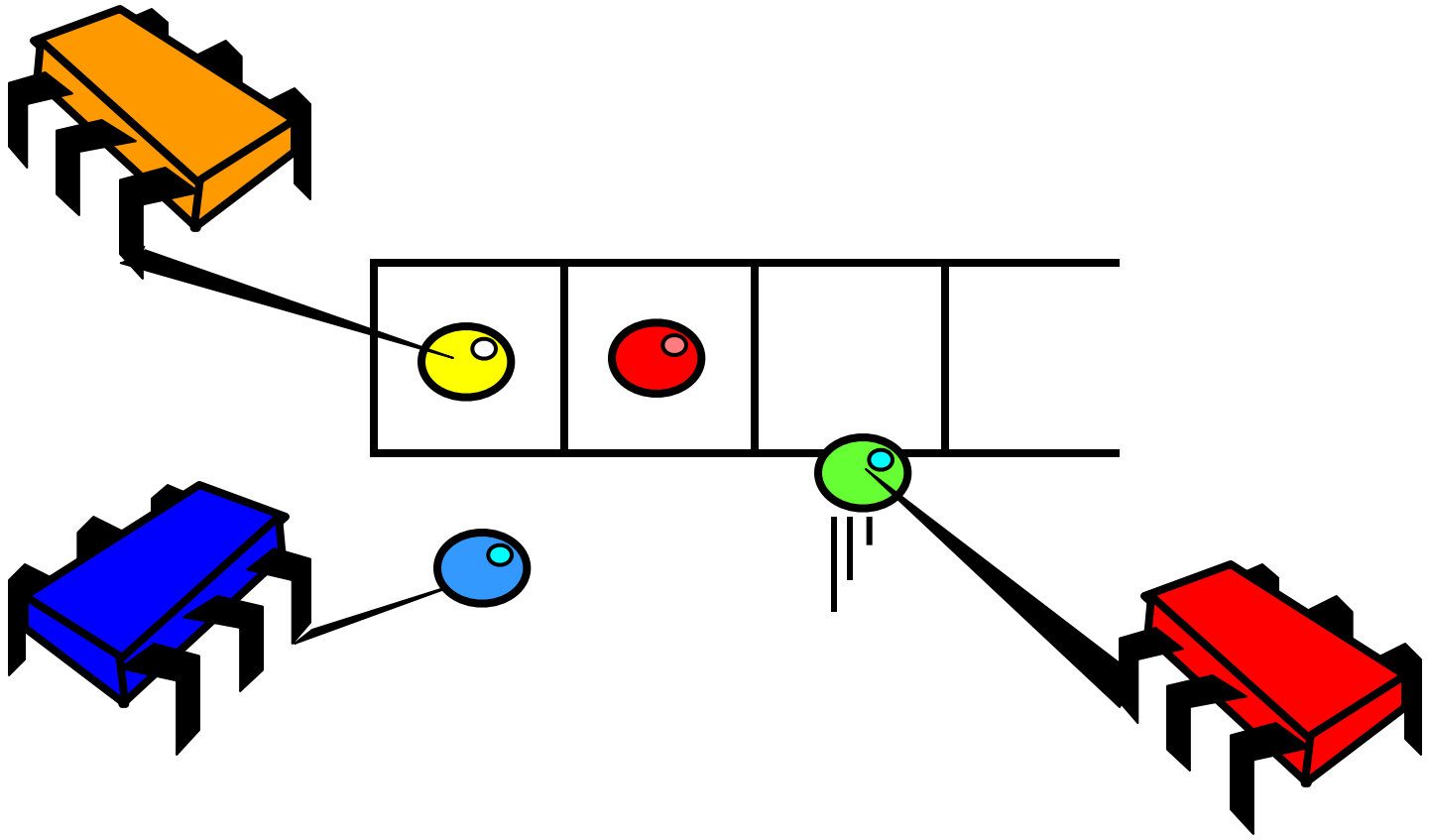


Two-Thread Wait-Free Queue

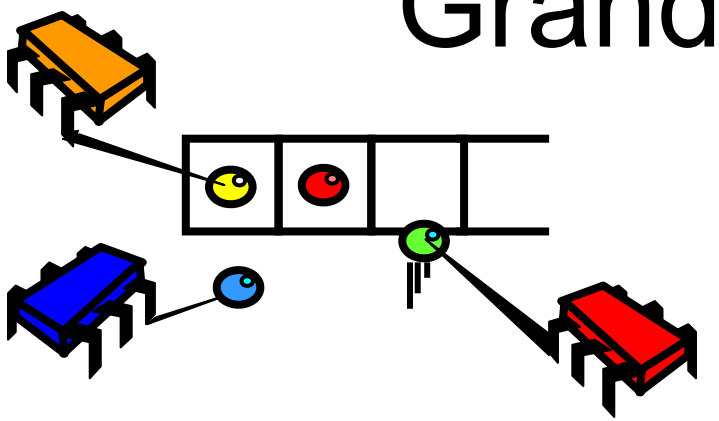
```
class LockFreeQueue[T: ClassTag](val capacity: Int) {

  @volatile
  private var head, tail: Int = 0
  private val items = new Array[T](capacity)

  def enq(x: T): Unit = {
    if (tail - head == items.length) throw FullException
    items(tail % items.length) = x
    tail = tail + 1
  }

  def deq(): T = {
    if (tail == head) throw EmptyException
    val x = items(head % items.length)
    head = head + 1
    x
  }
}
```



What About Multiple Dequeuers?

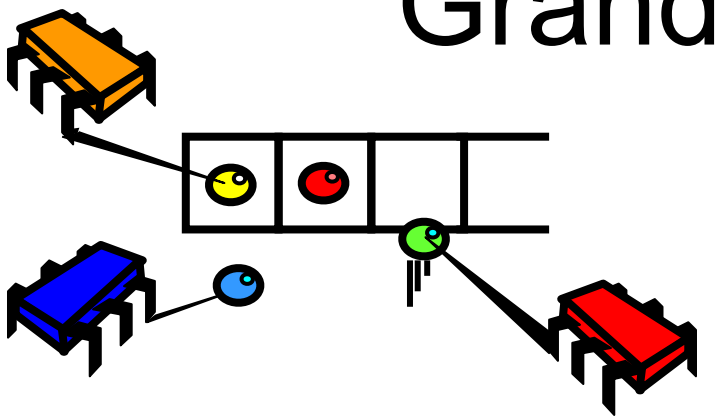


Grand Challenge



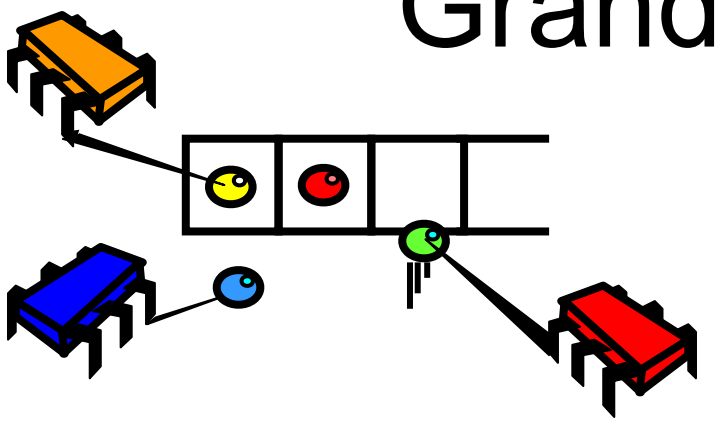
- Implement a FIFO queue

Grand Challenge



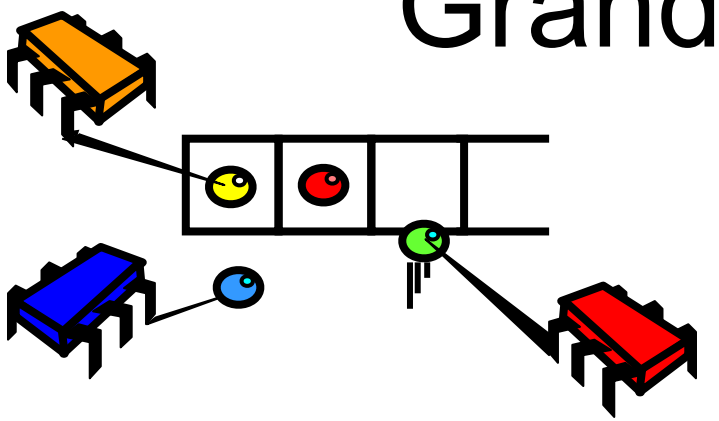
- Implement a FIFO queue
  - Wait-free

Grand Challenge



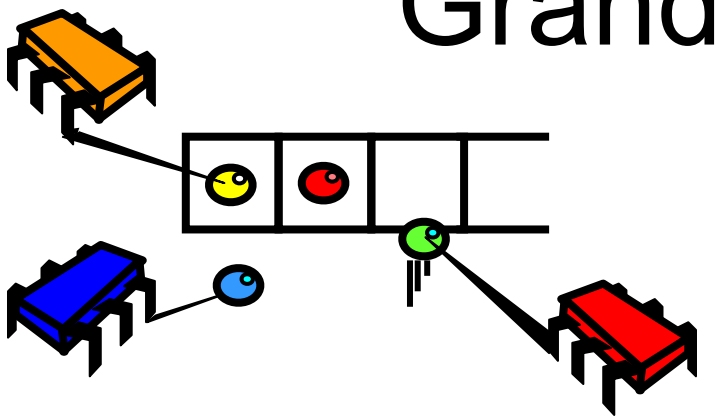
- Implement a FIFO queue
  - Wait-free
  - Linearizable

Grand Challenge



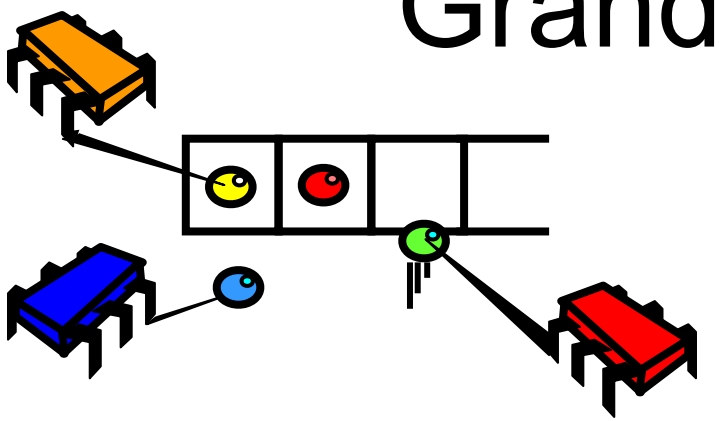
- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers

Grand Challenge



- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers

Grand Challenge



- Implement a FIFO queue
  - Wait-free
  - Linearizable
  - From atomic read-write registers
  - Multiple dequeuers

Only new aspect

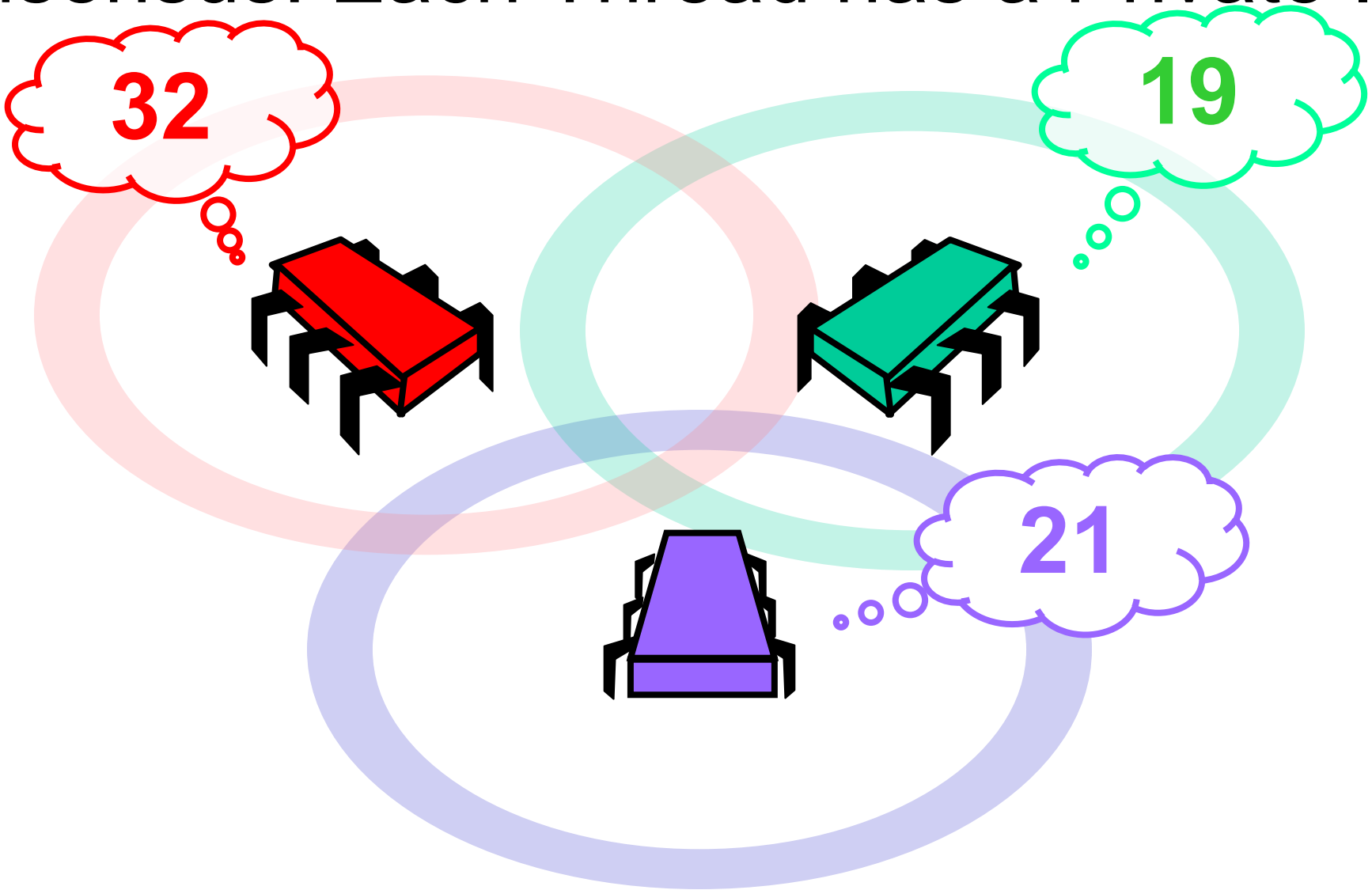
Puzzle

While you are ruminating on the grand challenge …

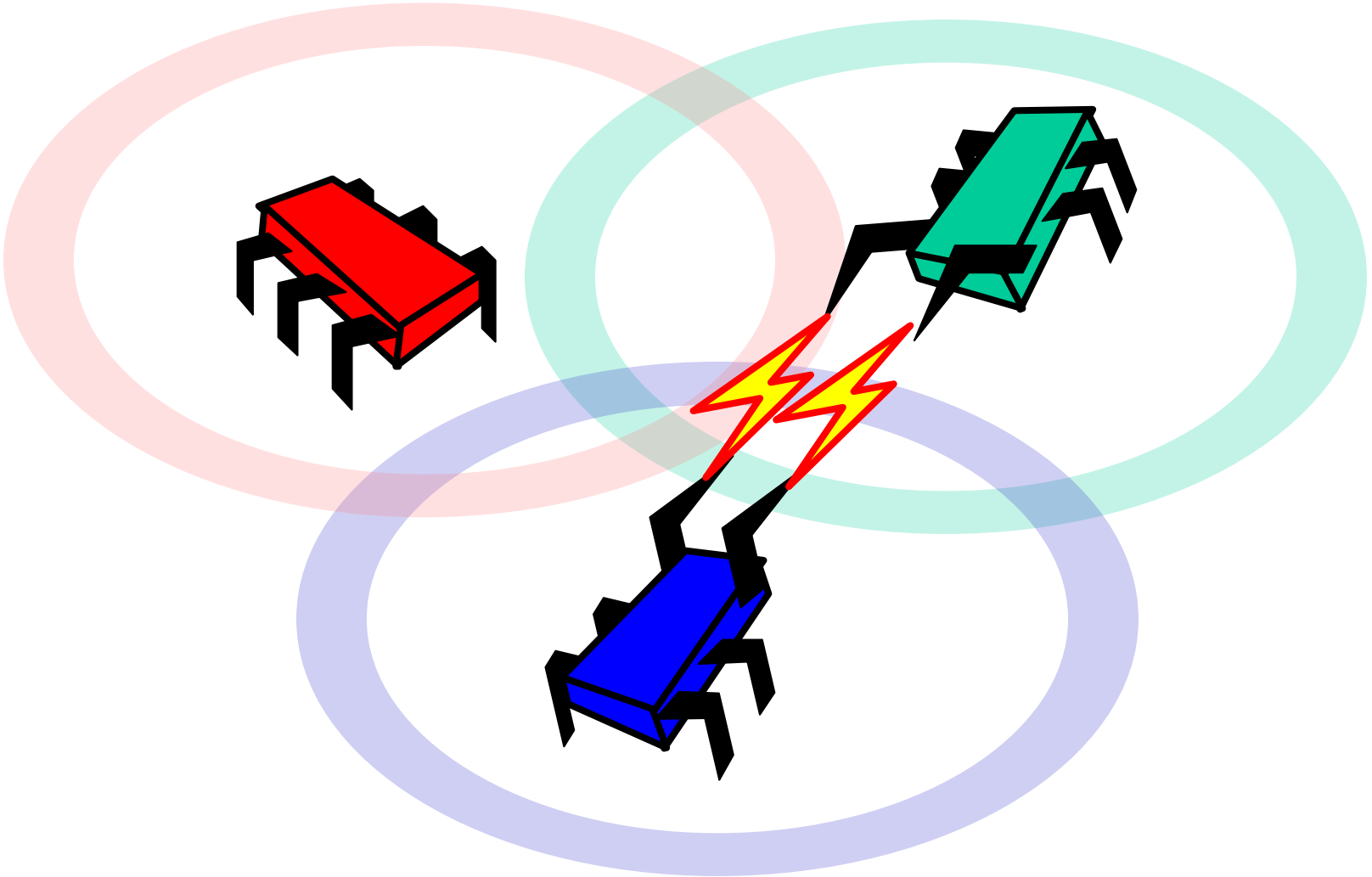
We will give you another puzzle …

Consensus!

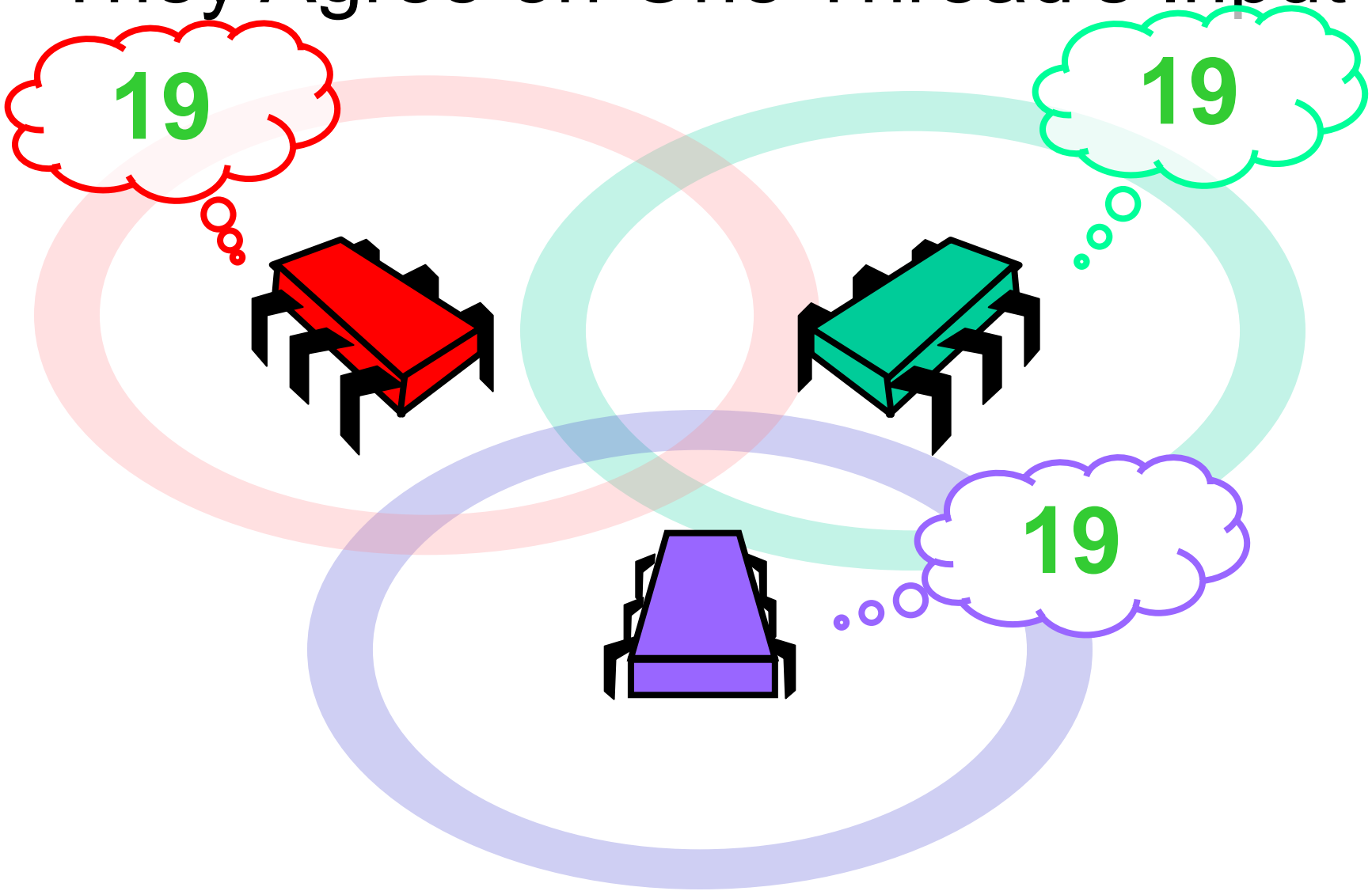
Consensus: Each Thread has a Private Input



They Communicate



They Agree on One Thread's Input



Formally: Consensus

- Consistent:
  - all threads decide the same value

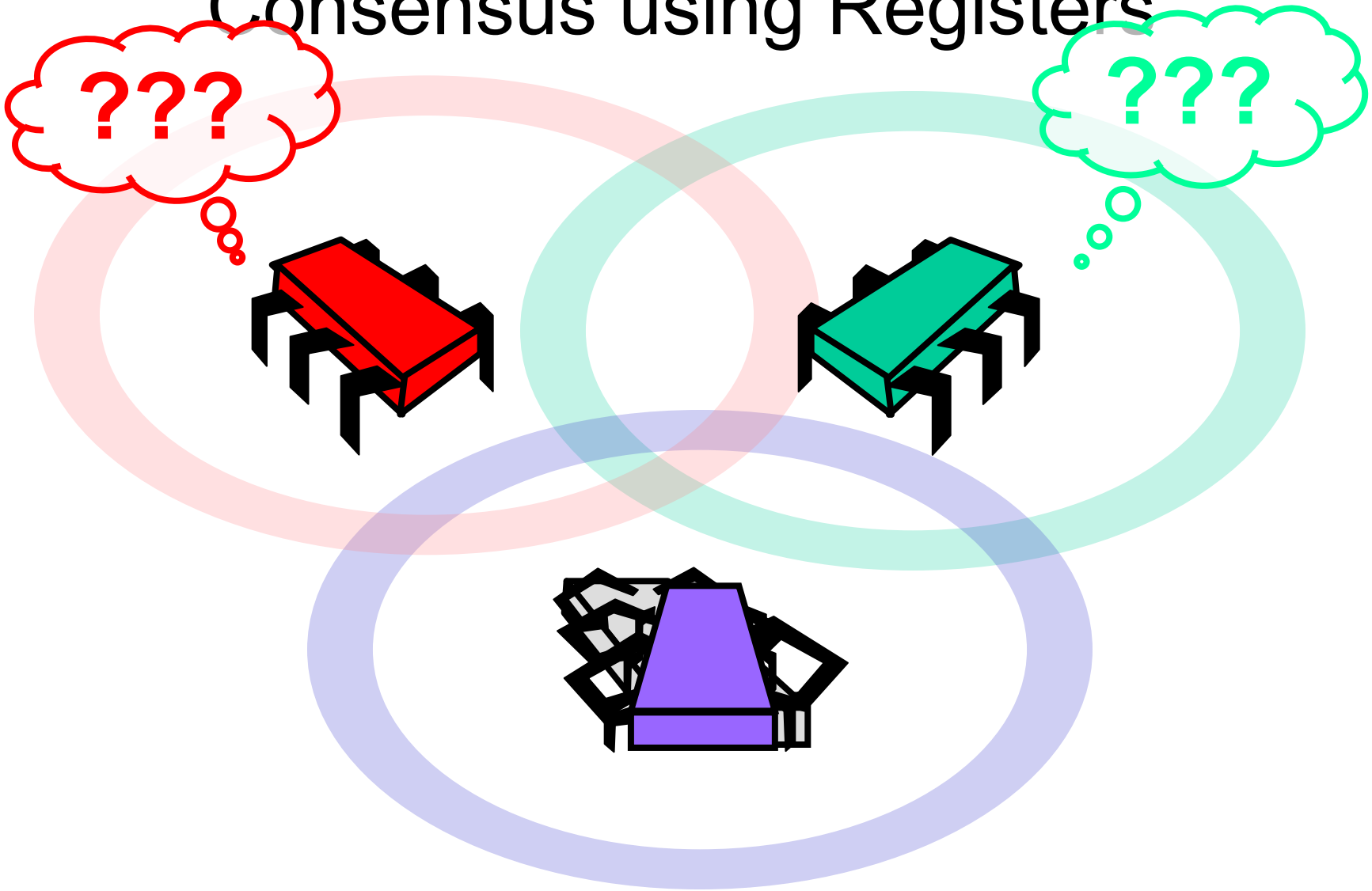
Formally: Consensus

- Consistent:
  - all threads decide the same value
- Valid:
  - the common decision value is some thread's input

In the past: *Consensus Game*

- Two of you need to agree on a value, e.g., A or B
- You need to devise *a protocol to reach a consensus*
- Tell me *the maximal number of steps* for each thread (<= 5, please)
- We are going to communicate using the white board
- Rules: *either* reading or writing **one** register (not both)
- *No other communication,*
- **No priorities** in “thread” identities or values:
  - Either of the values can be chosen (non-triviality)
  - One of the thread’s suggestions need to be chosen (validity)

No Wait-Free Implementation of Consensus using Registers



Formally

- Theorem
  - There is no wait-free implementation of *n*-thread consensus (n > 1) from read-write registers
- Proof
  - Using “valence trees” — see the textbook

Next:
Solving n-thread Consensus