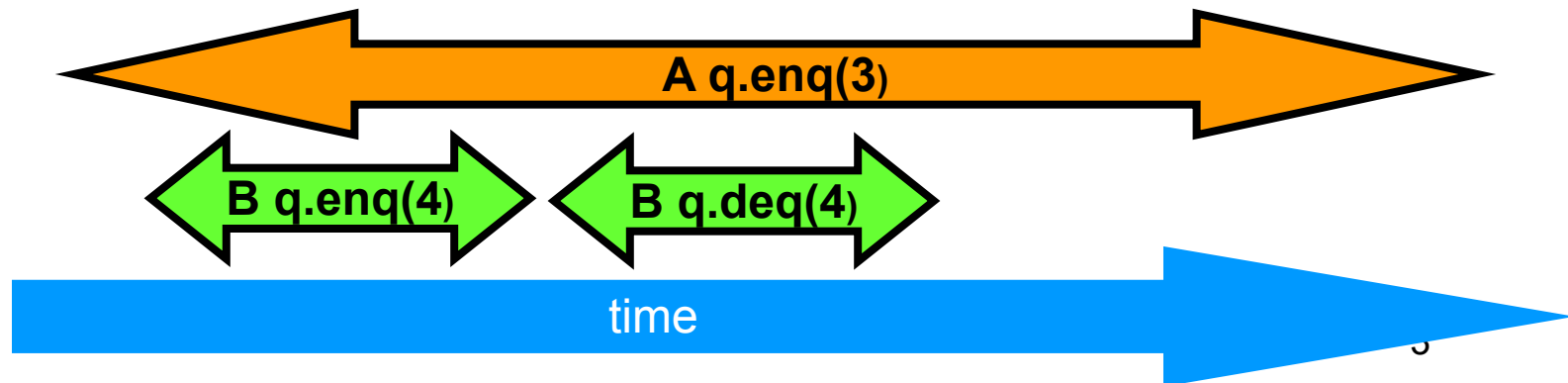# YSC3248: Parallel, Concurrent and Distributed Programming

Wait-Free Implementations

Last Week: Linearizability

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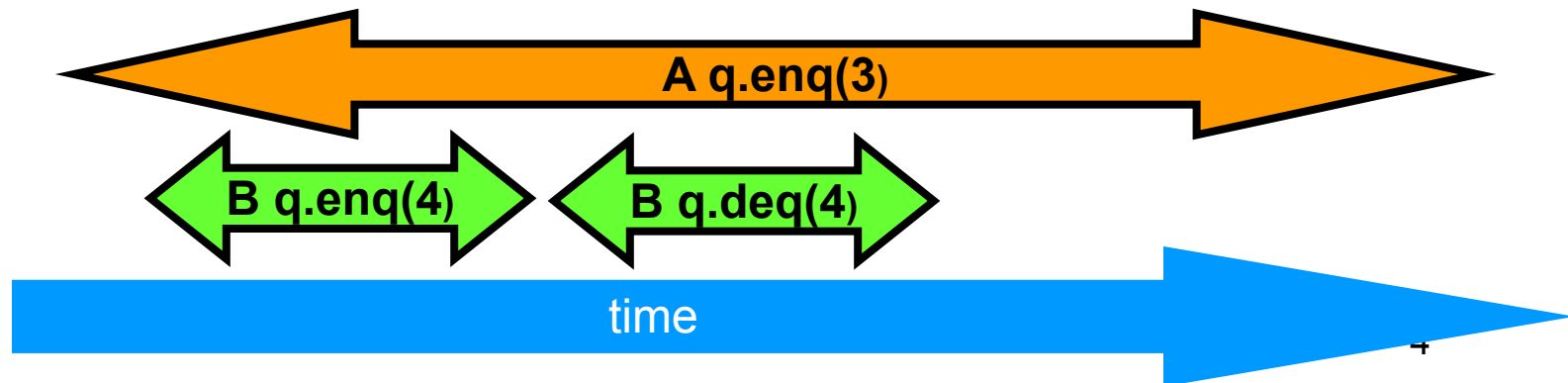B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

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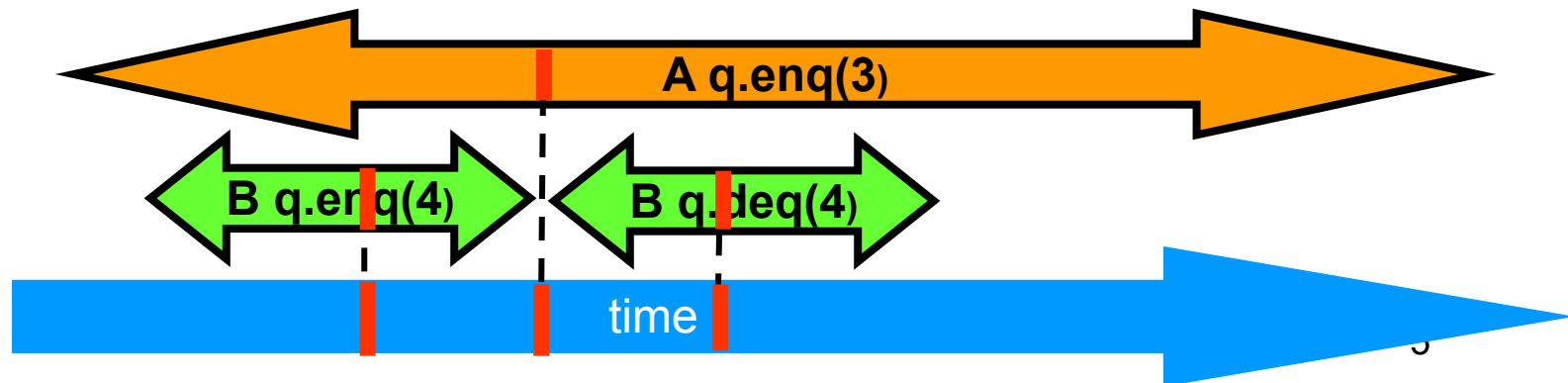A q.enq(3)
A q:void
B q.deq()
B q:4

A q.enq(3)

B q.enq(4)   B q.deq(4)

time

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

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 $\rightarrow_G \subset \rightarrow_S$

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**
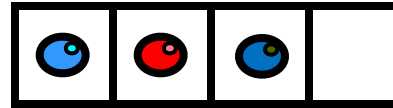 - ~~Where →**G** ⊆ →**S**~~

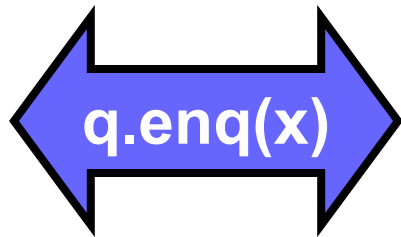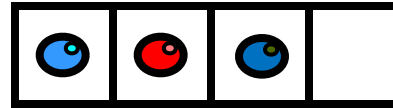**Differs from linearizability**

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
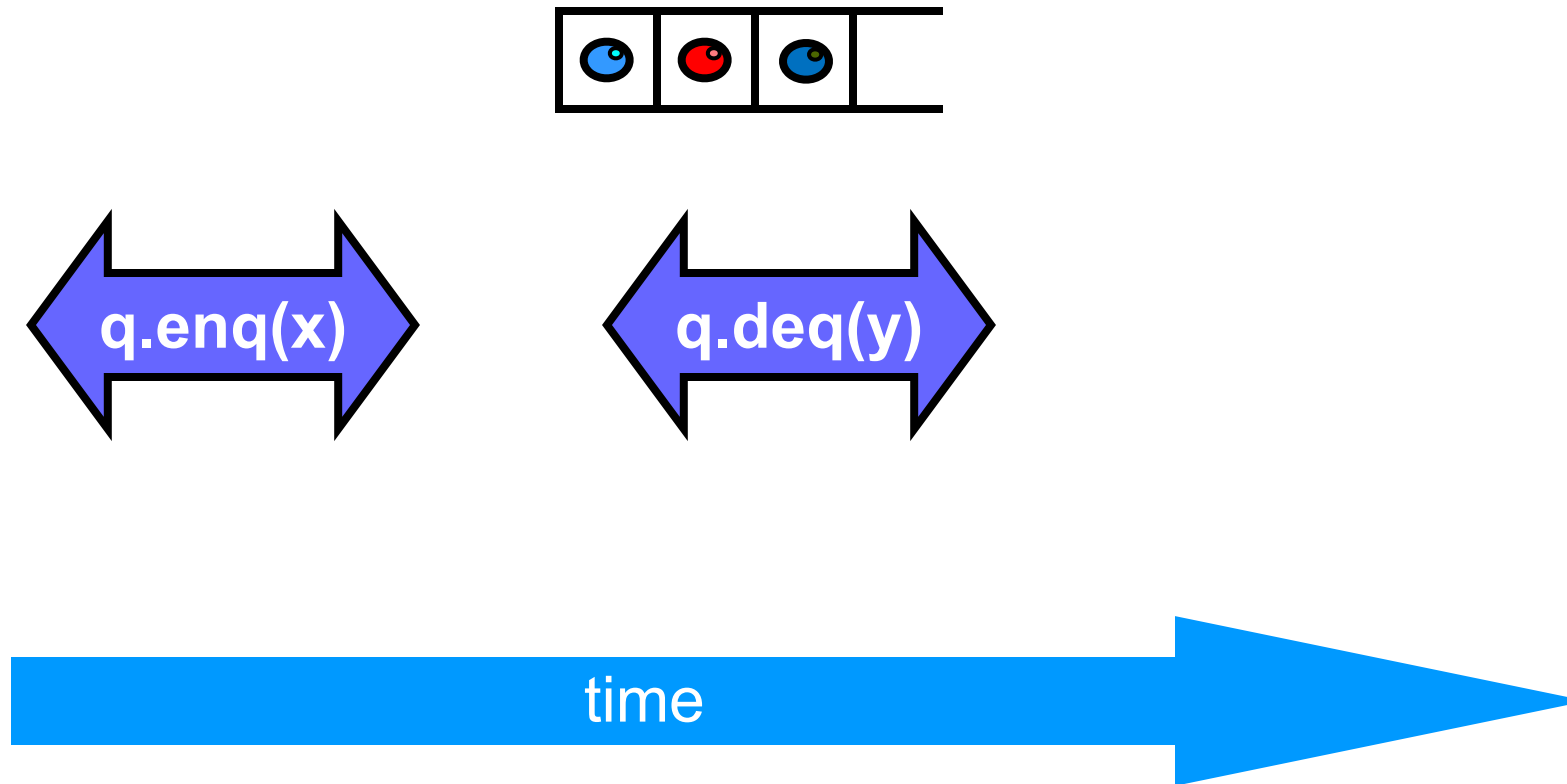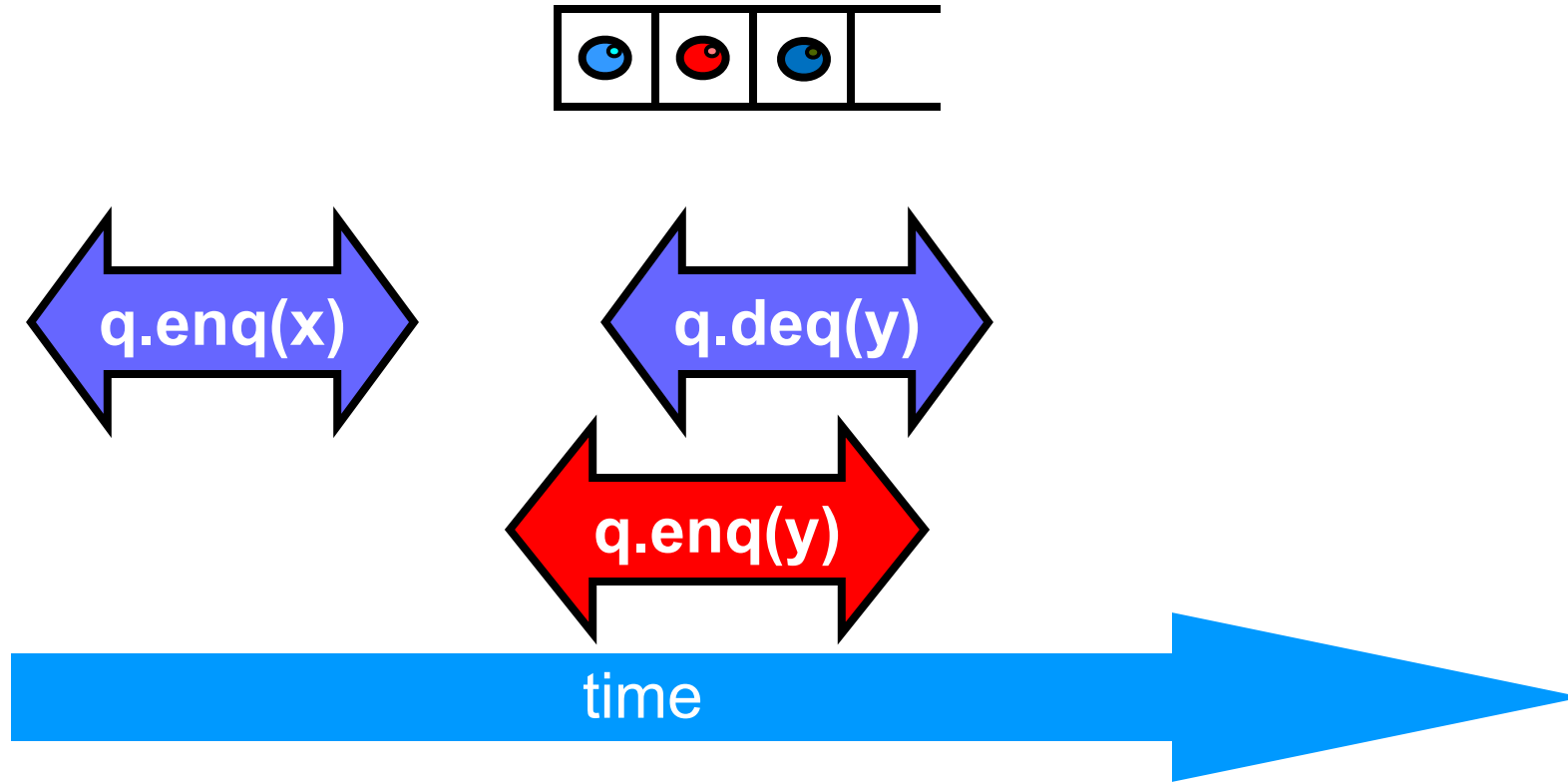- Often used to describe multiprocessor memory architectures
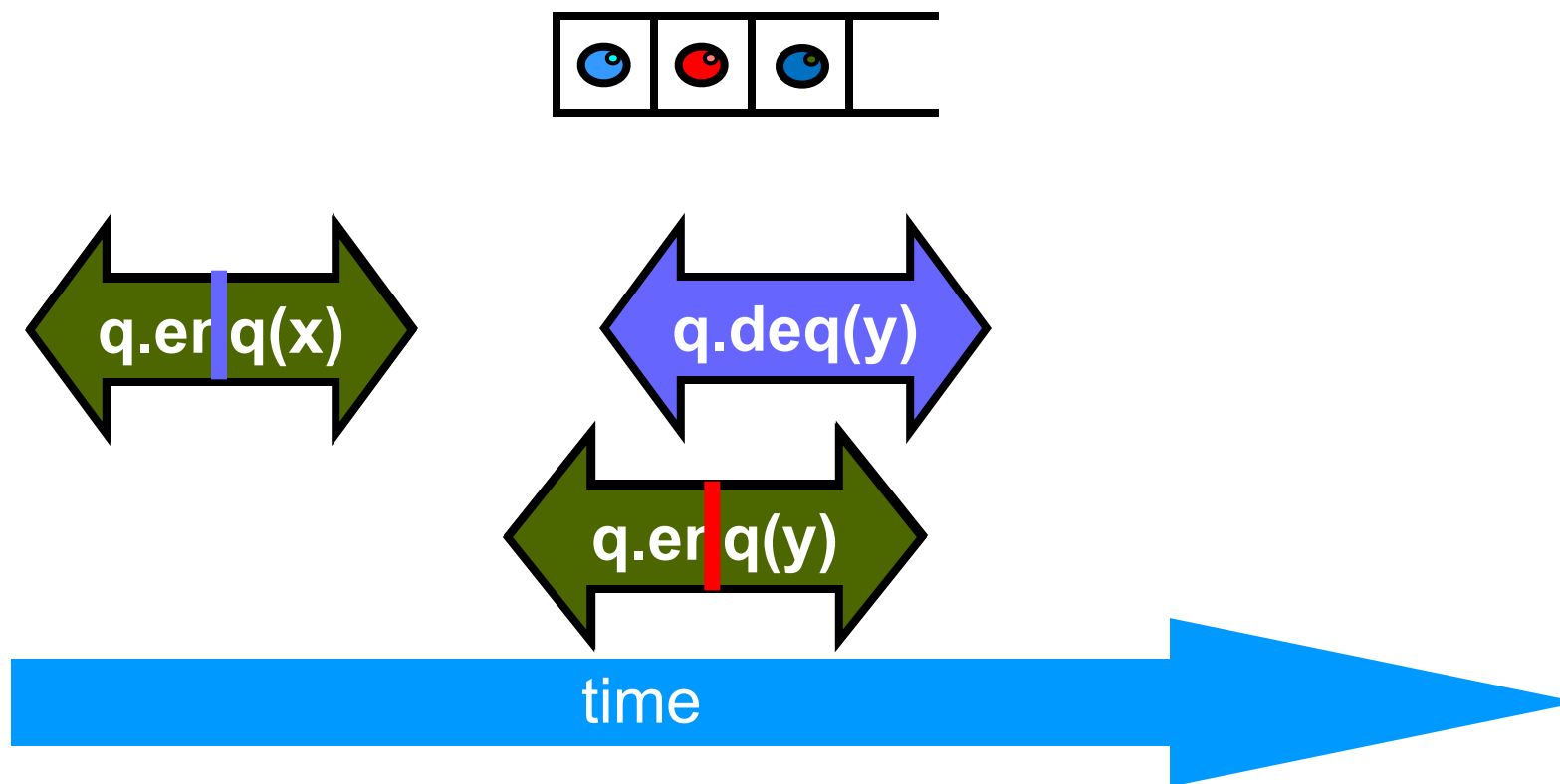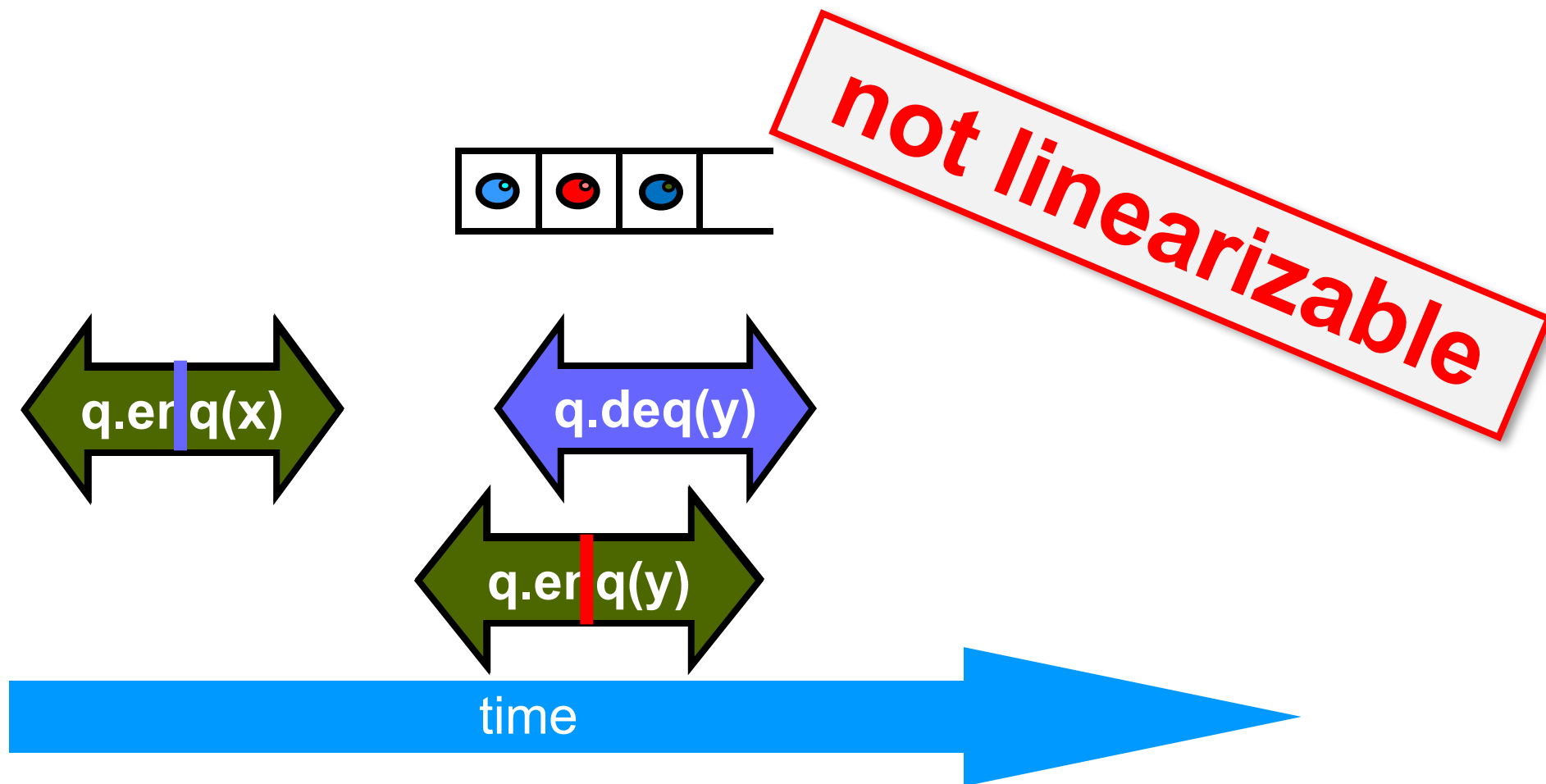
Example

time

Example

q.enq(x)

time

Example

q.enq(x)

q.deq(y)

time

11

Example

Example

q.enq(x)

q.deq(y)

q.enq(y)

time
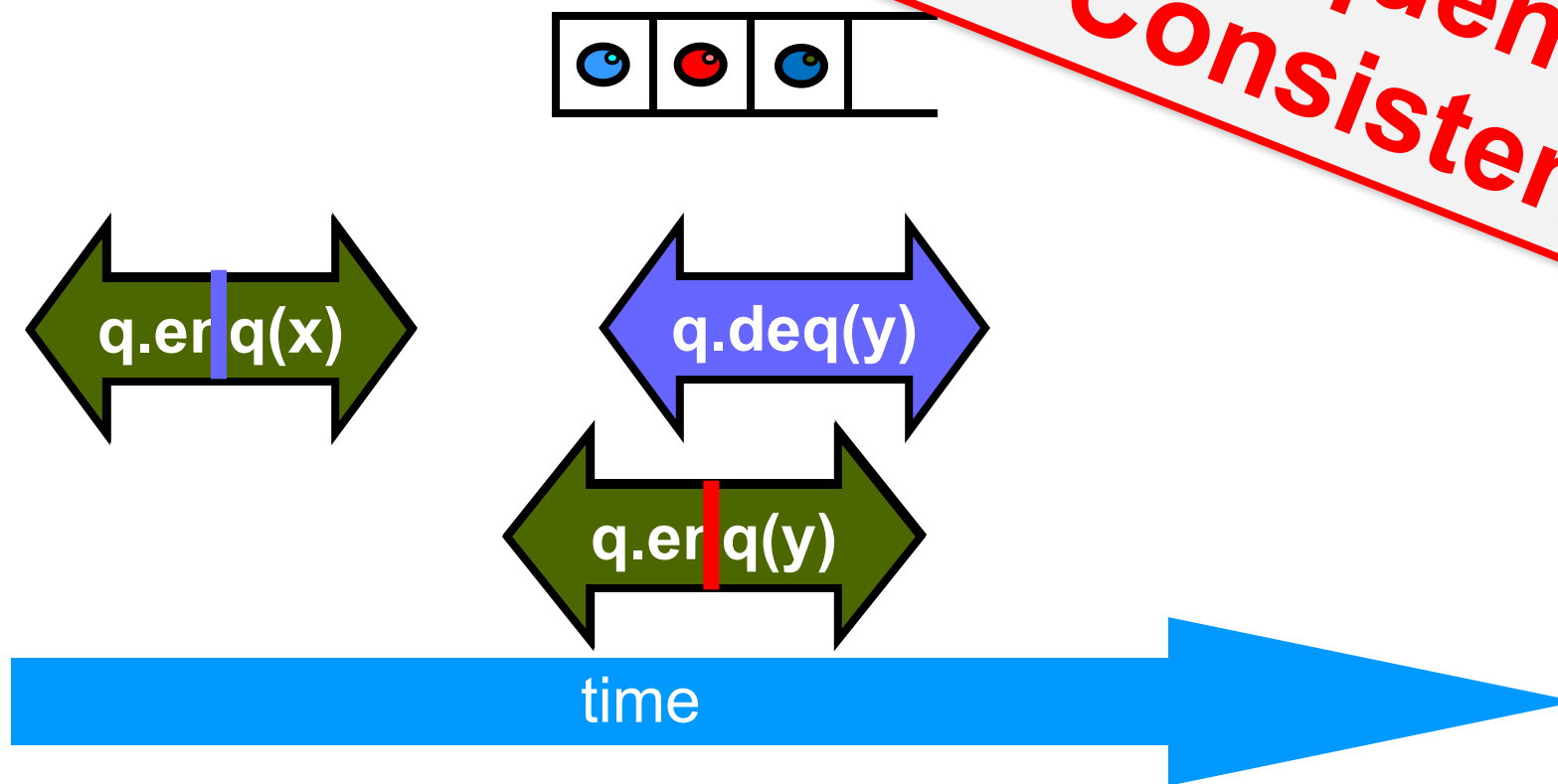
Example

Theorem

Sequential Consistency is not composable

FIFO Queue Example

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

time

FIFO Queue Example

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

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

time
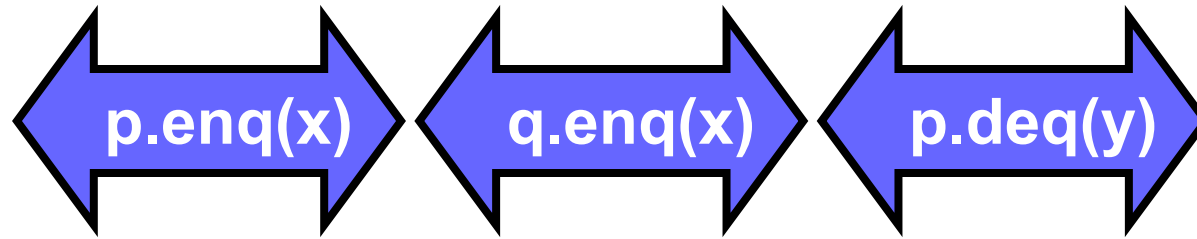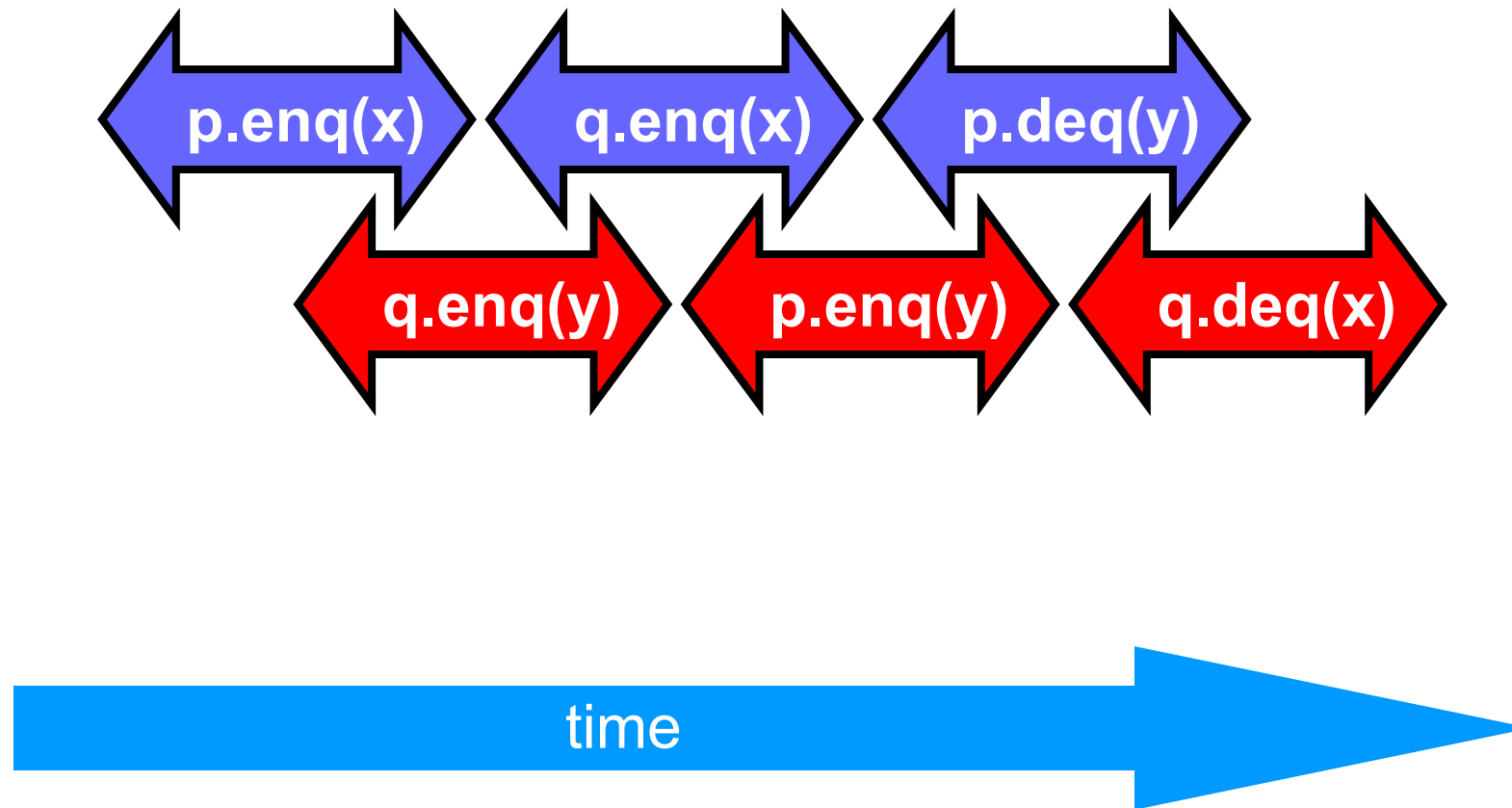
FIFO Queue Example

History H

time
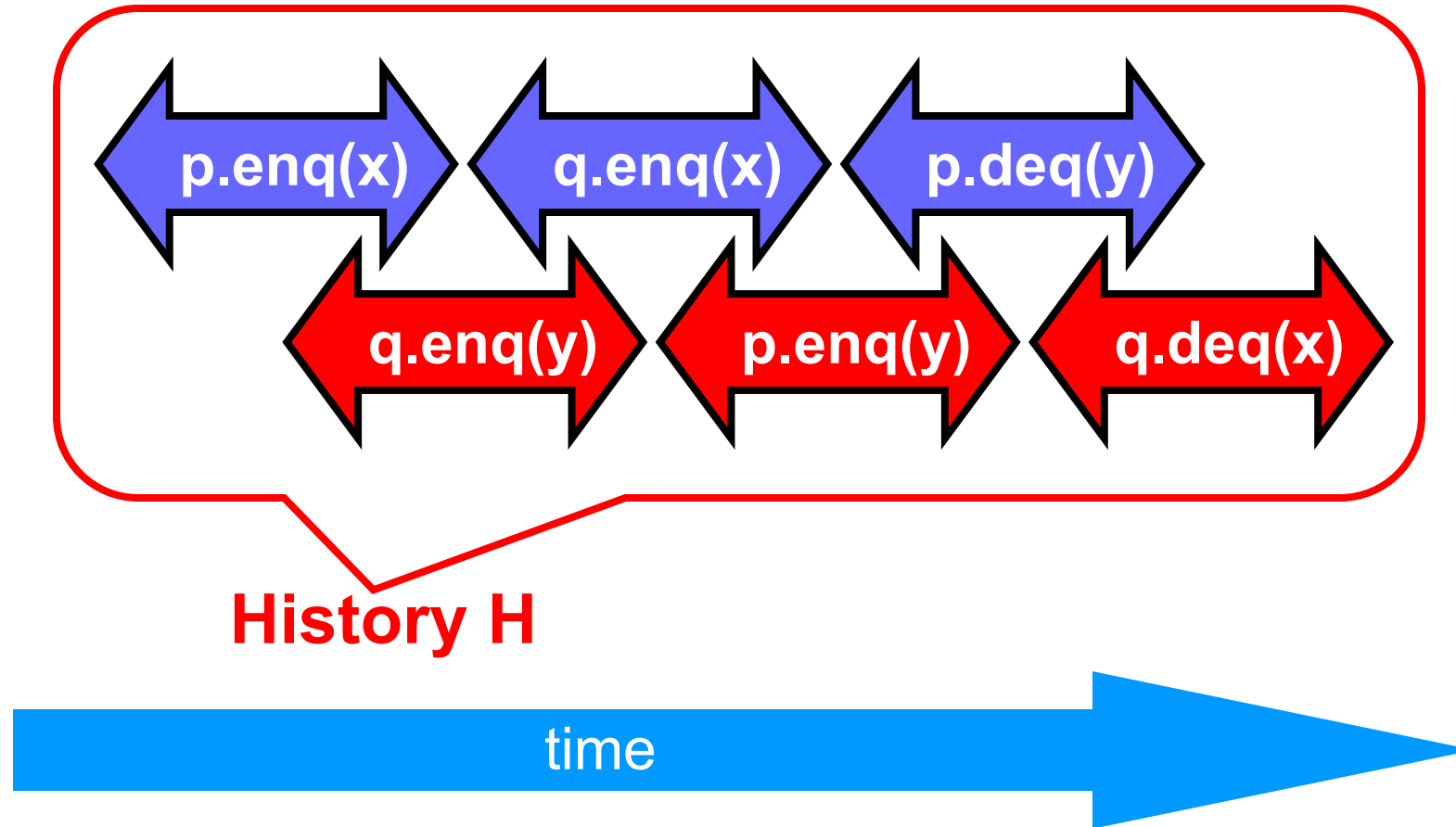
H|p Sequentially Consistent

H|q Sequentially Consistent

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

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

time
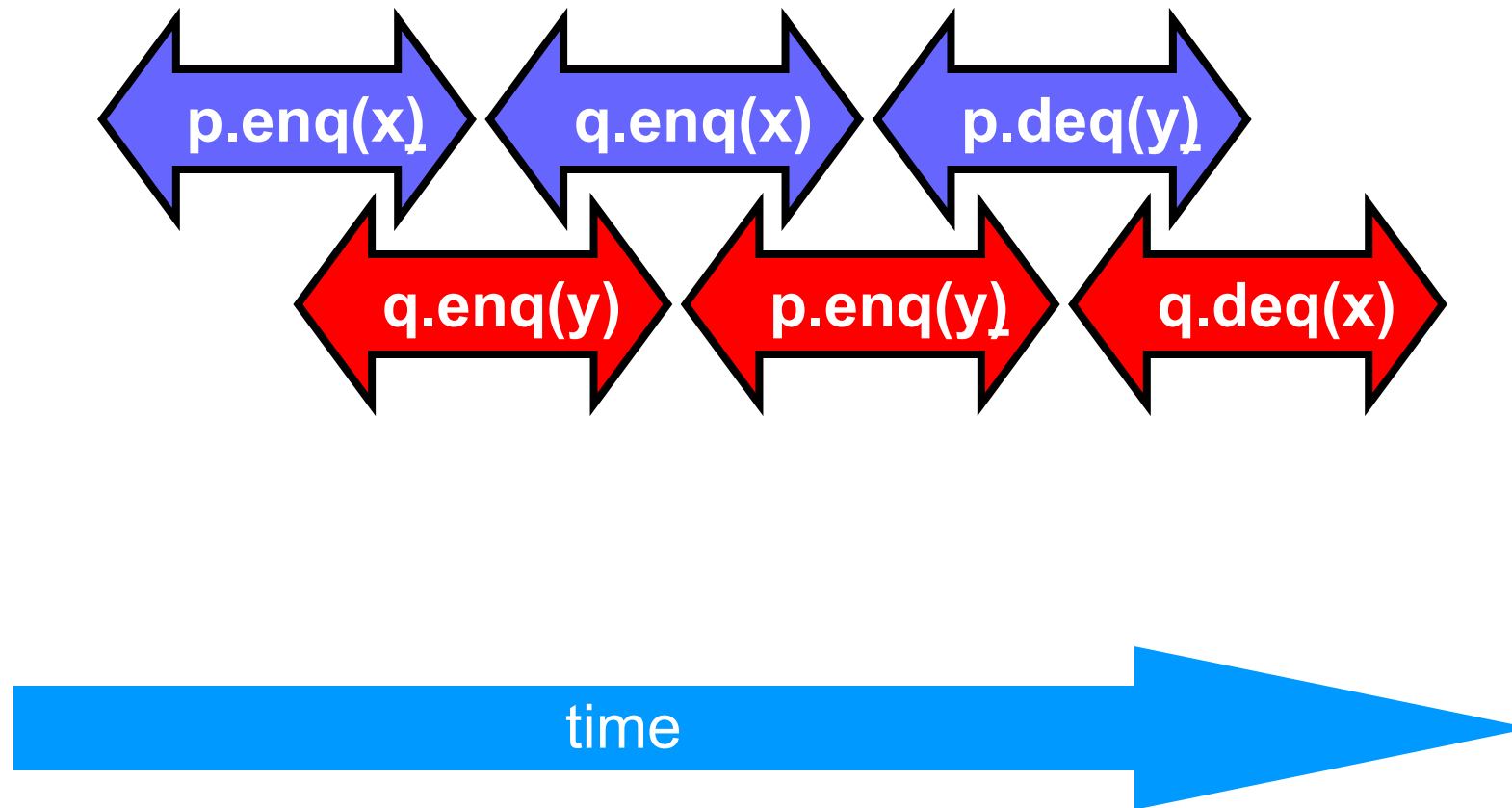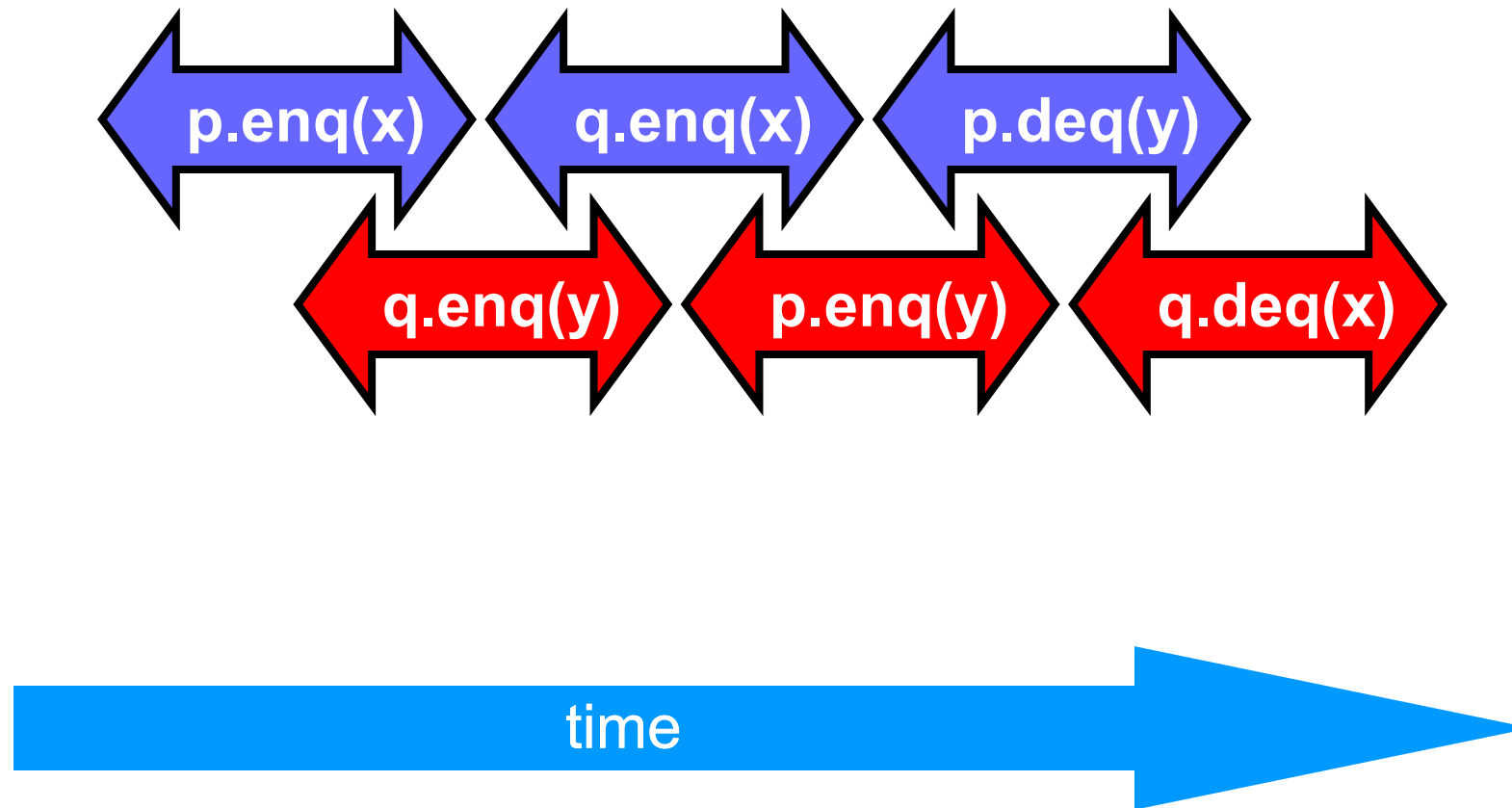
Ordering imposed by p

Ordering imposed by q

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

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

time
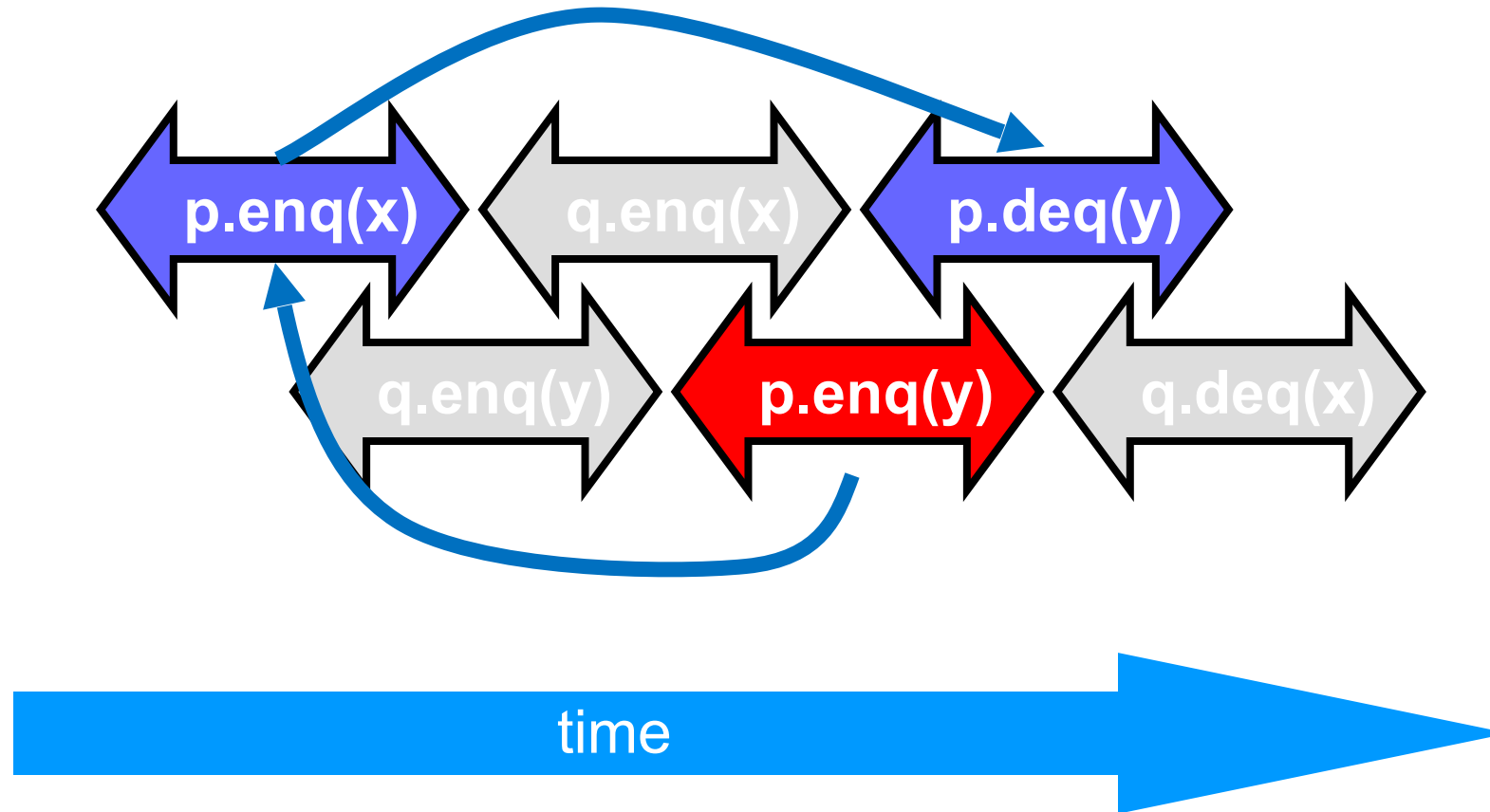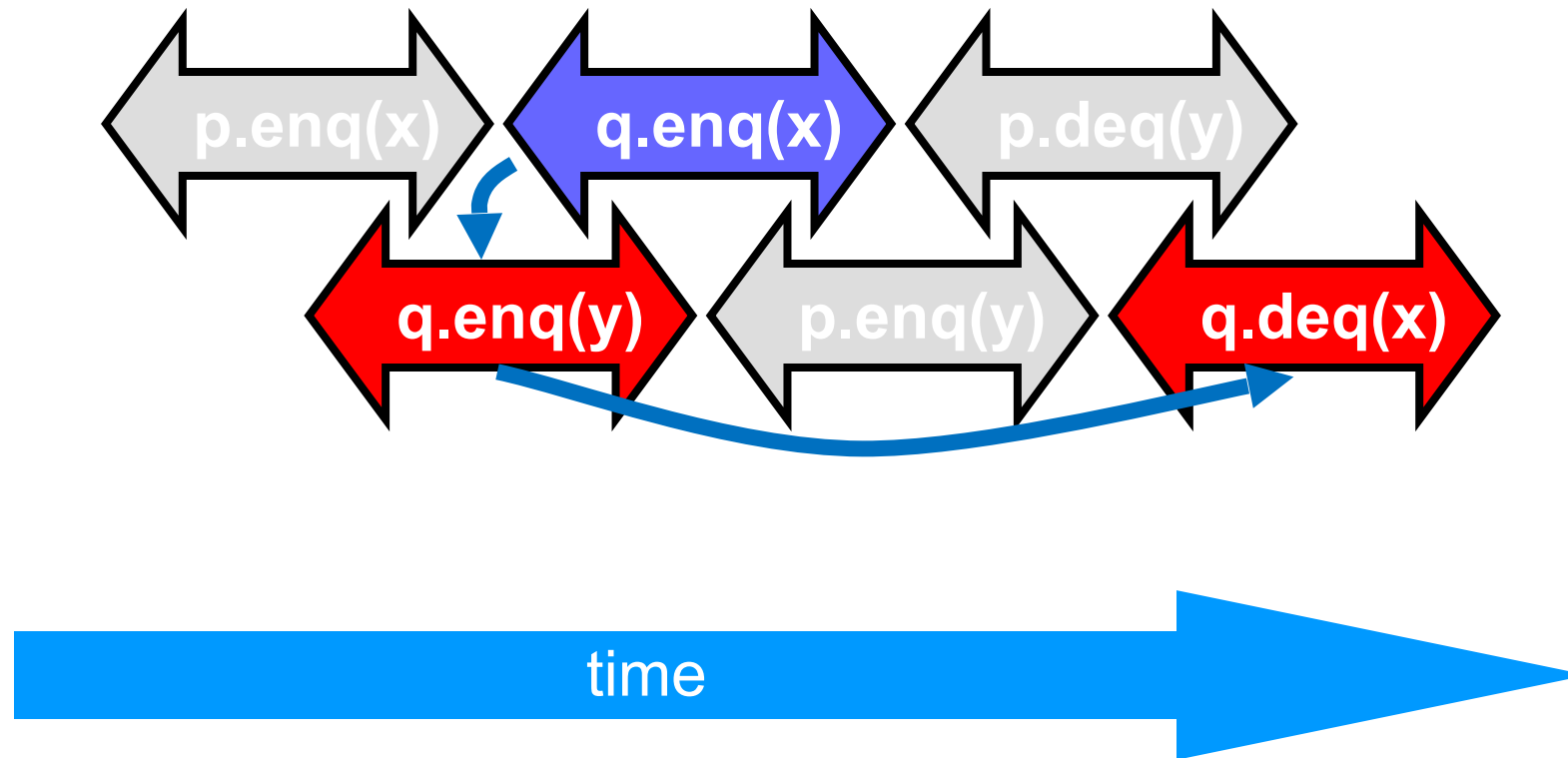
Ordering imposed by both

Combining orders

time
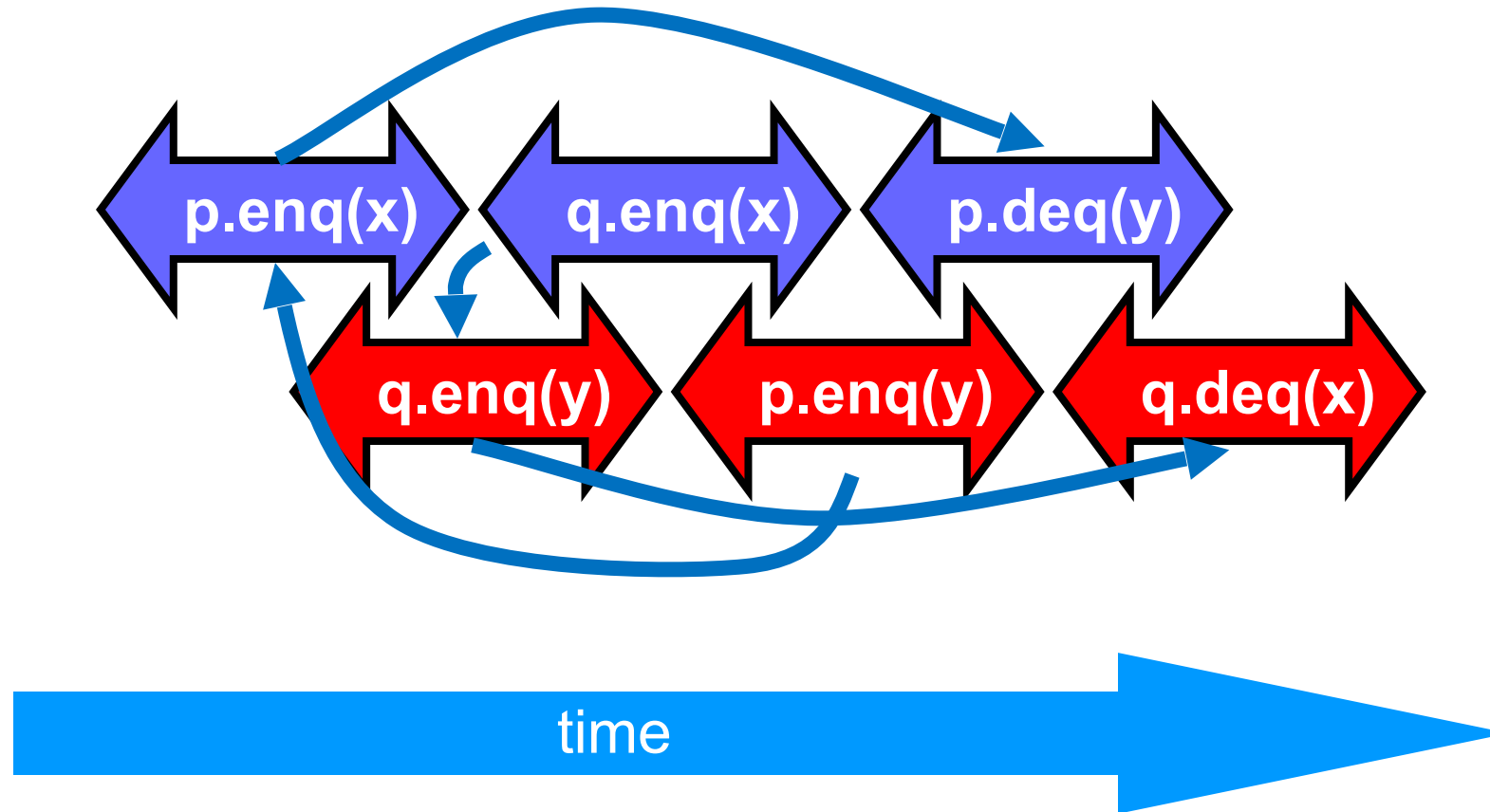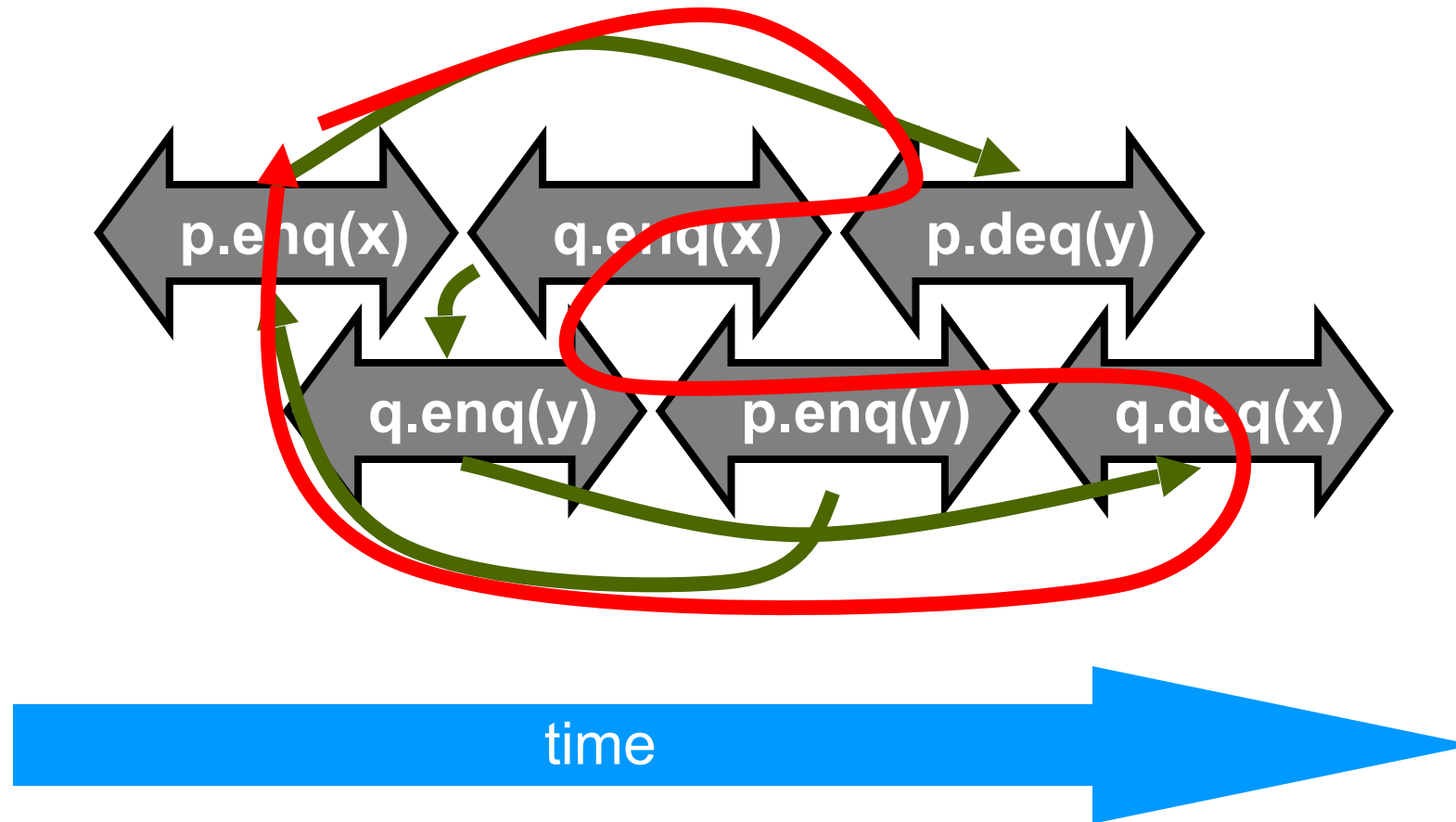
Fact

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

The Flag Example

x.write(1)

y.read(0)

y.write(1)

x.read(0)

time
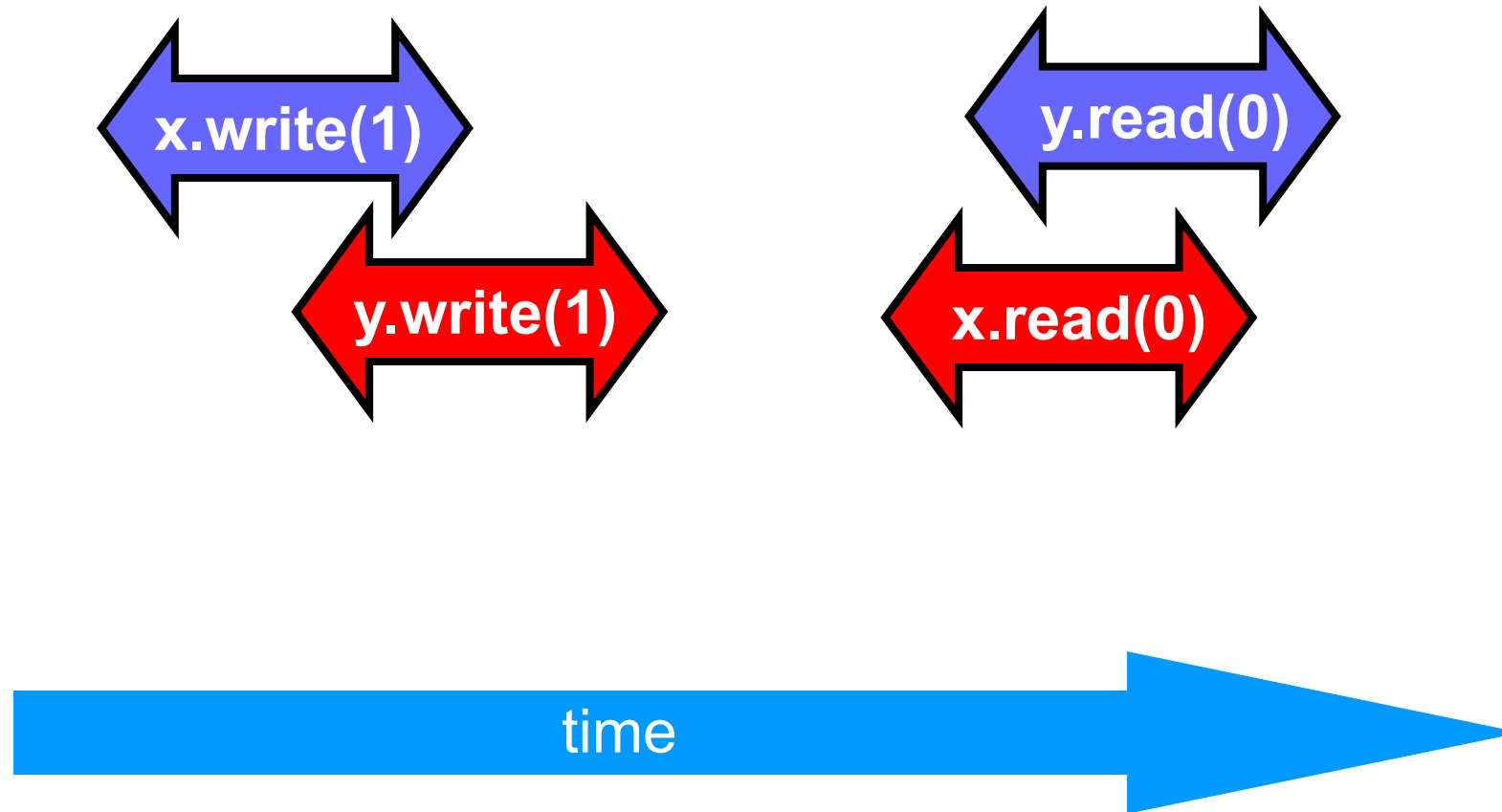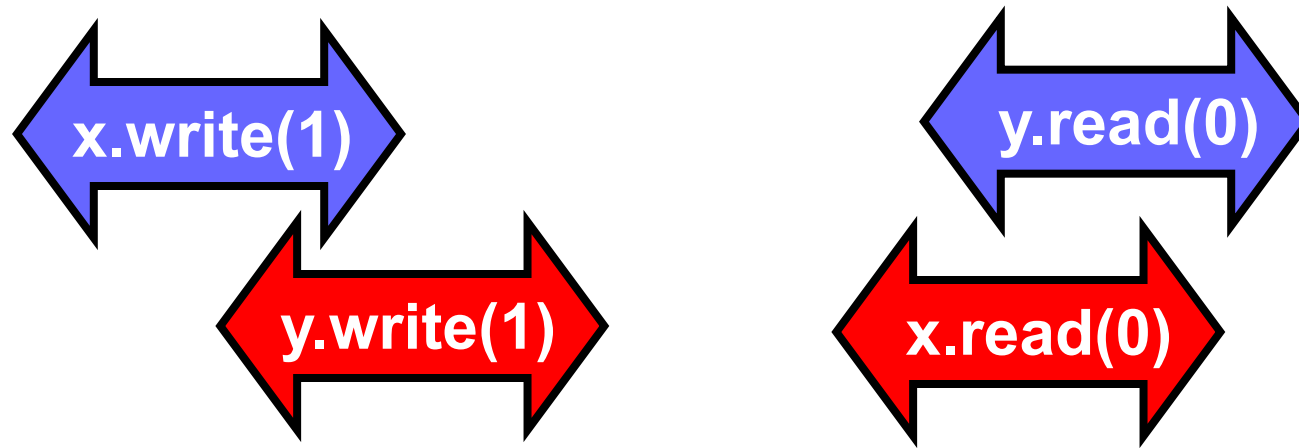
The Flag Example

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

The Flag Example

x.write(1)

y.read(0)

y.write(1)

x.read(0)
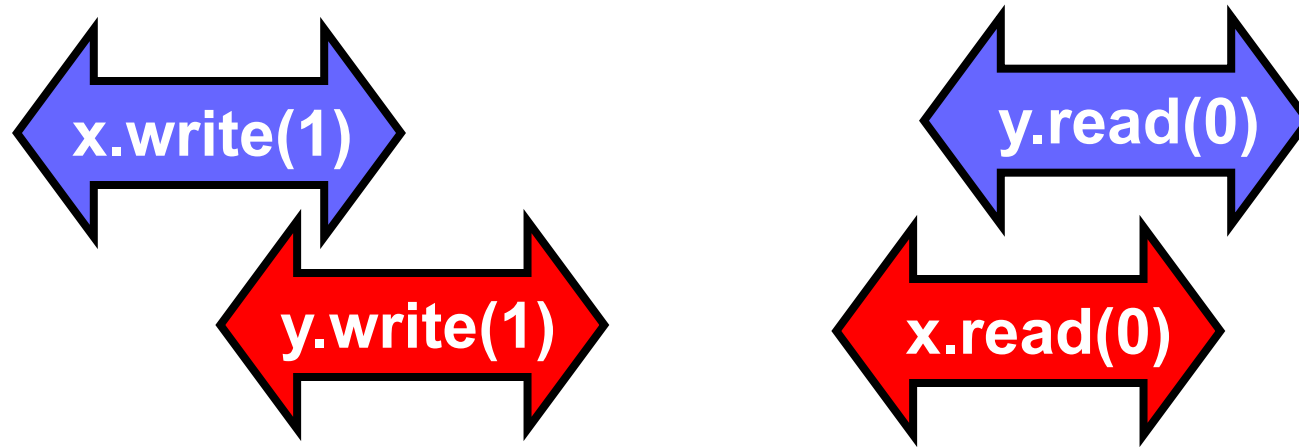
- Entire history isn't sequentially consistent
  - Can't *both* go first
  - Petersen't lock now got a problem!

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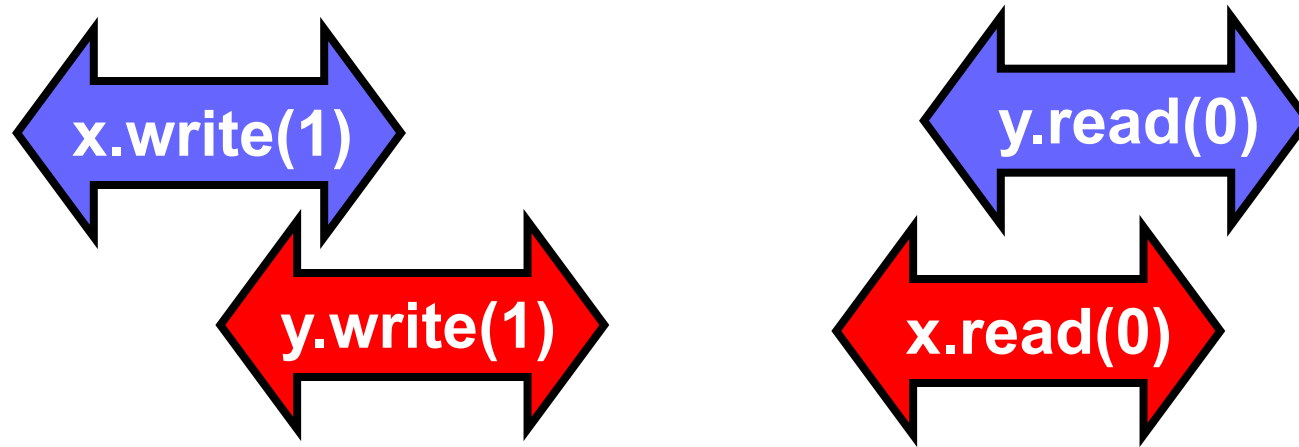
- Is this behavior really so wrong?
  - We can argue either way …

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

```scala
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) $write_B(flag[B]=true)\rightarrow$

(3) $write_B(victim=B)\rightarrow$

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

$\rightarrow\quad read_A(victim)$

Crux of Peterson Proof

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

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

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

$\rightarrow \quad \text{read}_A(\text{victim})$

<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 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

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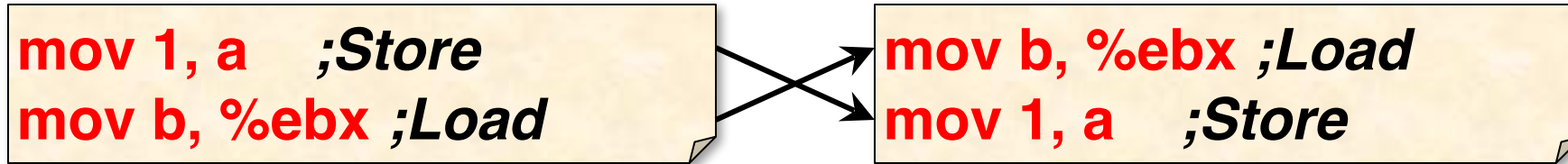
```
mov b, %ebx ;Load
mov 1, a     ;Store
```
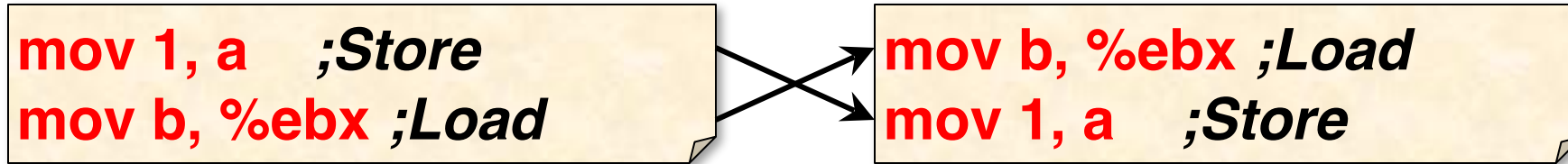
Program Order

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*.

Weak-Memory Instruction Reordering

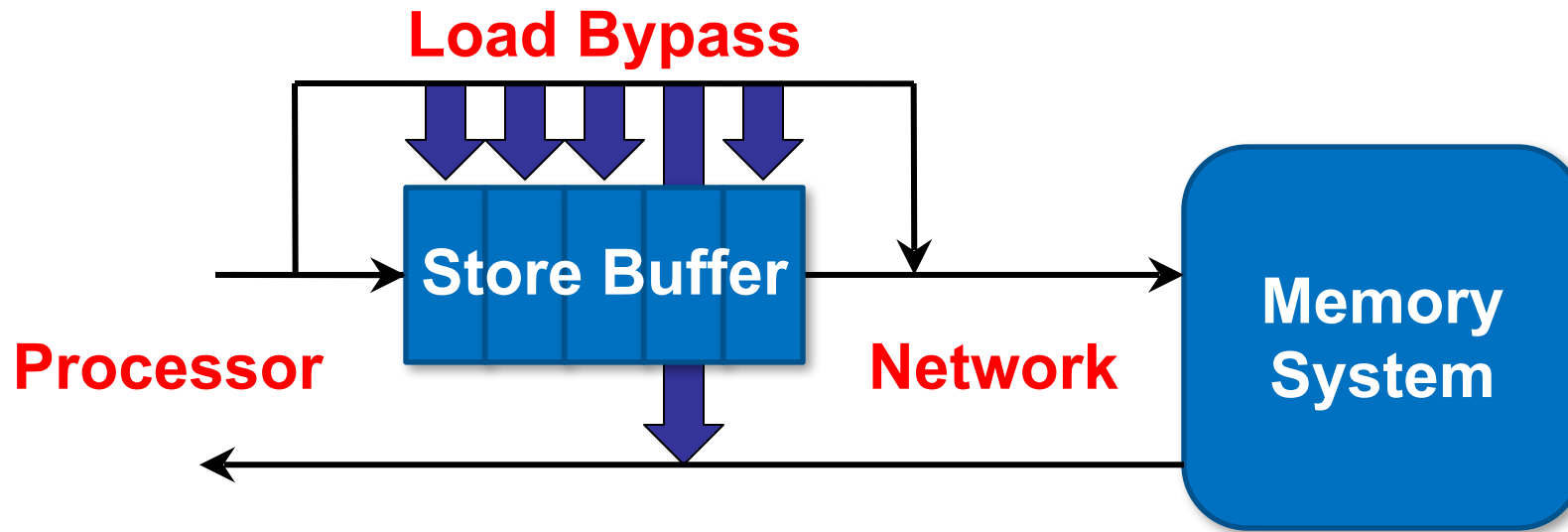| mov 1, a *;Store* mov b, %ebx *;Load*	mov b, %ebx *;Load* mov 1, a *;Store*

Program Order — Execution Order

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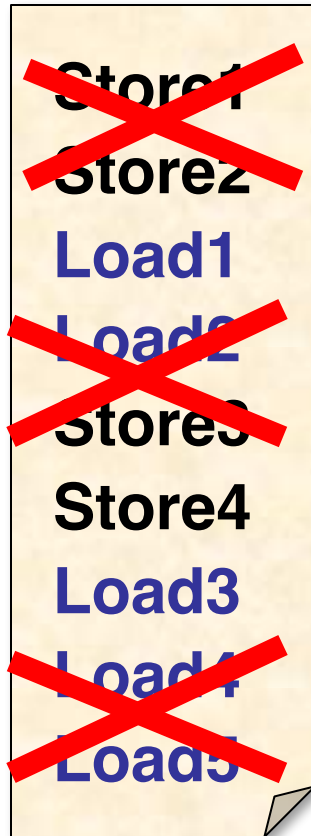
A. When a ≠ b.

A′. And there's no concurrency.

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.

X86 Relaxed Memory Model

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 Relaxed Memory Model

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5

LOADS ↑

OK!

1. Loads **are *not* reordered with** loads**.**
2. Stores **are *not* reordered with** stores**.**
3. ~~or~~
   **Total Store Ordering (TSO)
   …weaker than sequential
   consistency**
4. ~~rior~~
   ~~ot~~
   ~~with a prior store to the same location.~~
5. Stores to the same location **respect a global total order.**

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 ~~are not reordered with loads~~
2. Stor~~e~~
3. Store~~s are not reordered with~~ loads
4. **A** loa~~d~~ store~~ ~~ with a pri~~or store~~ to the same location.
5. ~~S~~tores to the same location **respect a global total order.**

> Total Store Ordering +
> properly placed memory barriers
> = sequential consistency

44

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

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

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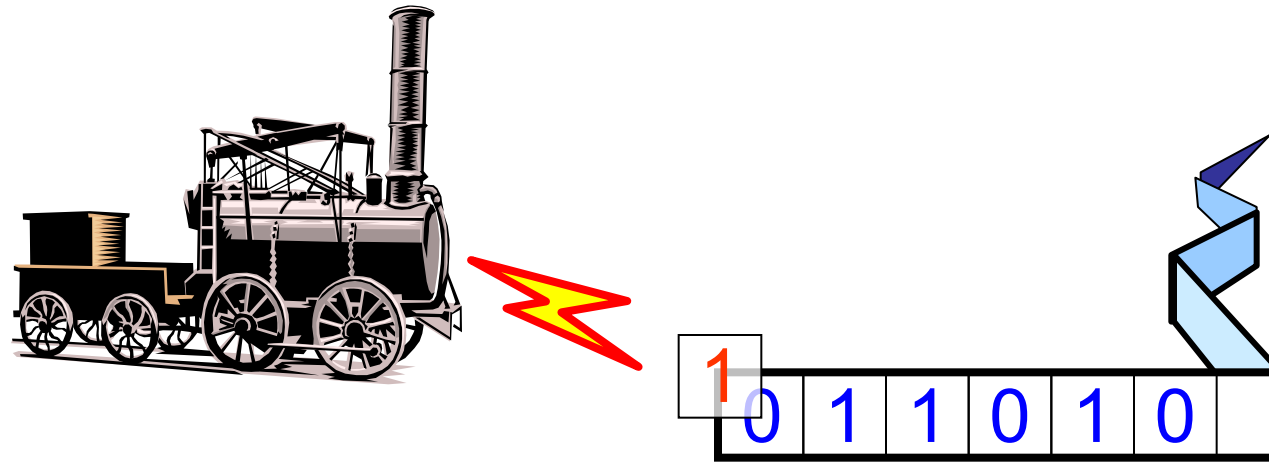
- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Shared-Memory Computability?

Shared Memory

10011
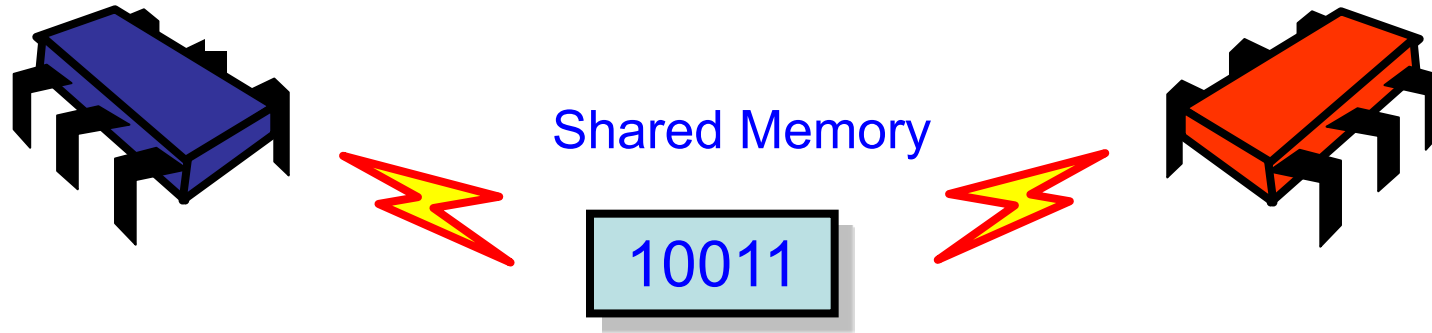
- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions ....
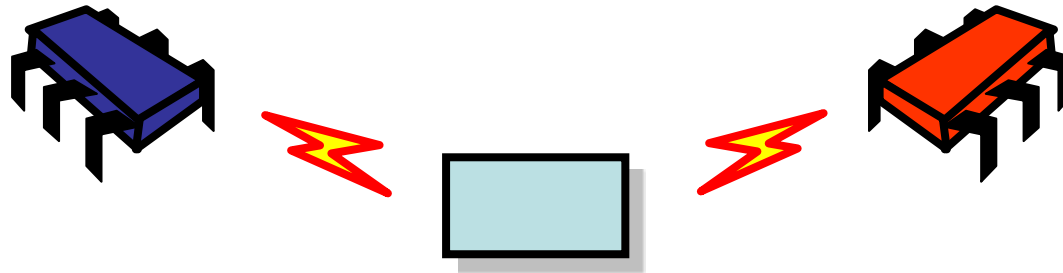
Foundations of Shared Memory

To understand modern

What is the **weakest** useful form of shared memory?
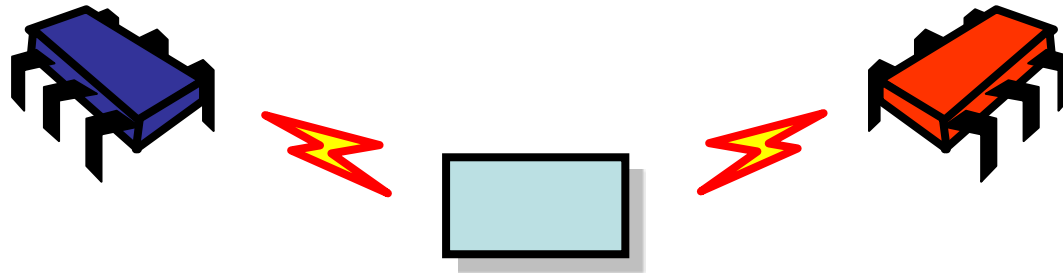
Foundations of Shared Memory

To understand modern

What is the weakest useful form of

What can it do?

58
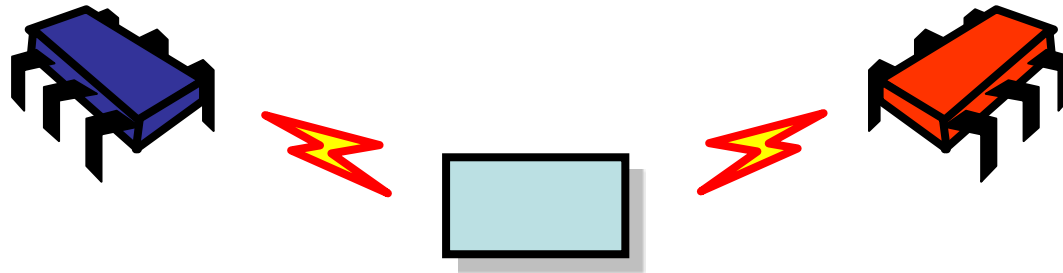
Register*

Holds a
(binary) value

10011
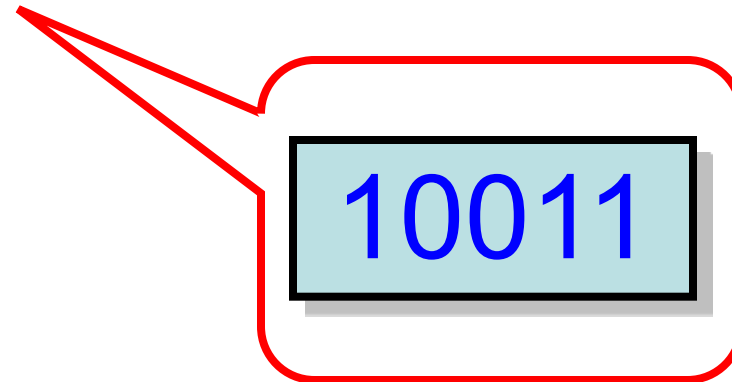
* A memory location: name is historical

Register

Register

Can be written

01100

10011

From Weakest Register

Single writer

1

Single reader

0   1
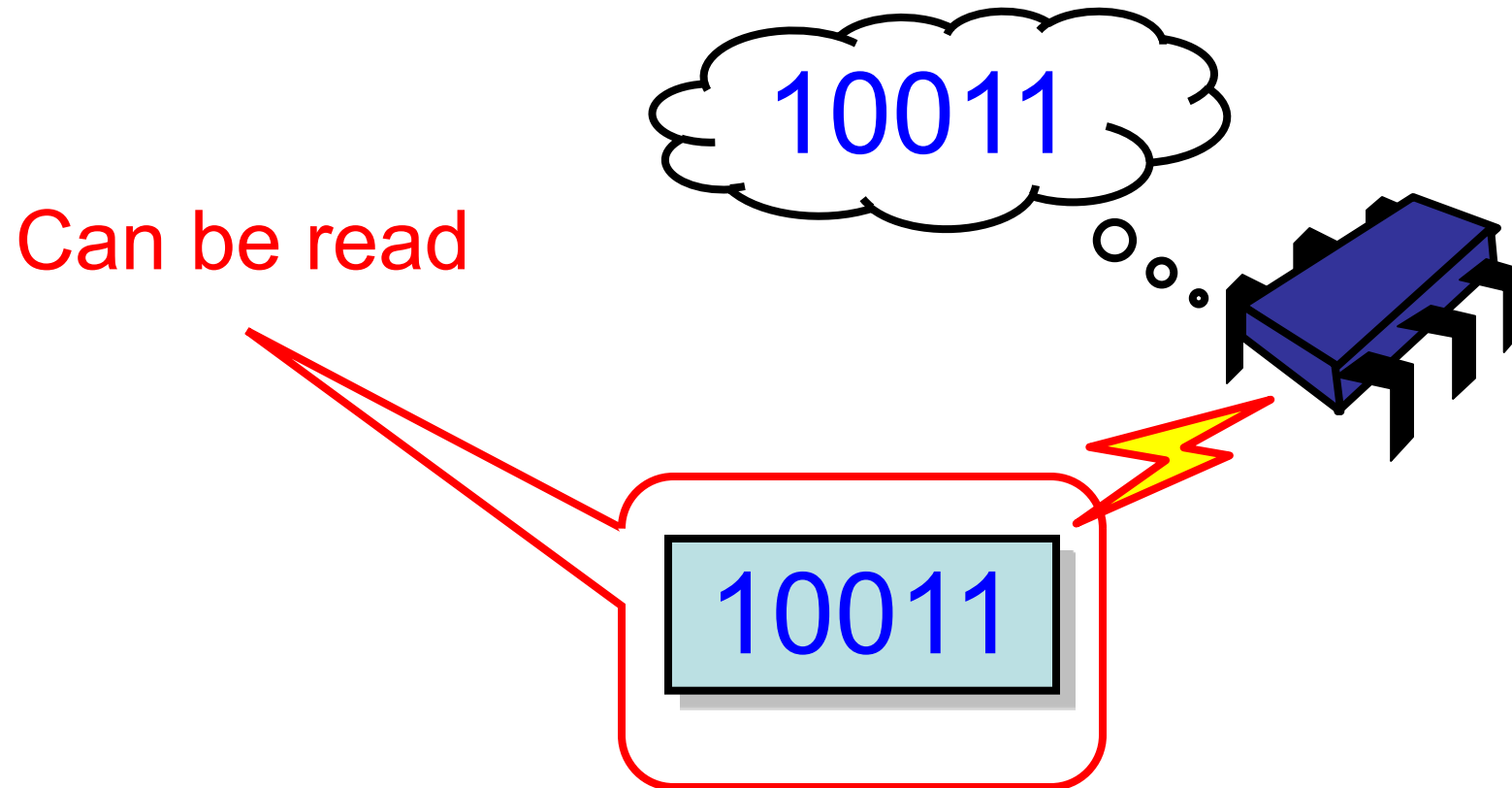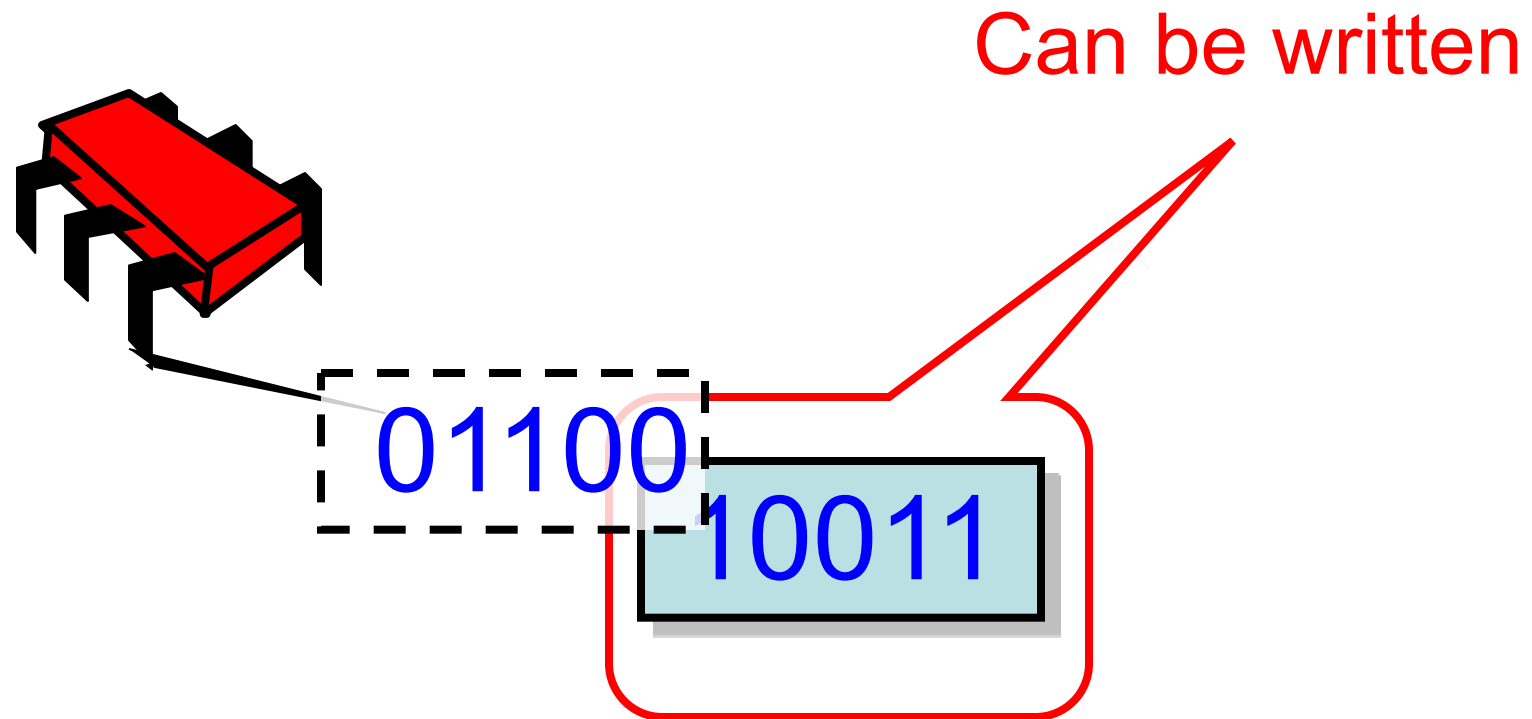
Safe Boolean register
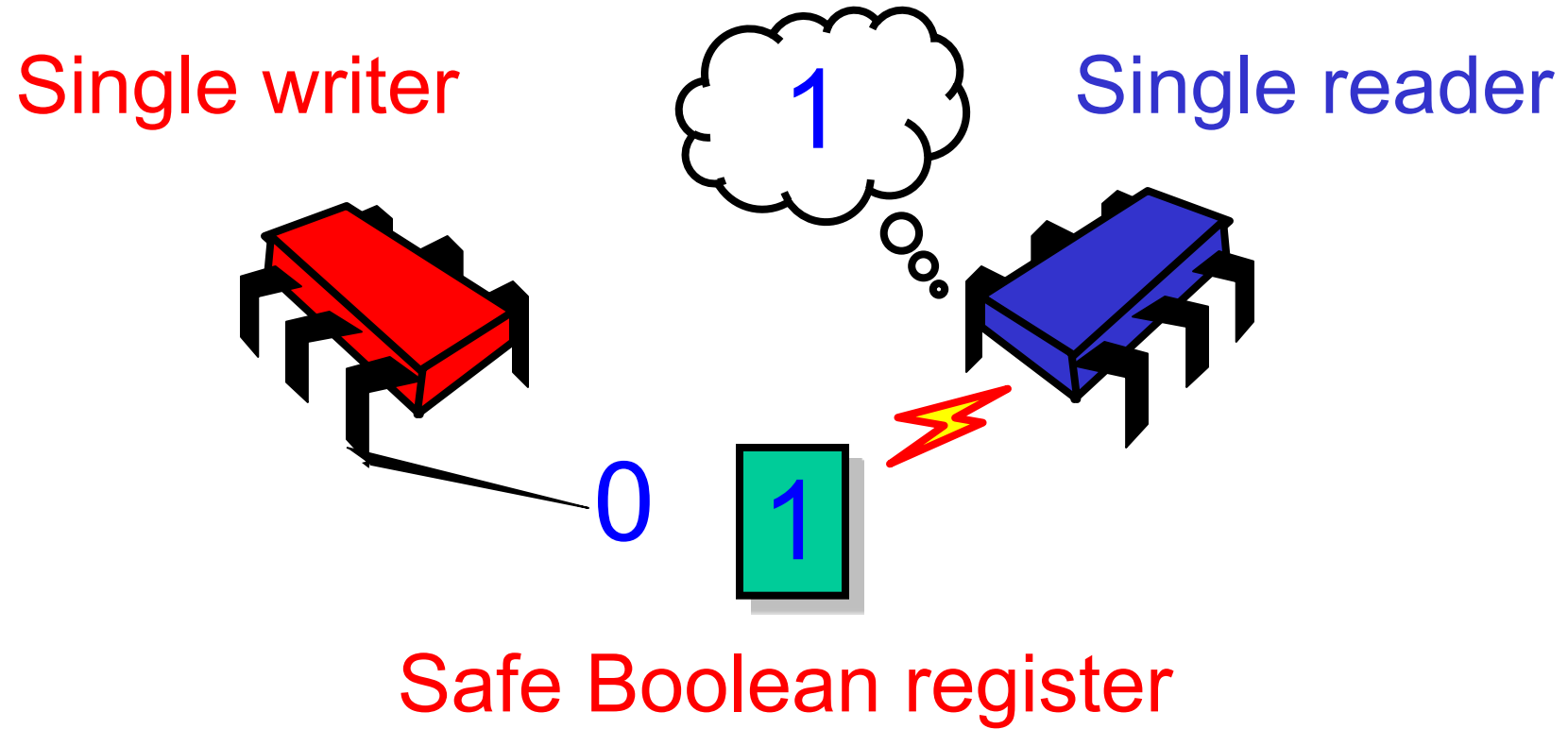
All the way to a Wait-free Implementation of Atomic Snapshots
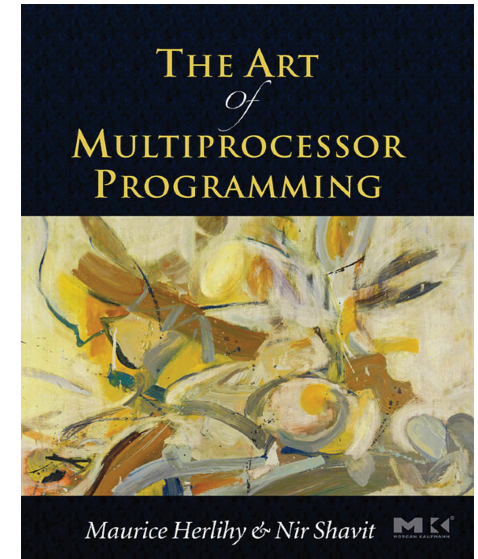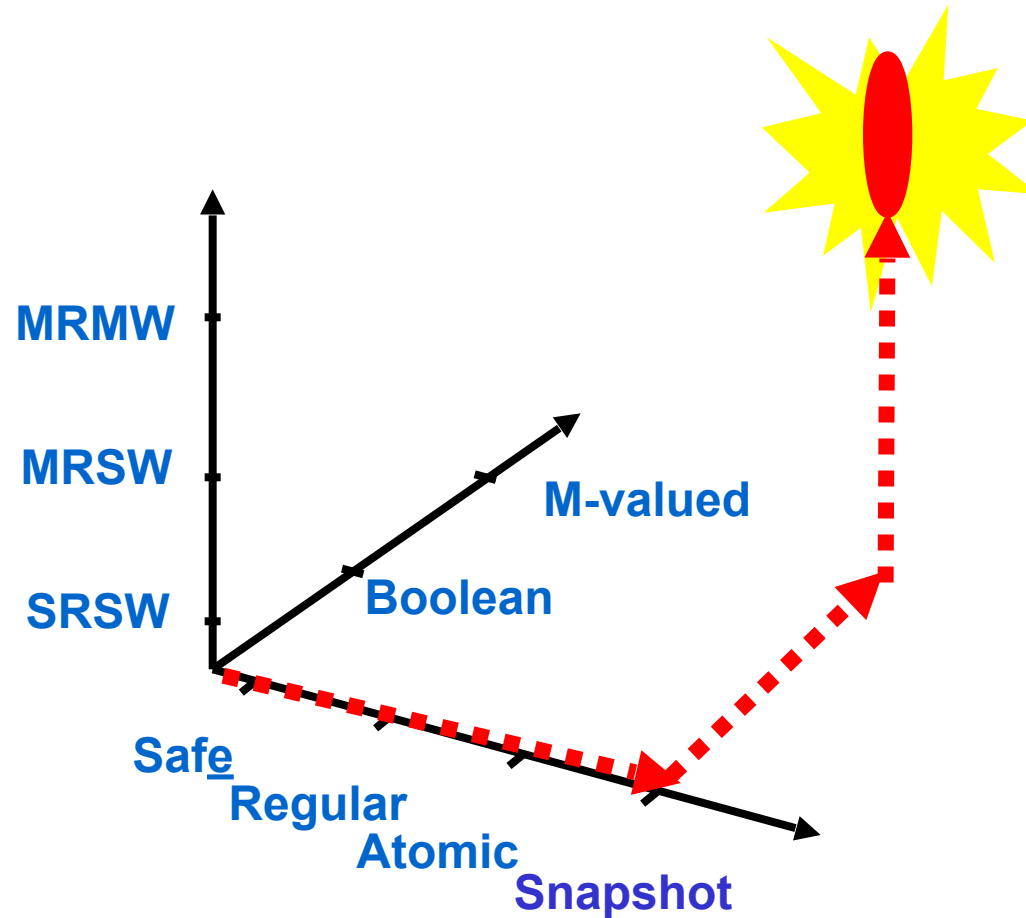
Chapter 4

Wait-Free Implementation

- 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
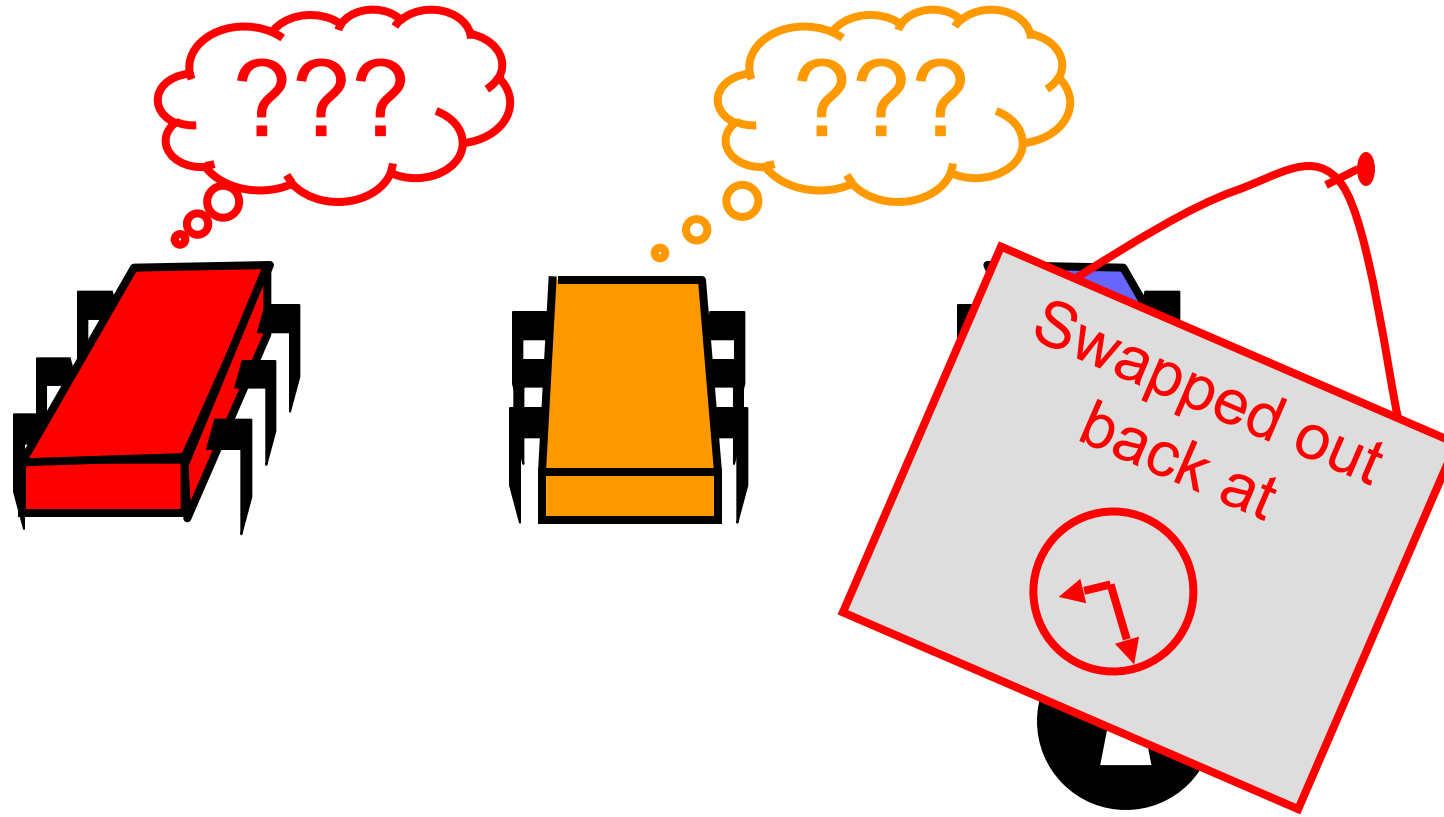
Heterogeneous Processors
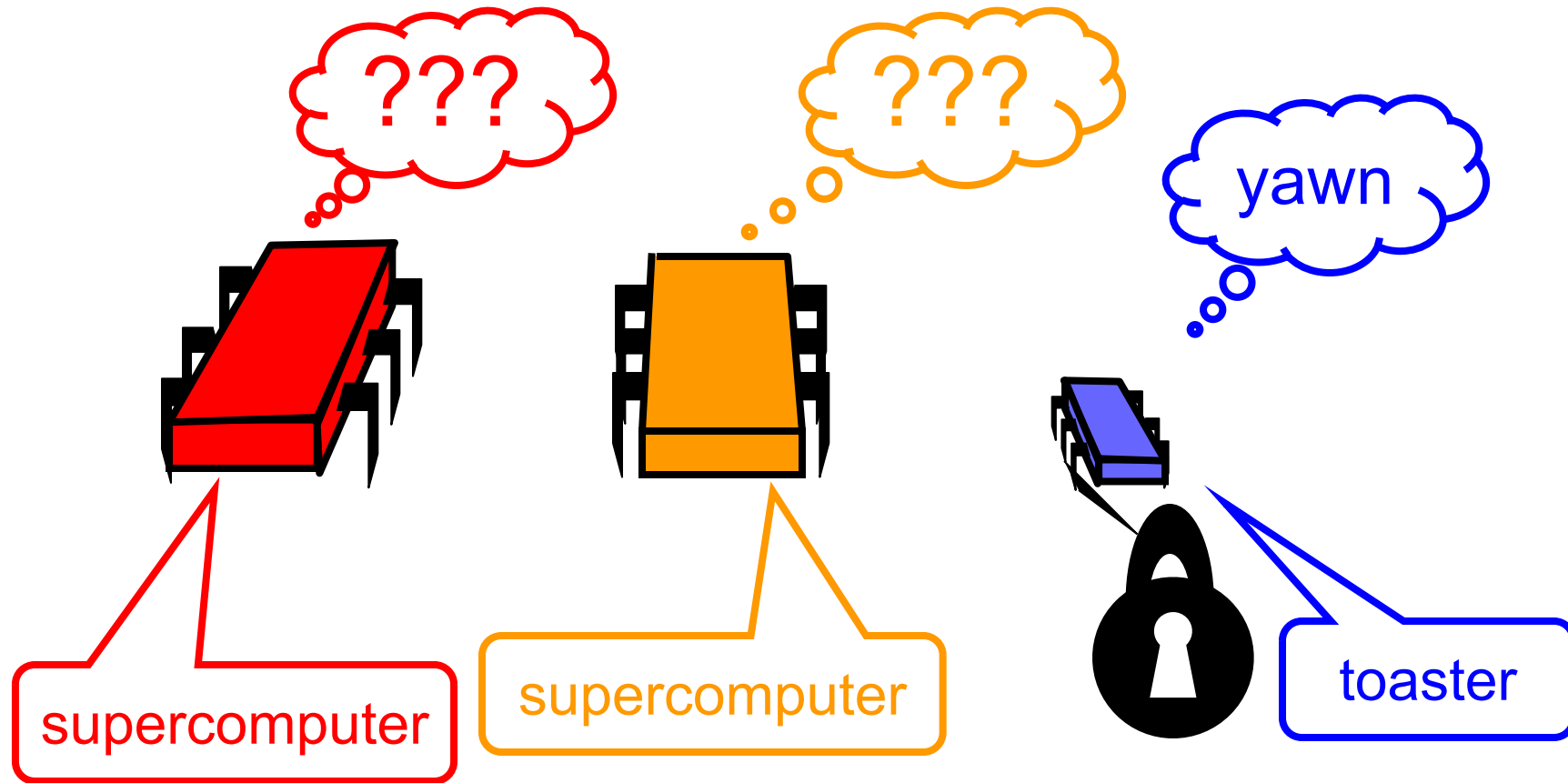
Fault-tolerance
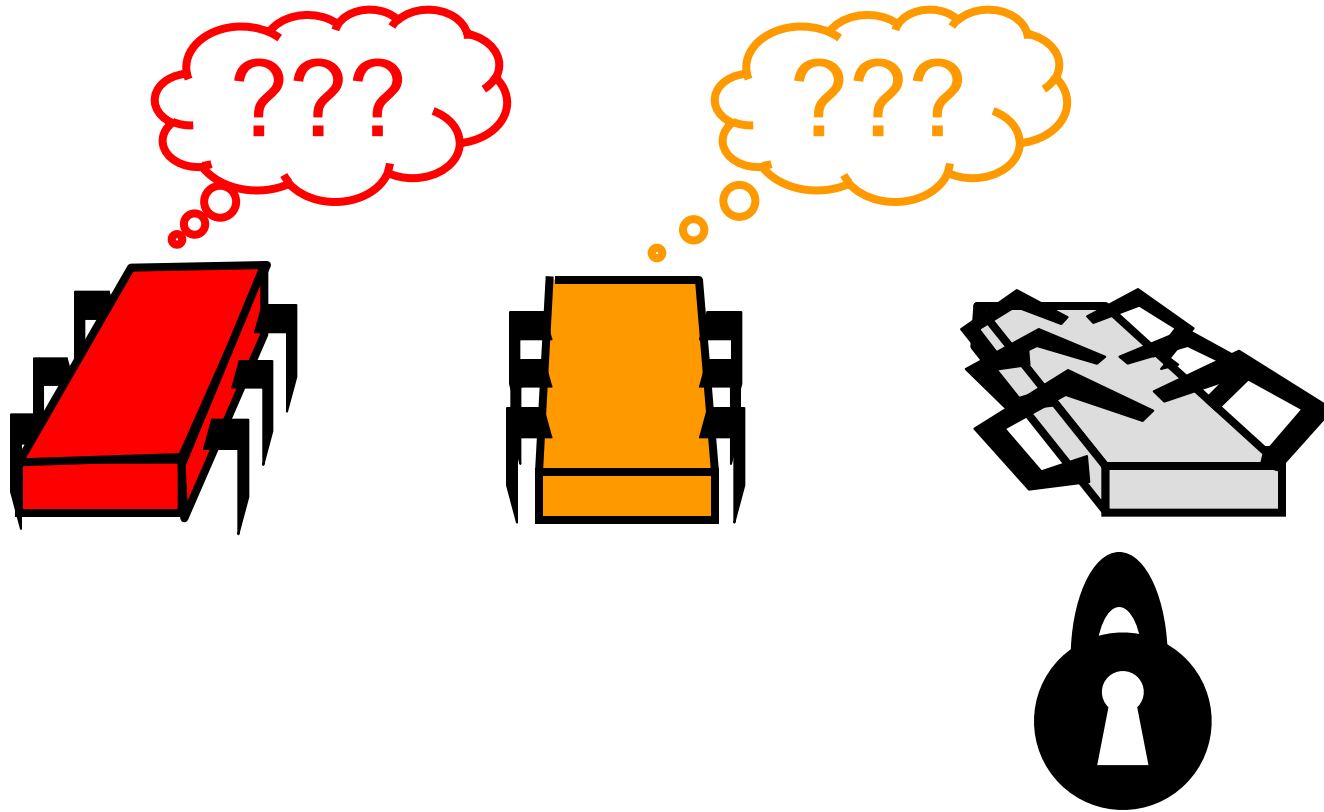
Machine Level Instruction Granularity
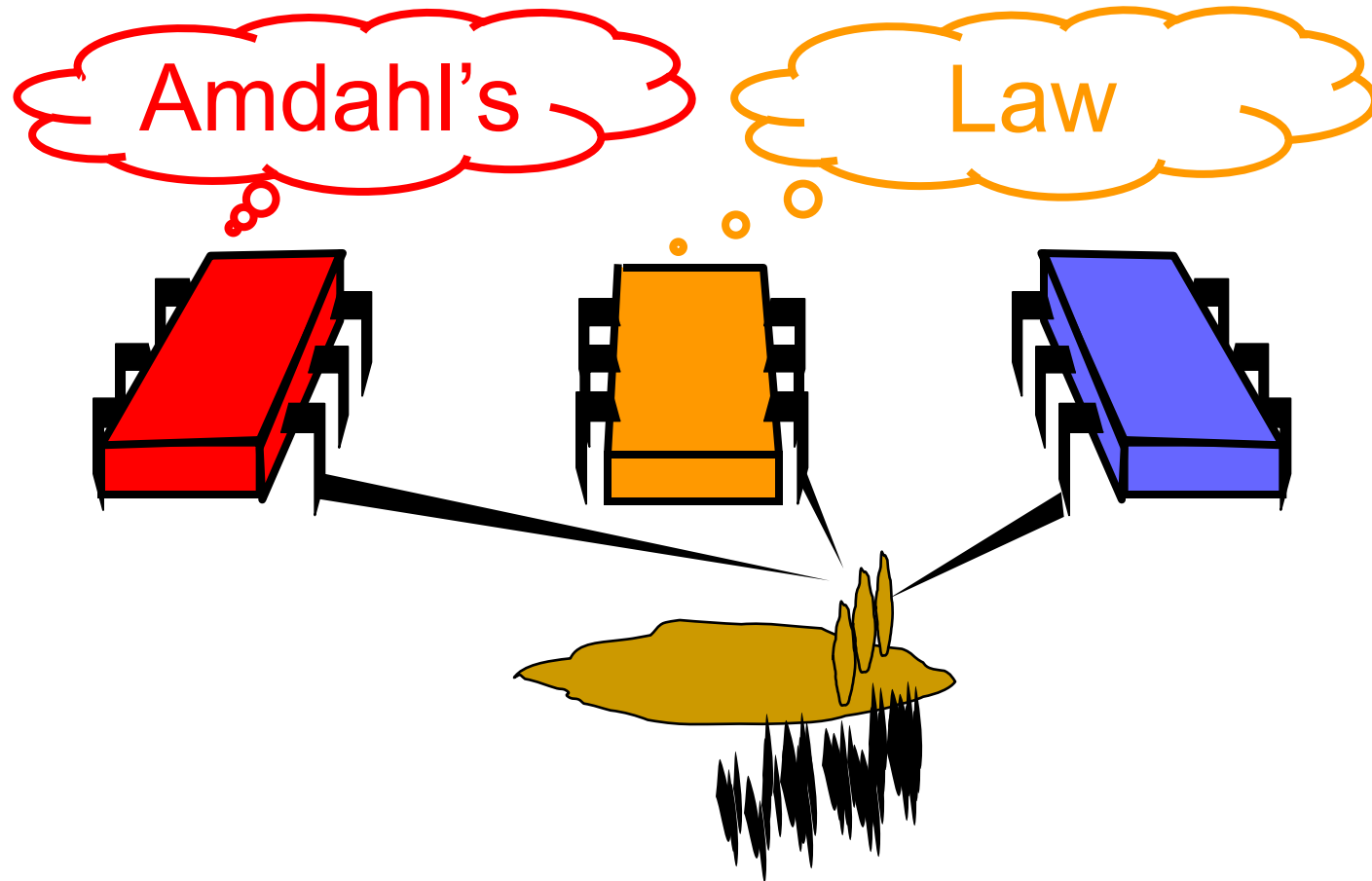
Basic Questions

- Wait-Free synchronization might be a good idea in principle

Basic Questions

- **Wait-Free** synchronization might be a good idea in principle

- But how do you do it …

Basic Questions

- **Wait-Free** synchronization might be a good idea in principle

- But how do you do it …
  - Systematically?

Basic Questions

- **Wait-Free** synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?

Basic Questions

- **Wait-Free** synchronization might be a good idea in principle
- But how do you do it …
  - Systematically?
  - Correctly?
  - Efficiently?

FIFO Queue: Enqueue Method

q.enq( )
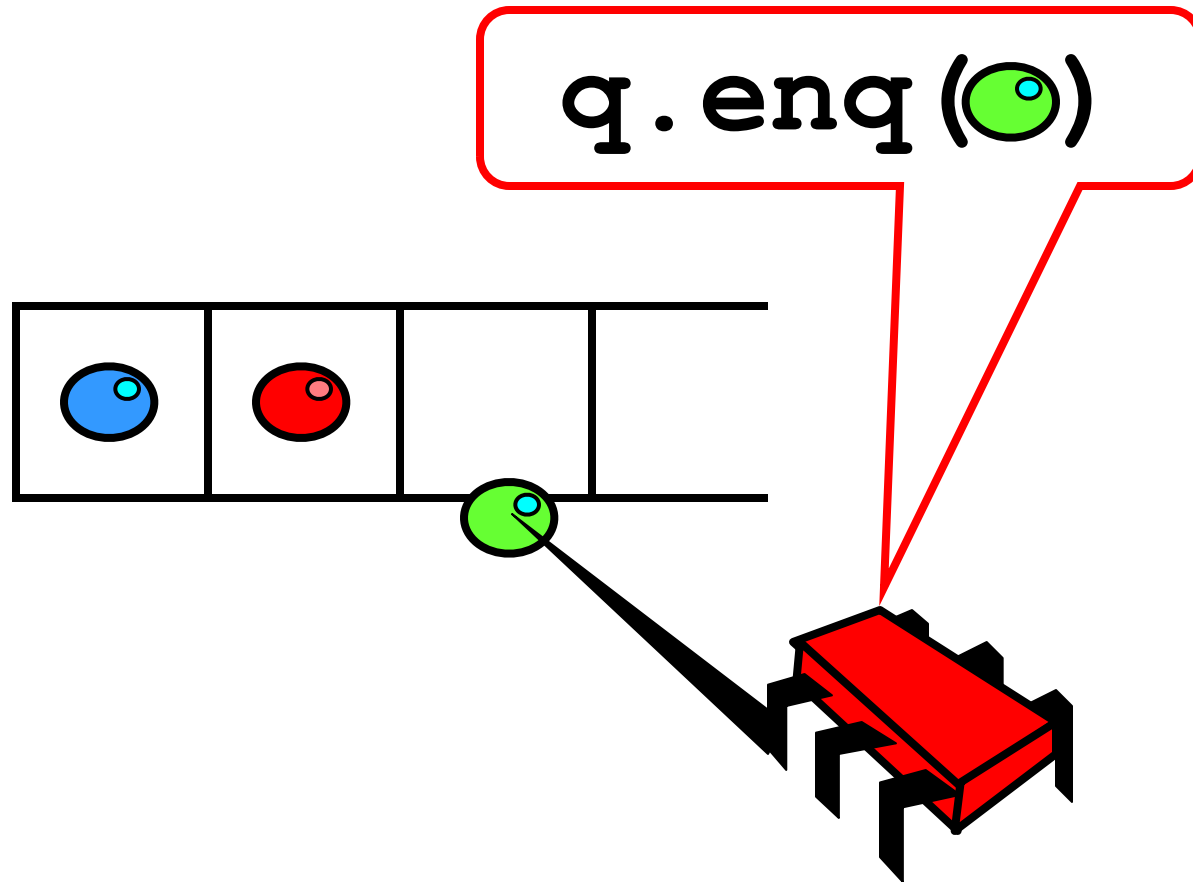
FIFO Queue: Dequeue Method
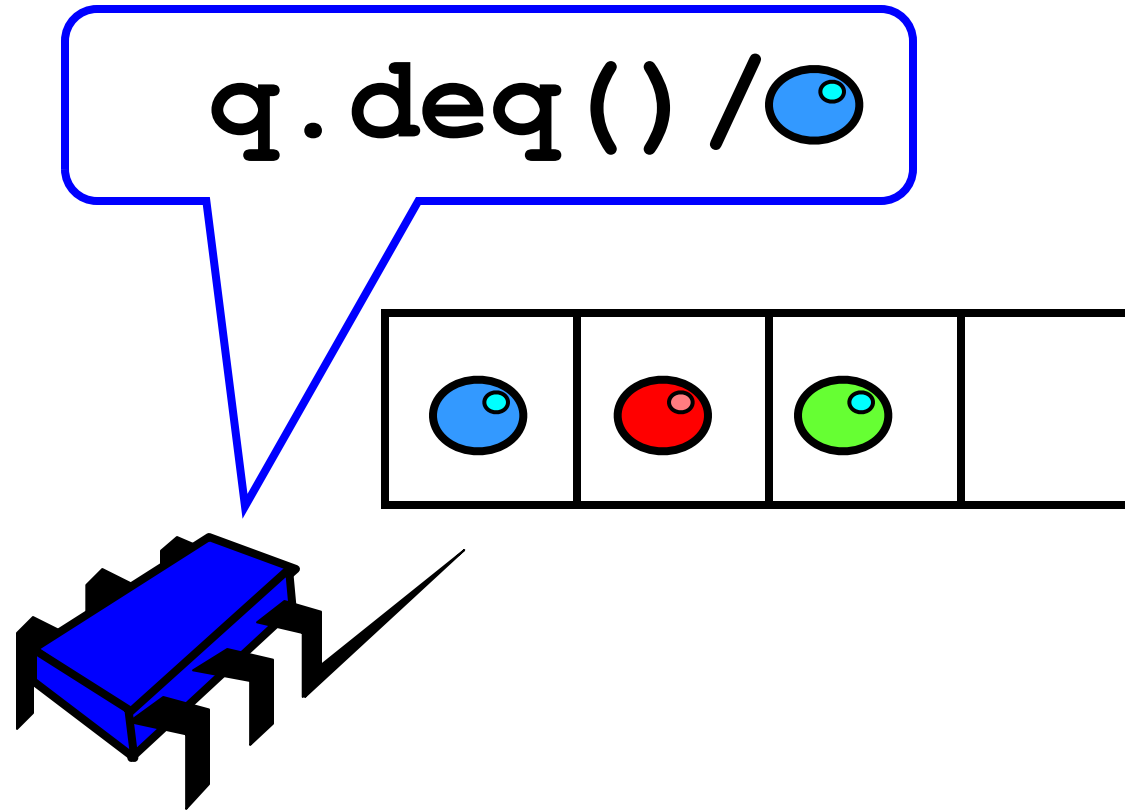
Two-Thread Wait-Free Queue

```scala
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
  }
}
```
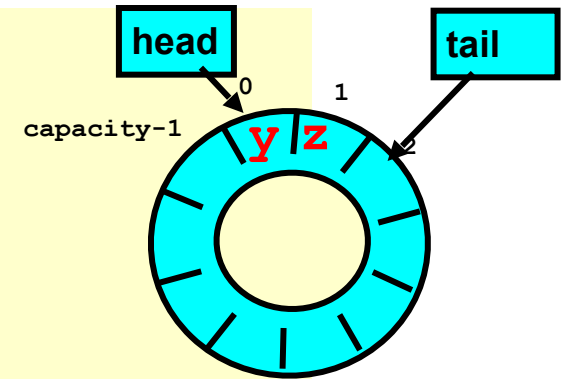
What About Multiple Dequeuers?
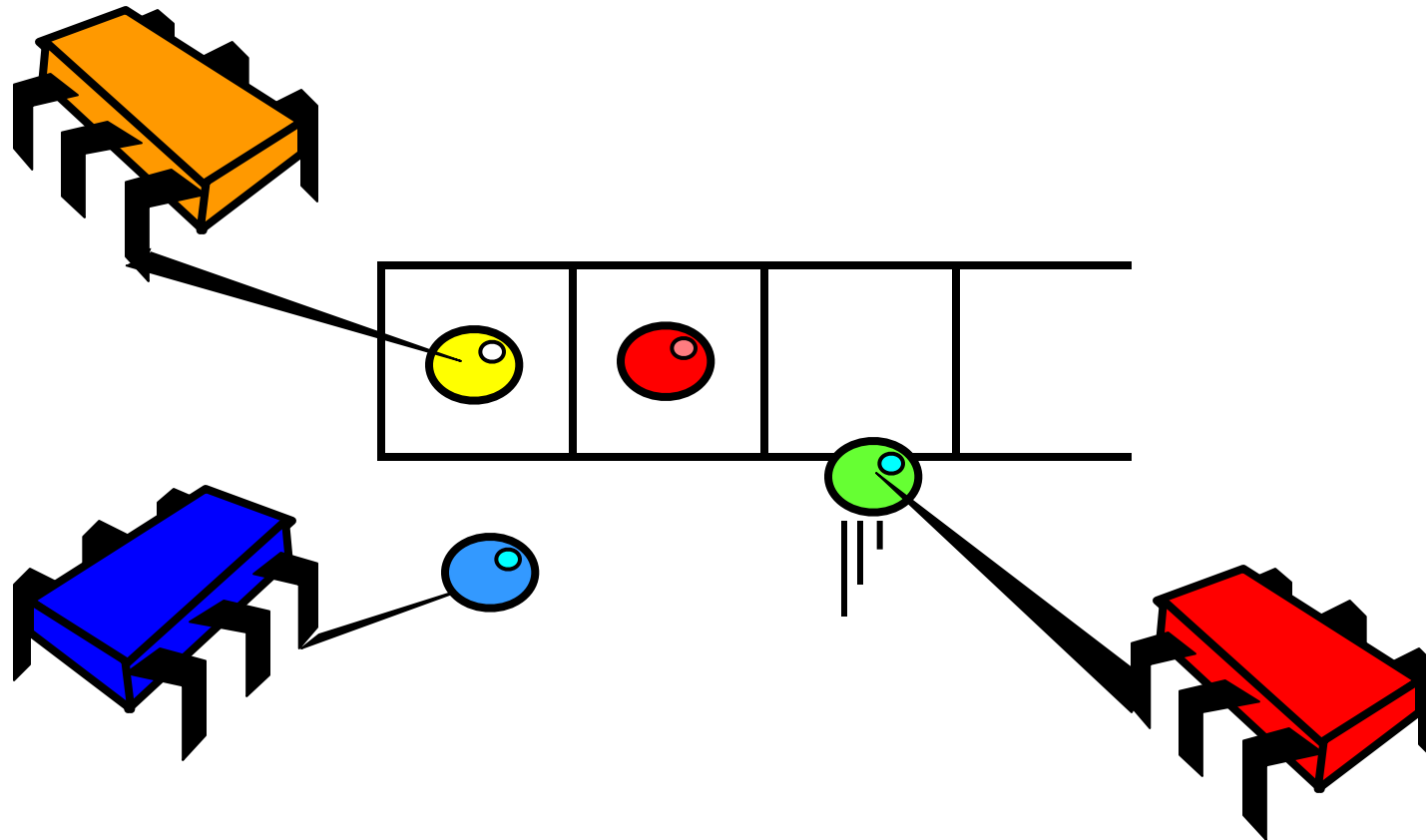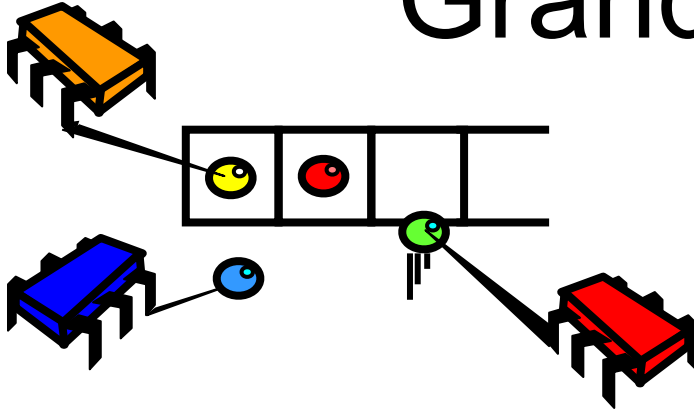
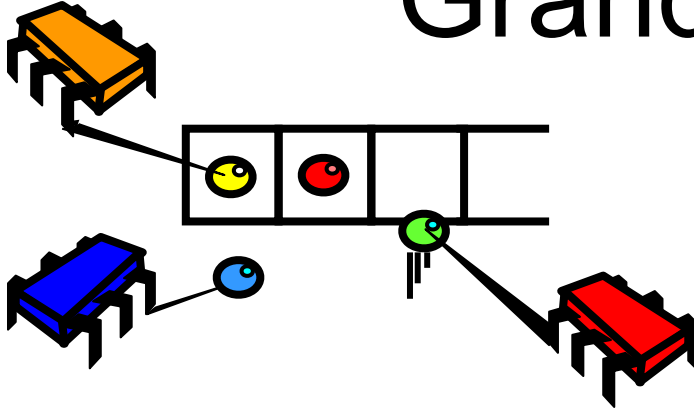# Grand Challenge

- Implement a FIFO queue

Grand Challenge
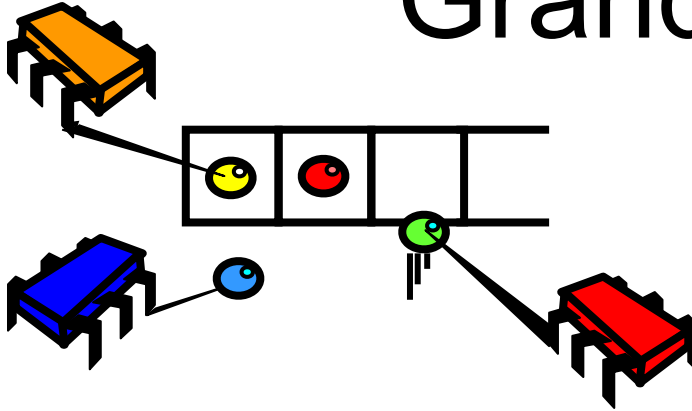
- Implement a FIFO queue
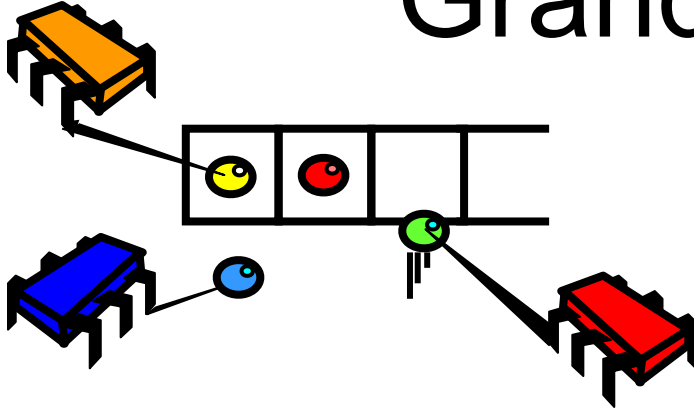 - Wait-free

Grand Challenge
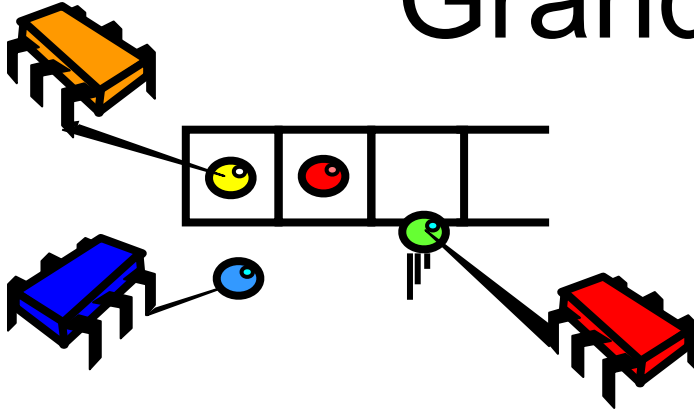
- **Implement a FIFO queue**
  - Wait-free
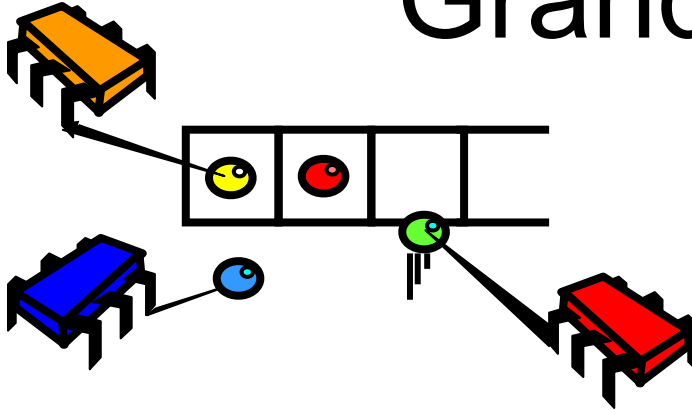 - Linearizable

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
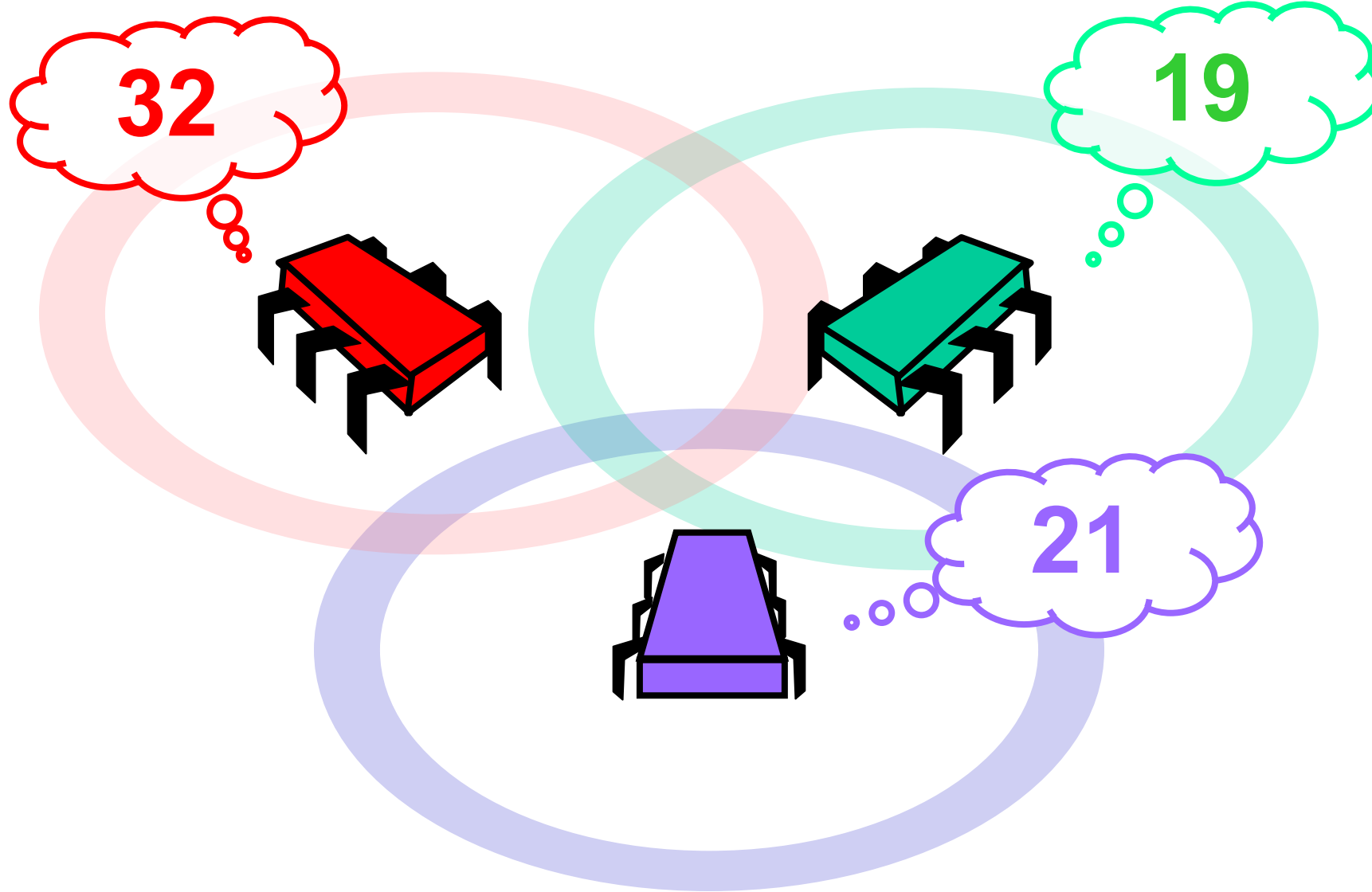 - **Multiple dequeuers**

Only new aspect

Puzzle

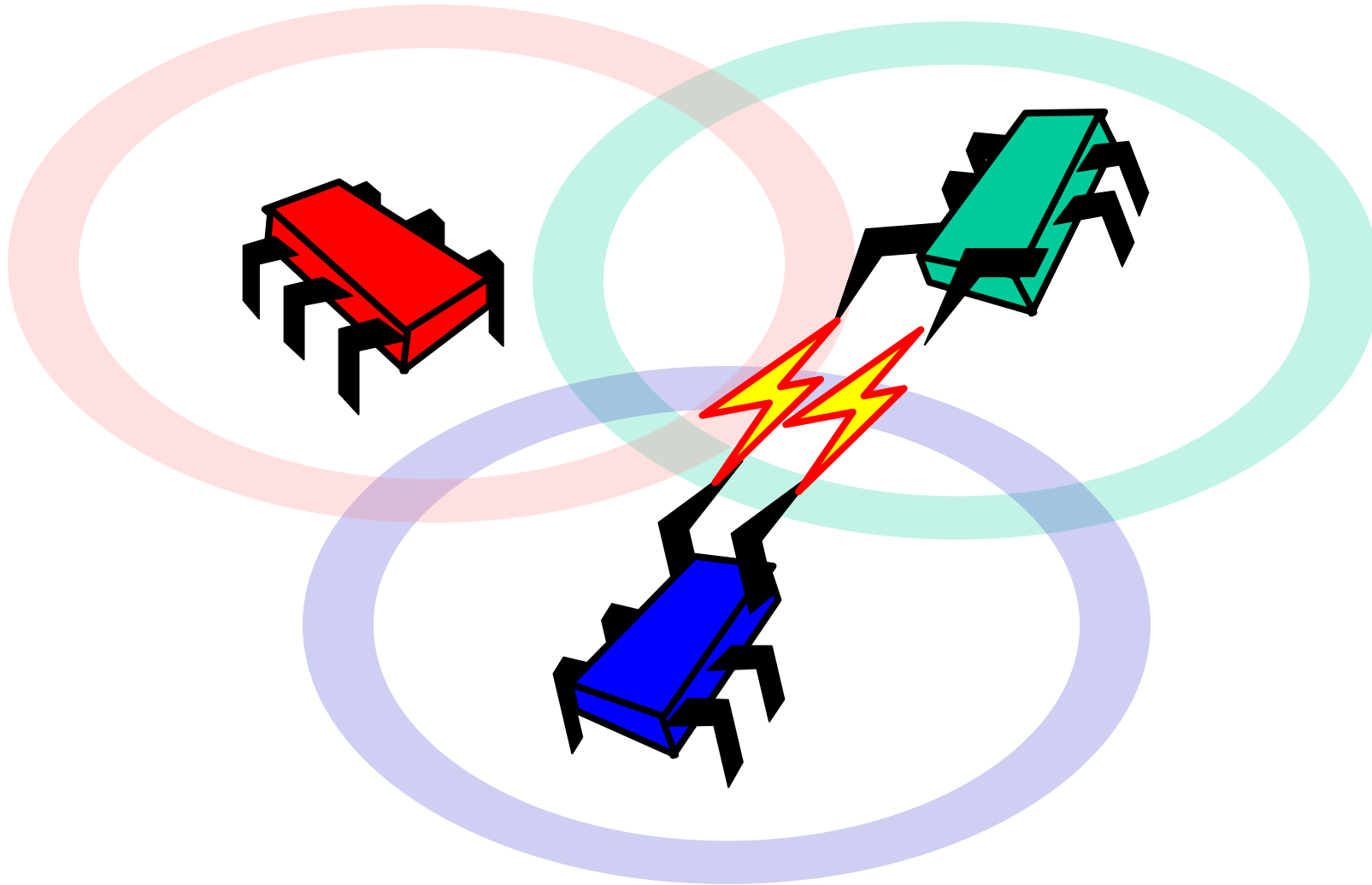While you are ruminating on the grand challenge …

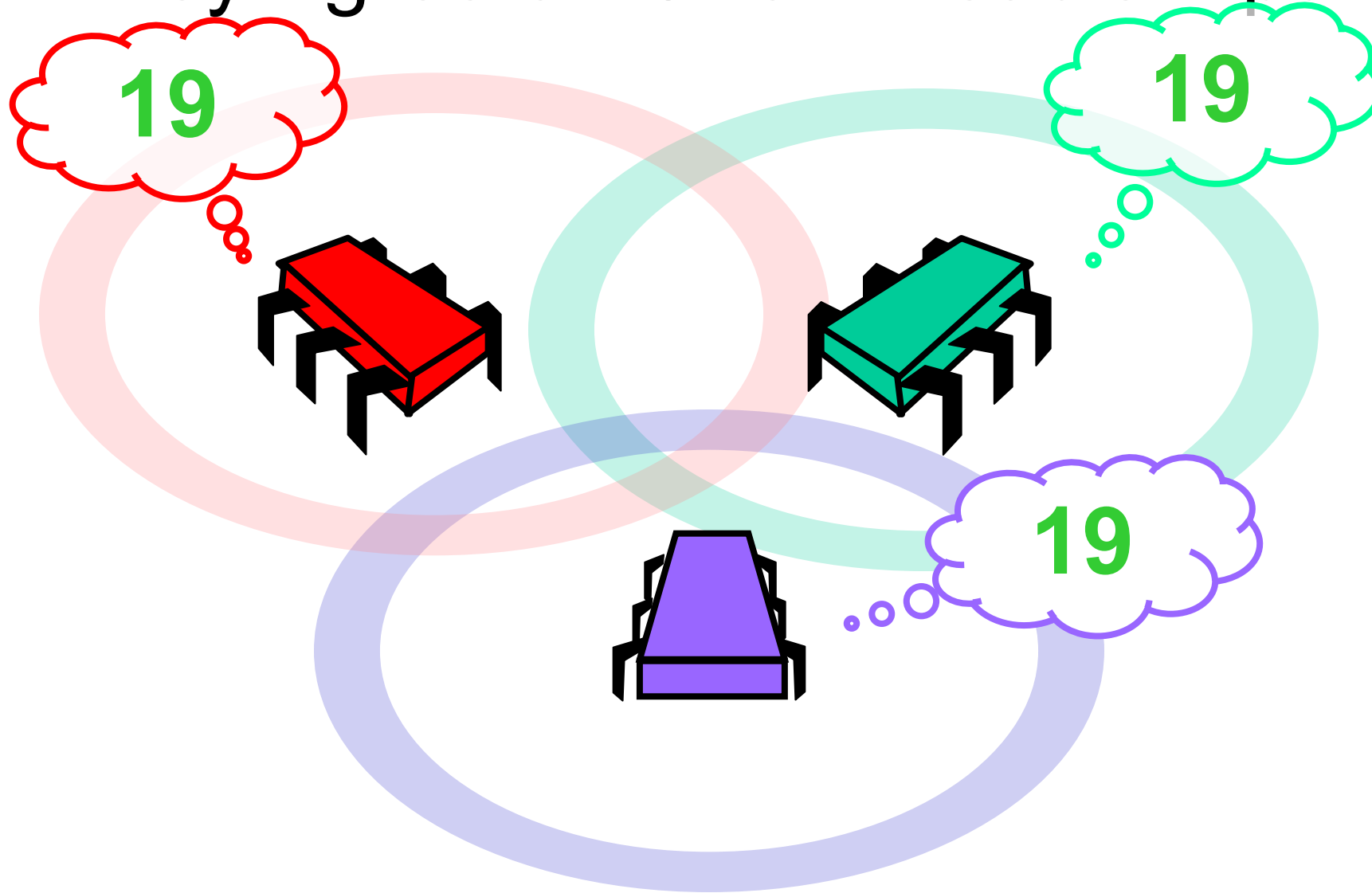We will give you another puzzle …

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

Let's play the consensus *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)

Game

No Wait-Free Implementation of Consensus using Registers
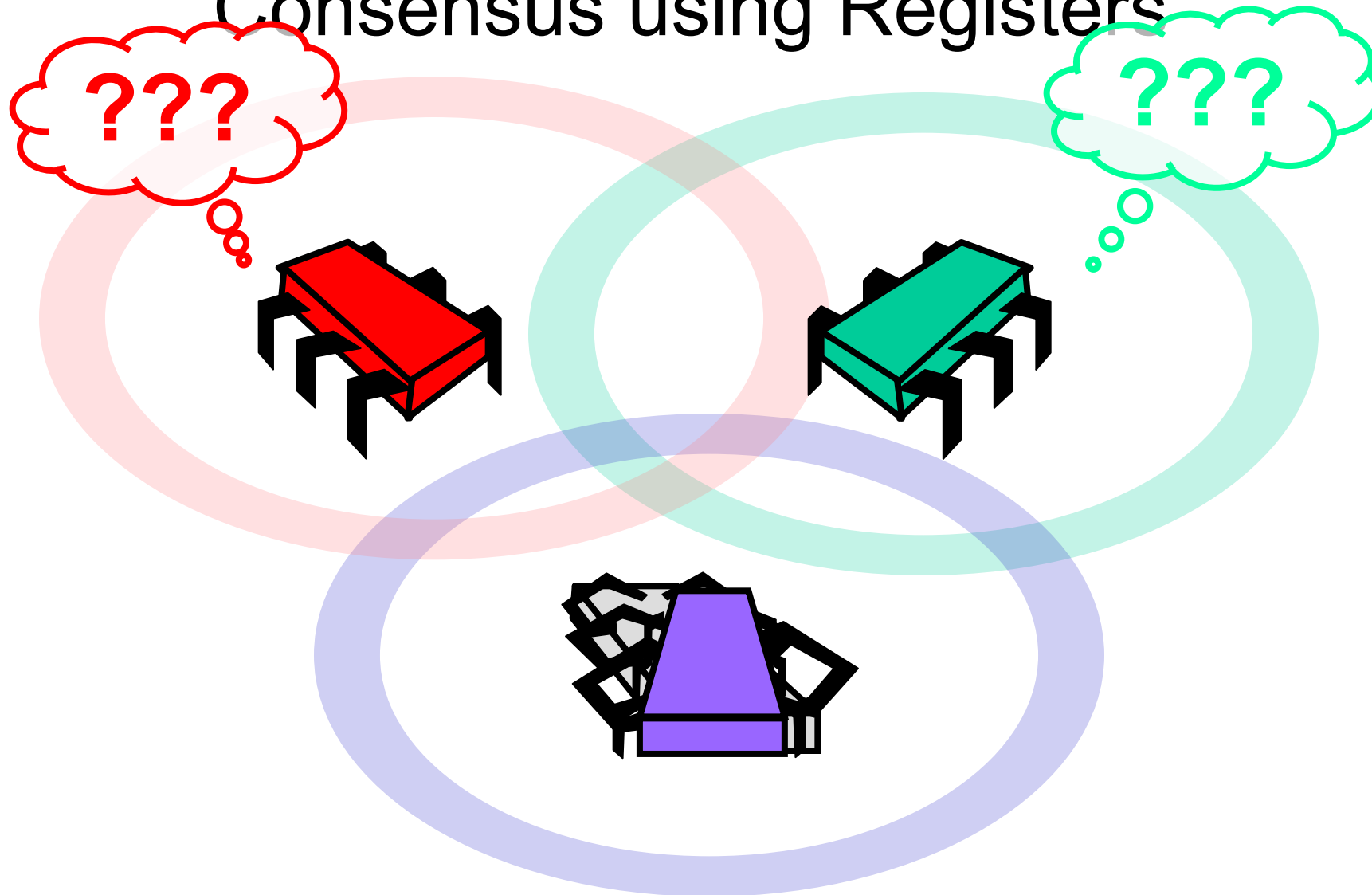
Formally

- ## Theorem
  - There is no wait-free implementation of *n*-thread consensus from read-write registers
- ## Implication
  - Asynchronous computability different from Turing computability
- ## Proof
  - Using "valence trees" — see the textbook

Next lecture:
Solving n-thread Consensus