Concurrent Objects

Please read sections 3.7 and 3.8

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

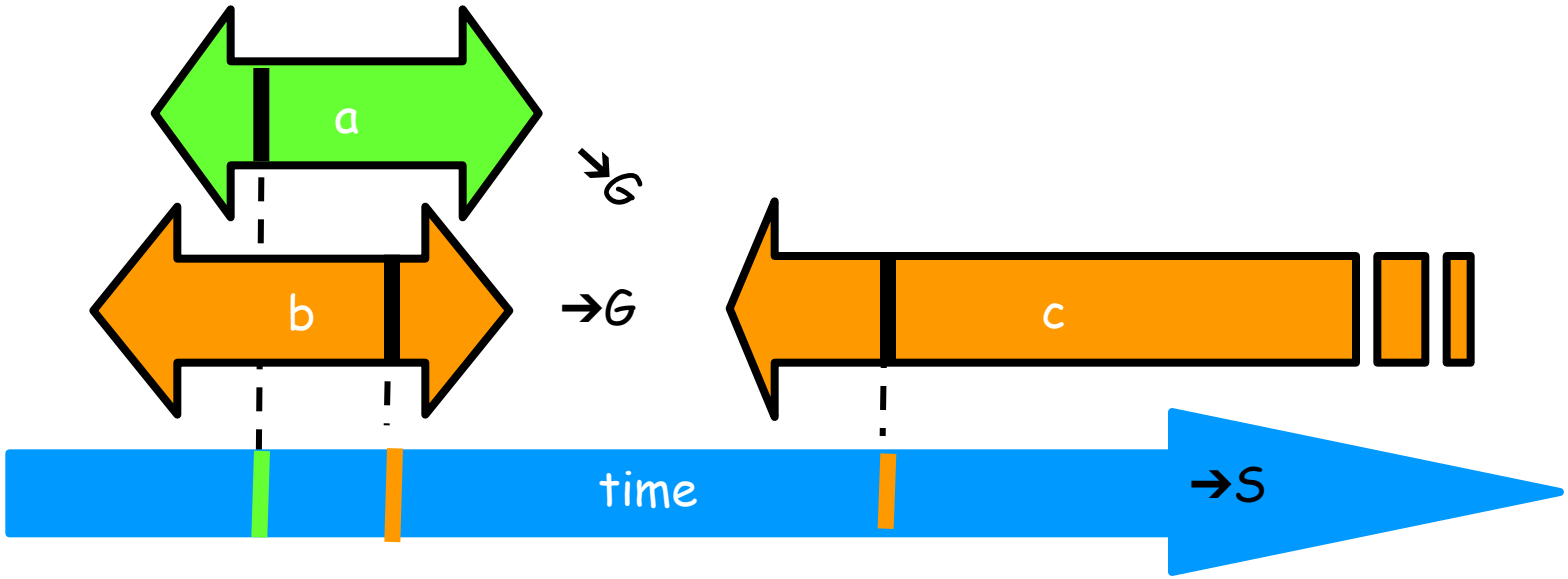
Linearizability

- 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 $\rightarrow_G \subset \rightarrow_S$

What is $\rightarrow_G \subset \rightarrow_S$

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

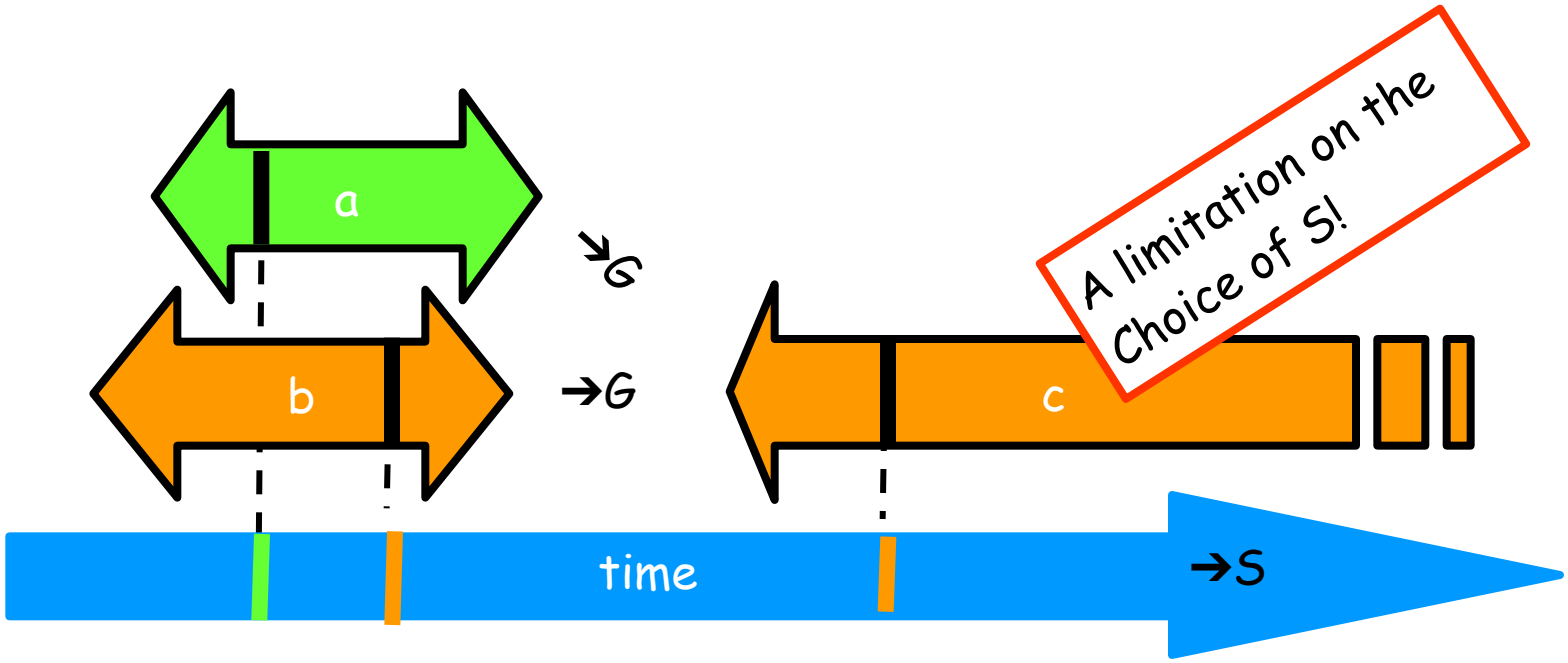
$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$



What is $\rightarrow_G \subset \rightarrow_S$

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

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

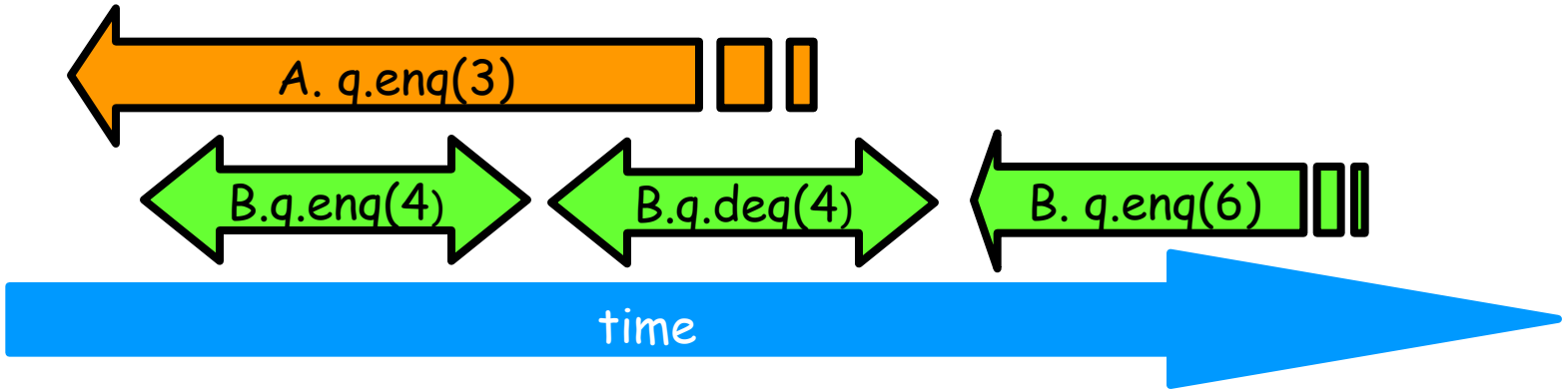


Remarks

- 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***

Example

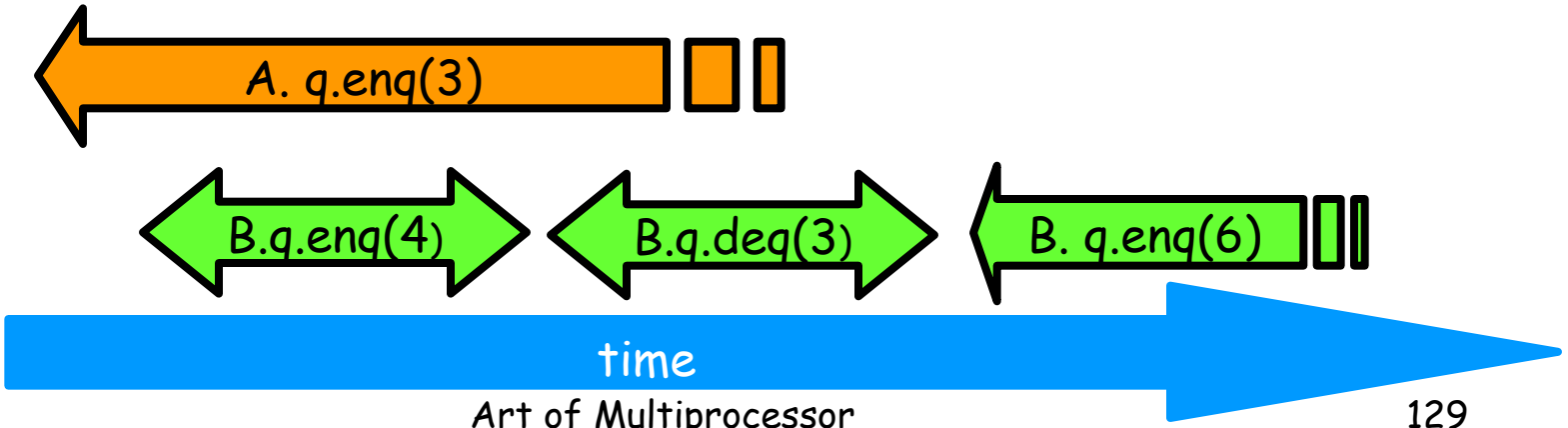
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)



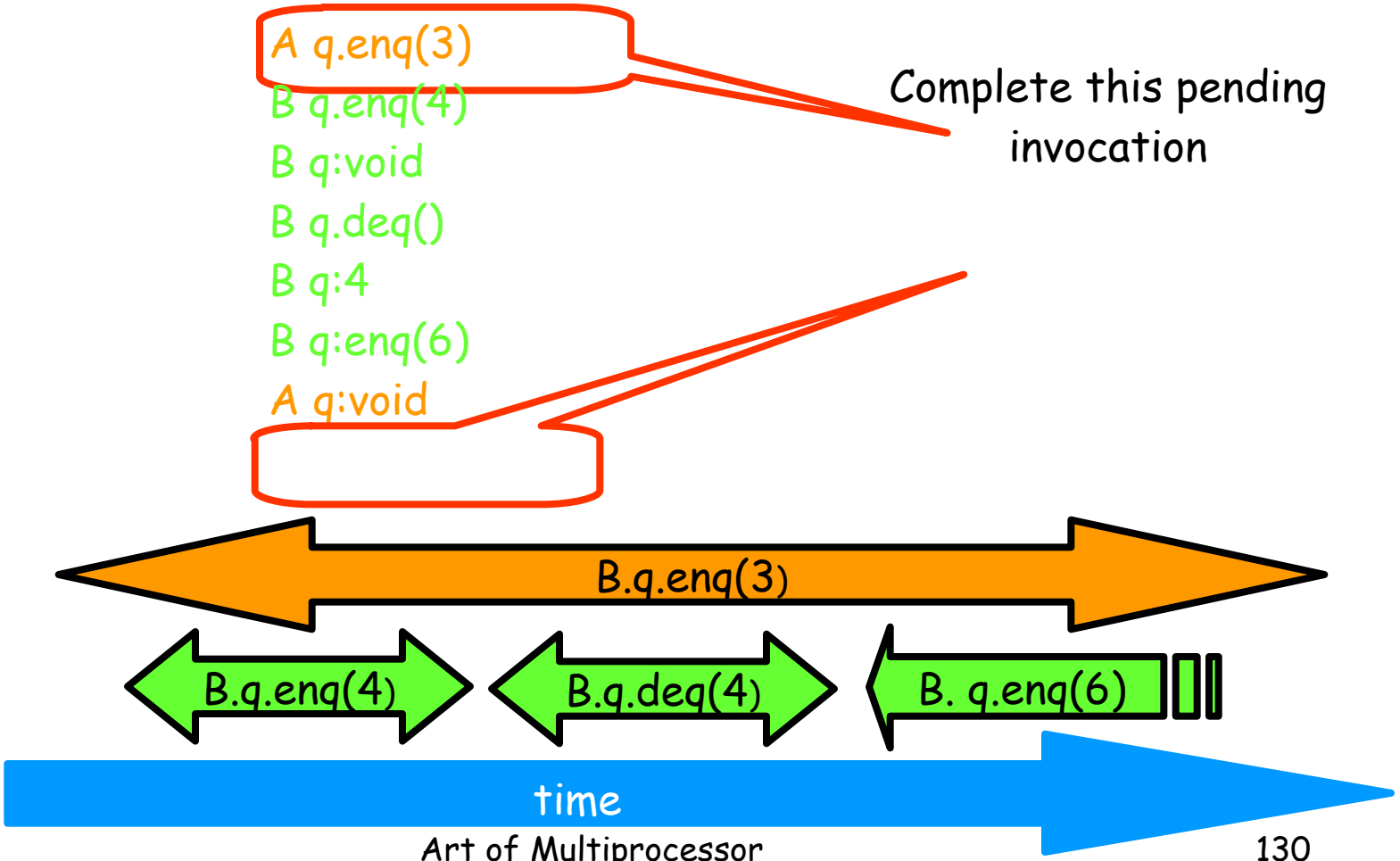
Example

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



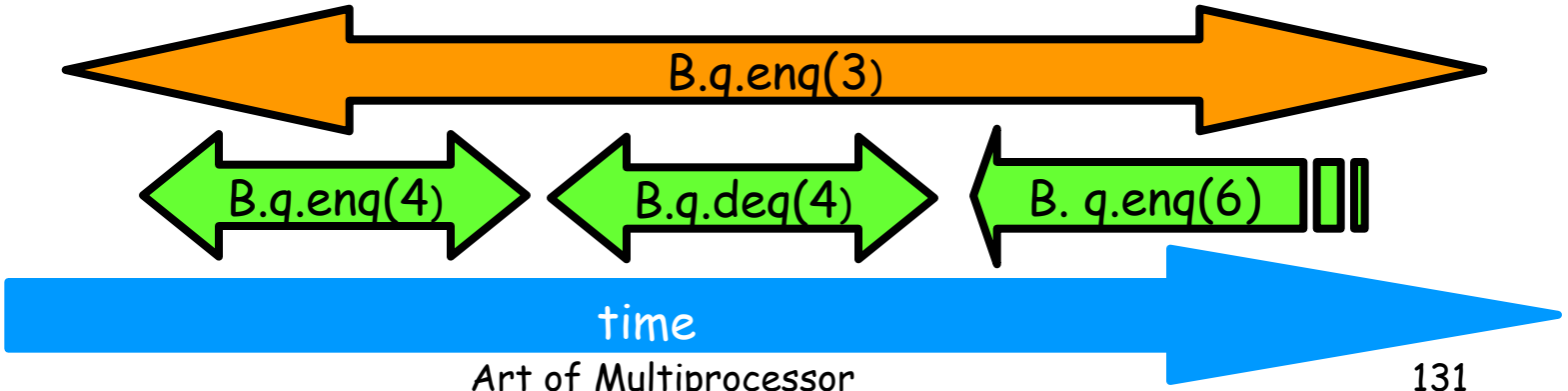
Example


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

Example

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

discard this one

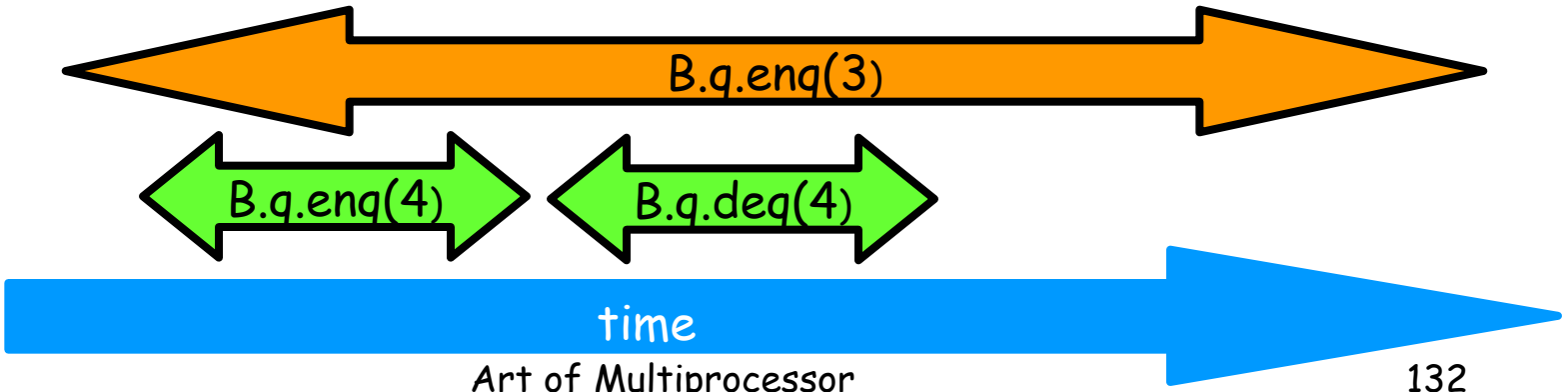


Example

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

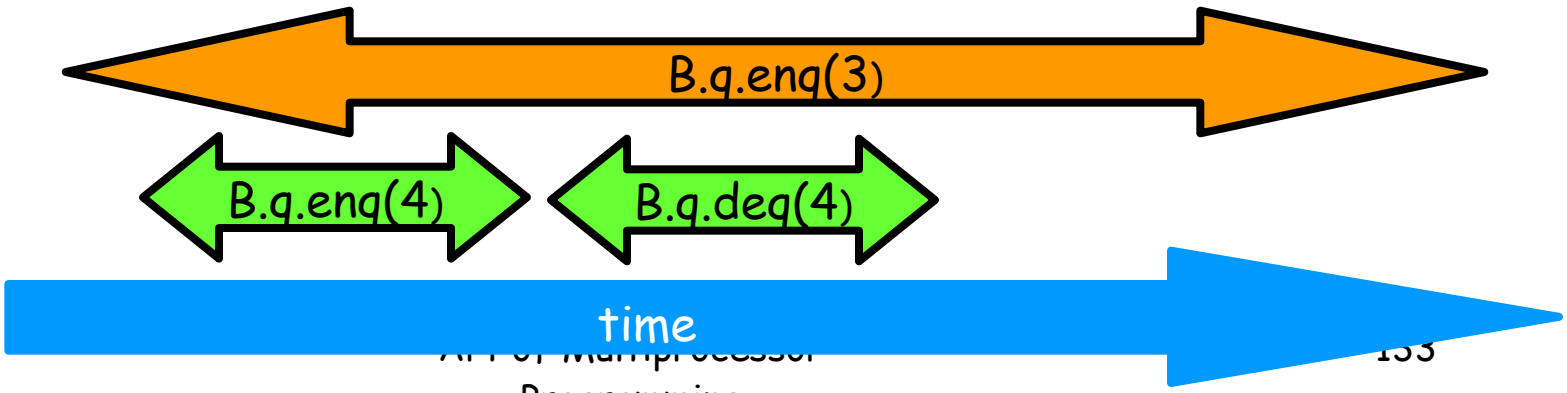
discard this one

A q:void



Example

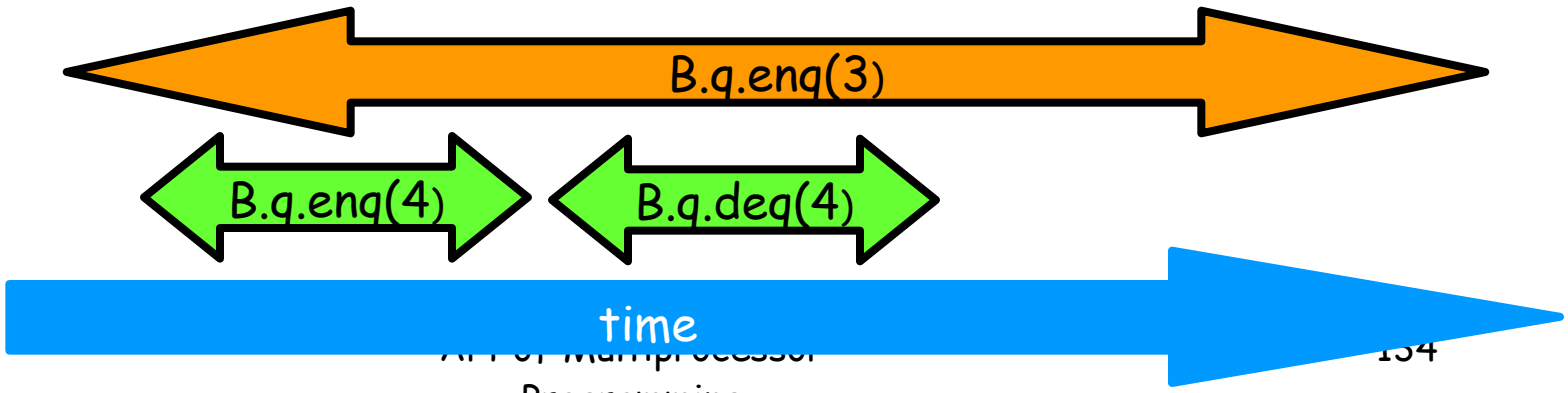
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void



Example

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

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

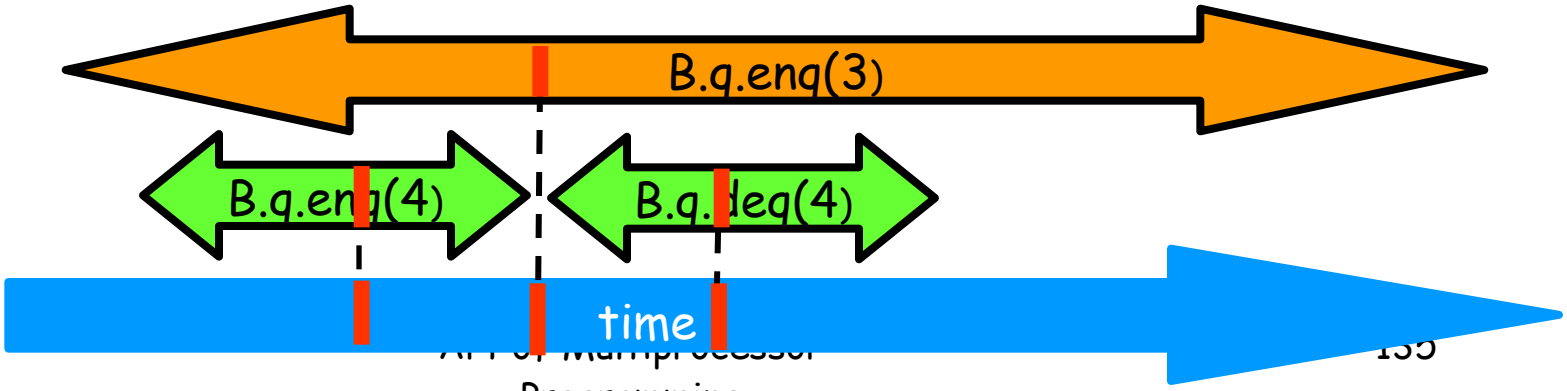


Example

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



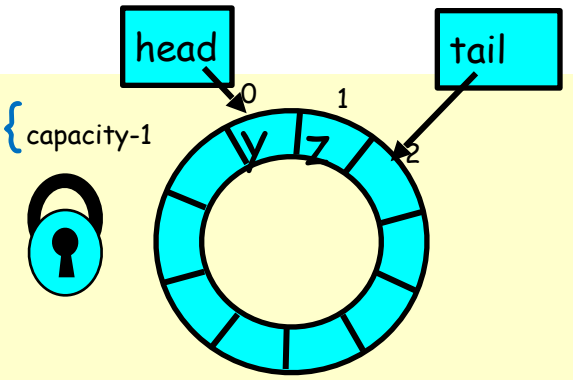
Composability Theorem

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

Why Does Composability Matter?

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

Reasoning About Lineraizability: Locking




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

Reasoning About Lineraizability: Locking

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

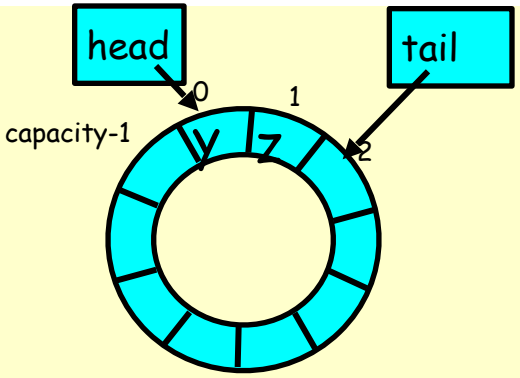
Linearization points are when locks are released

More Reasoning: Lock-free

```
public class LockFreeQueue {

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

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

More Reasoning: Lock-free

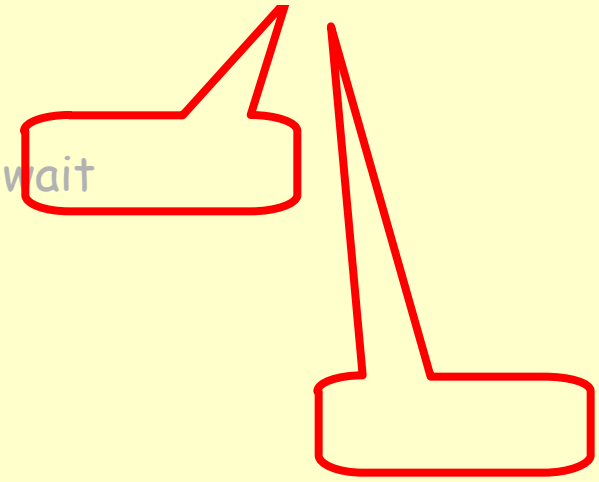
```
public class LockFreeQueue {

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

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```



More Reasoning

```
public class LockFreeQueue

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

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Linearization order is order head and tail fields modified

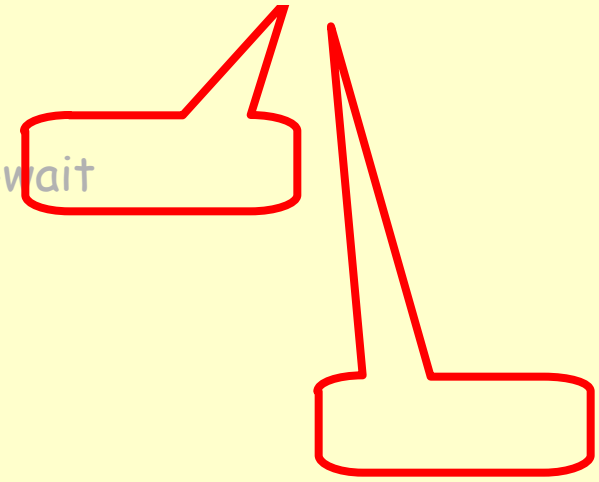


More Reasoning

```
public class LockFreeQueue {

  int h = 0;
  i  = new Object[c

  void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
    while (tail == head);     // busy-wait
    Item item = items[head % capacity]; head++;
    return item;
}}
```

Remember that there is only one enqueuer and only one dequeuer

Linearization order is order head and tail fields modified



Strategy

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

Linearizability: Summary

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

Alternative: Sequential Consistency

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

Alternative: Sequential Consistency

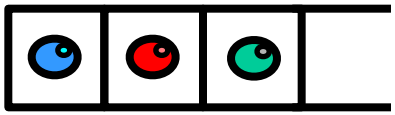
- History H is Sequentially Consistent if it can be extended to G by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to a
  - Legal sequential history S
  - Where $\rightarrow_G \subset \rightarrow_S$

Differs from linearizability

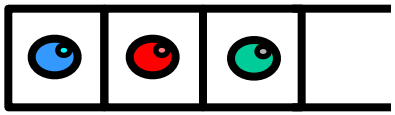
Alternative: Sequential Consistency

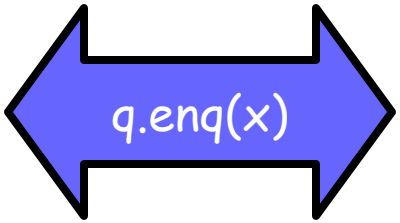
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

Example

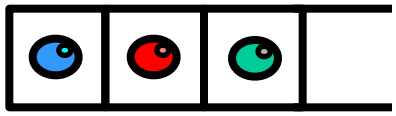


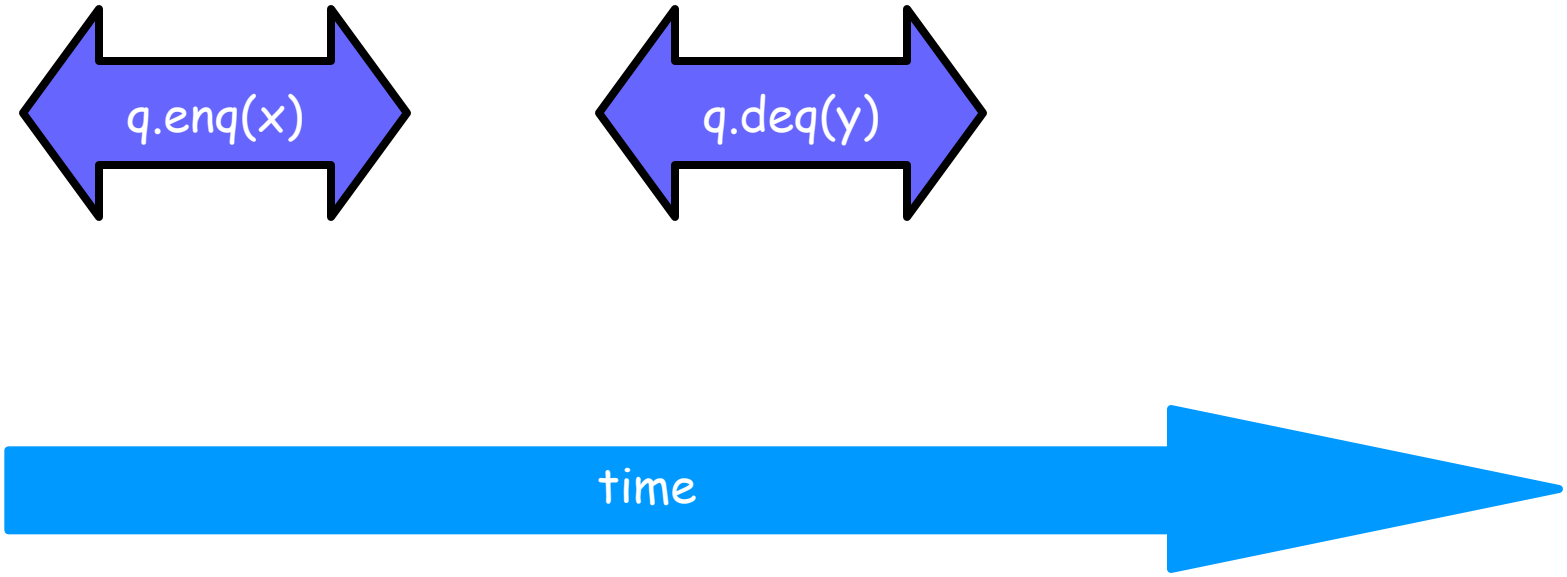
Example



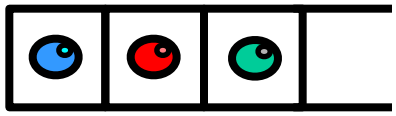


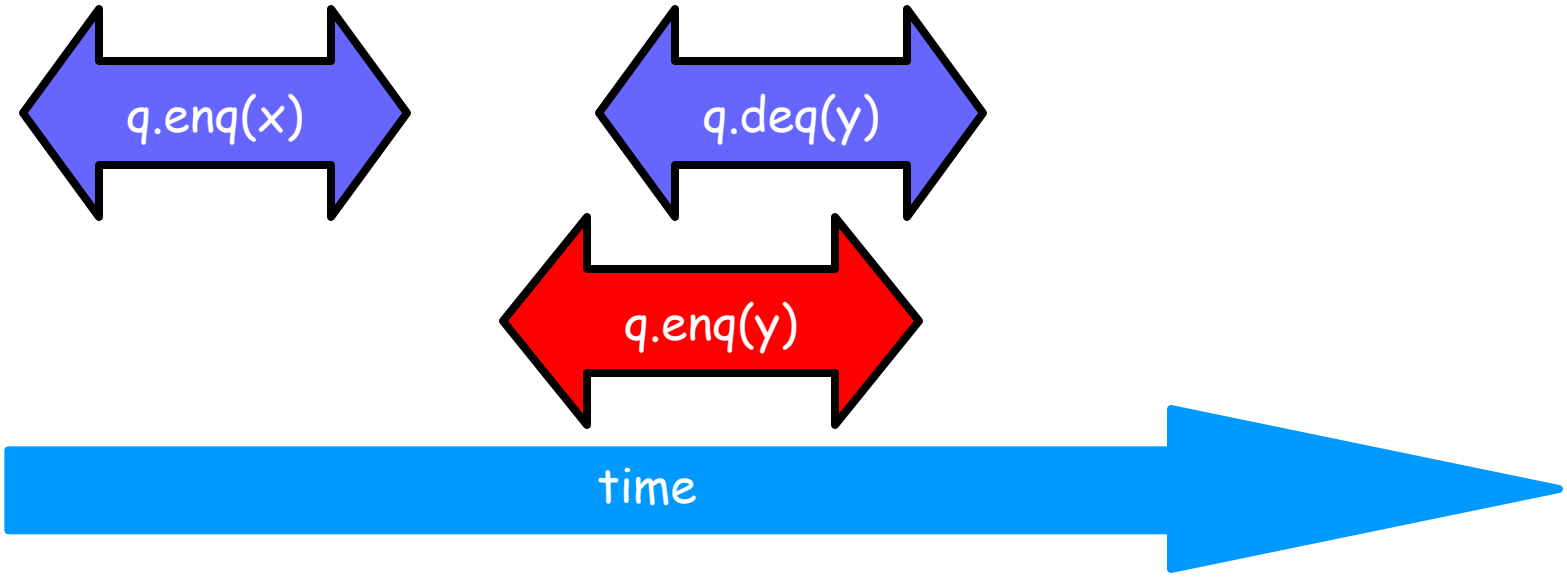
Example



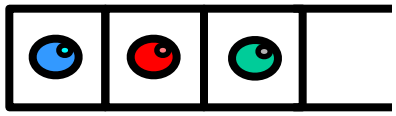


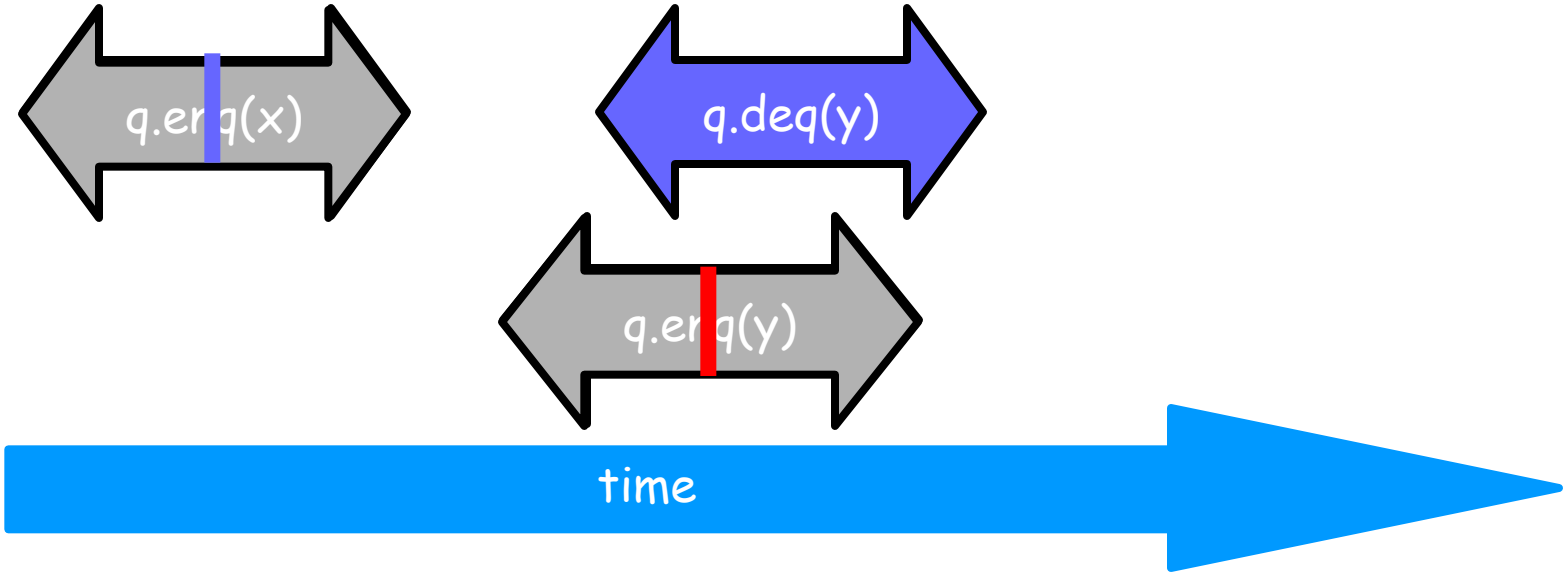
Example



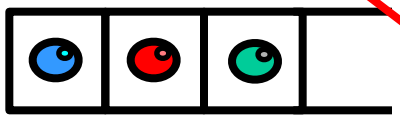


Example

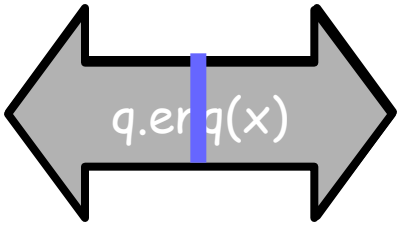


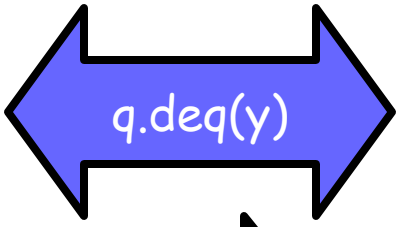


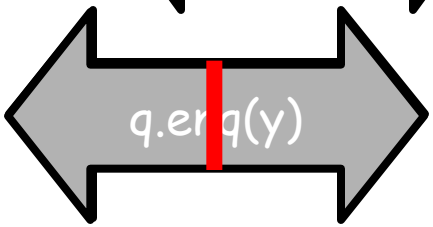
Example

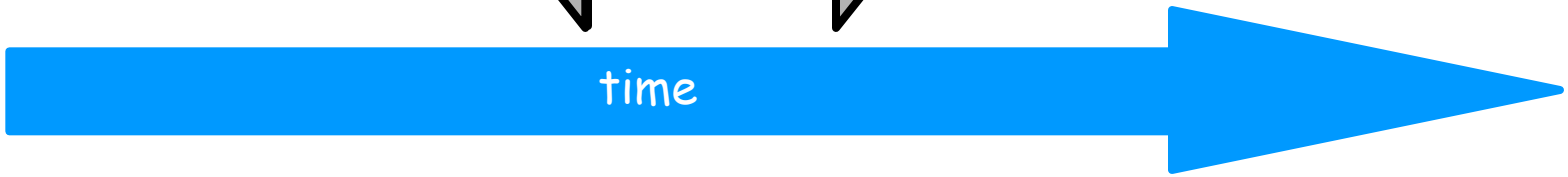


not linearizable



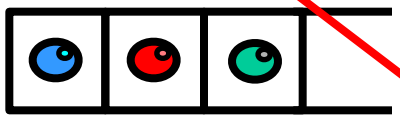


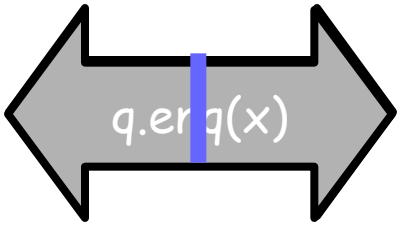


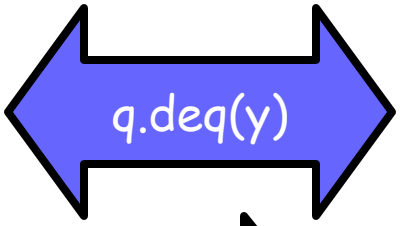


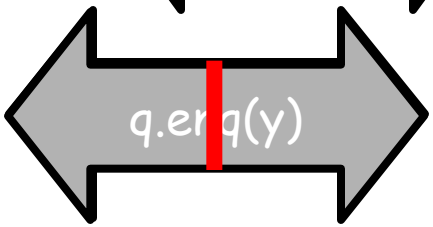
Example

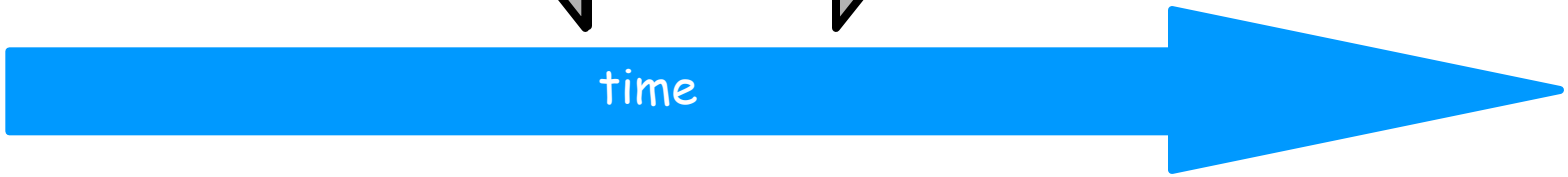
Sequentially Consistent









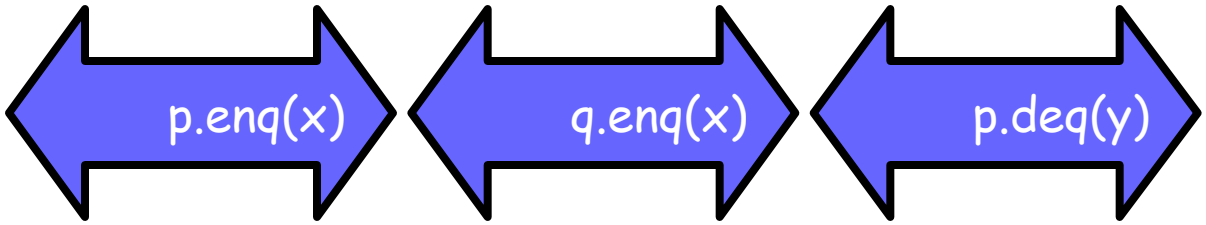


Theorem

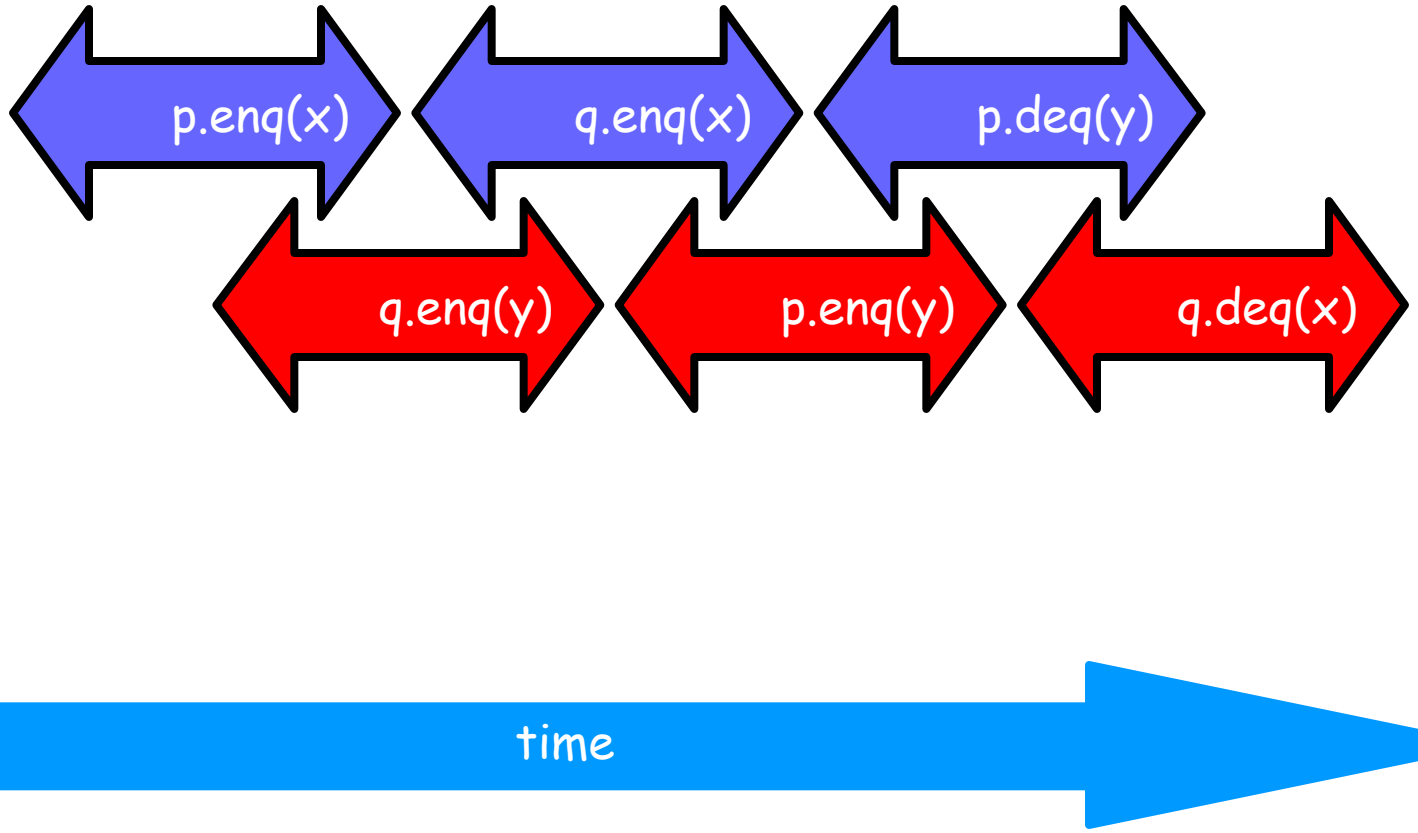
Sequential Consistency is not a local property

(and thus we lose composability…)

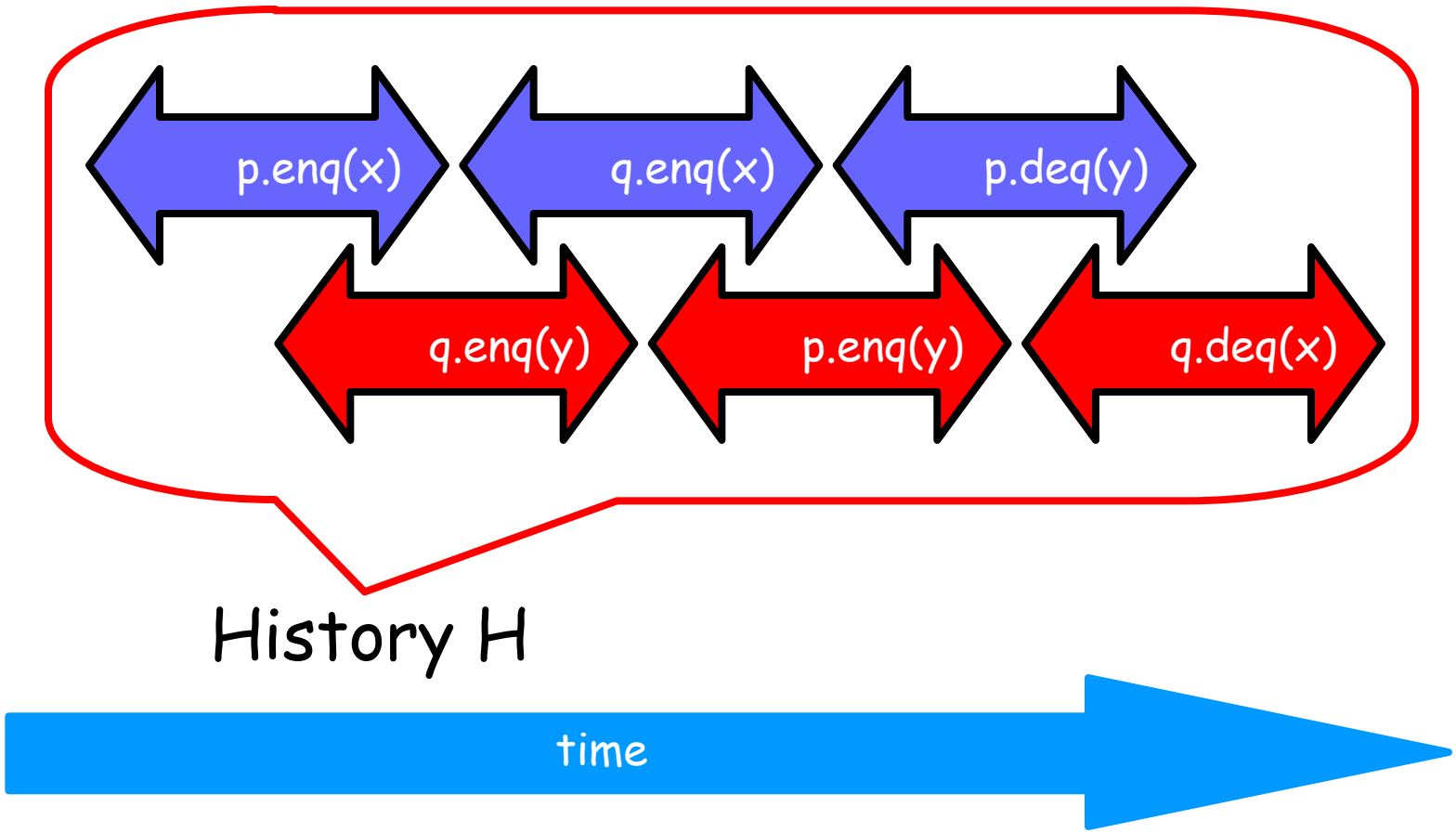
FIFO Queue Example



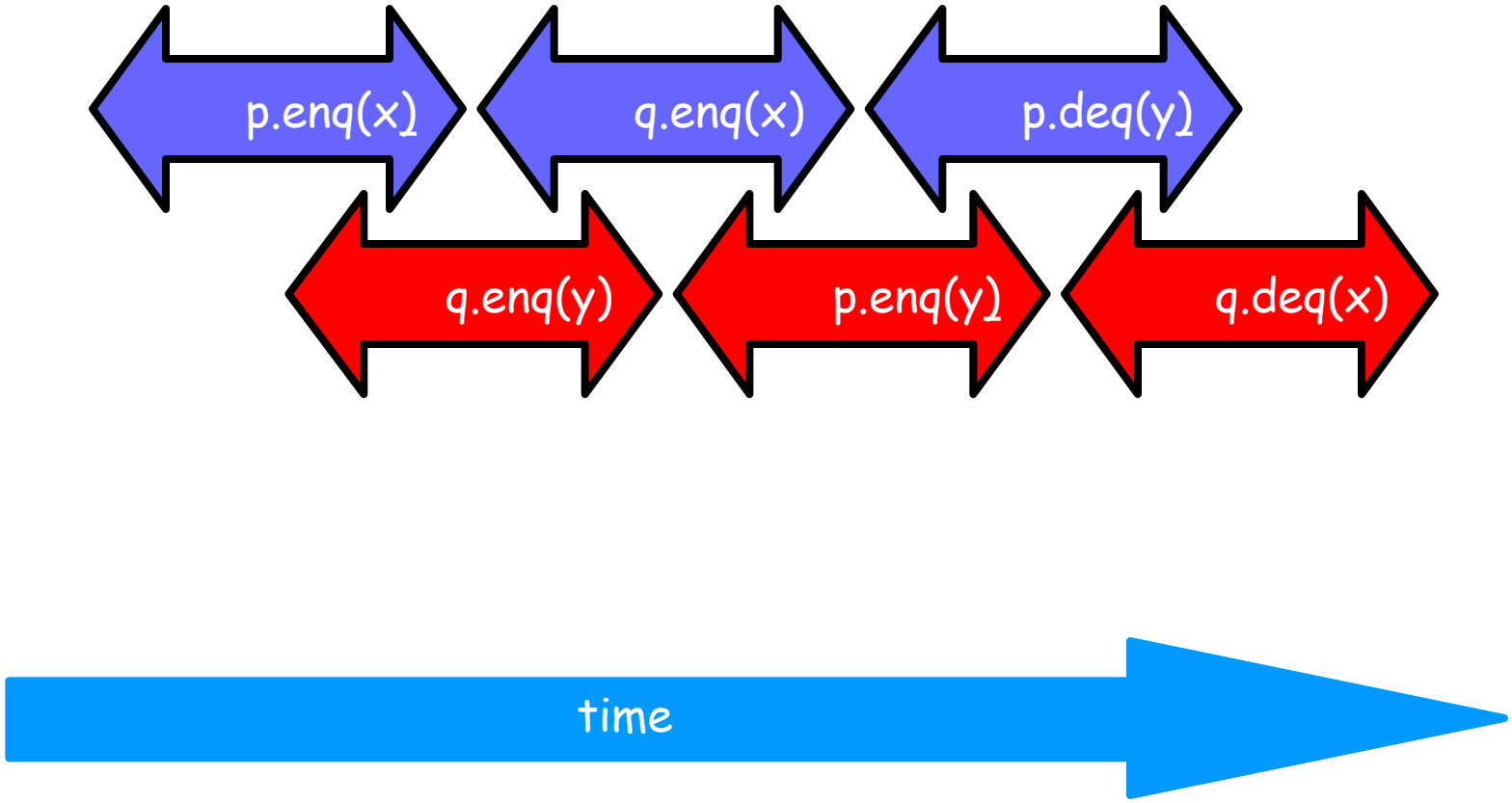
FIFO Queue Example



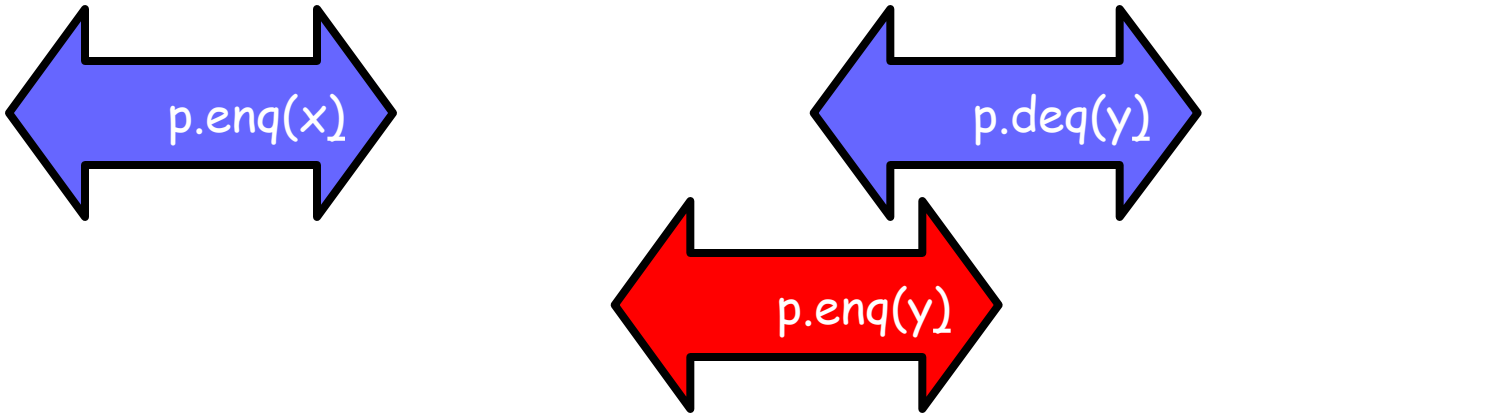
FIFO Queue Example



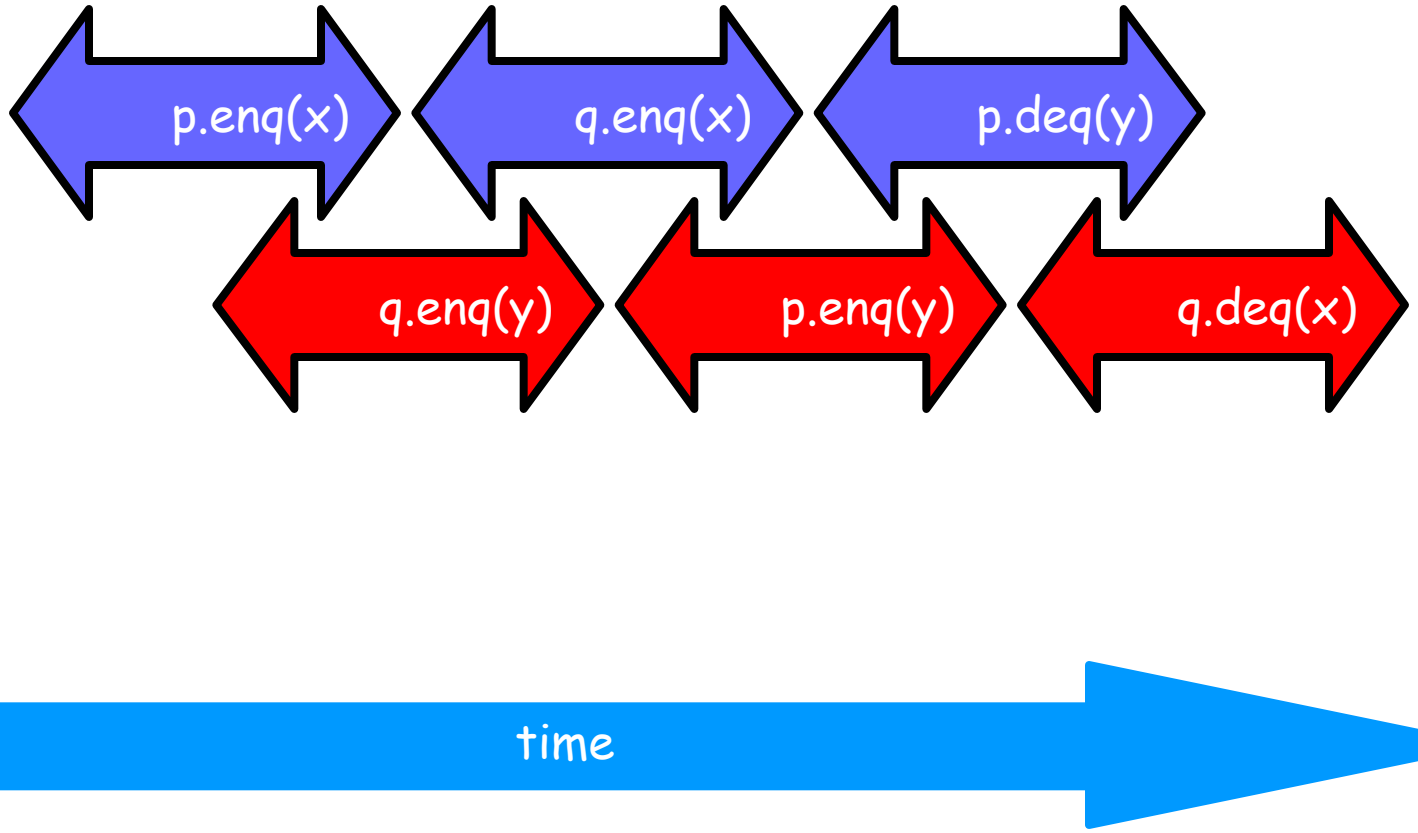
H|p Sequentially Consistent



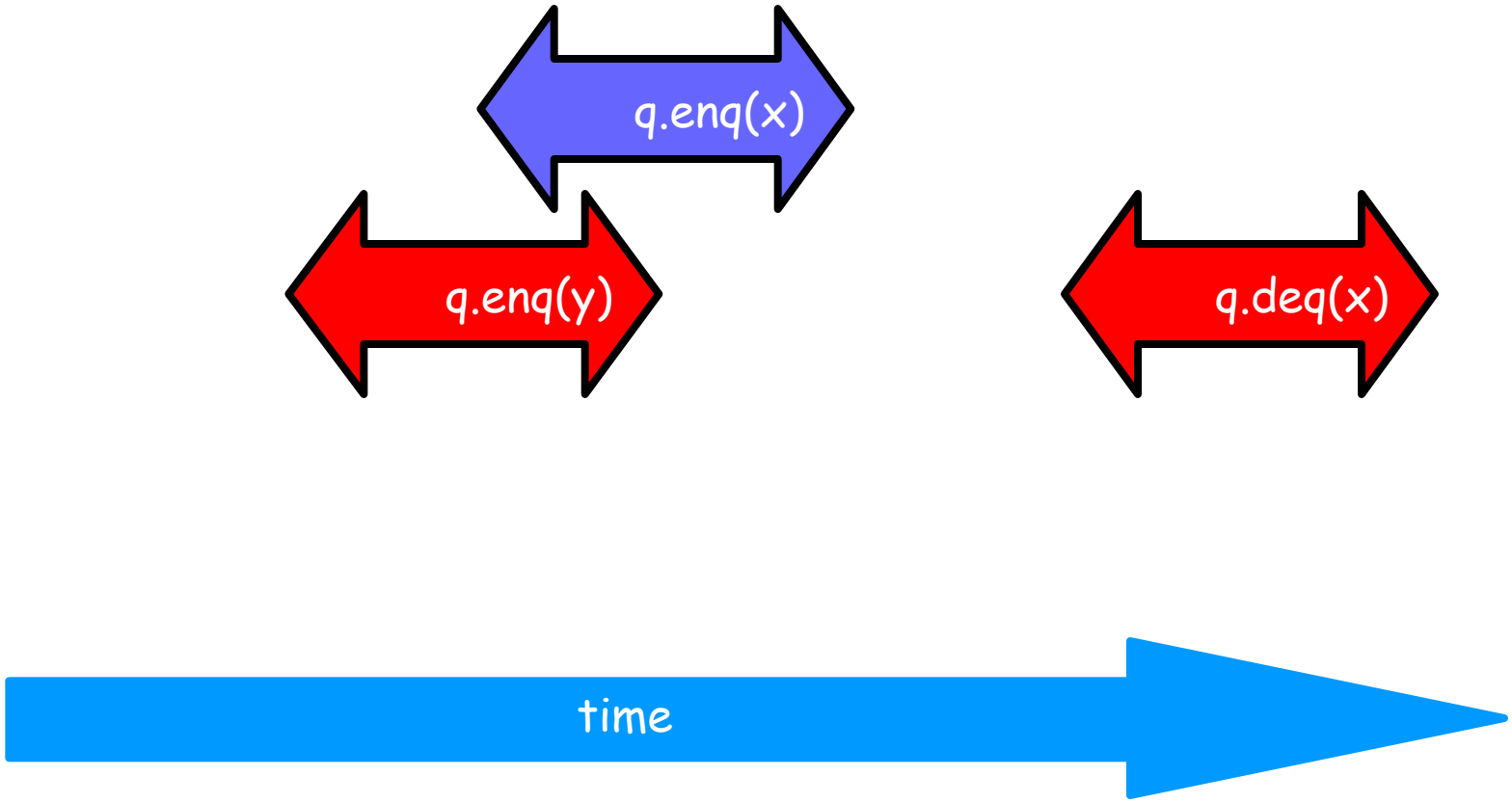
H|p Sequentially Consistent



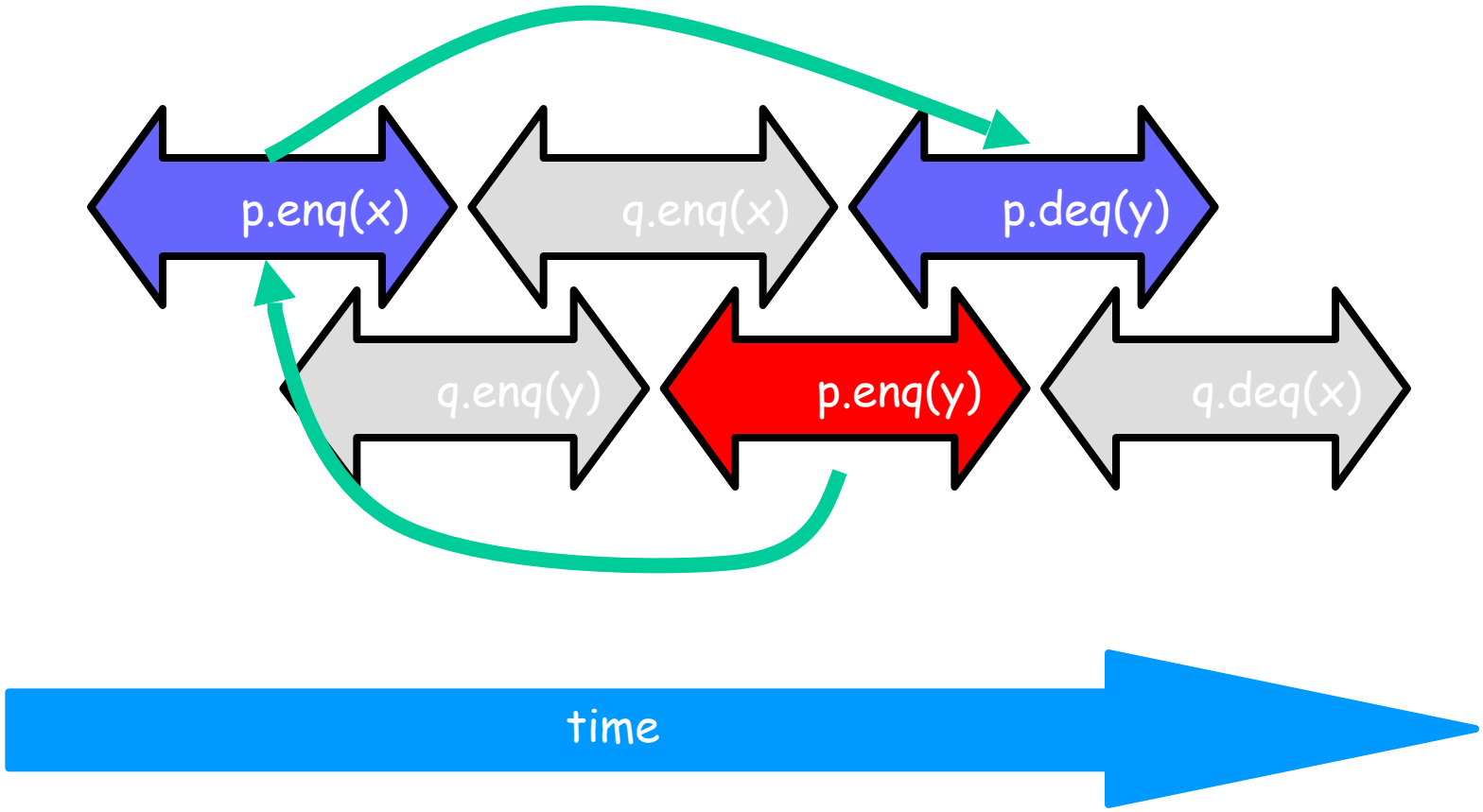
H|q Sequentially Consistent



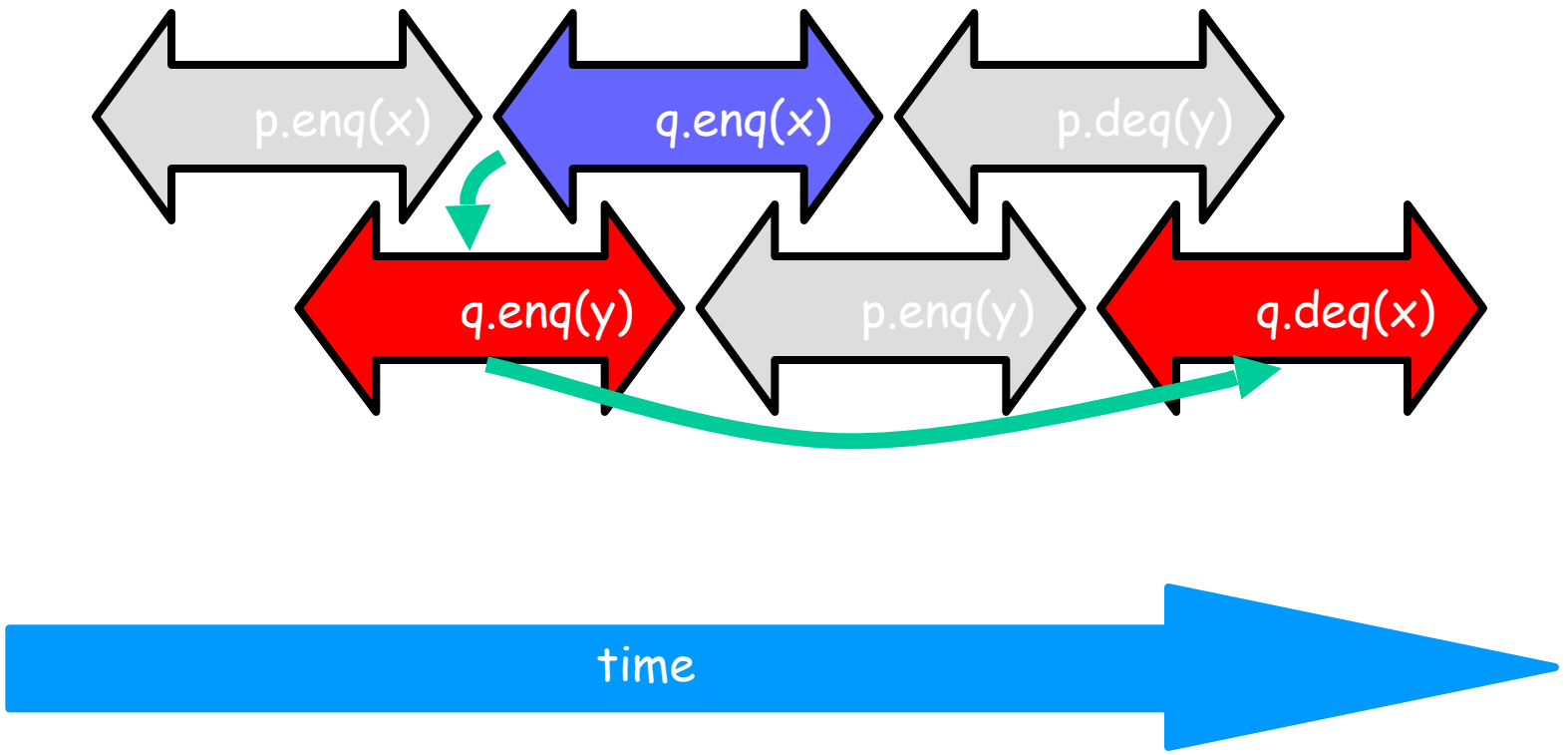
H|q Sequentially Consistent



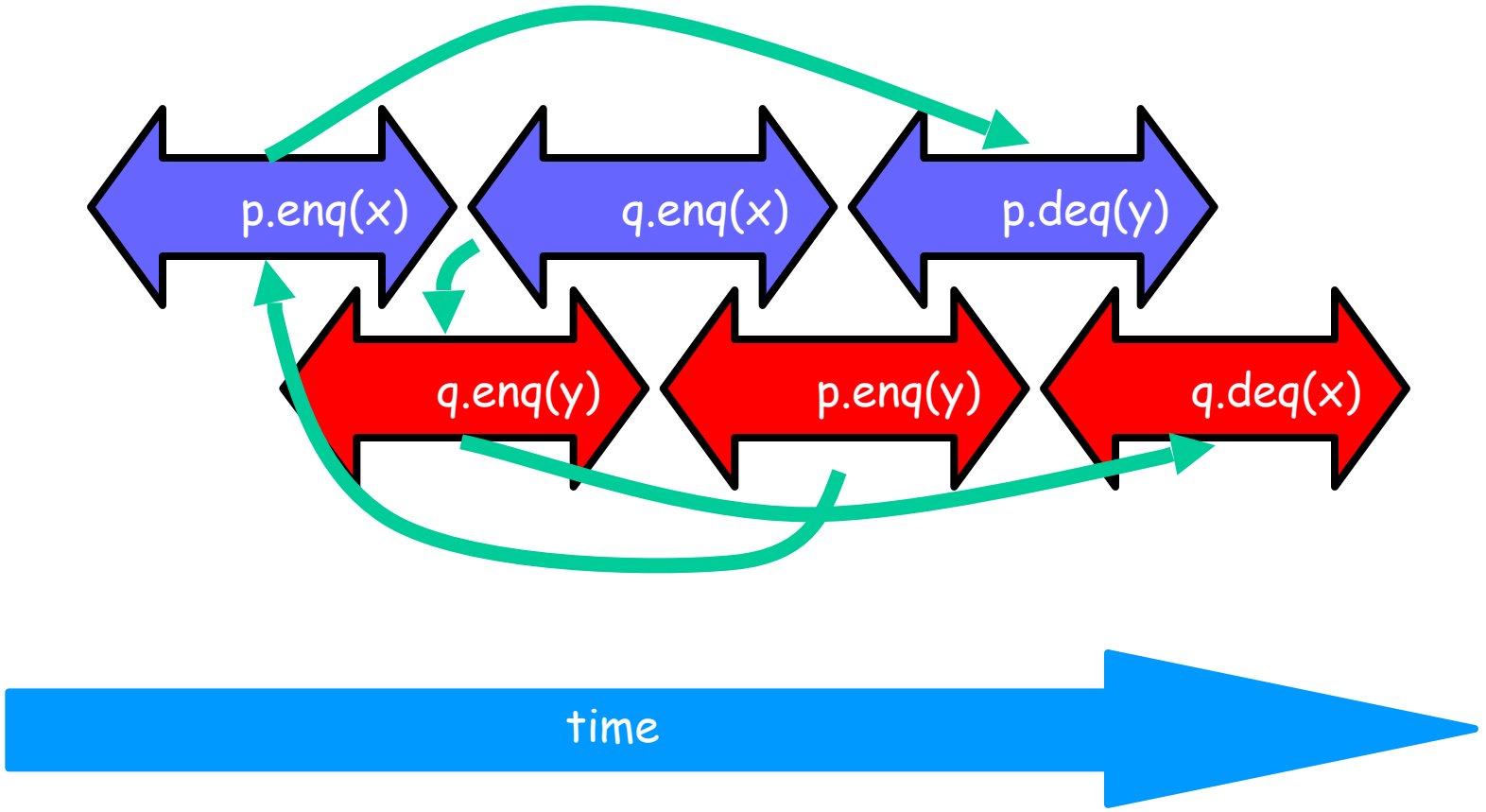
Ordering imposed by p



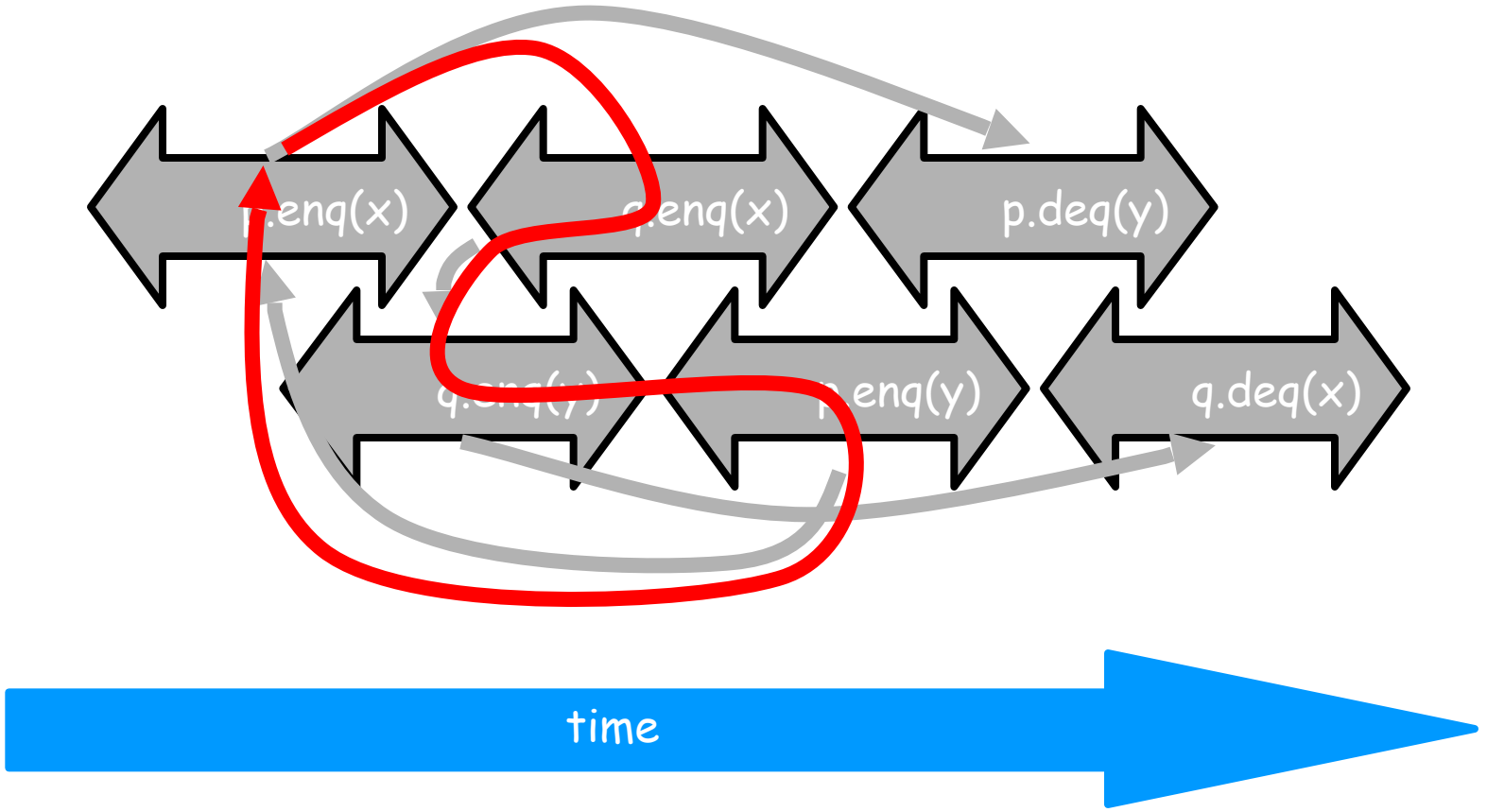
Ordering imposed by q



Ordering imposed by both



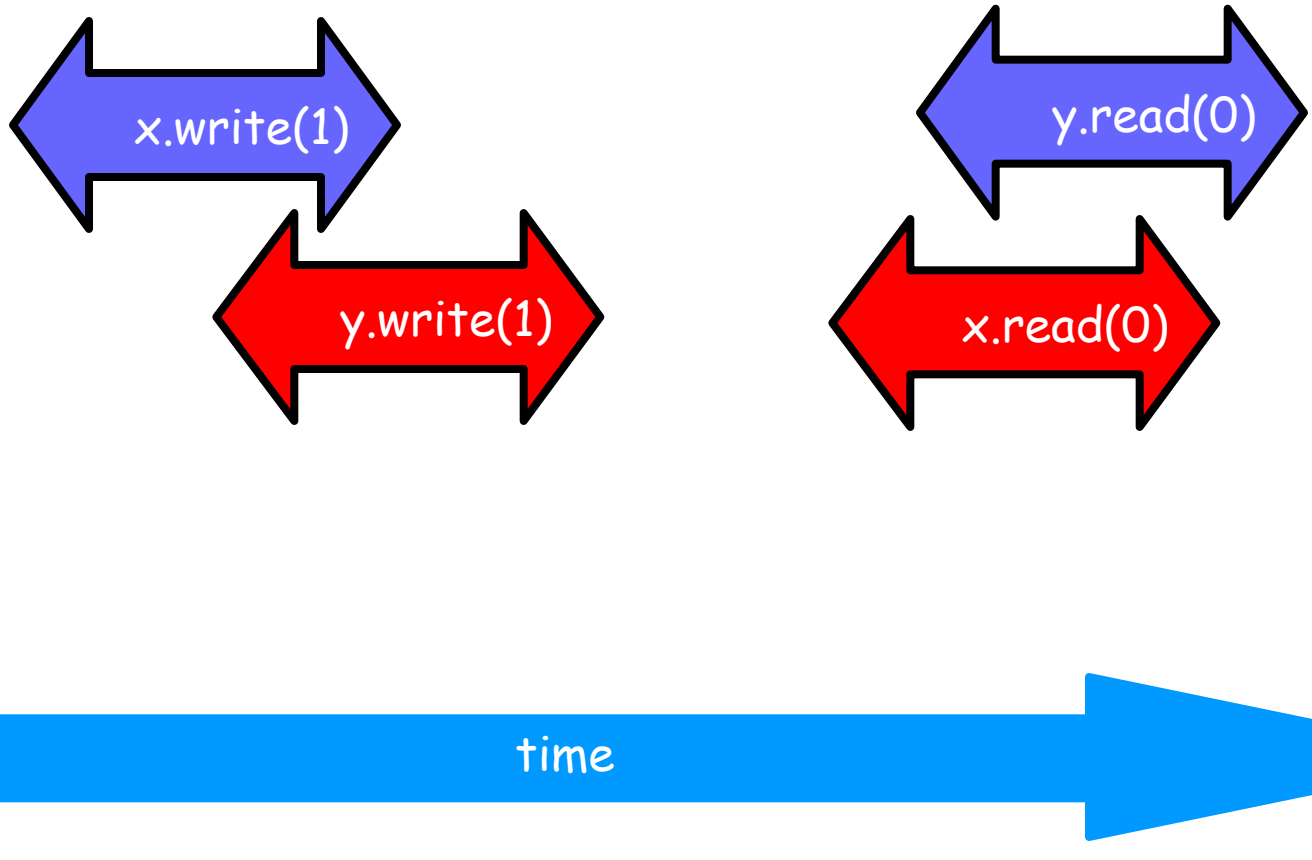
Combining orders



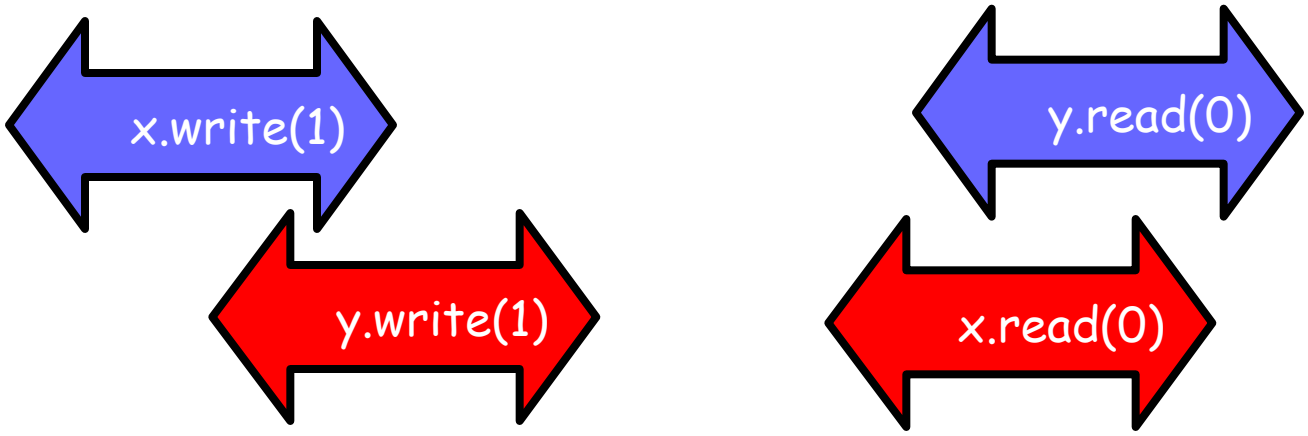
Fact

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

The Flag Example

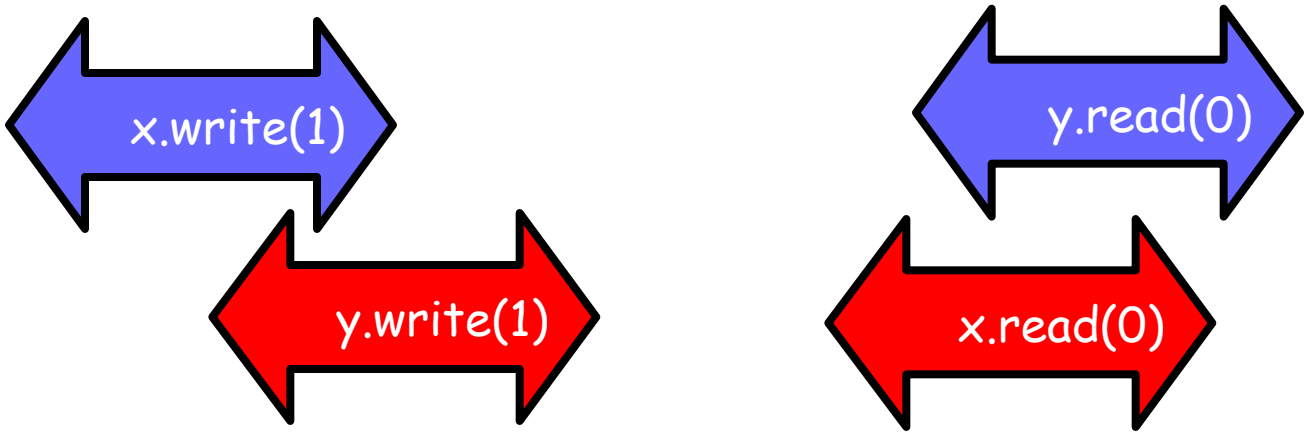


The Flag Example



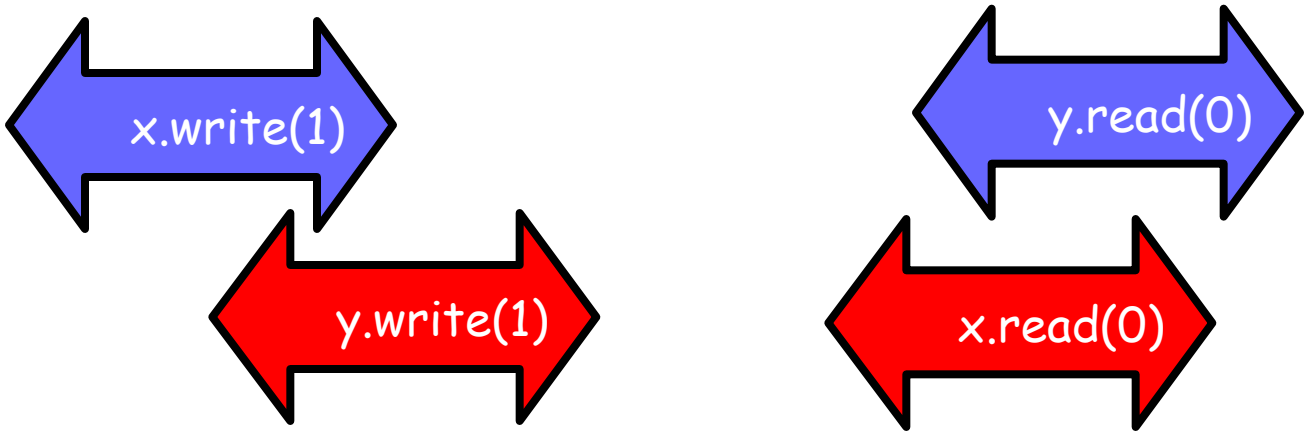
- Each thread's view is sequentially consistent
  - It went first

The Flag Example



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

The Flag Example



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

Opinion1: It's Wrong

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

Opinion2: But It Feels So Right …

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

Memory Hierarchy

- On modern multiprocessors, processors do not read and write directly to memory.
- Memory accesses are very slow compared to processor speeds,
- Instead, each processor reads and writes directly to a cache

Memory Operations

- To read a memory location,
  - load data into cache.
- To write a memory location
  - update cached copy,
  - Lazily write cached data back to memory

While Writing to Memory

- A processor can execute hundreds, or even thousands of instructions
- Why delay on every memory write?
- Instead, write back in parallel with rest of the program.

Revisionist History

- Flag violation history is actually OK
  - processors delay writing to memory
  - Until after reads have been issued.
- Otherwise unacceptable delay between read and write instructions.
- Who knew you wanted to synchronize?

Who knew you wanted to synchronize?

- Writing to memory = mailing a letter
- Vast majority of reads & writes
  - Not for synchronization
  - No need to idle waiting for post office
- If you want to synchronize
  - Announce it explicitly
  - Pay for it only when you need it

Explicit Synchronization

- Memory barrier instruction
  - Flush unwritten caches
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections
- Expensive

Volatile

- In Java, can ask compiler to keep a variable up-to-date with volatile keyword
- Also inhibits reordering, removing from loops, & other "optimizations"

Real-World Hardware Memory

- Weaker than sequential consistency
- But you can get sequential consistency at a price
- OK for expert, tricky stuff
  - assembly language, device drivers, etc.
- Linearizability more appropriate for high-level software

Critical Sections

- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Problems
  - Blocking
  - No concurrency

Linearizability

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

Correctness: Linearizability

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

Progress

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

Progress Conditions

- Deadlock-free: some thread trying to acquire the lock eventually succeeds.
- Starvation-free: every thread trying to acquire the lock eventually succeeds.
- Lock-free: some thread calling a method eventually returns.
- Wait-free: every thread calling a method eventually returns.

Progress Conditions

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

Summary

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

Foundations of Shared Memory

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Last Lecture

- 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

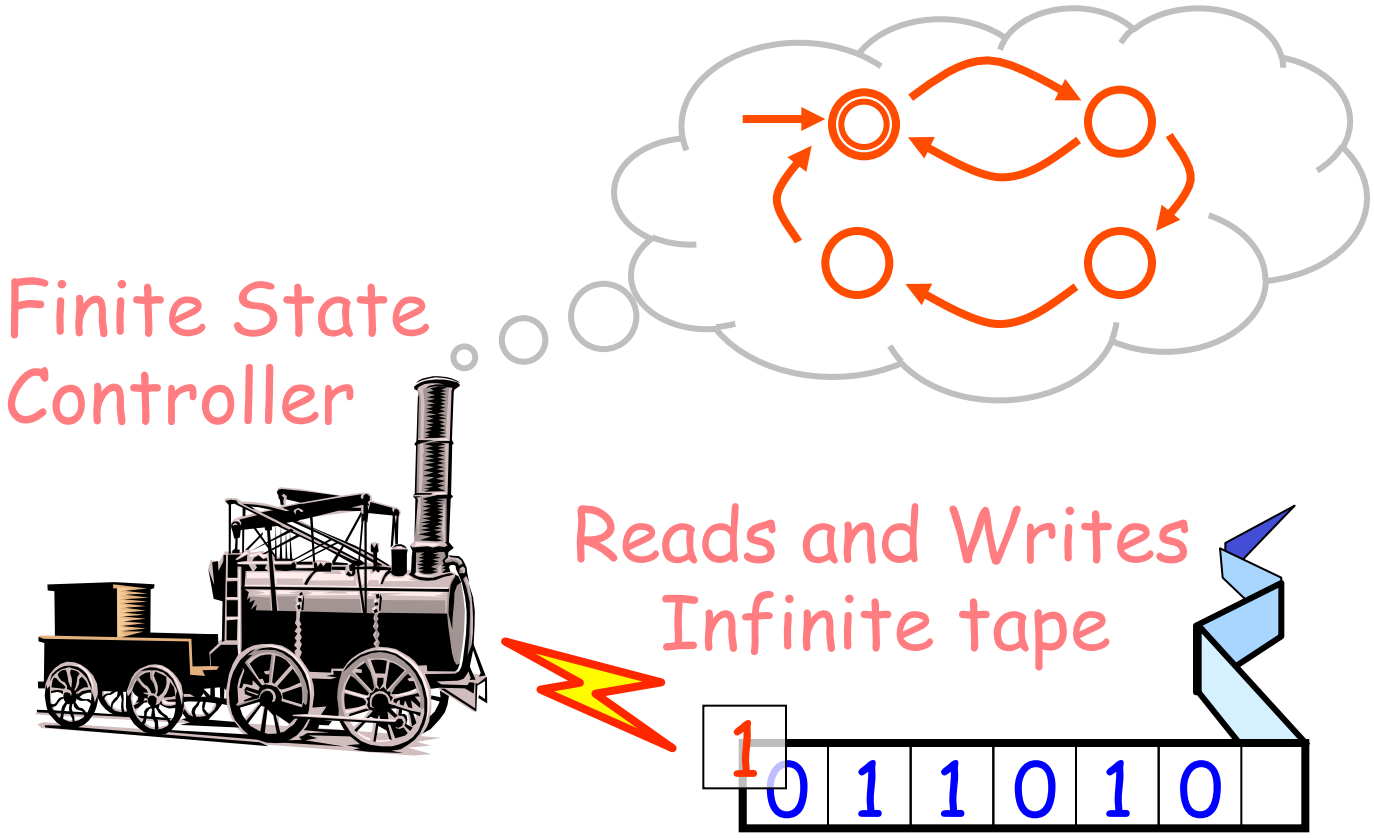
Fundamentals

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

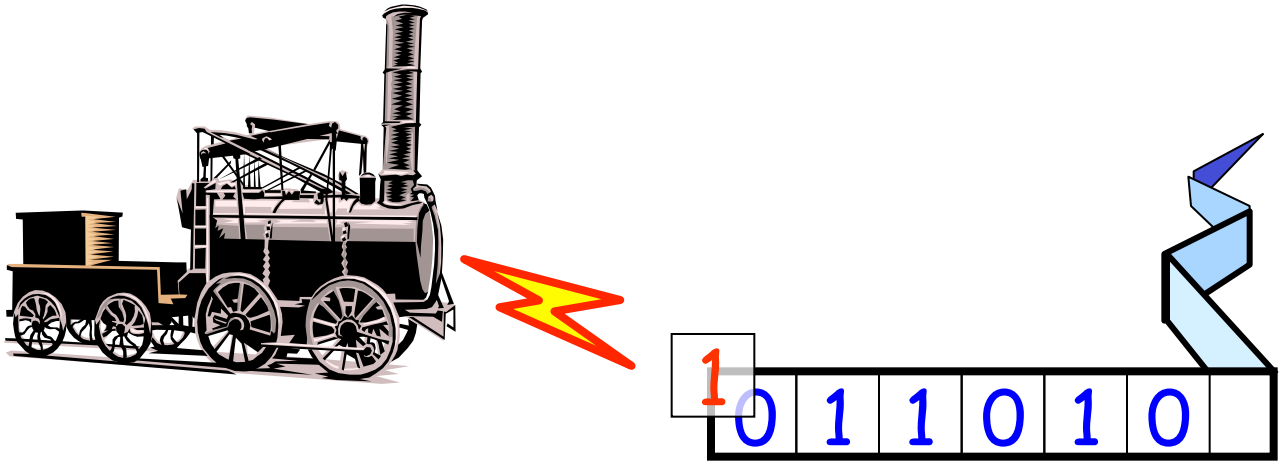
Alan Turing

- Helped us understand what is and is not computable on a sequential machine.
- Still best model available

Turing Machine



Turing Computability



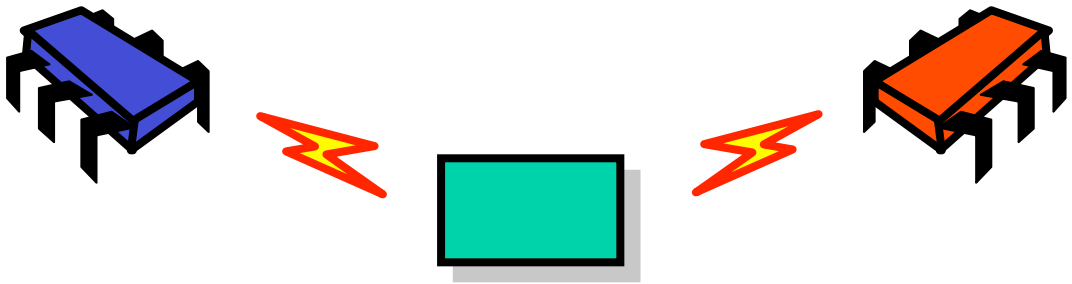
- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Shared-Memory Computability?

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

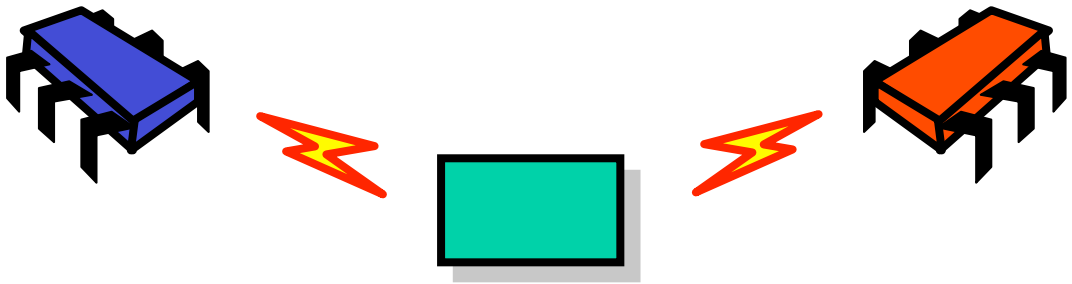
Foundations of Shared Memory

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



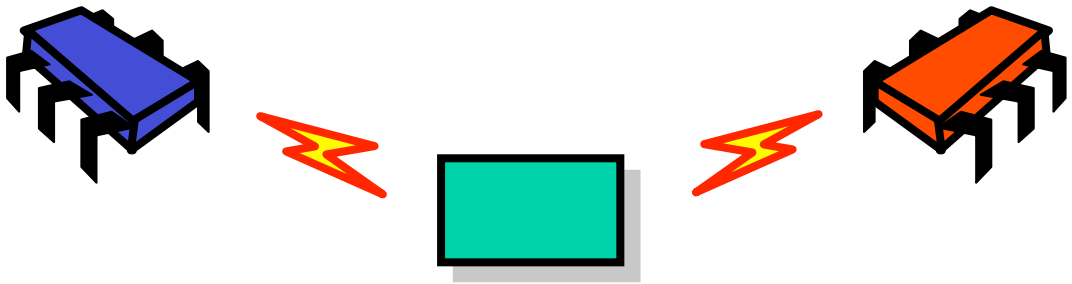
Foundations of Shared Memory

What is the weakest useful form of shared memory?



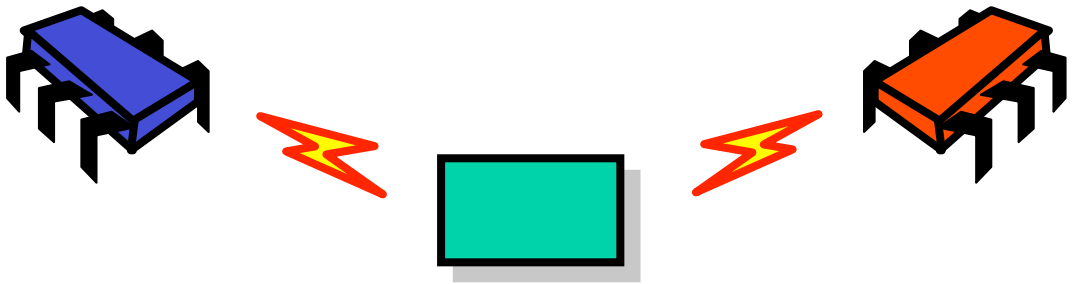
Foundations of Shared Memory

What can it do?

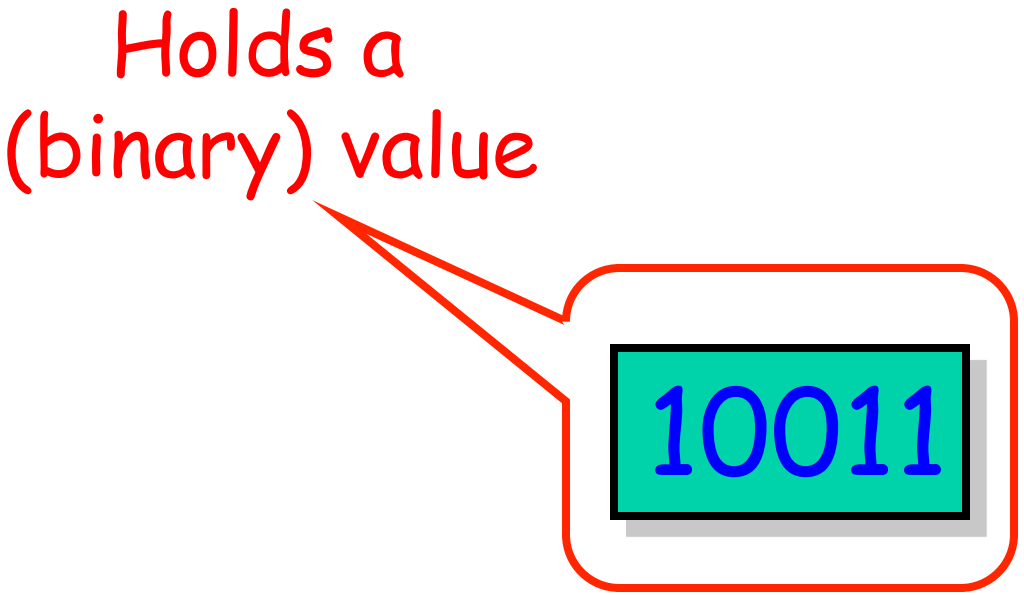


Foundations of Shared Memory

What can't it do?

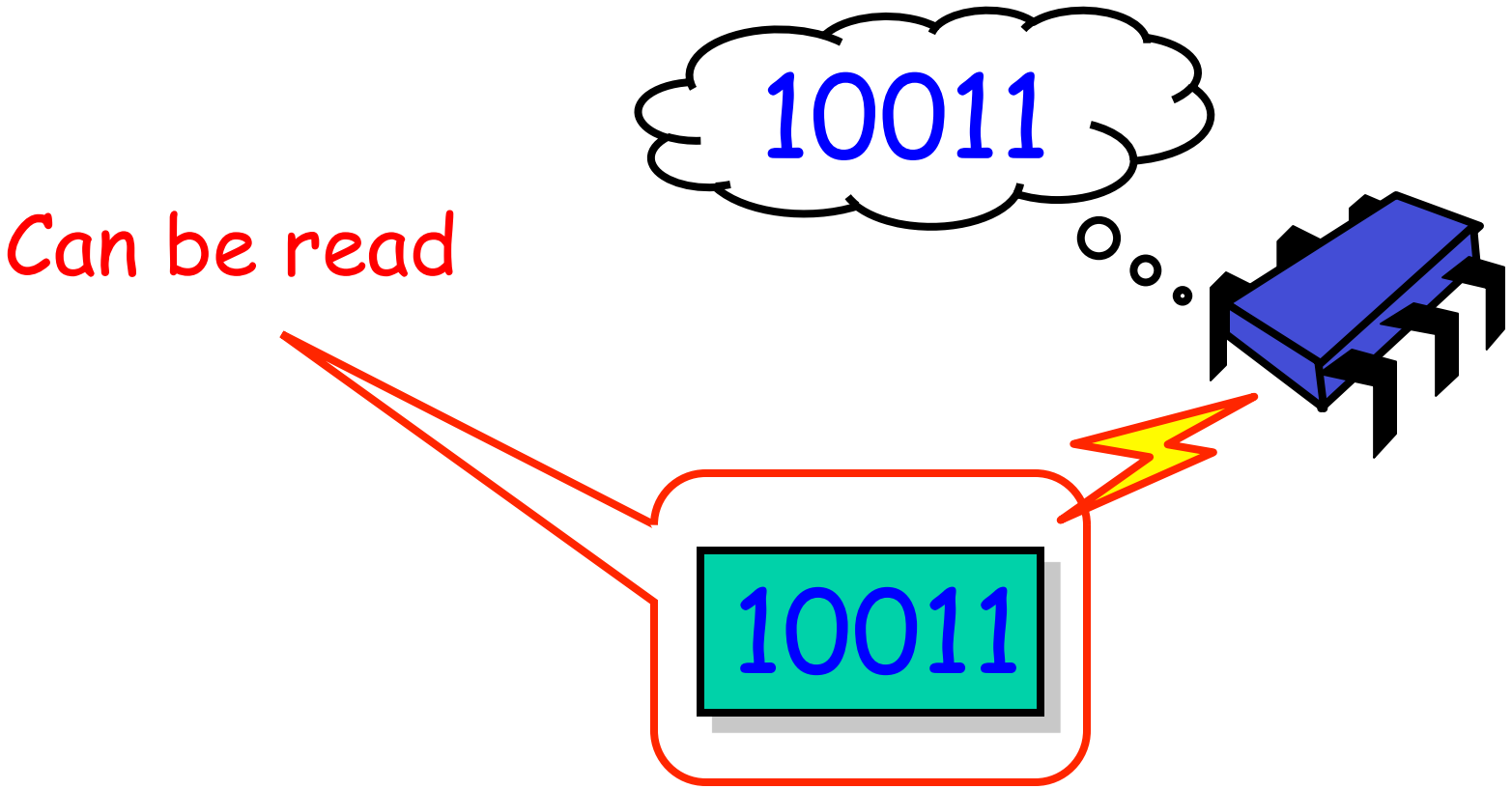


Register *

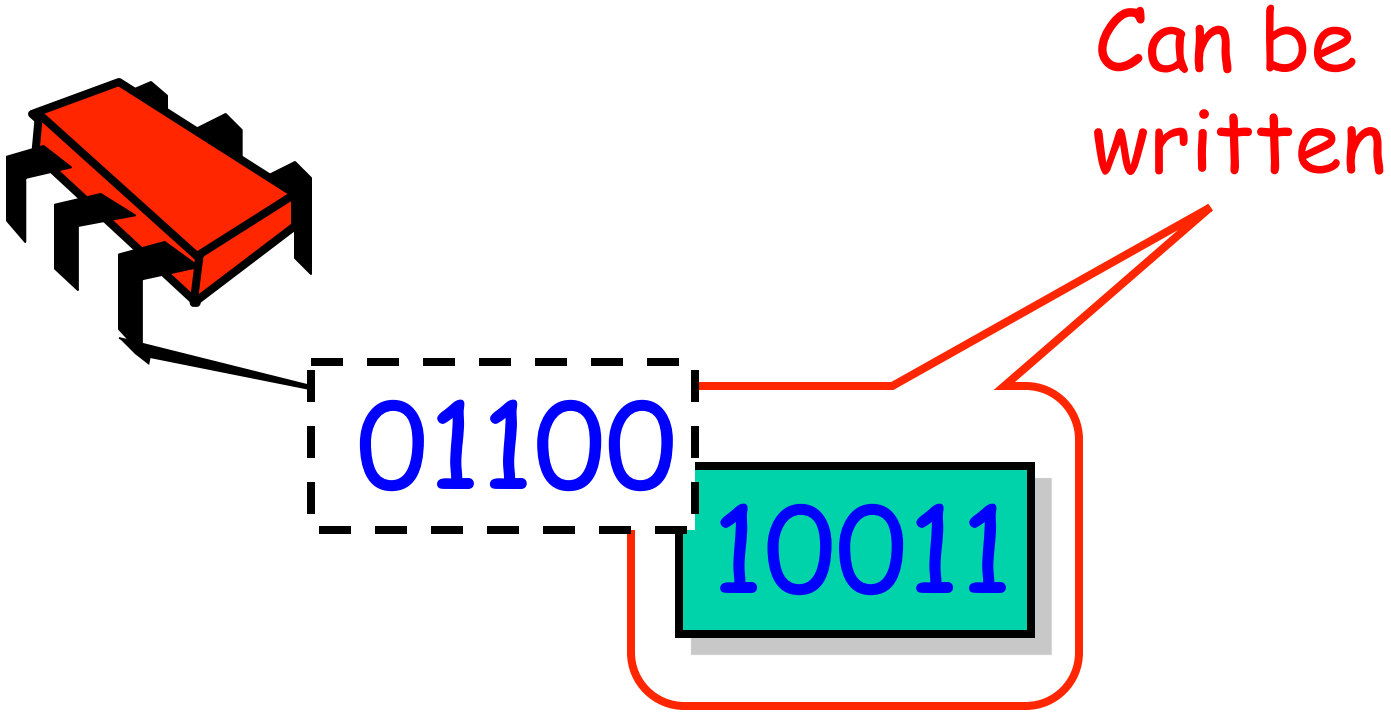


* A memory location: name is historical

Register



Register



Registers

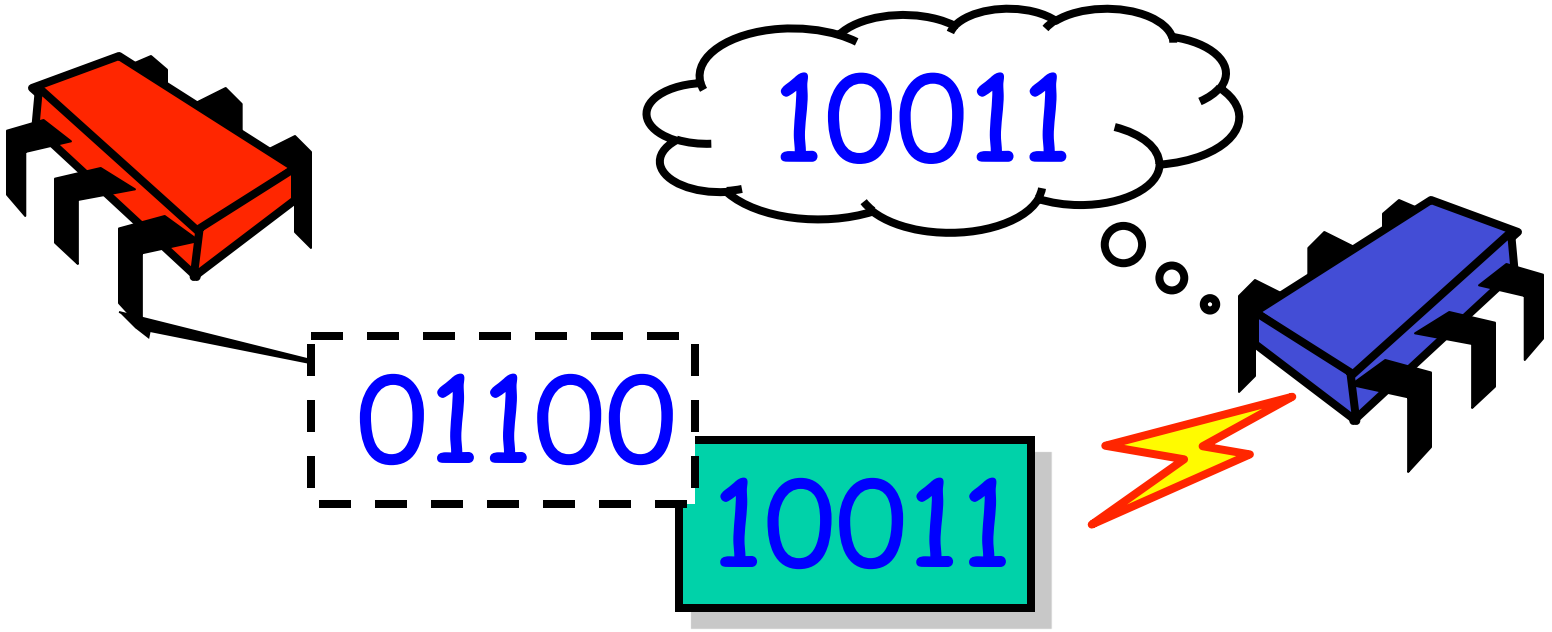
```
public interface Register<T> {
   public T read();
   public void write(T v);
}
```

Registers

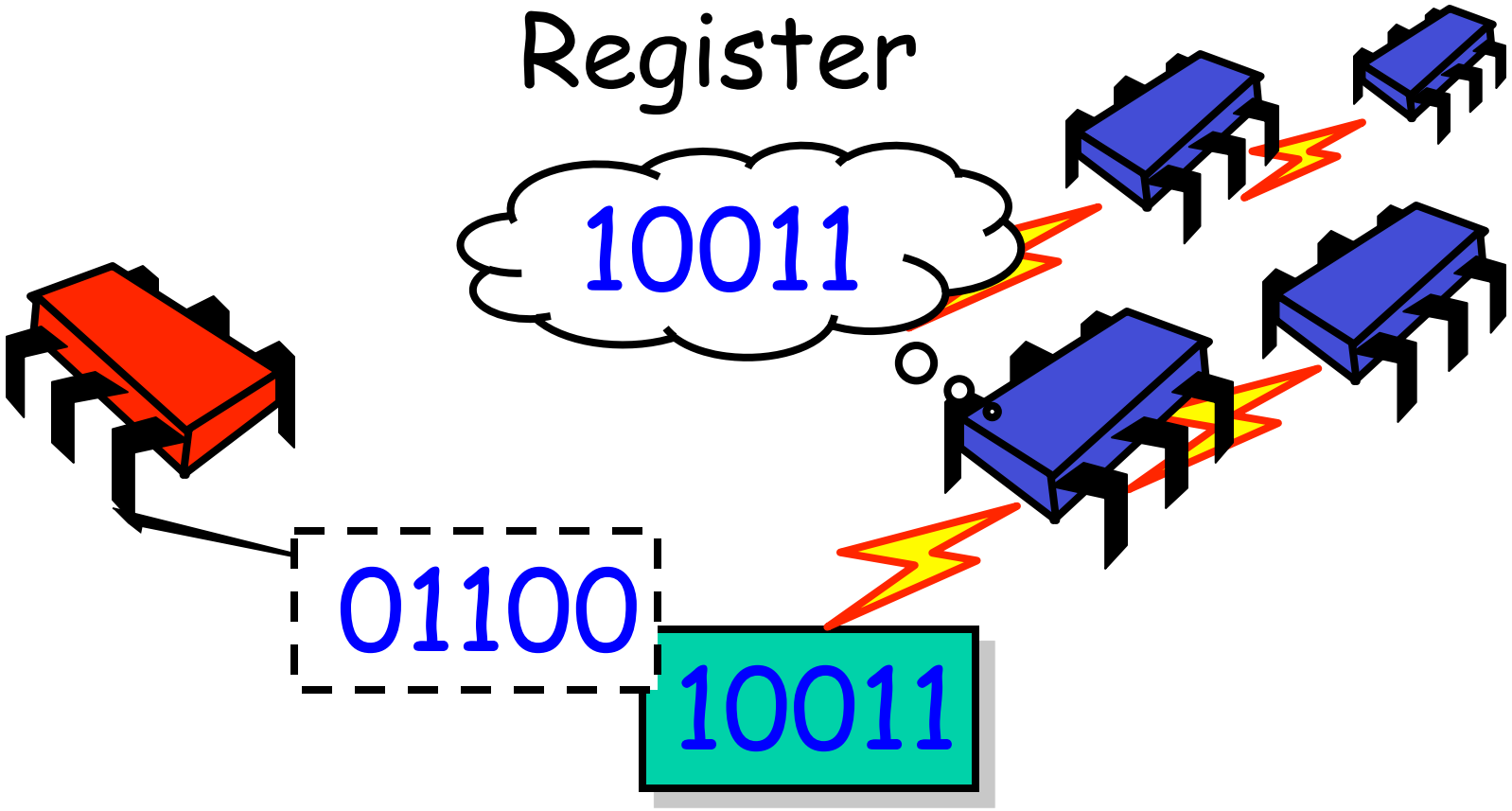
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

Type of register (usually Boolean or m-bit Integer)

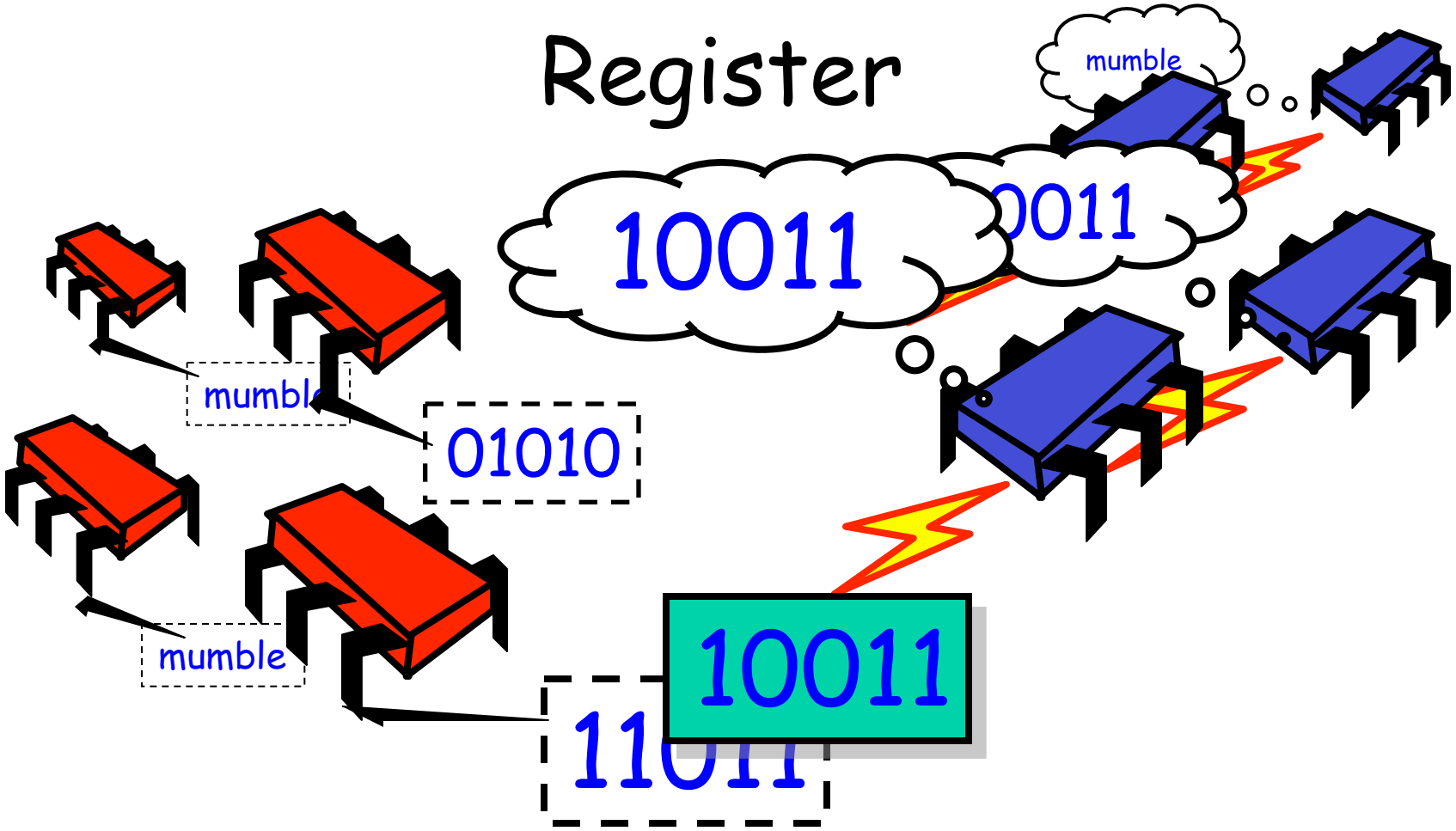
Single-Reader/Single-Writer Register



Multi-Reader/Single-Writer Register



Multi-Reader/Multi-Writer Register

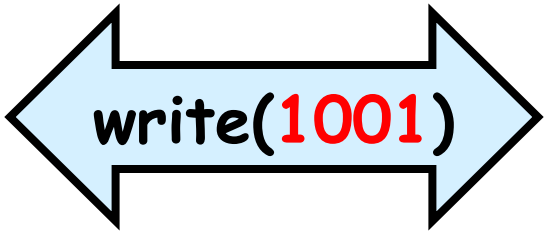


Jargon Watch

- SRSW
  - Single-reader single-writer
- MRSW
  - Multi-reader single-writer
- MRMW
  - Multi-reader multi-writer

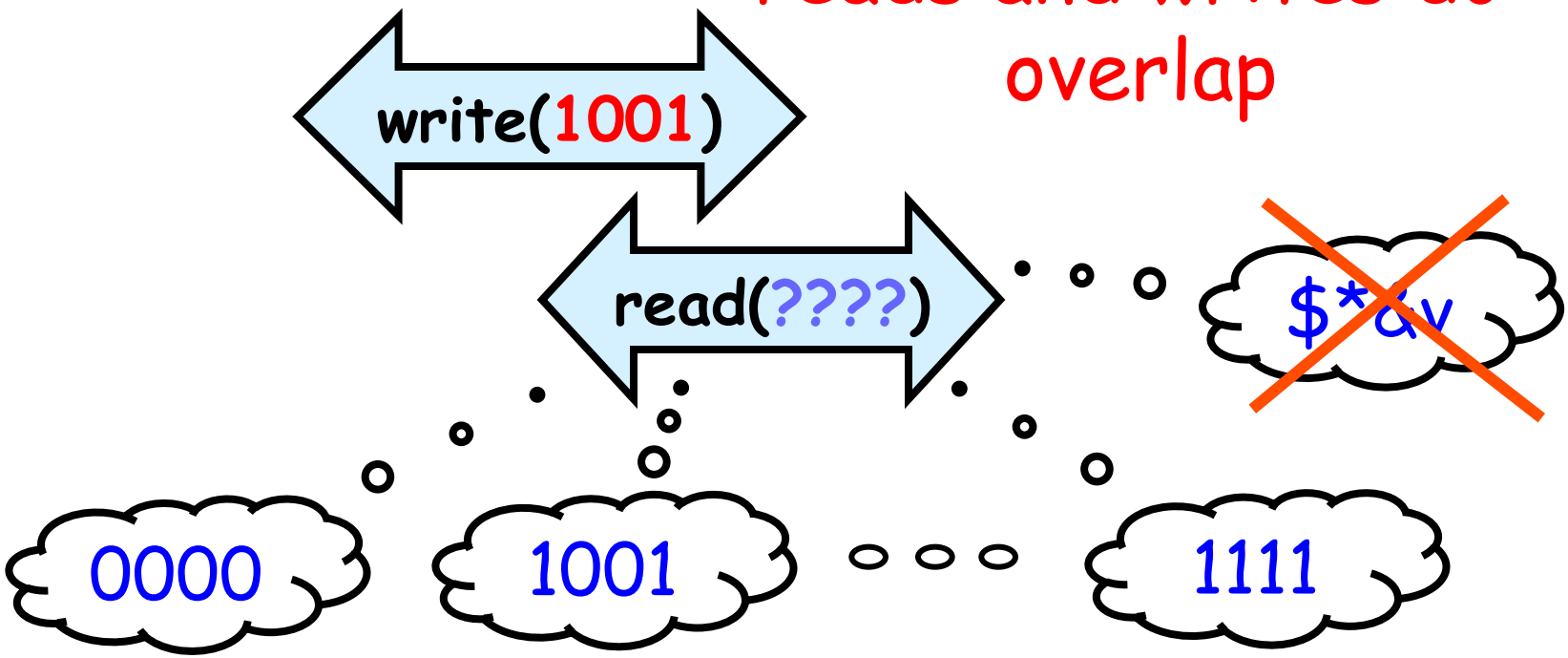
Safe Register

OK if reads and writes don't overlap

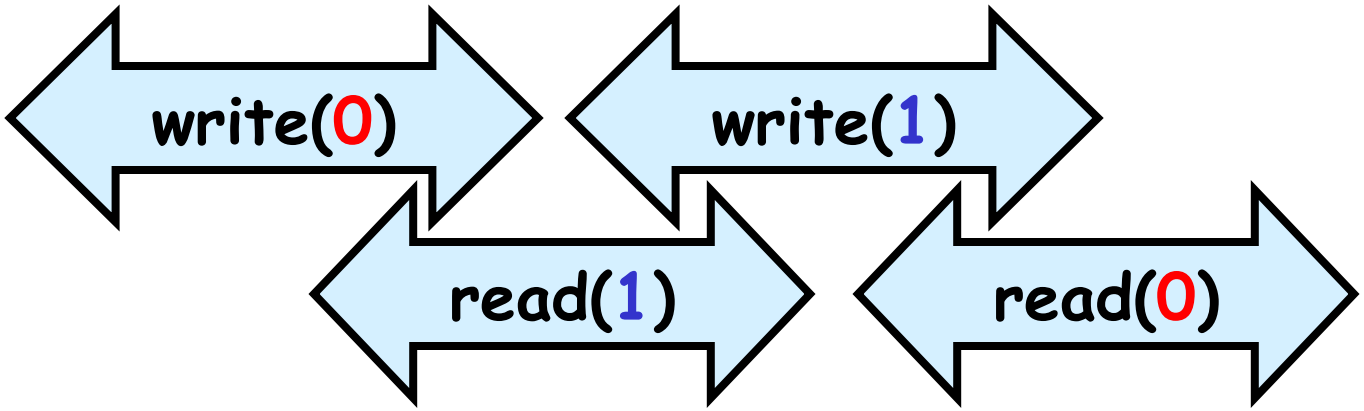


Safe Register

Some valid value if reads and writes do overlap

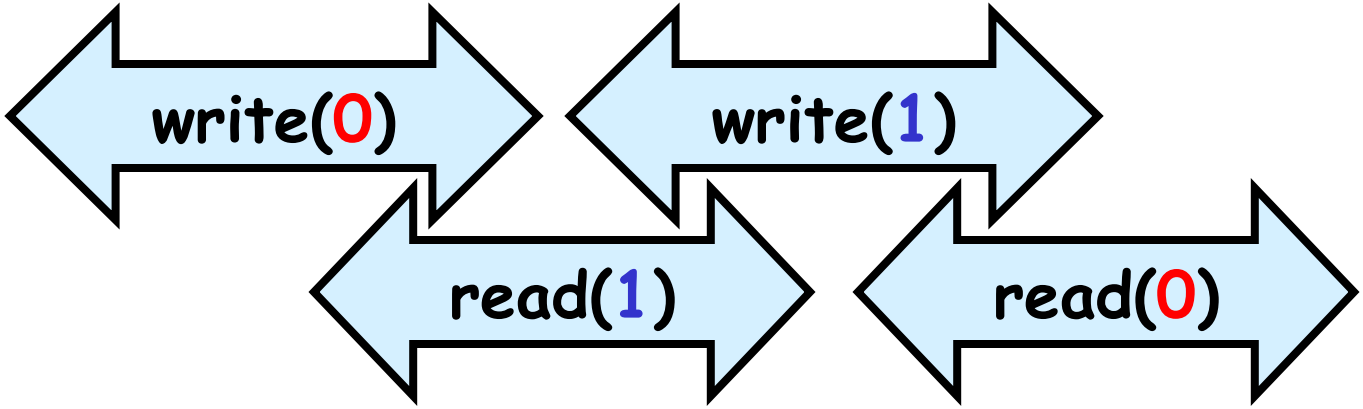


Regular Register

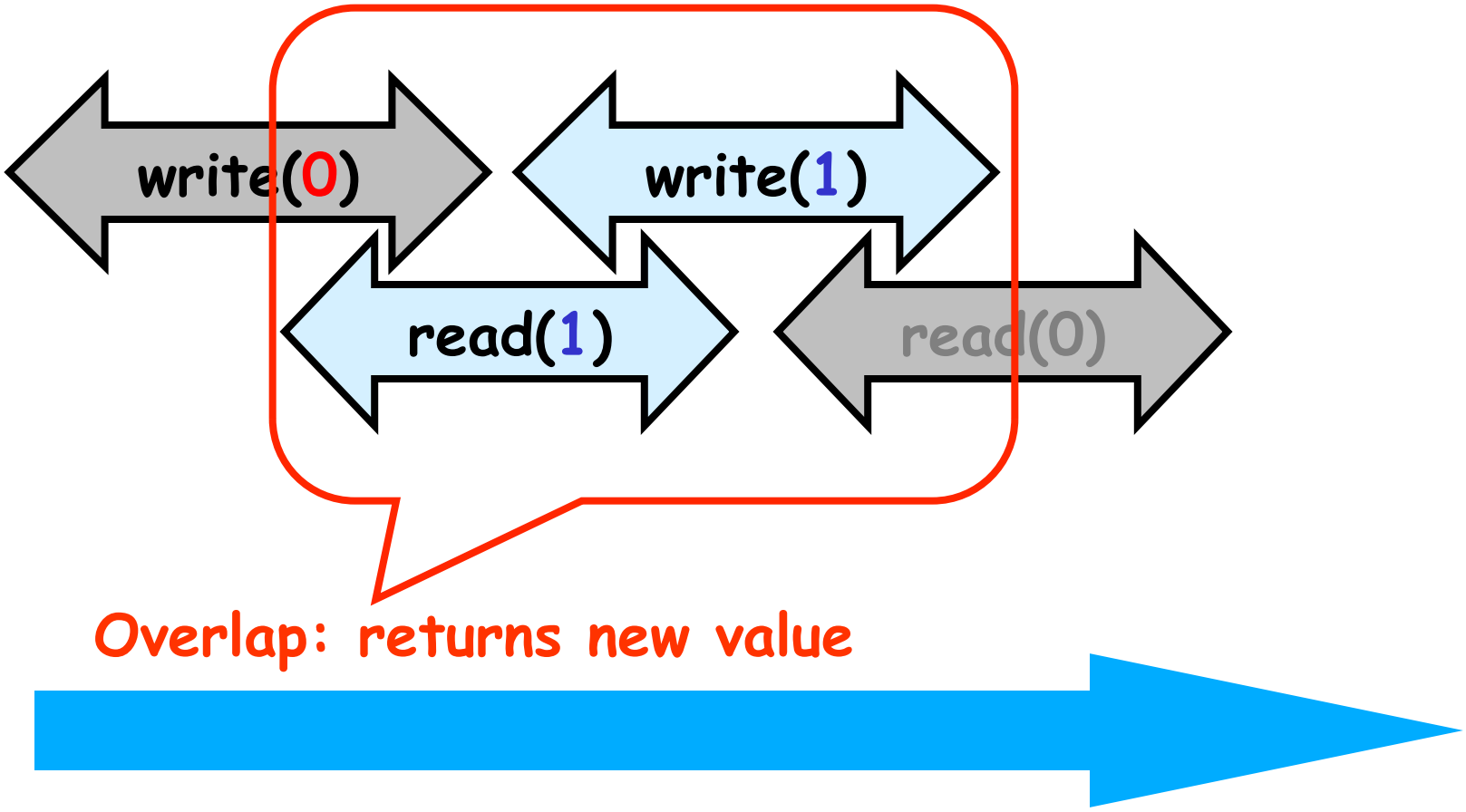


- Single Writer
- Readers return:
  - Old value if no overlap (safe)
  - Old or one of new values if overlap

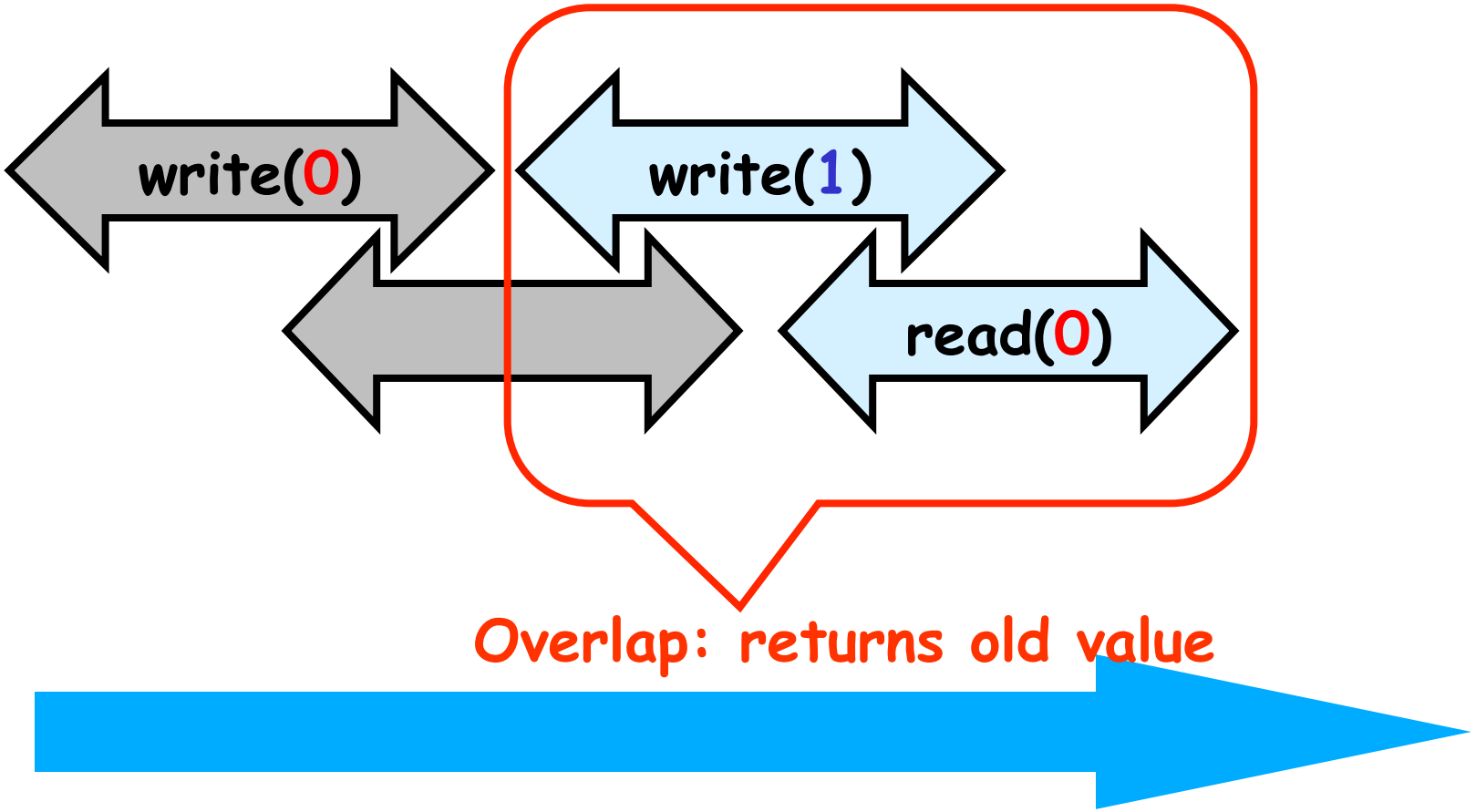
Regular or Not?



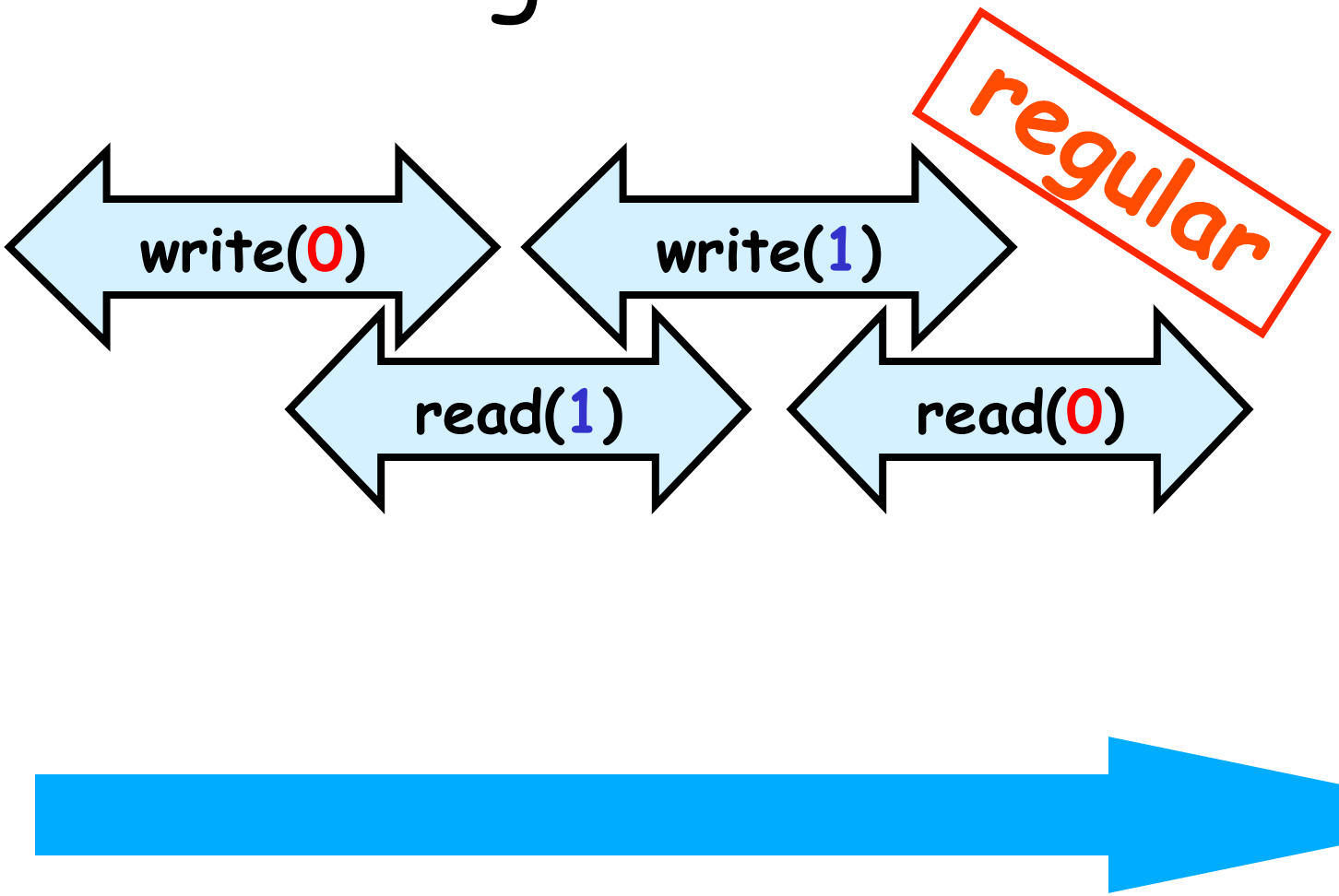
Regular or Not?



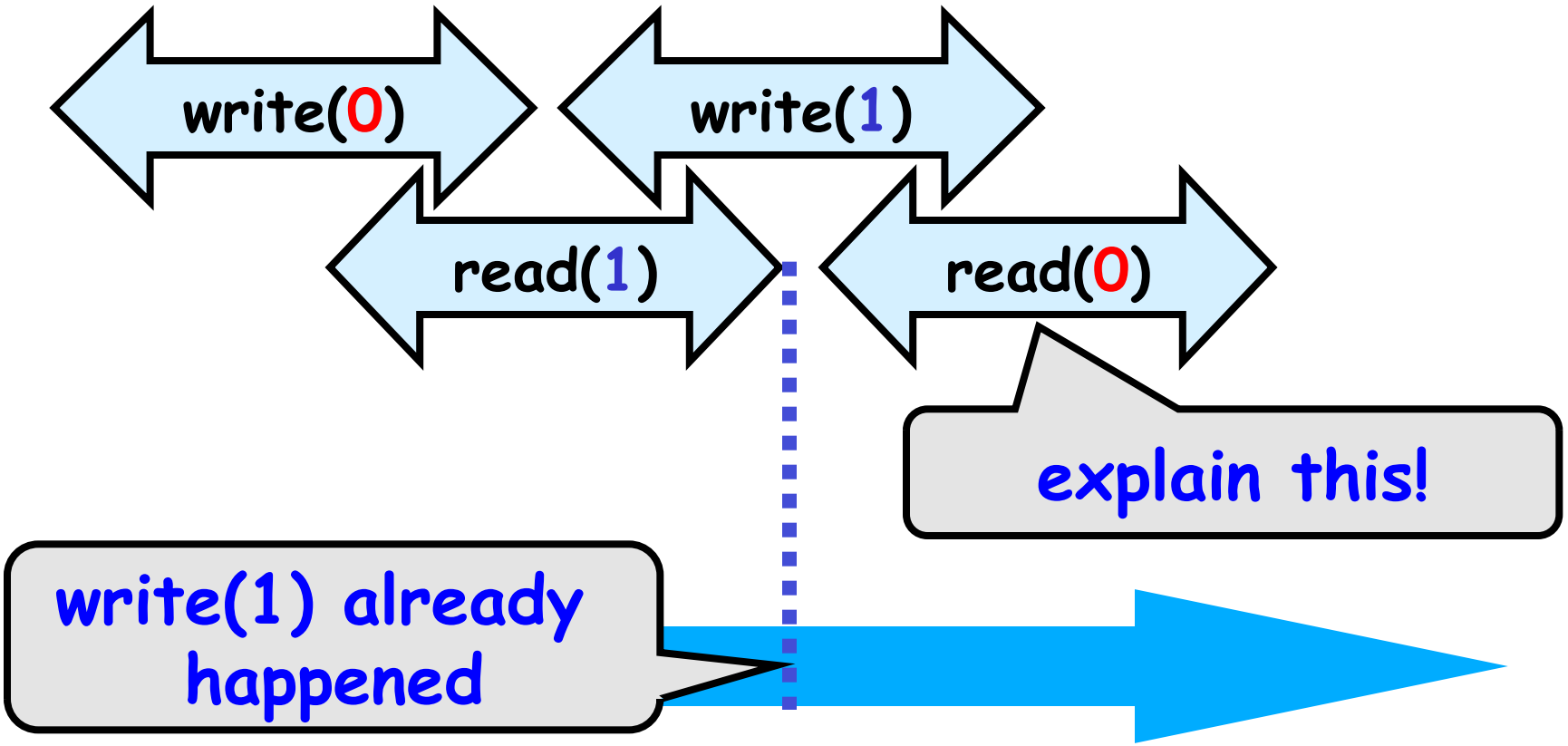
Regular or Not?



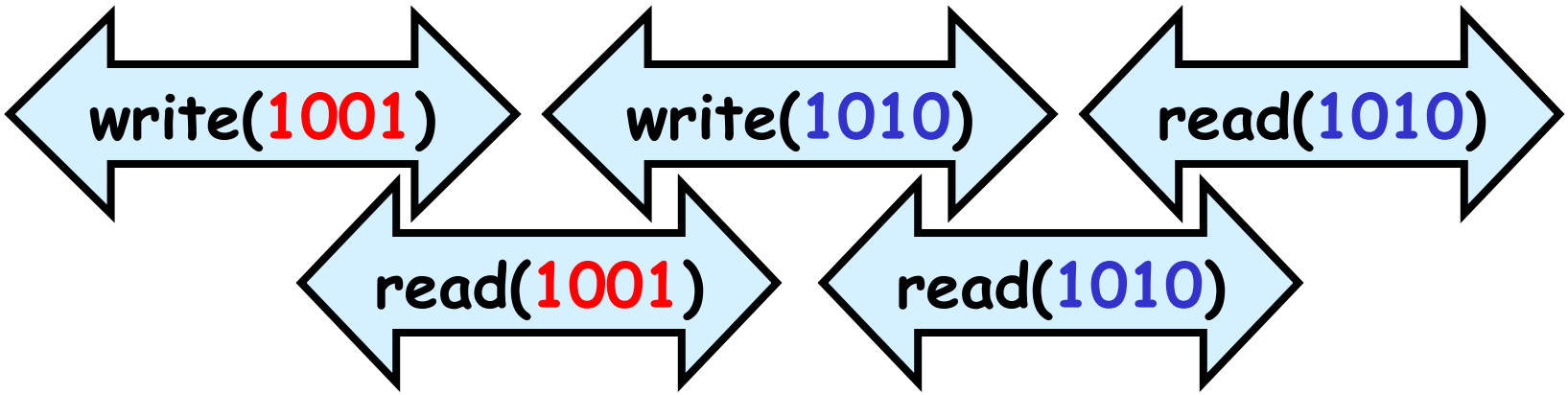
Regular or Not?



Regular ≠ Atomic

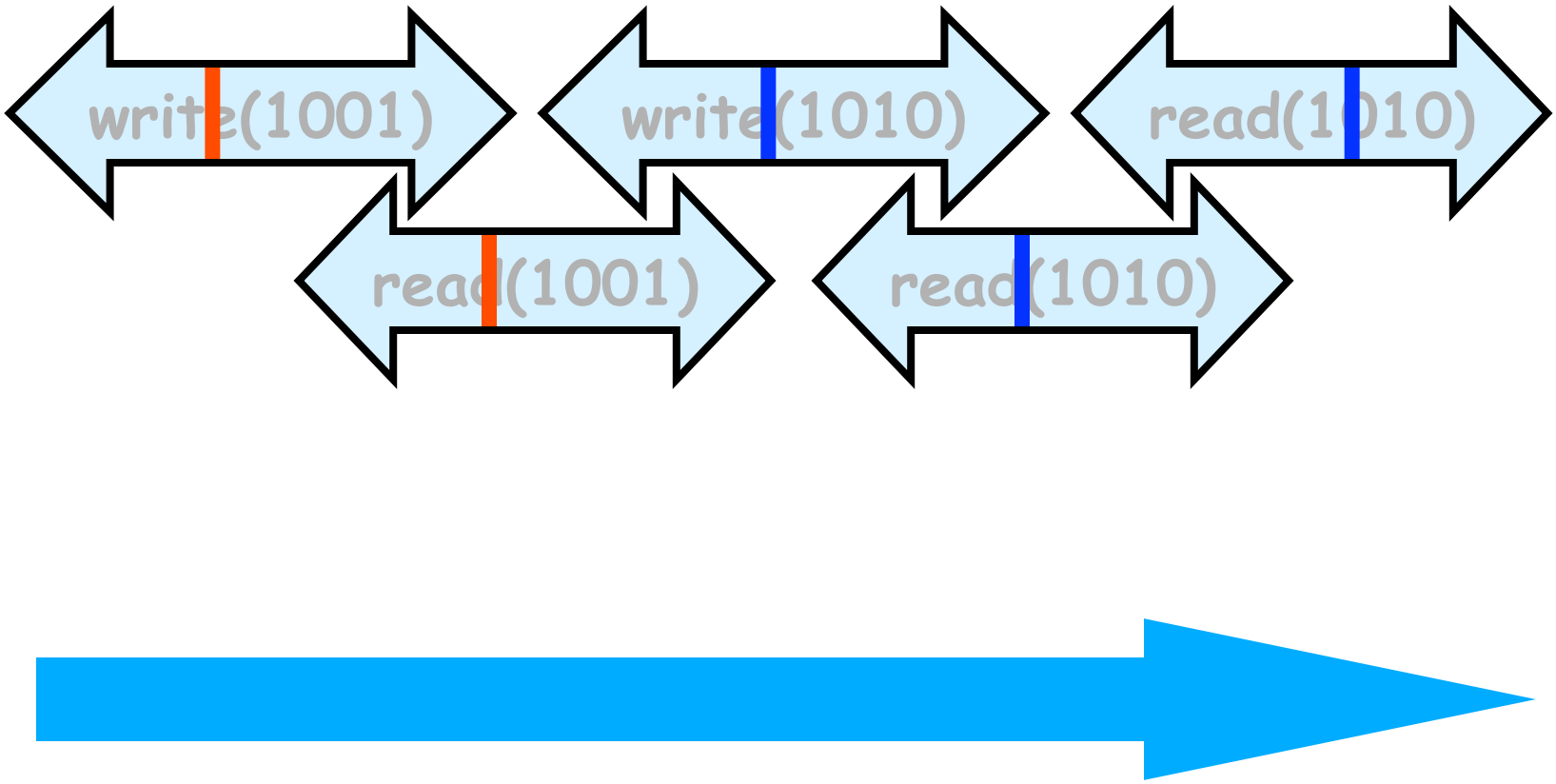


Atomic Register

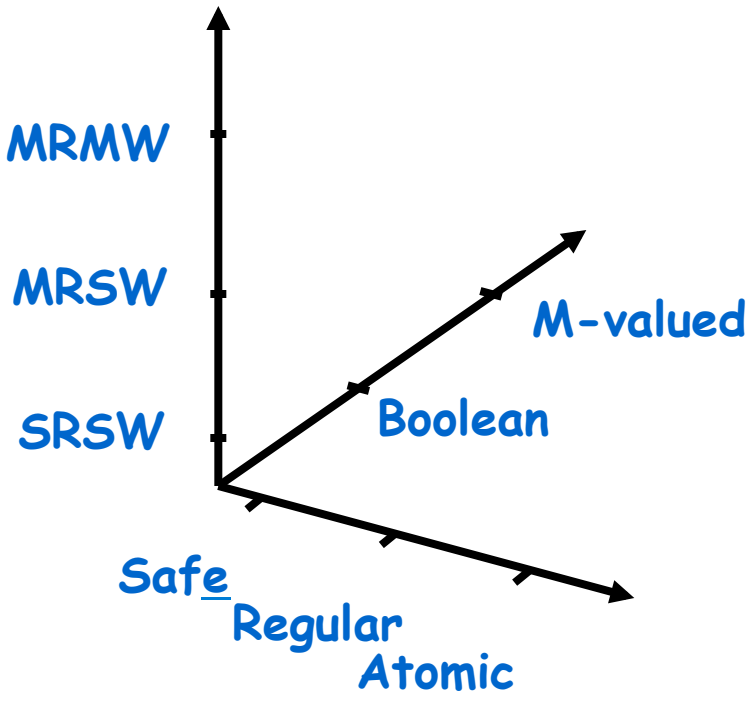


Linearizable to sequential safe register

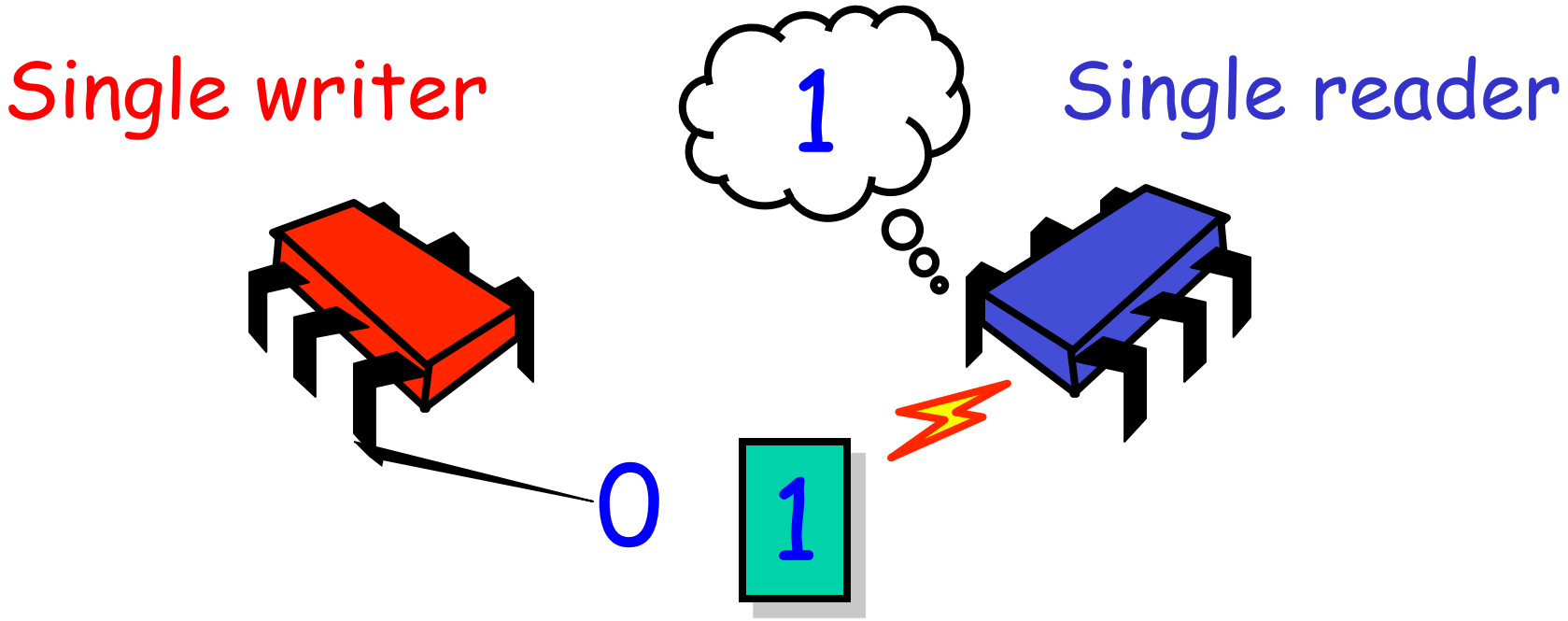
Atomic Register



Register Space



Weakest Register

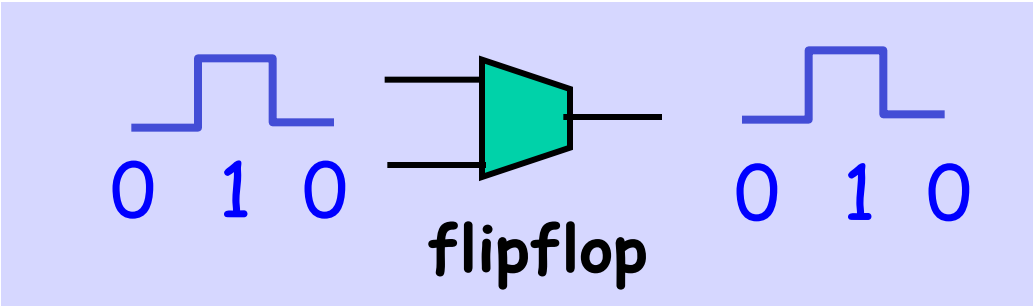


Safe Boolean register

Weakest Register

Single writer

Single reader



Get correct reading if not during state transition

Results

- From SRSW safe Boolean register
  - All the other registers
  - Mutual exclusion
- But not everything!
  - Consensus hierarchy

Foundations of the field

The really cool stuff …

Locking within Registers

- Not interesting to rely on mutual exclusion in register constructions
- We want registers to implement mutual exclusion!
- No fun to use mutual exclusion to implement itself!

Wait-Free Implementations

Definition: An object implementation is ***wait-free*** if every method call completes in a finite number of steps

No mutual exclusion

- Thread could halt in critical section
- Build mutual exclusion from registers

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot