TORSTEN HOEFLER

Parallel Programming
The ABA problem, a bit of Concurrency Theory: Linearizability, Sequential Consistency, Consensus

What puts the super in supercomputer?

The secret behind supercomputing? More of everything.

Speed read

- Supercomputers are amped-up versions of traditional computers
- Parallel computing allows supercomputers to process tasks faster than your PC
- Researchers share time on world's biggest computers to solve science's biggest problems

We've come a long way since MITS developed the first personal computer in 1974, which was sold as a kit that required the customer to assemble the machine themselves. Jump ahead to 2018, and around 77% of Americans currently own a smartphone, and nearly half of the global population uses the internet.

The devices we keep at home and in our pockets are pretty advanced compared to the technology of the past, but they can't hold a candle to the raw power of a supercomputer.

Superpowering science. *Faster processing speeds, extra memory, and super-sized storage capacity are what make supercomputers the tools of choice for many researchers.*

The capabilities of the HPC machines we talk about so often here at Science Node can be hard to conceptualize. That's why we're going to lay it all out for you and explain how supercomputers differ from the laptop on your desk, and just what it is these machines need all that extra performance for.

The need for speed

Computer performance is measured in FLOPS, which stands for floating-point operations per second. The more FLOPS a computer can process, the more powerful it is.

You've come a long way, baby. *The first personal computer, the Altair 8800, was sold in 1974 as a mail-order kit that users had to assemble themselves.*

For example, look to the Intel Core i9 Extreme Edition processor designed for desktop computers. It has 18 cores, or processing units that take in tasks and complete them based on received instructions.

This single chip is capable of one trillion floating point operations per second (i.e., 1 teraFLOP)—as fast as a supercomputer from 1998. You don't need that kind of performance to check email and surf the web, but it's great for hardcore gamers, livestreaming, and virtual reality.

Modern supercomputers use similar chips, memory, and storage as personal computers, but instead of a few processors they have tens of thousands. What distinguishes supercomputers is scale.

China's Sunway TaihuLight, which is currently the fastest supercomputer in the world, boasts 10,648,600 cores with a maximum performance of more than 93,014.6 teraFLOPS.

Theoretically, the Sunway TaihuLight is capable of reaching 125,436 teraFLOPS of performance—more than 125 thousand times faster than the Intel Core i9 Extreme Edition processor. And it 'only' cost around ¥1.8 billion ($270 million), compared to the Intel chip's price tag of $1,999.

Last week

- **Repeat: CAS and atomics**
  - Basis for lock-free and wait-free algorithms
- **Lock-free**
  - Stack - single update of head - simpler
  - List - manage multiple pointers, importance of mark bits again
- **Unbounded Queues**
  - More complex example for lock-free, how to design a more realistic datastructure

Learning goals today

Literature:
Herlihy: Chapter 10

- **Memory Reuse and the ABA Problem**
  - Understand one of the most complex pitfalls in shared memory parallel programming
  - Various solutions

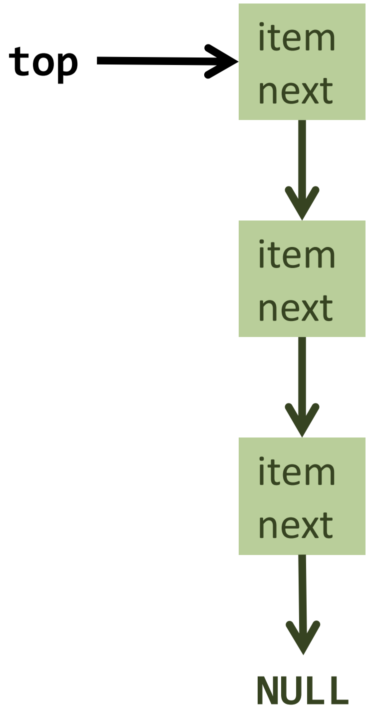
- **Theoretical background (finally!)**
  - Linearizability
  - Consistency
  - Histories
  - Composability

(Susser 1968)

- "... to practice without theory is to sail an uncharted sea; theory without practice is not to set sail at all".

REUSE AND THE ABA PROBLEM

For the sake of simplicity: back to the stack ☺



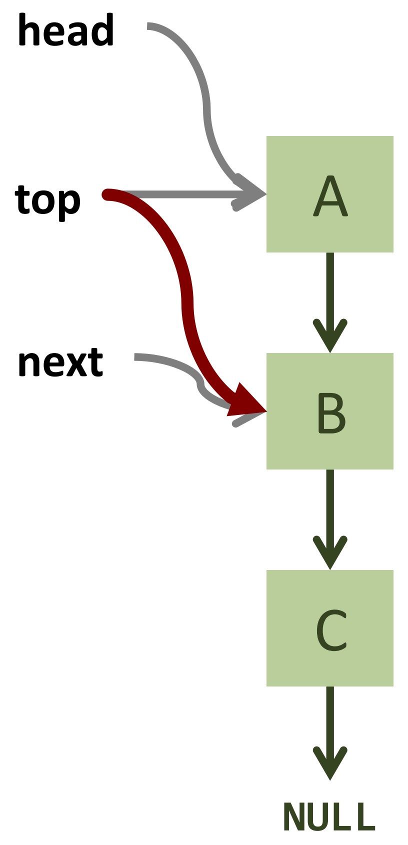

```
public class ConcurrentStack {
AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```

pop

```
public Long pop() {
  Node head, next;

  do {
    head = top.get();
    if (head == null) return null;
    next = head.next;
  } while (!top.compareAndSet(head, next));

  return head.item;
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change

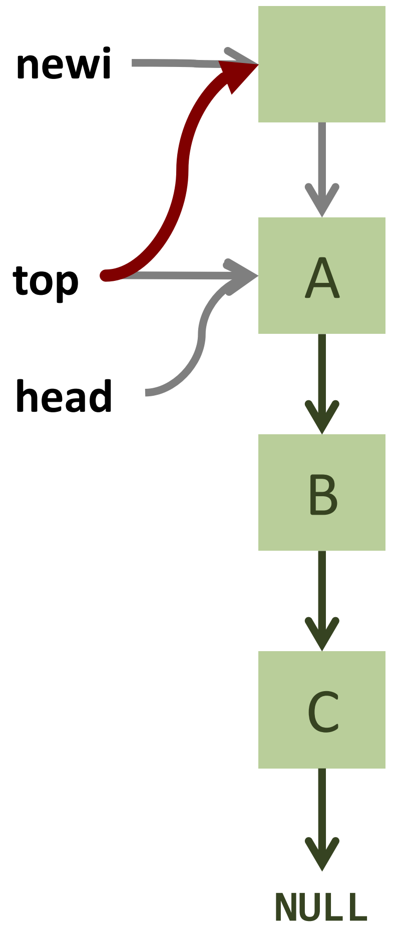


push

```
public void push(Long item) {
     Node newi = new Node(item);
     Node head;

     do {
          head = top.get();
          newi.next = head;
     } while (!top.compareAndSet(head, newi));
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change



Node reuse

Assume we do not want to allocate for each push and maintain a node pool instead. Does this work?

```
public class NodePool {
   AtomicReference<Node> top new AtomicReference<Node>();

   public void put(Node n) { … }
   public Node get() { … }
}

public class ConcurrentStackP {
   AtomicReference<Node> top = newAtomicReference<Node>();
   NodePool pool = new NodePool();
   ...
}
```

NodePool put and get

```
public Node get(Long item) {
   Node head, next;
   do {
      head = top.get();
      if (head == null) return new Node(item);
      next = head.next;
   } while (!top.compareAndSet(head, next));
   head.item = item;
   return head;
}

public void put(Node n) {
   Node head;
   do {
      head = top.get();
      n.next = head;
   } while (!top.compareAndSet(head, n));
}
```

Only difference to Stack above: NodePool is in-place.

A node can be placed in one and only one in-place data structure. This is ok for a global pool.

***So far* this works.**

Using the node pool

```
public void push(Long item) {
    Node head;
    Node new = pool.get(item);
    do {
        head = top.get();
        new.next = head;
    } while (!top.compareAndSet(head, new));
}

public Long pop() {
    Node head, next;
    do {
        head = top.get();
        if (head == null) return null;
        next = head.next;
    } while (!top.compareAndSet(head, next));
    Long item = head.item;
    pool.put(head);
    return item;
}
```

Experiment

- run n consumer and producer threads
- each consumer / producer pushes / pops 10,000 elements and records sum of values
- if a pop returns an "empty" value, retry
- do this 10 times with / without node pool
- measure wall clock time (ms)
- check that sum of pushed values == sum of popped values

Result (of one particular run)

nonblocking stack **without** reuse

n = 1, elapsed= 15, normalized= 15

n = 2, elapsed= 110, normalized= 55

n = 4, elapsed= 249, normalized= 62

n = 8, elapsed= 843, normalized= 105

n = 16, elapsed= 1653, normalized= 103

n = 32, elapsed= 3978, normalized= 124

n = 64, elapsed= 9953, normalized= 155

n = 128, elapsed= 24991, normalized= 195

nonblocking stack **with** reuse

n = 1, elapsed= 47, normalized= 47

n = 2, elapsed= 109, normalized= 54

n = 4, elapsed= 312, normalized= 78

n = 8, elapsed= 577, normalized= 72

n = 16, elapsed= 1747, normalized= 109

n = 32, elapsed= 2917, normalized= 91

n = 64, elapsed= 6599, normalized= 103

n = 128, elapsed= 12090, normalized= 94

yieppieh...

But other runs ...

nonblocking stack **with** reuse

n = 1, elapsed= 62, normalized= 62

n = 2, elapsed= 78, normalized= 39

n = 4, elapsed= 250, normalized= 62

n = 8, elapsed= 515, normalized= 64

n = 16, elapsed= 1280, normalized= 80

n = 32, elapsed= 2629, normalized= 82

Exception in thread "main"
java.lang.RuntimeException:
sums of pushes and pops don't match

at stack.Measurement.main(Measurement.java:107)

nonblocking stack **with** reuse

n = 1, elapsed= 48, normalized= 48

n = 2, elapsed= 94, normalized= 47

n = 4, elapsed= 265, normalized= 66

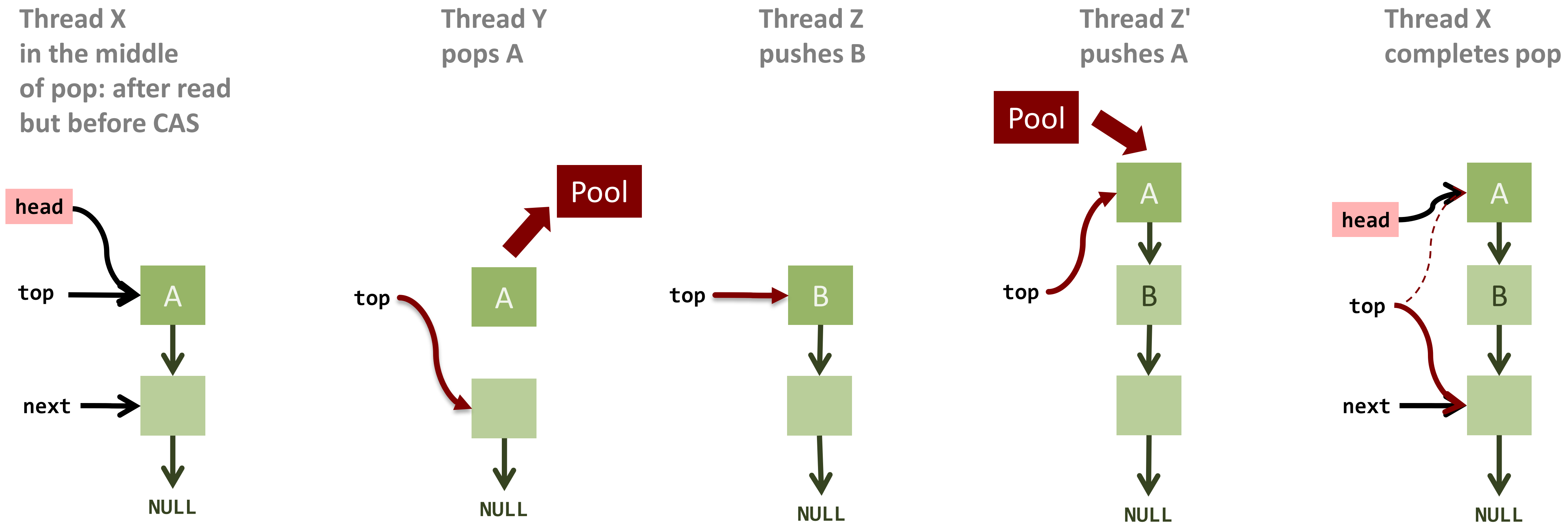
n = 8, elapsed= 530, normalized= 66

n = 16, elapsed= 1248, normalized= 78

[and does not return]

why?

ABA Problem

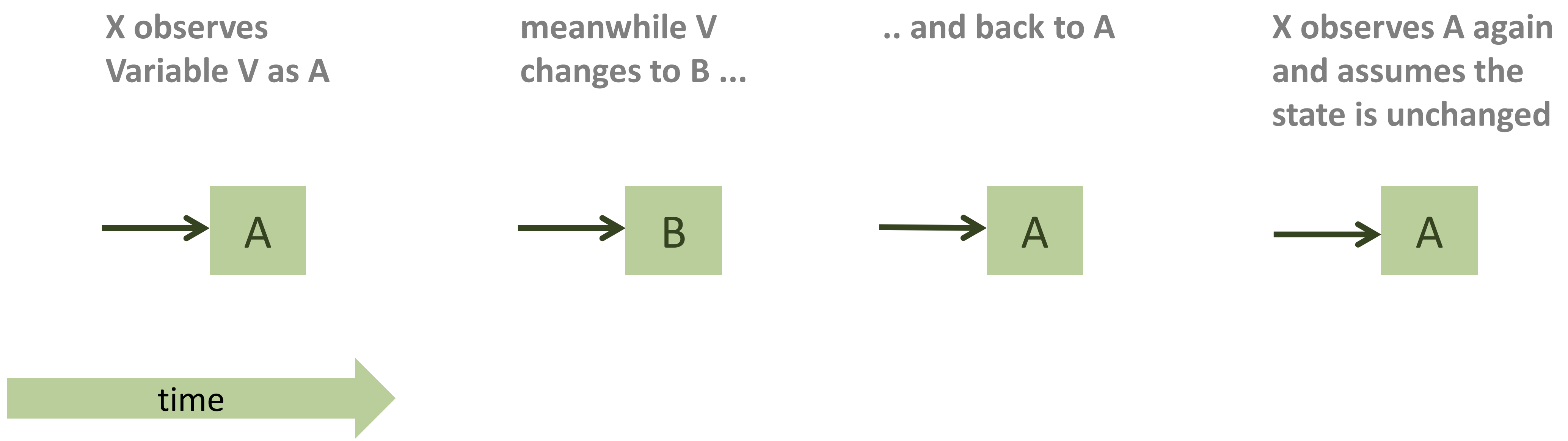



```
public Long pop() {
      Node head, next;
      do {
            head = top.get();
            if (head == null) return null;
            next = head.next;
      } while (!top.compareAndSet(head, next));
      Long item = head.item; pool.put(head); return item;
}
```

```
public void push(Long item) {
      Node head;
      Node new = pool.get(item);
      do {
            head = top.get();
            new.next = head;
      } while (!top.compareAndSet(head, new));
}
```

The ABA-Problem

"The ABA problem ... occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed."



How to solve the ABA problem?

DCAS (double compare and swap)

- not available on most platforms (we have used a variant for the lock-free list set)

Garbage Collection

- relies on the existence of a GC
- much too slow to use in the inner loop of a runtime kernel
- can you implement a lock-free garbage collector relying on garbage collection?

Pointer Tagging

- does not cure the problem, rather delay it
- can be practical

Hazard Pointers

Transactional memory (later)

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging

MSB	...	X	X	X	X	X	X	X	X	0	0	0	0	0

Each time a pointer is stored in a data structure, the tag is increased by one.
Access to a data structure via address x – (x mod 32)

This makes the ABA problem very much less probable because now 32 versions of each pointer exist.

Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

- before X reads P, it marks it hazarduous by entering it in one of the n (n= number threads) slots of an array associated with the data structure (e.g., the stack)
- When finished (after the CAS), process X removes P from the array
- Before a process Y tries to reuse P, it checks all entries of the hazard array

Hazard Pointers

```
public class NonBlockingStackPooledHazardGlobal extends Stack {
    AtomicReference<Node> top = new AtomicReference<Node>();
    NodePoolHazard pool;
    AtomicReferenceArray<Node> hazarduous;

    public NonBlockingStackPooledHazardGlobal(int nThreads) {
        hazarduous =  new AtomicReferenceArray<Node>(nThreads);
        pool = new NodePoolHazard(nThreads);
    }
}
```

null	null	null	null	null	null	null	null	null	null	null	null

0 nThreads-1

Hazard Pointers

null	null	null	null	null	**hd**	null	y	null	x	null	null

0 id nThreads-1

```
boolean isHazarduous(Node node) {
    for (int i = 0; i < hazarduous.length(); ++i)
        if (hazarduous.get(i) == node)
            return true;
    return false;
}

void setHazardous(Node node) {
    hazarduous.set(id, node); // id is current thread id
}
```

Hazard Pointers

null	null	null	null	null	hd	null	y	null	x	null	null
0					id						nThreads-1

```
public int pop(int id) {
    Node head, next = null;
    do {
        do {
            head = top.get();
            setHazarduous(head);
        } while (head == null || top.get() != head);
        next = head.next;
    } while (!top.compareAndSet(head, next));
    setHazarduous(null);
    int item = head.item;
    if (!isHazardous(head))
        pool.put(id, head);
    return item;
}
```

```
public void push(int id, Long item) {
    Node head;
    Node newi = pool.get(id, item);
    do{
        head = top.get();
        newi.next = head;
    } while (!top.compareAndSet(head, newi));
}
```

This ensures that no other thread is already past the CAS and has not seen our hazard pointer

How to protect the Node Pool?

The ABA problem also occurs on the node pool.
Two solutions:

Thread-local node pools

- No protection necessary
- Does not help when push/pop operations are not well balanced

Hazard pointers on the global node pool

- Expensive operation for node reuse
- Equivalent to code above: node pool returns a node only when it is not hazarduous

Remarks

The Java code above does not really improve performance in comparison to memory allocation plus garbage collection.

But it demonstrates how to solve the ABA problem principally.

The hazard pointers are placed **in thread-local storage.**

When thread-local storage can be replaced by processor-local storage, it scales better*.

* e.g., in Florian Negele, *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System*, PhD Thesis, ETH Zürich 2014

Lessons Learned

Lock-free programming: new kind of problems in comparison to lock-based programming:

- Atomic update of several pointers / values impossible, leading to new kind of problems and solutions, such as threads that help each other in order to guarantee global progress
- ABA problem (which disappears with a garbage collector)

Recap: we have seen …

- algorithms to implement critical sections and locks
- hardware support for implementing critical sections and locks
- how to reason about concurrent algorithms using state diagrams
- high-level constructs such as semaphores and monitors that raise the level of abstraction
- lock-free implementations that require Read-Modify-Write operations

Literature:
Herlihy: Chapter 3.1 - 3.6

But: we have not (yet) …

developed a clear overview of the theoretical concepts and notions behind such as

- consistency
- linearizability
- consensus
- a language to talk formally about concurrency

  I have been very hand-wavy when answering some tricky questions

- now that you appreciate the complexity

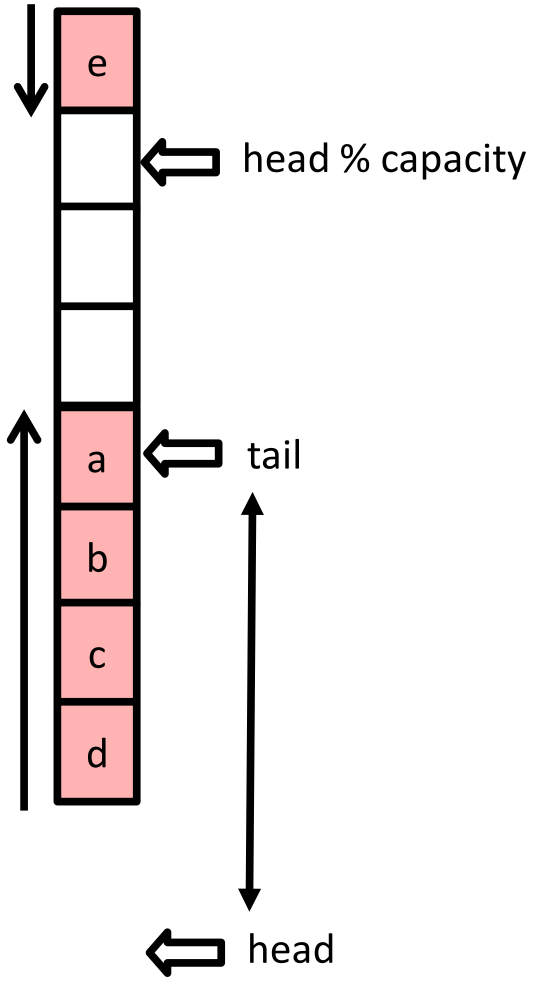
  Let us introduce some non-trivial formalism to capture it

Example: Single-Enqueuer/Dequeuer bounded FIFO queue

```
class WaitFreeQueue {
    volatile int head = 0, tail = 0;
    AtomicReferenceArray<T>[] items =
        new AtomicReferenceArray<T>(capacity);

    public boolean enq(T x) {
        if (tail – head == capacity) return false;
        items.set((tail+1) % capacity, x);
        tail++;
        return true;
    }

    public T deq() {
        if (tail - head == 0) return null;
        int x = items.get((head+1) % capacity);
        head++;
        return x;
    }
}
```

Given that there is only one enqueuer and one dequeuer process. Is the implementation of the FIFO queue from above correct? Why/why not?

For a concurrent, locking queue it is easier to argue. Why/why not?



Sequential Objects – Sequential Specifications (you know this)

An object (e.g., in Java or C++) is a container for data and provides

- a set of **methods** to manipulate data

An object has a well defined

- **state** being modified during method invocations

Well-established as Floyd-Hoare logic to prove correctness

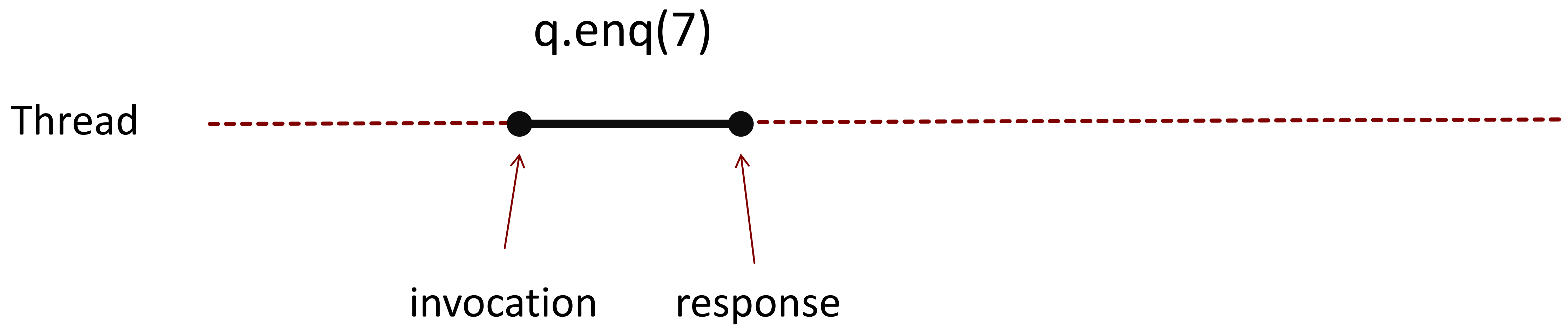
- Defining the objects behavior in terms of a set of **pre- and postconditions** for each method is inherently **sequential**

Can we carry that forward to a parallel formulation?

Method Calls

A **method call** is the interval that starts with an **invocation** and ends with a **response**.

A method call is called **pending** between invocation and response.

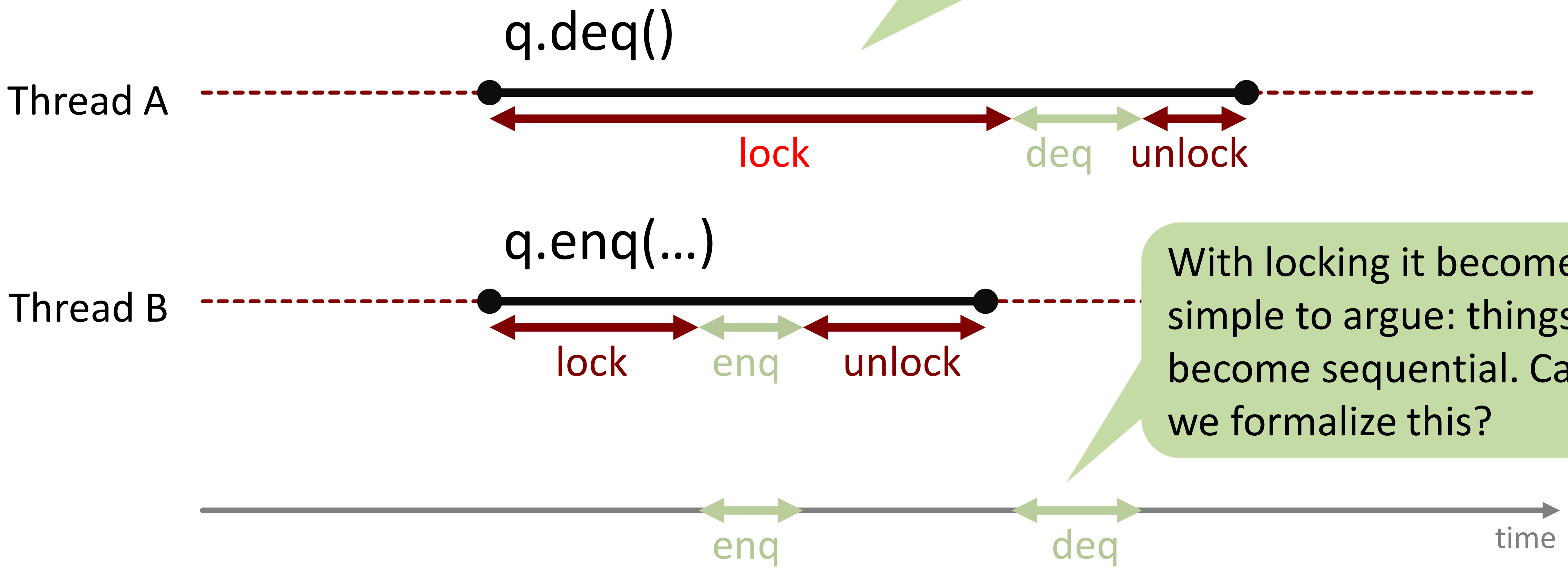


Sequential vs. Concurrent

Sequential	Concurrent
Meaningful state of objects only **between method calls**.	Method **calls can overlap**. Object might **never** be **between** method calls. Exception: periods of *quiescence*.
Methods described in **isolation**.	All possible **interactions** with concurrent calls must be taken into account.
Can **add new methods** without affecting older methods.	Must take into account that everything can **interact** with everything else.
"**Global** clock"	"**Object** clock"

Blocking Queue Behavior

Which thread got the lock first?



With locking it becomes simple to argue: things become sequential. Can we formalize this?

Linearizability

“What's the difference between theory and practice?
Well, in theory there is none.” - folklore

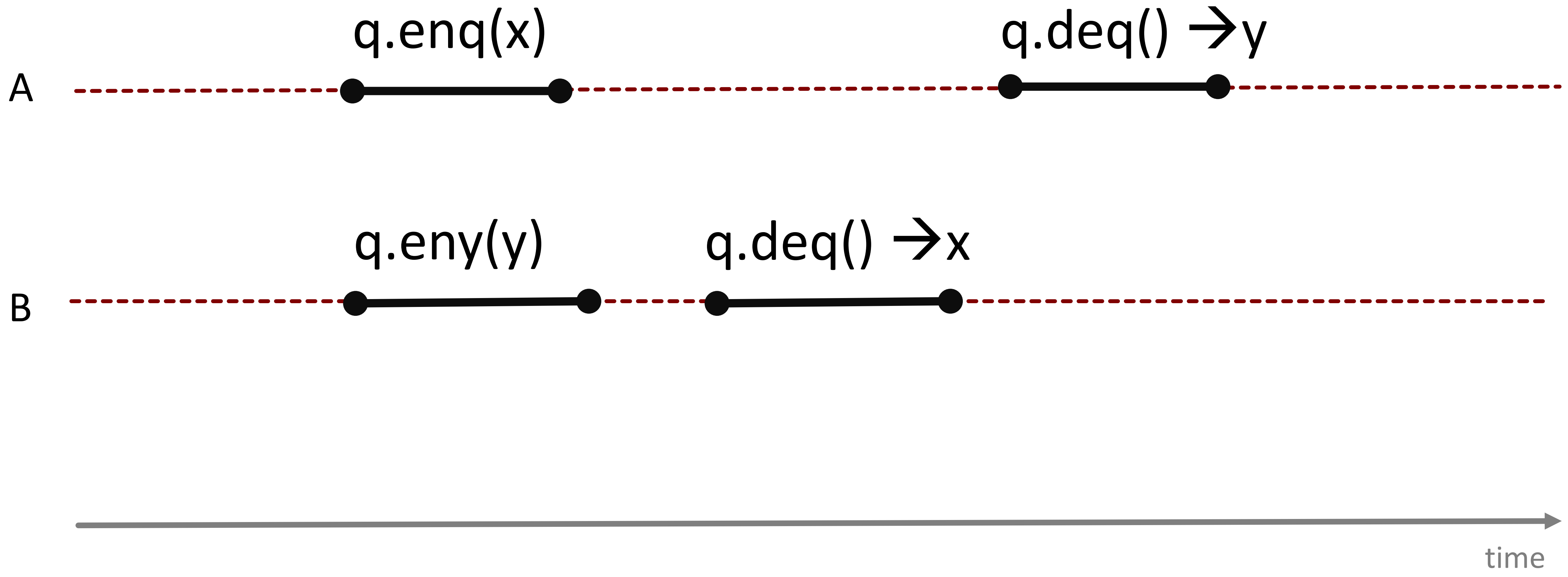
Linearizability

Each method should *appear* to **take effect *instantaneously* between invocation and response events.**

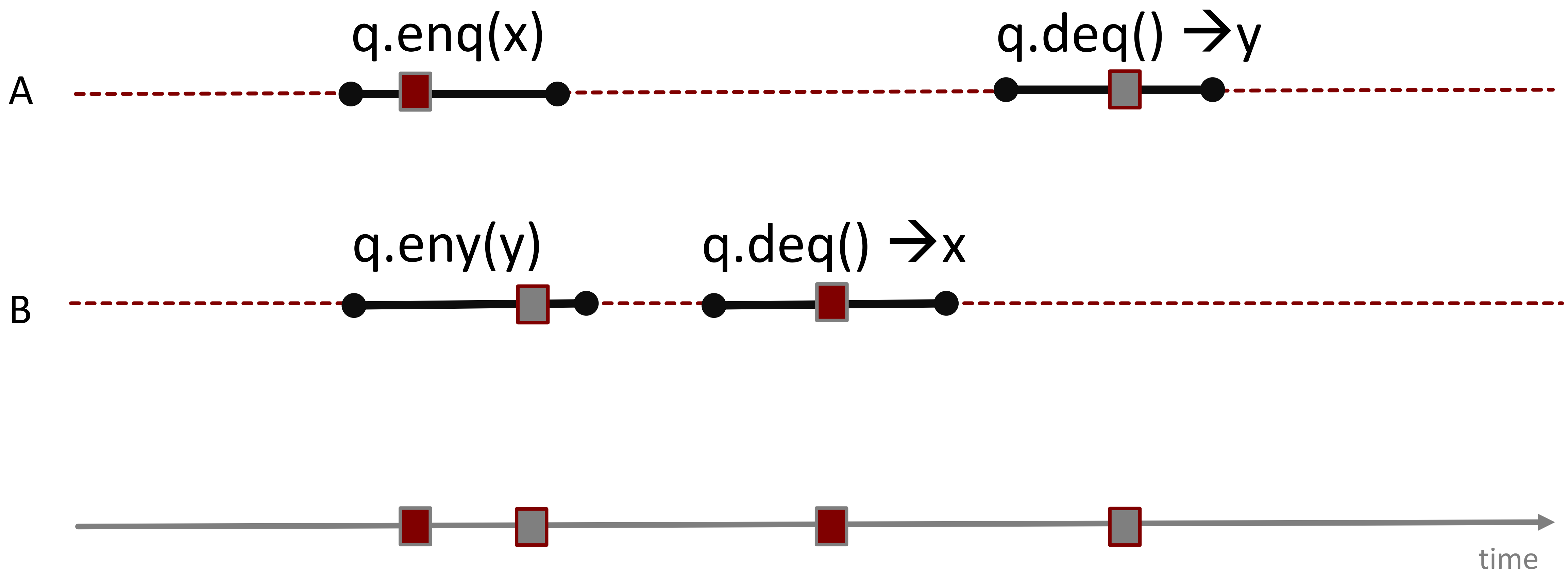
An object for which this is true for all possible executions is called **linearizable**.

The object is correct if the associated sequential behavior is correct.

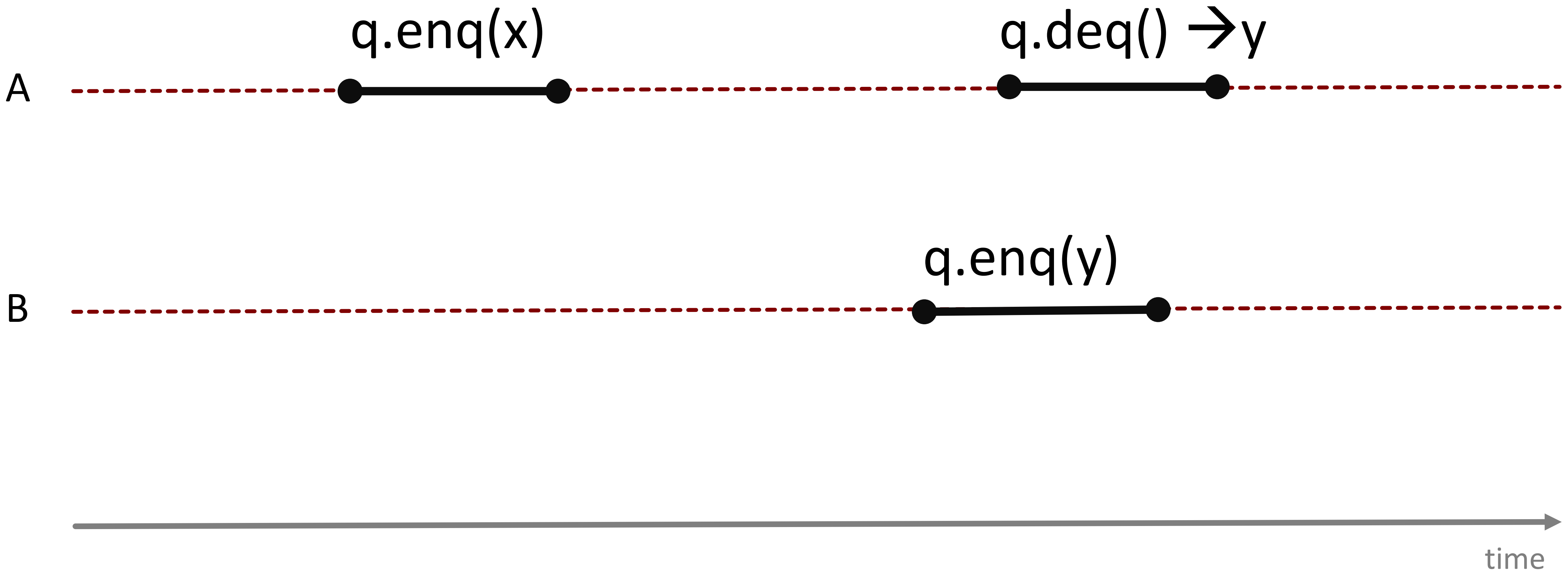
Is this particular execution linearizable?



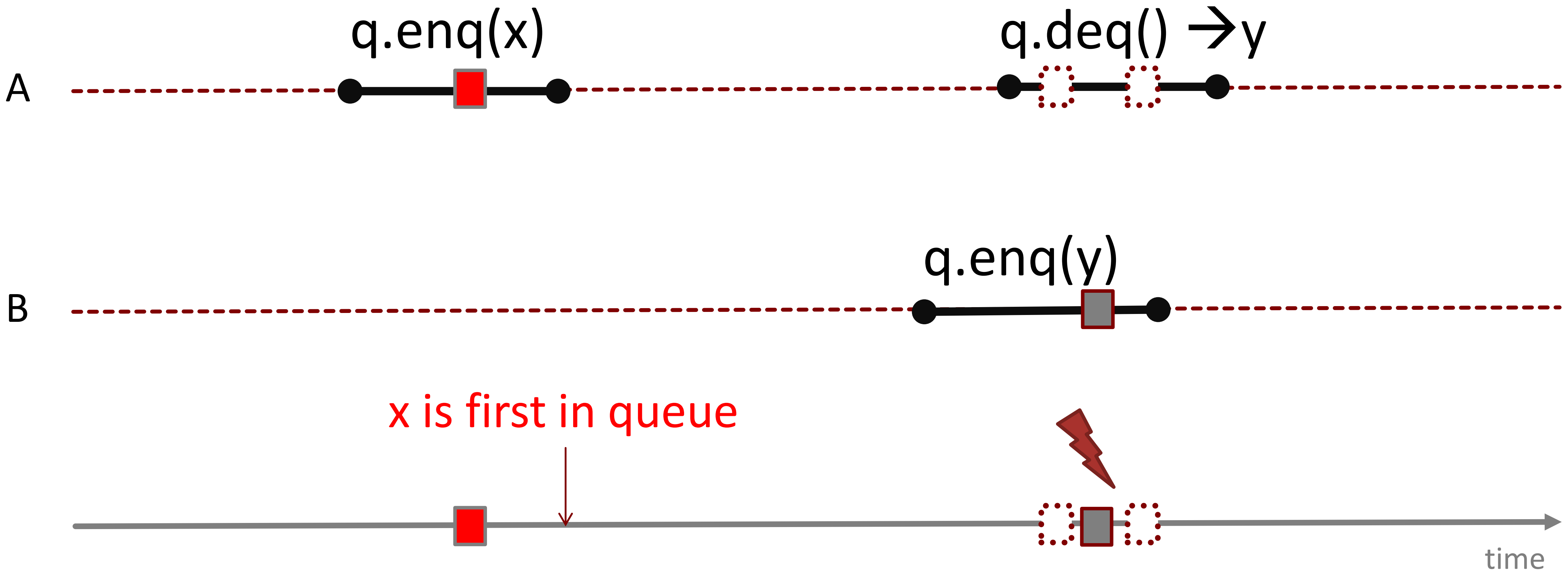
Yes



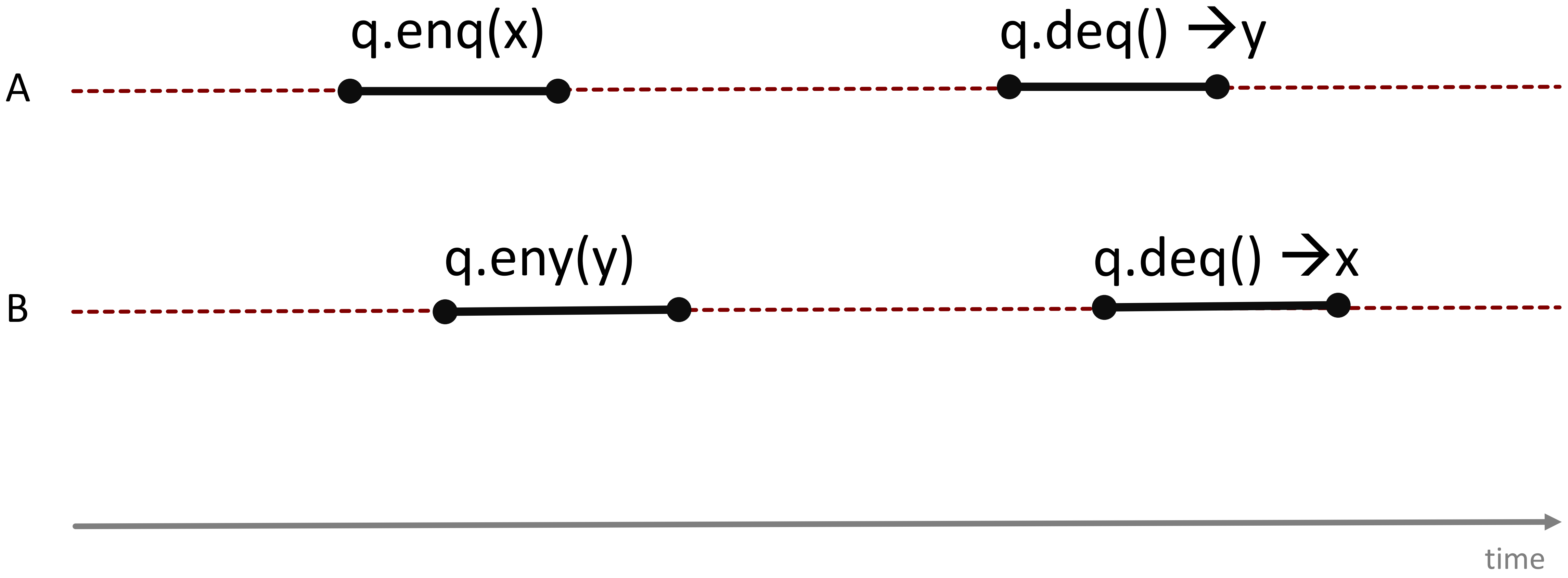
Linearizable?



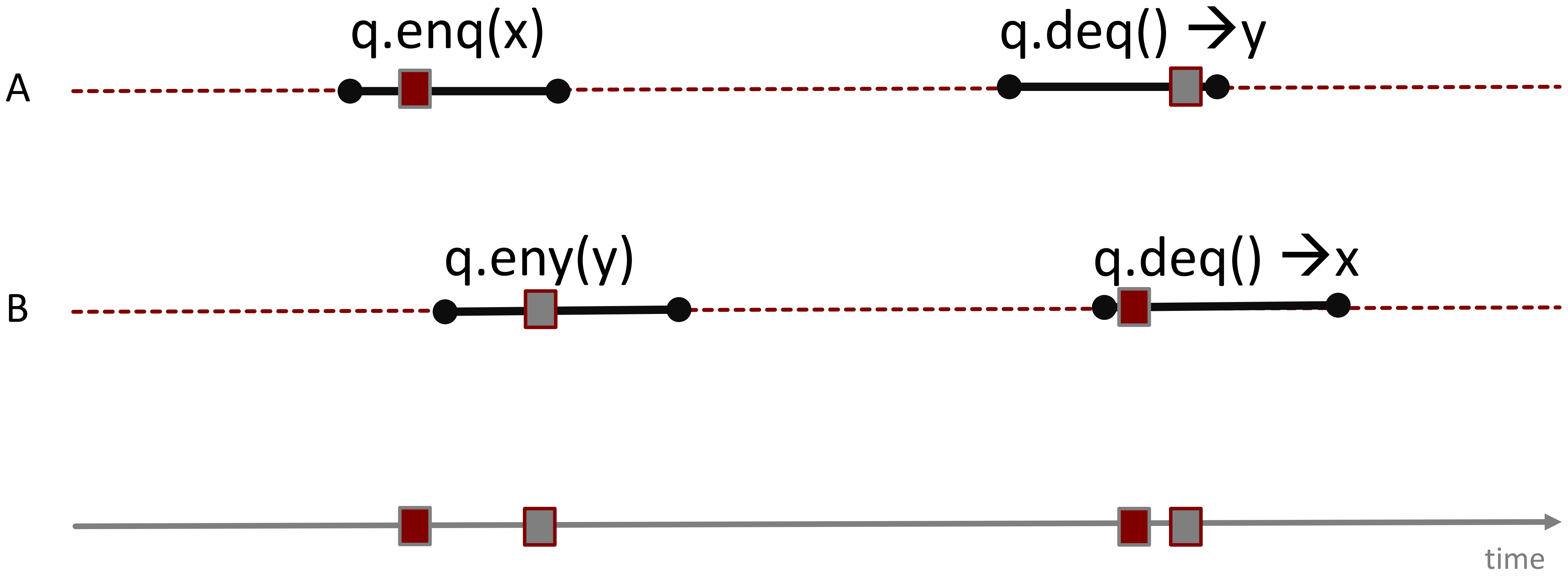
No



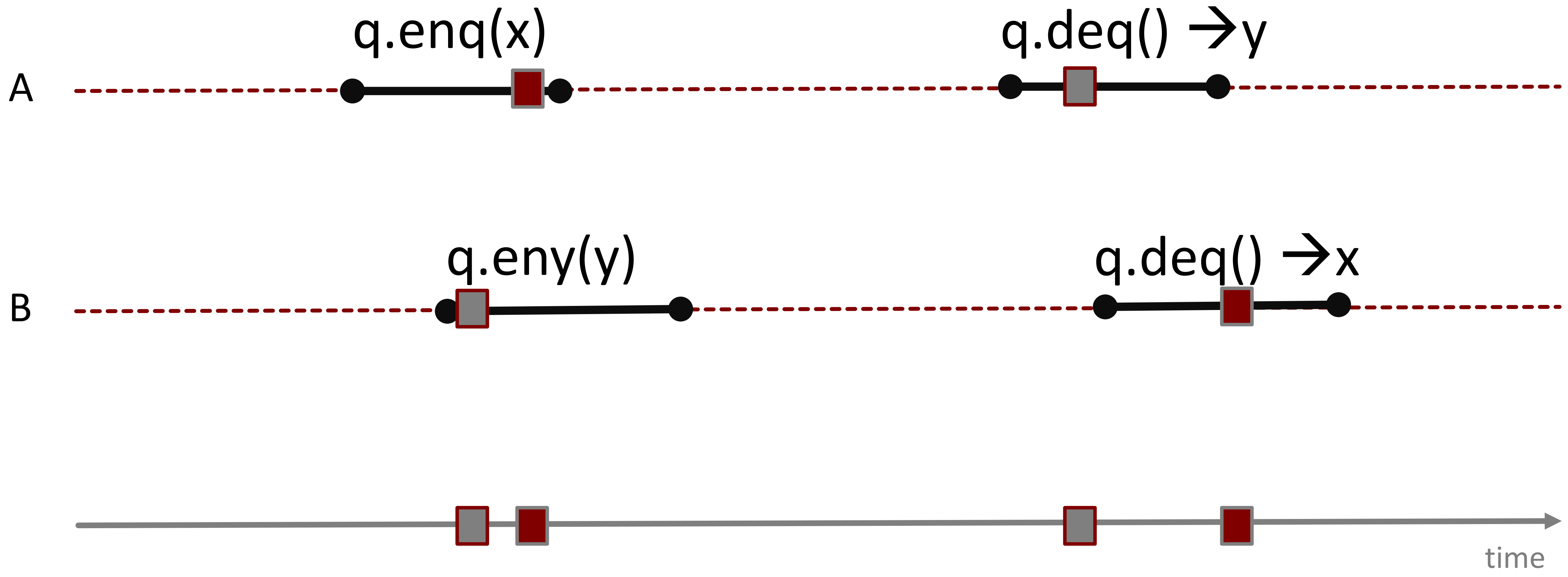
Linearizable ?



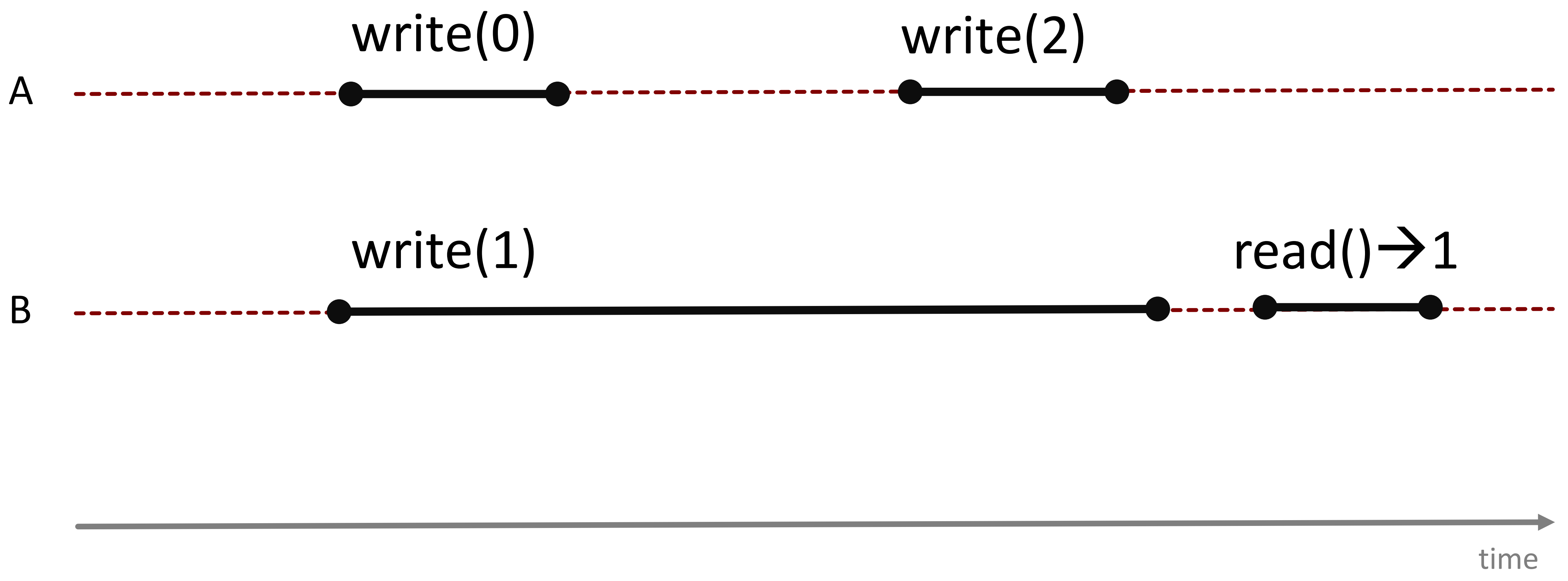
Yes



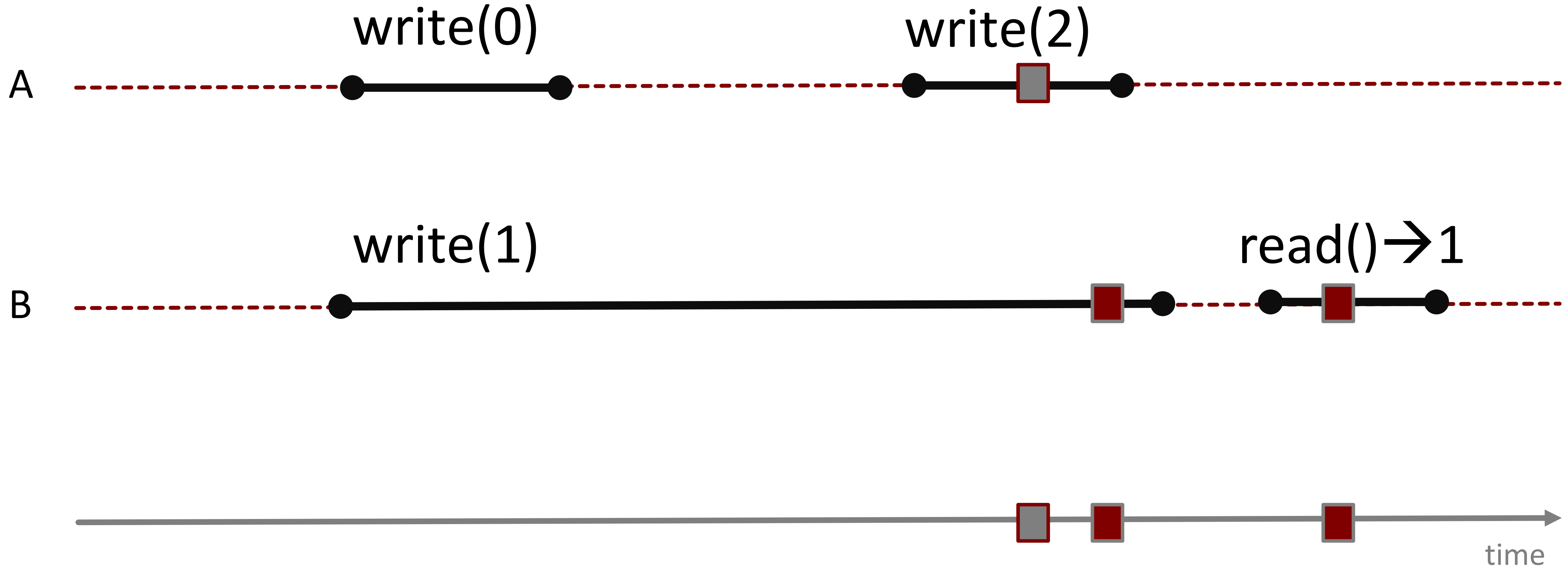
And yes, another scenario.



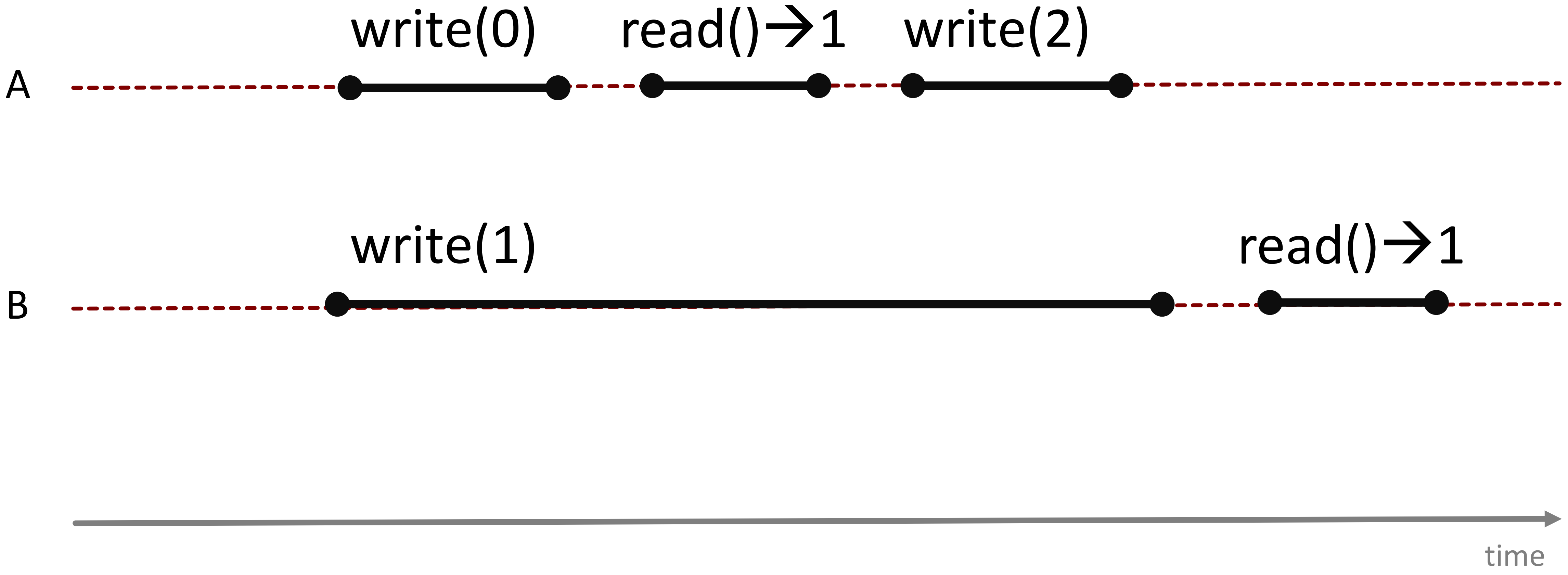
Read/Write Register Example



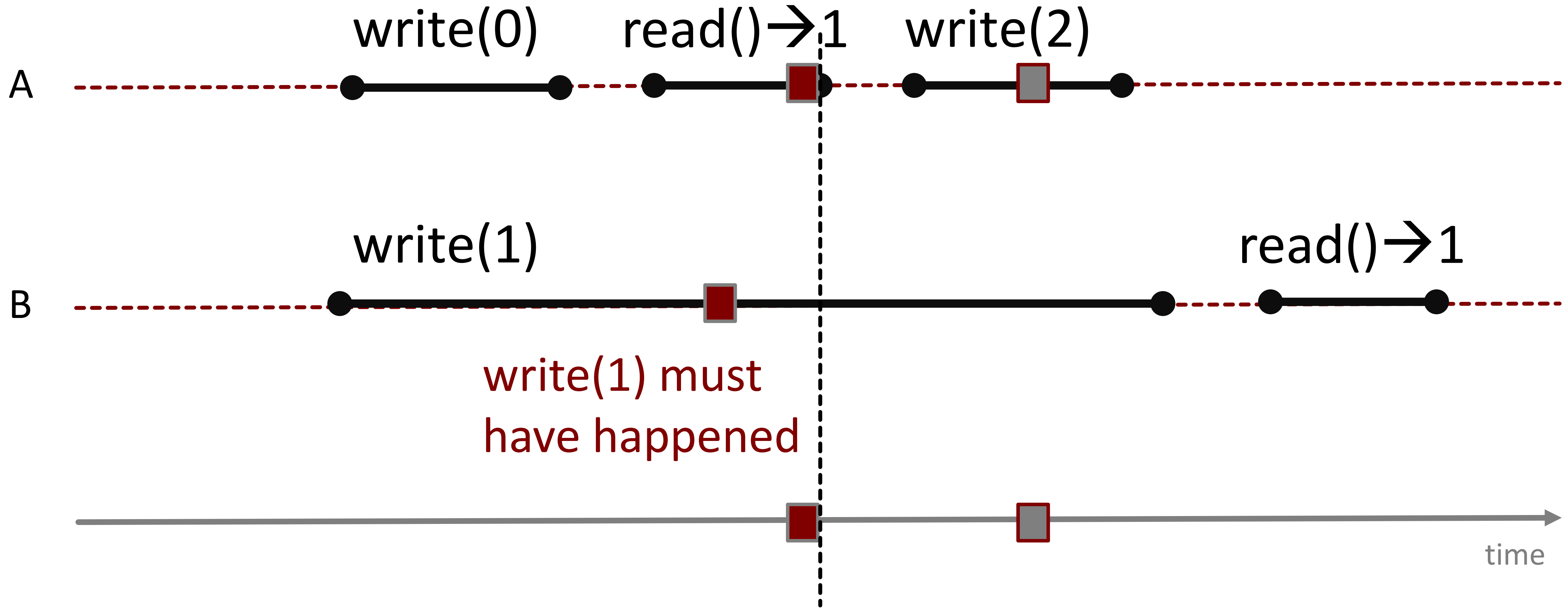
Linearizable!



Linearizable?



No


write(0)
read()→1
write(2)
A
write(1)
read()→1
B
write(1) must
have happened
time

Remark

- We talk about **executions** in order to abstract away from actual method content.
  - A simplification you need to revert (mentally?) for analyzing codes
- The **linearization points** can often be specified, but they may depend on the execution (not only the source code).

- Example: if the queue is empty, a dequeue may fail, while it does not fail with a non-empty queue

```
public int deq() throws EmptyException {
   if (tail == head)
      throw new EmptyException();
   int x = items.get(head++ % capacity);
   return x;
}
```

More formal

Split method calls into two events. Notation:

Invocation

A q.enq(x)

thread

object

method

arguments

Response

A q: void

thread

object

result

History

History H = sequence of invocations and responses

H

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

Invocations and response **match**, if thread names agree and object names agree

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

A subhistory is **complete** when it has no pending responses.

Projections

Object projections

A q.enq(3)
A q:void
A q.enq(5)

H|q =

B q.deq()
B q:3

Thread projections

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

H|B =

Complete subhistories

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

Complete subhistory

History H without its pending invocations.

Sequential histories

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

Sequential history:

- Method calls of different threads do not interleave.
- A final pending invocation is ok.

Well formed histories

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

Well formed history:

Per thread projections sequential

H|A = A q.enq(3)
A q:void

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

Equivalent histories

H=

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

G =

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

H and G **equivalent:**

H|A = G|A
H|B = G|B

Legal histories

Sequential specification tells if a single-threaded, single object history is **legal**

Example: pre- / post conditions

A sequential history H is **legal**, if

- for every object x
- H|x adheres to the sequential specification of x

Precedence

A method call **precedes** another method call if the response event precedes the invocation event

if no precedence then method calls **overlap**

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

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

Notation

Given: history H and method executions m_0 and m_1 on H

Definition: $\boldsymbol{m_0 \rightarrow_H m_1}$ means $\boldsymbol{m_0}$ precedes $\boldsymbol{m_1}$

$\rightarrow_H$ is a relation and implies a partial order on H. The order is total when H is sequential.

Linearizability

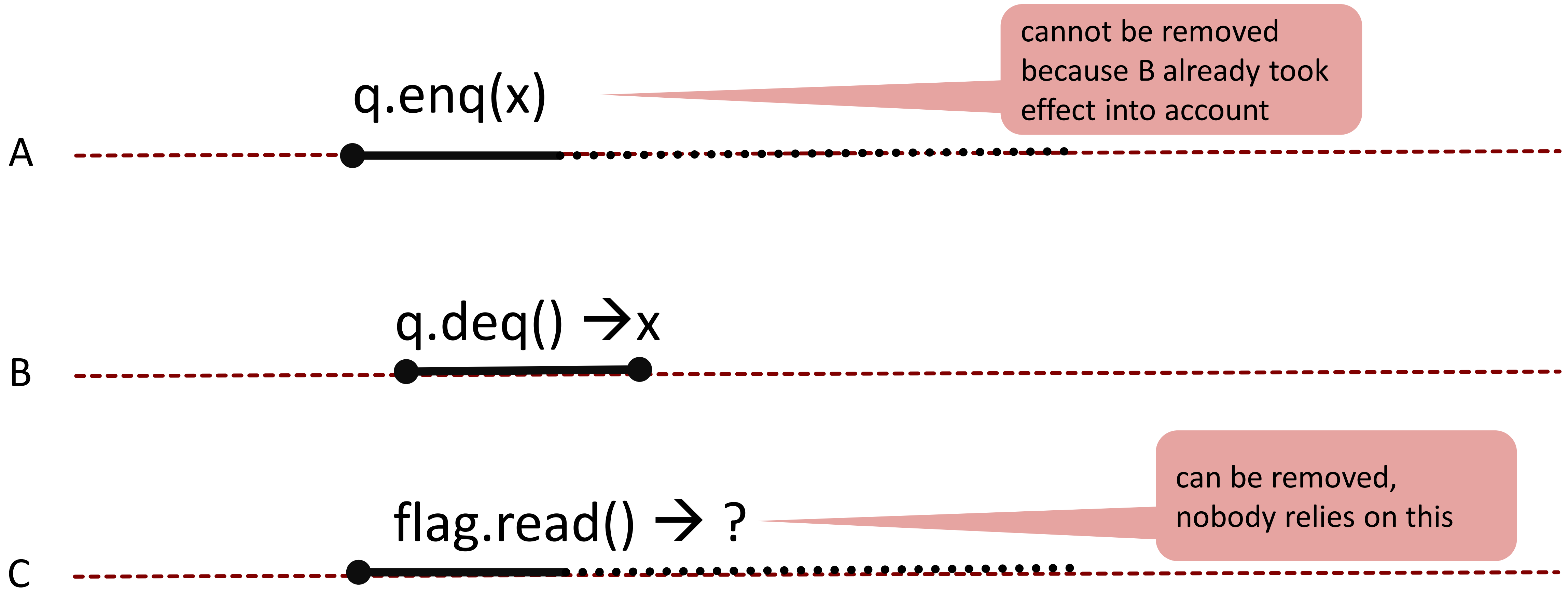
History H is linearizable if it can be extended to a history G

- **appending zero or more responses to pending invocations that took effect**
- **discarding zero or more pending invocations that did not take effect**

such that G is equivalent to a *legal sequential* history S with

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

Invocations that took effect … ?



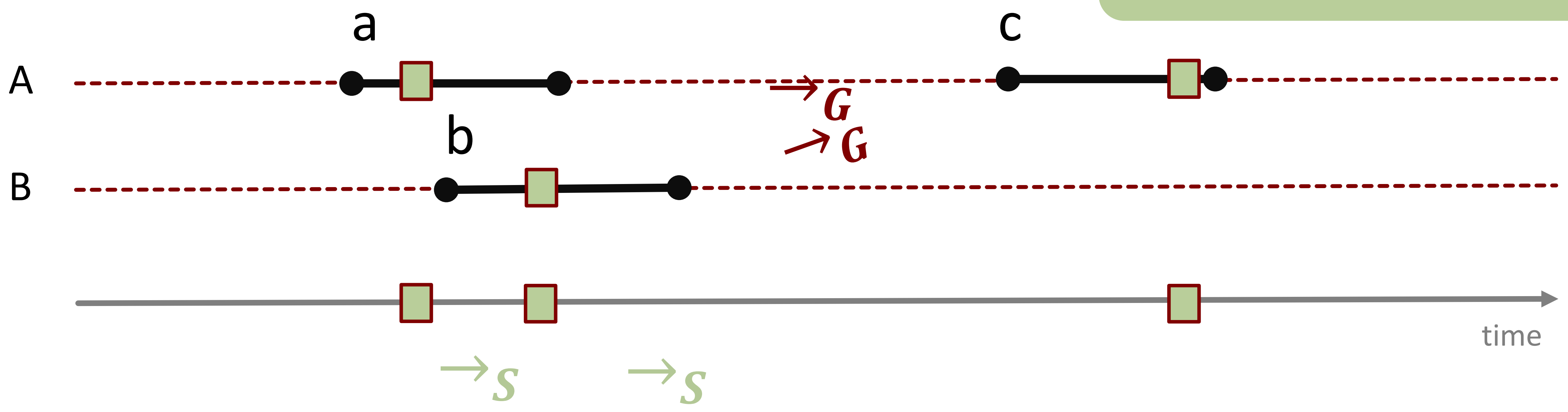
$\rightarrow_G \subset \rightarrow_S$? What does this mean?

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

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

In other words: S respects the real-time order of G

Linearizability: limitation on the possible choice of S



Composability

Composability Theorem

History H is linearizable if and only if

for every object x

H|x is linearizable

Consequence:

Modularity

- Linearizability of objects can be proven in isolation
- Independently implemented objects can be **composed**

Recall: Atomic Registers

Memory location for values of primitive type (boolean, int, ...)

- operations read and write

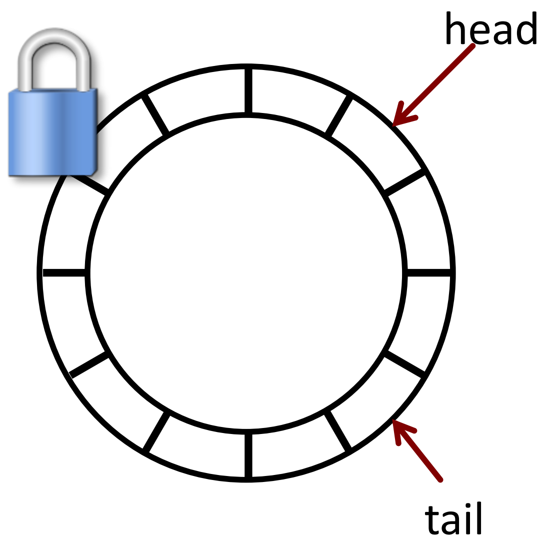
Linearizable with a single linearization point, i.e.

- sequentially consistent, every read operation yields most recently written value
- for non-overlapping operations, the realtime order is respected.

Reasoning About Linearizability (Locking)

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

Linearization points
are when locks are released



Reasoning About Linearizability (Wait-free example)

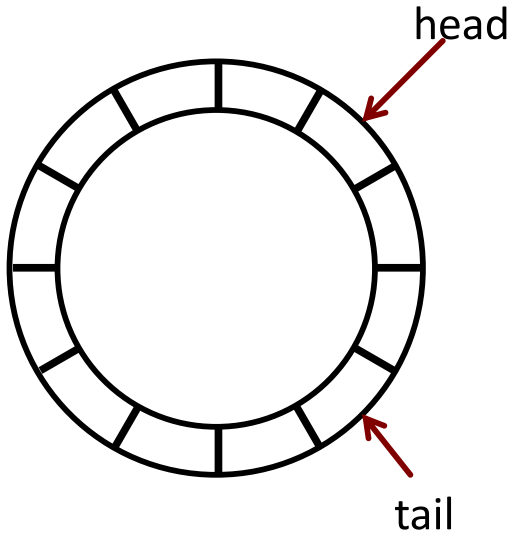
```
class WaitFreeQueue {
    volatile int head = 0, tail=0;
    AtomicReferenceArray<T>[] items =
        new AtomicReferenceArray<T>(capacity);

    public boolean enq (T x) {
        if (tail – heap == capacity) return false;
        items.set((tail+2) % capacity, x);
        tail++;
        return true;
    }

    public T deq() {
        if (tail - head == 0) return null;
        int x = items.get((head+1) % capacity);
        head++;
        return x;
    }
}
```

Linearization point

Linearization point for (only one) enqueuer

Linearization point

Linearization point for (only one) dequeuer



Reasoning About Linearizability (Lock-free example)

```
public T dequeue() {
   while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == last){
         if (next == null) return null;
         else tail.compareAndSet(last, next);
      }
      else {
         T value = next.item;
         if (head.compareAndSet(first, next))
            return value;
      }
   }
}
```

Linearization point

Linearization point

Linearization point

Appendix (for next lecture)

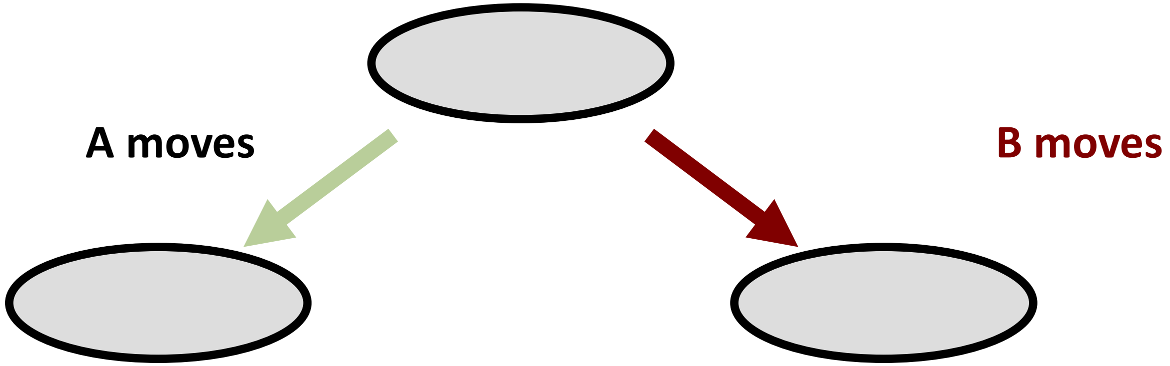
Appendix: Atomic Registers have consensus number 1.

Theorem: Atomic Registers have consensus number 1.

Proof strategy:

- Assume otherwise
- Reason about the properties of any such protocol
- Derive a contradiction
- Suffices to prove for binary consensus and n=2

Wait-Free Computation

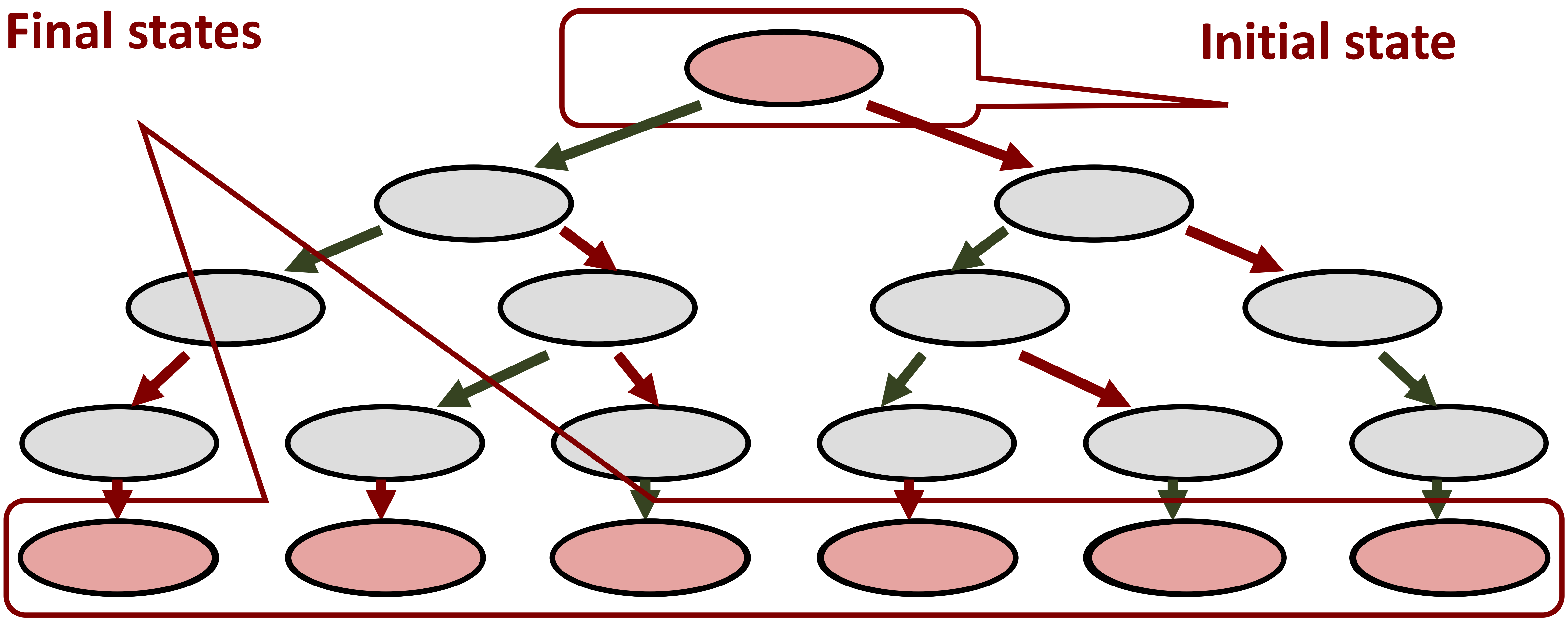


Either A or B "moves"

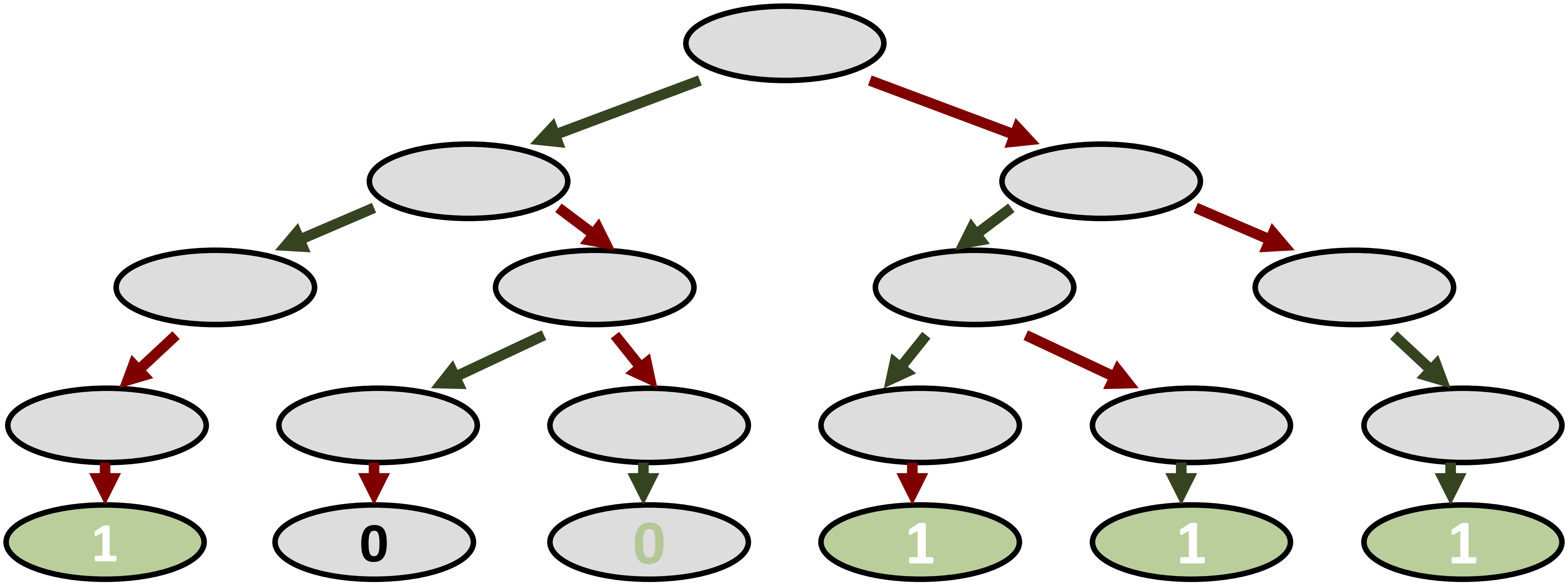
Moving means

- Register read or
- Register write

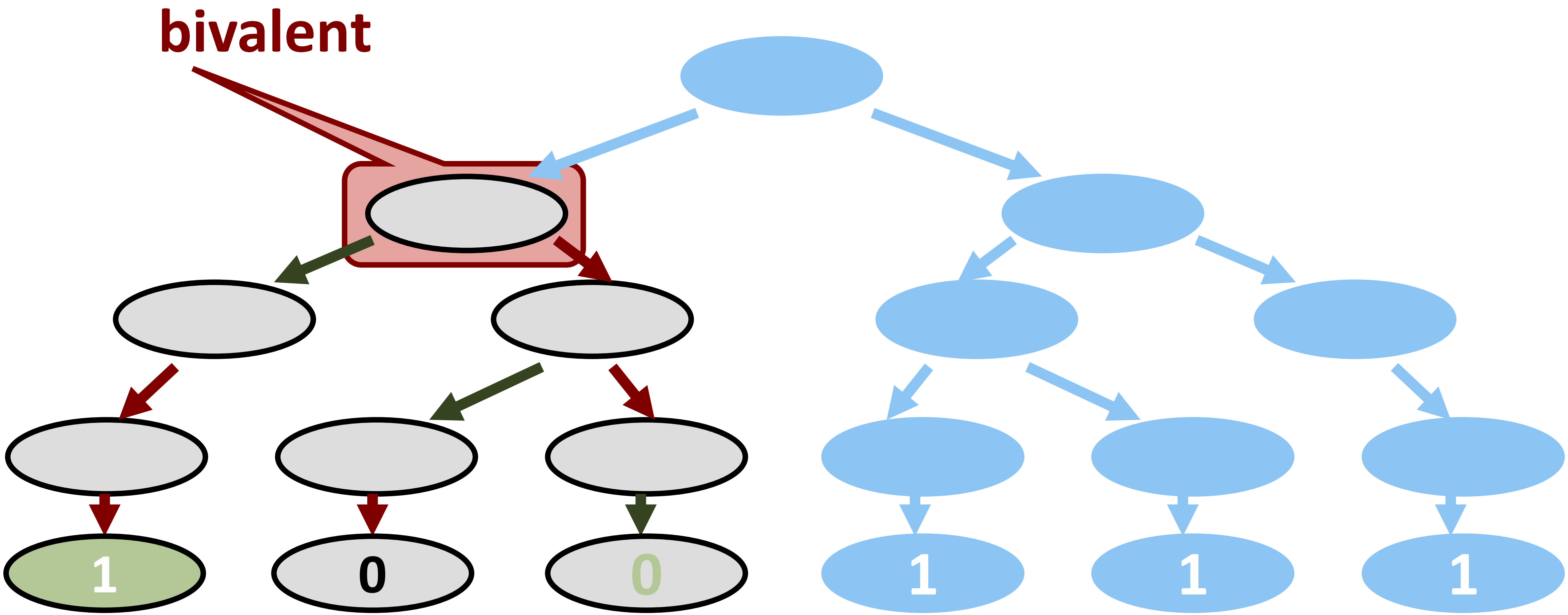
The Two-Move Tree



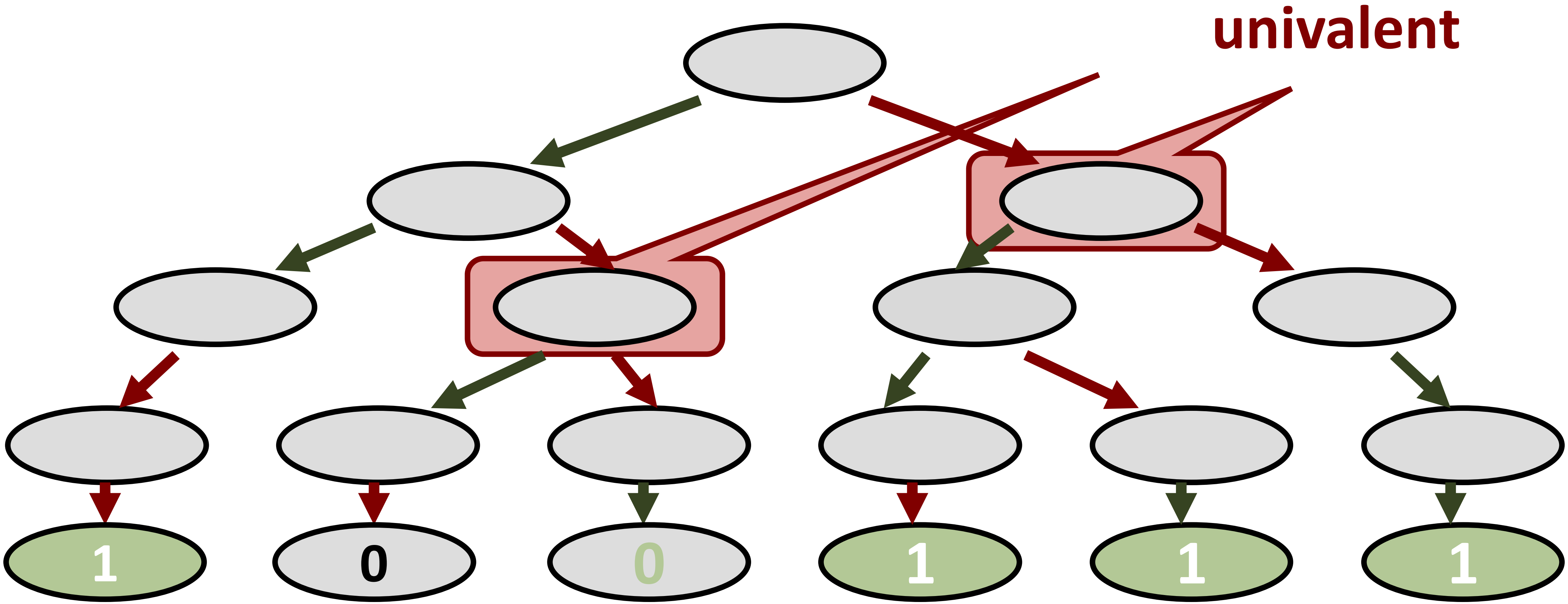
Decision Values



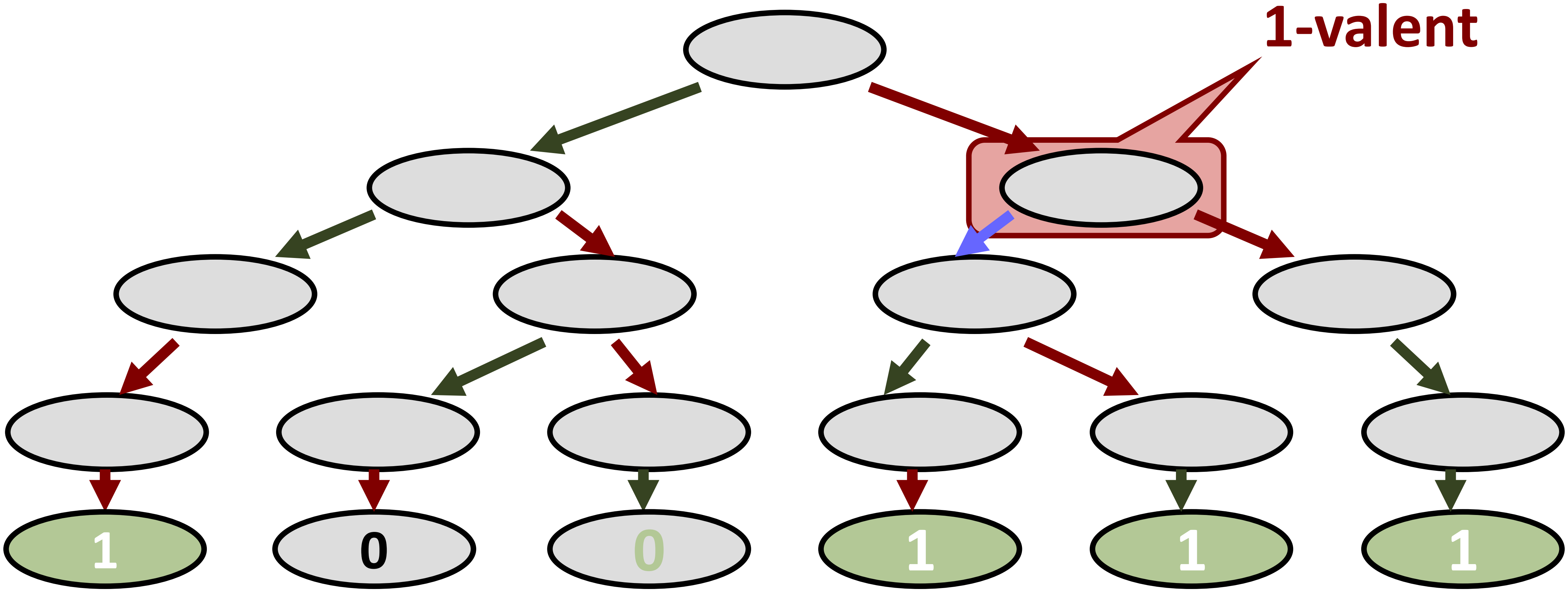
Bivalent: Both Possible



Univalent: Single Value Possible



x-valent: x Only Possible Decision



Summary

Wait-free computation is a tree

Bivalent system states

- Outcome not fixed

Univalent states

- Outcome is fixed
- May not be “known” yet

1-Valent and 0-Valent states

Claim

Some initial state is bivalent

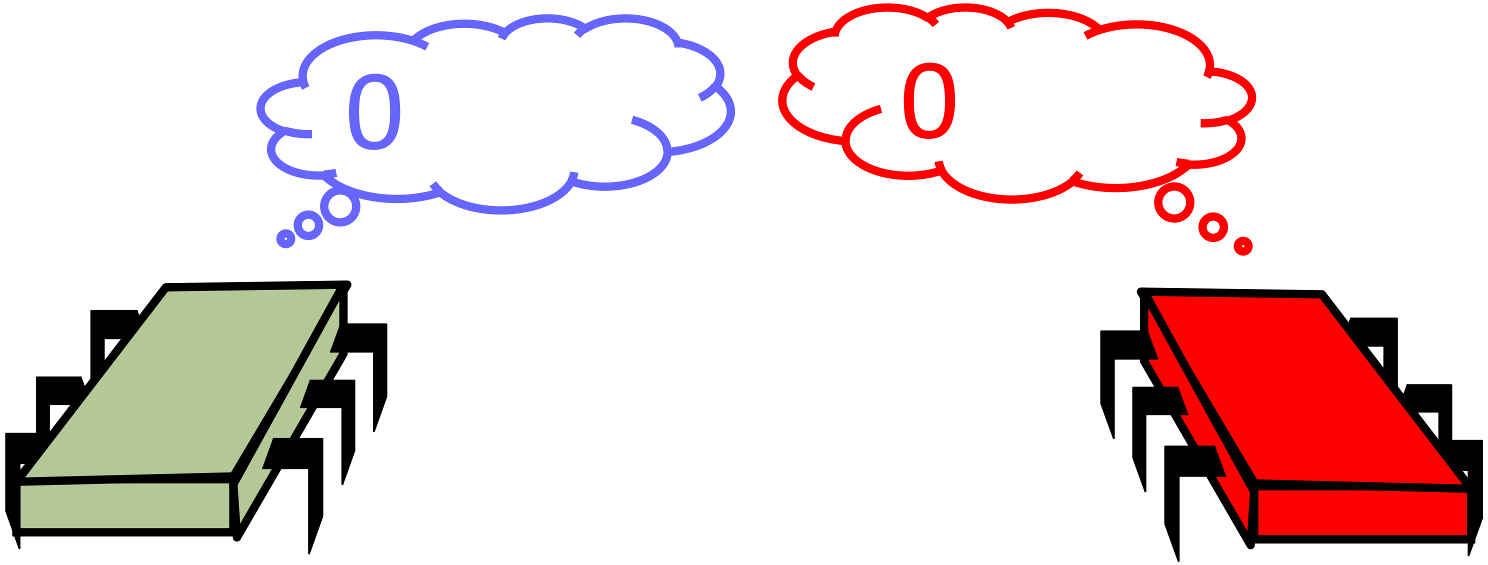
Outcome depends on

- Chance
- Whim of the scheduler

Multiprocessor gods do play dice …

Lets prove this claim

Both Inputs 0

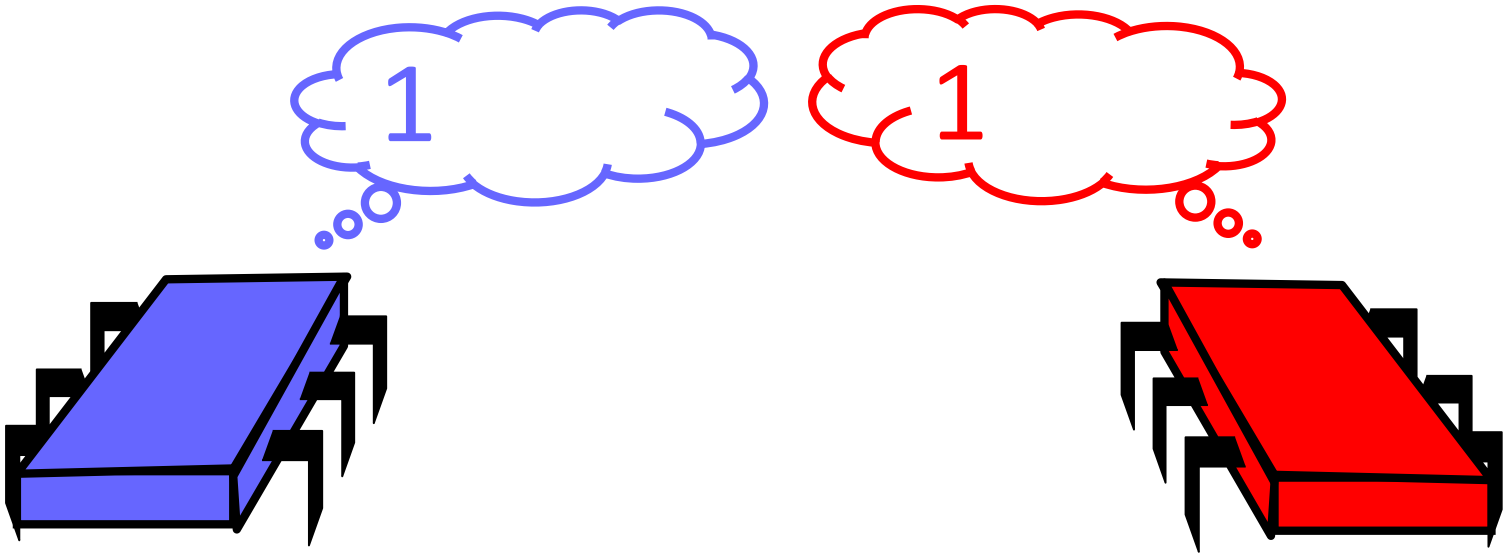


Univalent: all executions must decide 0

Both Inputs 0

Including this solo execution by A

Both Inputs 1

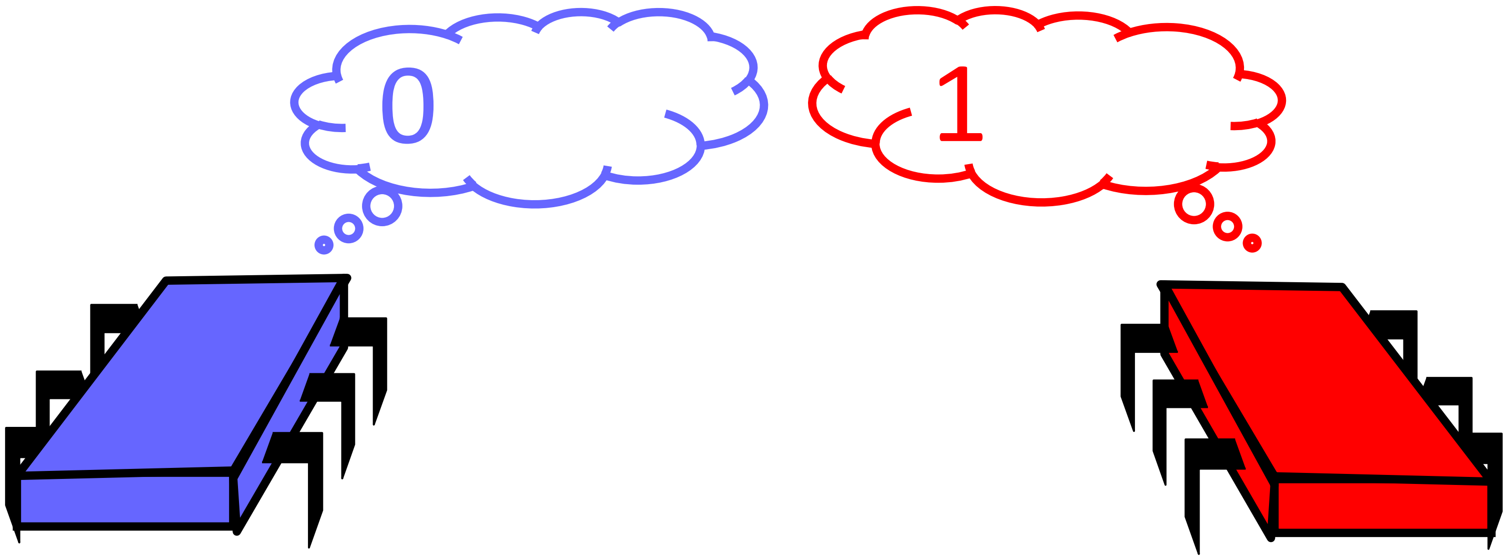


All executions must decide 1

Both Inputs 1

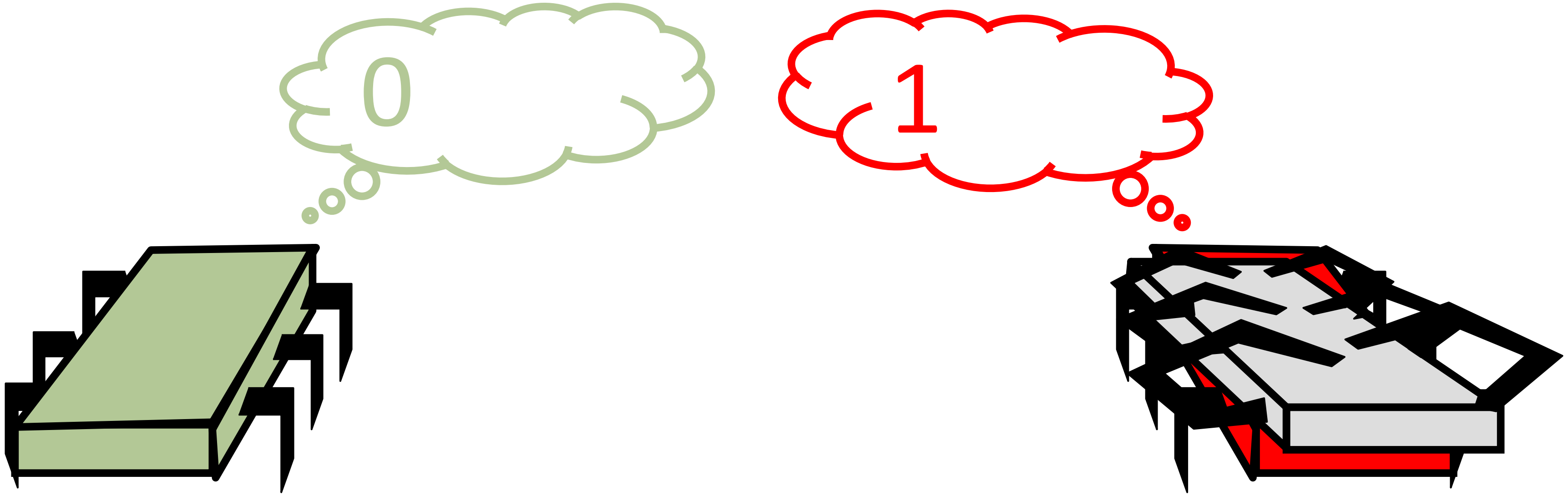
Including this solo execution by B

What if inputs differ?



By Way of contradiction: If univalent
all executions must decide on same value

The Possible Executions

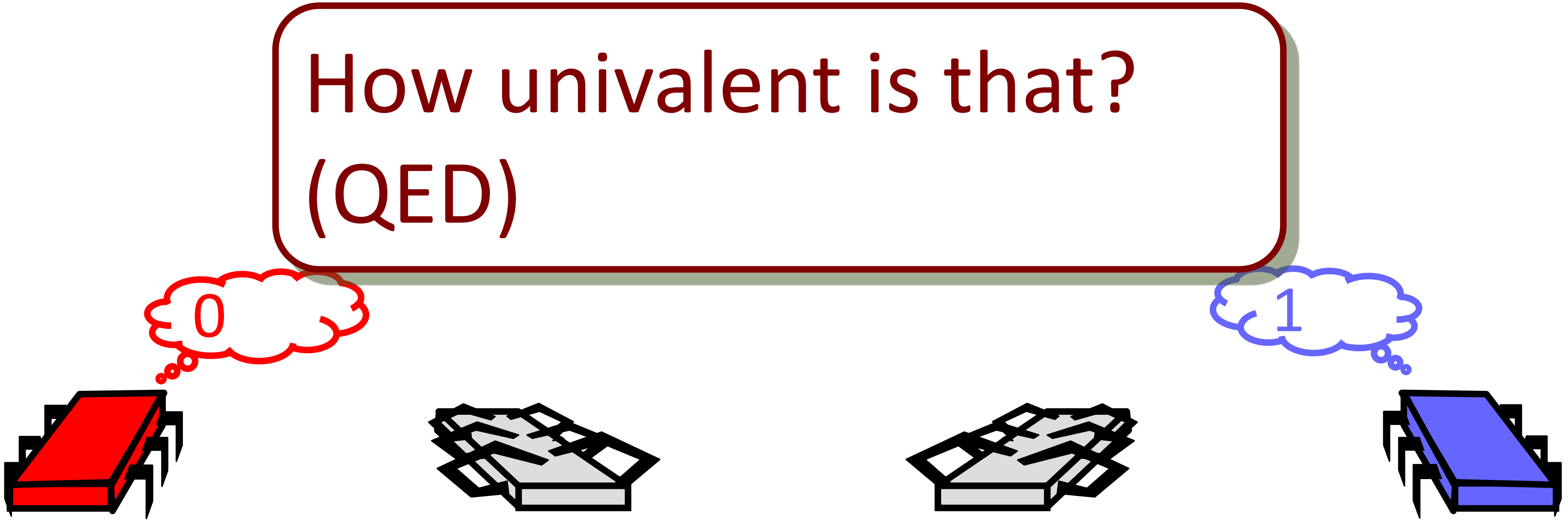


Include the solo execution by A that decides 0

The Possible Executions

**Also include the solo execution by B
which we know decides 1**

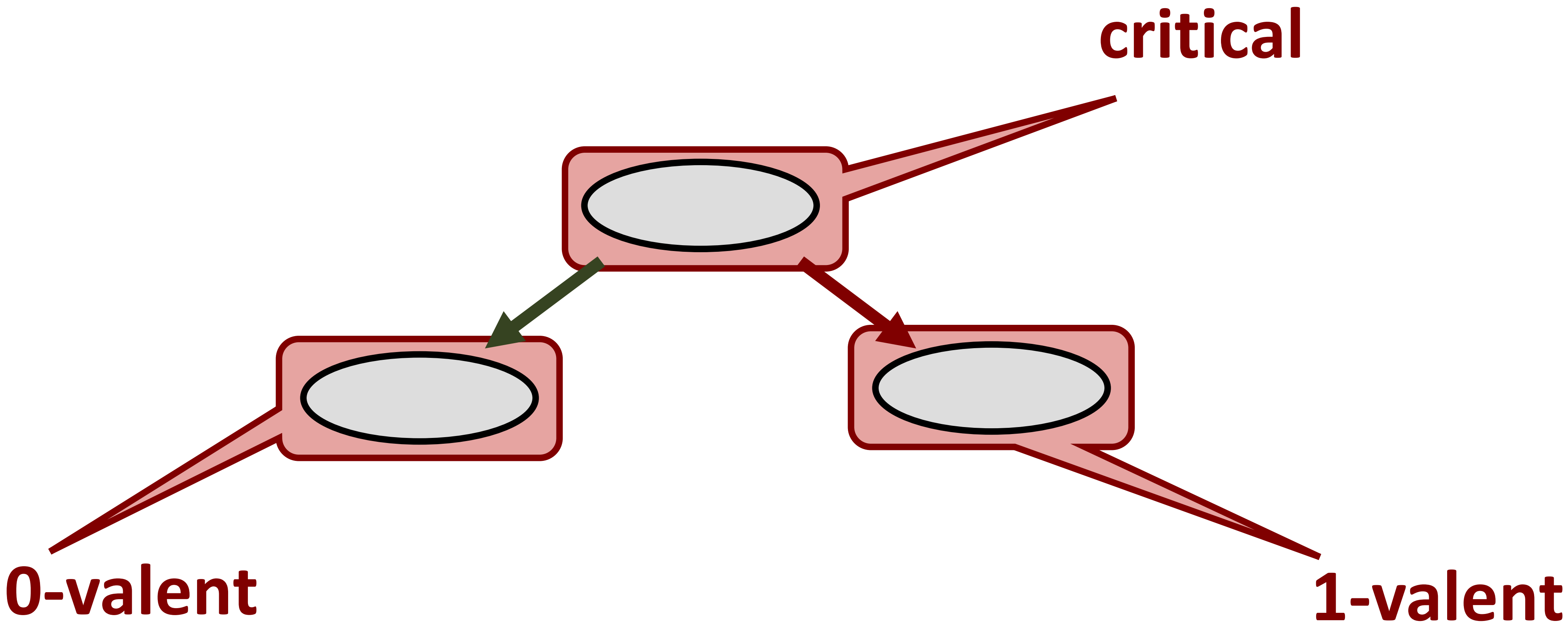
Possible Executions Include



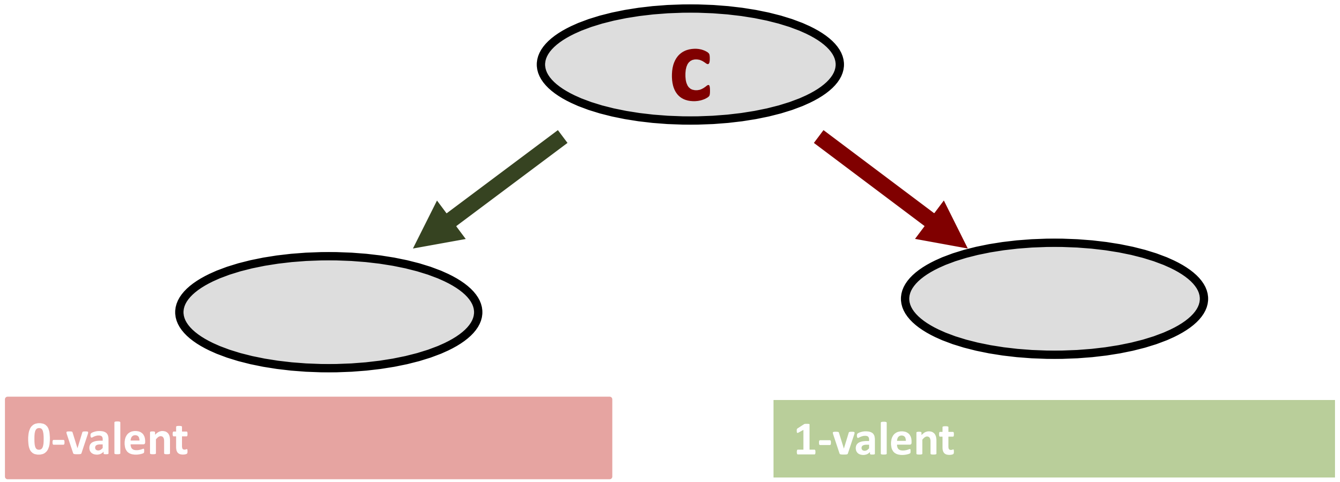
Solo execution by A must decide 0

Solo execution by B must decide 1

Critical States



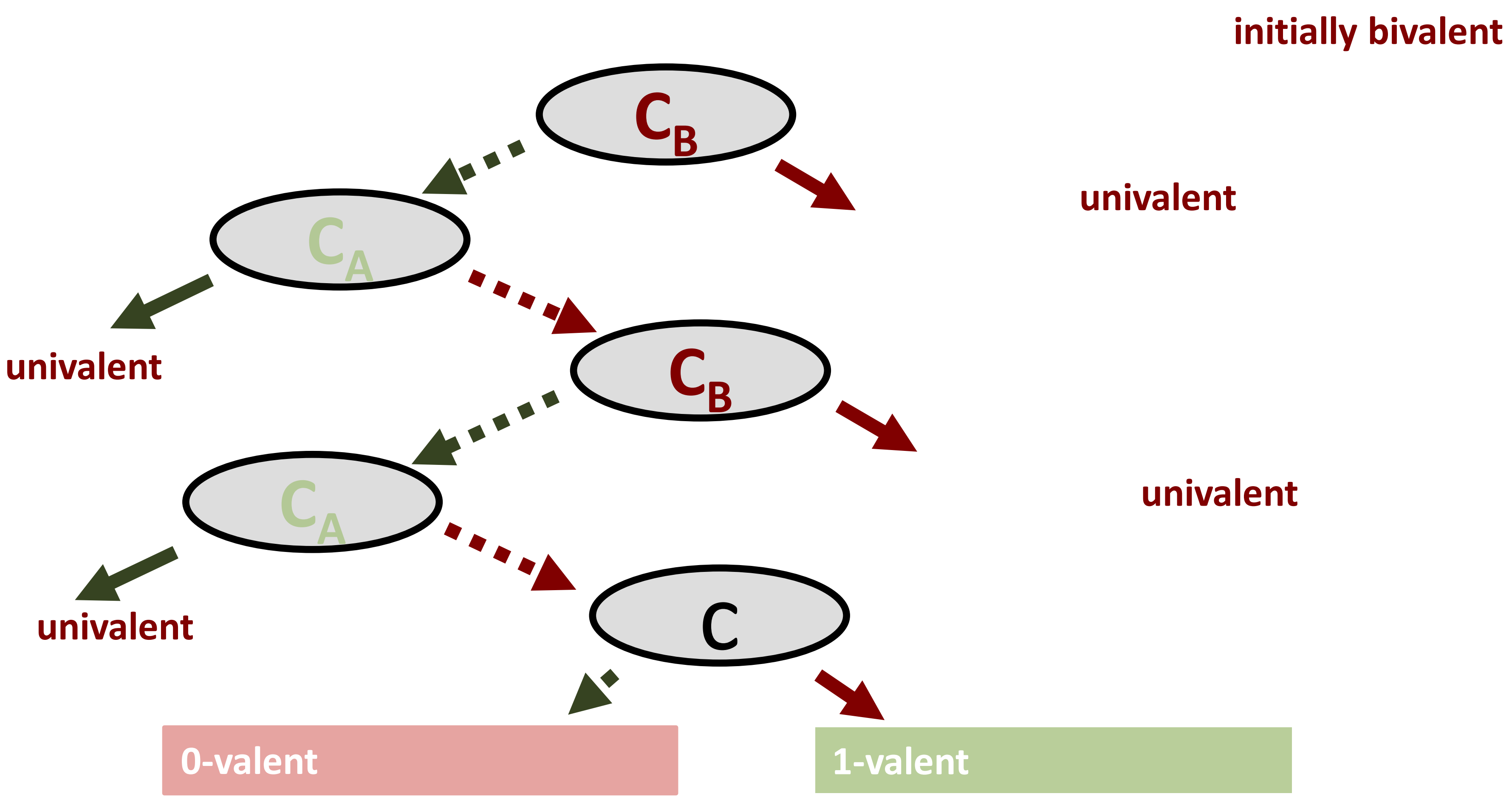
From a Critical State



If A goes first, protocol decides 0

If B goes first, protocol decides 1

Reaching Critical State



Critical States

Starting from a bivalent initial state

The protocol can reach a critical state

- Otherwise we could stay bivalent forever
- And the protocol is not wait-free

Model Dependency

So far, memory-independent!

True for

- Registers
- Message-passing
- Carrier pigeons
- Any kind of asynchronous computation

Read-Write Memory

- Reads and/or writes
- To same/different registers

Completing the Proof

- Lets look at executions that:
  - Start from a critical state
  - Threads cause state to become univalent by reading or writing to same/different registers
  - End within a finite number of steps deciding either 0 or 1
- Show this leads to a contradiction

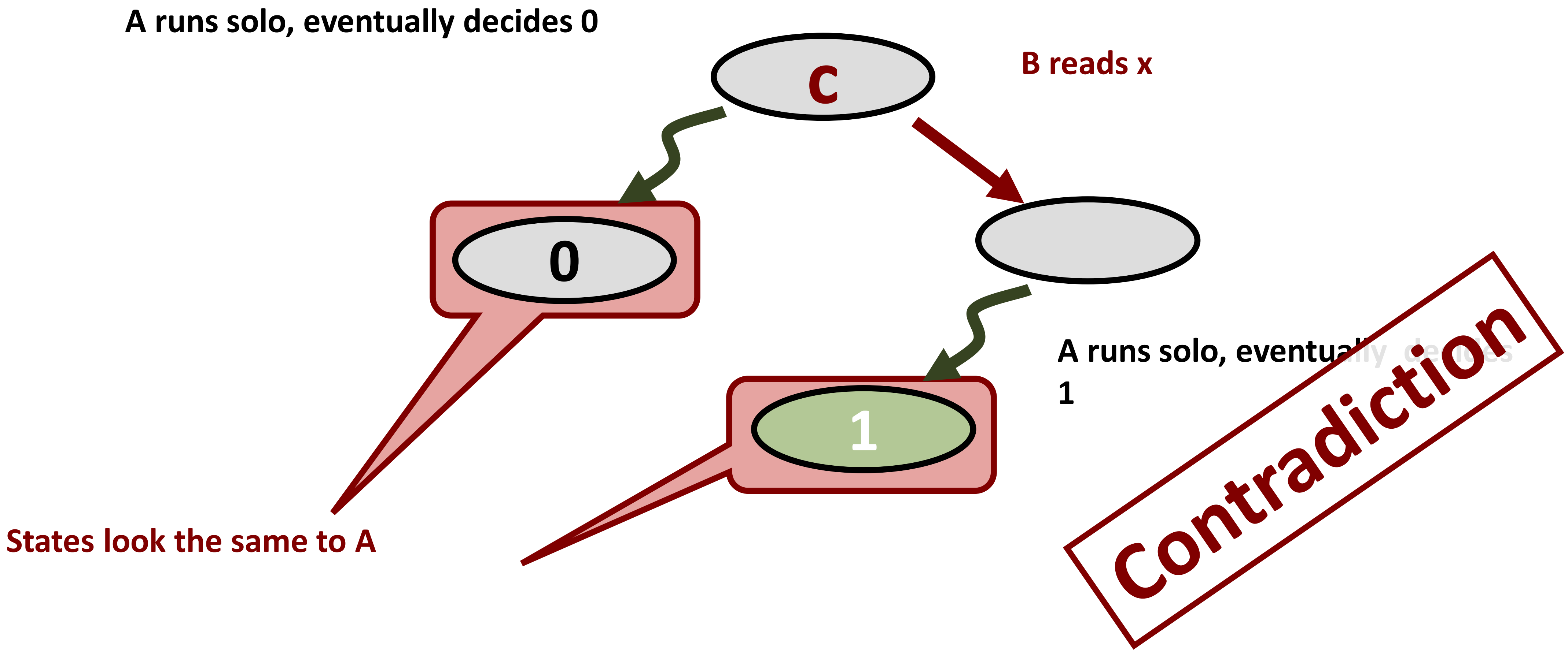
Possible Interactions

A reads x   A reads y

	x.read()	y.read()	x.write()	y.write()
x.read()	?	?	?	?
y.read()	?	?	?	?
x.write()	?	?	?	?
y.write()	?	?	?	?

A reads y, B writes y

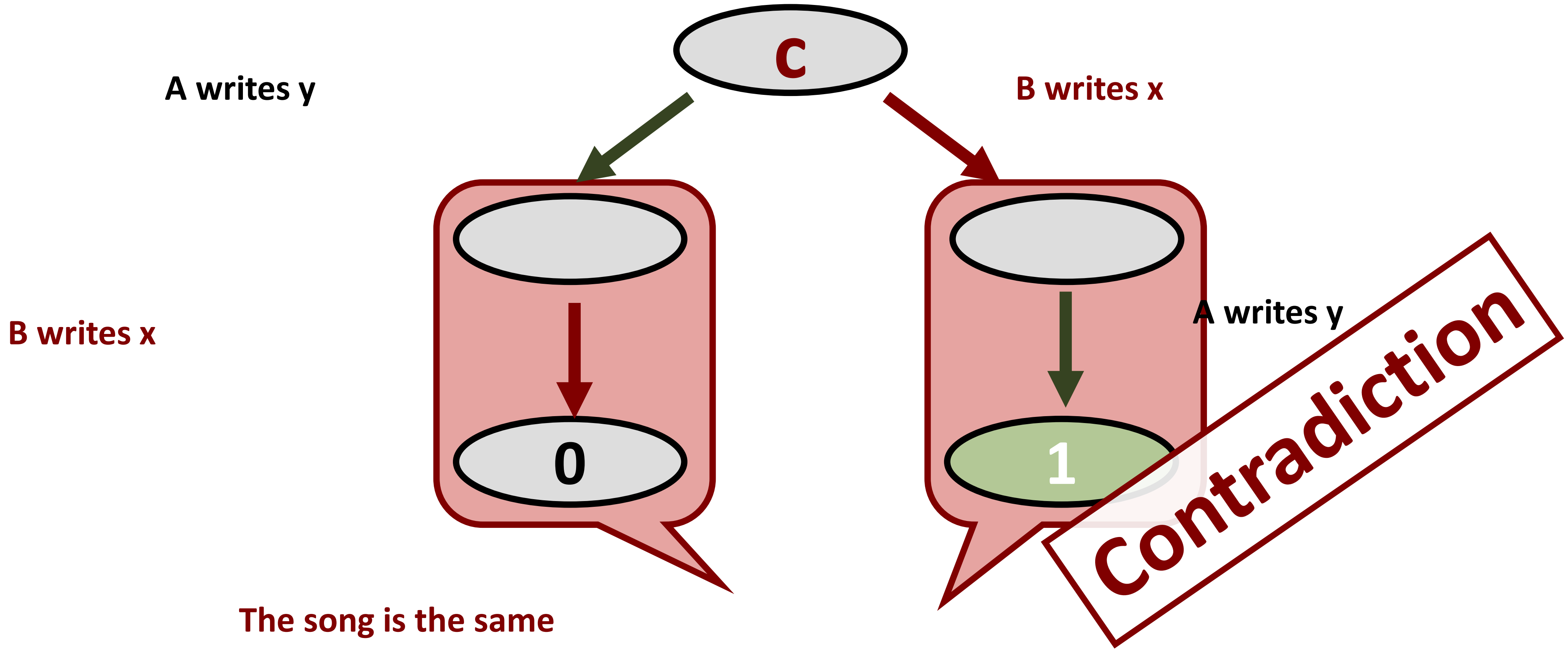
Some Thread Reads



Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	?
y.write()	no	no	?	?

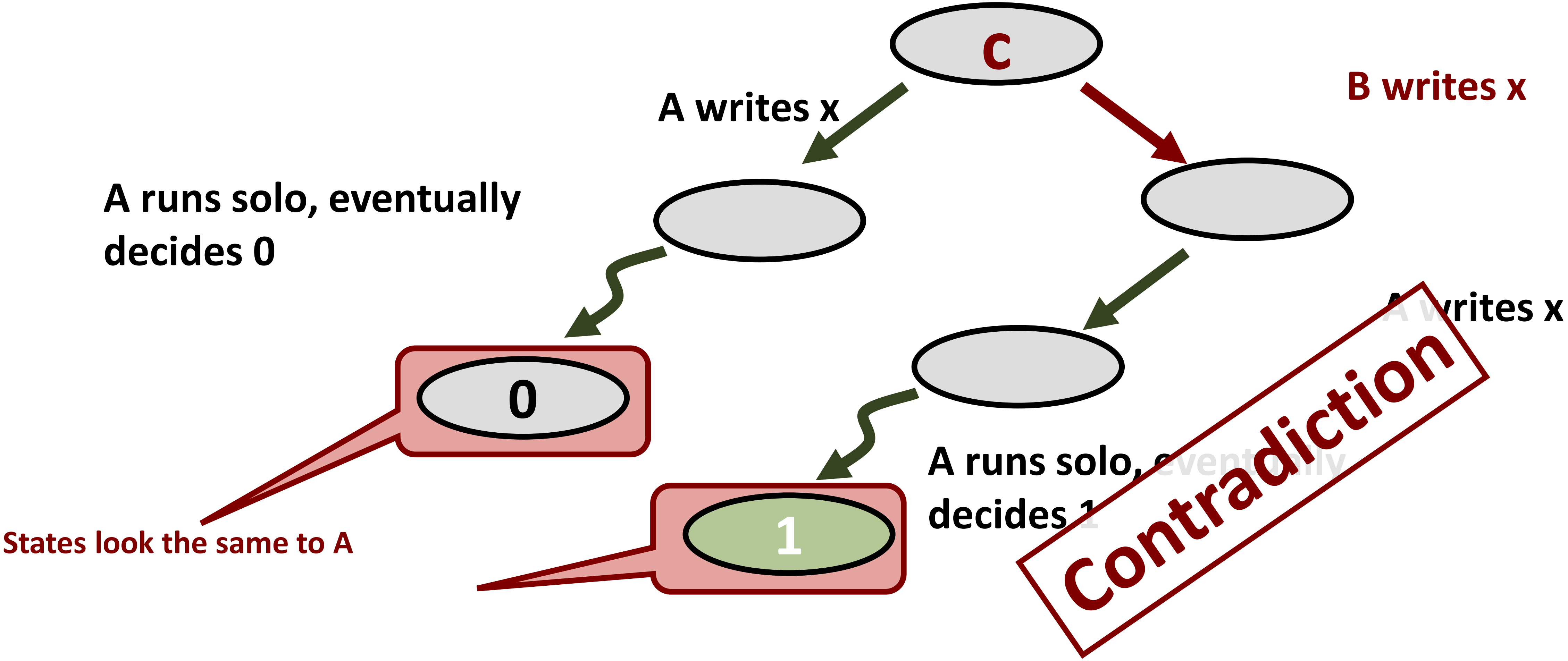
Writing Distinct Registers



Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	no
y.write()	no	no	no	?

Writing Same Registers



Proof complete.

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no

QED

Parts of this work are licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

SOME RIGHTS RESERVED

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution**. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Parts of the Material on these slides is based on Art of Multiprocessor Programming slides by Maurice Herlihy & Nir Shavit. License cf. above.