# Parallele Programmierung FS25

Exercise Session 13

Jonas Wetzel

# Plan für heute

- Organisation
- Nachbesprechung Assignment 12
- Theory
- Intro Assignment 13
- Exam questions
- Kahoot

# Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too

# Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too
- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9

# Organisation

- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9
- Falls ihr Feedback möchtet kommt bitte zu mir

# Organisation

- 2^64 ist nicht ungefähr gleich zu den Atomen im Universum!
- 2^64 ≈ 1.84×10^19
- 1mm^3 Sand hat 10^8 Atome, also ungefähr Anzahl Atome in 10 Sandkörnern ist 2^64
- Schätzungen für Anzahl Sandkörner auf der Erde ist 7.5 * 10^18, also ist 2^64 ungefähr das Doppelte davon

# Organisation

- Kahoot Allegations

# Organisation

- TA Award
- Danke!

# Organisation

- Wo sind wir jetzt?

Sequential Consistency and Linearizability

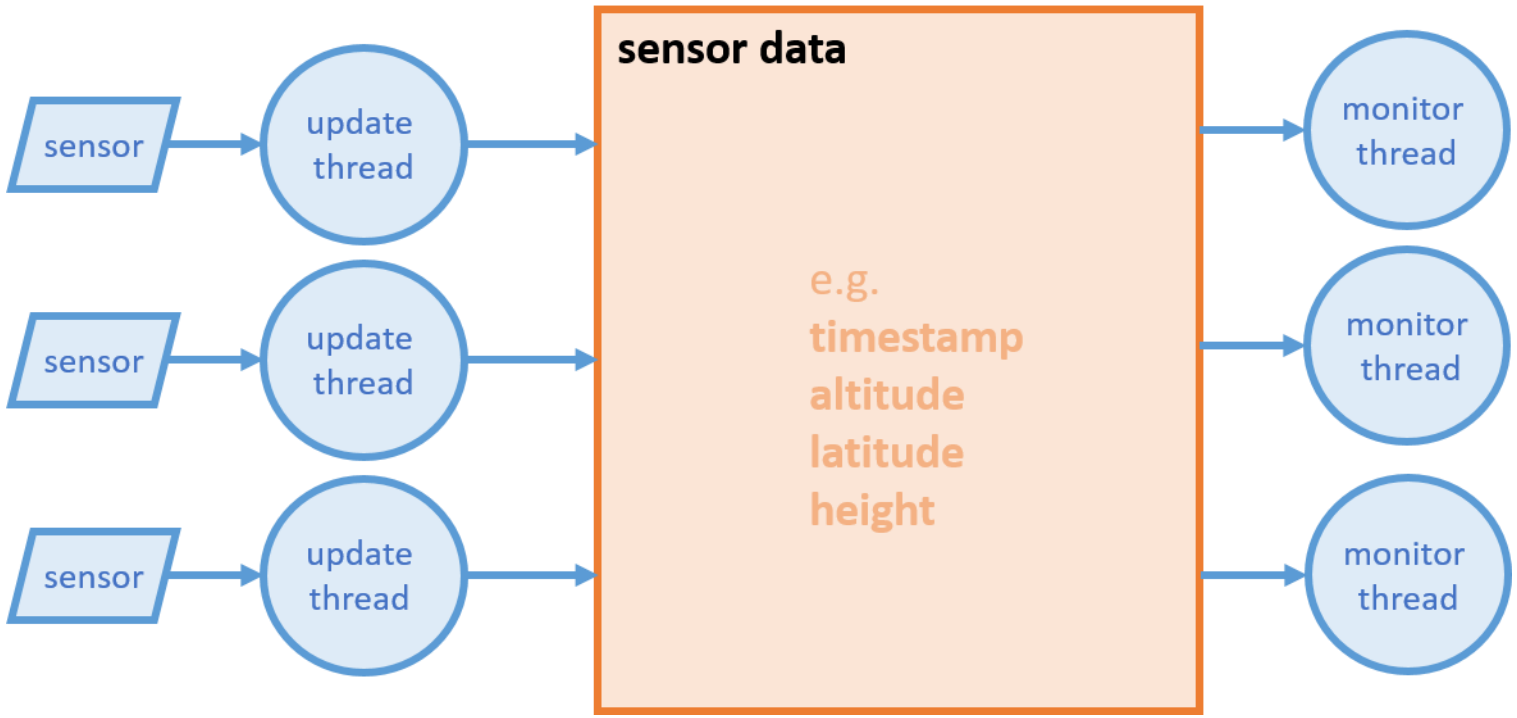
To come: Consensus

# Plan für heute

- Organisation
- **Nachbesprechung Assignment 12**
- Theory
- Intro Assignment 13
- Exam questions
- Kahoot

# Assignment 12

- Multisensor System.



# Multisensor System

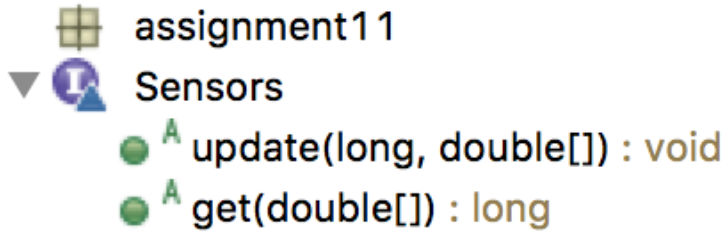
Implement two versions of the senor data set:
a) One blocking version based on a readers-writers lock (LockedSensors.java).
b) A lock-free version (LockFreeSensors.java)

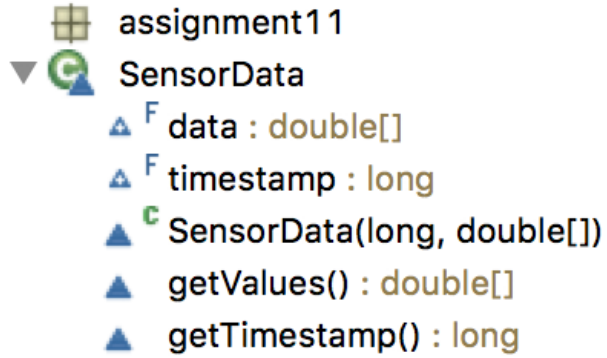
# Hint

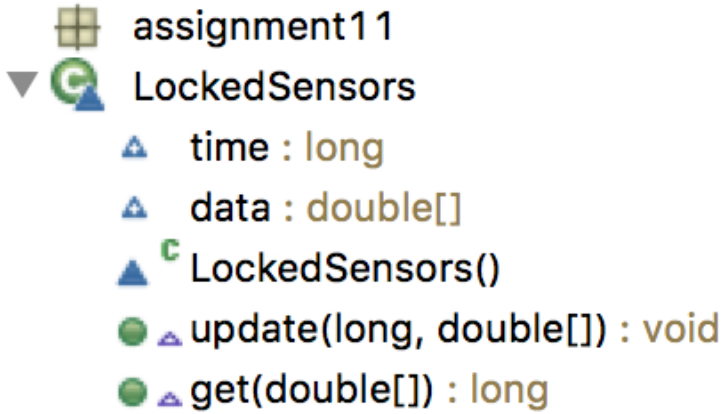
- `dataRef = new AtomicReference<SensorData>();`

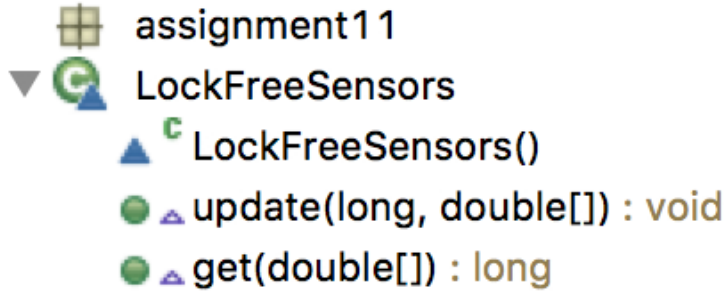
- AtomicReference allows for CAS on an object reference
- i.e. we can replace old object with new object in one step

# Multisensor System



# LockedSensors

```
class LockedSensors implements Sensors {

    long time = 0;
    double data[];

    private ReadWriteLock lock;
    private Lock readlock;
    private Lock writelock;

    LockedSensors() {
        this(new ReadWriteMonitorLock());
    }

    LockedSensors(ReadWriteLock l){
        time = 0;
        lock = l;
        readlock = lock.readLock();
        writelock = lock.writeLock();
    }
```

```
public long get(double val[])
{
    readlock.lock();
    try{
        if (time == 0)
            return 0;
        else{
            for (int i = 0; i<data.length; ++i)
                val[i] = data[i];
            return time;
        }
    }finally {
        readlock.unlock();
    }

}
```

```
public void update(long timestamp, double[] data)
{
    writelock.lock();
    try{
        if (timestamp > time) {
            if (this.data == null)
                this.data = new double[data.length];
            time = timestamp;
            for (int i=0; i<data.length;++i)
                this.data[i]= data[i];
            }
    }
    finally {
        writelock.unlock();
    }
}
```

# Lock implementation

```
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```
    private synchronized void aquireRead(){
        readersWating++;
        while(writers>0 || (writersWating>0 && readersToWait<=0)){
            try {
                wait();
            } catch (InterruptedException e) { e.printStackTrace(); }
        }
        readersWating--;
        readersToWait--;
        readers++;
    }
    private synchronized void releaseRead(){
        readers--;
        notifyAll();
    }
```

```
    private synchronized void aquireWrite(){
        writersWating++;
        while(writers>0 || readers>0 || readersToWait>0){
            try {
                wait();
            } catch (InterruptedException e) { e.printStackTrace(); }
        }
        writersWating--;
        writers++;
    }
    private synchronized void releaseWrite(){
        writers--;
        readersToWait = readersWating;
        notifyAll();
    }
```

# LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
         data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

```
public void update(long timestamp, double[] val)
{
    SensorData old_data;
    SensorData new_data = new SensorData(timestamp, val);
    do {
        old_data = data.get();
        if (old_data != null && old_data.getTimestamp() >= new_data.getTimestamp()) {
            return;
        }
    } while (!data.compareAndSet(old_data, new_data));

}
```

# LockFreeSensors

Is this wait free?

```
public void update(long timestamp, double[] val)
{
    SensorData old_data;
    SensorData new_data = new SensorData(timestamp, val);
    do {
        old_data = data.get();
        if (old_data != null && old_data.getTimestamp() >= new_data.getTimestamp()) {
            return;
        }
    } while (!data.compareAndSet(old_data, new_data));

}
```

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

# Plan für heute

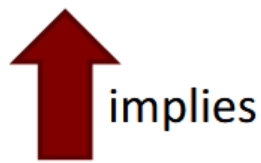
- Organisation
- Nachbesprechung Assignment 12
- **Theory**
- Intro Assignment 13
- Exam questions
- Kahoot

# Recap

# Lock-Free Programming

## Definitions for Lock-free Synchronisation

- L**ock-freedom**: at least one thread always makes progress even if other threads run concurrently.
  Implies system-wide progress but not freedom from starvation.




- W**ait-freedom**: all threads eventually make progress.
  Implies freedom from starvation.

# Lock-free algorithm

```
Object readSomething() {
    return atomicReference.get();
}

Void writeSomething(Object new_object) {
    Object old_object;
    do {
        old_object = atomicReference.get();
        // Check if we want to overwrite the latest data (i.e. only write newer or better data)
        if ( … ) {
            return;
        }
    } while (!atomicReference.compareAndSet(old_object, new_object));
}
```

## Progress conditions with and without locks

| | Non-blocking (no locks) | Blocking (locks) |
|---|---|---|
| Everyone makes progress | Wait-free | Starvation-free |
| Someone make progress | Lock-free | Deadlock-free |

# Implications

- Wait-free $\Longrightarrow$ Lock-free
- Wait-free $\Longrightarrow$ Starvation-free
- Lock-free $\Longrightarrow$ Deadlock-free
- Starvation-free $\Longrightarrow$ Deadlock-free
- Deadlock-free AND Fair $\Longrightarrow$ Starvation-free

A CS has to be Deadlock-free and mutually exclusive!

**Non-blocking algorithms**

Locks/blocking: a thread can indefinitely delay another thread

Non-blocking: failure or suspension of one thread cannot cause failure or suspension of another thread !

# Non-blocking counter

Deadlock/Starvation?

```
public class CasCounter {
    private AtomicInteger value;

    public int getVal() {
       return value.get();
    }

    // increment and return new value
    public int inc() {
       int v;
       do {
              v = value.get();
       } while (!value.compareAndSet(v, v+1));
       return v+1;
    }
}
```

What happens if some processes see the same value?

Assume one thread dies.
Does this affect other threads?

Why not “guarantees”?

Mechanism

(a) read current value v

(b) modify value v'

(c) try to set with CAS

(d) return if success
restart at (a) otherwise

Positive result of CAS of (c) *suggests* that no other thread has written between (a) and (c)

# Lock free data structures

# Disadvantages of locking

## Locks are pessimistic by design

- Assume the worst and enforce mutual exclusion

## Performance issues

- Overhead for each lock taken even in uncontended case
- Contended case leads to significant performance degradation
- Amdahl's law!

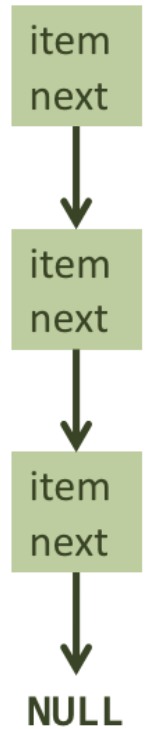
## Blocking semantics (wait until acquire lock)

- If a thread is delayed (e.g., scheduler) when in a critical section → all threads suffer
- What if a thread dies in the critical section
- Prone to deadlocks (and also livelocks)
- Without precautions, locks cannot be used in interrupt handlers

# So how do we build lock free data structures?

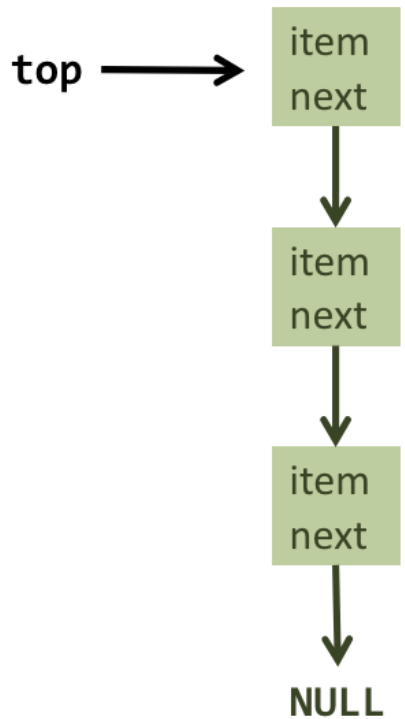
# Stack Node

```
public static class Node {
   public final Long item;
   public Node next;

   public Node(Long item) {
       this.item = item;
   }

   public Node(Long item, Node n) {
       this.item = item;
       next = n;
   }

}
```



# Non-blocking Stack

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

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

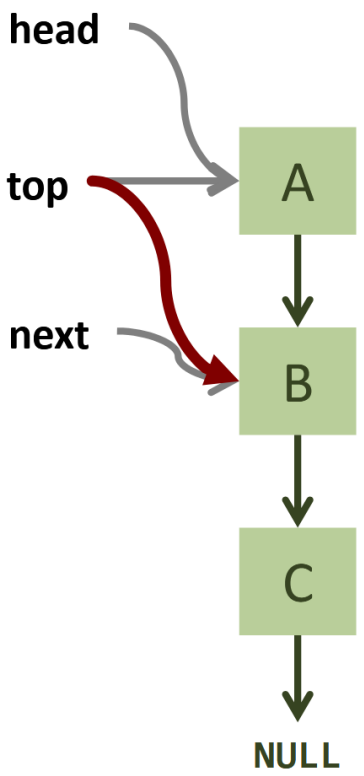


## pop

```
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

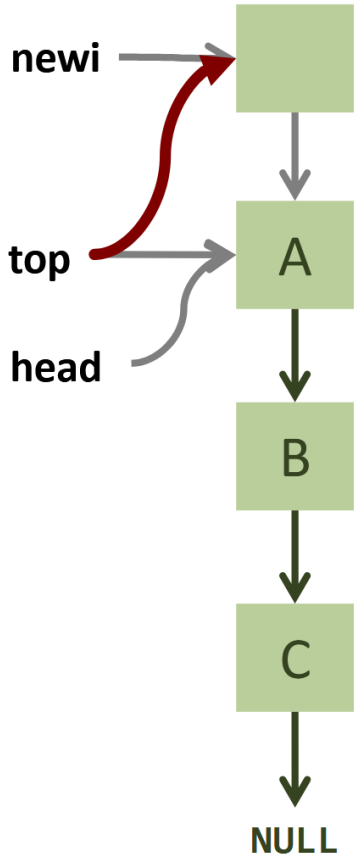


## push

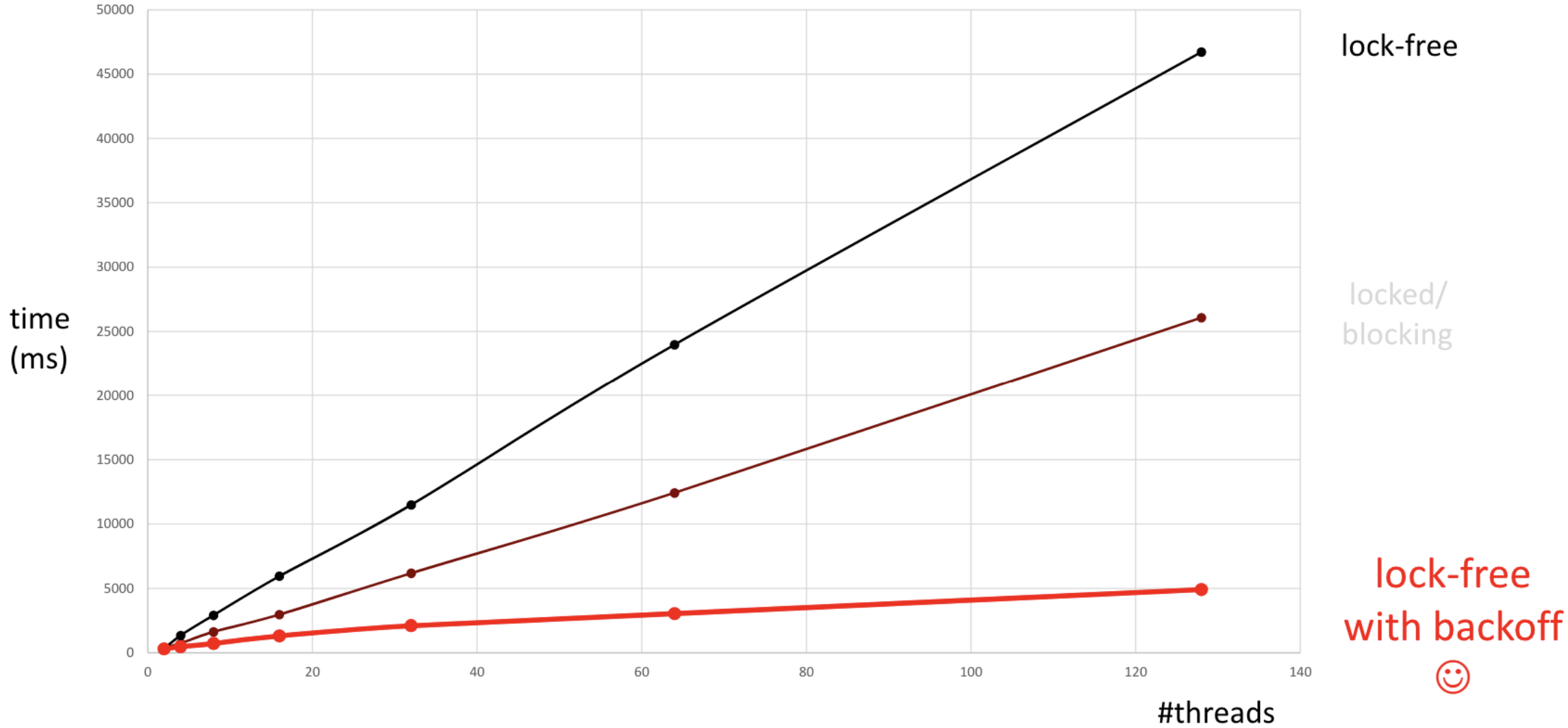
```
public void push(Long item) {
      Node newi = new Node(item);
      Node head;

      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



# With backoff



# Problems with this implementation?

- Say we want to use a node pool instead of always creating new nodes (i.e. not always use new Node() but instead take it out of a list)
- -> ABA Problem (exam relevant)

# Node reuse

Assume we do not want to allocate for each push and maintain a node pool instead (e.g., inside the OS or in an unmanaged language). 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();
   ...
}
```

Not the same get() as from Atomic!

# 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.

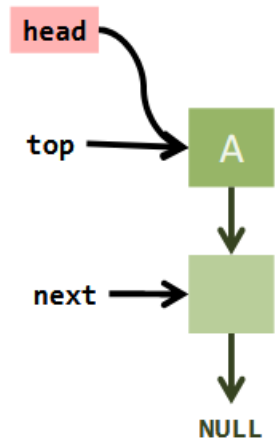
***So far* this works.**

# Using the node pool

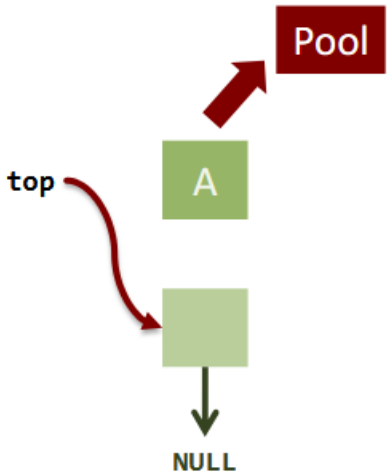
```
public void push(Long item) {
    Node head;
    Node new = pool.get(item);
    do {
        head = top.get();
        new.next = head;
    } while (!top.compareAndSet(head, new));
}

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;
}
```

# ABA Problem

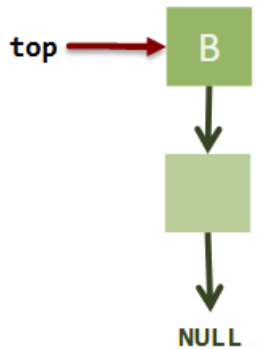
Thread X
in the middle
of pop: after read
but before CAS



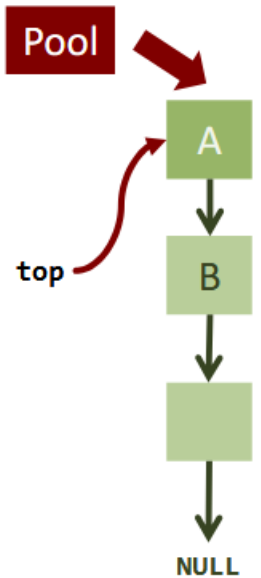

Thread Y
pops A



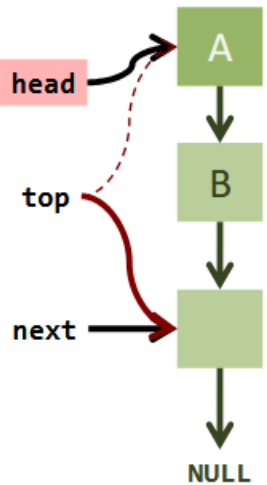

Thread Z
pushes B




Thread Z'
pushes A




Thread X
completes pop




time

```
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));
}
```

## 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)

# Hazard Pointers

# Hazard Pointers

- 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 thread X
- Modification takes place meanwhile by some other thread Y
- Thread X doesn’t realize that state changed and still performs operation

# Hazard Pointers

- Our idea to solve this, is that we introduce an array with n slots, where n is the number of threads
- Before X now reads P, it marks it as hazardous by entering it into the array (in slot assigned to thread X, i.e. ThreadID mod Arraysize)
- After the CAS, X removes P from the array
- If a process Y tries to reuse P, it first checks all entries of the hazard array, and, if it finds P in there, it simply requests a new pointer for use

# Hazard Pointers

- Examine the changed pop() method:

- We rely on garbage collection if we could run into problems, i.e., when we want to put something back into the pool that is hazardous

new pop()

```
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;
}
```

# Hazard Pointers

- The ABA problem also occurs on the node pool we are using
- We could make the pools thread-local. This does not help when push/pop operations aren’t well balanced within the thread
- Alternatively, we could just use Hazard pointers on the global node pool
- Previous Java code does not really improve performance in comparison to memory allocation and garbage collection, but it demonstrates how to solve the ABA problem

# Hazard Pointers

- Questions?

# SC and Linearizability

# How to define correctness of concurrent programs

- We can define different models
- Mostly just theoretical, CPU manufacturer decides on model

# Program correctness in a sequential world

Essenti
al

Objects encapsulate some representation of state

- We don’t reason about the representation directly, but about its visibility from outside (via public methods) (e.g. stack.top()==3)
- State must be consistent, i.e., according to the public class invariant (e.g., forall x. stack.push(x).pop()==x)
- Each method satisfies its post-condition, given its pre-condition
  - Hoare Tripel aus EProg

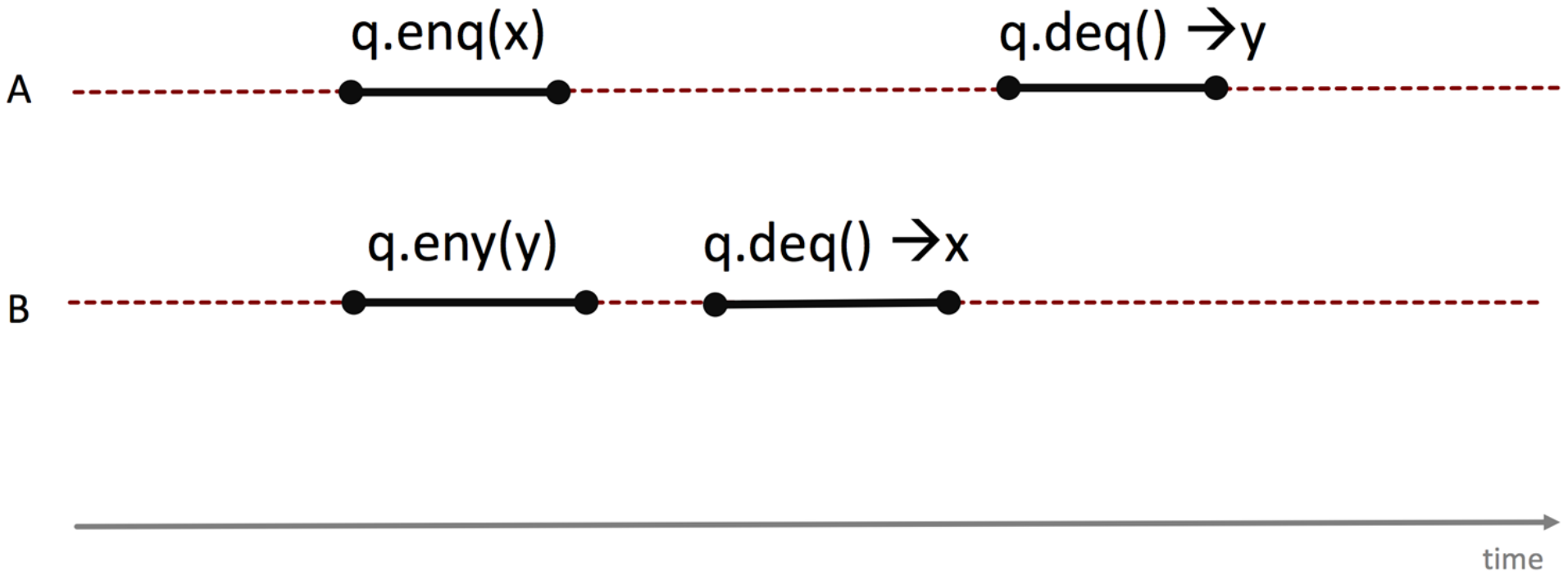
# Program correctness in a concurrent world

Essenti
al

| Sequential | Concurrent |
| --- | --- |
| Each method described independently. | Need to describe all possible interactions between methods. (what if enq and deq overlap? …) |
| Object's state is defined between method calls. | Because methods can overlap, the object may never be between method calls… |
| Adding new method does not affect older methods. | Need to think about all possible interactions with the new method. |

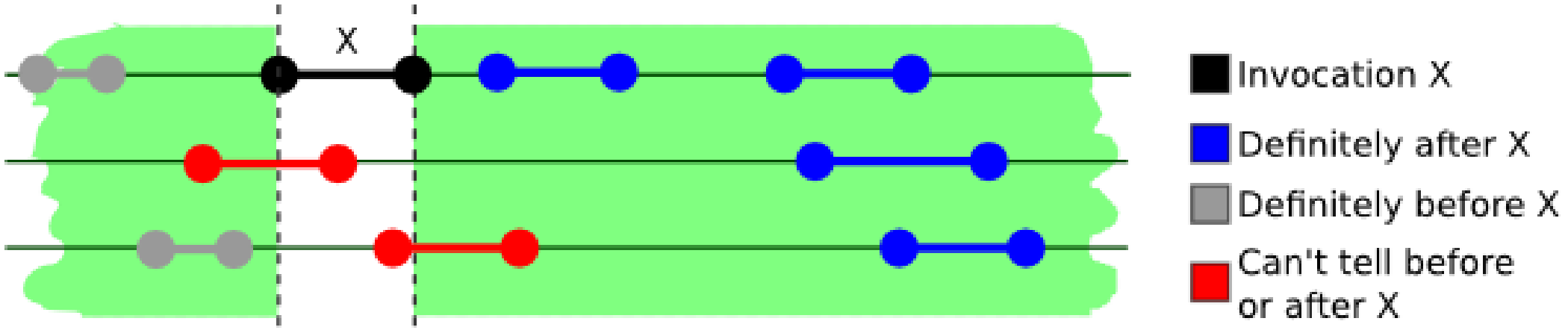
# Execution

Essenti
al



# Execution

Essenti
al



# Histories

A history is a series of invocations and responses of methods.

```
A:r.write(1)
B.r.write(2)
A:void
A:r.write(3)
B:void
B:r.read()
B:1
A.void
```

# Histories

Histories clan be categorized by some fundamental properties:

Sequential
Complete
Equivalence to some other History
Legal
Well formed
Quiescent Consistent
Sequentially Consistent
Linearizable

# Sequential

No interleaving at all.
First event is an invocation.
Each invocation is immediately followed by a response.

A: r.write(1)
A: void
B: r.read()
B: 1

# Complete

No pending invocations at the end

Not complete:
A: r.write(1)

Complete:
A: r.write(1)
A: void

# What are projections?

**We write H|A and to say:**

**All events in H by thread A**

# What are projections?

**We write H|q and to say:**

**All events in H on object q**

# Equivalence?

**Histories H1 and H2 are equivalent if their per-thread projections are the same.**

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

# Legal

For all objects o: H|o is sequential and correct

Correct in the sense of the object specification

# Well formed

For all threads t: H|t is sequential

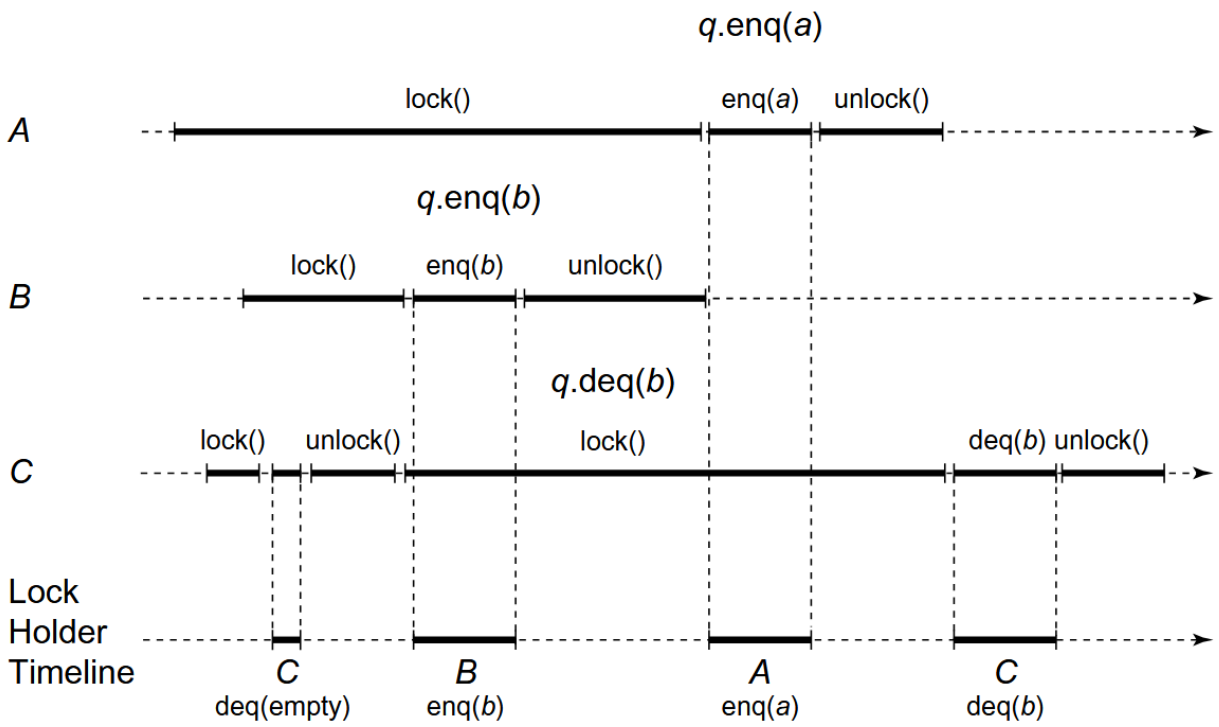
- A system can be …
  - Quiescent consistent
  - Sequentially consistent
  - Linearizable
- And many more, it’s up to us to define

# Example: Queue

- What does it mean for a concurrent object to be correct?
- each method accesses and updates fields while holding an exclusive lock
- method calls take effect sequentially

```
1  class LockBasedQueue<T> {
2    int head, tail;
3    T[] items;
4    Lock lock;
5    public LockBasedQueue(int capacity) {
6      head = 0; tail = 0;
7      lock = new ReentrantLock();
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) throws FullException {
11     lock.lock();
12     try {
13       if (tail - head == items.length)
14         throw new FullException();
15       items[tail % items.length] = x;
16        tail++;
17     }  finally {
18       lock.unlock();
19     }
20   }
21   public T deq() throws EmptyException {
22     lock.lock();
23     try {
24       if (tail == head)
25         throw new EmptyException();
26       T x = items[head % items.length];
27       head++;
28       return x;
29     } finally {
30       lock.unlock();
31     }
32   }
33 }
```

# Example: Queue




```
1  class LockBasedQueue<T> {
2    int head, tail;
3    T[] items;
4    Lock lock;
5    public LockBasedQueue(int capacity) {
6      head = 0; tail = 0;
7      lock = new ReentrantLock();
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) throws FullException {
11     lock.lock();
12     try {
13       if (tail - head == items.length)
14         throw new FullException();
15       items[tail % items.length] = x;
16        tail++;
17     }  finally {
18       lock.unlock();
19     }
20   }
21   public T deq() throws EmptyException {
22     lock.lock();
23     try {
24       if (tail == head)
25         throw new EmptyException();
26       T x = items[head % items.length];
27       head++;
28       return x;
29     } finally {
30       lock.unlock();
31     }
32   }
33 }
```

# Alternative concurrent queue implementation

- queue is correct only if it is shared by a single enqueuer and a single dequeuer
- It has almost the same internal representation as the lock-based queue
- only difference is the absence of a lock

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

# How to reason about concurrent objects that have no locks?

- objects whose methods hold exclusive locks are less desirable than ones with finer-grained locking or no locks
- We therefore need a way to specify the behavior of concurrent objects, and to reason about their implementations, without relying on method-level locking
- lock-based queue example illustrates a useful principle: it is easier to reason about concurrent objects if we can somehow map their concurrent executions to sequential ones, and limit our reasoning to these sequential executions

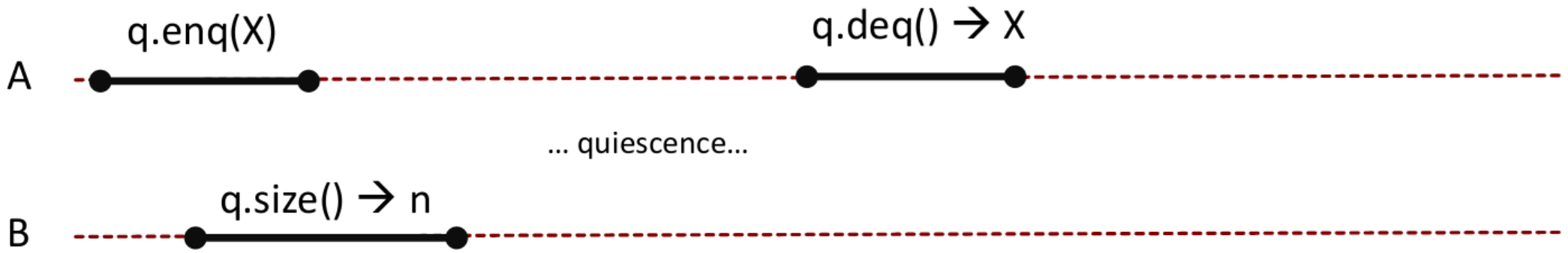
# Quiescent Consistency

# Quiescent consistency

- Method calls should appear to happen in a one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real-time order

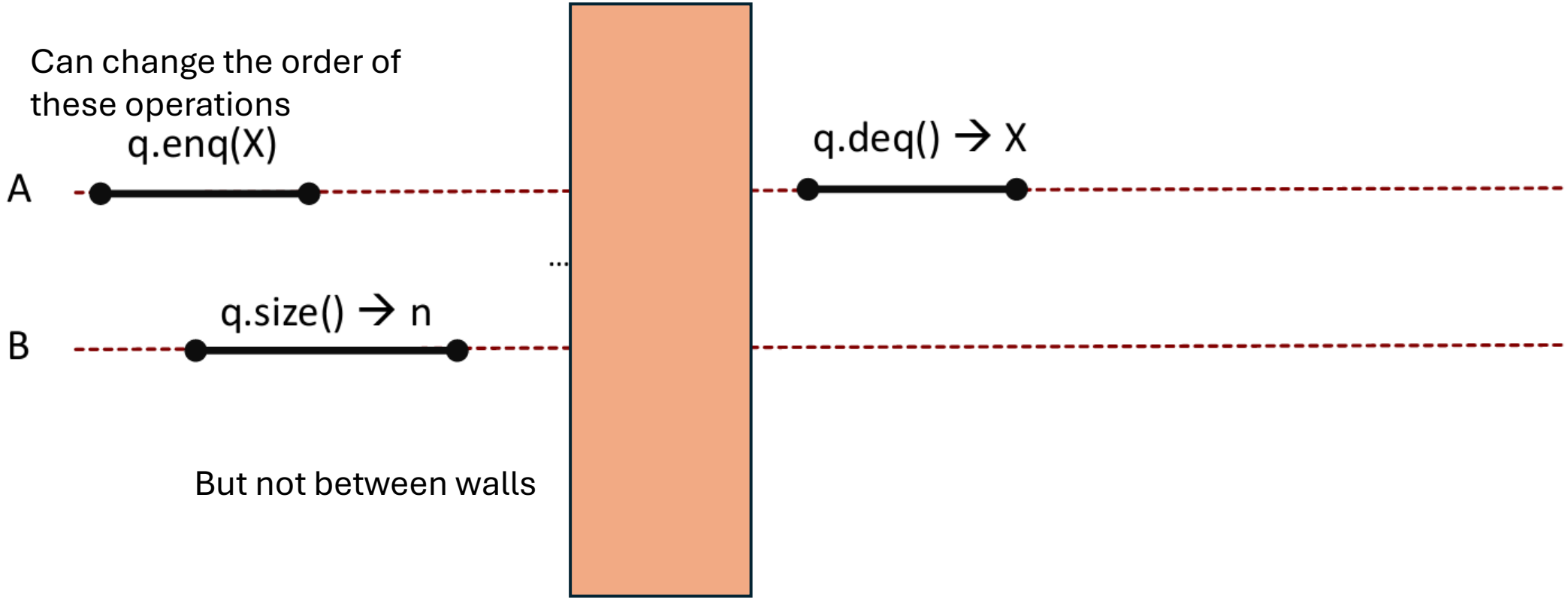
# Quiescent consistency

requires non-overlapping operations to appear to take effect in their real-time order, but overlapping operations might be reordered



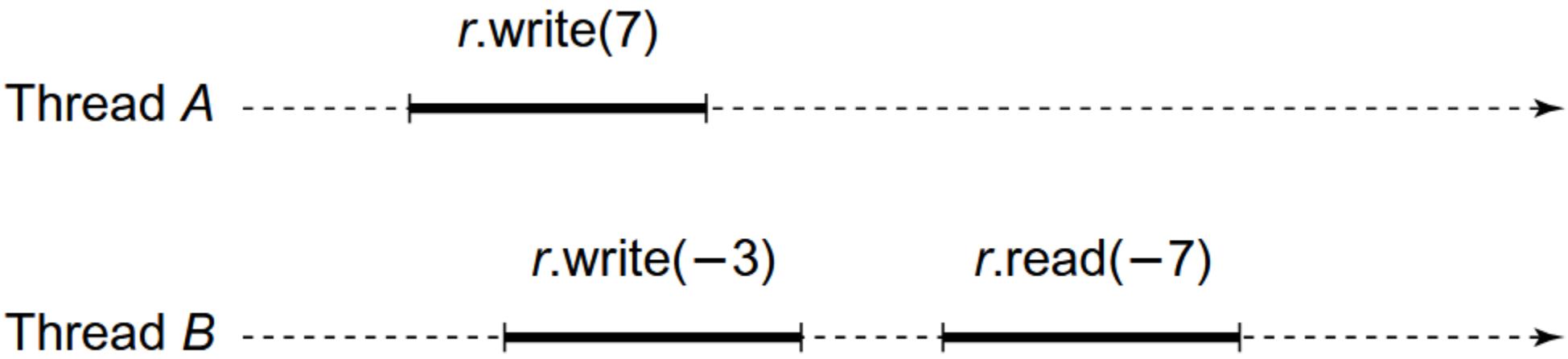
# Quiescent consistency

requires non-overlapping operations to appear to take effect in their real-time order, but overlapping operations might be reordered



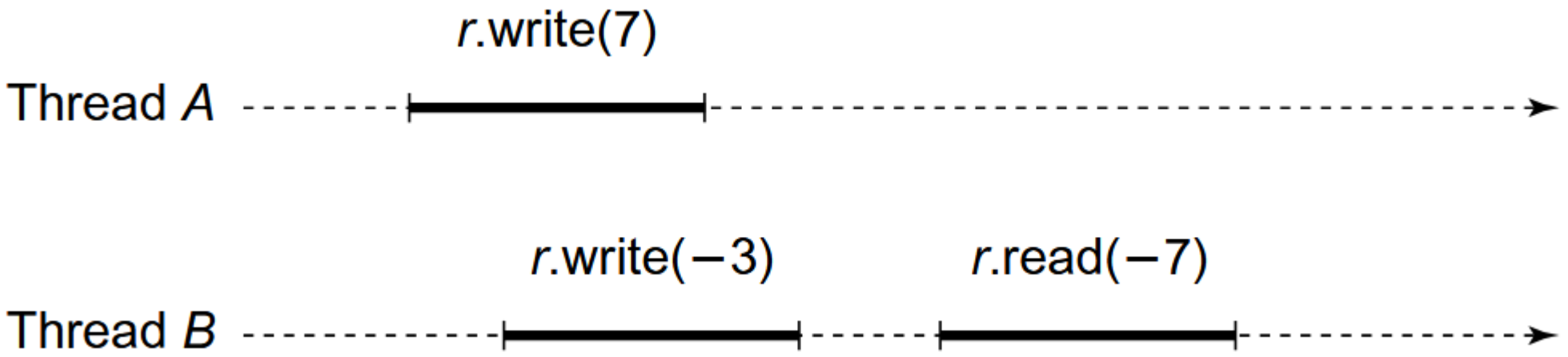
# Sequential Consistency

# Motivation




- two threads concurrently write −3 and 7 to a shared register r
- Later, one thread reads r and returns the value −7
- This behavior is clearly not acceptable!
- We expect to find either 7 or −3 in the register, not a mixture of both!

# Motivation




- two threads concurrently write −3 and 7 to a shared register r
- Later, one thread reads r and returns the value −7
- This behavior is clearly not acceptable
- We expect to find either 7 or −3 in the register, not a mixture of both!
- Method calls should appear to happen in a one-at-a-time, sequential order!

# Motivation

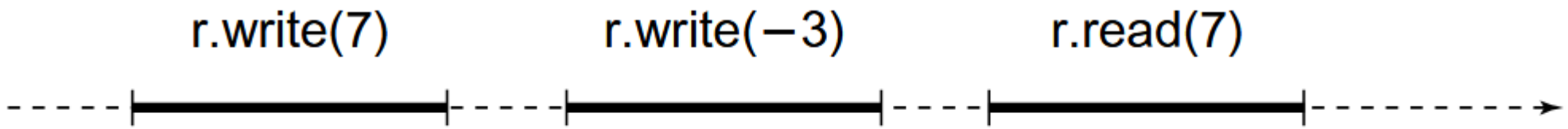



**Figure 3.5** Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote.

- single thread writes 7 and then −3 to a shared register r. Later, it reads r and returns 7
- For some applications, this behavior might not be acceptable because the value the thread read is not the last value it wrote

# Motivation

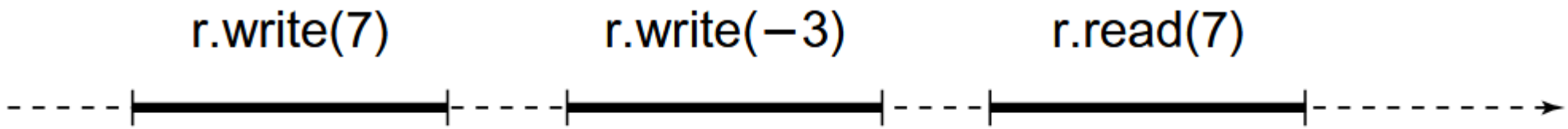



**Figure 3.5** Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote.

- We want Method calls to appear to take effect in program order!

# Combining both we get SC

- Method calls should appear to happen in a one-at-a-time, sequential order
- Method calls should appear to take effect in program order

# Combining both we get SC

- Method calls should appear to happen in a one-at-a-time, sequential order
- Method calls should appear to take effect in program order

- That is, in any concurrent execution, there is a way to order the method calls sequentially so that they (1) are consistent with program order, and (2) **meet the object's sequential specification**

# Sequential consistency

A multiprocessing system has sequential consistency if:

"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

- Leslie Lamport (inventor of sequential consistency, GOAT Turing Award Winner, concurrent master mind)

# In other words

- A History *H* is **sequential**, if there are no overlapping methods (when every invocation is immediately followed by the matching response)

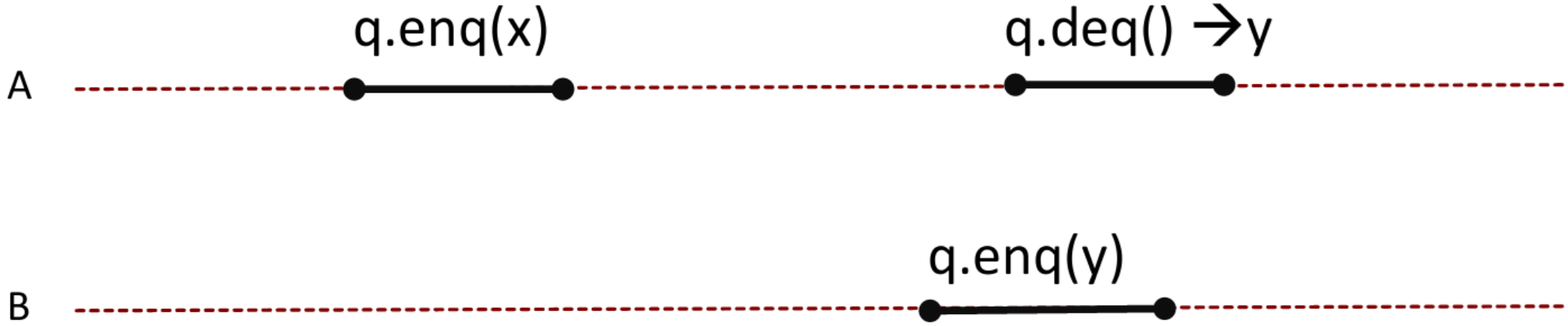
- A History is *SC (Sequentially Consistent)*, if:
  1. Every Thread projection is a sequential history
  2. Method calls appear to follow **PO** (Program Order), which allows for "reordering" of method-calls if they follow the ordering determined by the corresponding Thread-projection
  - Thread projection: we write H|A to say: All events in H by thread A

# Sequential consistency requirements

Essenti
al

1. All instructions are executed in order.

2. Every write operation becomes instantaneously visible throughout the system.

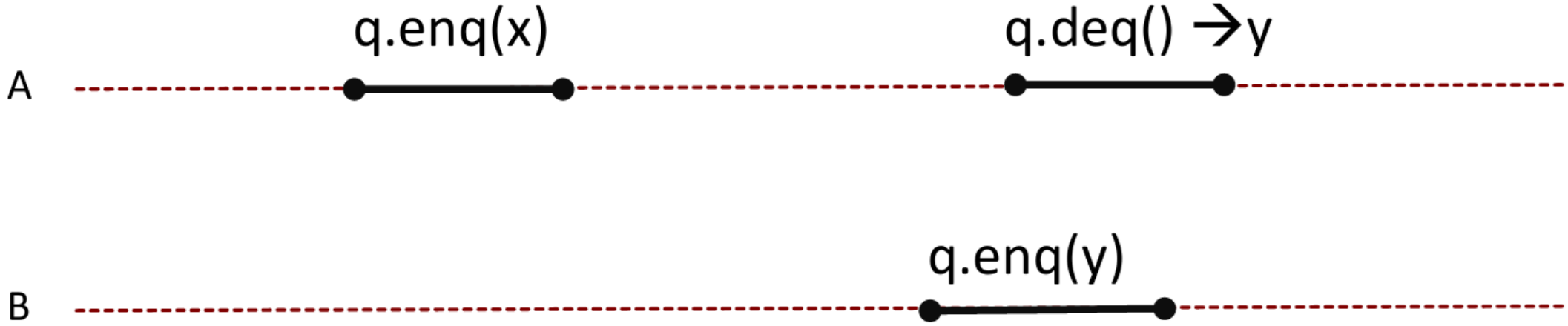


# Sequential consistency requirements

Essenti
al

1. All instructions are executed in order.

2. Every write operation becomes instantaneously visible throughout the system. (all variables volatile! Shows us that standard java is not sequentially consistent)



# Sequential consistency and the real world

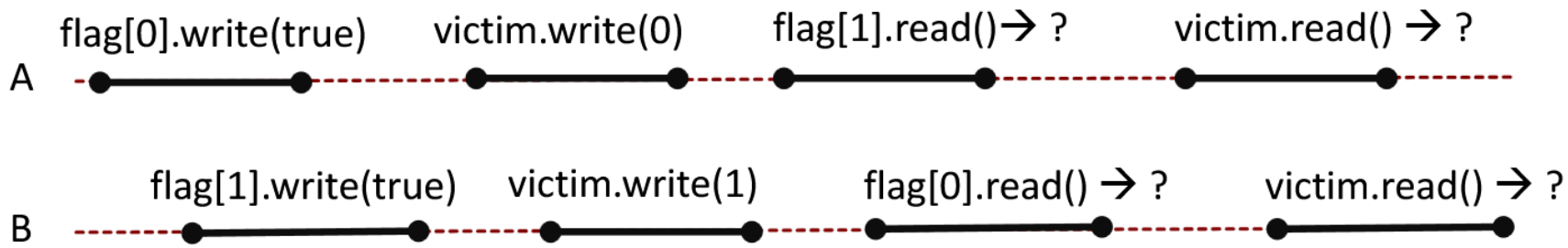
- In the real world, hardware architects do not adhere to this by default
- We need to explicitly announce that we want this property (i.e. volatile keyword)

# Sequential consistency and the real world

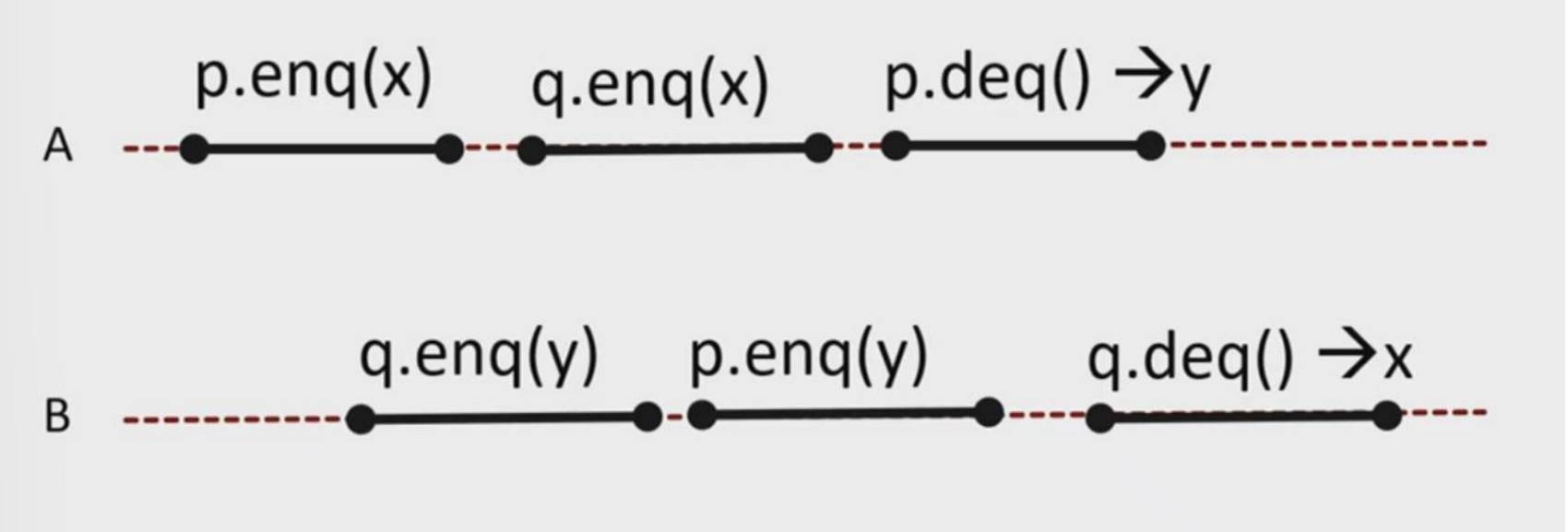
- We need to explicitly announce that we want this property (i.e. volatile keyword)
- This lock is only correct if we have SC

**Reminder: Consequence for Peterson Lock (Flag Principle)**

```
flag[id] = true;
victim = id;
while (flag[1-id] && victim == id);
```



# SC is not compositable




H|p is sequentially consistent
H|q is sequentially consistent
H is not sequentially consistent

In general: sequential consistency is not compositable

# Sequential consistency vs Quiescent consistency

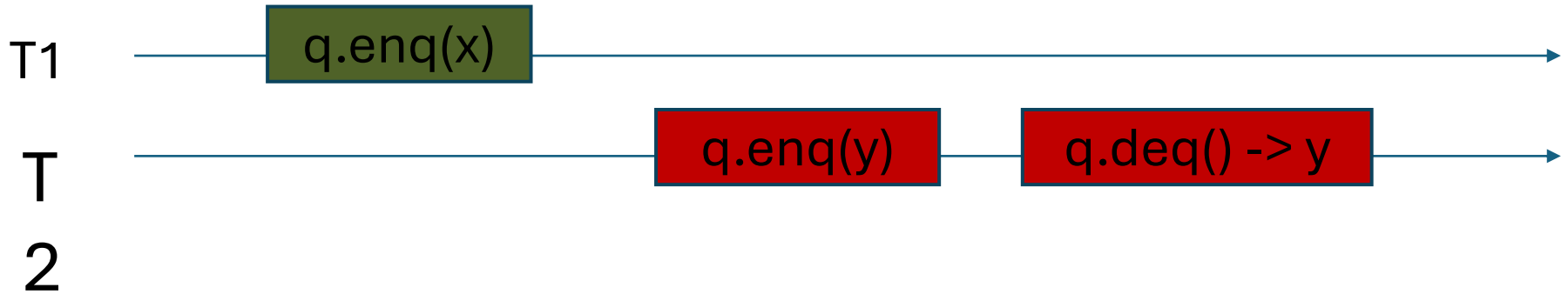
sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa

# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

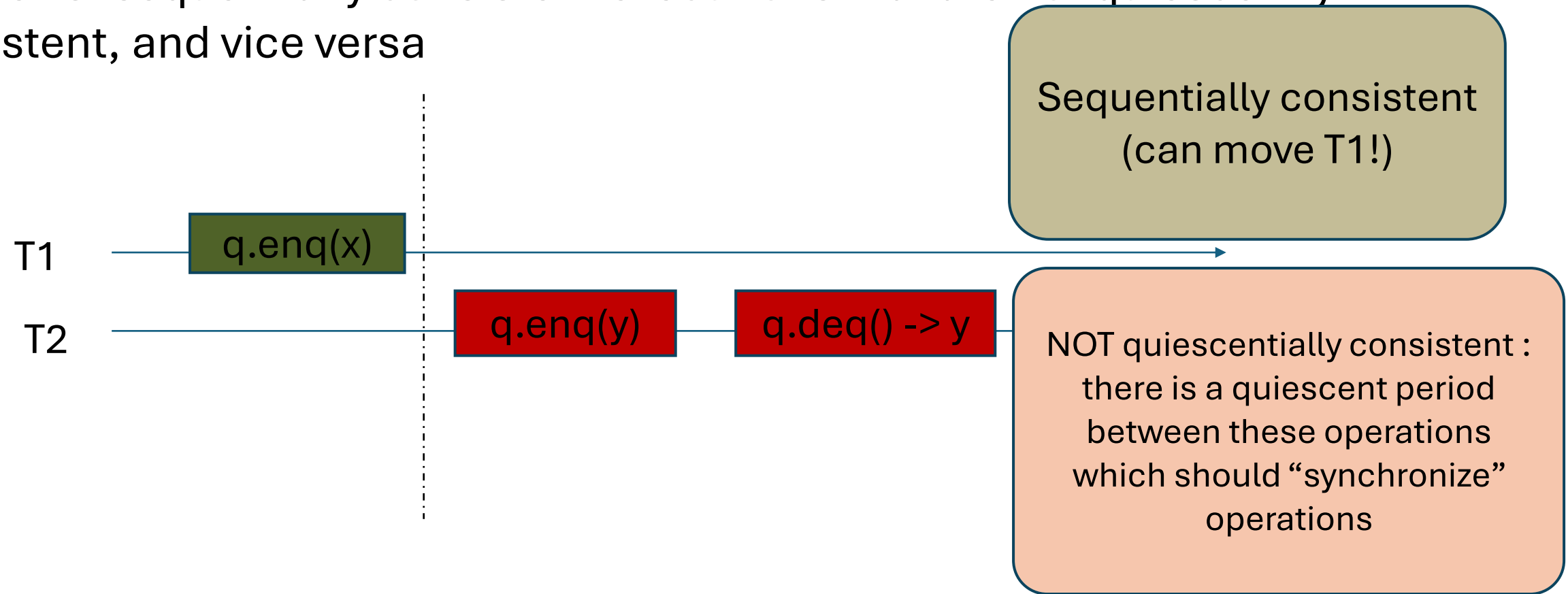
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

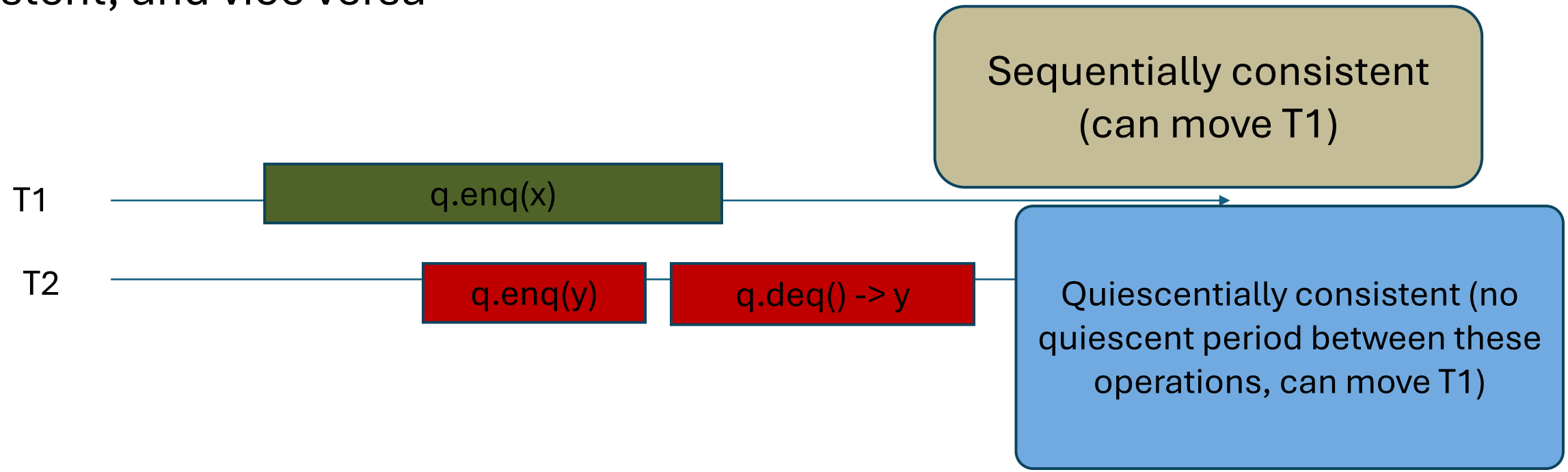
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa



# Sequential consistency vs Quiescent consistency

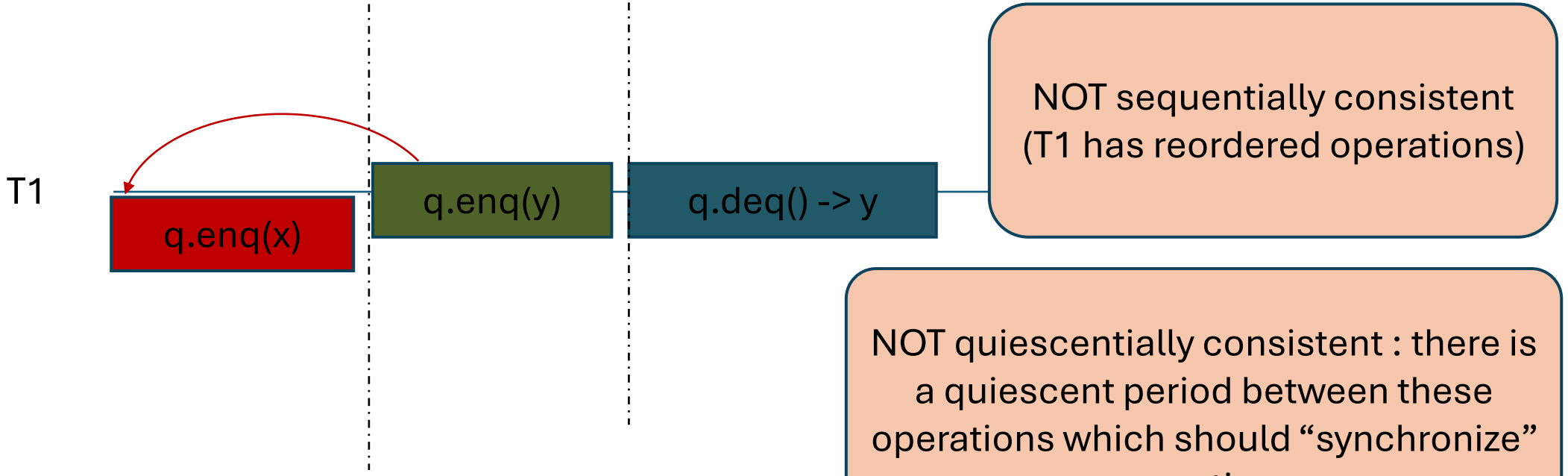
sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa

# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

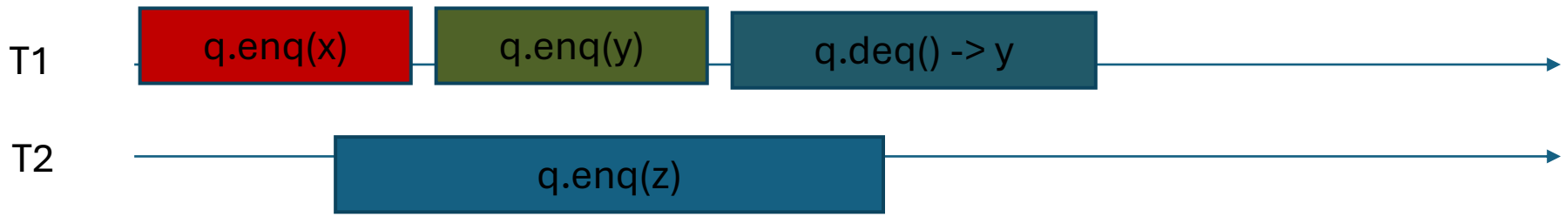
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

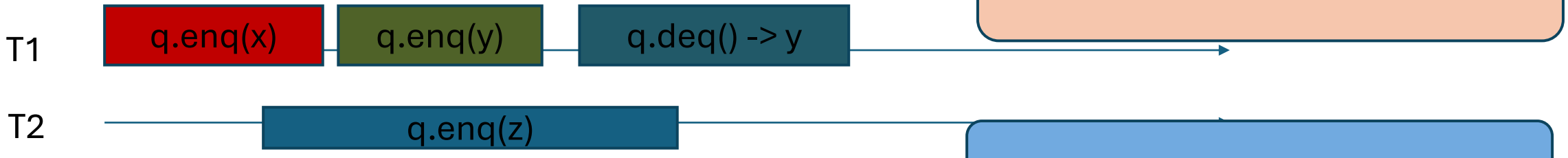
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



# Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa

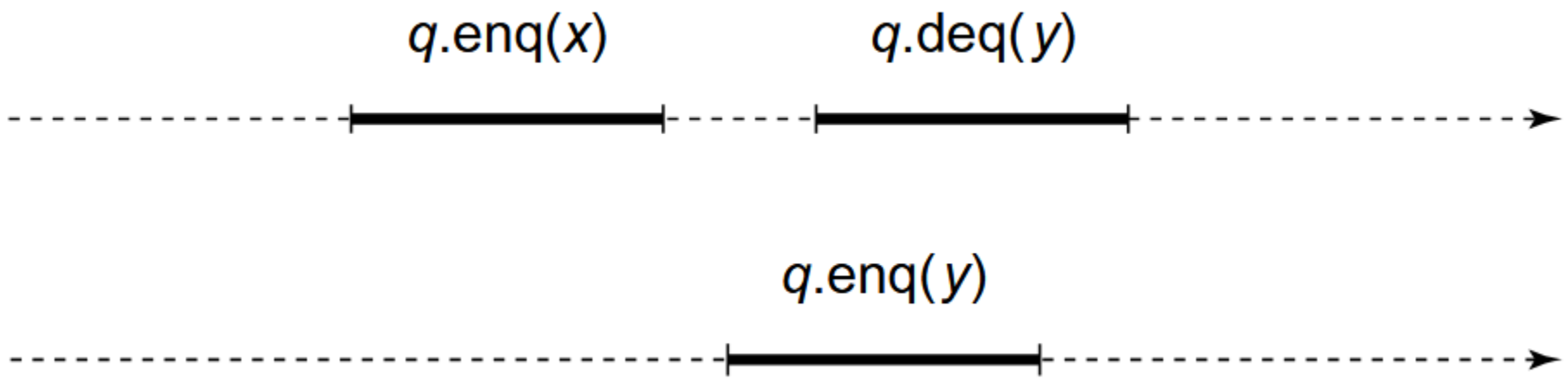



NOT sequentially consistent (T1 has reordered operations)

Quiescentially consistent (no quiescent period between these operations, all is good)

# Linearizability

# Motivation




- This is SC
- Goes against our intuition, q.enq(x) finished before q.enq(y)!

# How do we fix this

- replace the requirement that method calls appear to happen in program order with the following stronger restriction:
- Each method call should appear to take effect instantaneously at some moment between its invocation and response

# Idea

- Now we have:

- Method calls should appear to happen in a one-at-a-time, sequential order
- Each method call should appear to take effect instantaneously at some moment between its invocation and response

# Consistency model: Linearizability

Essenti
al

- Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously between its invocation and its response.

# Consistency model: Linearizability

Essenti
al

- Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously between its invocation and its response.

- An object for which this is true for all possible executions is called **linearizable**
- Has nice properties like composability

# In other words

- If given parallel/concurrent execution is equal to some sequential history, where a preceding method call shows effect before the later one
- overlapping method calls can be "reordered" as wished
- reordering means, that if $m_1$ and $m_2$ overlap (independent of which methods invocation or response was first/second) we can choose, if $m_1$ or $m_2$ shows its effect first

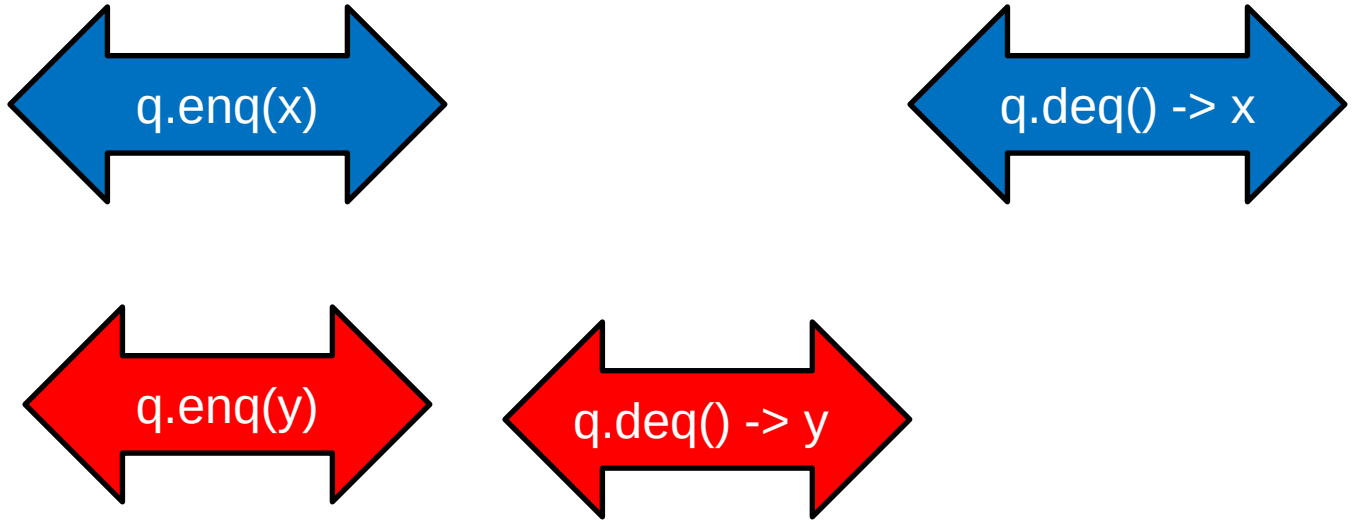
# In other words

- If given parallel/concurrent execution is equal to some sequential history, where a preceding method call shows effect before the later one
- overlapping method calls can be "reordered" as wished
- reordering means, that if $m_1$ and $m_2$ overlap (independent of which methods invocation or response was first/second) we can choose, if $m_1$ or $m_2$ shows its effect first

- Difference to SC: we can’t reorder operations even if they follow thread projection

# Example with FIFO Queue (1)

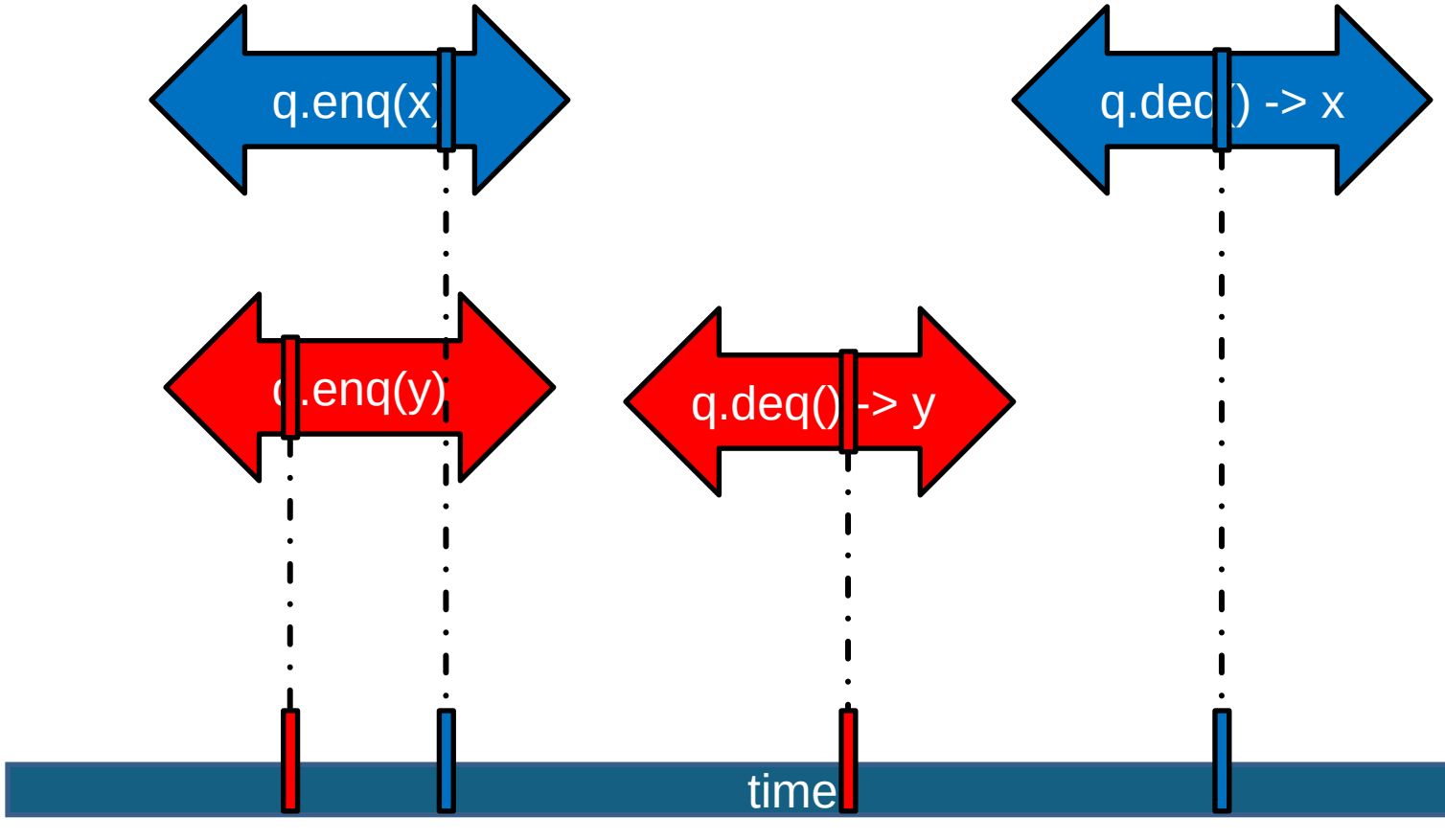
Is this linearizable?



# Example with FIFO Queue (1)

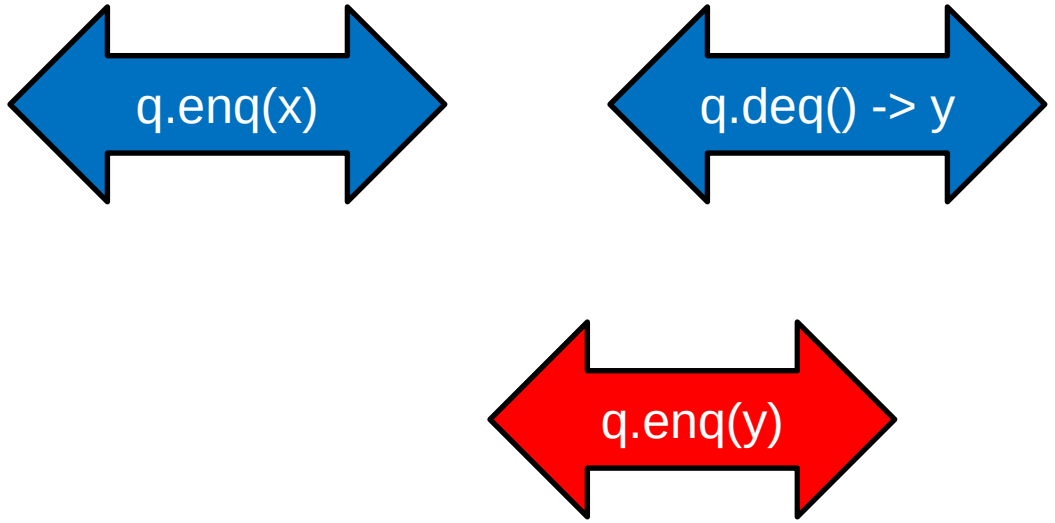
Is this linearizable?

**Yes!**



# Example (2)

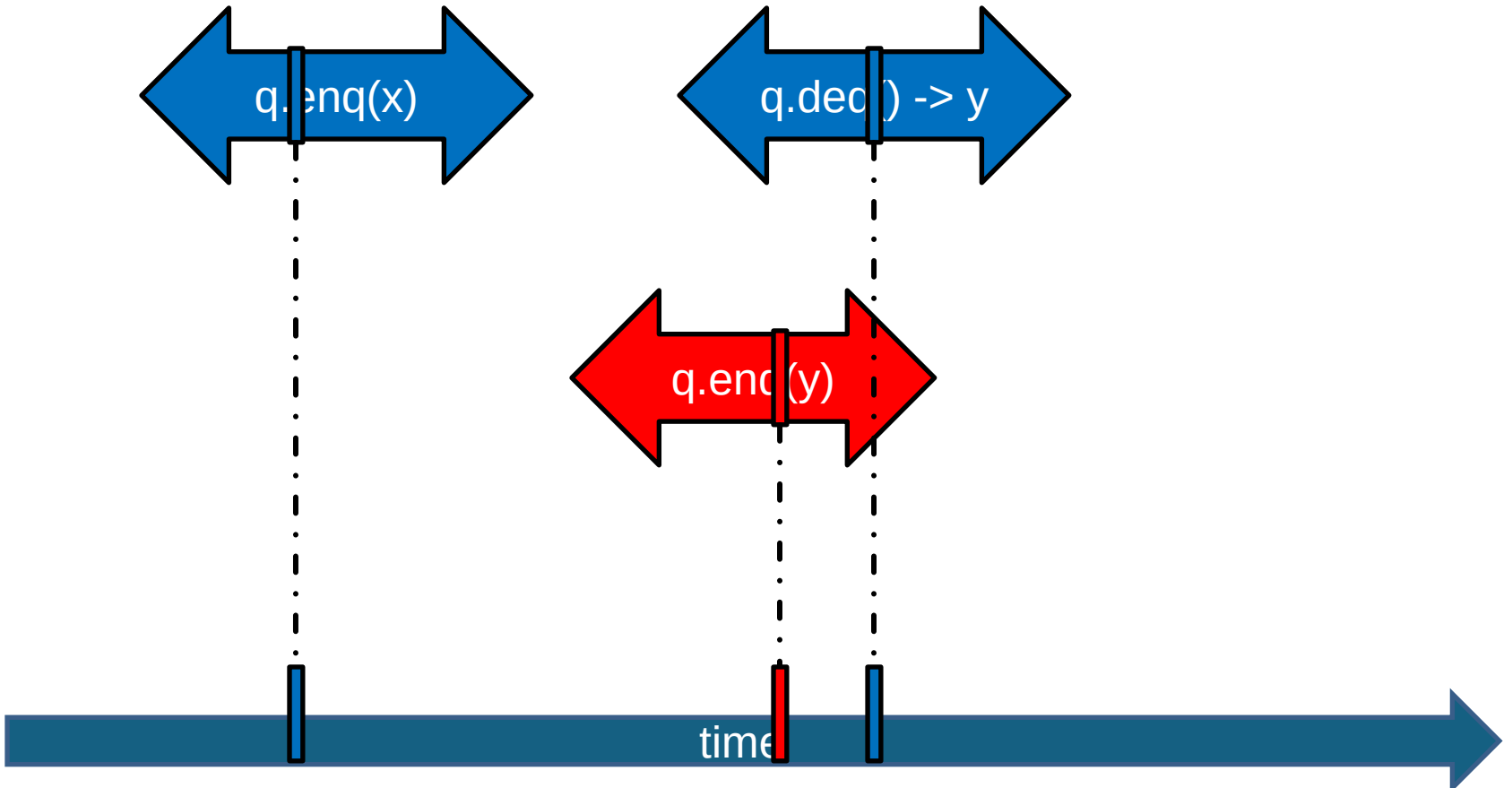
Is this linearizable?



# Example (2)

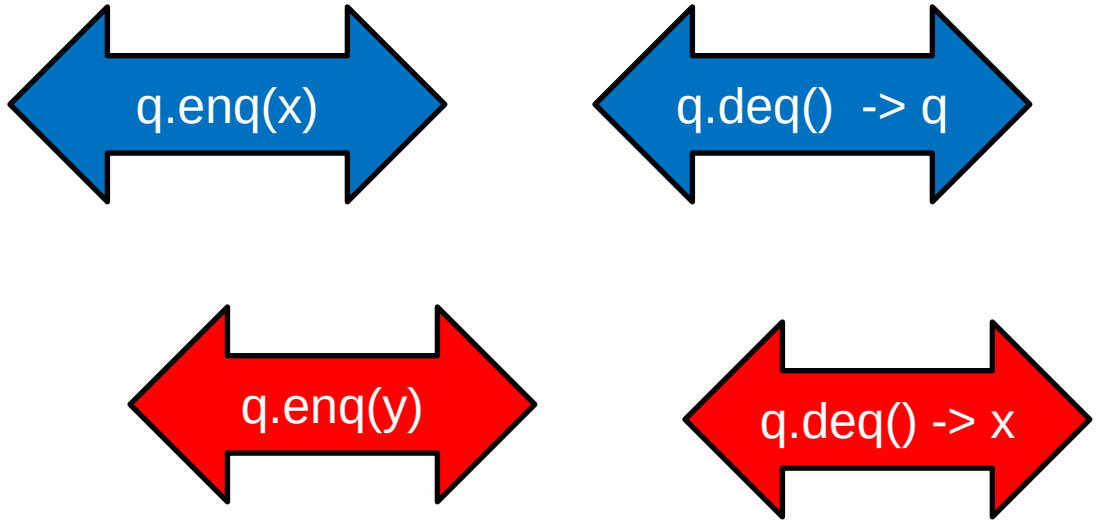
Is this linearizable?

**No!**



# Example (3)

Is this linearizable?



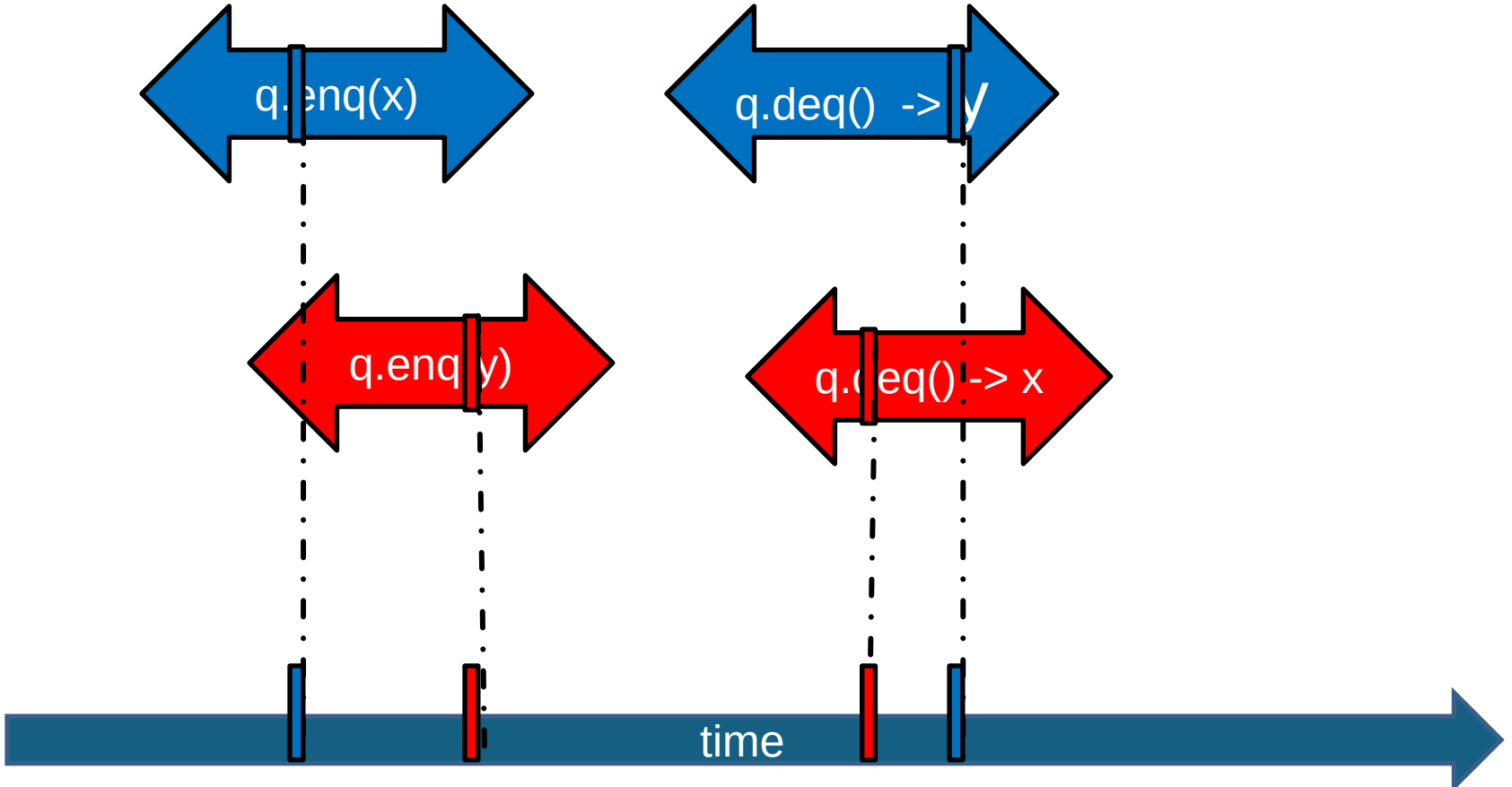
# Example (3)

Here we got multiple orders!

Essenti al

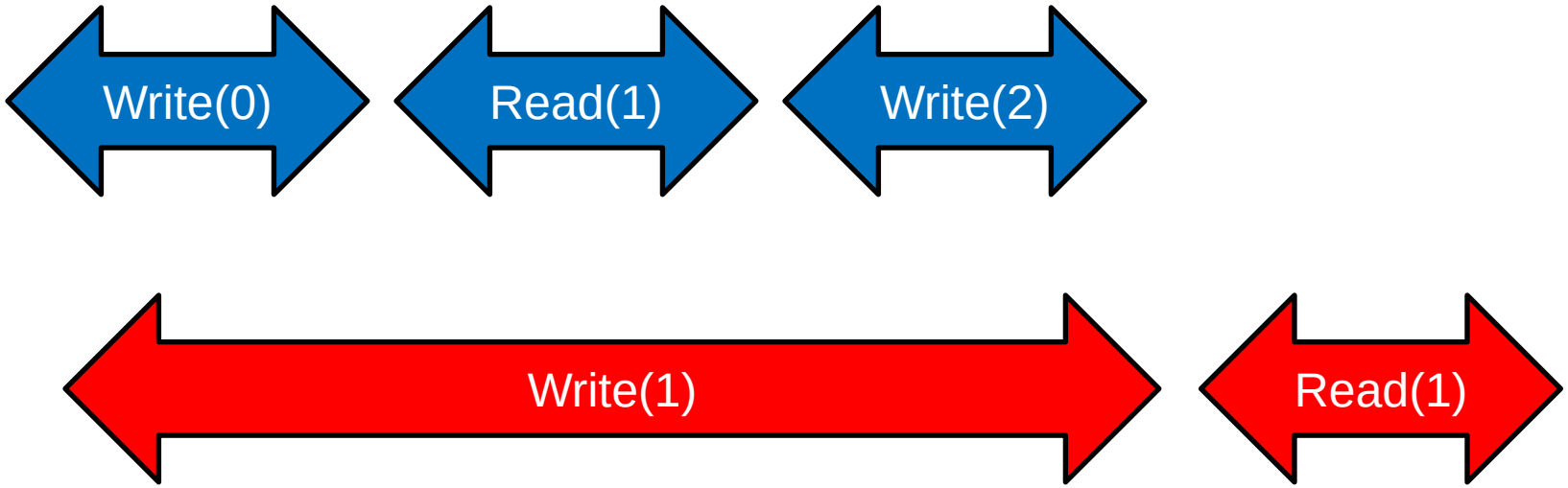
Is this linearizable?

**Yes!**



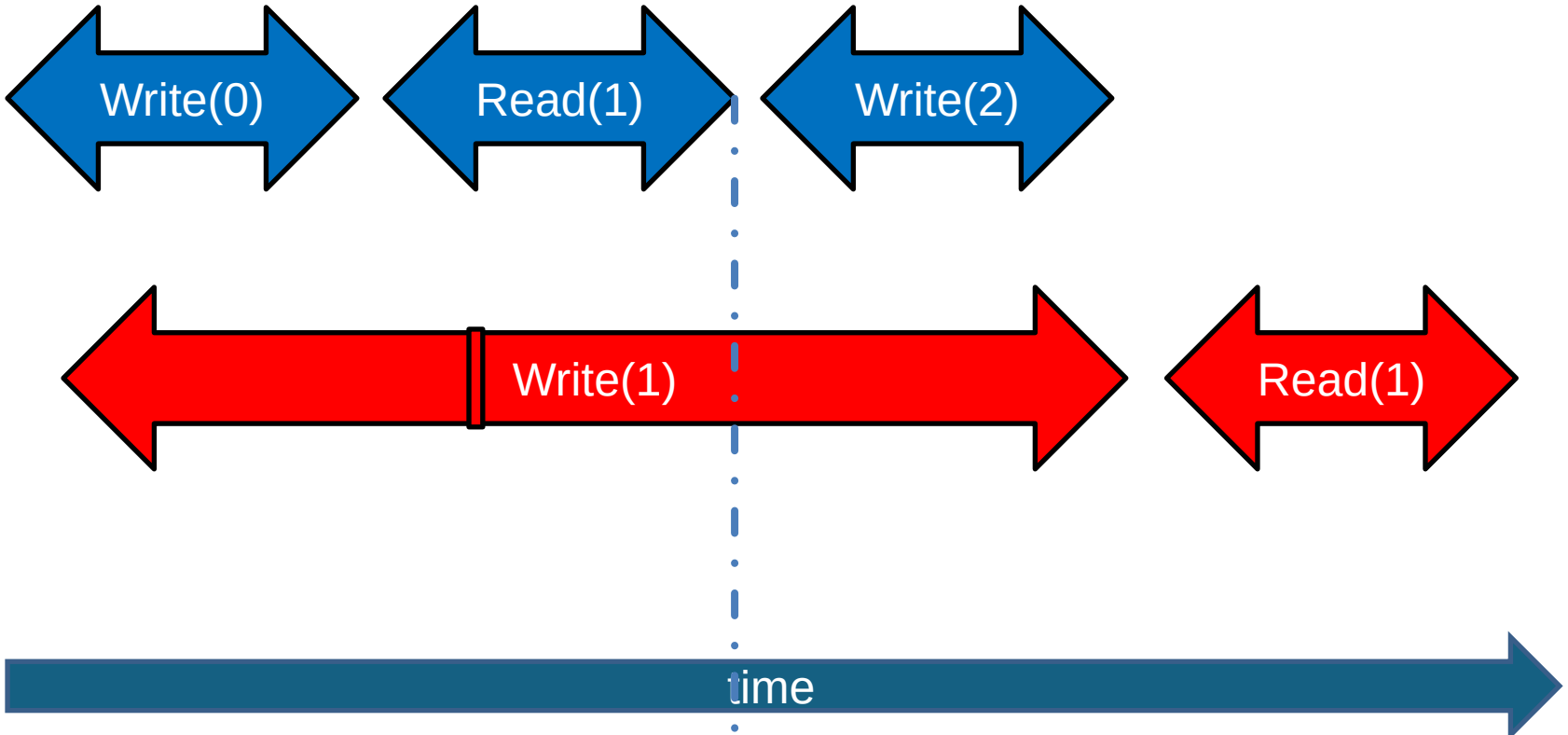
# Example (4)

Is this
linearizable?



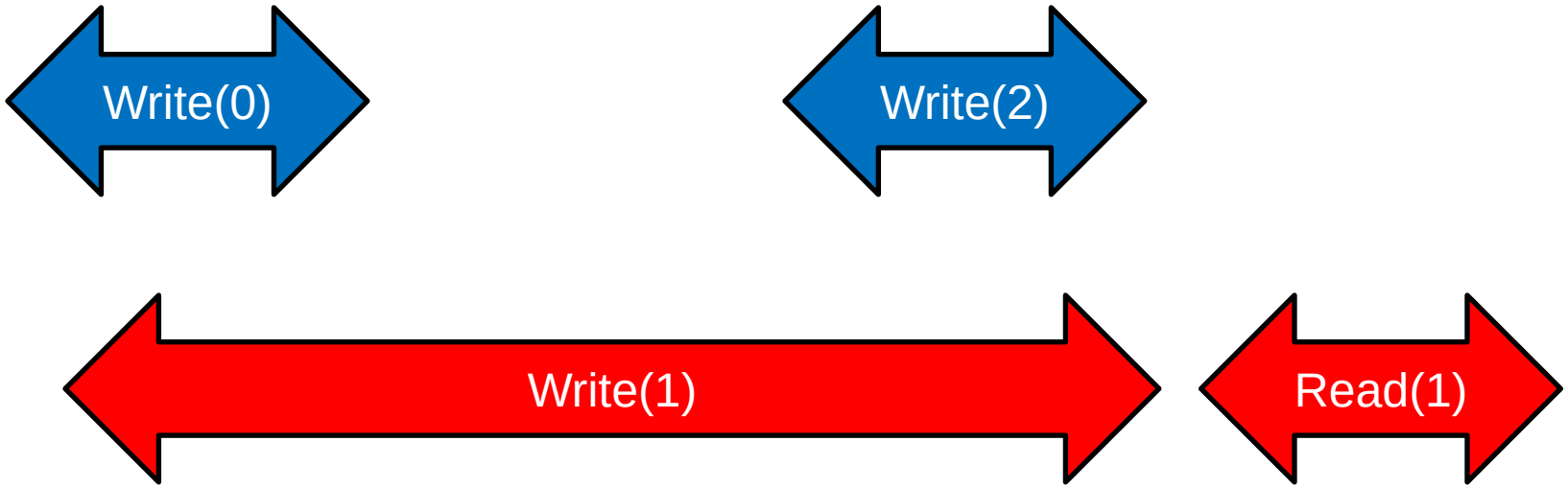
# Example (4)

Is this linearizable? **No!**



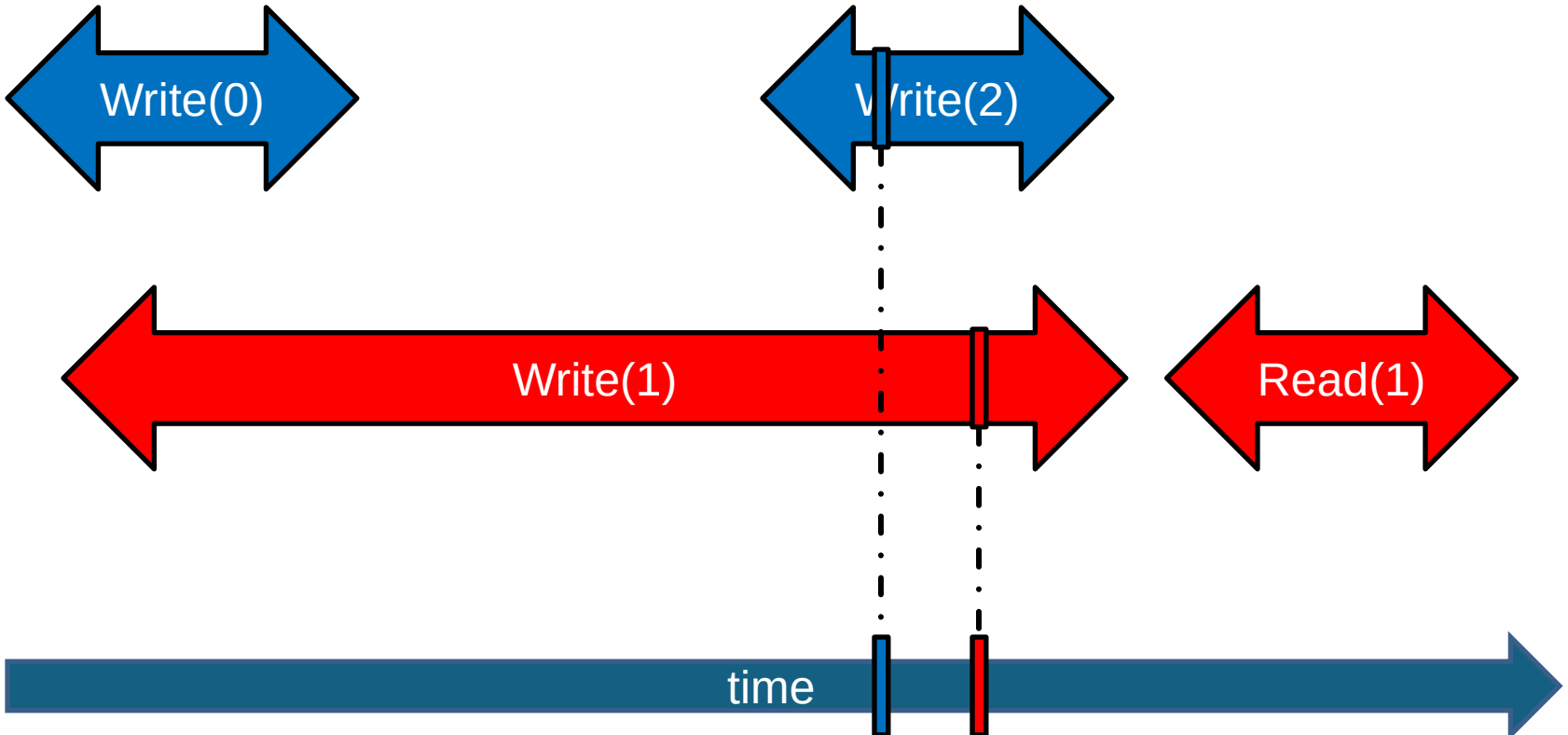
# Example (4.5)

Is this
linearizable?



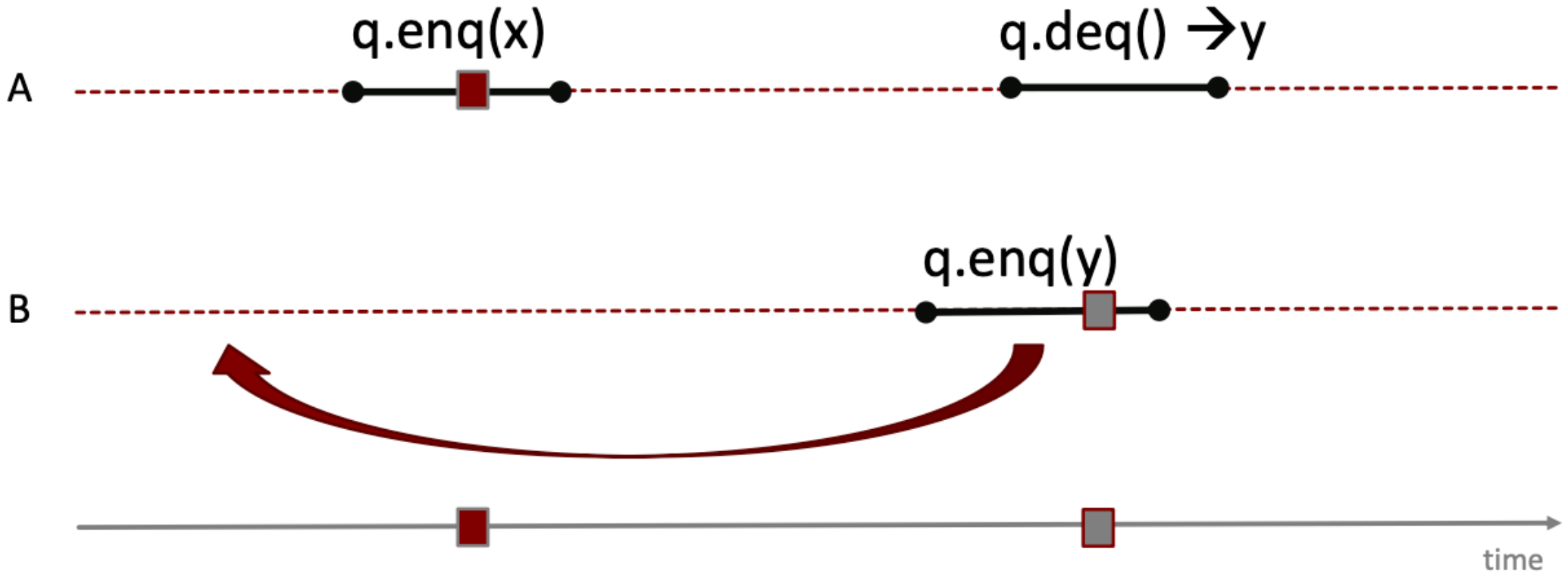
# Example (4.5)

Is this linearizable? **Yes!**



Essenti
al

# Difference between SC and Linearizability?




**In SC we can reorder events – as long as per-thread order is preserved!**

# Summary of definitions

- Histories A and B are called equivalent, if A and B's per-thread projections are identical
- A History H is called complete, if every invocation has a matching response (not necessarily immediately after the invocation).
- A History H is called well-formed, if its per-thread projections are all sequential. Histories that are not well formed usually do not make sense.
- A history H is called legal, if for every object x, the projections H|x all behave like the sequential specification of the object x.
- A History H is sequential, if there are no overlapping methods (when every invocation is immediately followed by the matching response)

# Summary of definitions

- A History is **SC (Sequentially Consistent),** if:

- Every Thread projection is a sequential history.

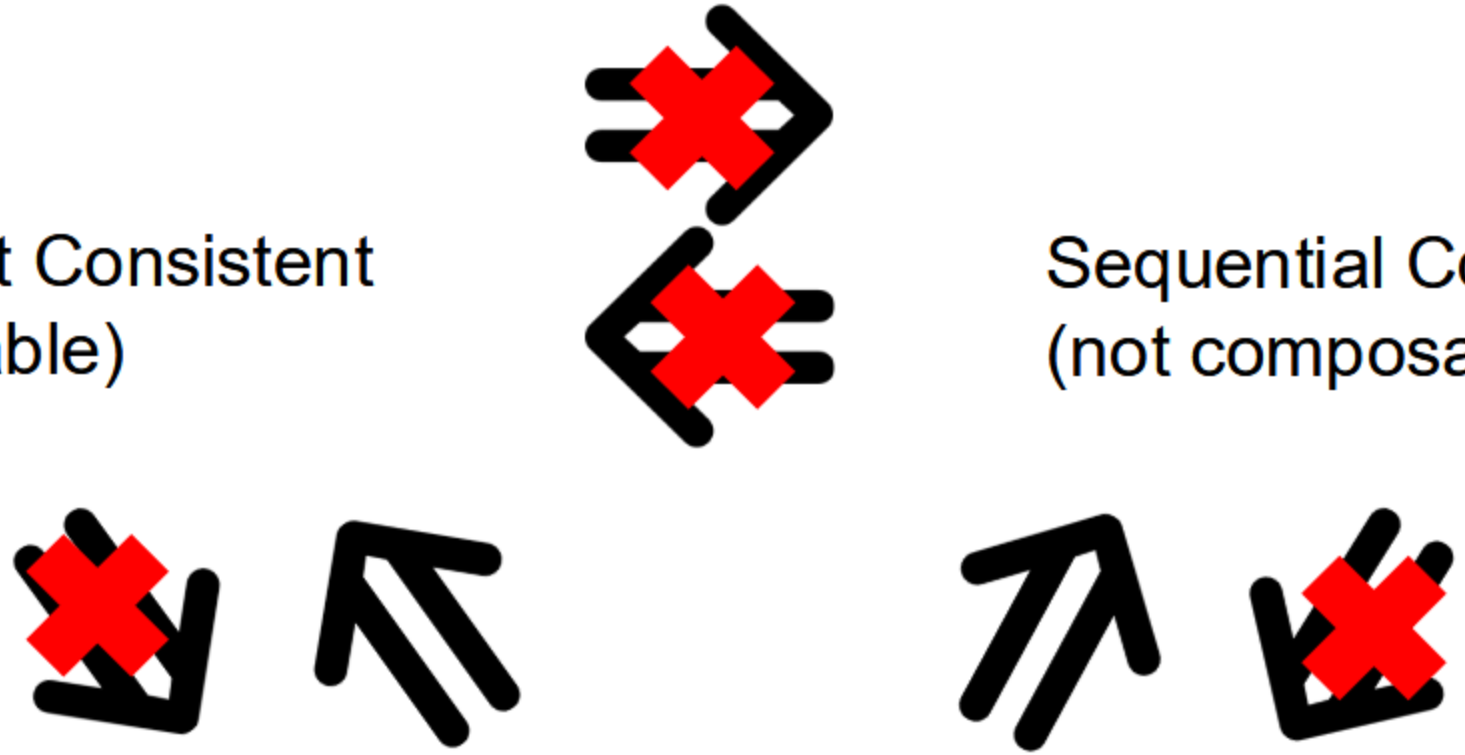
- Method calls appear to follow PO (Program Order), which allows for "reordering" of method-calls as long as they follow the ordering determined by the corresponding Thread-projection.

- For a program to be called SC (Sequentially Consistent), every possible execution history has to be SC.

# Summary of definitions

- A History H is **linearizable**, if there is an extension H' to H, which is equivalent (thread-projection-wise) to a legal sequential History S, where for all methods $m_x \rightarrow_h m_y \Longrightarrow m_x \rightarrow_s m_y$
- What linearizability means, is that the given parallel/concurrent execution is equal to some sequential history, where a preceding method call shows effect before the later one, but overlapping method calls can be "reordered" as wished
- The reordering means, that if $m_1$ and $m_2$ overlap (independent of which methods invocation or response was first/second) we can choose, if $m_1$ or $m_2$ shows its effect first
- Linearizability implies SC

Quiescent Consistent
(composable)



Sequential Consistent
(not composable)

Linearizable
(composable)

Thanks to @Erxuan Li, PProg25

# Linearization Point

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

*The linearization point is the point where the method takes effect, i.e., other threads see the change*

# Linearization Point

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();  <--
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

*The linearization point is the point where the method takes effect.*

# Linearization Point

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

*The linearization point is the point where the method takes effect, i.e. other threads see the change*

# Linearization Point

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

*The linearization point is the point where the method takes effect.*

# Linearization Point

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

*The linearization point is the point where the method takes effect.*

# Exam questions

# Exam question

(c) Markieren Sie alle wahren Aussagen fuer jede der folgenden Historien.

*Mark all true statements for each of the following histories.* (6)

```
1 A p.enq(3)
2 B p.enq(4)
3 B p:void
4 B p.deq()
5 A p:void
6 B p:3
```

◯ Wenn das Objekt p ein zu Beginn leerer Stack ist, ist die Historie linearisierbar.

*If the object p is an initially empty stack, the history is linearizable.*

# Exam question

(c) Markieren Sie alle wahren Aussagen fuer jede der folgenden Historien.

*Mark all true statements for each of the following histories.* (6)

```
1 A p.enq(3)
2 B p.enq(4)
3 B p:void
4 B p.deq()
5 A p:void
6 B p:3
```

◯ Wenn das Objekt p ein zu Beginn leerer Stack ist, ist die Historie linearisierbar.

*If the object p is an initially empty stack, the history is linearizable.*

- yes

# Exam question

(c) Markieren Sie alle wahren Aussagen fuer jede der folgenden Historien.

*Mark all true statements for each of the following histories.* (6)

```
1 A p.enq(3)
2 B p.enq(4)
3 B p:void
4 B p.deq()
5 A p:void
6 B p:3
```

◯ Wenn das Objekt p eine zu Beginn leere (FIFO) Queue ist, ist die Historie linearisierbar.

*If the object p is an initially empty (FIFO) queue, the history is linearizable.*

# Exam question

(c) Markieren Sie alle wahren Aussagen fuer jede der folgenden Historien.

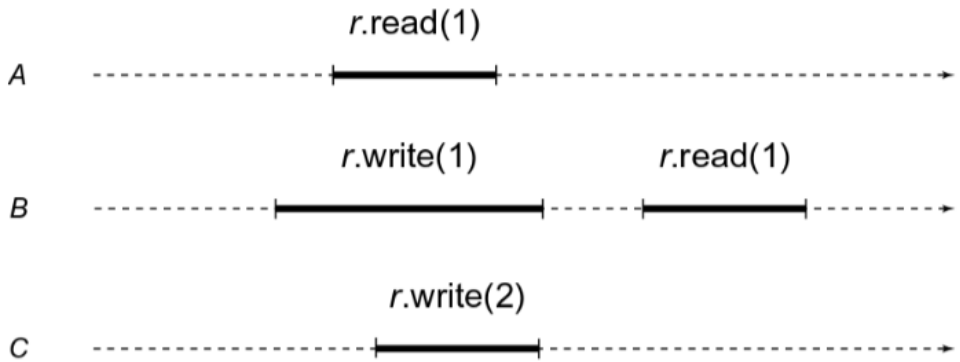
*Mark all true statements for each of the following histories.* (6)

```
1 A p.enq(3)
2 B p.enq(4)
3 B p:void
4 B p.deq()
5 A p:void
6 B p:3
```

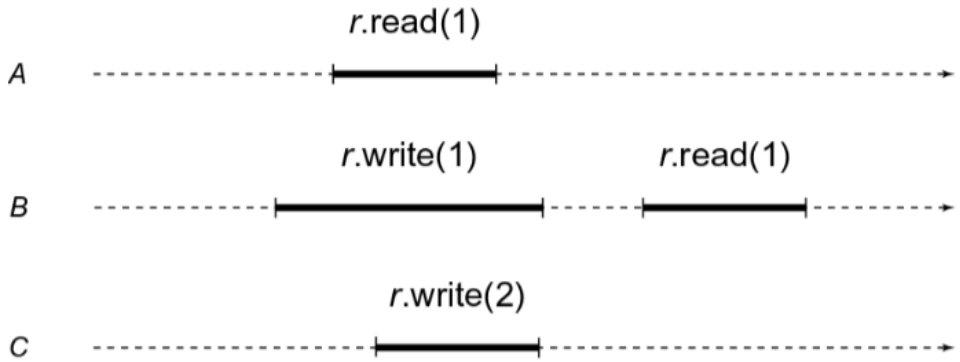
☐ Wenn das Objekt p eine zu Beginn leere (FIFO) Queue ist, ist die Historie linearisierbar.

*If the object p is an initially empty (FIFO) queue, the history is linearizable.*

- yes

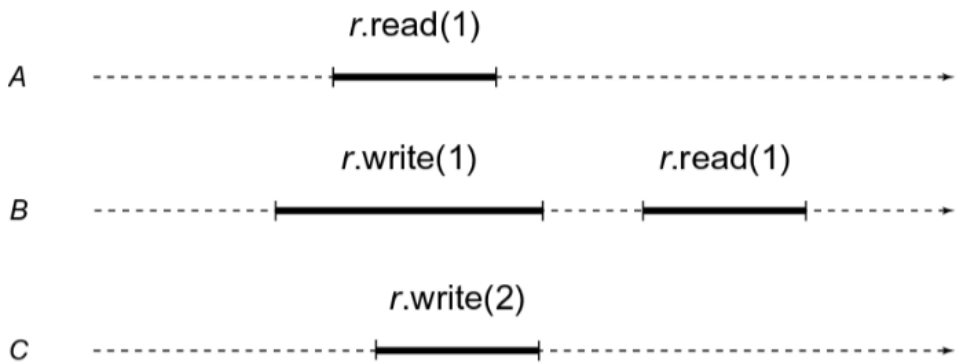



| | |
|---|---|
| ◯ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie linearisierbar. | *If the object r is a register initialized to zero, the history is linearizable.* |



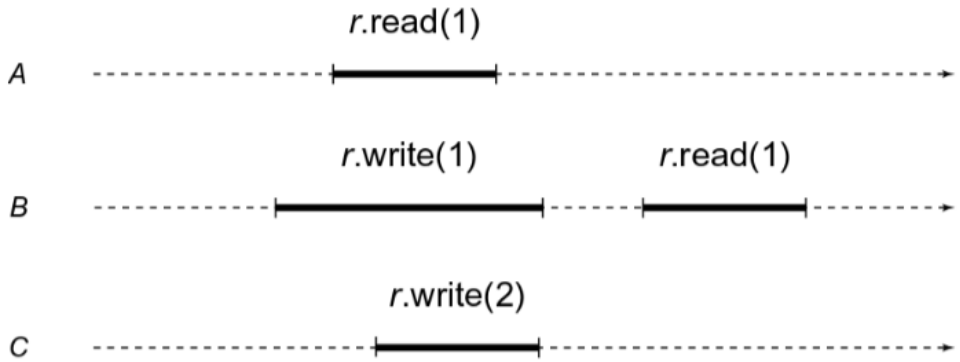

| | |
|---|---|
| ◯ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie linearisierbar. | *If the object r is a register initialized to zero, the history is linearizable.* |

- yes



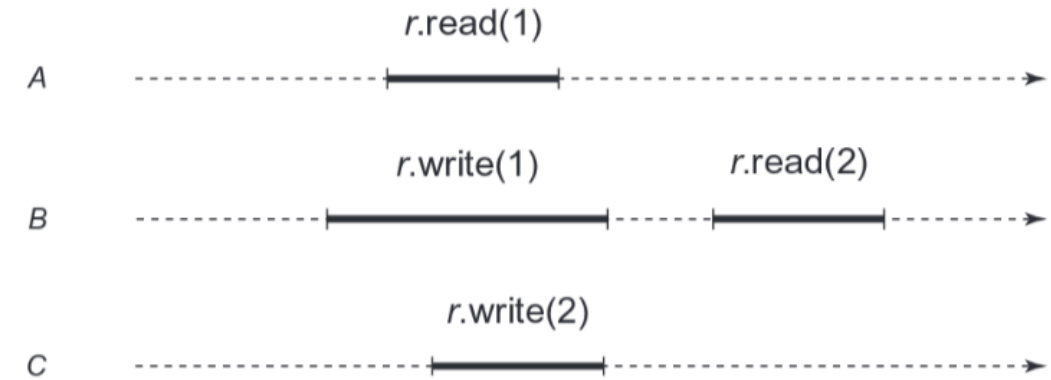

○ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie sequentiell konsistent.

*If the object r is a register initialized to zero, the history is sequentially consistent.*



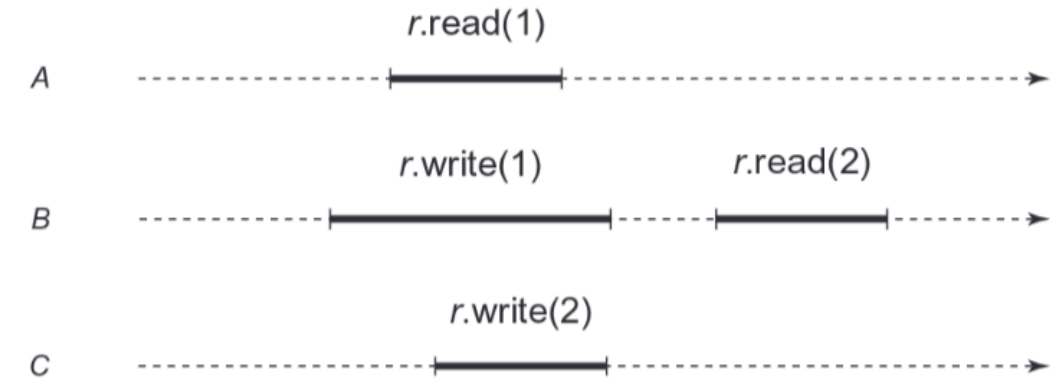

| | |
|---|---|
| ○ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie sequentiell konsistent. | *If the object r is a register initialized to zero, the history is sequentially consistent.* |

- Yes, linearizability implies SC




◯ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie linearisierbar.

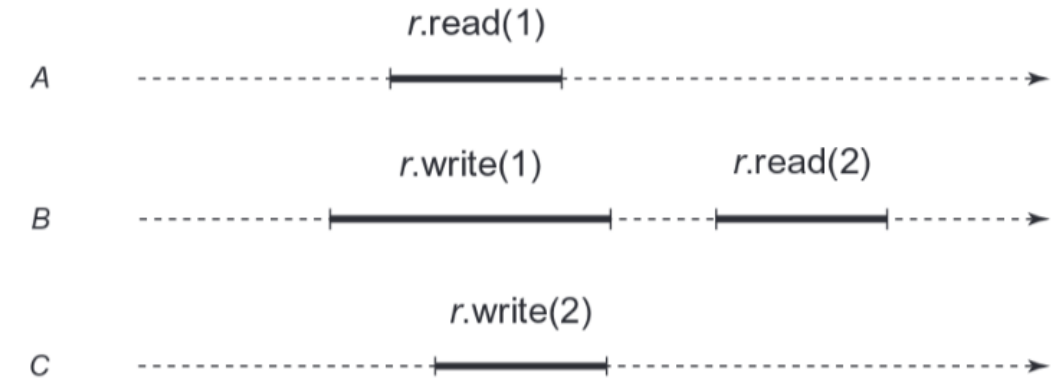
*If the object r is a register initialized to zero, the history is linearizable.*




○ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie linearisierbar.

*If the object r is a register initialized to zero, the history is linearizable.*

- Yes




| | |
|---|---|
| ◯ Wenn das Objekt r ein mit null initialisiertes Register ist, ist die Historie sequentiell konsistent. | *If the object r is a register initialized to zero, the history is sequentially consistent.* |

- Yes, Linearizability implies sequential consistency.

Shared stack object

Linearizable or not?

1.
```
A push (1) <-------------->
B push (0)          <--------------->
A pop  (1)                              <---------->
```

2.
```
A push (1) <----------------->
B push (0)          <------------->
B pop  (0)                             <------------->
A pop  (1)                        <------------->
```

3.
```
B push (0) <-------->
A pop  (0)                       <------>
B push (1)             <----->
B push (1)                        <------------->
A pop  (1)                                <-------->
```

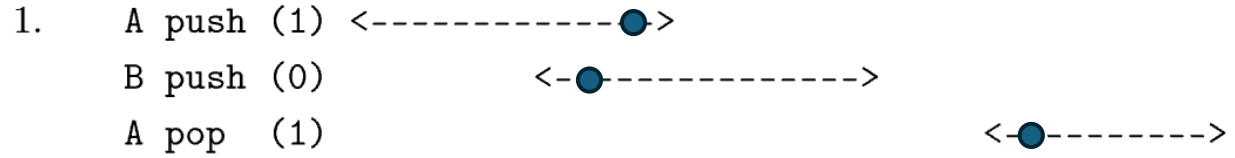
4.
```
A push (1) <------------>
B push (0)     <---->
B push (1)                 <----------->
A pop  (0)                        <------------>
```

5.
```
A push (0)        <----------------->
B push (1) <------------>
B pop  (0)                      <--------->
A pop  (1)                                <---------->
```

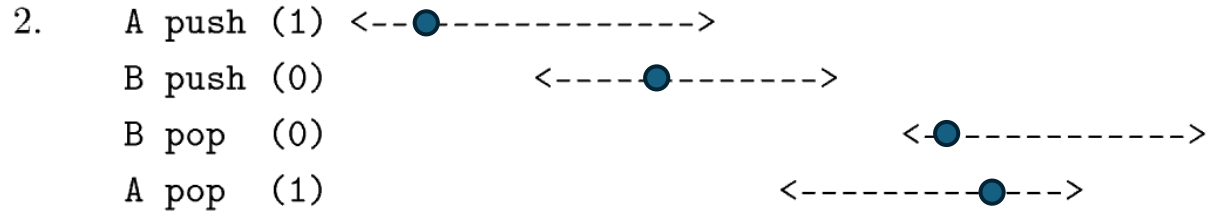
Shared stack object

Linearizable or not?

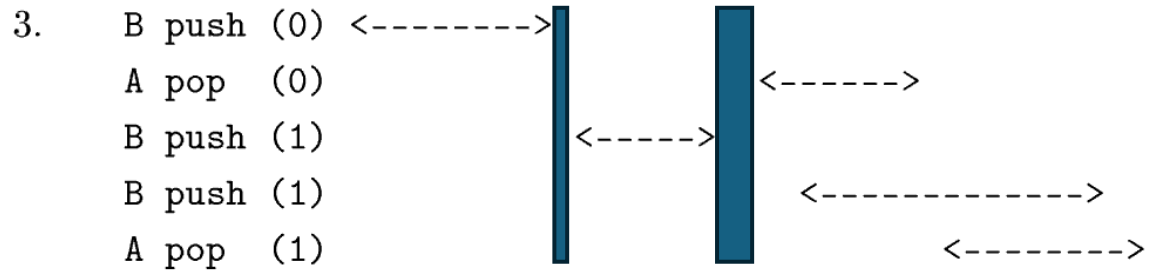
1. yes



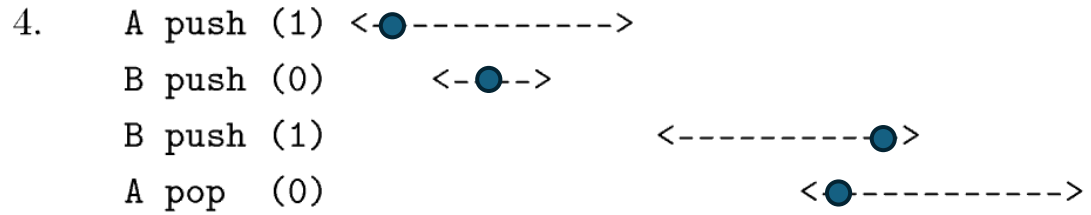

2. yes



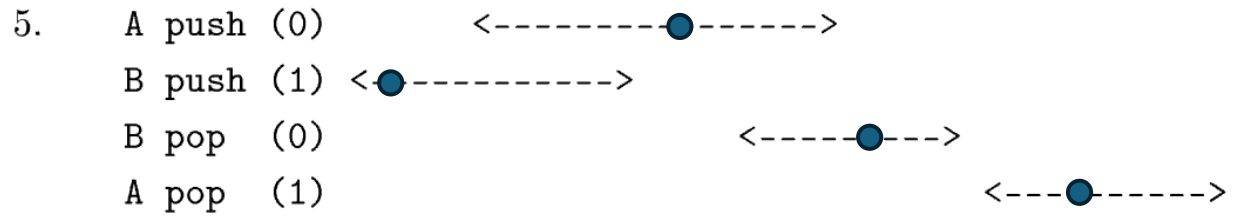

3. no




4. yes




5. yes



```
H= A s.push(x)
   B s.pop()
   B s:x
   A s:void
   A s.push(y)
   B s.push(y)
   A s:void
   A s.pop
   A s:y
   B s:void
```

```
G= A s.push(x)
   A s:void
   A s.push(y)
   A s:void
   B s.pop()
   B s:x
   A s.pop
   A s:y
   B s.push(y)
   B s:void
```

*Are these histories equivalent?*

*[ ] Yes [ ] No*

```
H= A s.push(x)
   B s.pop()
   B s:x
   A s:void
   A s.push(y)
   B s.push(y)
   A s:void
   A s.pop
   A s:y
   B s:void
```

```
G= A s.push(x)
   A s:void
   A s.push(y)
   A s:void
   B s.pop()
   B s:x
   A s.pop
   A s:y
   B s.push(y)
   B s:void
```

*Are these histories equivalent?*

*[✖] Yes [ ] No*

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

```
H= A s.push(x)
   B s.pop()
   B s:x
   A s:void
   A s.push(y)
   B s.push(y)
   A s:void
   A s.pop
   A s:y
   B s:void
```

```
G= A s.push(x)
   A s:void
   A s.push(y)
   A s:void
   B s.pop()
   B s:x
   A s.pop
   A s:y
   B s.push(y)
   B s:void
```

*Are they sequential?*

*H: [ ] Yes [ ] No*

*G: [ ] Yes [ ] No*

```
H= A s.push(x)
   B s.pop()
   B s:x
   A s:void
   A s.push(y)
   B s.push(y)
   A s:void
   A s.pop
   A s:y
   B s:void
```

```
G= A s.push(x)
   A s:void
   A s.push(y)
   A s:void
   B s.pop()
   B s:x
   A s.pop
   A s:y
   B s.push(y)
   B s:void
```

*Are they sequential?*

*H: [ ] Yes [X] No*

*G: [X] Yes [ ] No*

A history is sequential iff:

1. The first event is an invocation
2. Each invocation is immediately followed by a response

Kreuzen sie alle korrekten Aussagen an.

*Mark all correct statements.*

- ☐ Es existieren sequentiell konsistente Historien die nicht linearisierbar sind.
  *There exist sequentially consistent histories which are not linearizable.*
- ☐ Alle linearisierbaren Historien sind sequentiell konsistent.
  *All linearizable histories are sequentially consistent.*
- ☐ Unter der Annahme, das $H$ nur die Objekte $x$ und $y$ enthält, gilt: Wenn $H|x$ and $H|y$ linearisierbar sind, dann ist $H$ linearisierbar.
  *If $H|x$ and $H|y$ are linearizable, $H$ is linearizable (assuming $x$ and $y$ are the only objects present in $H$)*
- ☐ Unter der Annahme, das $H$ nur die Objekte $x$ und $y$ enthält, gilt: Wenn $H|x$ and $H|y$ sequentiell konsistent sind, dann ist $H$ sequentiell konsistemt.
  *If $H|x$ and $H|y$ are sequentially consistent, $H$ is sequentially consistent (assuming $x$ and $y$ are the only objects present in $H$)*

Kreuzen sie alle korrekten Aussagen an.

*Mark all correct statements.*

- [x] Es existieren sequentiell konsistente Historien die nicht linearisierbar sind.

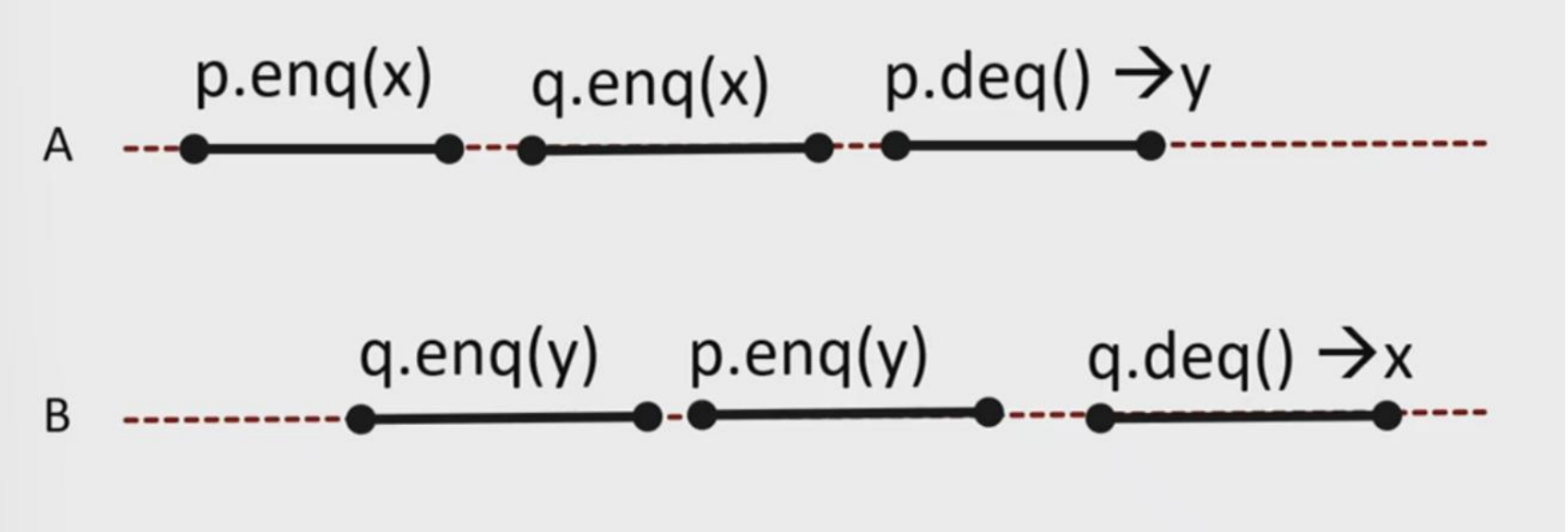
  *There exist sequentially consistent histories which are not linearizable.*
- [x] Alle linearisierbaren Historien sind sequentiell konsistent.

  *All linearizable histories are sequentially consistent.*
- [x] Unter der Annahme, das $H$ nur die Objekte $x$ und $y$ enthält, gilt: Wenn $H|x$ and $H|y$ linearisierbar sind, dann ist $H$ linearisierbar.

  *If $H|x$ and $H|y$ are linearizable, $H$ is linearizable (assuming $x$ and $y$ are the only objects present in $H$)*
- [ ] Unter der Annahme, das $H$ nur die Objekte $x$ und $y$ enthält, gilt: Wenn $H|x$ and $H|y$ sequentiell konsistent sind, dann ist $H$ sequentiell konsistemt.

  *If $H|x$ and $H|y$ are sequentially consistent, $H$ is sequentially consistent (assuming $x$ and $y$ are the only objects present in $H$)*

# Counterexample last statement

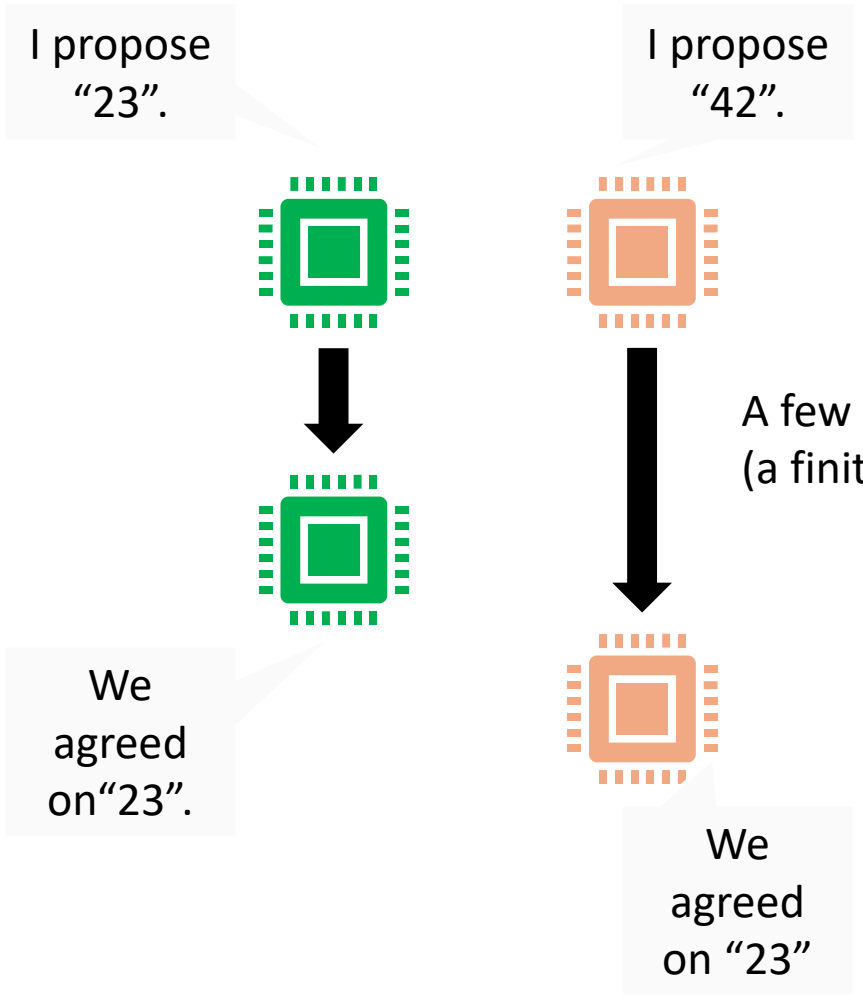



H|p is sequentially consistent
H|q is sequentially consistent
H is not sequentially consistent

In general: sequential consistency is not composable

# Consensus

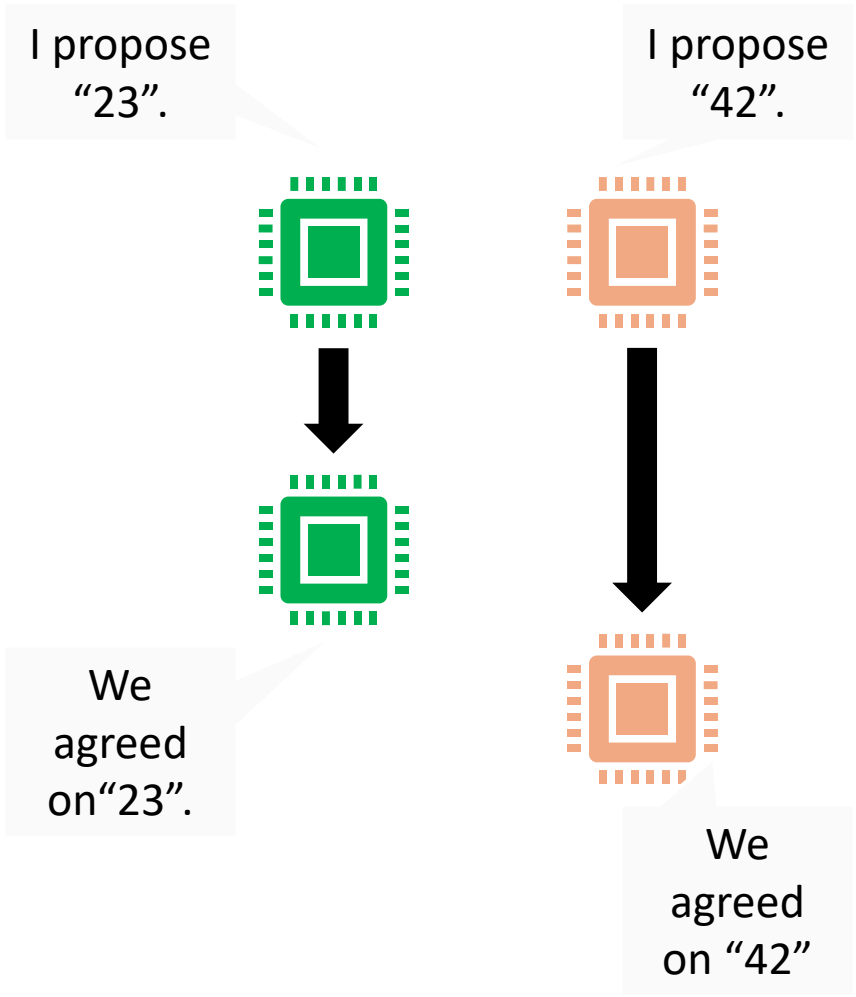
# Recap: Consensus Protocols






Which other scenarios are allowed?

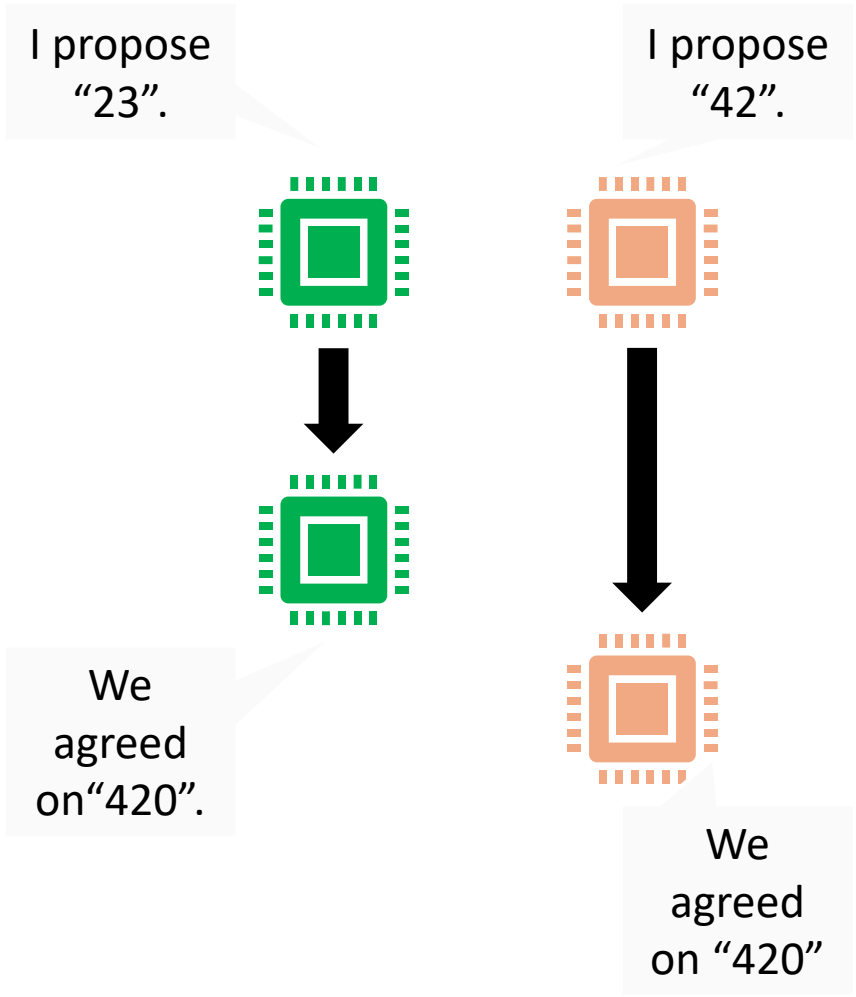
# Consistent Result




This is illegal!

Consensus result needs to be consistent: the same on all threads.

# Valid Result



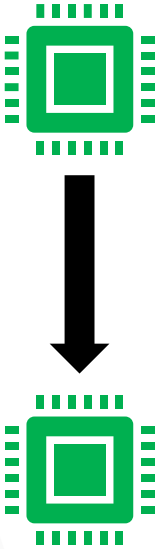

This is illegal!

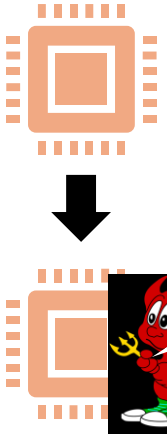
Consensus result needs to be valid: proposed by some thread.

# Wait-Free

I propose “23”.

I propose “42”.





I will not schedule you now!

I cannot finish because I am waiting for the other thread.

This is illegal!

Consensus needs to be wait-free: All threads finish after a finite number of steps, independent of other threads.

# Consistent, Valid, Wait-free

- You need to know these 3 properties

# Simplification: Binary Consensus

- **Instead of proposing an integer, every thread now proposes either 0 or 1**
- **Equivalent to “normal” consensus for two threads**
  - How can we proof this?

```
binary_decide(bit b) {
  return int_decide(b)
}
```

We can implement binary consensus using normal consensus.

```
int_decide(int d) {
  prop[id] = d; //prop is shared
  other = (id + 1)%2;
  int win = bin_decide(id);
  return prop[win];
}
```

We can implement binary consensus using normal consensus (id in {0,1} and unique).

# Consenus Number

- The consensus number of C is the largest n for which C solves n-thread consensus
- Atomic Registers have consensus number 1. CAS has consensus number ∞. Can be shown by construction

CAS consensus

```
class CASConsensus {
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST); // supports CAS
    private AtomicIntegerArray proposed; // suffices to be atomic register

    ... // constructor (allocate array proposed etc.)

    public Object decide(Object value) {
        int i = ThreadID.get();
        proposed.set(i, value);
        if (r.compareAndSet(FIRST, i)) // I won
            return proposed.get(i);    // = value
        else
            return proposed.get(r.get());
    }
}
```

# Implementing two thread consensus with TAS

- **Assume you have a machine with atomic registers and an atomic test-and-set operation with the following semantics (X is initialized to 1):**

```
int TAS() {
    res = X;
    if (res == 1) {
        X = 0;
    }
    return res;
}
```

- **Implement a two-process consensus protocol using TAS() and atomic registers.**

# Implementing two thread consensus - Solution

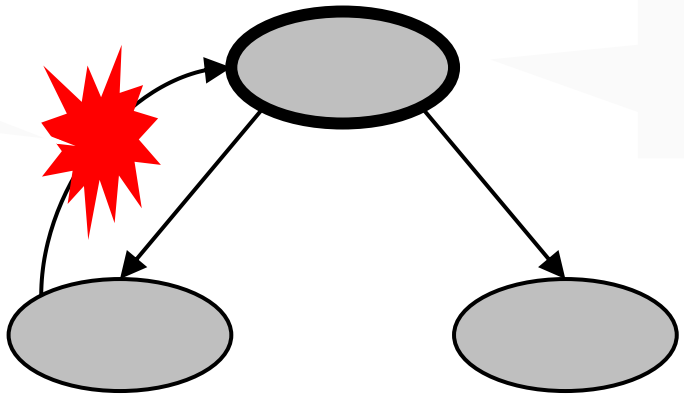
- **Code for both threads**

```
read own_value;
read other_value;
if (TAS() == 0) {
   return own_value;
} else
   return other_value;
```

# State Diagrams of Two-thread Consensus Protocols

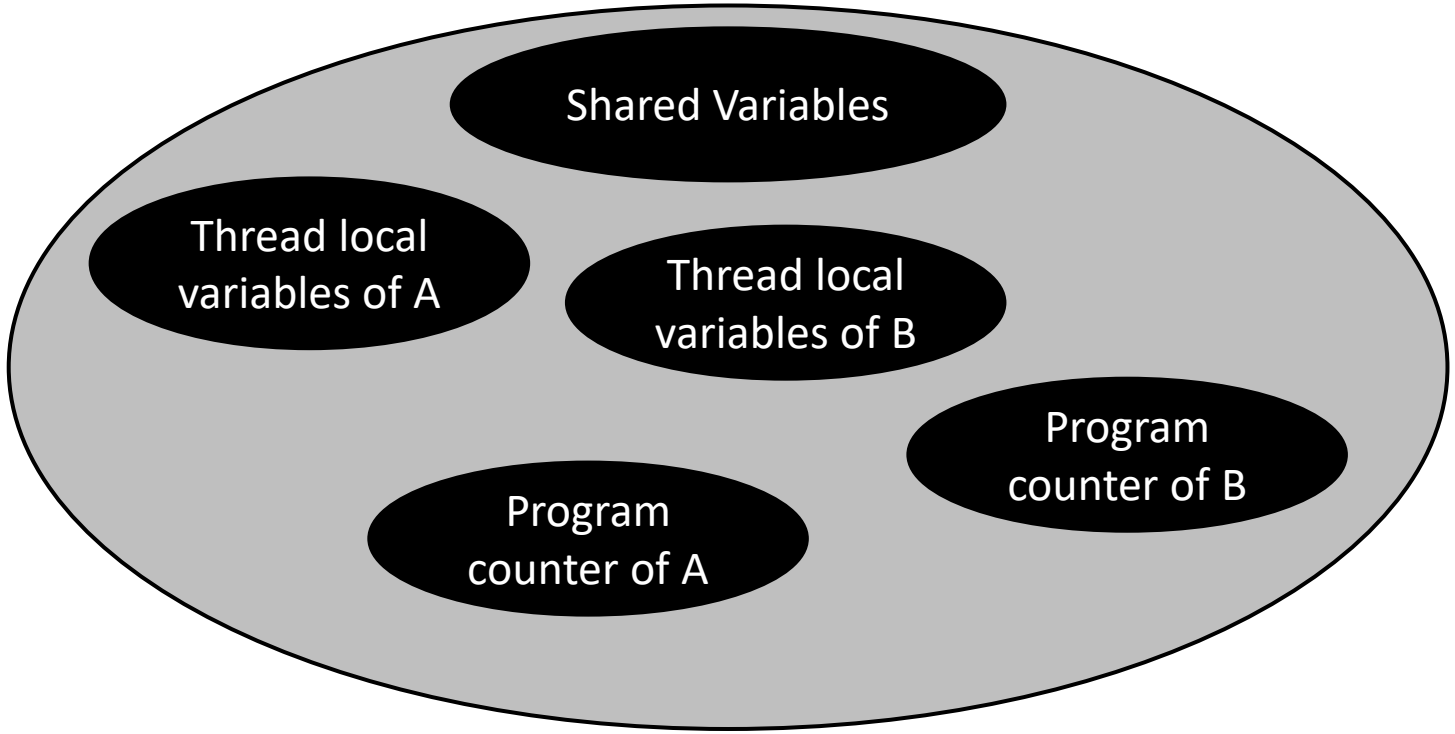
Cycles among states cannot exist in a wait-free algorithm: The state “looks” the same each time we visit, so we are trapped forever in the loop and not wait-free.

Each state has at most two **successors**: Either A or B execute an instruction.

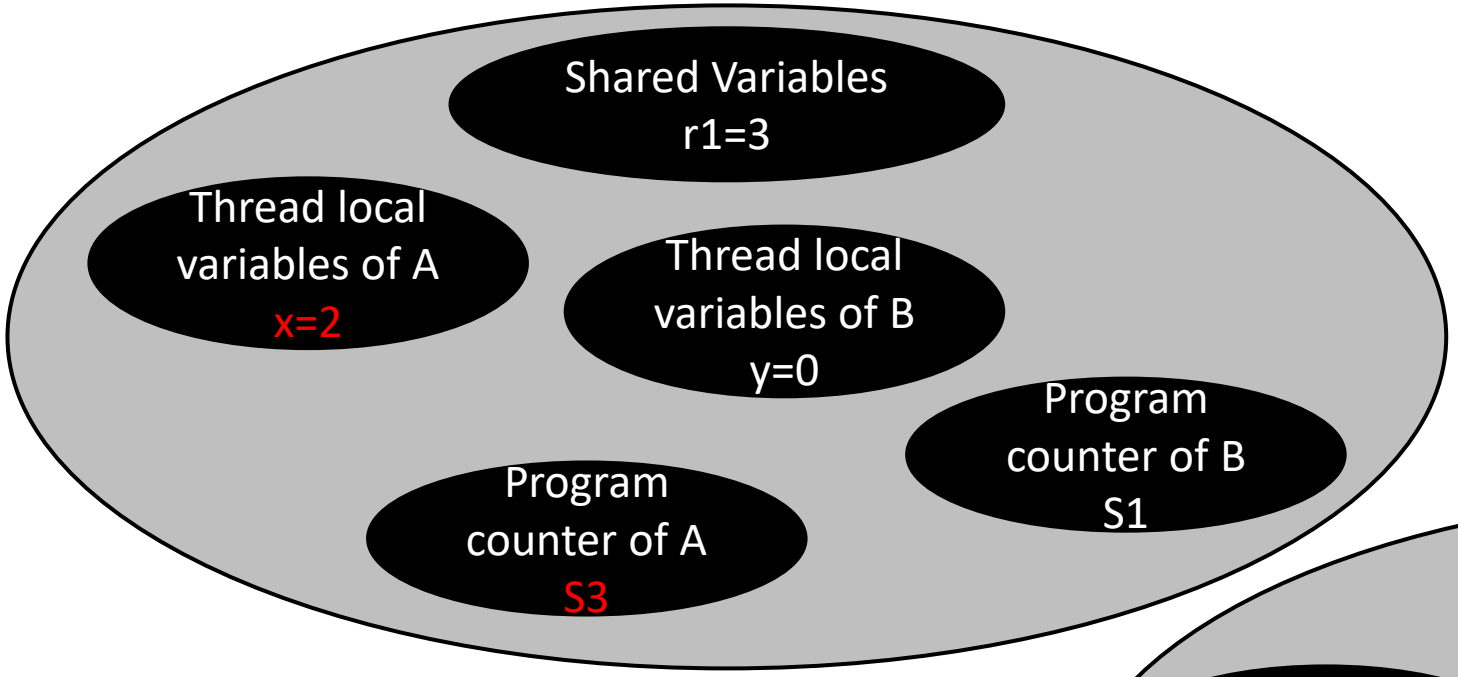


Start state, both threads (A and B) have not yet executed the first instruction of the consensus protocol.

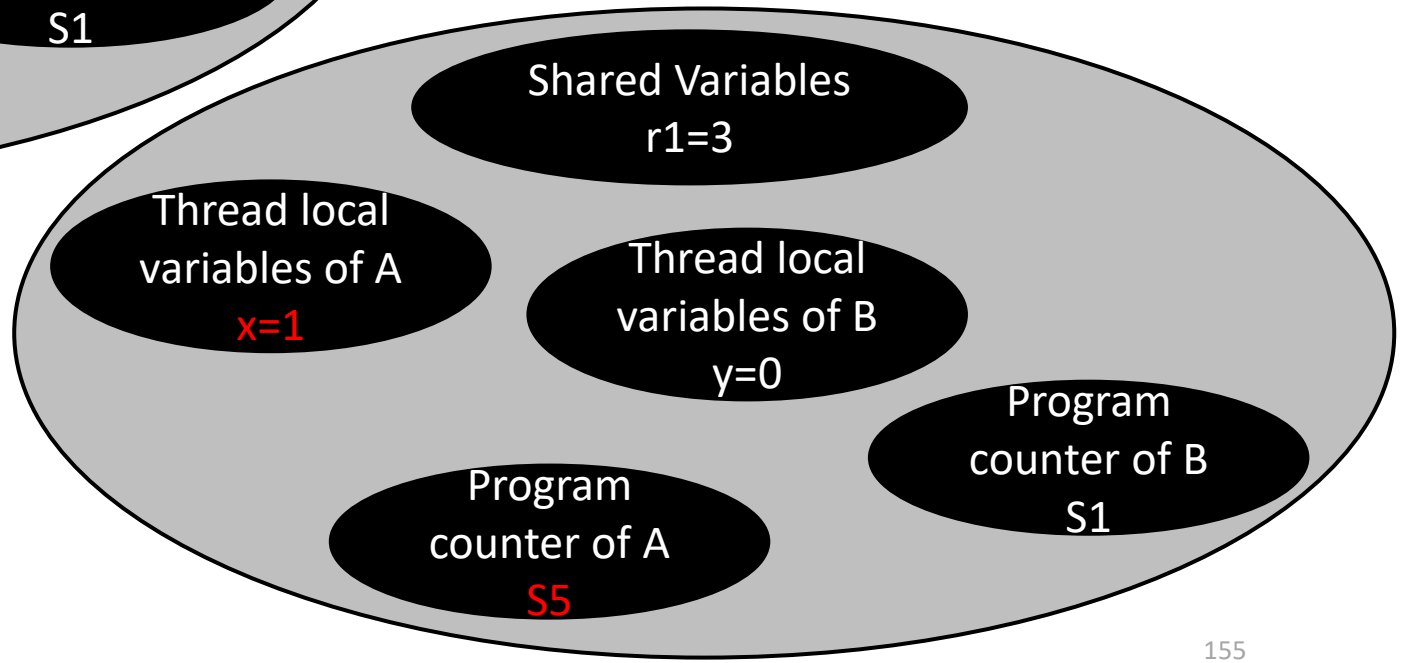
# Anatomy of a State (in two-thread consensus)



# Anatomy of a State



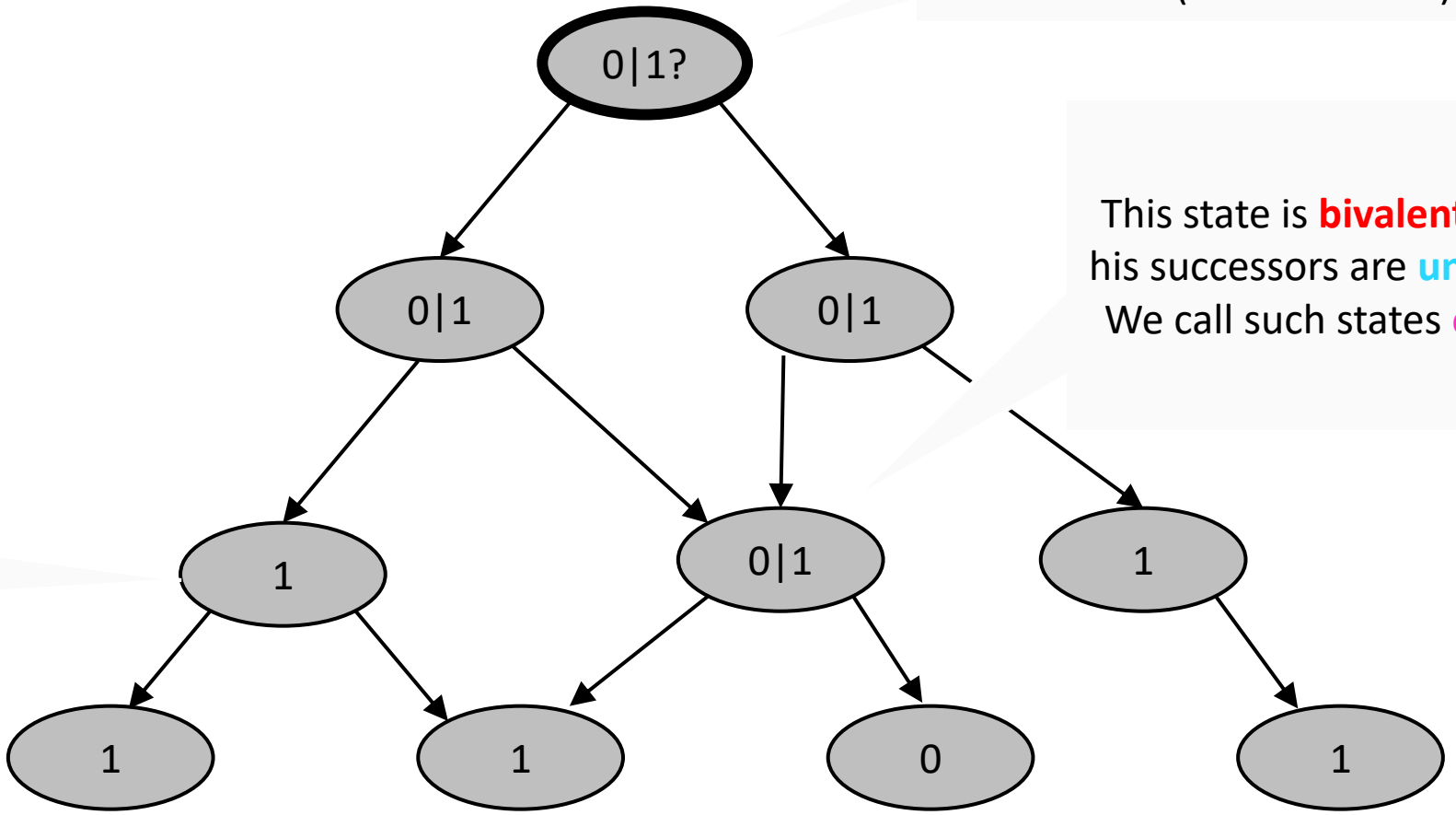

The states are different, since A has different local variables and program counter values.




**Yet from B's perspective they look the same! (Until A writes x into a shared variable!)**

# Critical States

There is always at least one **bivalent** state (the start state).




This state is **bivalent** but all his successors are **univalent**. We call such states **critical**.

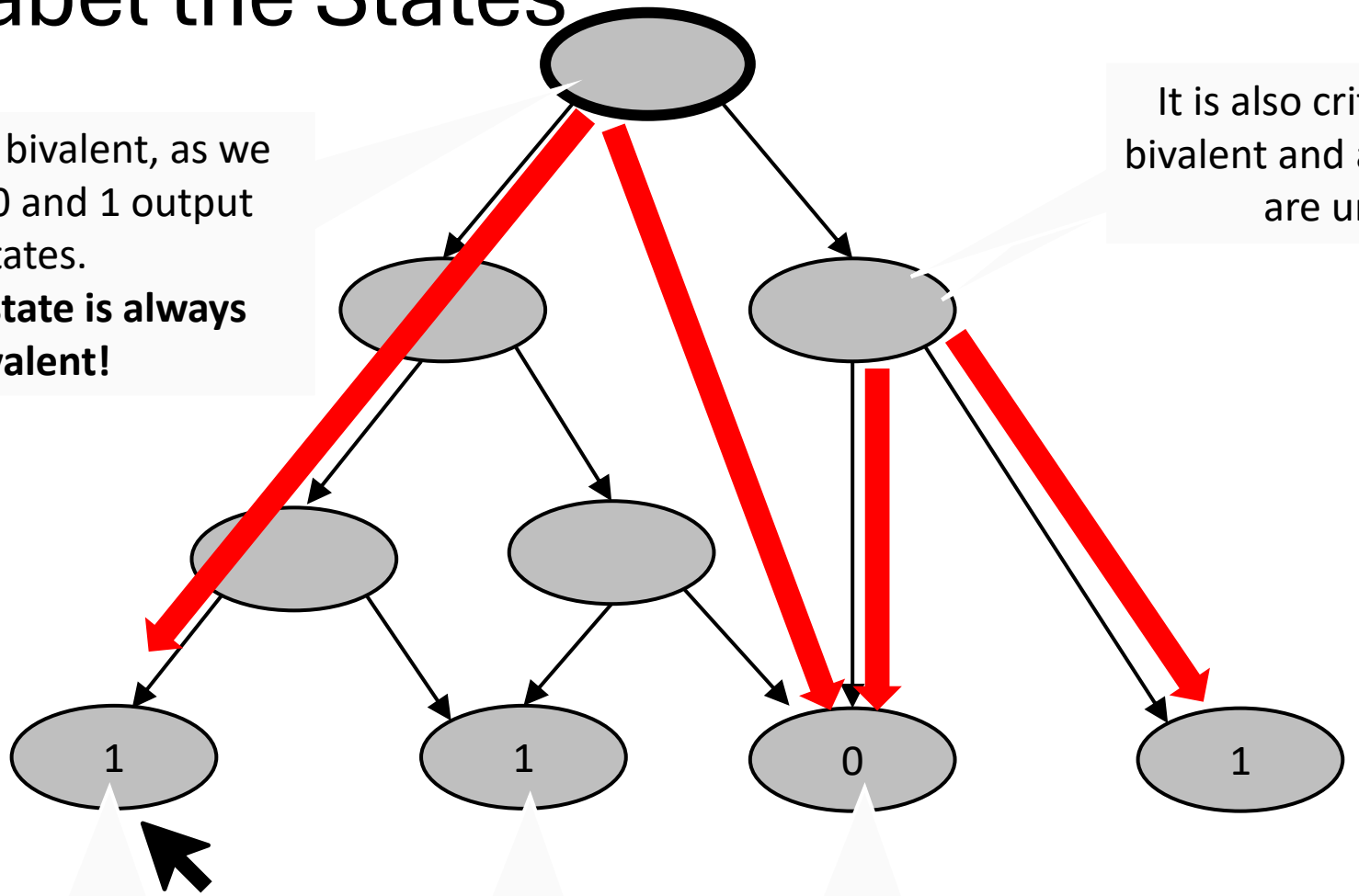
Frm this state we only reach states with output 1, so it is also **univalent.**

Output states are always **univalent.**

# Quiz: Label the States

This state is bivalent, as we can reach 0 and 1 output states.
**The start state is always bivalent!**

It is also critical, since it is bivalent and all its successors are univalent.




Output states are al
**univalent.**

Output states a
**univaler**

Output states are always
**univalent.**

# Critical State Existence Proof

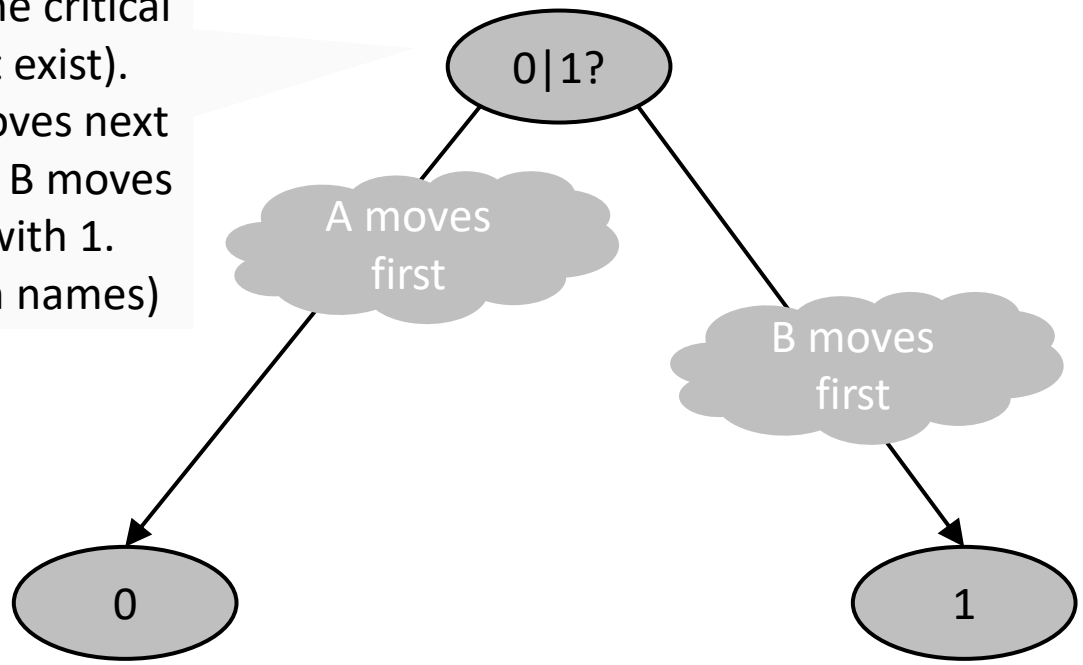
**Lemma: Every consensus protocol has a critical state.**

**Proof: From (bivalent) start state, let the treads only move to other bivalent states.**

- **If it runs forever the protocol is not wait free.**
- **If it reaches a position where no moves are possible this state is critical.**

# Impossibility Proof Setup – Critical State

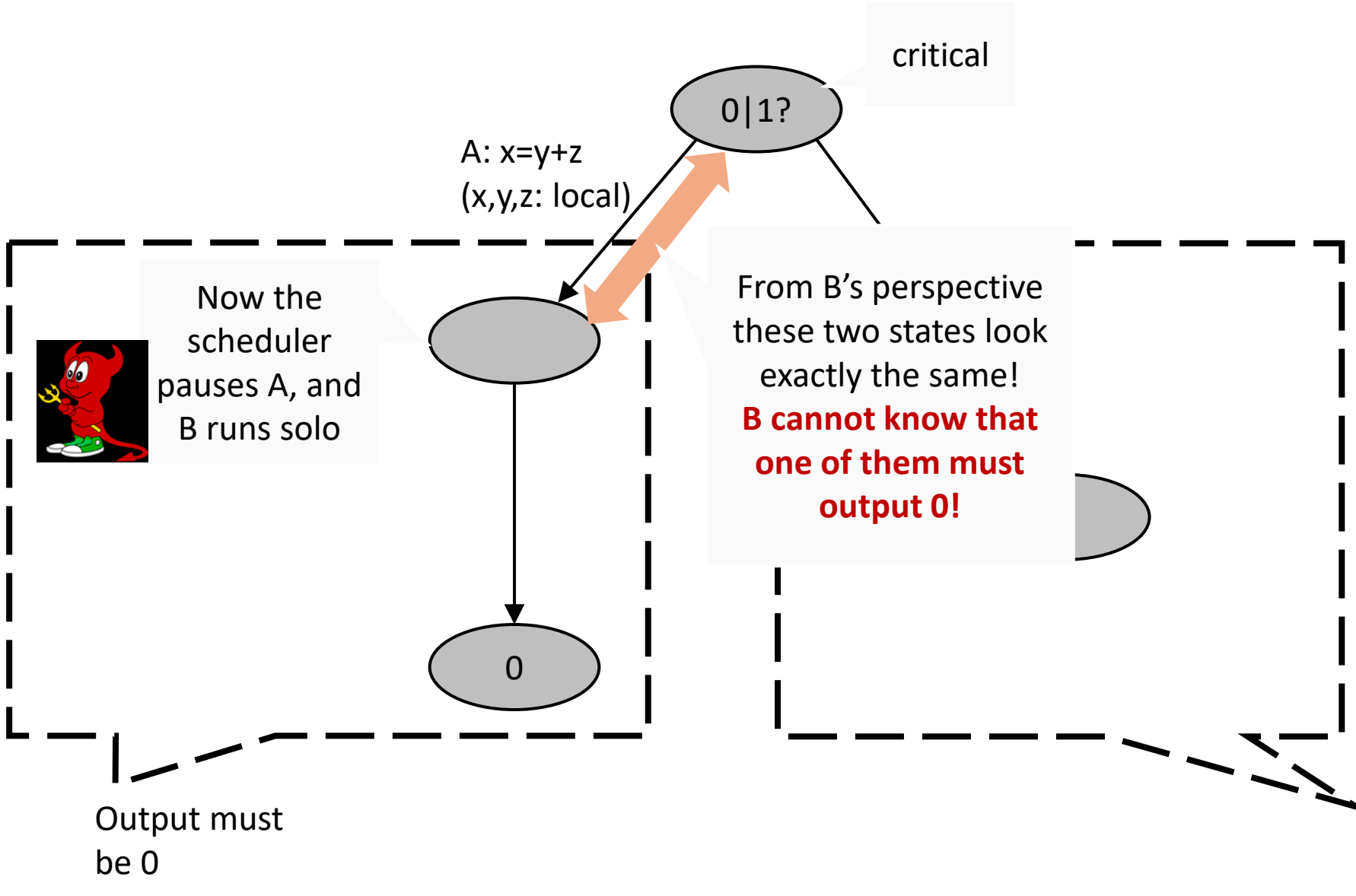
Assume we are in the critical state (which must exist). Assume that if A moves next we end up with 0, if B moves next we end up with 1. (w.l.o.g., can switch names)




So what actions can a thread perform in his “move”?

Either read or write a shared register! – Let’s see why.

# Impossibility Proof Setup – Possible actions of a thread




So what actions can a thread perform in his "move"?

What happens if A just reads from and writes to local vars?

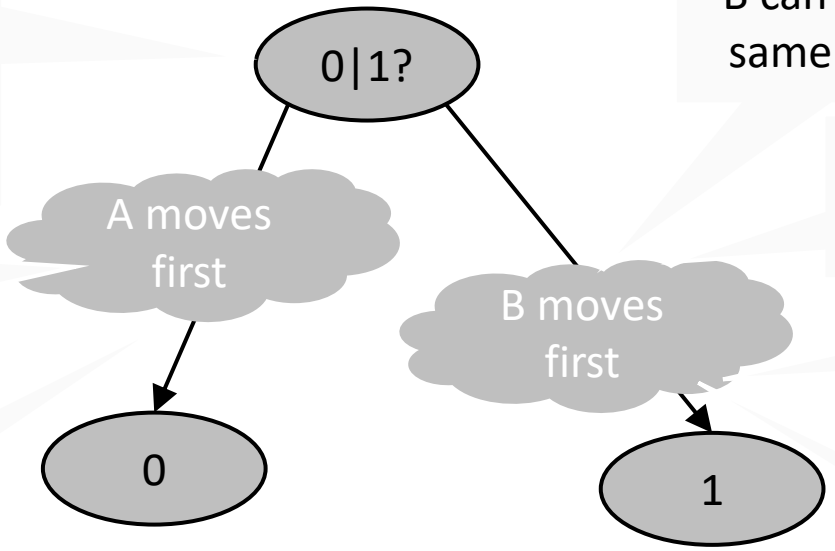
**Conclusion: First instruction after critical state must be a read or write of a shared variable!**

# Impossibility Proof Setup – Possible actions of a thread

We know reading/writing local variables cannot lead out of a critical state – what remains?

A can read a shared variable

A can write a shared variable




B can read the same variable

B can read a different variable

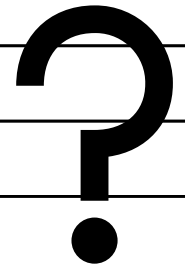
B can write the same variable

B can write a different variable

**Many cases…**
**let's make tables**

# Many Cases to check

| | | First Action | | | |
|---|---|---|---|---|---|
| | | A: r1.read() | A: r1.write() | A: r1.write() | A: r2.write() |
| Second Action | B: r1.read() | | | | |
| | B: r2.read() | | | | |
| | B: r1.write() | | | | |
| | B: r2.write() | | | | |



| | | Second Action | | | |
|---|---|---|---|---|---|
| | | A: r1.read() | A: r2.read() | A: r1.write() | A: r2.write() |
| First Action | B: r1.read() | | | | |
| | B: r2.read() | | | | |
| | B: r1.write() | | | | |
| | B: r2.write() | | | | |

**Managable… Let's look at the cases where A reads**

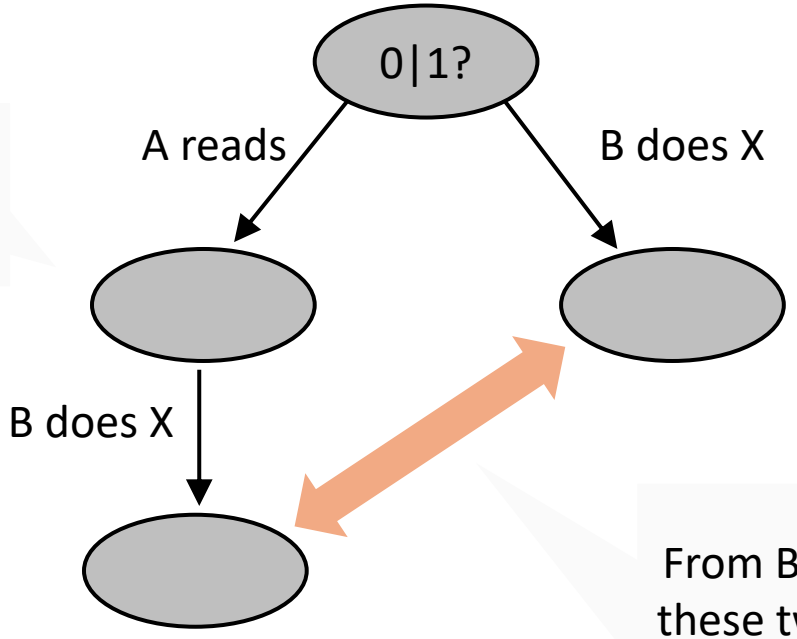
Is binary consensus possible for any of those?

Can we simplify somehow?

Let's say A always moves first, otherwise, switch names.

Similarly, we can call the register A reads r1 in both cases.

# Impossibility Proof Case I: A reads




Output is decided (0) due to critical state.

Output is decided (1) due to critical state.

From B's perspective these two states look exactly the same!

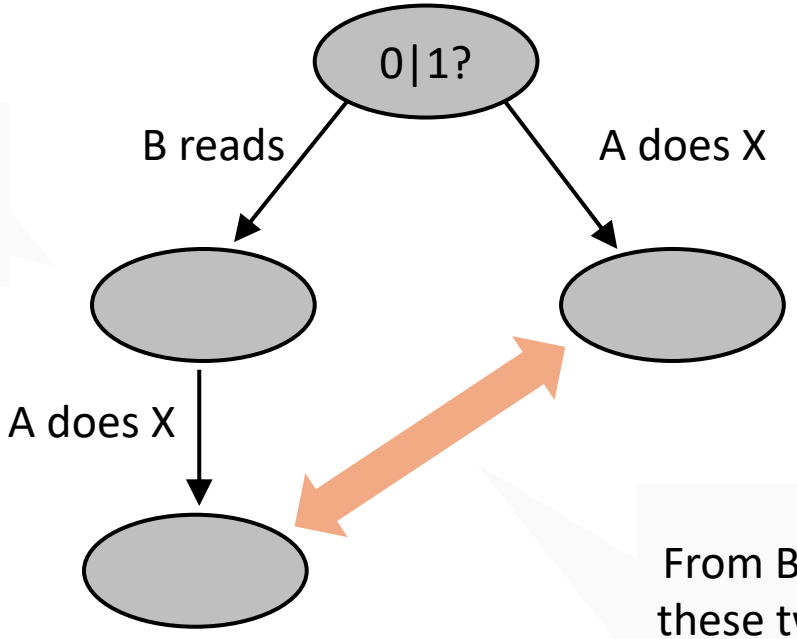
**However B needs to output different values!**

# What did we just prove?

| | | First Action | |
|---|---|---|---|
| | | A: r1.read() | A: r1.write() |
| Second Action | B: r1.read() | No, Case I | |
| | B: r2.read() | No, Case I | ? |
| | B: r1.write() | No, Case I | |
| | B: r2.write() | No, Case I | |

Is binary consensus possible for any of those?

# Impossibility Proof Case I': B reads




Output is decided (0) due to critical state.

Output is decided (1) due to critical state.

From B's perspective these two states look exactly the same!

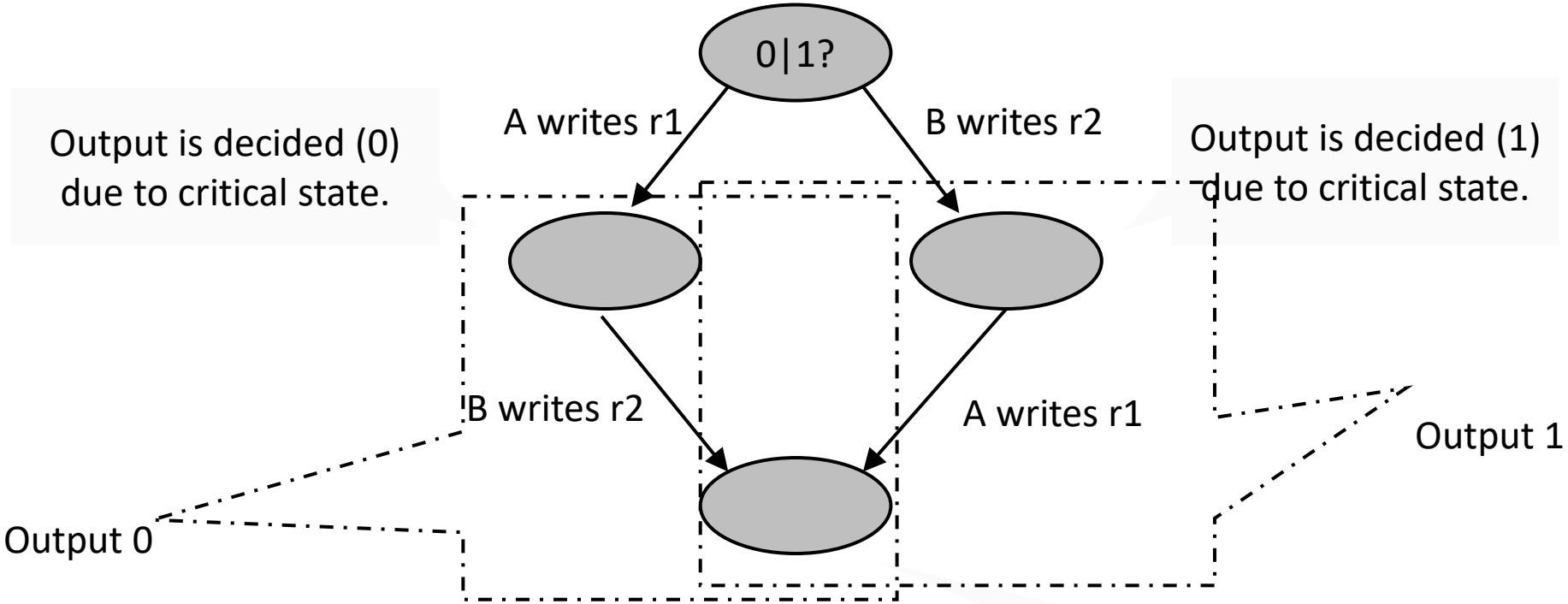
**However A needs to (eventually) output different values!**

# What did we just prove?

| | | First Action | |
|---|---|---|---|
| | | A: r1.read() | A: r1.write() |
| Second Action | B: r1.read() | No, Case I | No, Case I’ |
| | B: r2.read() | No, Case I | No, Case I’ |
| | B: r1.write() | No, Case I | ? |
| | B: r2.write() | No, Case I | ? |

Is binary consensus possible for any of those?

# Impossibility Proof Case II: A and B write to different registers




Exactly the same state!

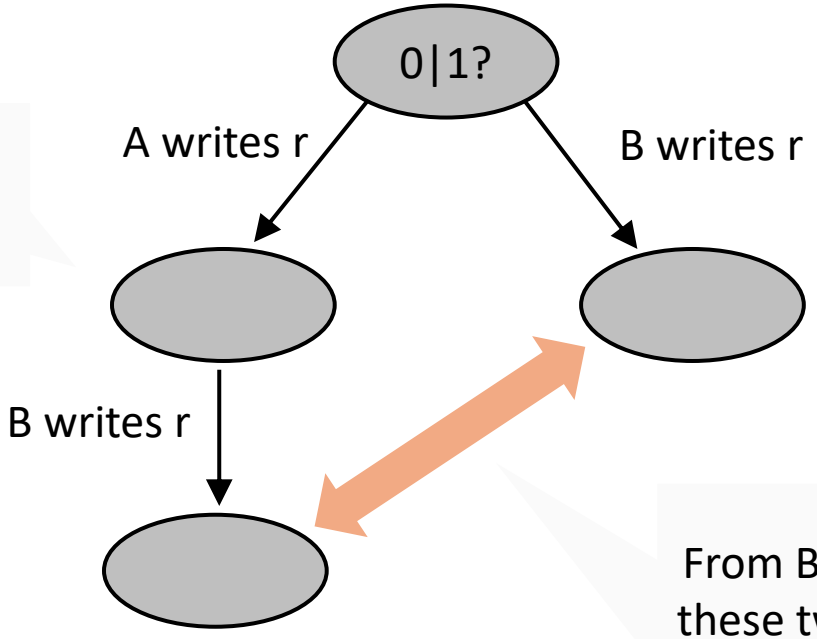
**However it should be outputting 0 / 1 depending on where it was reached from!**

# What did we just prove?

| | | First Action | |
|---|---|---|---|
| | | A: r1.read() | A: r1.write() |
| Second Action | B: r1.read() | No, Case I | No, Case I’ |
| | B: r2.read() | No, Case I | No, Case I’ |
| | B: r1.write() | No, Case I | ? |
| | B: r2.write() | No, Case I | No, Case II |

Is binary consensus possible for any of those?

# Impossibility Proof Case III: A and B write to the same register




Output is decided (0) due to critical state.

Output is decided (1) due to critical state.

From B's perspective these two states look exactly the same!

**However B needs to output different values!**

# That’s all

| | | First Action | |
|---|---|---|---|
| | | A: r1.read() | A: r1.write() |
| Second Action | B: r1.read() | No, Case I | No, Case I’ |
| | B: r2.read() | No, Case I | No, Case I’ |
| | B: r1.write() | No, Case I | No, Case III |
| | B: r2.write() | No, Case I | No, Case II |

Is binary consensus possible for any of those?

**No**

1985, 2.5k citations

**Impossibility of Distributed Consensus with One Faulty Process**

MICHAEL J. FISCHER

*Yale University, New Haven, Connecticut*

NANCY A. LYNCH

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

MICHAEL S. PATERSON

*University of Warwick, Coventry, England*

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be

# Plan für heute

- Organisation
- Nachbesprechung Assignment 12
- Theory
- **Intro Assignment 13**
- Kahoot
- Exam questions

# Assignment 13

- Is about SC and linearizability
- Use my slides and definitions if you struggle
- Histories and their properties:
  - Sequential Consistency
  - Linearizability
  - Equivalence
  - Completeness
  - etc.

# Assignment 13

- Is about SC and linearizability

### Sequential Consistency

For each of the following histories, indicate if they are sequentially consis
objects r and s are registers (initially zero), q is a FIFO (initially empty).

```
A:   --|r.write(1)|------------------------
B:   ------|r.read():0|--------------------
C:   --------------------|r.read():1|------
```

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

```
A:   --|s.write(1)|------------------------
B:   ------|r.read():0|--------------------
C:   --------------------|r.read():1|------
```

```
A:   --|s.write(1)|------------------------
B:   ------|r.read():1|---|r.read():0|------
```

### Linearizability

Which of the following histories are linearizable? Infer the object type from the supp
registers are initially zero, stacks/queues initially empty.

```
A: s.push(1)
A: void
B: s.push(2)
B: void
B: s.pop()
A: s.pop()
B: 1
A: 2
```

```
A:   --|s.write(1)|------------------------
B:   ------|r.read():1|---|r.read():0|------
```

### Equivalence

# Recap Histories

Histories can be categorized by some fundamental properties:

**Sequential**: 1st action invocation; no interleavings
**Complete**: no pending invocations
**Equivalence to some other History**: for all threads A: H|A = G|A
**Legal**: for all objects r: H|r is sequential and correct
**Well formed**: for all threads A: H|A is sequential
**Quiescent Consistent**: correct with reordering of “overlapping” calls
**Sequentially Consistent**: correct with reordering regarding threads
**Linearizable**: choosing linearization points to make execution correct

Thanks to @Erxuan Li, PProg25

Note: the above definitions are not formal

# Questions

# LOCK FREE LIST-BASED SET

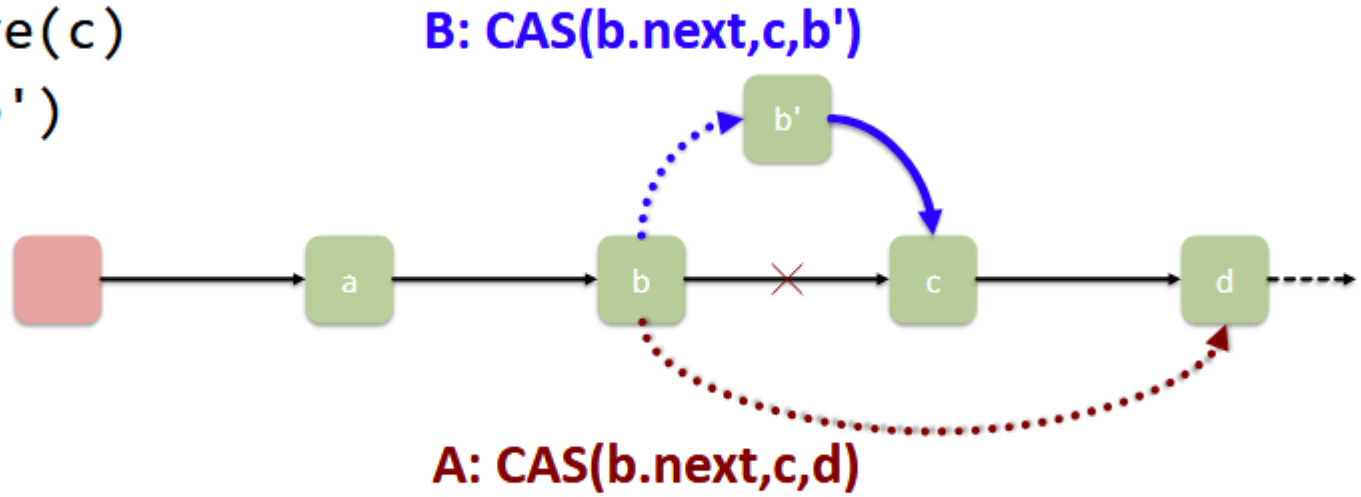
**(NOT SKIP LIST!)**

Some of the material from "Herlihy: Art of Multiprocessor Programming"

# First Approach, simple CAS for remove

**Does this work?**

A: remove(c)
B: add(b')




ok?

CAS decides who wins → this seems to work

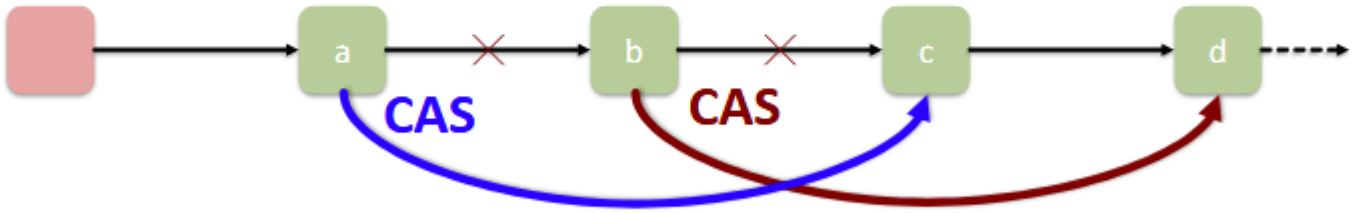
So does this CAS approach work generally??

- Note that CAS(b.next,c,b') means if b.next == c then set b.next to b' otherwise don't do anything

# First Approach, simple CAS for remove

**Another scenario**

```
A: remove(c)
B: remove(b)
```




c not deleted! ☹

- We read a stale value for b.next = c! Thread B will do CAS(a.next,b,c)

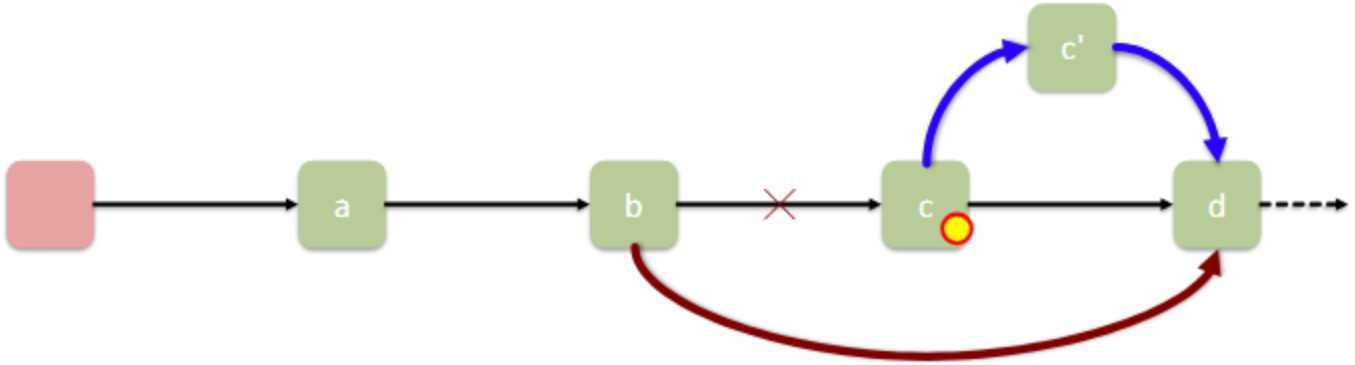
# Second Approach, try to fix the issue by using mark bit which tells us if an element was removed

A: remove(c)

B: add(c')

B: c.mark ?

B: CAS(c.next,d,c')



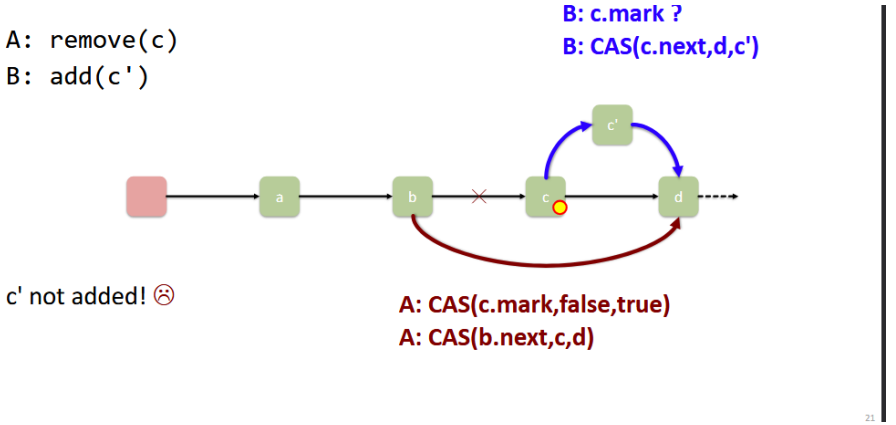

c' not added! ☹

A: CAS(c.mark,false,true)

A: CAS(b.next,c,d)

# Second Approach, try to fix the issue by using mark bit which tells us if an element was removed

- Thread b checks if node c is removed, sees mark bit is false, proceeds
- Now thread a wants to remove c, sets mark bit of node c
- Reads c.next = d and does a cas(b.next,c,d), which succeeds.
- Node c is removed
- Now thread b does CAS(c.next,d,c'). c' is inserted but not in the list, as c got removed



**The problem**

The difficulty that arises in this and many other problems is:

- We cannot (or don't want to) use synchronization via locks
- We still want to atomically establish consistency **of two things**
  Here: mark bit & next-pointer

- We want to set the mark bit and the next pointer in one step otherwise we face the issue from the previous slide

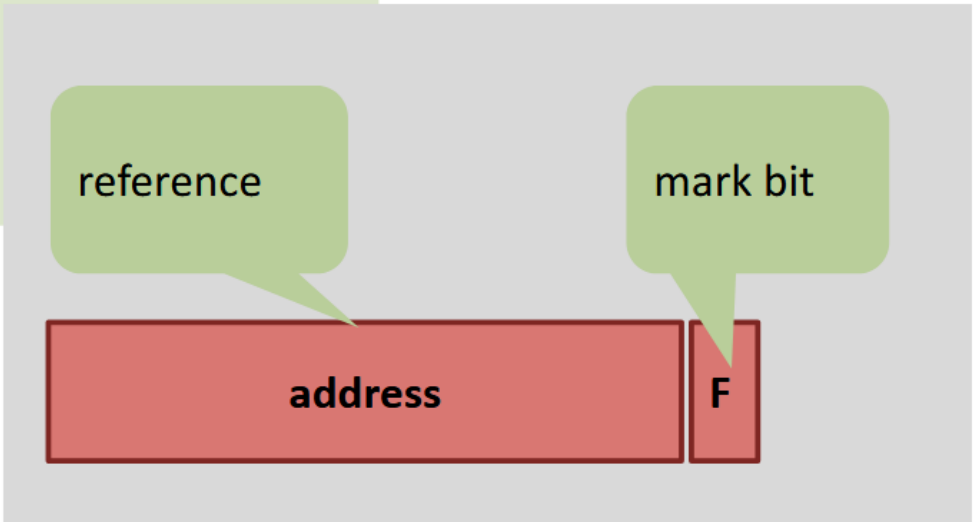
- We want to set the mark bit and the next pointer in one step otherwise we face the issue from the previous slide

- So how do we do this?

# The Java solution

```
Java.util.concurrent.atomic
AtomicMarkableReference<V> {
    boolean attemptMark(V expectedReference, boolean newMark)
    boolean compareAndSet(V expectedReference, V newReference,
                          boolean expectedMark, boolean newMark)
    V get(boolean[] markHolder)
    V getReference()
    boolean isMarked()
    set(V newReference, boolean newMark)
}
```

DCAS on V and mark

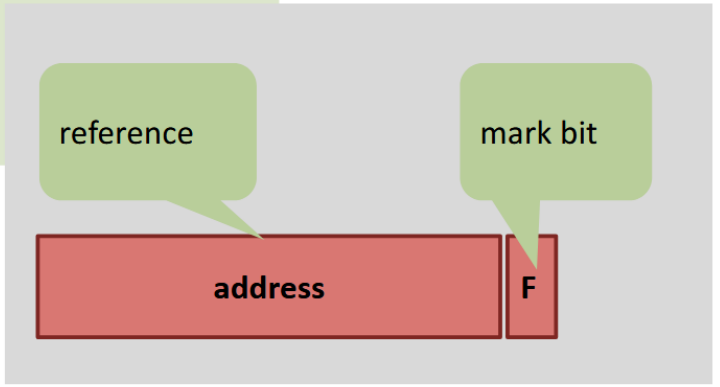



$2^{64}$Bytes=562,949,953,421,312 Petabytes

## The Java solution

```
Java.util.concurrent.atomic
AtomicMarkableReference<V> {
    boolean attemptMark(V expectedReference, boolean newMark)
    boolean compareAndSet(V expectedReference, V newReference,
                          boolean expectedMark, boolean newMark)
    V get(boolean[] markHolder)
    V getReference()
    boolean isMarked()
    set(V newReference, boolean newMark)
}
```

DCAS on V and mark




$2^{64}$Bytes=562,949,953,421,312 Petabytes

- The mark bit we were talking about is just hidden in the reference now! E.g. c.mark is now hidden in c.next!
- We can update a reference with a single CAS!

## The algorithm using AtomicMarkableReference

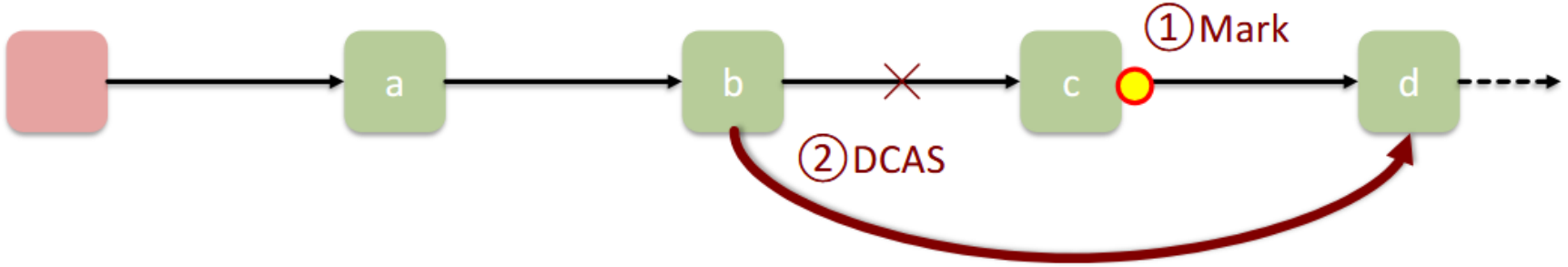
- Atomically
  - Swing reference *and*
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

## Algorithm idea

A: remove(c)

Why "try to"? How can it fail? What then?

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,unmarked], [d,unmarked]);

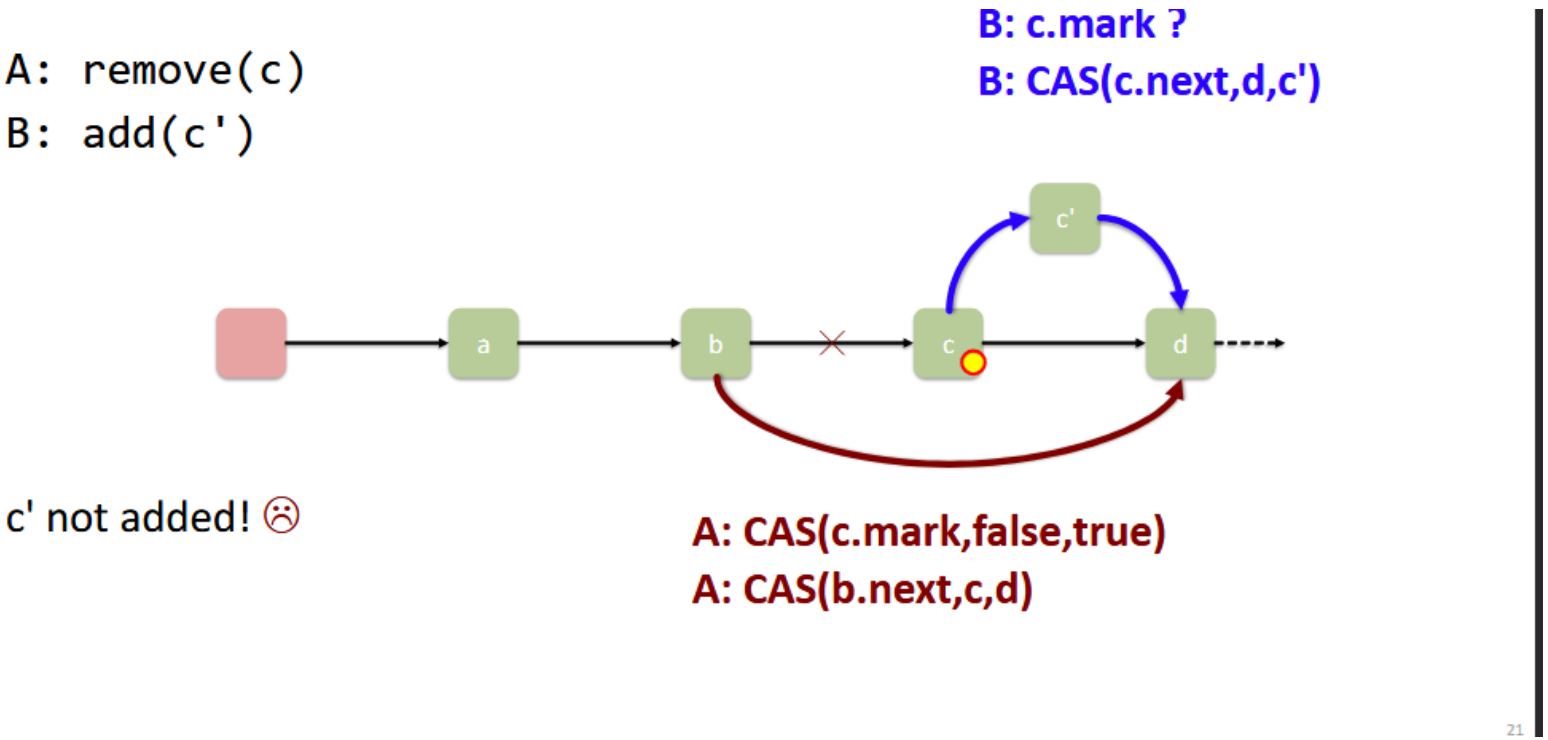



- Note that

CAS( [ b.next.reference, b.next.marked ], [ c,unmarked] , [d, unmarked] )

- checks if b.next = c and b.mark = 0 (unmarked = not removed) then set b.next = d and leave b.mark = 0 (unmarked)

# Did it fix our previous problem?





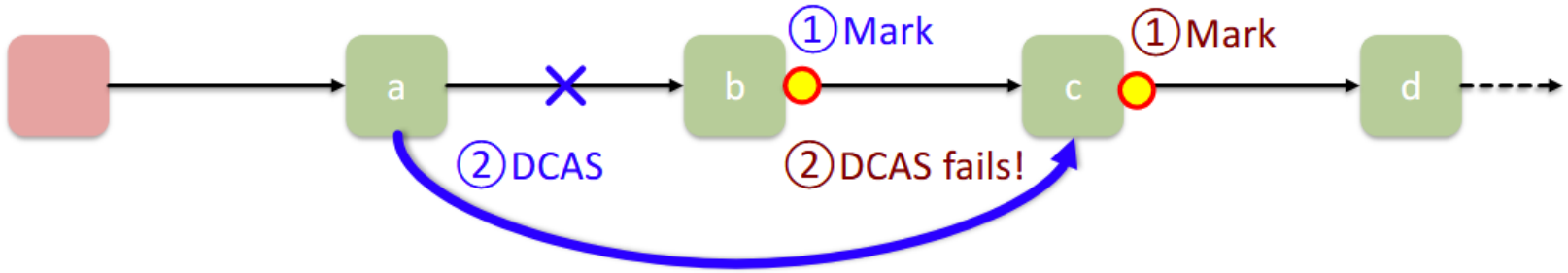

- Yes, we can't have bad interleaving anymore because thread B checks c.mark and updates c.next in one step
- Thread B will either see mark = 0 → can insert c' in one step or mark = 1 which means it needs to retry

# Remove , remove case

**It helps!**

```
A: remove(c)
B: remove(b)
```

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,**unmarked**], [d,unmarked]);




Results in node C not being removed but still marked!

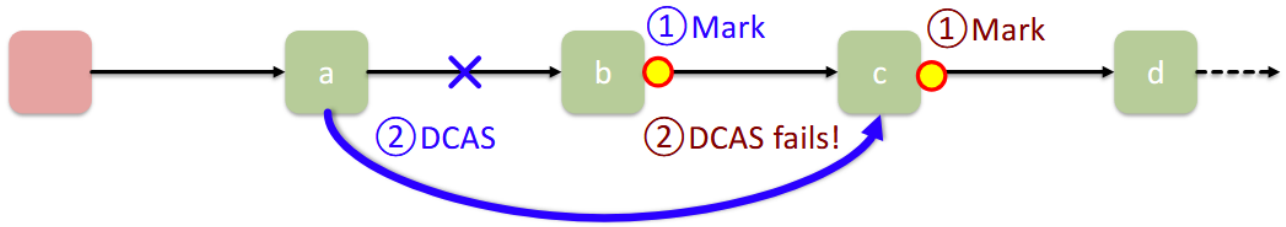
1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

**It helps!**

A: remove(c)

B: remove(b)

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,**unmarked**], [d,unmarked]);




Results in node C not being removed but still marked!

1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

- If we implement our methods correctly and watch out for mark bit being set, this is fine, i.e., this implementation works
- However, we would like marked nodes to be removed physically at some point too

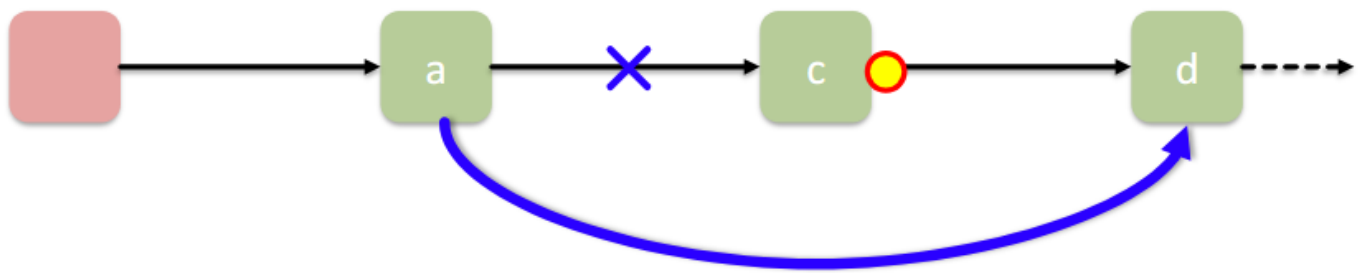
## Traversing the list

Q: what do you do when you find a “logically” deleted node in your path?

A: finish the job.

CAS the predecessor’s next field

Proceed (repeat as needed)



# Find node

```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    while (true) {
        pred = head;
        curr = pred.next.getReference();
        boolean done = false;
        while (!done) {
            marked = curr.next.get(marked);
            succ = marked[1:n]; // pseudo-code to get next ptr
            while (marked[0] && !done) { // marked[0] is marked bit
                if pred.next.compareAndSet(curr, succ, false, false) {
                    curr = succ;
                    marked = curr.next.get(marked);
                    succ = marked[1:n];
                }
                else done = true;
            }
            if (!done && curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
} } }
```

loop over nodes until position found

if marked nodes are found, delete them, if deletion fails restart from the beginning

```
class Window {
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        this.pred = pred;
        this.curr = curr;
    }
}
```

# Remove

```
public boolean remove(T item) {
   Boolean snip;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) {
         return false;
      } else {
         Node succ = curr.next.getReference();
         snip = curr.next.attemptMark(succ, true);
         if (!snip) continue;
         pred.next.compareAndSet(curr, succ, false, false);
         return true;
      }
   }
}
```

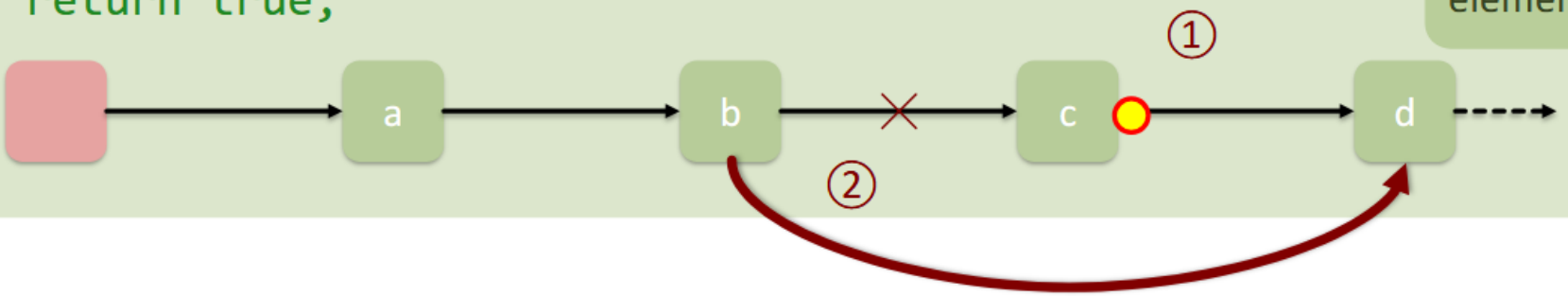
Find element and prev element from key

If no such element -> return false

Otherwise try to logically delete (set mark bit). ①

If no success, restart from the very beginning

Try to physically delete the element, ignore result ②



# Add

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

and try to insert. If insertion fails (next set by other thread or mark bit set), retry

- In our previous example

## Add

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

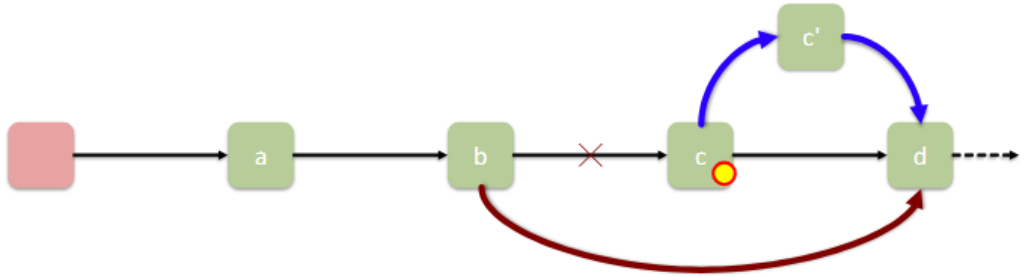
and try to insert. If insertion fails (next set by other thread or mark bit set), retry

A: remove(c)

B: add(c')

B: c.mark ?

B: CAS(c.next,d,c')




c' not added! ☹

A: CAS(c.mark,false,true)

A: CAS(b.next,c,d)

## Observations

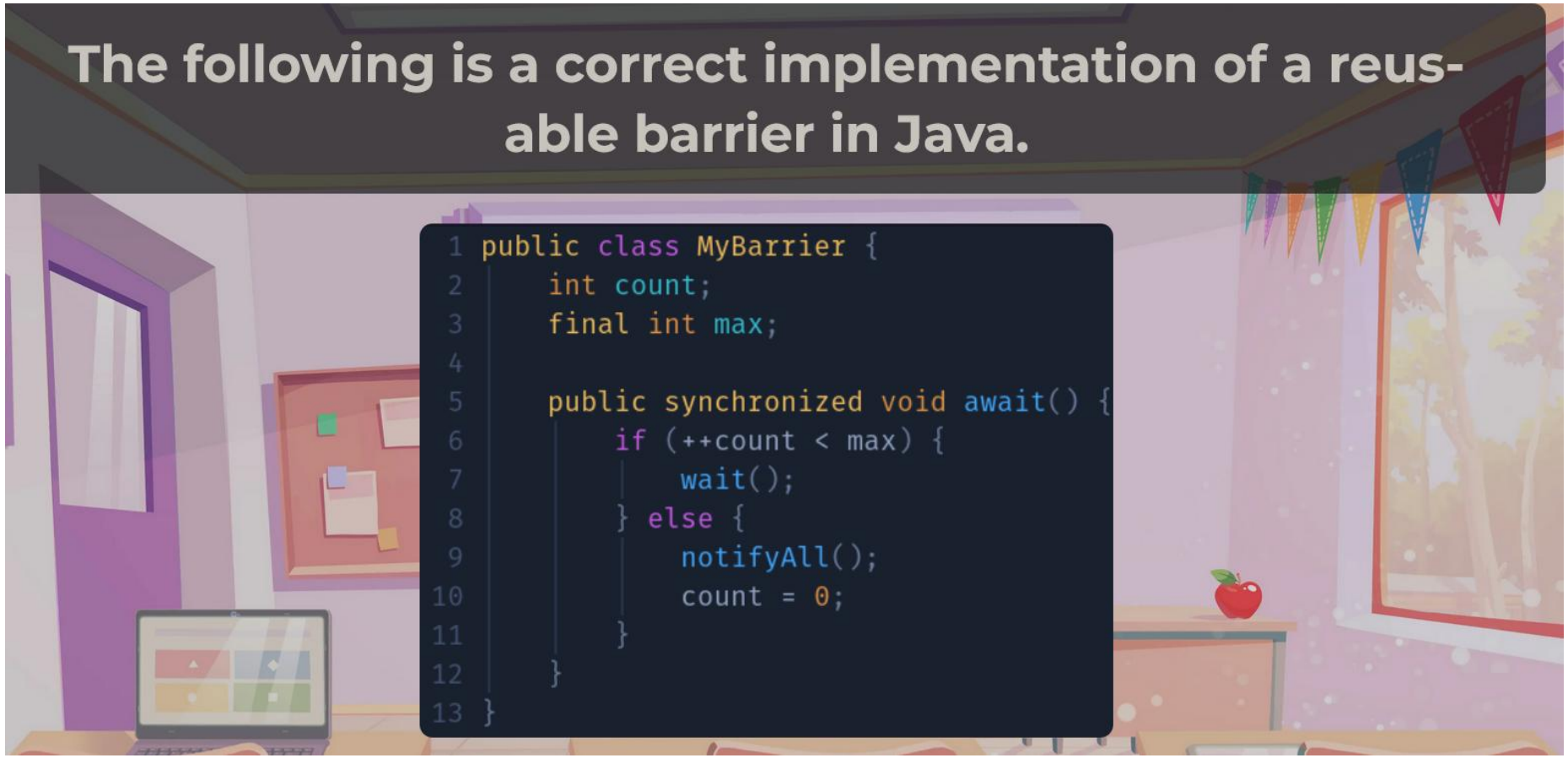
- We used a special variant of DCAS (double compare and swap) in order to be able check two conditions at once.
  This DCAS was possible because one bit was free in the reference.
- We used a lazy operation in order to deal with a consistency problem. Any thread is able to repair the inconsistency.
  If other threads would have had to wait for one thread to cleanup the inconsistency, the approach would not have been lock-free!
- This «helping» is a recurring theme, especially in wait-free algorithms where, in order to make progress, threads must help others (that may be off in the mountains ☺)

# Plan für heute

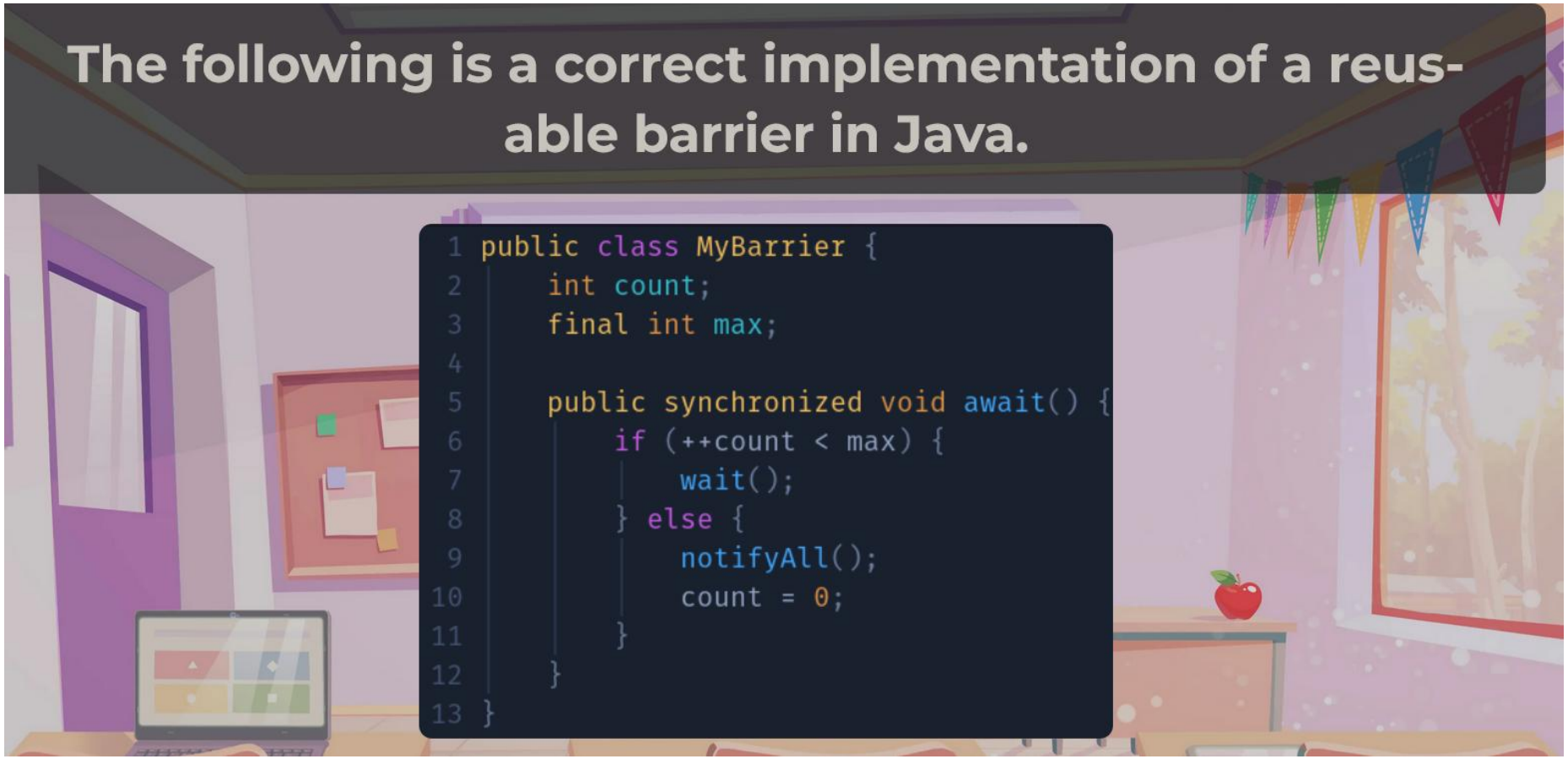
- Organisation
- Nachbesprechung Assignment 12
- Theory
- Intro Assignment 13
- **Kahoot**
- Exam questions

# Kahoot!

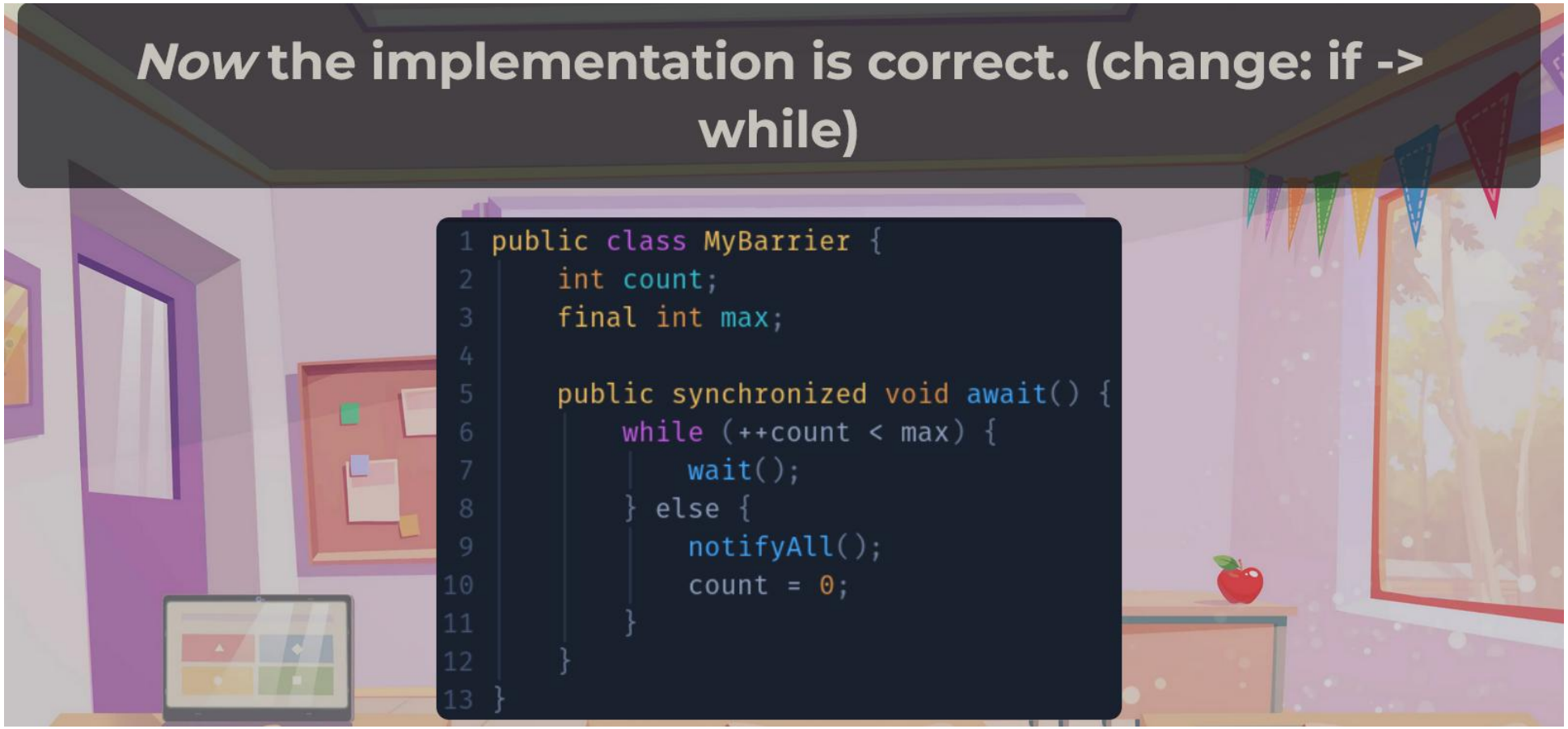
# The following is a correct implementation of a reusable barrier in Java.

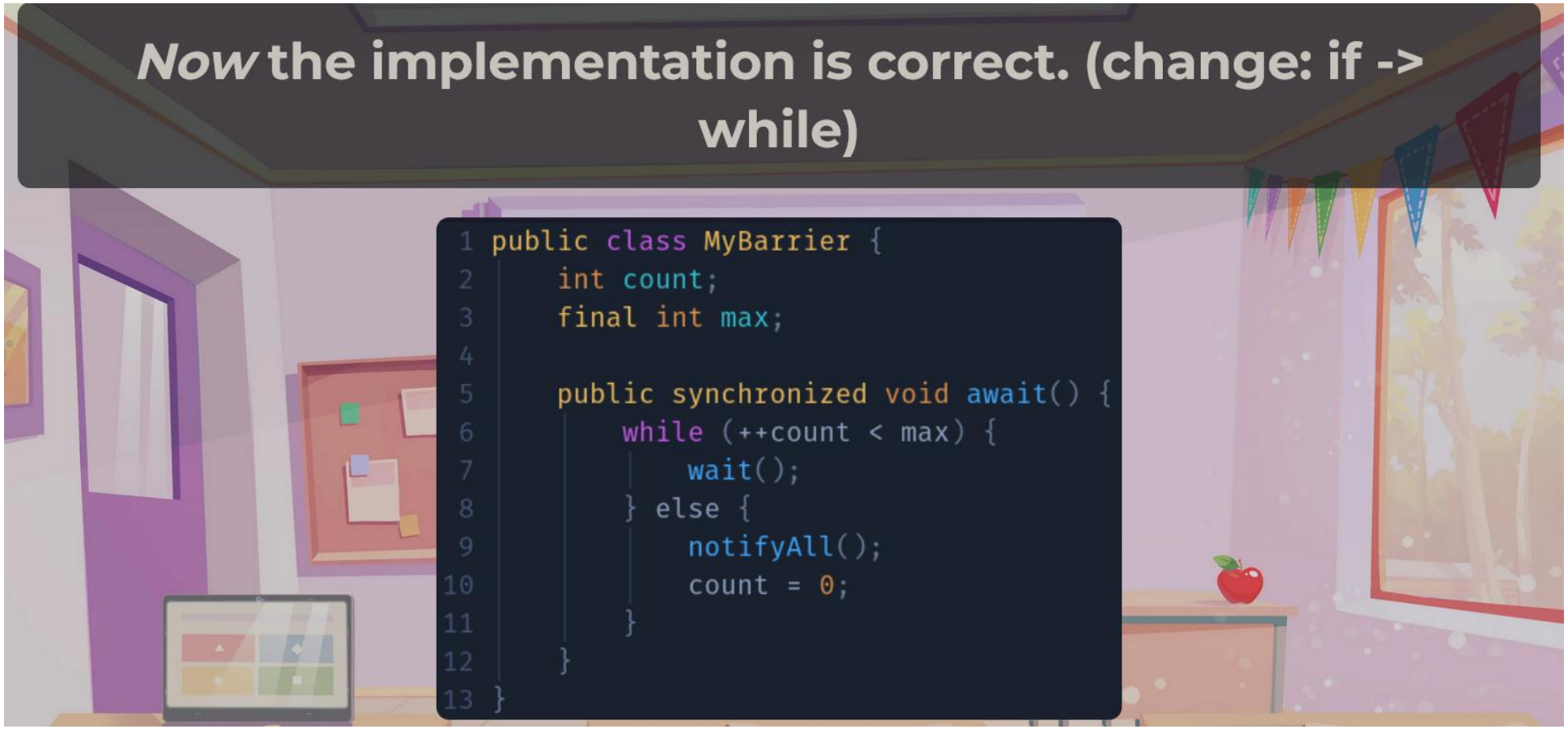


```java
public class MyBarrier {
    int count;
    final int max;

    public synchronized void await() {
        if (++count < max) {
            wait();
        } else {
            notifyAll();
            count = 0;
        }
    }
}
```




- False, remember spurious wake ups


*Now* the implementation is correct. (change: if -> while)
public class MyBarrier {
int count;
final int max;
public synchronized void await() {
while (++count < max) {
wait();
} else {
notifyAll();
count = 0;
}
}
}




- False, imagine max = 3. Then A,B,C arrive. Now C notifies threads A and B but sets count to 0. Thus, only C can leave the barrier while A and B are still stuck.

# Plan für heute

- Organisation
- Nachbesprechung Assignment 12
- Theory
- Intro Assignment 13
- Kahoot
- **Exam questions**

# Types of exercises that might come in the exam

*Disclaimer: This list is not guaranteed to be complete and is only meant to give you an idea of what has been asked on previous exams.*

**Locks**

- Usually there are not too many question on this topic. true/false questions of which lock has which properties (fairness, starvation free)
- find bug in lock code (violation of mutual exclusion or deadlock freedom)
- draw state space diagram and/or read off correctness properties
- reproduce Peterson/Filter/Bakery lock
- prove correctness of Peterson lock or similar (but not Filter or Bakery)

**Monitors, semaphores, barriers**

- semaphore implementation (mostly with monitors)
- (never seen rendezvous with semaphores in an exam)
- barrier implementation (mostly with monitors)
- (only seen a task on implementing a barrier with semaphores *once* in FS21, 8b)
- fill out some program using monitors (similar to wait/notify exercises, maybe with lock conditions)

Credits @aellison PProg23

# Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.* (4)

i.
```
1   class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5           int cur_threads = i.incrementAndGet();
6           if(cur_threads < threads) {
7             while (i.get() < threads) {}
8           }
9       }
10  }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

## Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.*

(4)

i.
```
1  class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5               int cur_threads = i.incrementAndGet();
6               if(cur_threads < threads) {
7                 while (i.get() < threads) {}
8               }
9       }
10 }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

True, there is no data race since `incrementAndGet` increases `i` atomically.

## Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.* (4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5          int cur_threads = i.incrementAndGet();
6          if(cur_threads < threads) {
7            while (i.get() < threads) {}
8          }
9      }
10 }
```

◯ Der Code beendet sich immer. *Code will always complete.*

## Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.*

(4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5              int cur_threads = i.incrementAndGet();
6              if(cur_threads < threads) {
7                while (i.get() < threads) {}
8              }
9      }
10 }
```

◯ Der Code beendet sich immer. *Code will always complete.*

True, it is a correct barrier implementation.

# Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.*

(4)

i.
```
1  class Barrier {
2      AtomicInteger i = new AtomicInteger(0);
3      final int threads = N;
4      public void await() throws InterruptedException {
5          int cur_threads = i.incrementAndGet();
6          if(cur_threads < threads) {
7            while (i.get() < threads) {}
8          }
9      }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

*Code might not use compute resources efficiently. Why?*

## Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

*We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why.*

(4)

i.
```
1  class Barrier {
2       AtomicInteger i = new AtomicInteger(0);
3       final int threads = N;
4       public void await() throws InterruptedException {
5               int cur_threads = i.incrementAndGet();
6               if(cur_threads < threads) {
7                 while (i.get() < threads) {}
8               }
9       }
10 }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

*Code might not use compute resources efficiently. Why?*

True, the waiting threads are busy waiting.

ii.

```
1  class Barrier {
2      int i = 0;
3      final int threads = N;
4      public synchronized void await() throws InterruptedException {
5          ++i;
6          while (i < threads) { wait(); }
7          notify();
8      }
9  }
```

◯ Der gezeigte Code hat die gewünschte Semantik.

*Code has the desired semantics.*

ii.

```
1  class Barrier {
2      int i = 0;
3      final int threads = N;
4      public synchronized void await() throws InterruptedException {
5          ++i;
6          while (i < threads) { wait(); }
7          notify();
8      }
9  }
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

Yes

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code beendet sich immer. *Code will always complete.*

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code beendet sich immer. *Code will always complete.*

True

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

*Code might not use compute resources efficiently. Why?*

ii.

```
1   class Barrier {
2       int i = 0;
3       final int threads = N;
4       public synchronized void await() throws InterruptedException {
5           ++i;
6           while (i < threads) { wait(); }
7           notify();
8       }
9   }
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

*Code might not use compute resources efficiently. Why?*

---

False, the code makes use of wait/notify and thus does not waste compute resources.

# Feedback

- Falls ihr Feedback möchtet sagt mir bitte Bescheid!
- Schreibt mir eine Mail oder auf Discord

# Danke

- Bis nächste Woche!