Parallele Programmierung FS25

Exercise Session 14

Jonas Wetzel

Parallele Programmierung FS25

Exercise Session 14

The final edition

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- Theory
- Exam questions
- Intro Assignment 14
- Kahoot
- Extra: A&W DP Trick

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

- TA Award
- Danke!

Organisation

- Wo sind wir jetzt?

Sequential Consistency and Linearizability, Consensus
Transactional Memory, MPI

- Ende der Vorlesung! 🥹

Plan für heute

- Organisation
- **Nachbesprechung Assignment 13**
- Theory
- Exam questions
- Intro Assignment 14
- Kahoot
- A&W DP Tricks

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

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- **Theory**
- Exam questions
- Intro Assignment 14
- Kahoot
- A&W DP Trick

SC and Linearizability

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

Same reasoning doesn’t apply to concurrent objects!

- notion of an object’s state becomes confusing
- In single-threaded programs, an object must assume a meaningful state only between method calls
- For concurrent objects, however, overlapping method calls may be in progress at every instant, so the object may never be between method calls

Program correctness in a concurrent world

Essenti
al

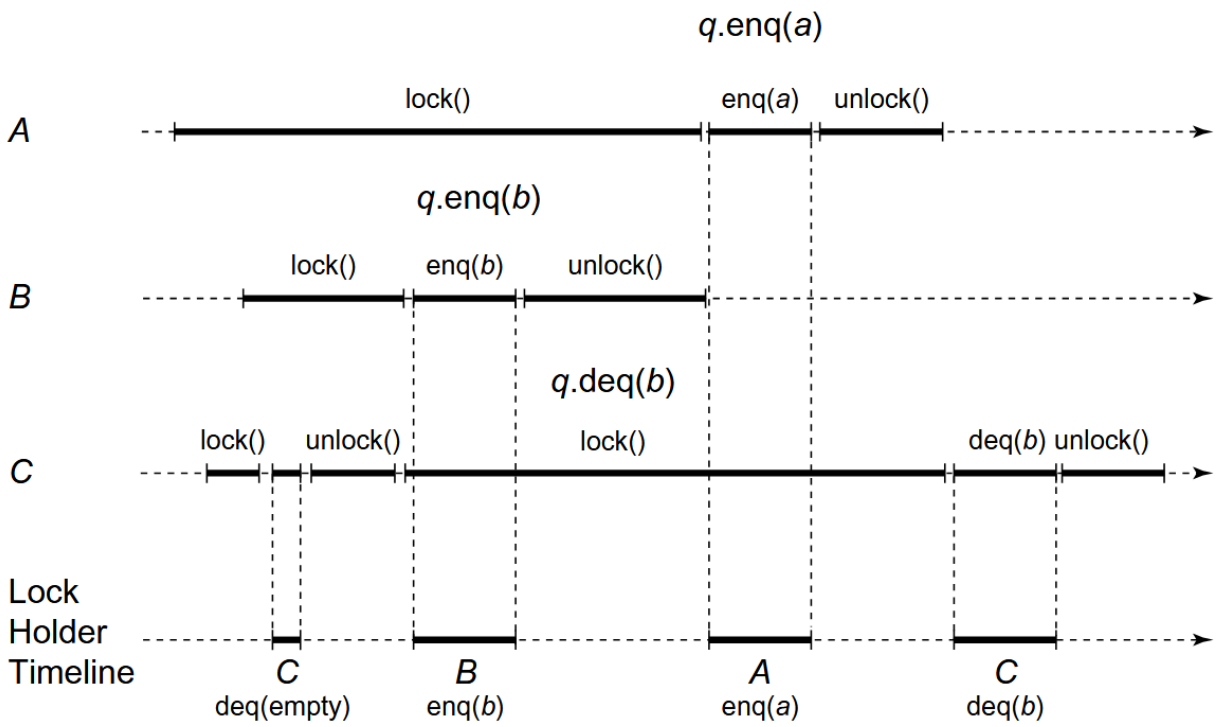
Sequential	Concurrent
Each method described independently.	Need to describe all possible interactions between methods. (what if enq and deq overlap? …)
Object's state is defined between method calls.	Because methods can overlap, the object may never be between method calls…
Adding new method does not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

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

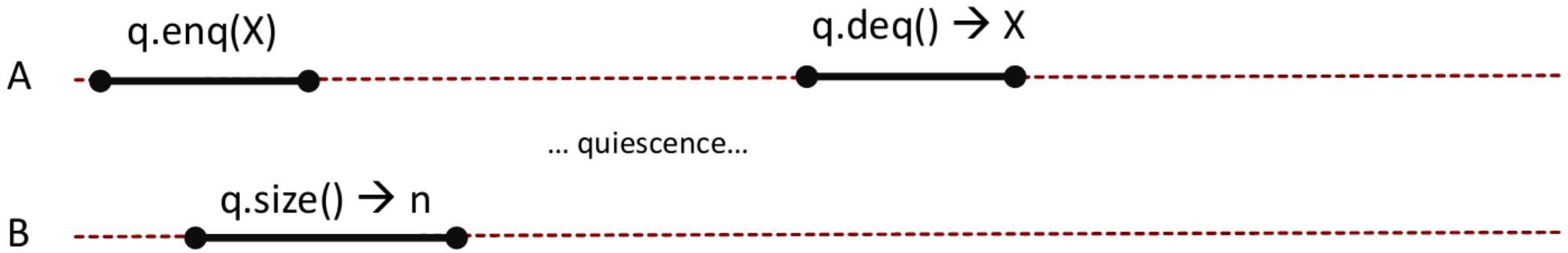
Quiescent Consistency

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

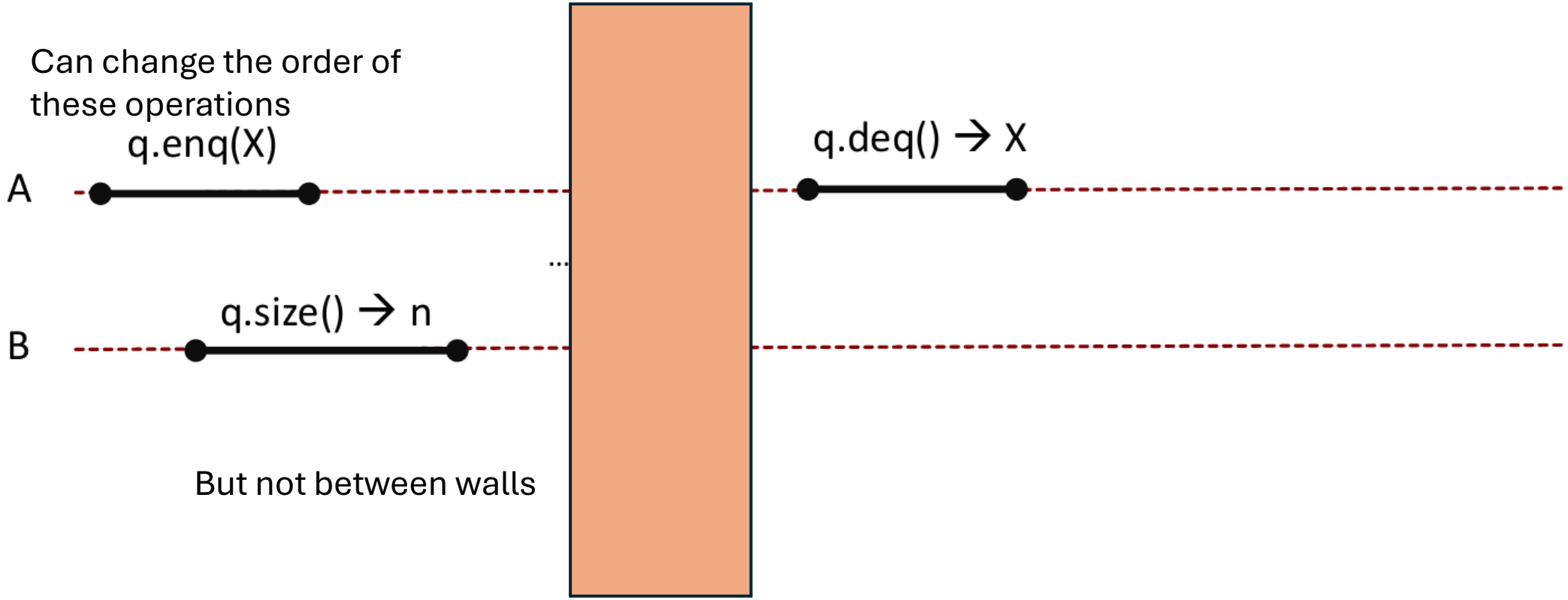
Quiescent consistency

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



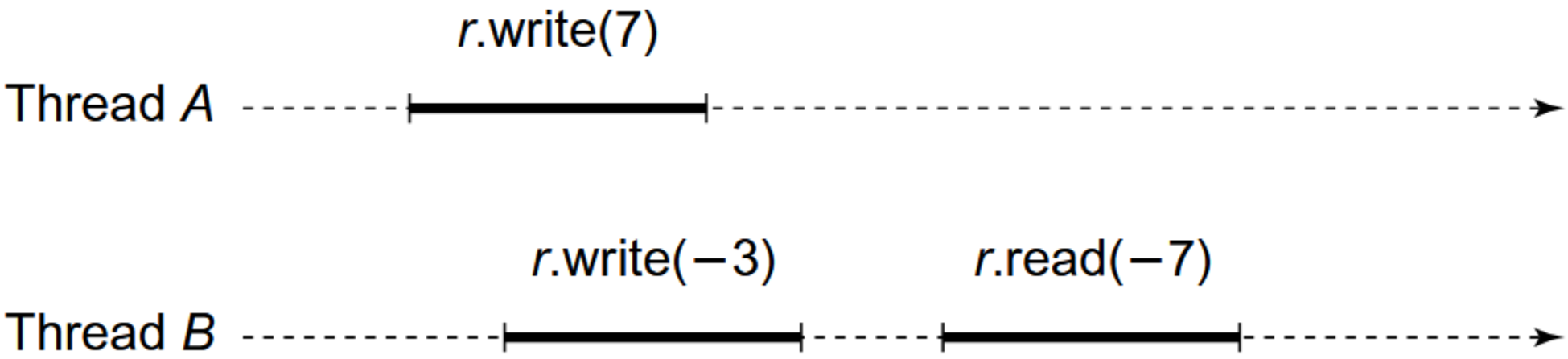
Quiescent consistency

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



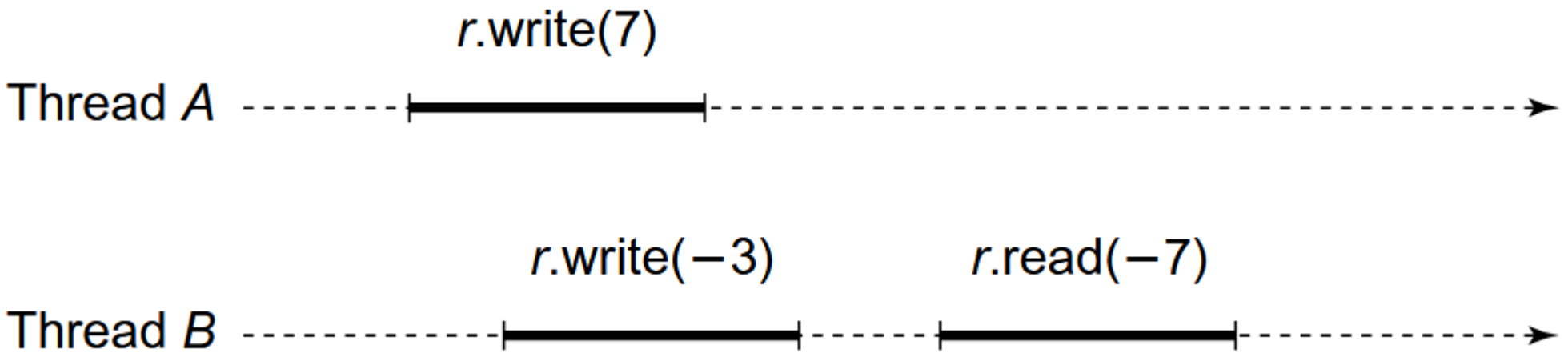
Sequential Consistency

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

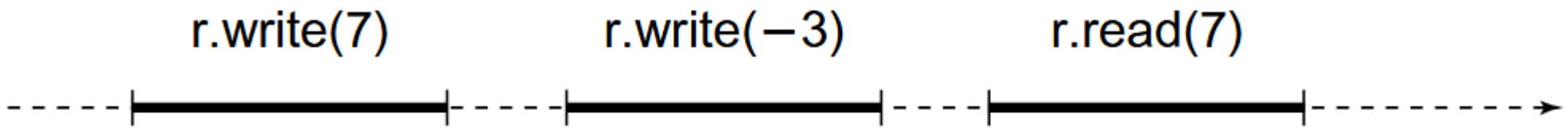


Figure 3.5 Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote.

- 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

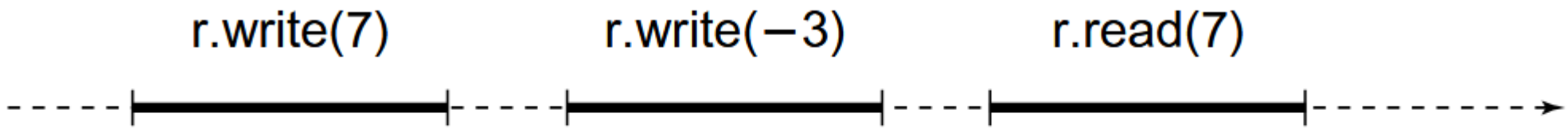


Figure 3.5 Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote.

- 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."

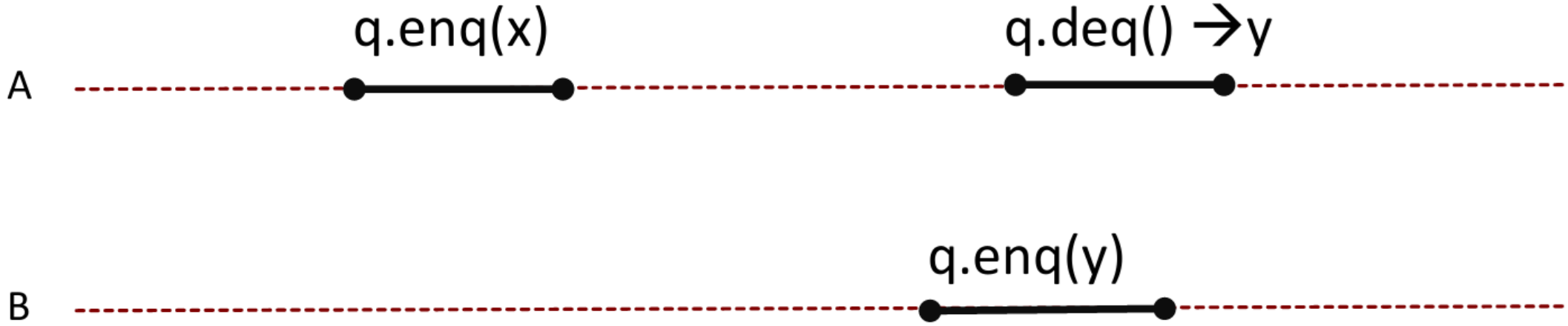
- Leslie Lamport (inventor of sequential consistency, GOAT Turing Award Winner, concurrent master mind)

Sequential consistency requirements

Essenti
al

1. All instructions are executed in order.

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

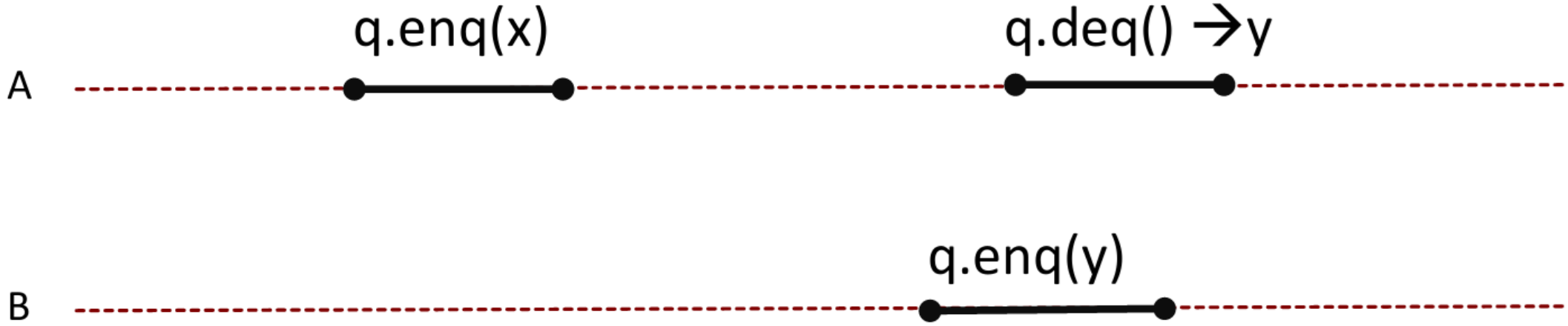


Sequential consistency requirements

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

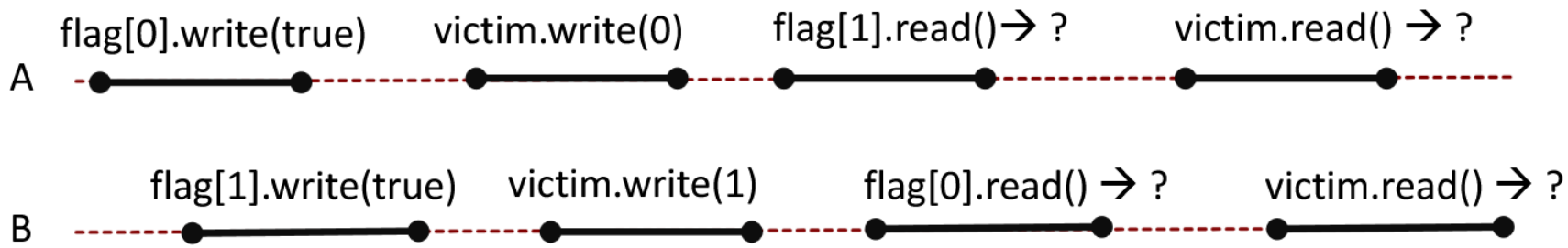
- In the real world, hardware architects do not adhere to this by default
- We need to explicitly announce that we want this property (i.e. volatile keyword)

Sequential consistency and the real world

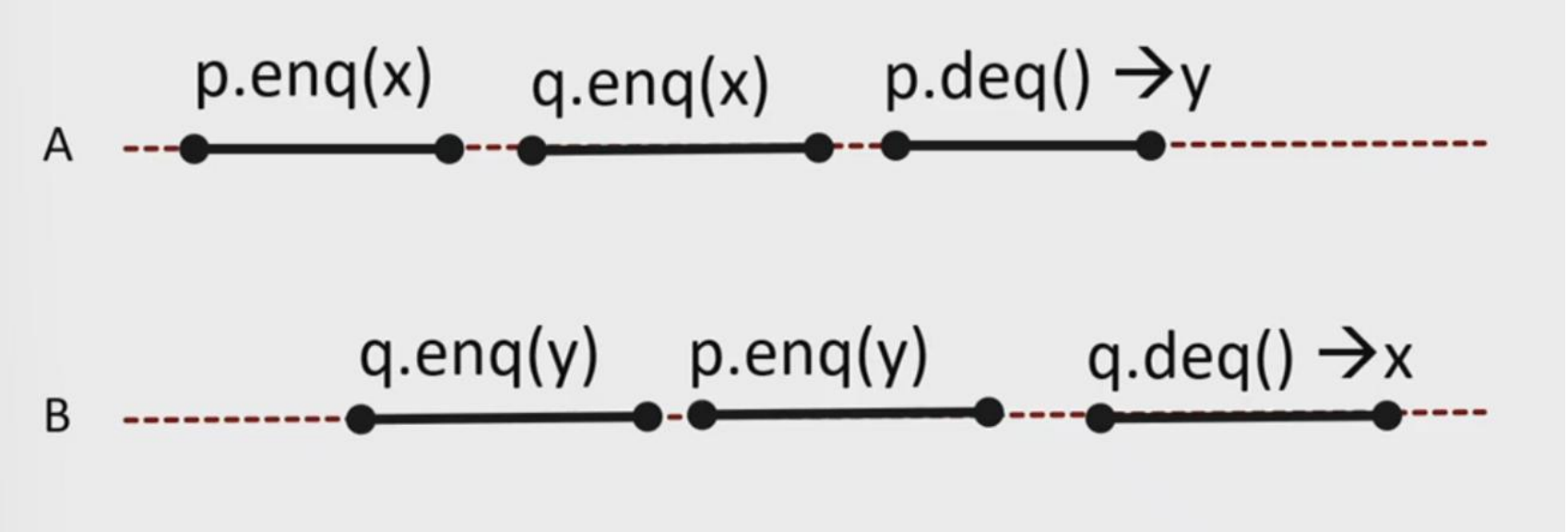
- We need to explicitly announce that we want this property (i.e. volatile keyword)
- This lock is only correct if we have SC

Reminder: Consequence for Peterson Lock (Flag Principle)

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



SC is not compositable

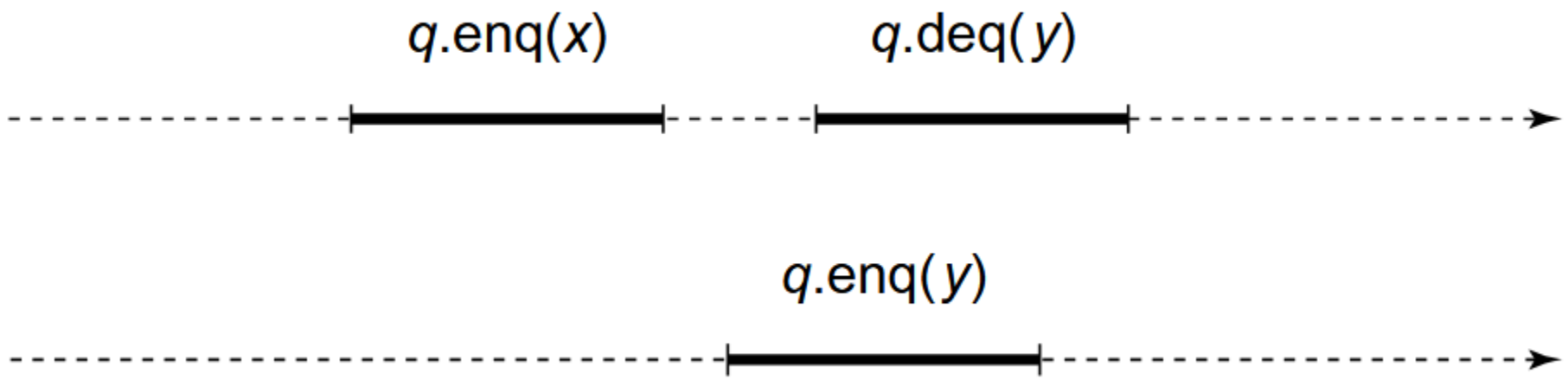


H|p is sequentially consistent
H|q is sequentially consistent
H is not sequentially consistent

In general: sequential consistency is not compositable

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

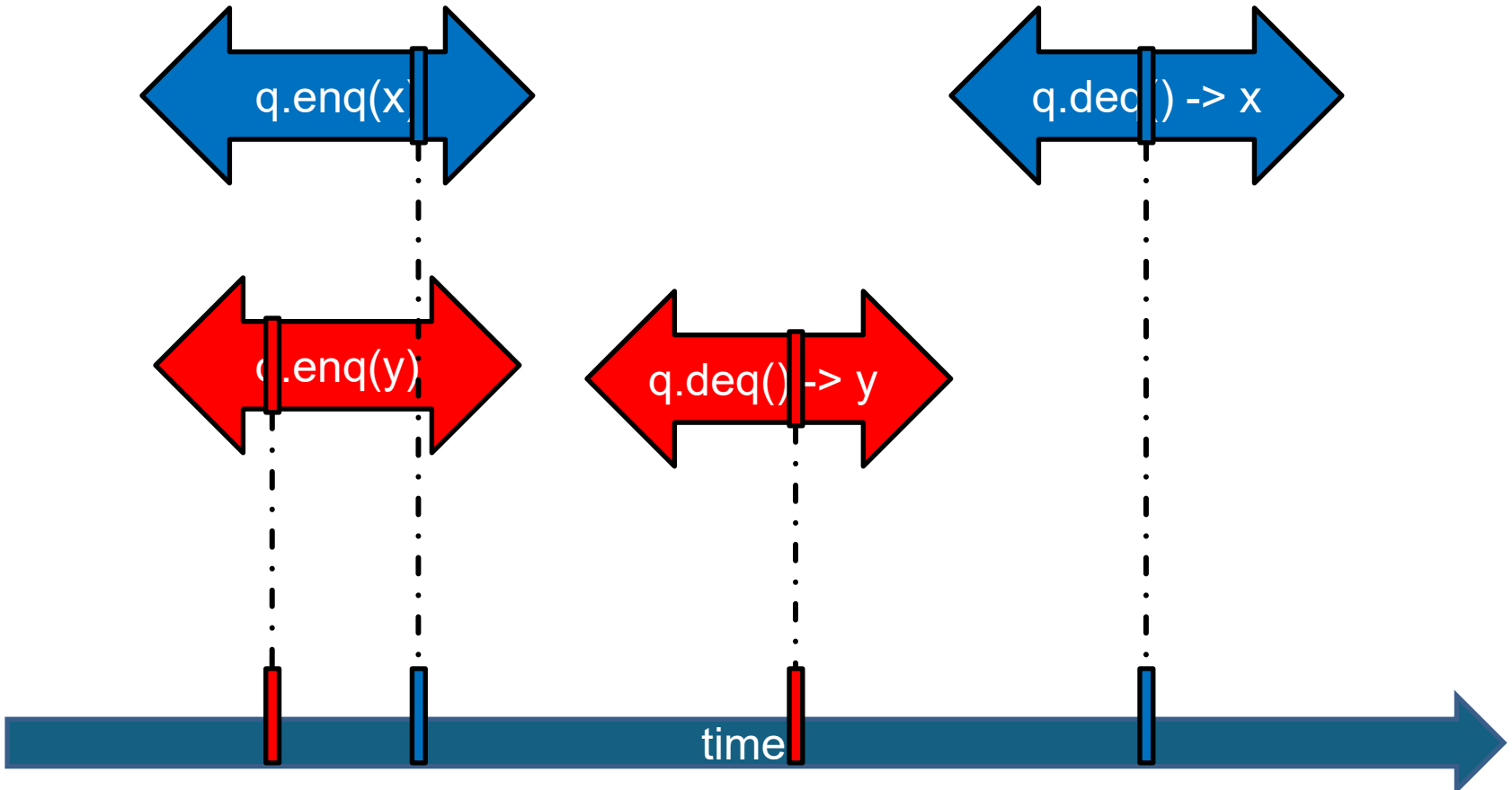
- Now we have:

- 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

Example with FIFO Queue (1)

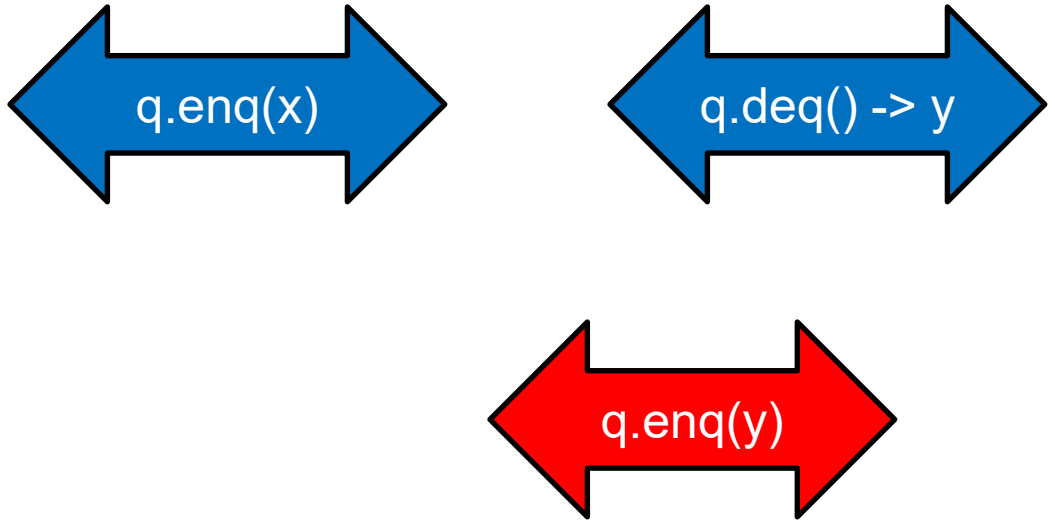
Is this linearizable?

Yes!



Example (2)

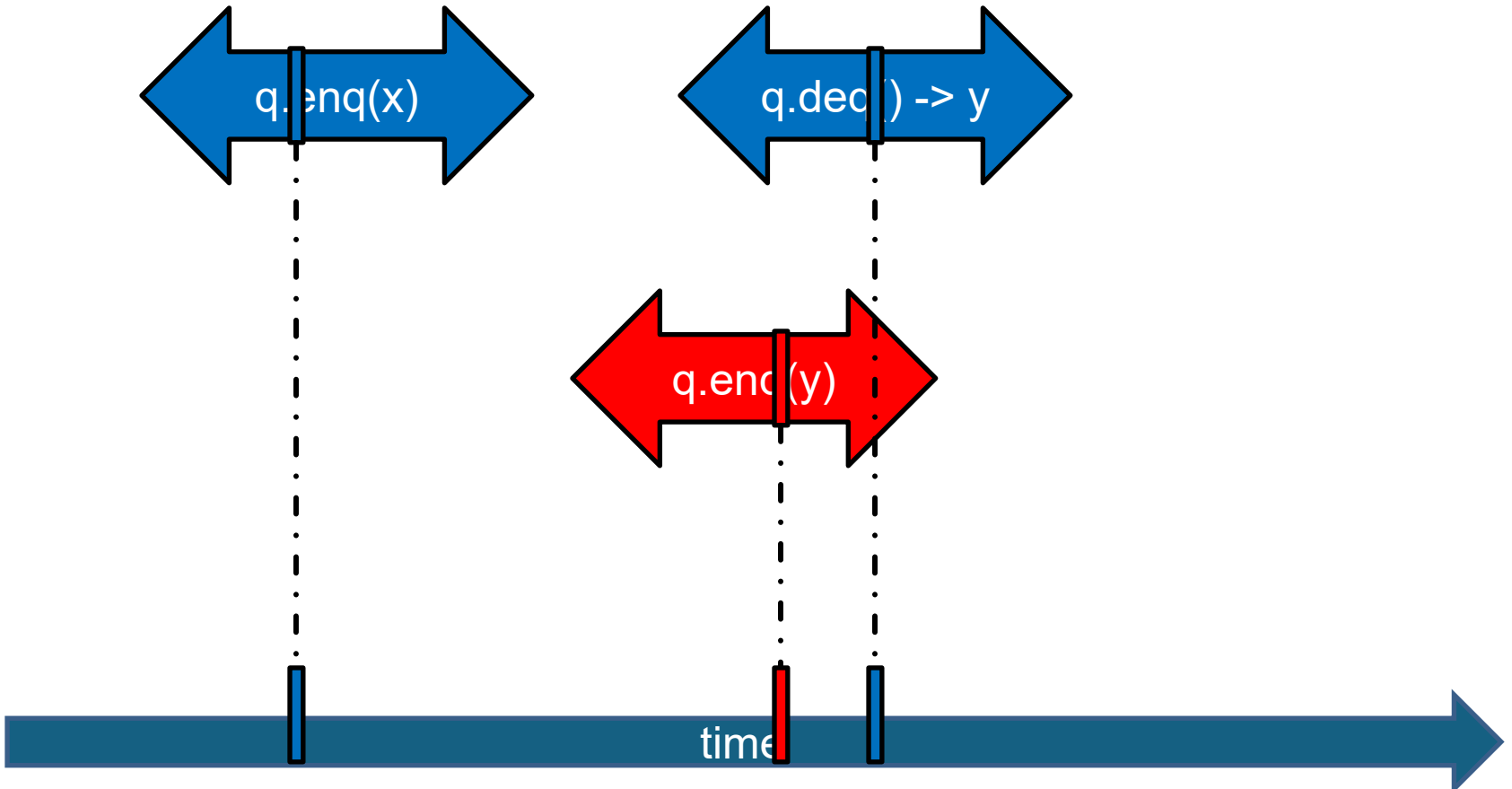
Is this linearizable?



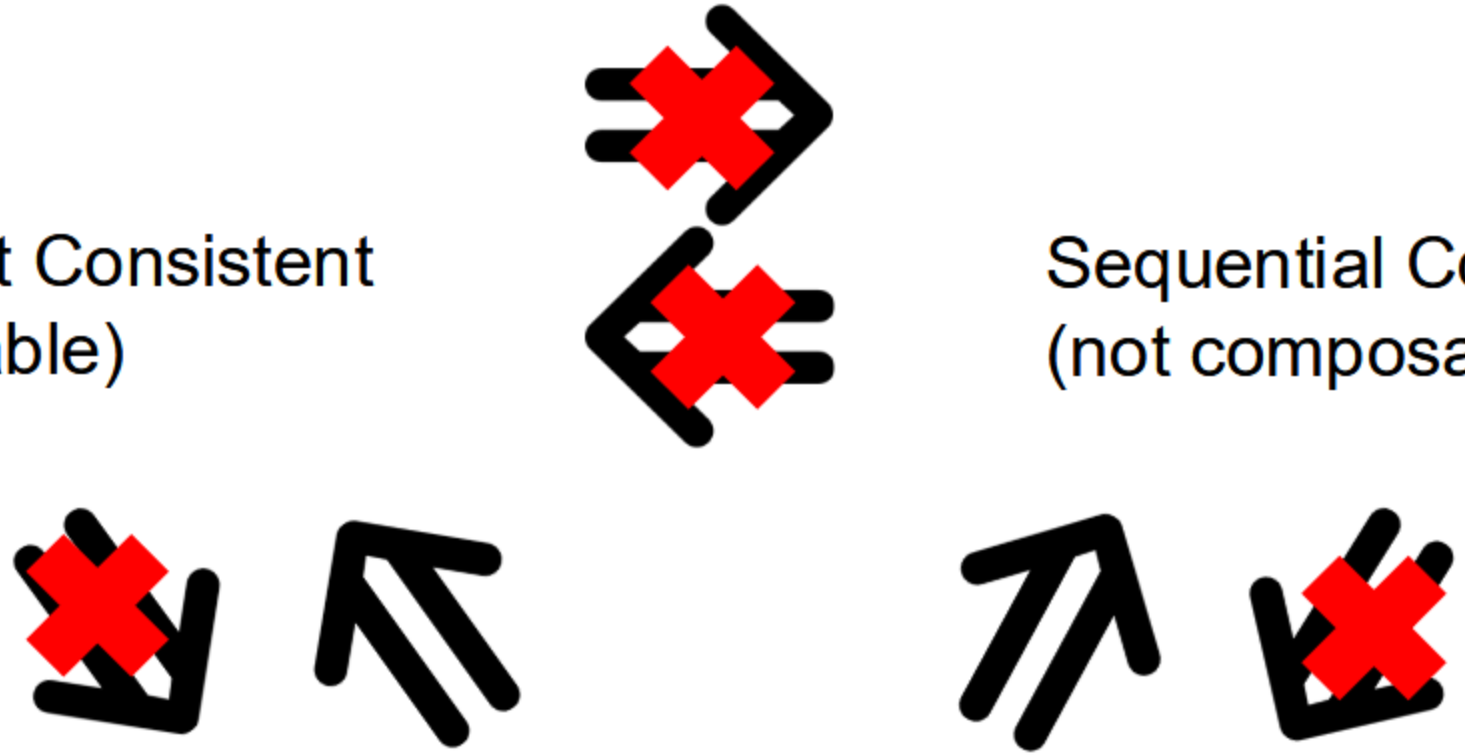
Example (2)

Is this linearizable?

No!



Quiescent Consistent
(composable)



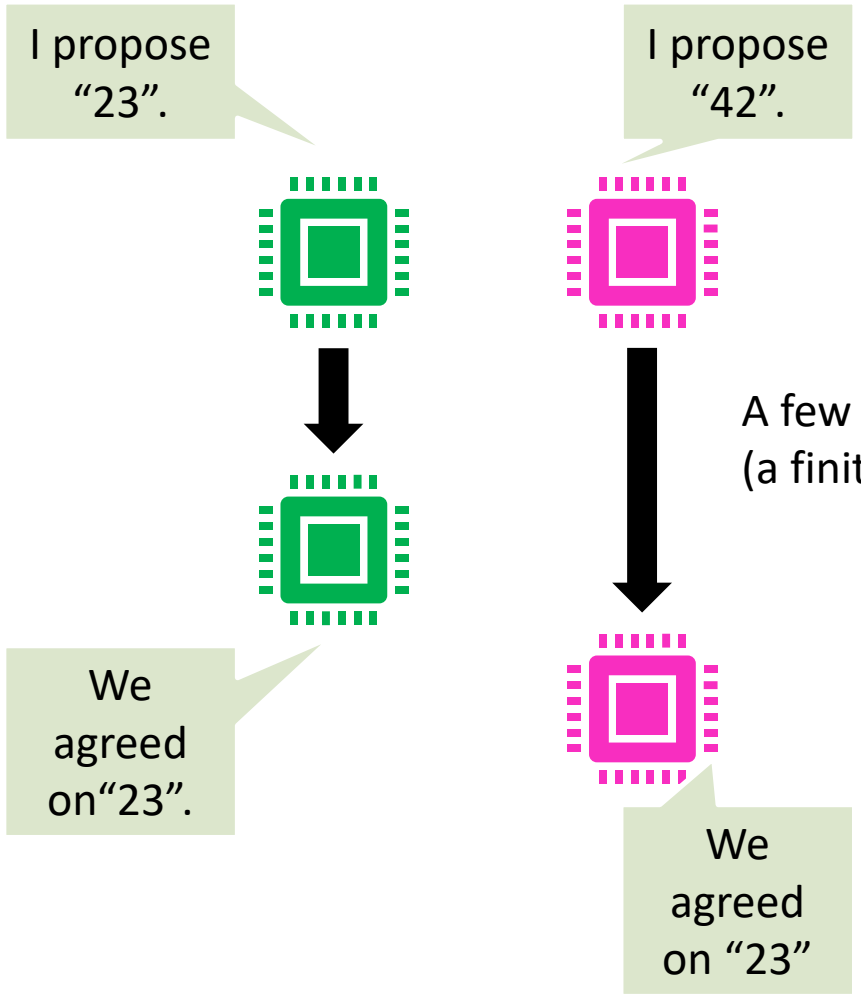
Sequential Consistent
(not composable)

Linearizable
(composable)

Thanks to @Erxuan Li, PProg25

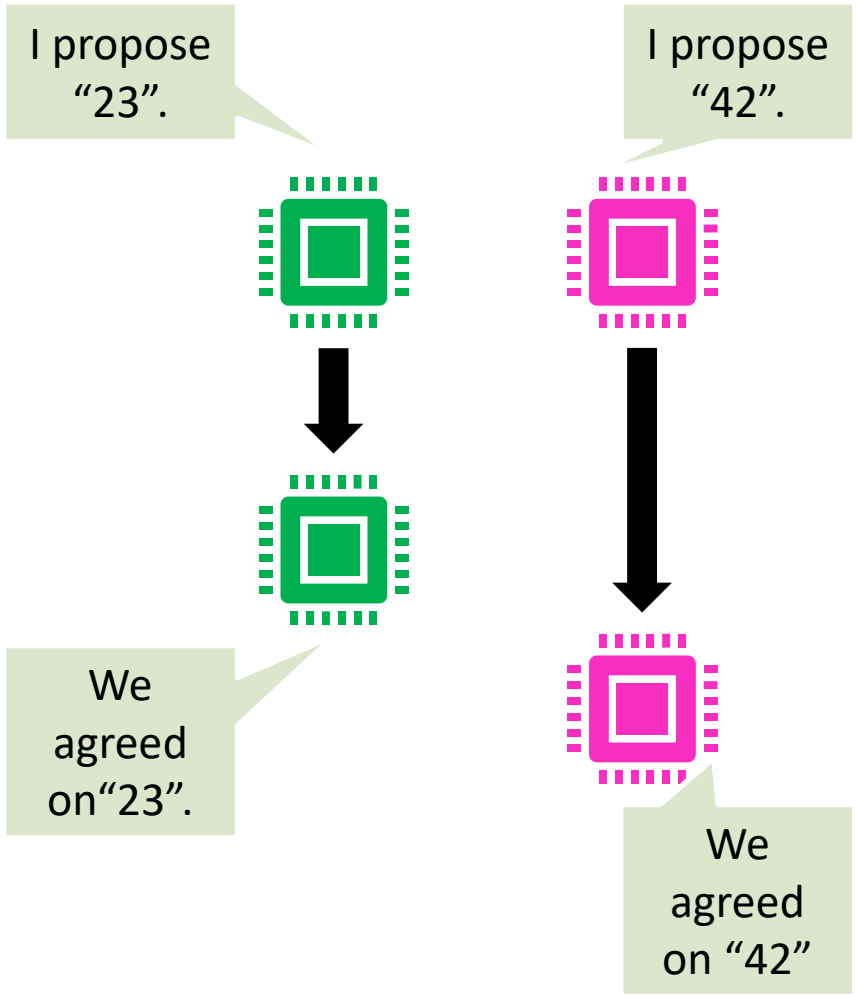
Consensus

Recap: Consensus Protocols



Which other scenarios are allowed?

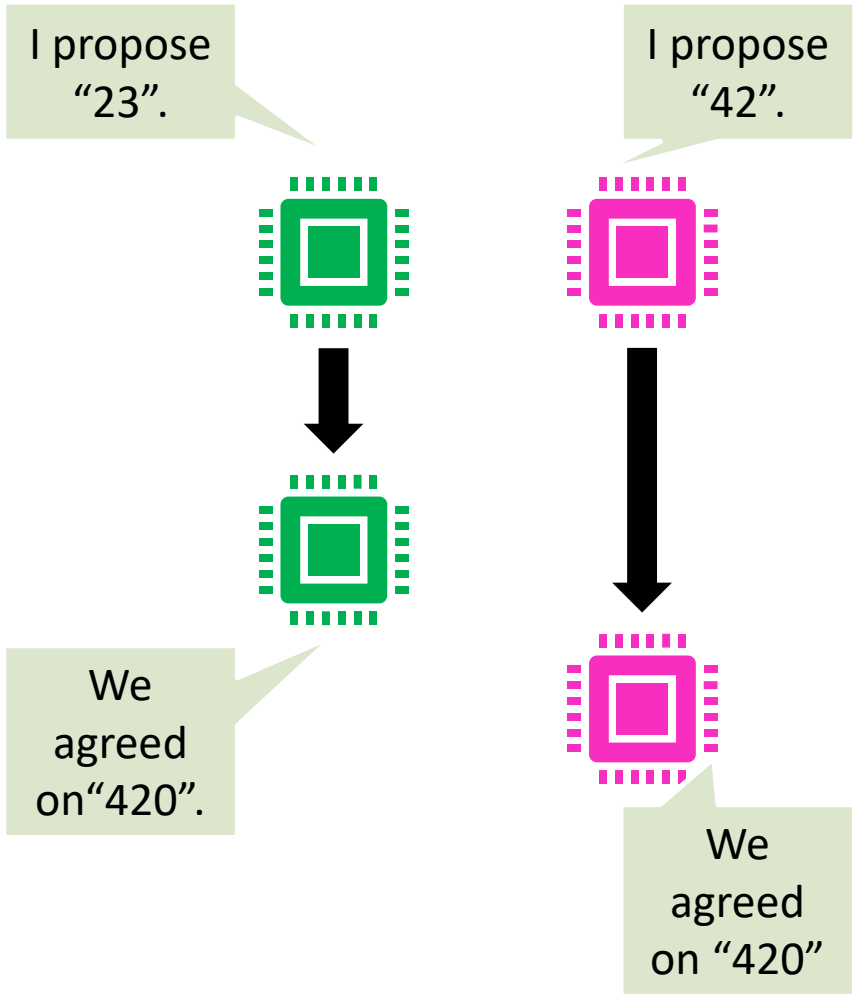
Consistent Result



This is illegal!

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

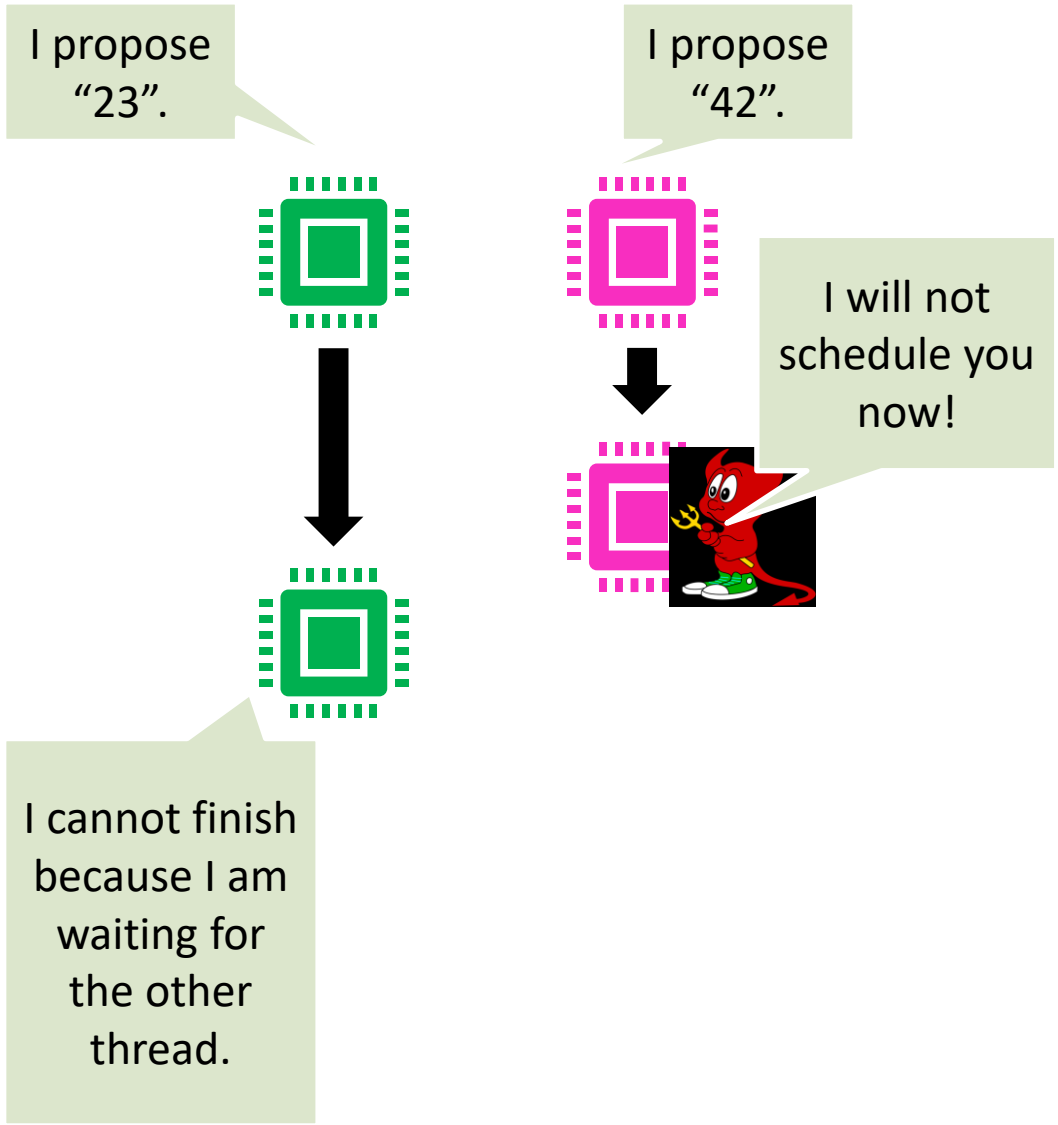
Valid Result



This is illegal!

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

Wait-Free



This is illegal!

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

Valid, Consistent, Wait-free

- This will be asked on the exam 100%

Consensus Number

- The consensus number of C is the largest n for which C solves n-thread consensus
- Atomic Registers have consensus number 1, we'll show this later
- TAS has consensus number 2
- CAS has consensus number ∞ (Can be shown by construction)

Why is Consensus Number important?

- It gives us the consensus hierarchy
- Is backed by mathematical proof
- allows us to say that implementing a lock free FIFO queue is impossible using atomic registers, because queue has CN 2 and AR have CN 1!

The Consensus Hierarchy

1	Read/Write Registers	
2	getAndSet, getAndIncrement, …	FIFO Queue LIFO Stack
. .		
∞	CompareAndSet, …	Multiple Assignment

Implementing one thread consensus using AR

- Atomic Register:

```
int decide(int proposed)
        return proposed;
```

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**

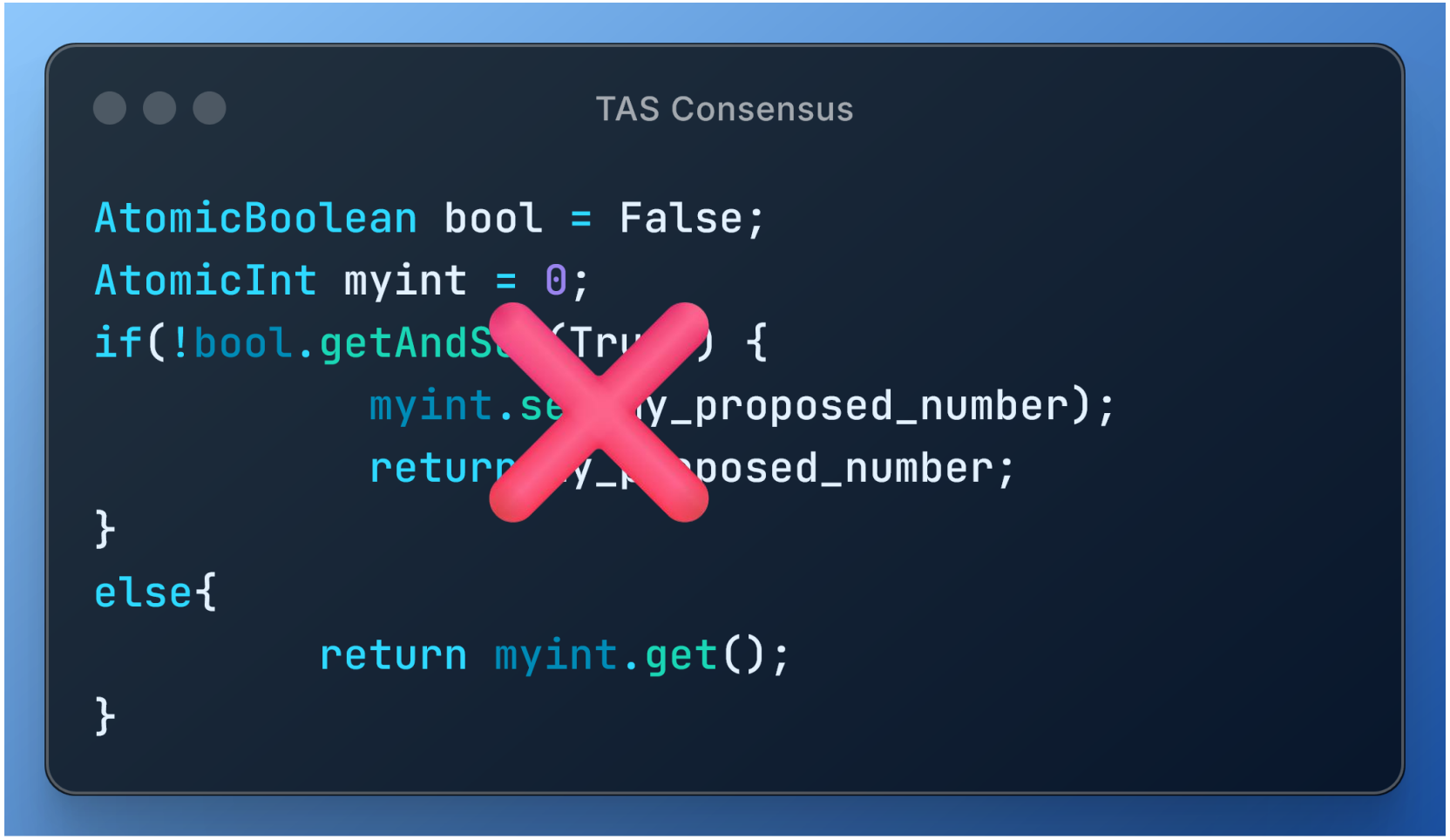
```
read own_value;
read other_value;
if (TAS() == 0) {
    return own_value;
} else
    return other_value;
```

Implementing n thread consensus with TAS

TAS Consensus

```
AtomicBoolean bool = False;
AtomicInt myint = 0;
if(!bool.getAndSet(True)) {
            myint.set(my_proposed_number);
            return my_proposed_number;
}
else{
          return myint.get();
}
```

Implementing n thread consensus with TAS



N thread consensus with CAS

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());
    }
}
```

Simplification: Binary Consensus

- **Instead of proposing an integer, two threads now propose either 0 or 1**
- **Equivalent to "normal" consensus for two threads**
  - How can we prove this?
  - If we can implement one, we directly get the other

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

We can implement binary consensus using normal consensus.

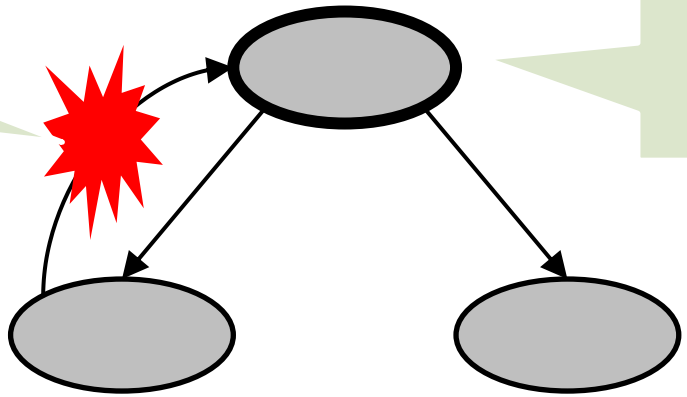
```
int_decide(int d) {
  prop[id] = d; //prop is shared
  other = (id + 1)%2;
  int win = bin_decide(id);
  return prop[win];
}
```

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

State Diagrams of Two-thread Consensus Protocols

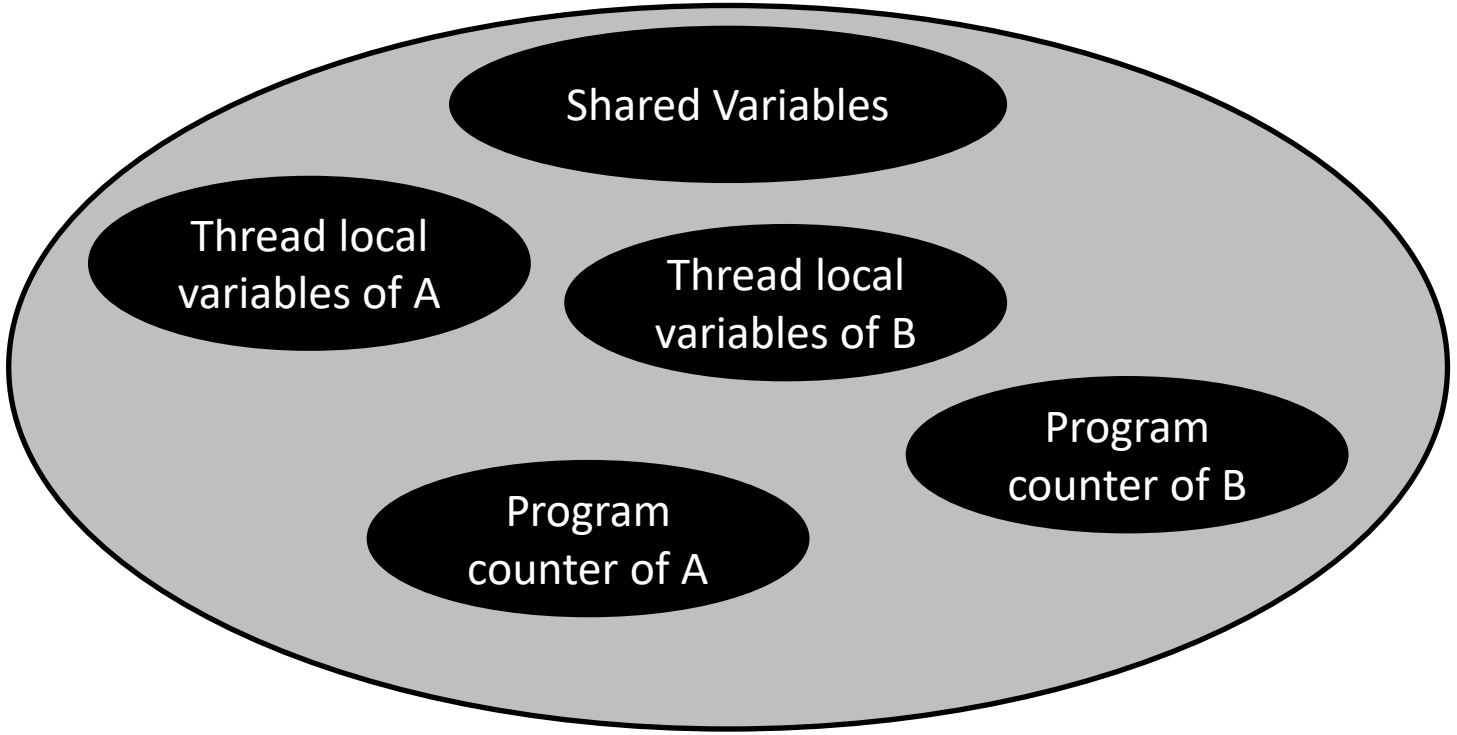
Cycles among states cannot exist in a wait-free algorithm: The state “looks” the same each time we visit, so we are trapped forever in the loop and not wait-free.

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

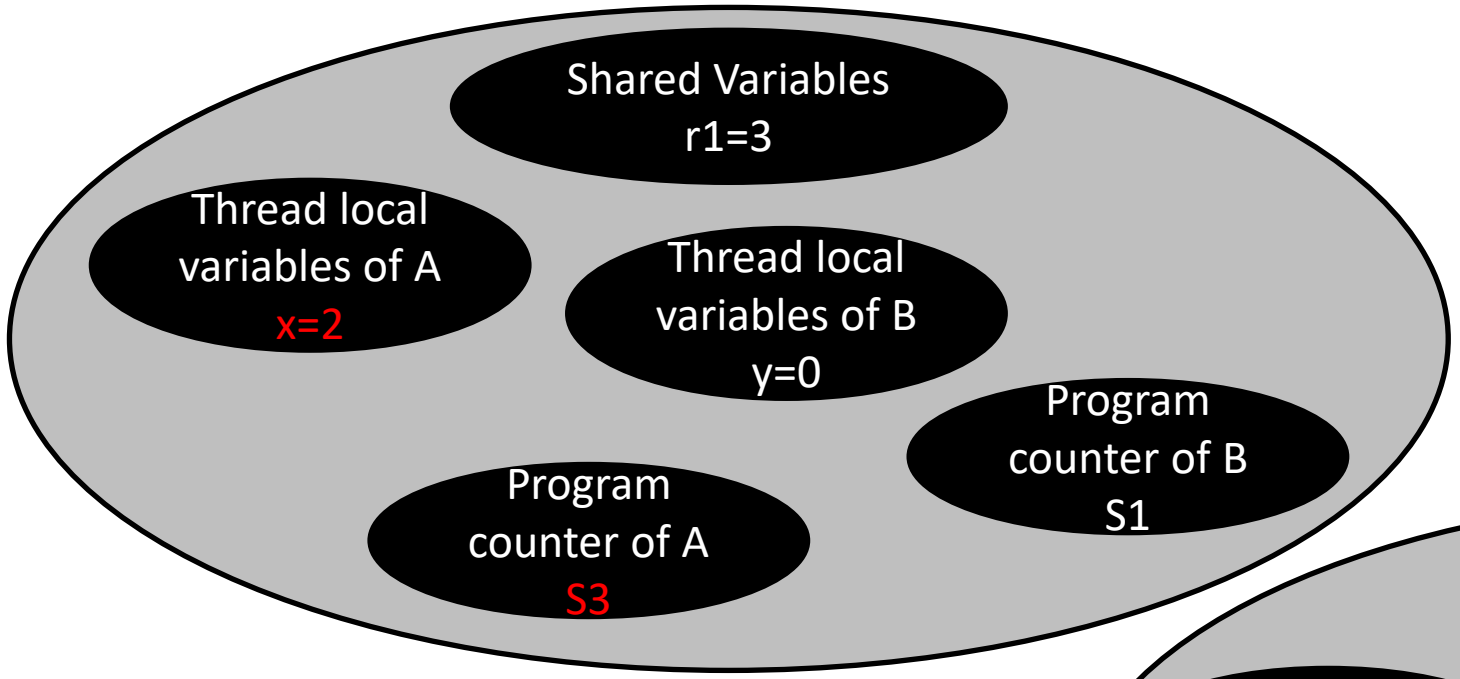


Start state, both threads (A and B) have not yet executed the first instruction of the consensus protocol.

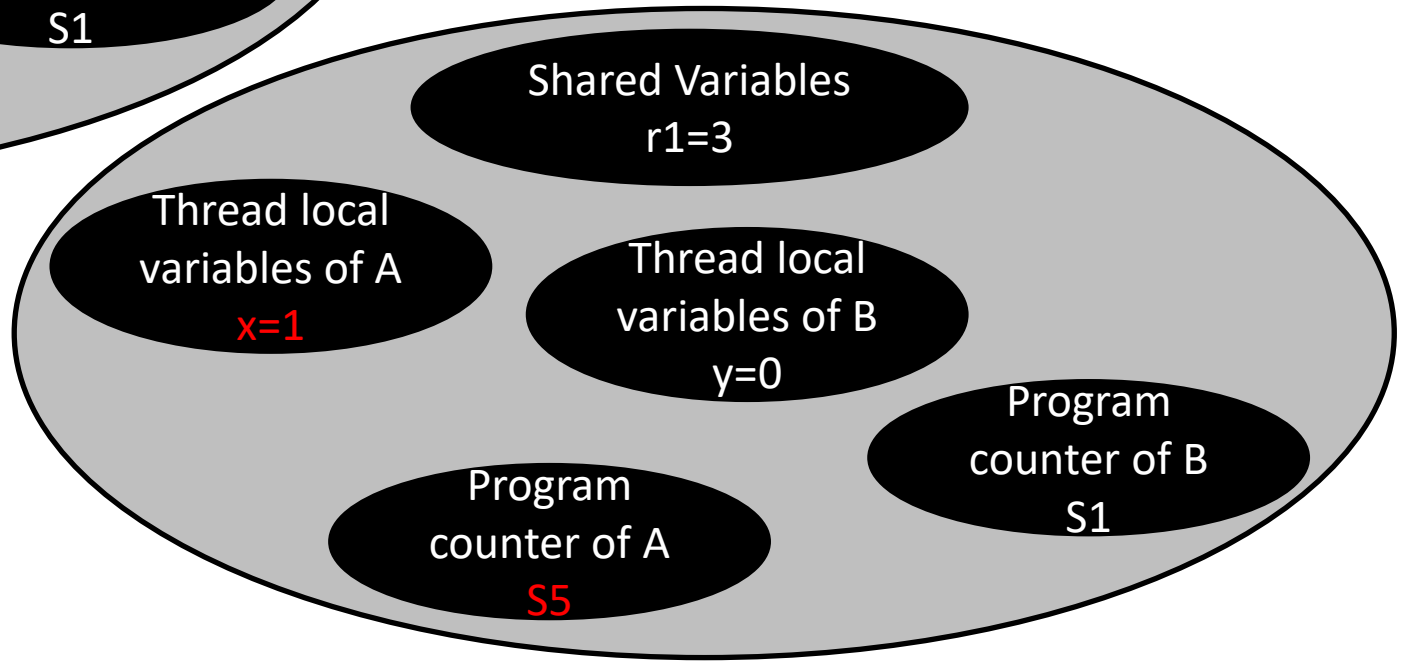
Anatomy of a State (in two-thread consensus)



Anatomy of a State



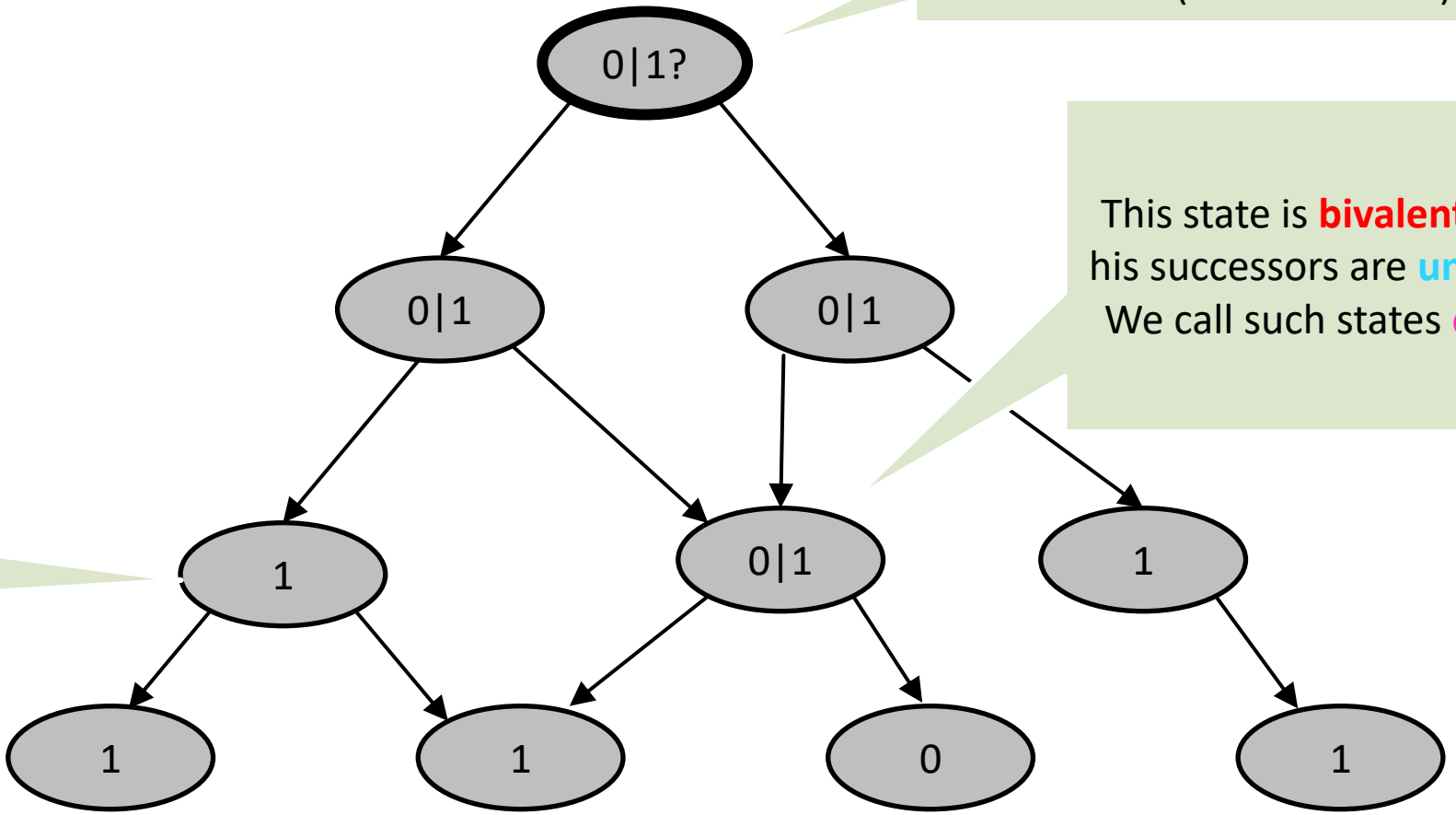
The states are different, since A has different local variables and program counter values.



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**.

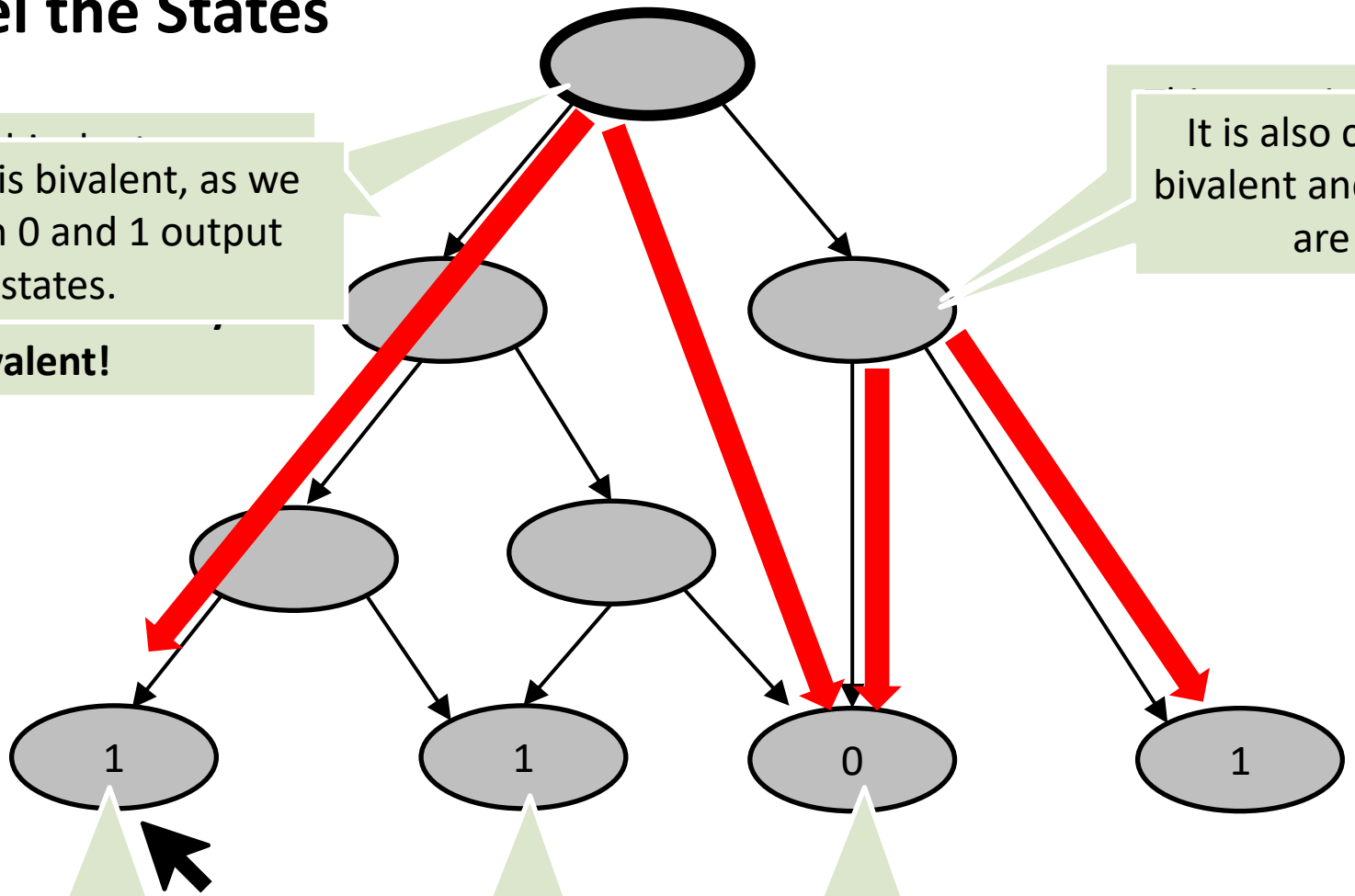
From this state we only reach states with output 1, so it is also **univalent.**

Output states are always **univalent.**

Consensus States

- If we are solving binary consensus, there are 3 different types of states:
- univalent: State, where the output is settled on either 0 or 1
- bivalent: both outputs 0 and 1 are still possible
- critical: bivalent & the following state-transition ends in two univalent states

Quiz: Label the States



This state is bivalent, as we can reach 0 and 1 output states.

bivalent!

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

Output states are alv
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 threads 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.**

Critical State Existence Proof

Lemma: Every consensus protocol has a critical state.

Proof: From (bivalent) start state, let the threads 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.**

If it is wait-free we need to finish in finite time (finish means we return a number) => tree will be of finite length

Proof that Atomic Registers have Consensus Number 1

- Want to show this by checking all possible ways we could try to implement binary consensus using just atomic registers
- Then we show a contradiction for each case, which means that it's impossible to create any higher consensus than for one single thread (which is trivial)

Proof that Atomic Registers have Consensus Number 1

- If binary consensus is not possible for two threads using atomic registers, then it's also not possible for normal consensus (we've seen that they are equivalent)
- Also, not possible for more than two threads

Let’s start

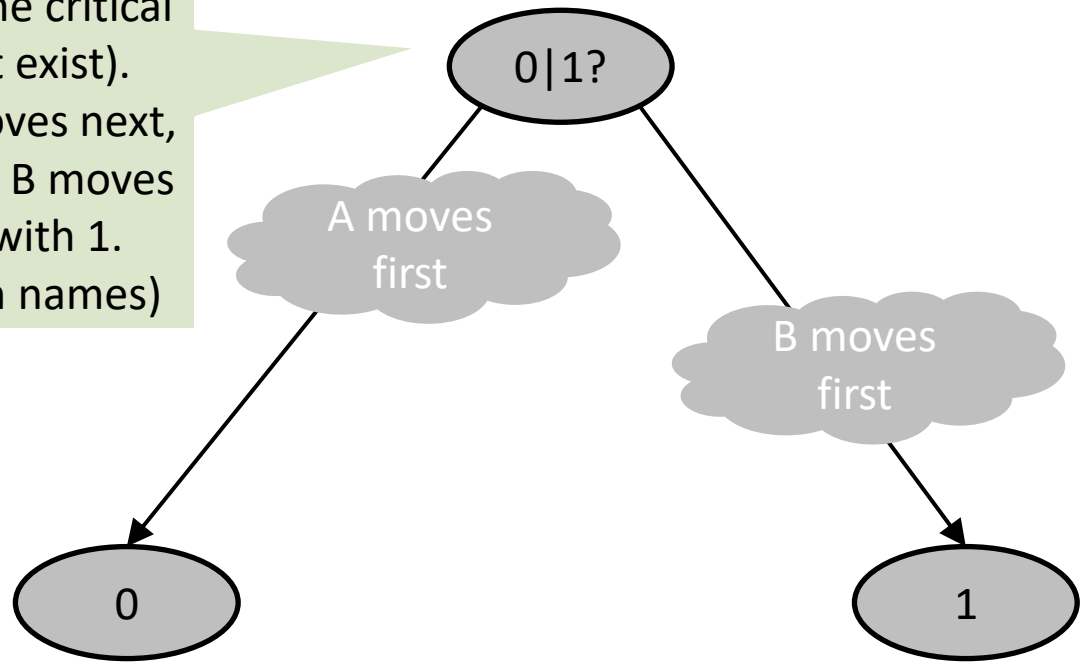
- We know that our wait free consensus protocol must reach a critical state at some point
- So, let’s start there

Impossibility Proof Setup – Critical State

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

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

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)

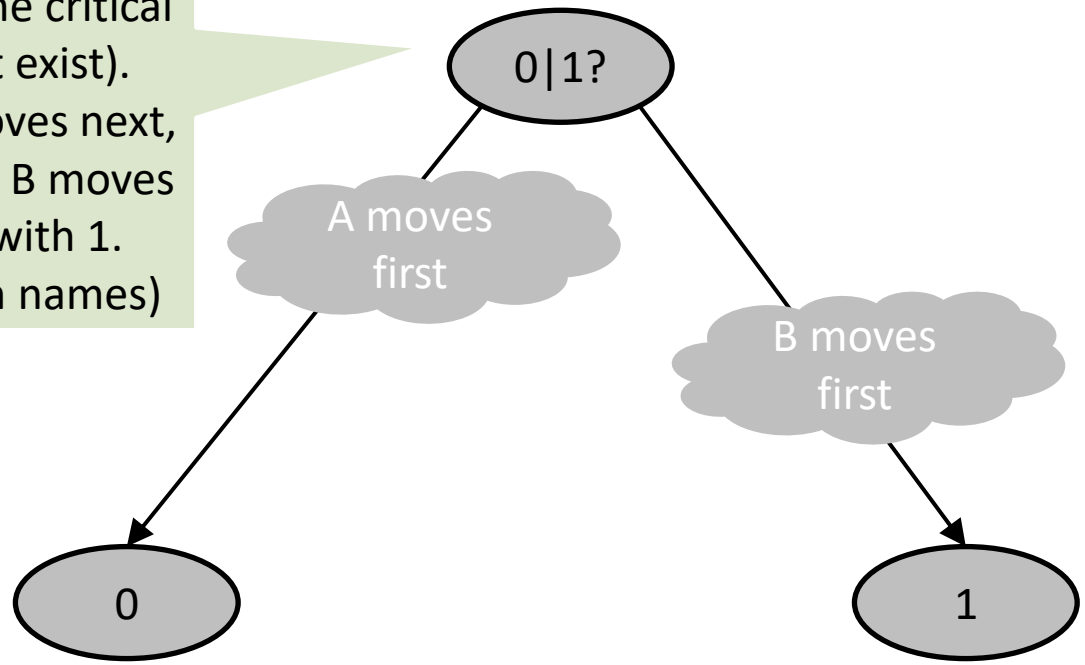


Impossibility Proof Setup – Critical State

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

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

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)

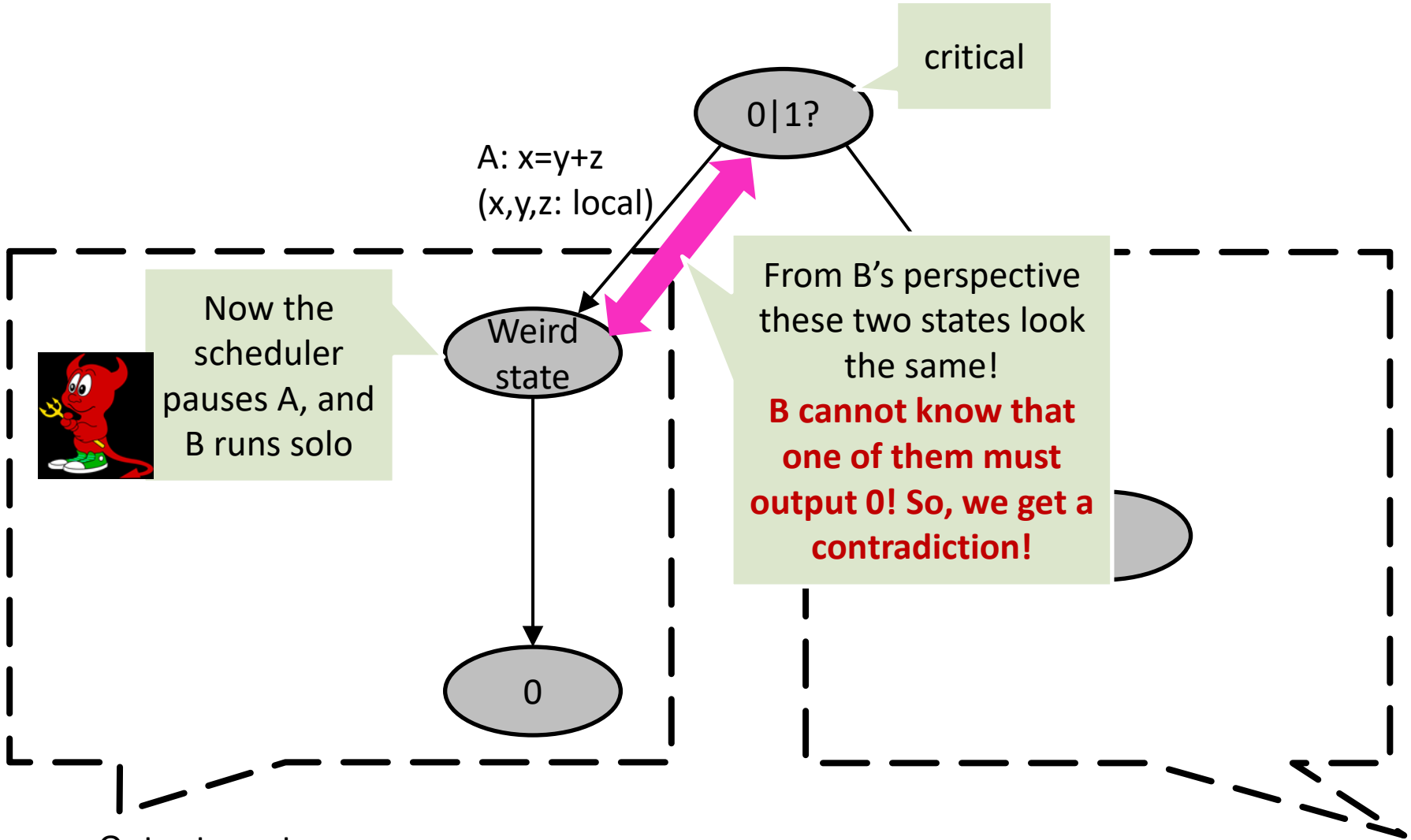


You can think of the transition as which thread performs the CAS or TAS first, and thereby decides the number

We want to know what operations A can do

- A can only write or read a shared variable
- Why? Let’s assume A just reads and writes local variables
- Then the outward state won’t change from the perspective of B!
- Let’s see this played out

Impossibility Proof Setup – Possible actions of a thread



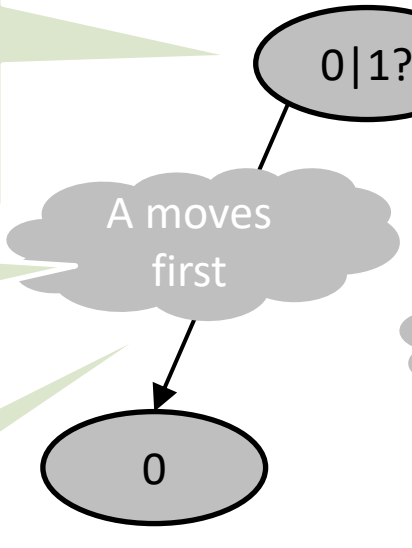
So what actions can a thread perform in his "move"?

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!

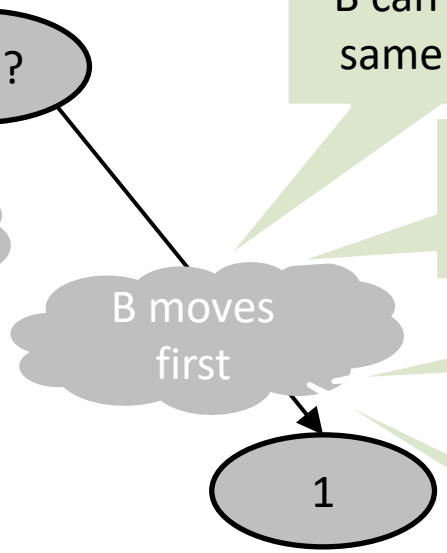
Impossibility Proof Setup – Possible actions of a thread

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

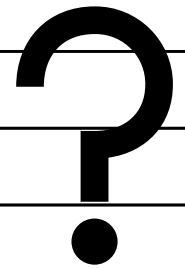
B can write the same variable

B can write a different variable

Many cases…
let's make tables

Many Cases to check

		First Action		n	
		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()				

Manageable… 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/writes r1 in both cases.

Which cases do we need to check?

- Note that First and Second Action don’t really say anything about what happens first and what happens second as we abstracted this away in the previous step

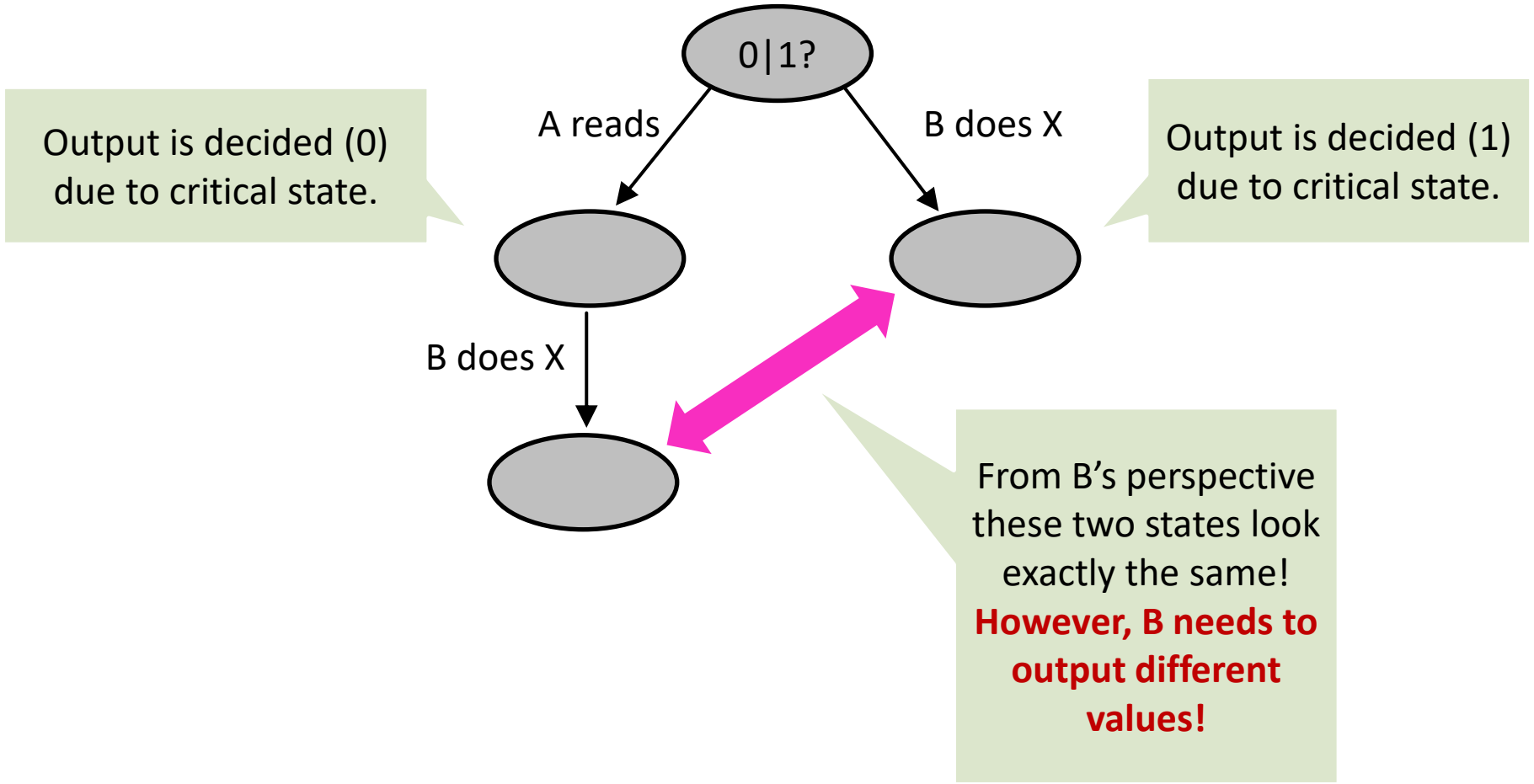
		First Action	
		A: r1.read()	A: r1.write()
Second Action	B: r1.read()		
	B: r2.read()		
	B: r1.write()		
	B: r2.write()		

Which cases do we need to check?

- Let's start with the case where A does r1.read() and B does X (read or write).

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

Impossibility Proof Case I: A reads



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	

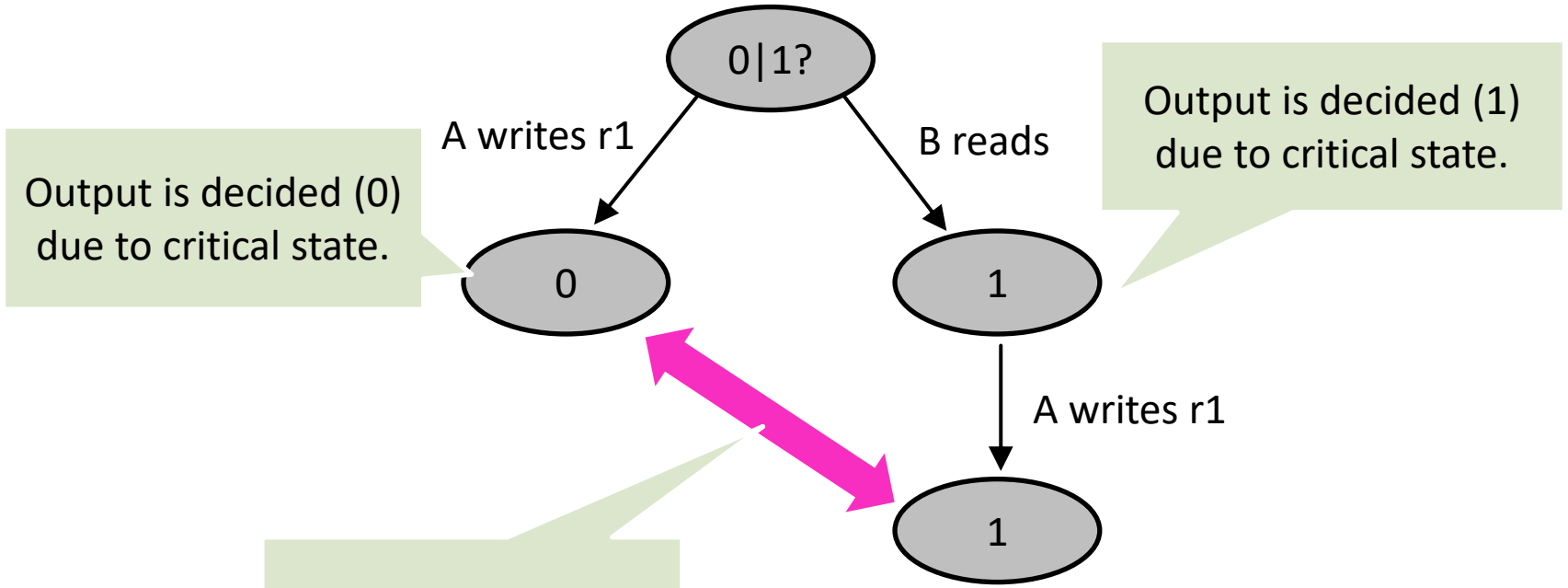
Is binary consensus possible for any of those?

Which cases do we need to check?

- Let's start with the case where A does r1.read() and B does X. ✅
- Next, lets look at when A does r1.write() and B reads either register.

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

Impossibility Proof Case I': B reads



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	?

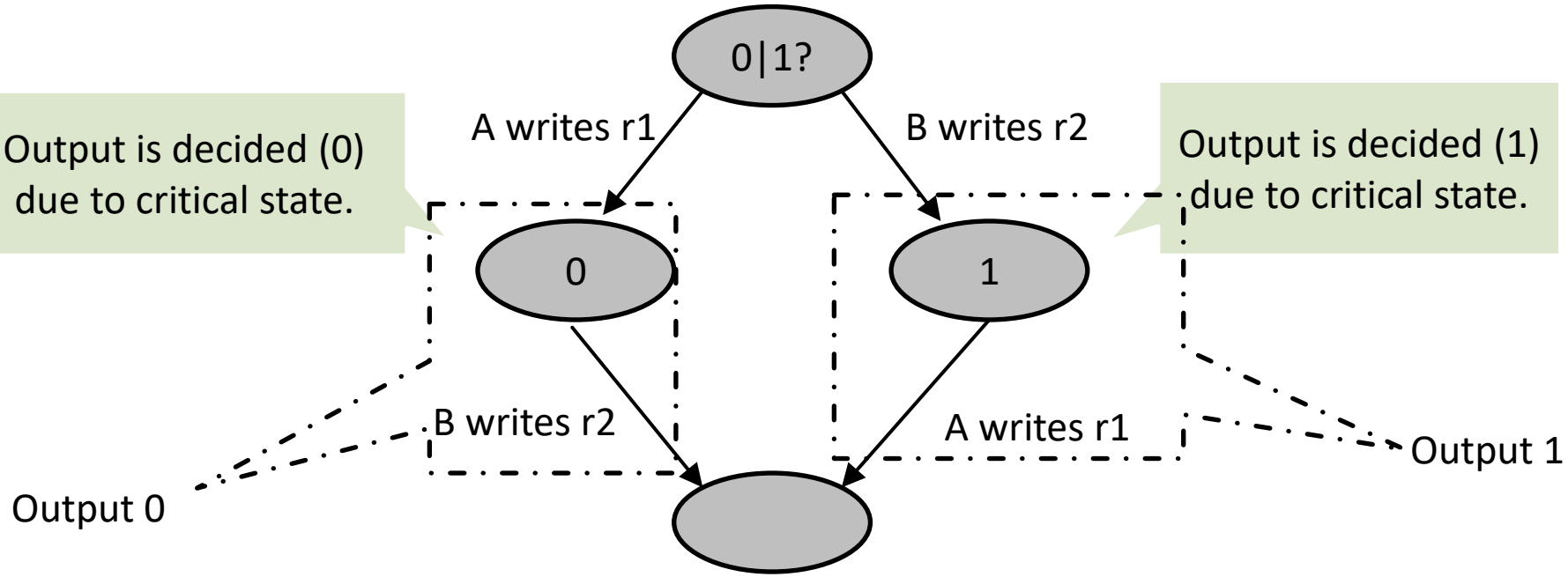
Is binary consensus possible for any of those?

Which cases do we need to check?

- Let's start with the case where A does r1.read() and B does X. ✅
- Next, lets look at when A does r1.write() and B reads either register. ✅
- Now, lets look at the case when A does r1.write() and B does r2.write().

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

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



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

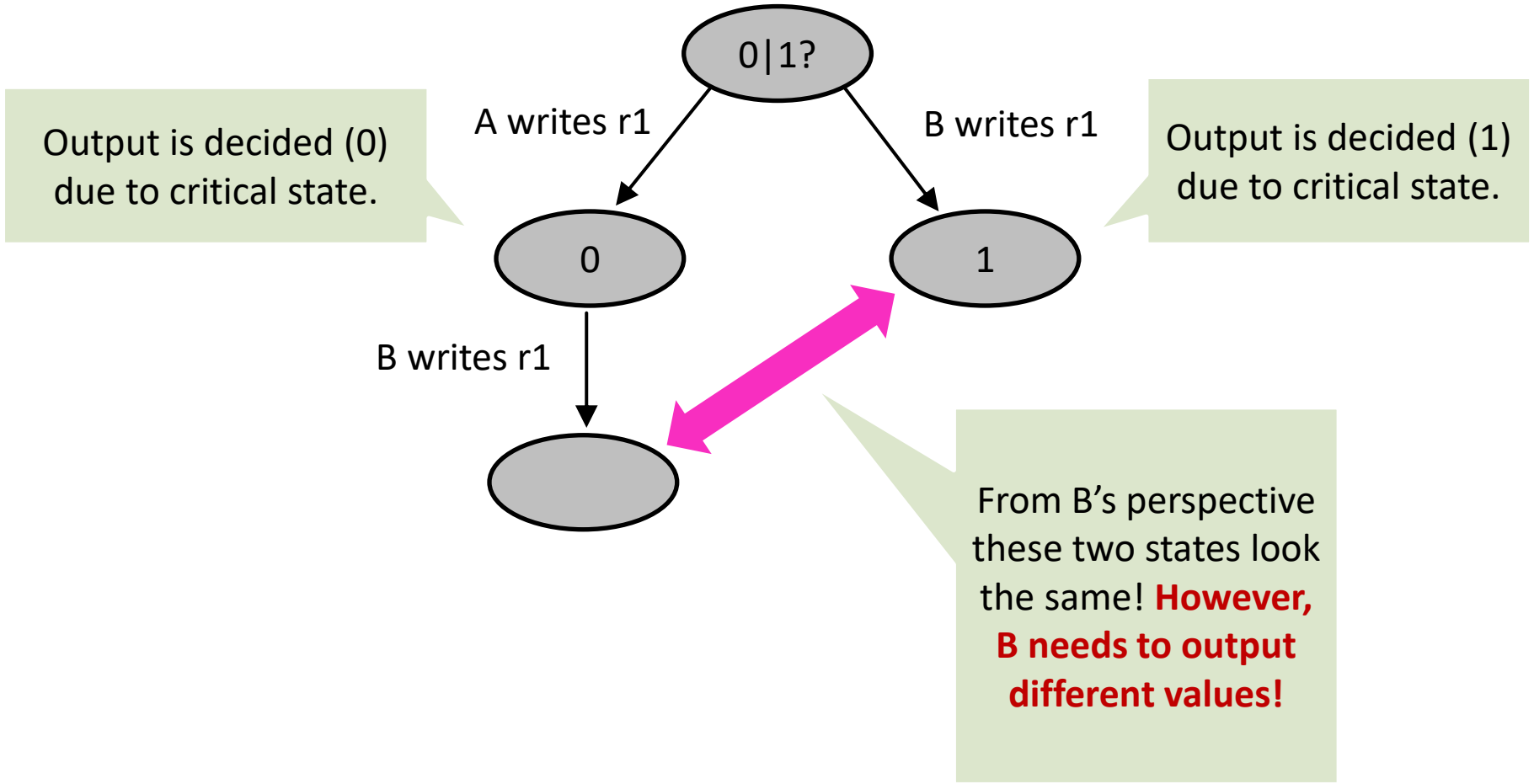
Is binary consensus possible for any of those?

Which cases do we need to check?

- Let's start with the case where A does r1.read() and B does X. ✅
- Next, lets look at when A does r1.write() and B reads either register. ✅
- Now, lets look at the case when A does r1.write() and B does r2.write(). ✅
- Finally, let's check A does r1.write() and B does r1.write() too.

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

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



Which cases do we need to check?

- Let's start with the case where A does r1.read() and B does X. ✅
- Next, lets look at when A does r1.write() and B reads either register. ✅
- Now, lets look at the case when A does r1.write() and B does r2.write(). ✅
- Finally, let's check A does r1.write() and B does r1.write() too. ✅

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

What did we show?

- We proved that there is no possible way to build a consensus protocol using just atomic registers by enumerating all possible ways we could try to implement such a protocol

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

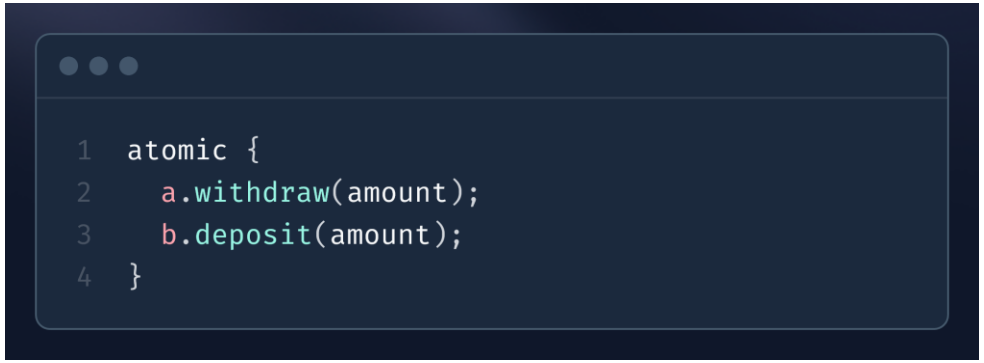
University of Warwick, Coventry, England

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

Questions?

Transactional Memory

- goal of transactional memory is to remove the burden of synchronization away from the programmer and place it in the system (be that hardware or software)
- Ideally, programmer only has to say



Transactional Memory

- Has nice properties:
  - simpler, less error-prone code
  - higher-level semantics (what vs. how)
  - composable (unlike locks)
  - analogy to garbage collection
  - optimistic by design (does not require mutual exclusion)
- Downsides:
  - semantics are not clear (e.g., nesting)
  - getting a good performance can be challenging
  - how we should deal with I/O in transactions (i.e., how would one rollback these changes?) is not clear.

Transactional Memory - ACID

- Transactions follow ACID principle
  - Atomic – changes by transaction are made visible atomically, threads either see the state before a transaction or after a transaction finished successfully. They don’t observe intermediate states.
  - Consistent – objects will never be in an inconsistent state, i.e., between method calls, transactions always move from one consistent state to another consistent state
  - Isolated – while a transaction is running, effects from other transactions are not observed. Transaction takes “snapshot” and works on it
  - Durable – (mostly used for databases that we don’t lose data when a power loss occurs)

Transactional Memory

- How do we build it?

Transactional Memory

- How do we build it?
- create a “snapshot” of the current state and make sure that transaction only affects a local copy of this state, which can then be either committed or aborted
- If a transaction which has yet to commit has read a value (at the start of its operation) that was changed by a transaction that has committed, a conflict (non repeatable read) arises, and we need to abort the transaction and retry
- i.e., check if the state of the system changed in the meantime, if yes, we need to retry

Transactional Memory

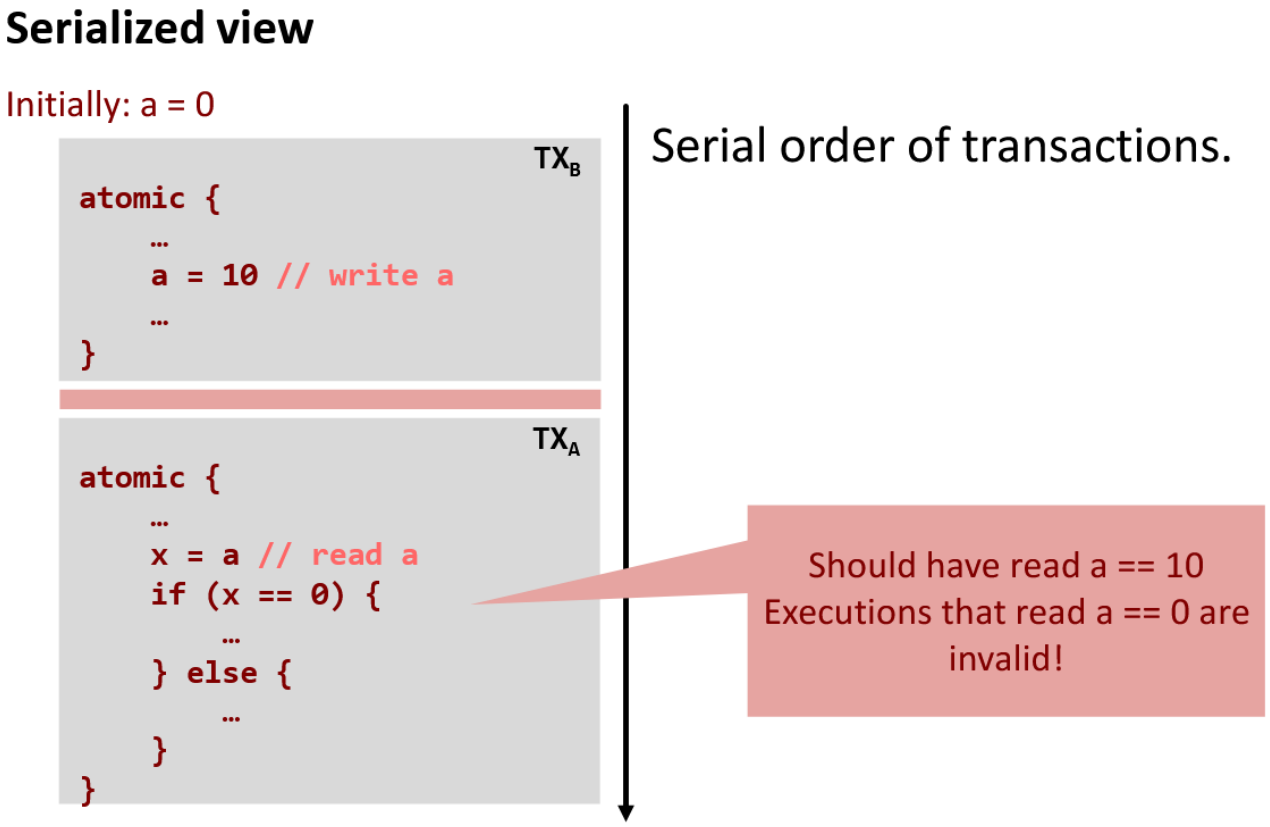
- Consider the following example, where the initial state is a=0
- assume that transaction B commits the changes it has made before A does
- Now, in a serialized view, the execution with a==0 is invalid!

```
1  // Transaction A
2  atomic {
3    ...
4    x = a; // read a
5    if (x == 0){
6      // do something
7    } else {
8      // do something else
9    }
10 }
```

```
1  // Transaction B
2  atomic {
3    ...
4    a = 10; // write a
5    ...
6  }
```

Transactional Memory

- Consider the following example, where the initial state is a=0
- assume that transaction B commits the changes it has made before A does
- Now, in a serialized view, the execution with a==0 is invalid!



Transactional Memory

- can implement TM either in hard- or software
- HTM is fast, but has bounded resources that often cannot handle big transactions
- STM allows greater flexibility, but achieving good performance might be very challenging

Transactional Memory - Nesting

- We need to make some design choices
- Transactions should be composable
- What happens if we have transactions within transactions? This is called nesting
- Two possible approaches:

Transactional Memory - Nesting

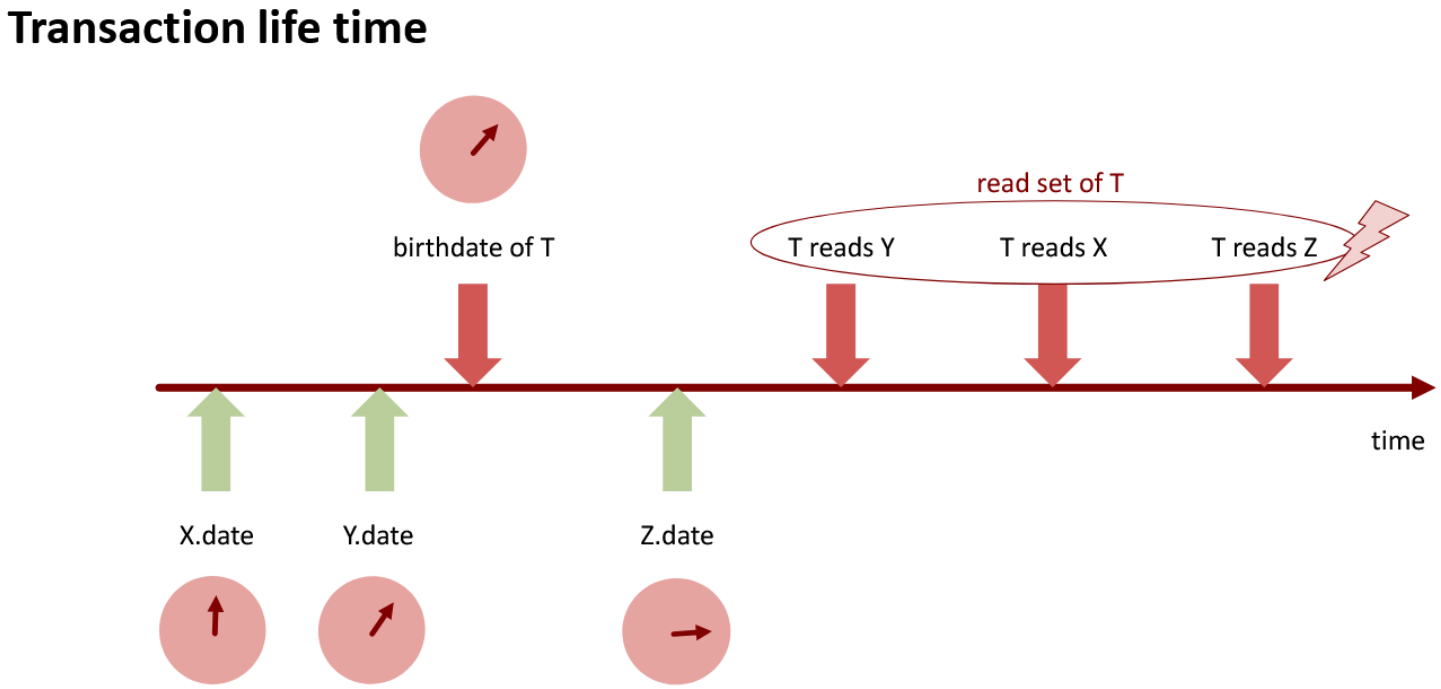
- What happens if we have transactions within transactions? This is called nesting
- Two possible approaches:
  - Flat/Flattened Nesting: Inner and outer transactions are treated as one. If an inner transaction aborts, all abort. Changes from the inner are visible only if the outer commits.
  - Closed Nesting: Inner aborts don't affect the outer. If an inner commits, changes are visible to the outer but not to others until the outer commits.

Transactional Memory - ScalaSTM

- Uses a clock-based system, i.e., every transaction has a time when it was started and when it was committed
- Each transaction has a local read-set and a local write-set, holding all locally read and written objects
- Allows us to check if our data became “outdated”, e.g., some other thread committed, and we need to retry

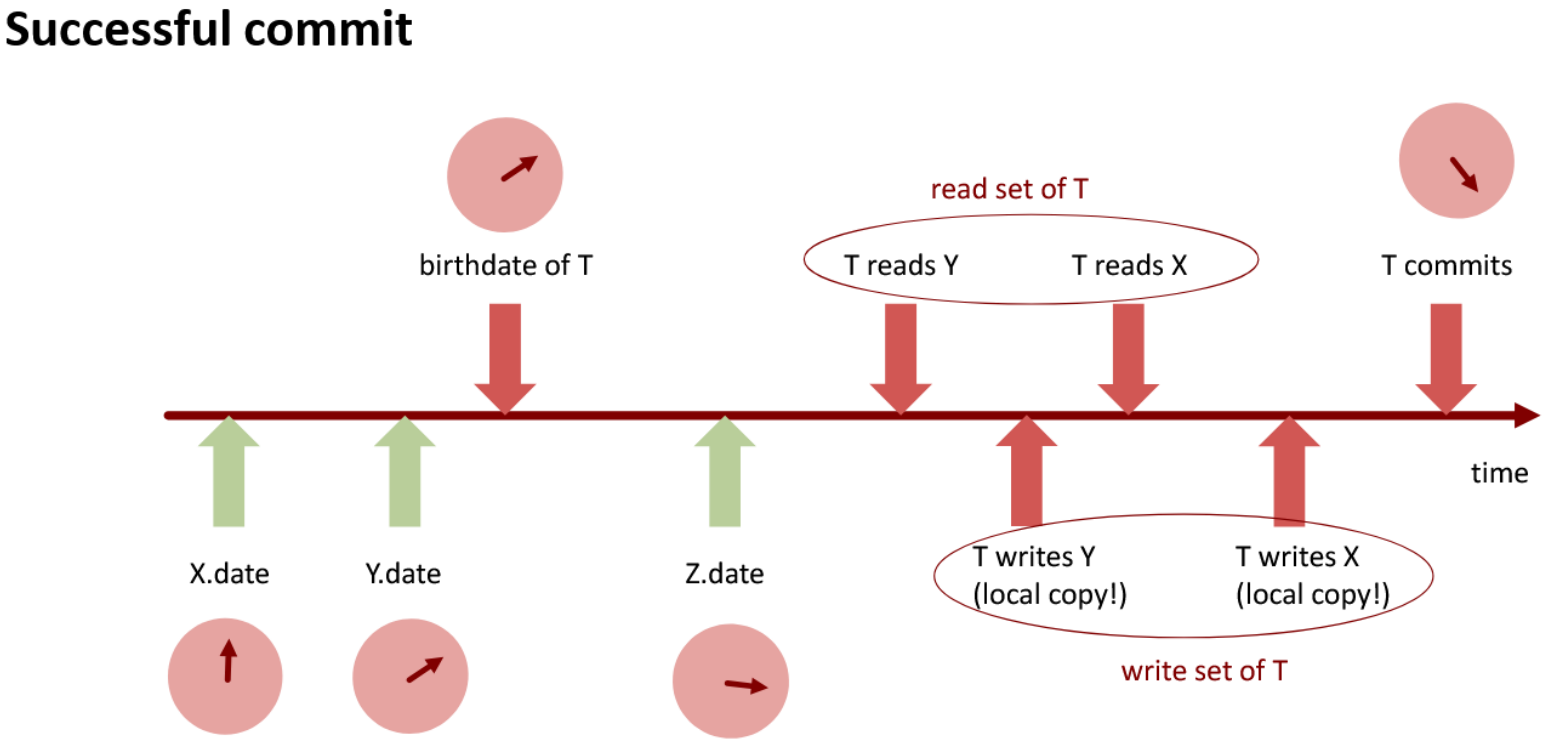
Transactional Memory - ScalaSTM

- In this example, the T snapshot became outdated, so we need to abort and retry



Transactional Memory - ScalaSTM

- In this example, the T snapshot was the most recent version of all variables used, so we can commit



MPI

- Many of the problems of parallel/concurrent programming come from sharing state. What if we simply avoid this?
- Message Passing has isolated mutable state, each thread/task has its private, mutable state, and separate tasks only cooperate via explicit message passing

MPI

- messages can be divided into synchronous and asynchronous
- Synchronous messages mean that the sender of the message blocks/waits until the message is received
- Asynchronous messages do not block, but are placed into a buffer (“postbox”) for the receiver to get at some point

- Example Go Code which uses channels

```go
func main() {
  msgs := make(chan string)
  done := make(chan bool)
  go hello(msgs, done);
  msgs <- "Hello"
  msgs <- "bye"
  ok := <-done
  fmt.Println("Done:", ok);
}
func hello(msgs chan string, done chan bool) {
  for {
    msg := <-msgs
    fmt.Println("Got:", msg)
    if msg == "bye" {
      break
    }
  }
  done <- true;
}
```

So, what is MPI?

- MPI is a standard application/programming interface (API), meaning it is a portable, flexible library not bound to a particular language.
- It is the most used interface for distributed parallel computing, which is nearly the entirety of high performance computing

So, what is MPI?

- Works SPMD: Single Program Multiple Data (Multiple Instances)
- We compile only one program, which gets executed by multiple different instances
- Every MPI program can be written using just six core functions:
  - MPI_Init – Initializes the MPI environment (this must be the first function called).
  - MPI_Comm_size – Determines the number of processes in a communicator.
  - MPI_Comm_rank – Returns the rank (ID) of the calling process within the communicator.
  - MPI_Send – Sends a message to another process.
  - MPI_Recv – Receives a message from another process.
  - MPI_Finalize – Cleans up the MPI environment (this must be the last function called).

MPI

- Einfaches Beispiel, zwei Threads schicken sich eine Nachricht

```java
import mpi.*;

public class HelloWorld {
    public static void main(String[] args) throws MPIException {
        // Initialize the MPI execution environment
        MPI.Init(args);

        // Get the rank (ID) of the current process
        int rank = MPI.COMM_WORLD.Rank();
        // Get the total number of processes
        int size = MPI.COMM_WORLD.Size();

        // Each process prints its rank and total size
        System.out.println("Hello from process " + rank + " of " + size);

        // Example of point-to-point communication:
        if (size >= 2) {
            if (rank == 0) {
                // Prepare a message to send
                String message = "Greetings from process 0";
                // Send the message to process 1
                MPI.COMM_WORLD.Send(new Object[]{message}, 0, 1, MPI.OBJECT, 1, 99);
                System.out.println("Process 0 sent message: '" + message + "'");

            } else if (rank == 1) {
                // Prepare a buffer to receive the message
                Object[] buffer = new Object[1];
                // Receive the message from process 0
                MPI.COMM_WORLD.Recv(buffer, 0, 1, MPI.OBJECT, 0, 99);
                String received = (String) buffer[0];
                System.out.println("Process 1 received message: '" + received + "'");
            }
        } else {
            if (rank == 0) {
                System.out.println("Need at least 2 processes to demonstrate send/
receive");
            }
        }

        // Finalize the MPI environment
        MPI.Finalize();
```

MPI Collectives

- Up until now, we saw only point-to-point communication
- MPI also supports communications among groups of processors
- Reduce: to reduce a result from different processes to one (called root)
- Broadcast: Broadcasts a message from the root process to all other processes in the group
- Allreduce: Like reduce, but hands result to all processes involved
- Gather: Each process sends the contents of its send buffer to the root process

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- Theory
- **Exam questions**
- Intro Assignment 14
- Kahoot
- A&W DP Tricks

What’s important for the exam?

- Properties of consensus protocol: valid, consistent, wait-free
- Knowing what bivalent, univalent and critical states are
- Being able to reason about why there can’t be cycles in a wait free program
- identifying whether given code is a correct consensus protocol
- making a statement about the consensus number of an object

7. (a) Unten sehen Sie fehlerhafte Versuche wait-free consensus Protokolle für zwei Threads zu implementieren. Erklären Sie für jede Implementierung weshalb diese fehlerhaft ist, indem Sie aufzeigen welche der Eigenschaften von wait-free consensus die Implementierung nicht erfüllt (sollte eine Implementierung mehrere Eigenschaften nicht erfüllen so wählen Sie eine davon aus).

Below you see incorrect attempts to implement a wait-free consensus protocol for two threads. For each of them briefly explain why it is incorrect by explaining which property of wait-free consensus the implementation does not fulfill (if it violates multiple properties pick one).

(6)

```
1 int decide(int val) {
2   return val;
3 }
```

7. (a) Unten sehen Sie fehlerhafte Versuche wait-free consensus Protokolle für zwei Threads zu implementieren. Erklären Sie für jede Implementierung weshalb diese fehlerhaft ist, indem Sie aufzeigen welche der Eigenschaften von wait-free consensus die Implementierung nicht erfüllt (sollte eine Implementierung mehrere Eigenschaften nicht erfüllen so wählen Sie eine davon aus).

Below you see incorrect attempts to implement a wait-free consensus protocol for two threads. For each of them briefly explain why it is incorrect by explaining which property of wait-free consensus the implementation does not fulfill (if it violates multiple properties pick one). (6)

```
1 int decide(int val) {
2   return val;
3 }
```

- Consistency is violated
- For two threads calling the method with two different values, the method will return different values

```
1  int decided = 0; //shared variable
2  int proposed[2]; //shared variable
3
4  int decide(int val, int thread_id /* either 0 or 1*/) {
5    decided += 1;
6    proposed[thread_id] = val;
7    while (decided < 2) {}
8    decided = 0
9    return proposed[1];
10 }
```

```
1  int decided = 0; //shared variable
2  int proposed[2]; //shared variable
3
4  int decide(int val, int thread_id /* either 0 or 1*/) {
5    decided += 1;
6    proposed[thread_id] = val;
7    while (decided < 2) {}
8    decided = 0
9    return proposed[1];
10 }
```

- Not wait-free!
- suspension of one thread can cause the other thread to spin indefinitely (the first thread must wait for the second). Thus, a thread calling the method would not be guaranteed to finish in a finite number of steps

```
1 int decide(int val) {
2   return 0;
3 }
```

```
1 int decide(int val) {
2   return 0;
3 }
```

- Not valid!
- No thread could propose 0 then it’s wrong to return 0

(b) Geben Sie jeweils ein Beispiel für ein Objekt mit der Konsensuszahl 1, 2 und unendlich. Begründen Sie warum das angegebene Objekt mindestens diese Konsensuszahl haben muss.

Give an example of an object (one each) which has consensus number 1, 2, and infinity. Argue why the given object has at least that consensus number. (7)

(b) Geben Sie jeweils ein Beispiel für ein Objekt mit der Konsensuszahl 1, 2 und unendlich. Begründen Sie warum das angegebene Objekt mindestens diese Konsensuszahl haben muss.

Give an example of an object (one each) which has consensus number 1, 2, and infinity. Argue why the given object has at least that consensus number. (7)

- Atomic Register – 1
- TAS – 2
- CAS – Infinity

(b) Geben Sie jeweils ein Beispiel für ein Objekt mit der Konsensuszahl 1, 2 und unendlich. Begründen Sie warum das angegebene Objekt mindestens diese Konsensuszahl haben muss.

Give an example of an object (one each) which has consensus number 1, 2, and infinity. Argue why the given object has at least that consensus number. (7)

- Atomic Register:

```
int decide(int proposed)
        return proposed;
```

(b) Geben Sie jeweils ein Beispiel für ein Objekt mit der Konsensuszahl 1, 2 und unendlich. Begründen Sie warum das angegebene Objekt mindestens diese Konsensuszahl haben muss.

Give an example of an object (one each) which has consensus number 1, 2, and infinity. Argue why the given object has at least that consensus number. (7)

- TAS:

```
boolean decided = false;
int[] value = new int[2];
int decide(int proposed, int id)
   value[id] = proposed;
   if (TAS(decided)){
      return value[1-id];
         }
   return value[id];
```

(b) Geben Sie jeweils ein Beispiel für ein Objekt mit der Konsensuszahl 1, 2 und unendlich. Begründen Sie warum das angegebene Objekt mindestens diese Konsensuszahl haben muss.

Give an example of an object (one each) which has consensus number 1, 2, and infinity. Argue why the given object has at least that consensus number. (7)

- CAS:

```
int decided = NaN;
int decide(int proposed)
    CAS(decided, NaN, proposed);
    return decided;
```

(a) Nennen und erklären Sie die drei definierenden Eigenschaften eines wait-free Konsensus Protokolls.

Name and explain the three defining properties of wait-free consensus. (3)

(a) Nennen und erklären Sie die drei definierenden Eigenschaften eines wait-free Konsensus Protokolls.

Name and explain the three defining properties of wait-free consensus. (3)

(a) Nennen und erklären Sie die drei definierenden Eigenschaften eines wait-free Konsensus Protokolls.

Name and explain the three defining properties of wait-free consensus. (3)

wait-free: consensus returns in finite time for each thread.

consistent: all threads decide the same value (i.e., reach consensus)

valid: the decision value is some thread's input

(b) Kann wait-free binary consensus für zwei Threads mithilfe von Locks implementiert werden? Begründen Sie Ihre Antwort.

Can binary wait-free consensus for two threads be implemented with locks? Explain your answer. (2)

(b) Kann wait-free binary consensus für zwei Threads mithilfe von Locks implementiert werden? Begründen Sie Ihre Antwort.

Can binary wait-free consensus for two threads be implemented with locks? Explain your answer. (2)

(b) Kann wait-free binary consensus für zwei Threads mithilfe von Locks implementiert werden? Begründen Sie Ihre Antwort.

Can binary wait-free consensus for two threads be implemented with locks? Explain your answer. (2)

- **No.** Wait-free implies lock-free which means that no locks can be used

Exam allgemein

Exam allgemein

- Man kann PProg sehr gut lernen und jeder von euch kann eine super Note erreichen
- Löst die alten exams auf community solutions
- Falls ihr Verständnisprobleme habt, lest im Buch (link auf meiner website) nach, da ist es oft nochmal besser erklärt
- Ihr müsst nicht alle Slides nochmal anschauen, wenn ihr in der Vorlesung wart, ich denke Exams sind viel wichtiger
- Theory Assignments sind gut zum Üben, die Coding Probleme sind auch wichtig! Oft gibt es so Code Skelett Aufgaben (meine Prüfung)

Code Skelett Aufgaben

```
        this.exit = exit;
        this.ticket_number = entry.get_ticket();
        int zone_index = this.entry.get_index();
        this.n_cars_on_roundabout = n_cars_on_roundabout;
        // this.zones enthält alle Zonen, welche das Auto
        // passieren will, geordnet.
        /* this.zones contains all the zones the car wants
         * to cross, in order. */
        while (zone_index != this.exit.get_index()) {
            this.zones.add(zones[zone_index]);
            zone_index = (zone_index + 1) % zones.length;
        }
    }


    @Override
    public void run() {
        while (true) {
            synchronized (...................) {                    // 1 pt
                if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
                /* Give priority to cars that arrived at the
                 * same entry earlier. */
                ..............................................    // 1 pt
                ..............................................
                } else {
                    break;
                }
        }

        // Bevor das Auto in den Kreisverkehr einfährt,
        // darf höchstens ein Auto im Kreisverkehr sein.
        /* Before entering the roundabout,
         * ensure that at most one is already in the roundabout. */
        while (true) {
        ................................................       // 4 pt
        ................................................
        ................................................
        ................................................
        ................................................
        ................................................
        ................................................
        }

        Zone current_zone = this.zones.get(0);
```

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 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

Types of exercises that might come in the exam

- Proving that an object x has consensus number ≥y(or ∞). Then you must provide an algorithm solving y-thread consensus using any number of instances of x (or for the ∞ case, provide an algorithm solving n-thread consensus for arbitrary n)
- Proving that an object x has consensus number at most z. This would involve proving that it is impossible to implement (z+1)-thread consensus with x
- E.g., if we know x is implemented using atomic registers then it can’t solve consensus for more than one thread

Types of exercises that might come in the exam

- I don’t expect much about MPI and Transactional Memory as it was only covered shortly (no guarantee!)
- Maybe some Mixer questions about it at the end (check the old exams)

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- Theory
- Exam questions
- **Intro Assignment 14** (yes, there is another one 😑)
- Kahoot
- A&W DP Tricks

Assignment 14

- Is about Consensus
- “Below we show incorrect implementations of a consensus protocol in pseudocode. Which property does each snippet violate when used with two threads?”

- Show that lock free FIFO queue has consensus number 2
- Show that lock free FIFO queue with peek() has consensus number infinity

- Show how to implement two thread consensus using binary consensus

- I would recommend solving it, these are exam relevant questions

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- Theory
- Exam questions
- Intro Assignment 14
- **Kahoot**
- A&W DP Tricks

Kahoot!

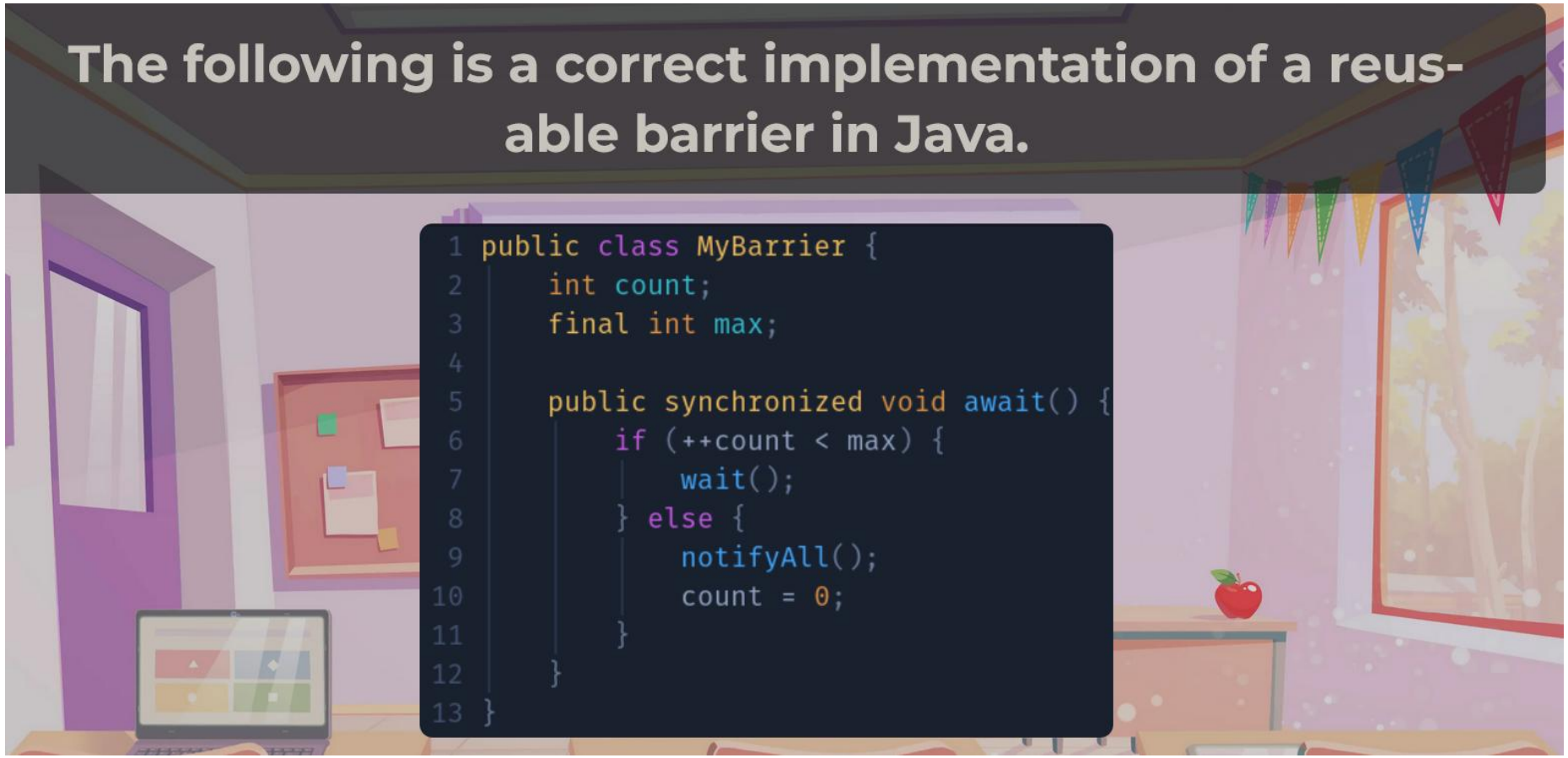
- Ich werde alle Kahoots noch auf meiner Website hochladen, schaut da gerne nach, auch für die Lernphase

Consensus Protocol for two threads using lock free queue

```
1  public class QueueConsensus<T> extends ConsensusProtocol<T>
2    private static final int WIN = 0; // first thread
3    private static final int LOSE = 1; // second thread
4    Queue queue;
5    // initialize queue with two items
6    public QueueConsensus() {
7      queue = new Queue();
8      queue.enq(WIN);
9      queue.enq(LOSE);
10   }
11   // figure out which thread was first
12   public T decide(T Value) {
13     propose(value);
14     int status = queue.deq();
15     int i = ThreadID.get();
16     if (status == WIN)
17       return proposed[i];
18     else
19       return proposed[1-i];
20   }
21 }
```

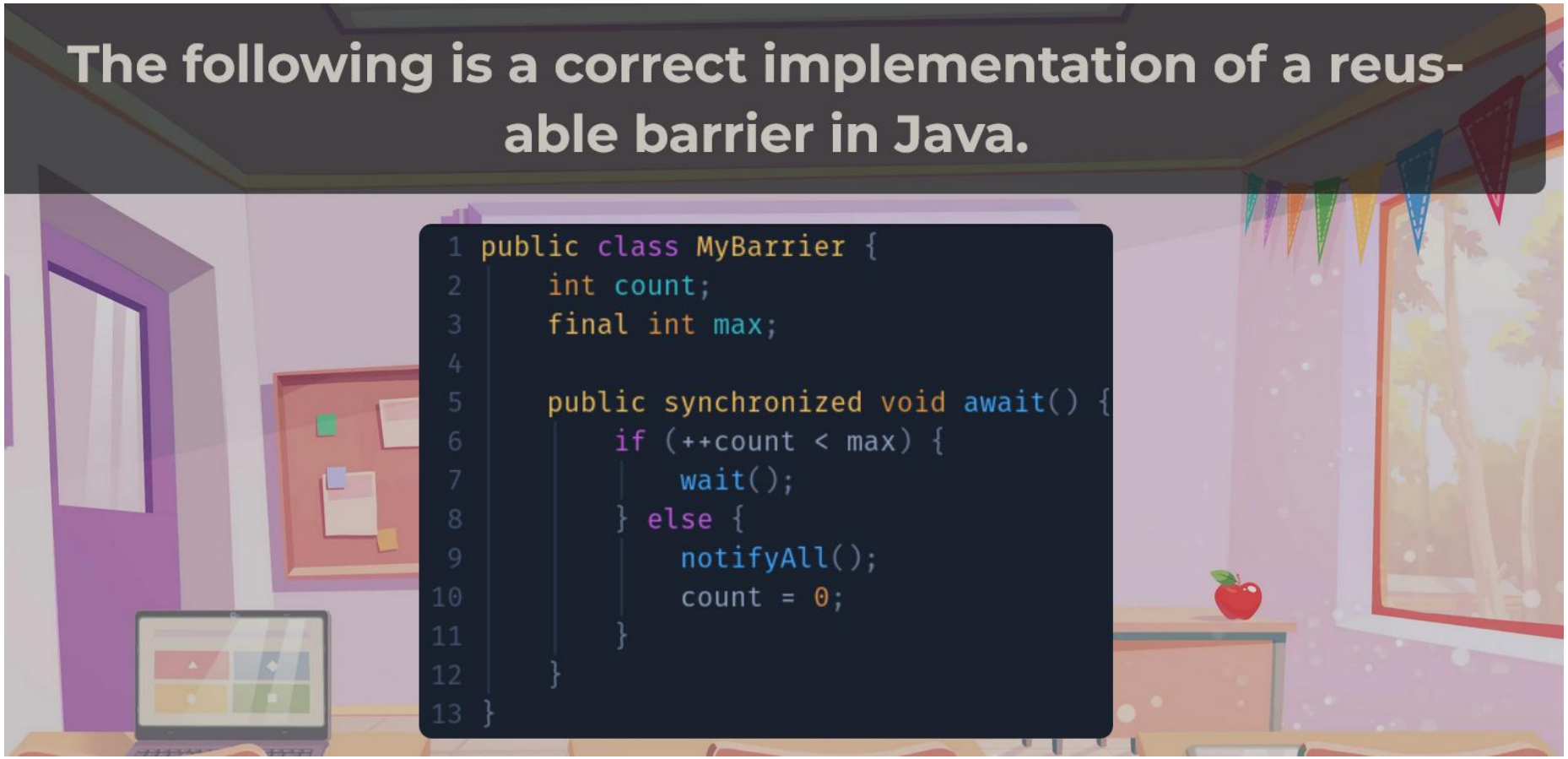
Figure 5.7 2-thread consensus using a FIFO queue.

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

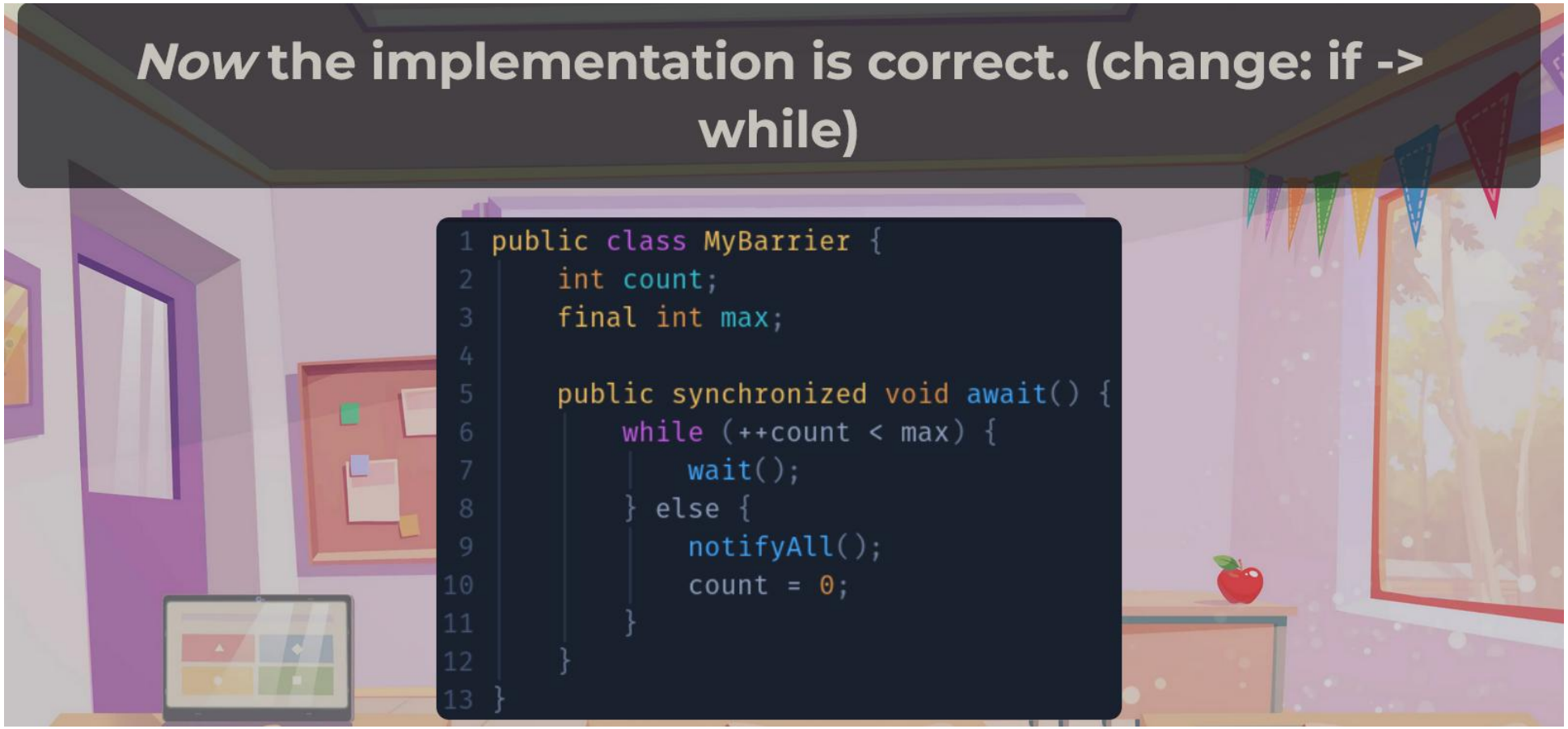


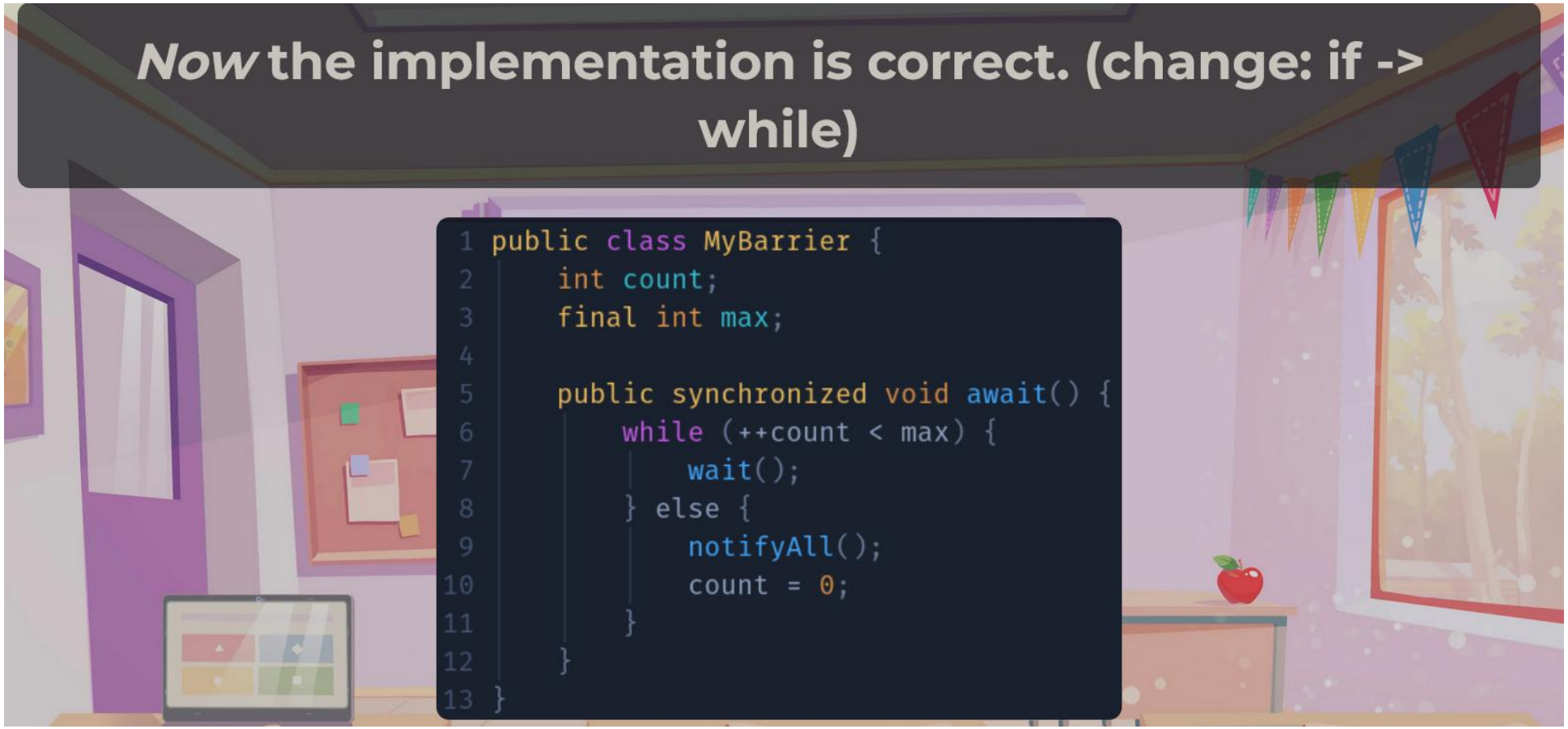
```java
public class MyBarrier {
    int count;
    final int max;

    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.

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.

Plan für heute

- Organisation
- Nachbesprechung Assignment 13
- Theory
- Exam questions
- Kahoot
- **Extra: A&W DP Tricks**

Danke

- Es war mir eine grosse Freude euch unterrichten zu dürfen
- Nächstes Semester bin ich im Austausch, aber ich hoffe man sieht sich im FS wieder (nicht in PProg), sprecht mich gerne an, wenn ihr mich irgendwo sieht
- Ich drücke euch die Daumen, für die Lernphase, alles wird gut
- Nehmt euch auch ein bisschen Auszeit, die Lernphase ist doppelt so lang wie im Herbst!

Schöne Ferien!

Airport Security Task

- Falls ihr den rekursiven Ansatz für diese Aufgaben noch nicht gesehen habt, würde ich ihn euch gerne nochmal zeigen
- Hat mir damals extrem geholfen und kann eure Note direkt verbessern