Parallele Programmierung FS25

Exercise Session 12

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Assignment 11
- Theory
- Intro Assignment 12
- 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

- Wo sind wir jetzt?

Reader Writer Lock

Lock free programming

wait free, lock free, starvation free, deadlock free

Lock free linked list

To come: Linearizability, Consensus

Plan für heute

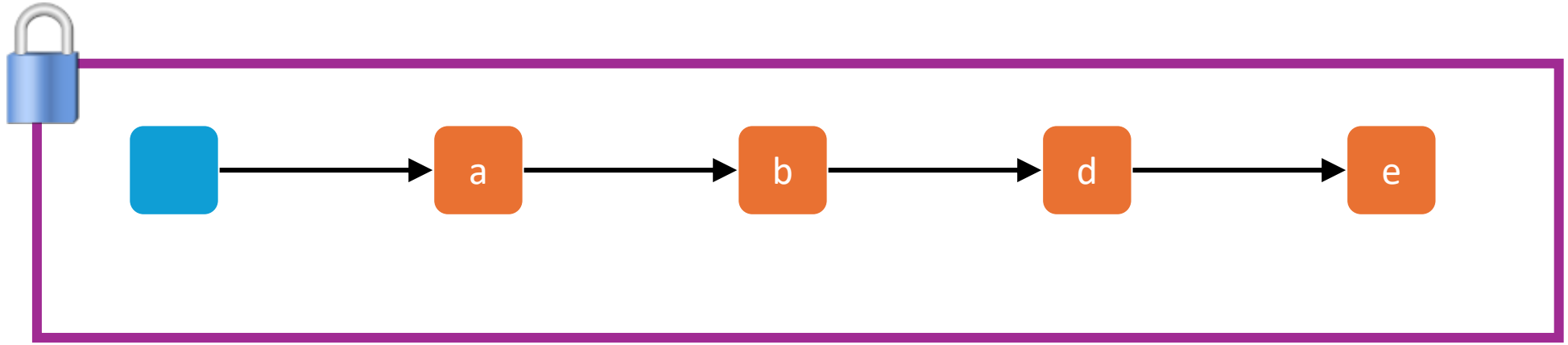
- Organisation
- **Nachbesprechung Assignment 11**
- Theory
- Intro Assignment 12
- Exam questions
- Kahoot

Assignment 11

- Implement SortedList with different lock strategies
- Exercise about effective use of locks
  - Coarse grained vs. fine grained locks
  - Tricks to avoid locking altogether for certain operations
- Measure the performance impact of your implementation choice

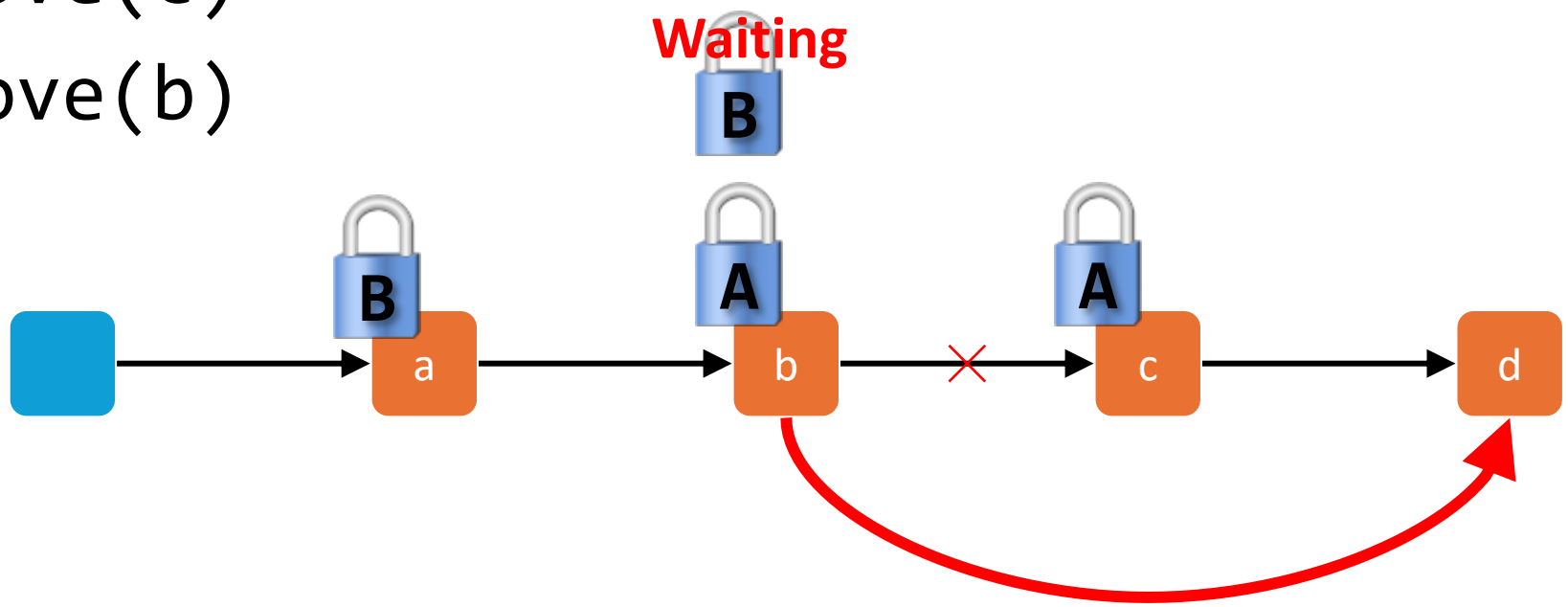
Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```



Fine Grained locking

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



When removing, lock the **successor** defensively.

Optimistic Locking

Remove case

A: add(c)

A: find insertion point

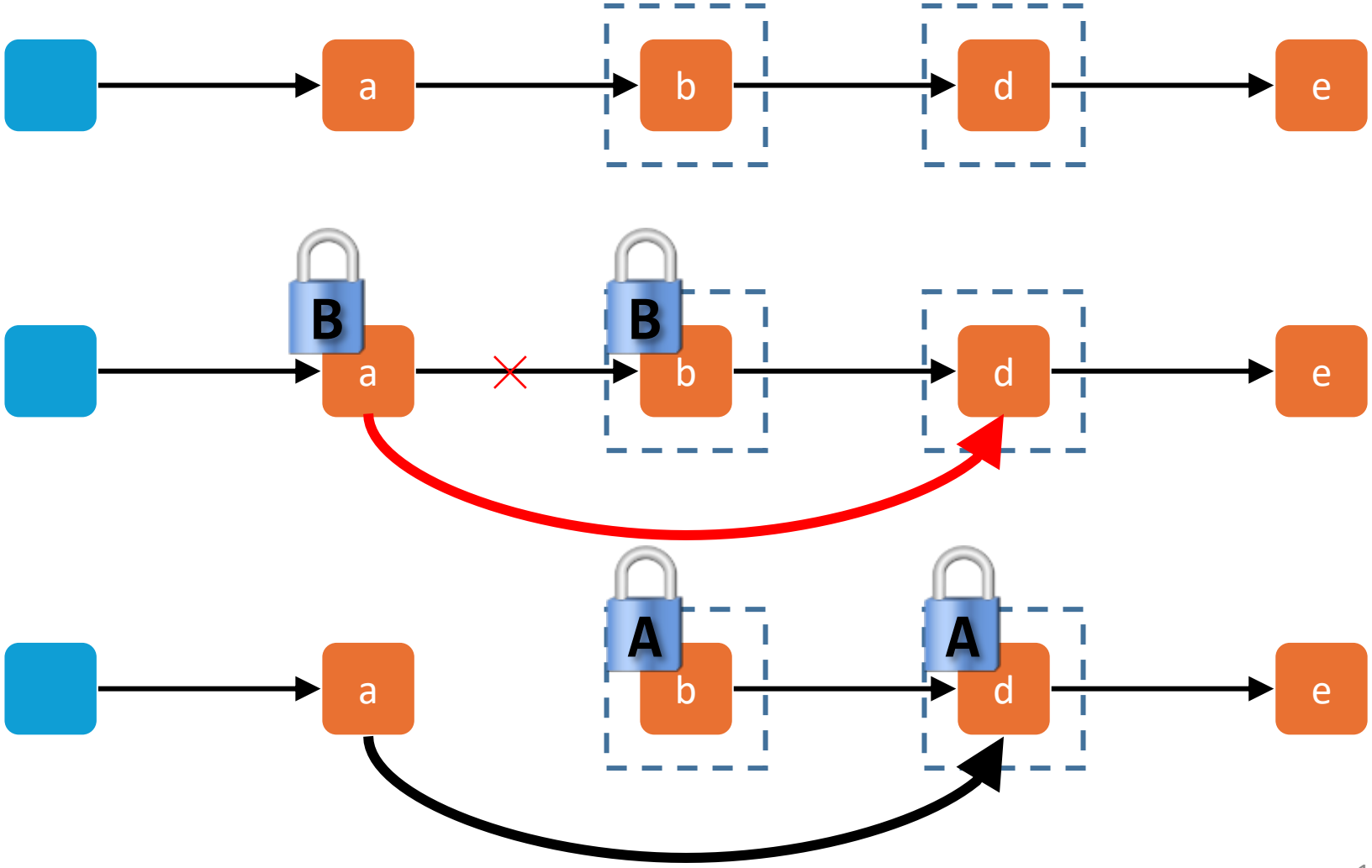
B: remove(b)

A: lock

A: validate: rescan

A: b not reachable

→return false



Lazy Locking

Find nodes to remove (as before)

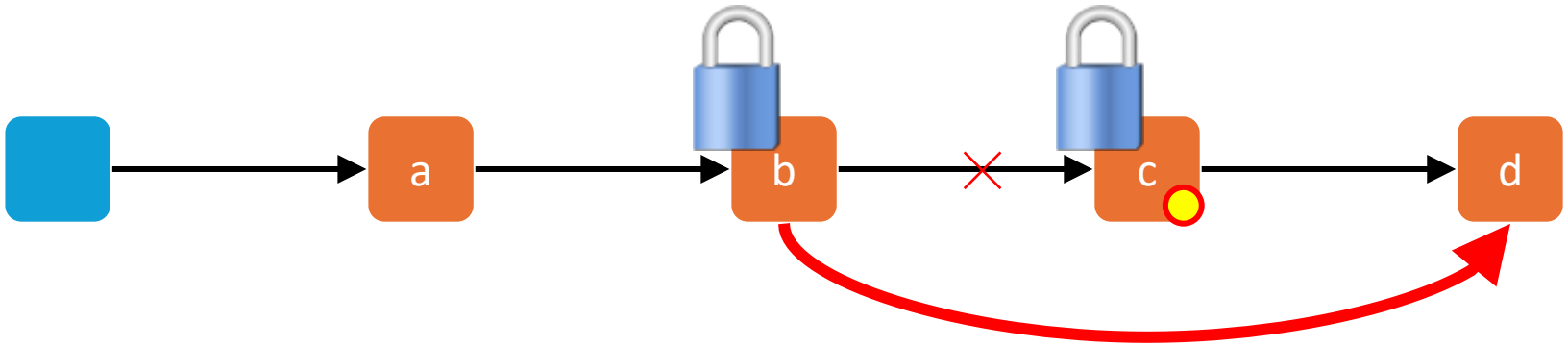
Lock predecessor and current (as before)

Validate (new validation)

Logical delete: mark current node as removed volatile?

Physical delete: redirect predecessor's next

e.g. remove(c)



Fine grained Locking

```
@Override
public boolean add(T item) {
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    Node curr = pred.next;
    curr.lock();
    try {
        while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
        }
        if (curr.key == key) {
            return false;
        }
        Node newNode = new Node(item);
        newNode.next = curr;
        pred.next = newNode;
        return true;
    } finally {
        curr.unlock(); pred.unlock();
    }
}
```

```
@Override
public boolean remove(T item) {
    Node pred = null, curr = null;
    int key = item.hashCode();
    head.lock();
    try {
        pred = head;
        curr = pred.next;
        curr.lock();
        try {
            while (curr.key < key) {
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            if (curr.key == key) {
                pred.next = curr.next;
                return true;
            }
            return false;
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

```
@Override
public boolean contains(T item) {
    Node pred = null, curr = null;
    int key = item.hashCode();
    head.lock();
    try {
        pred = head;
        curr = pred.next;
        curr.lock();
        try {
            while (curr.key < key) {
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            return (curr.key == key);
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

Optimistic Locking

```
@Override
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) { // present
                    return false;
                } else { // not present
                    Node entry = new Node(item);
                    entry.next = curr;
                    pred.next = entry;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

```
@Override
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

```
@Override
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        try {
            pred.lock();
            //curr.lock();
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally {
            pred.unlock();
            //curr.unlock();
        }
    }
}
```

```
private boolean validate(Node pred, Node curr) {
    Node entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred)
            return pred.next == curr;
        entry = entry.next;
    }
    return false;
}
```

Lazy Locking

```
@Override
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = this.head;
        Node curr = head.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) { // presen
                    return false;
                } else { // not present
                    Node Node = new Node(item);
                    Node.next = curr;
                    pred.next = Node;
                    return true;
                }
            }
        } finally { // always unlock
            pred.unlock(); curr.unlock();
        }
    }
}
```

```
@Override
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
      Node pred = this.head;
      Node curr = head.next;
      while (curr.key < key) {
        pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {
            return false;
          } else {
            curr.marked = true;
            pred.next = curr.next;
            return true;
          }
        }
      } finally { // always unlock
        pred.unlock(); curr.unlock();
      }
    }
}
```

```
@Override
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
        curr = curr.next;
    return curr.key == key && !curr.marked;
}
```

```
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr;
}
```

Plan für heute

- Organisation
- Nachbesprechung Assignment 11
- **Theory**
- Intro Assignment 12
- Exam questions
- Kahoot

Readers-writers lock

Recall:

- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!

Readers-writers lock

A new abstract data type for synchronization : The readers/writer lock

- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

$0 \leq$ **writers** ≤ 1

$0 \leq$ **readers**

writers * readers==0

Readers-writers lock

`new:`	make a new lock, initially “not held”
`acquire_write:`	block if currently “held for reading” or “held for writing”, else make “held for writing”
`release_write:`	make “not held”
`acquire_read:`	block if currently “held for writing”, else make/keep “held for reading” and increment *readers count*
`release_read:`	decrement readers count, if 0, make “not held”

Readers-writers lock in Java

```
double readSomething() {
    readerWriterLock.readLock().lock();
    try {
        double value = retrieveDoubleValue();
        return value;
    } finally {
        readerWriterLock.readLock().unlock();
    }
}

Void writeSomething(double new_value) {
    readerWriterLock.writeLock().lock();
    try {
        storeDoubleValue(new_value);
    } finally {
        readerWriterLock.writeLock().unlock();
    }
}
```

No fairness guarantees!

Simple reader-writer lock with monitors

```
class RWLock {
  int writers = 0;
  int readers = 0;

synchronized void acquire_read() {
  while (writers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
  readers++;
}

synchronized void release_read() {
  readers--;
  notifyAll();
}
```

```
synchronized void acquire_write() {
  while (writers > 0 || readers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
  writers++;
}

synchronized void release_write() {
  writers--;
  notifyAll();
}
}
```

Reader-writer lock with monitors

```
class RWLock {
  int writers = 0;
  int readers = 0;
  int writersWaiting = 0;

  synchronized void acquire_read() {
    while (writers>0 || writersWaiting>0)
      try { wait(); }
      catch (InterruptedException e) {}
    readers++;
  }

  synchronized void release_read() {
    readers--;
    notifyAll();
  }

  synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    writersWaiting--;
    writers++;
  }

  synchronized void release_write() {
    writers--;
    notifyAll();
  }
}
```

Fair reader-writer lock with monitors

```
class RWLock{
  int writers = 0; int readers = 0;
  int writersWaiting = 0; int readersWaiting = 0;
  int writersWait = 0;

  synchronized void acquire_read() {
   readersWaiting++;
   while (writers>0 || (writersWaiting>0 && writersWait <= 0))
      try { wait(); }
      catch (InterruptedException e) {}
   readersWaiting --;
   writersWait--;
   readers++;
  }

  synchronized void release_read() {
   readers--;
   notifyAll();
  }

  synchronized void acquire_write() {
   writersWaiting++;
   while (writers > 0 || readers > 0 || writersWait > 0)
       try { wait(); }
      catch (InterruptedException e) {}
   writersWaiting--;
   writers++;
  }

  synchronized void release_write() {
   writers--;
   writersWait = readersWaiting;
   notifyAll();
  }
}
```

Lock-Free Programming

Recap: Definitions for blocking synchronization

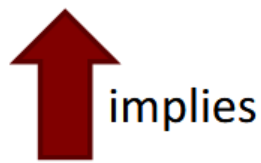
- D**eadlock**: group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed

- L**ivelock**: competing processes are able to detect a potential deadlock condition but make no observable progress while trying to resolve it

- S**tarvation**: repeated but unsuccessful attempt of a recently unblocked process to continue its execution

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 !

CAS (again)

compare old with data at memory location

if and only if data at memory
equals old overwrite data with
new

return previous memory value
(in Java: return whether CAS succeeded)

```
int CAS (memref a, int old, int new)
atomic {
    oldval = mem[a];
    if (old == oldval)
        mem[a] = new;
    return oldval;
}
```

- We assume the operation CAS itself is wait free, i.e., it finishes after constant time always

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

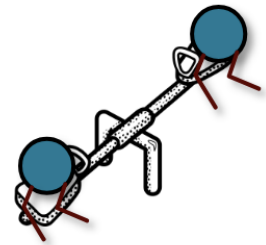
Positive result of CAS of (c) *suggests* that no other thread has written between (a) and (c)

In general

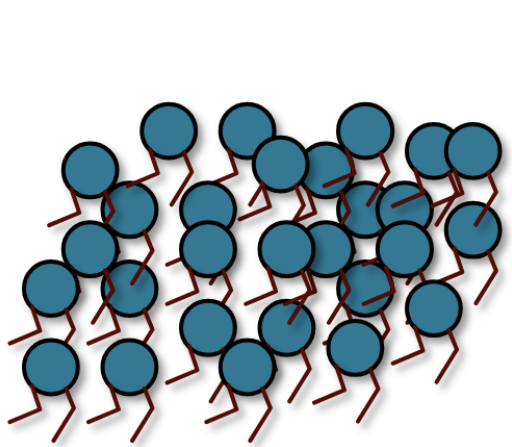
- Using CAS is **atomic**: the *operation* (read modify write) is indivisible
- However, using CAS to update e.g., a counter is only **lock-free**, not **wait-free**: you don’t get a guarantee that *your* thread will finish in a bounded number of steps under unbounded contention

Lock free data structures

Locks performance



- **Uncontended case**
  - when threads do not compete for the lock
  - lock implementations try to have minimal overhead
  - typically "just" the cost of an atomic operation



- **Contended case**
  - when threads do compete for the lock
  - can lead to significant performance degradation
  - also, starvation
  - there exist lock implementations that try to address these issues

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!

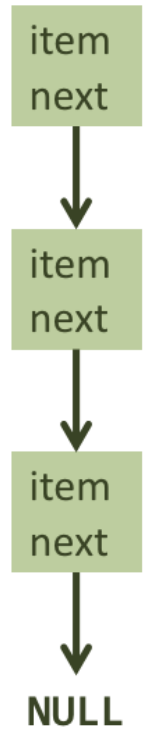
Blocking semantics (wait until acquire lock)

- 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?

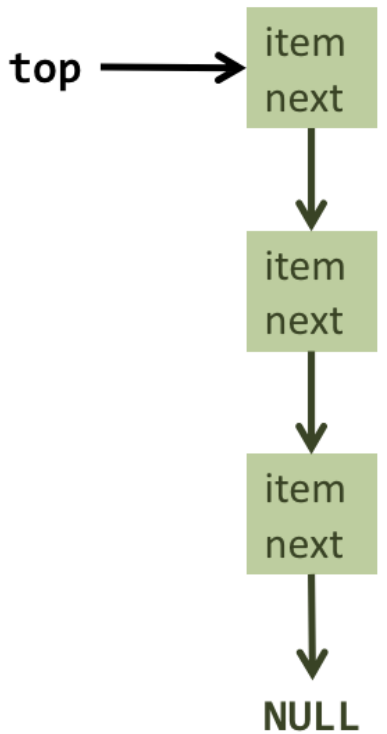
Stack Node

```
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;
   }

}
```



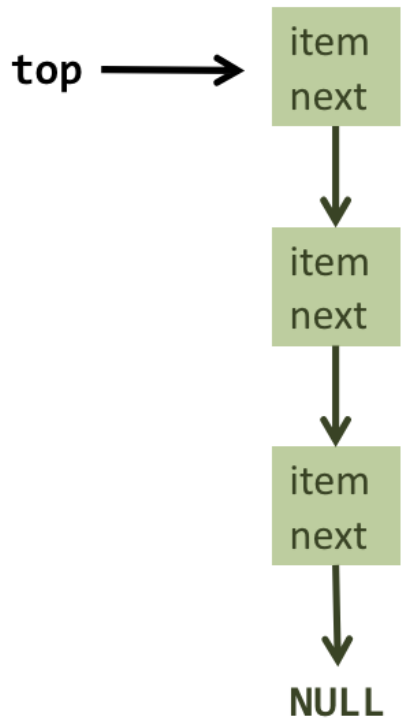
Blocking Stack

```
public class BlockingStack {
   Node top = null;

   synchronized public void push(Long item) {
      top = new Node(item, top);
   }

   synchronized public Long pop() {
      if (top == null)
         return null;
      Long item = top.item;
      top = top.next;
      return item;
   }
}
```



Non-blocking Stack

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

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

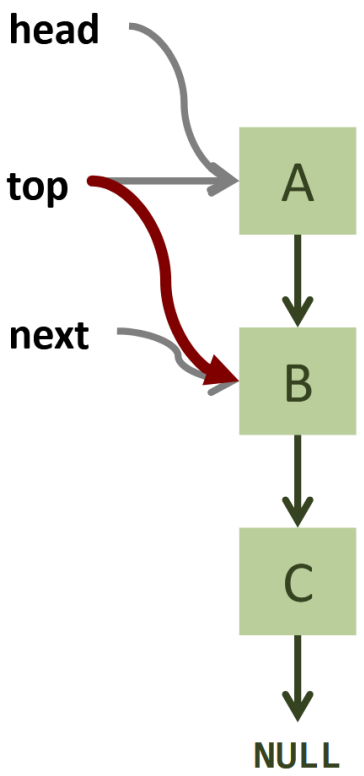


pop

```
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

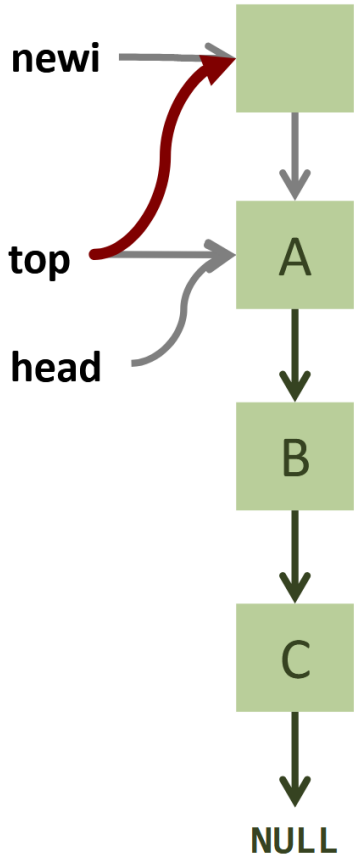


push

```
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



What's the benefit?

Lock-free programs are deadlock-free by design.

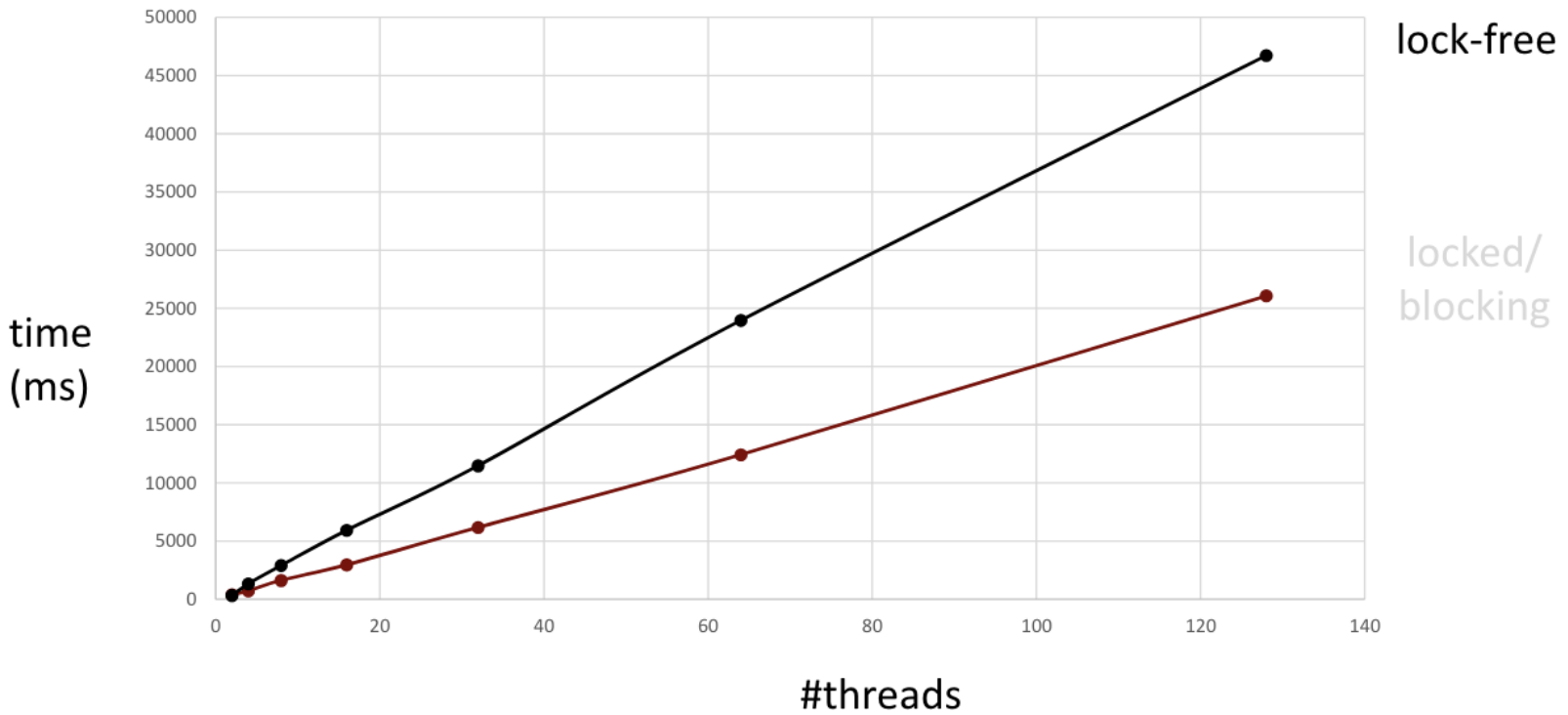
How about performance?

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

n threads

100,000 push/pop operations

10 times

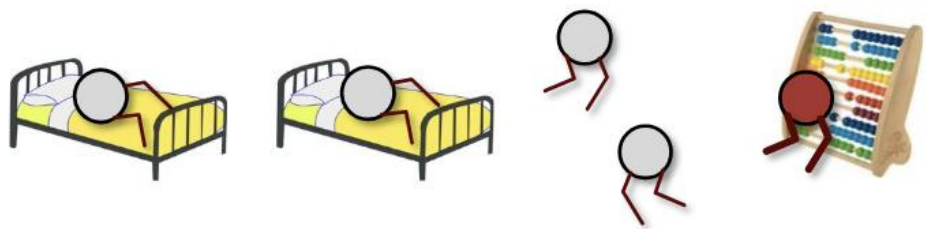


Performance

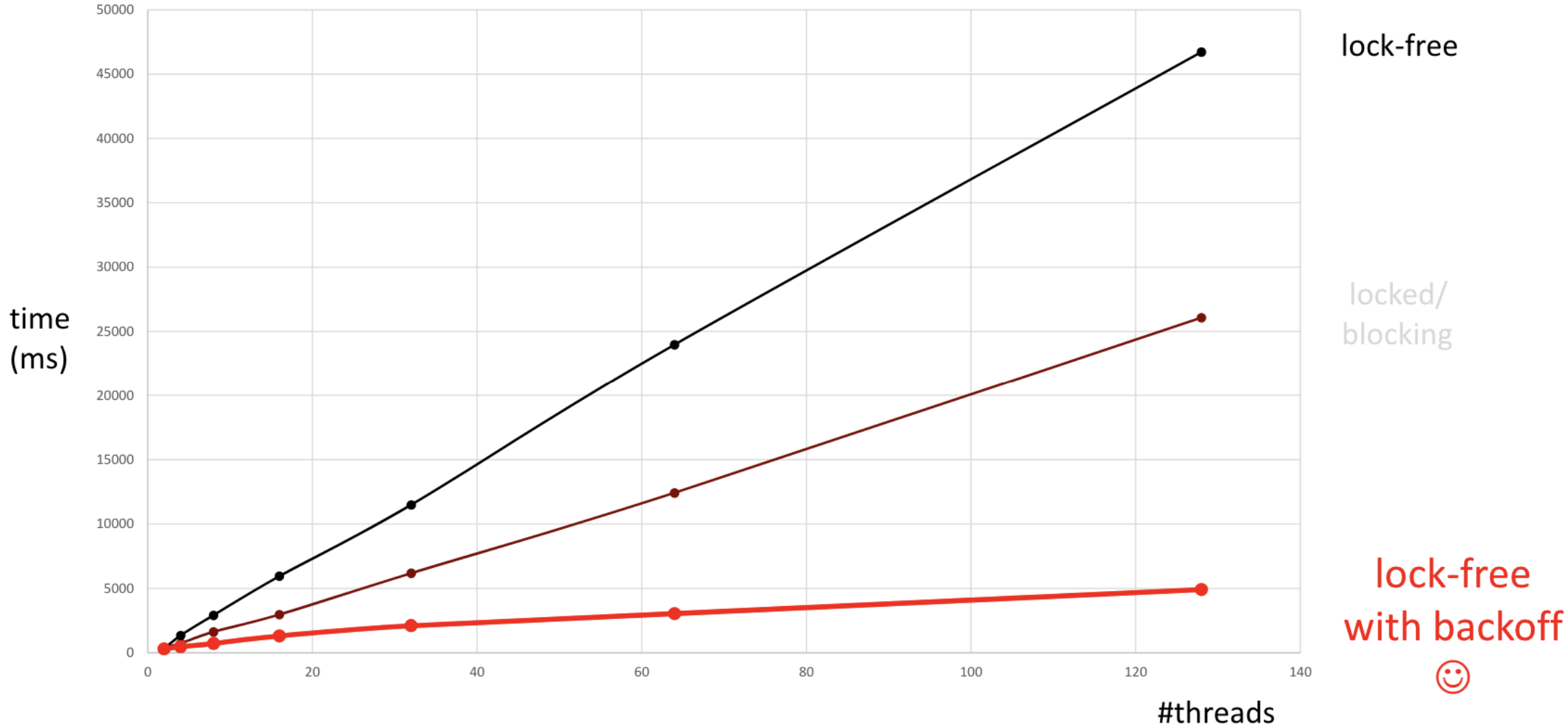
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

Atomic operations are expensive and contention can still be a problem.

→ Backoff, again.



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.

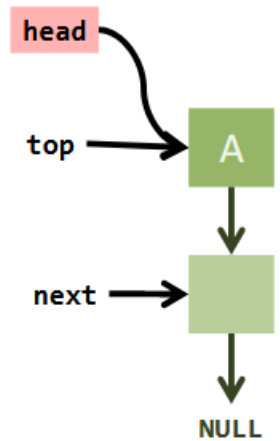
***So far* this works.**

Using the node pool

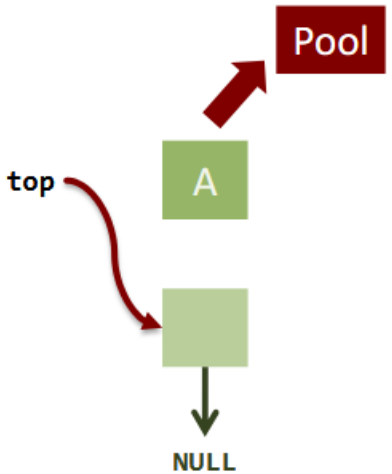
```
public void push(Long item) {
    Node head;
    Node new = pool.get(item);
    do {
        head = top.get();
        new.next = head;
    } while (!top.compareAndSet(head, new));
}

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

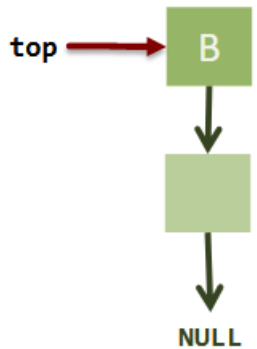
Thread X
in the middle
of pop: after read
but before CAS



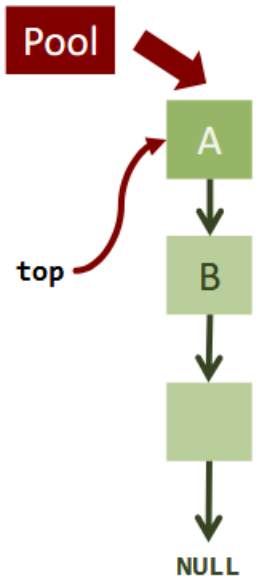
Thread Y
pops A



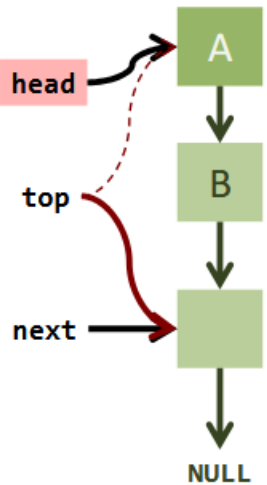
Thread Z
pushes B



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)

LOCK FREE LIST-BASED SET

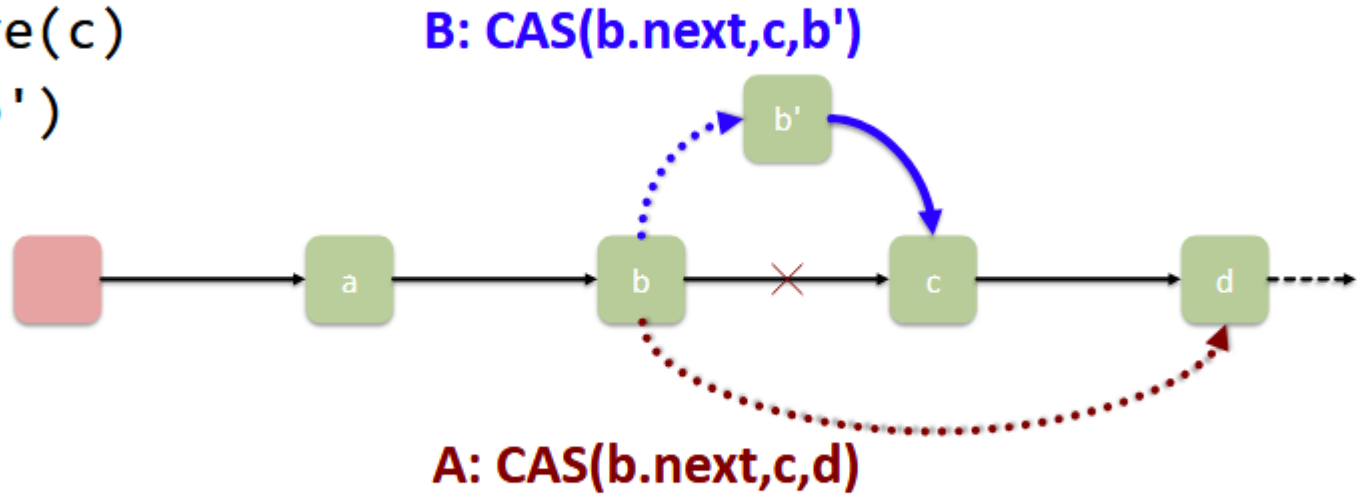
(NOT SKIP LIST!)

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

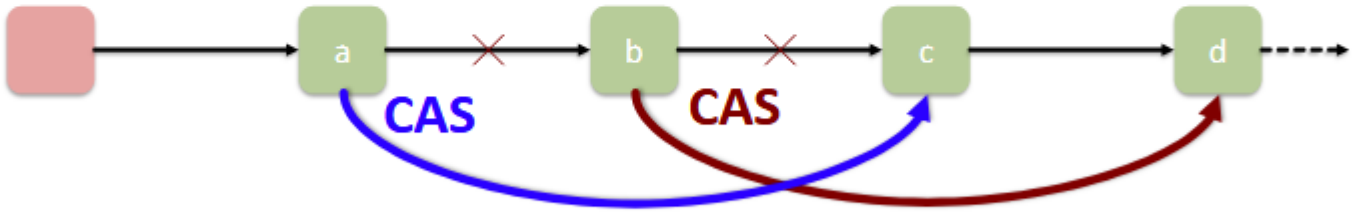
So does this CAS approach work generally??

- 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)

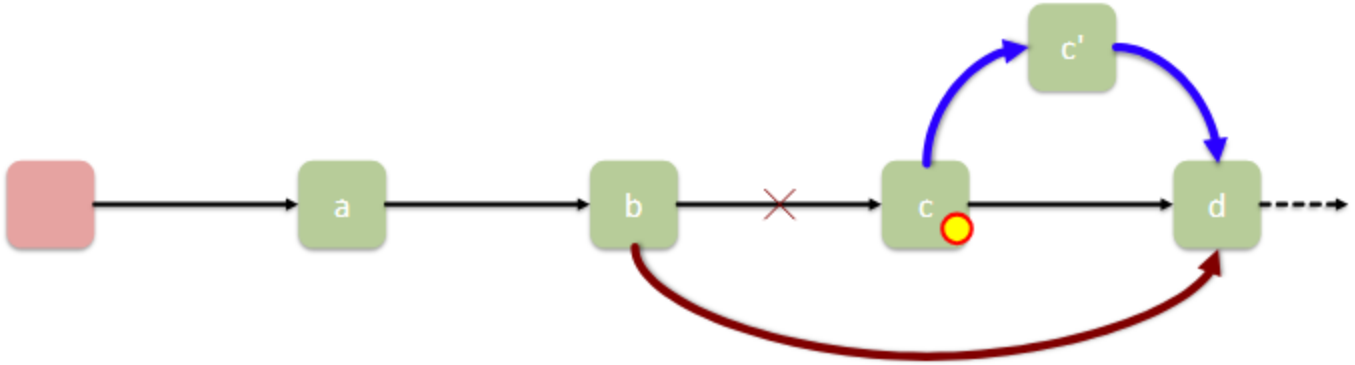
Second Approach, try to fix the issue by using mark bit which tells us if an element was removed

A: remove(c)

B: add(c')

B: c.mark ?

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



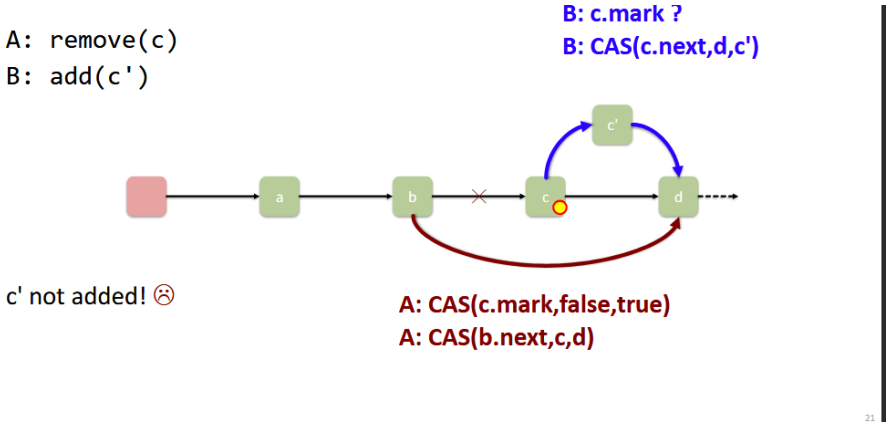
c' not added! ☹

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

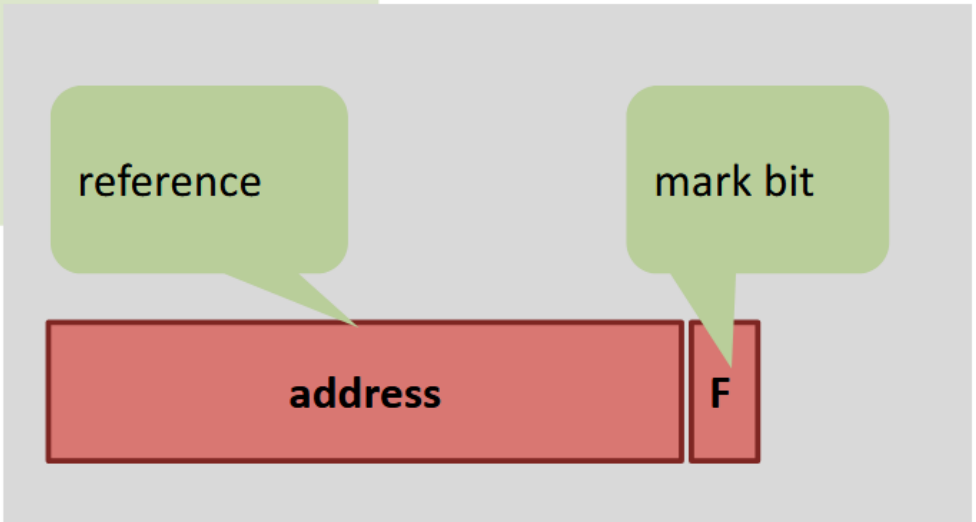
- We want to set the mark bit and the next pointer in one step otherwise we face the issue from the previous slide

- 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

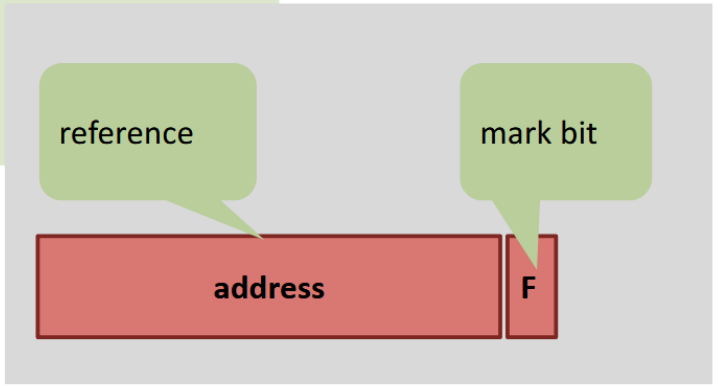


2^{64}Bytes=562,949,953,421,312 Petabytes

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

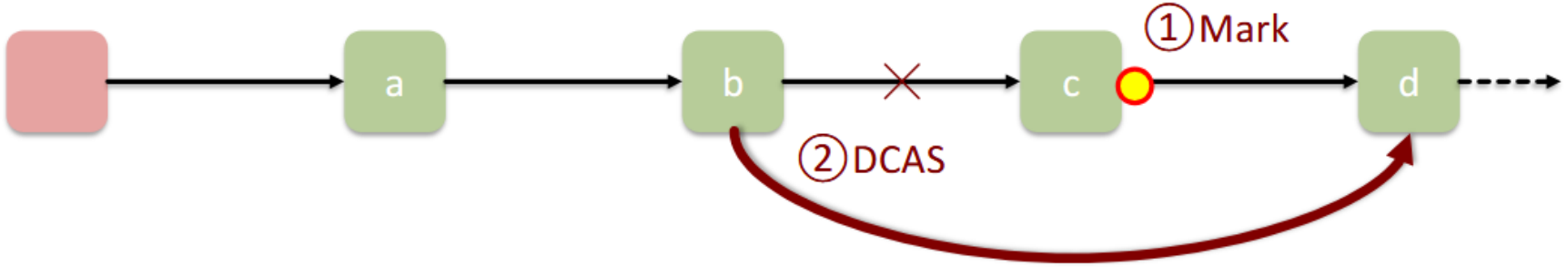
- Atomically
  - Swing reference *and*
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

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]);

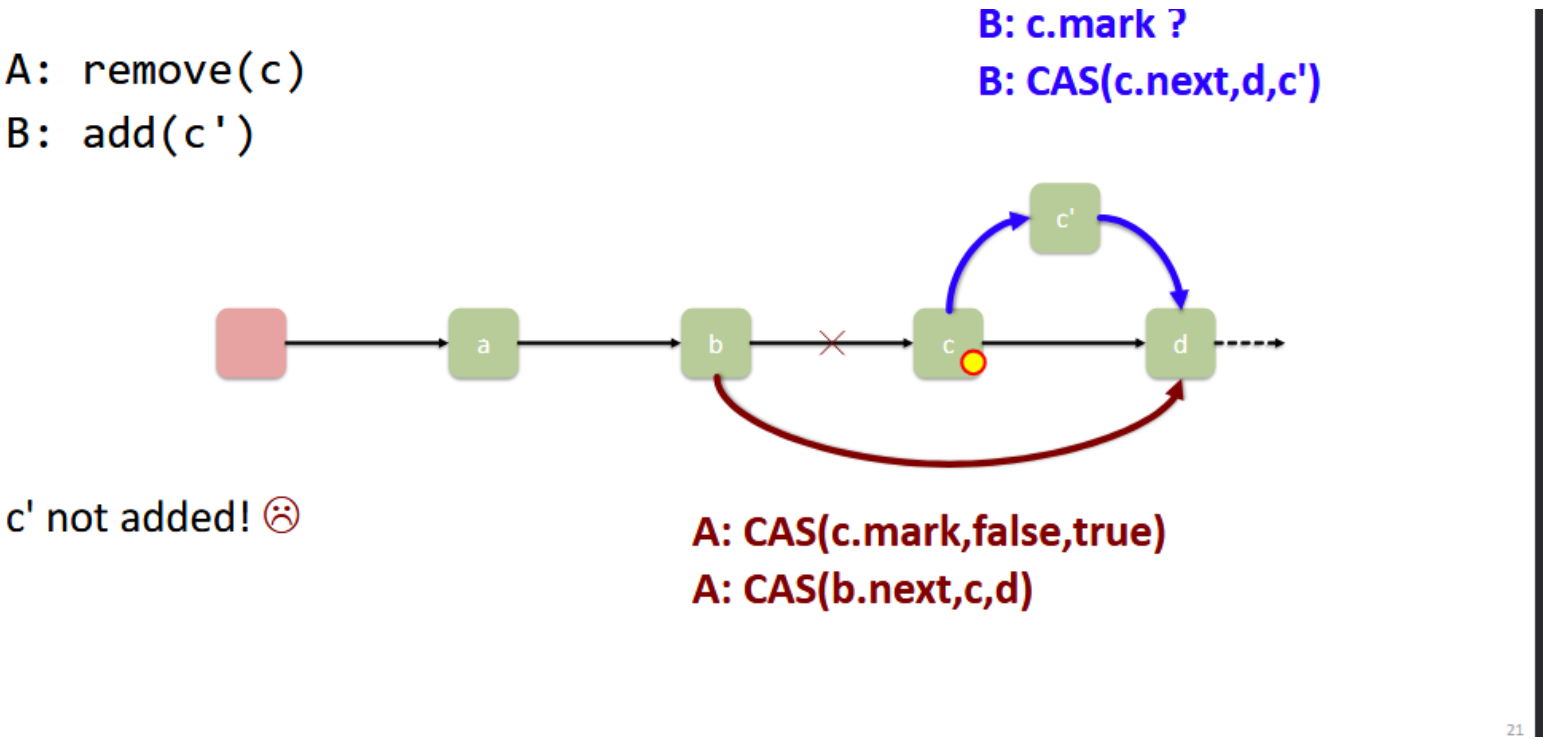


- Note that

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?



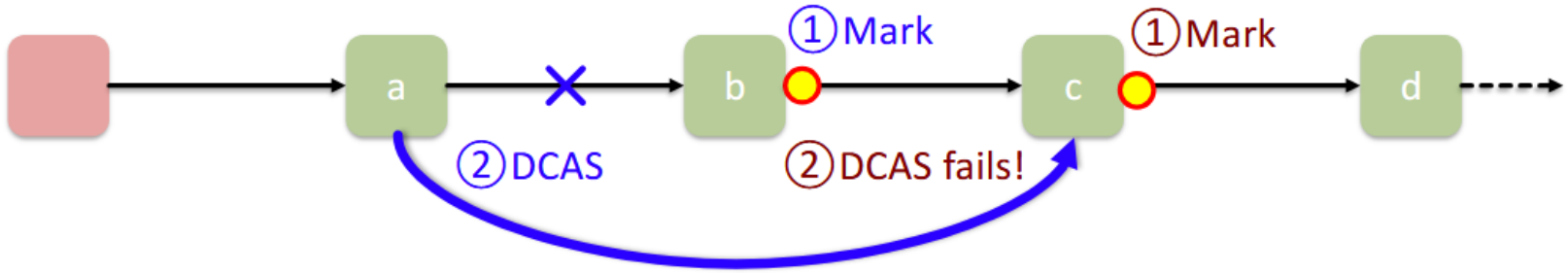
- Yes, we can't have bad interleaving anymore because thread B checks c.mark and updates c.next in one step
- Thread B will either see mark = 0 → can insert c' in one step or mark = 1 which means it needs to retry

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!

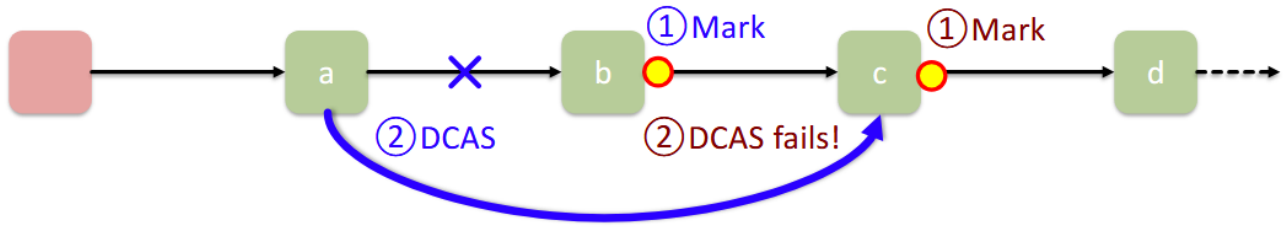
1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

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

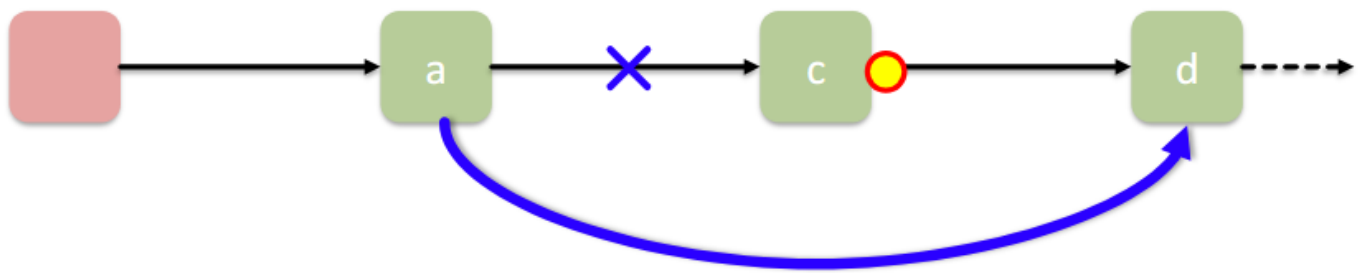
Traversing the list

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;
      }
   }
}
```

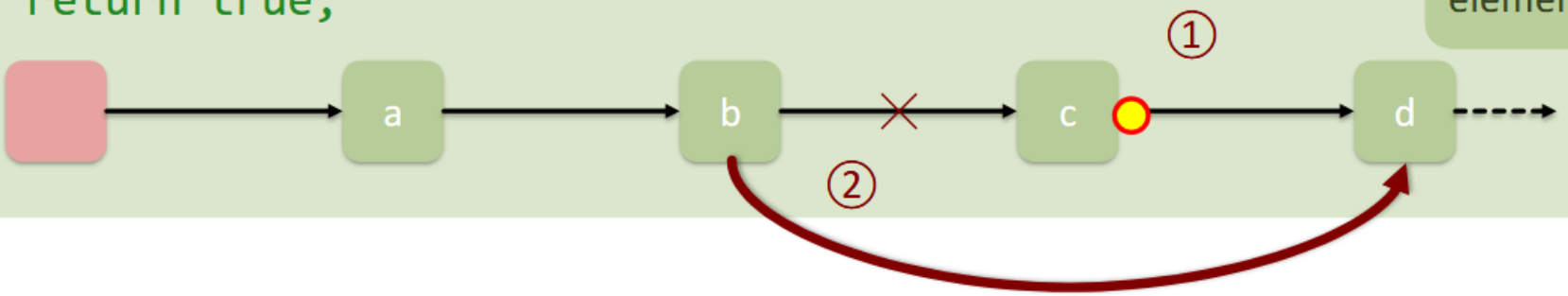
Find element and prev element from key

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

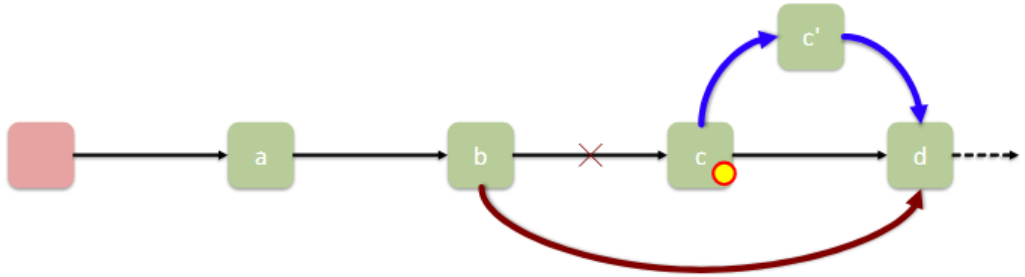
and try to insert. If insertion fails (next set by other thread or mark bit set), retry

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

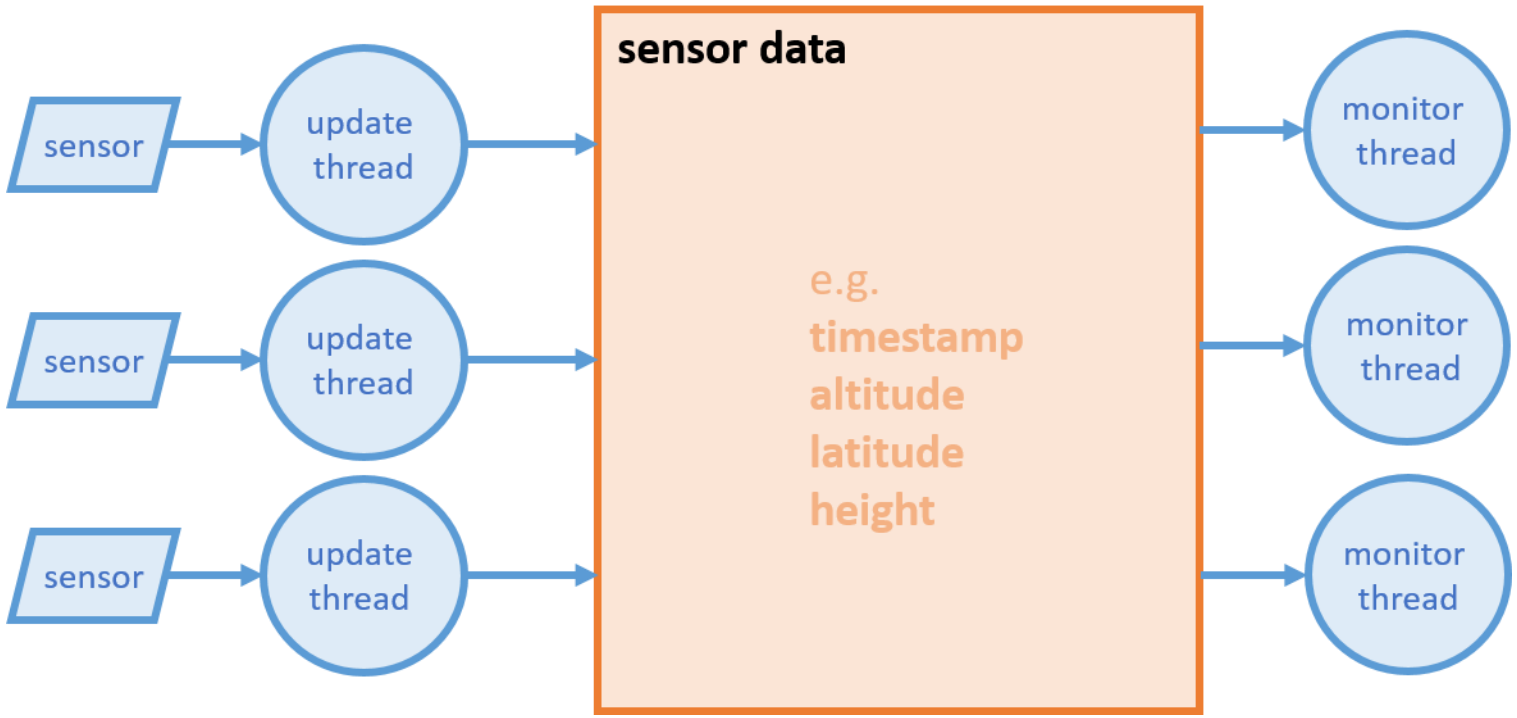
- We used a special variant of DCAS (double compare and swap) in order to be able check two conditions at once.
  This DCAS was possible because one bit was free in the reference.
- We used a lazy operation in order to deal with a consistency problem. Any thread is able to repair the inconsistency.
  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

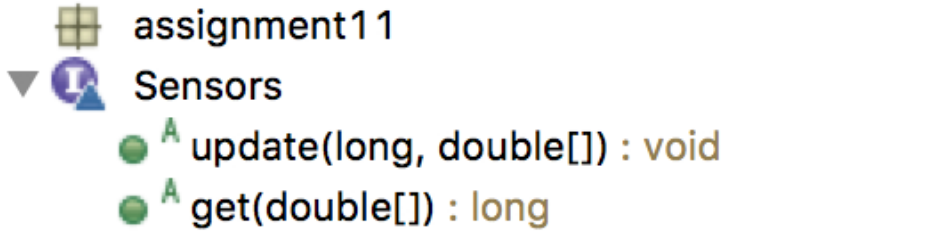
- Organisation
- Nachbesprechung Assignment 11
- Theory
- **Intro Assignment 12**
- Kahoot
- Exam questions

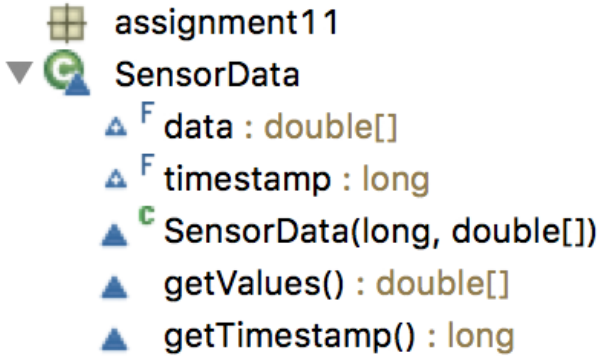
Assignment 12

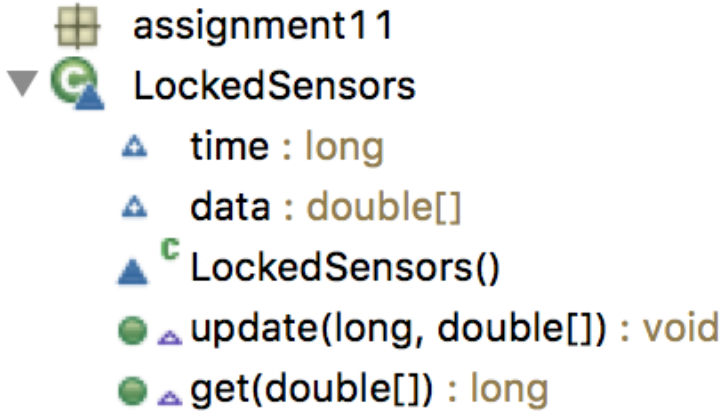
- Multisensor System.

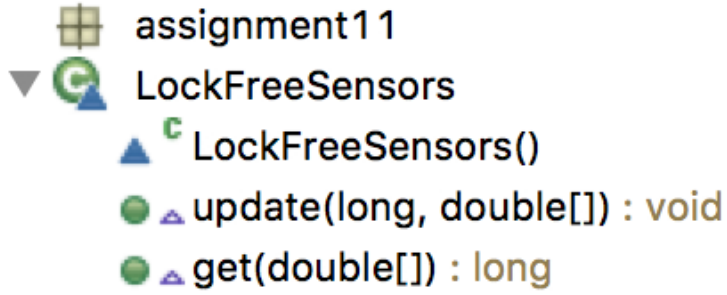


Multisensor System









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)

Hints:

- Before you implement the readers-writers lock based version, start with a simple locked version in order to understand. Then try a readers-writers lock but be aware that the Java-implementation does not give fairness guarantees. What can this imply? In any case, you have the code from the lecture slides presenting a fair RW-Lock implementation.

- The lock-free implementation solutions does NOT rely on mechanisms such as Double-Compare- And-Swap. Also it does not rely on a lazy update mechanism. Somehow you have to make sure that with a single reference update you change all data at once. How?

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

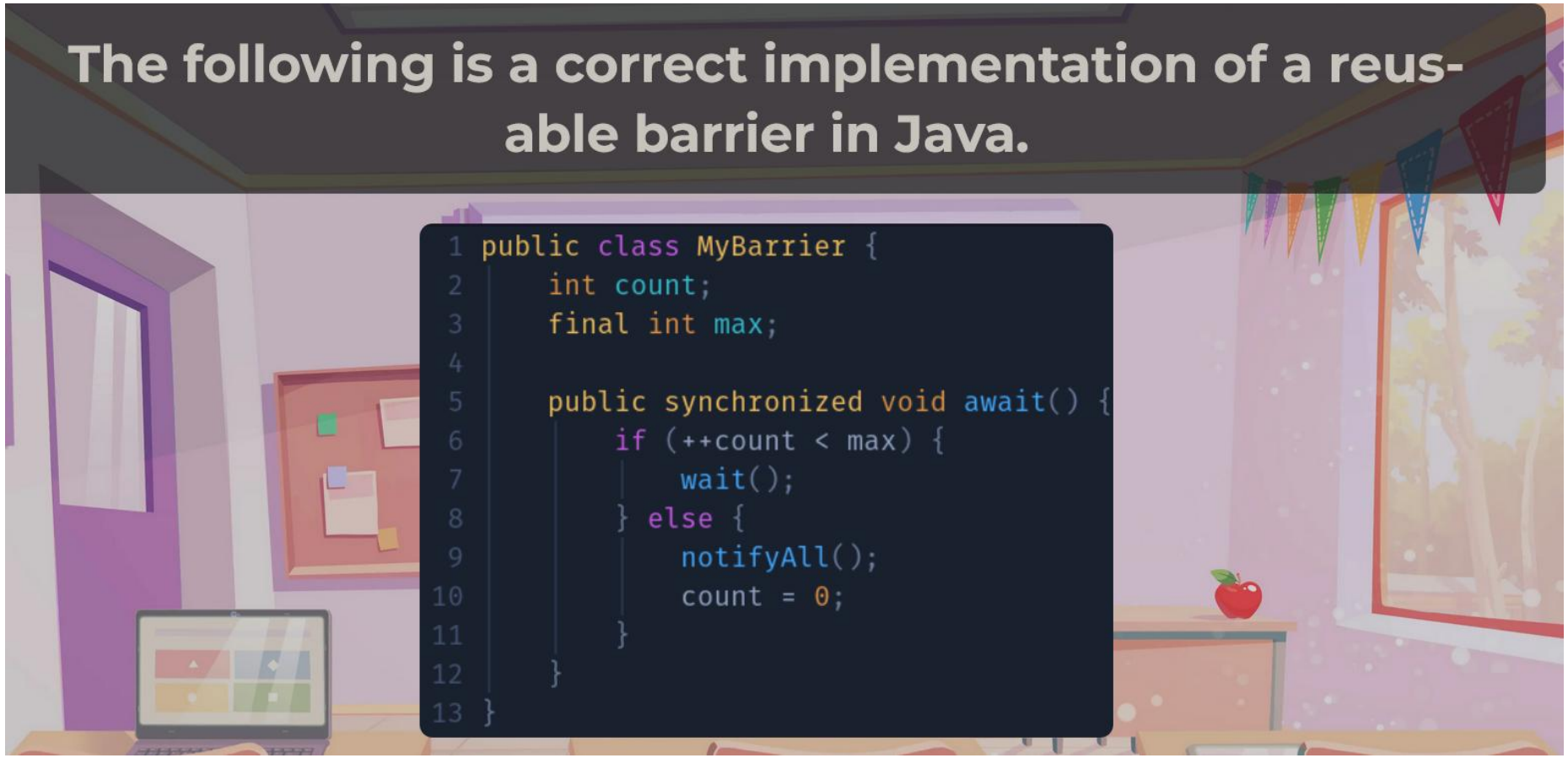
Questions

Plan für heute

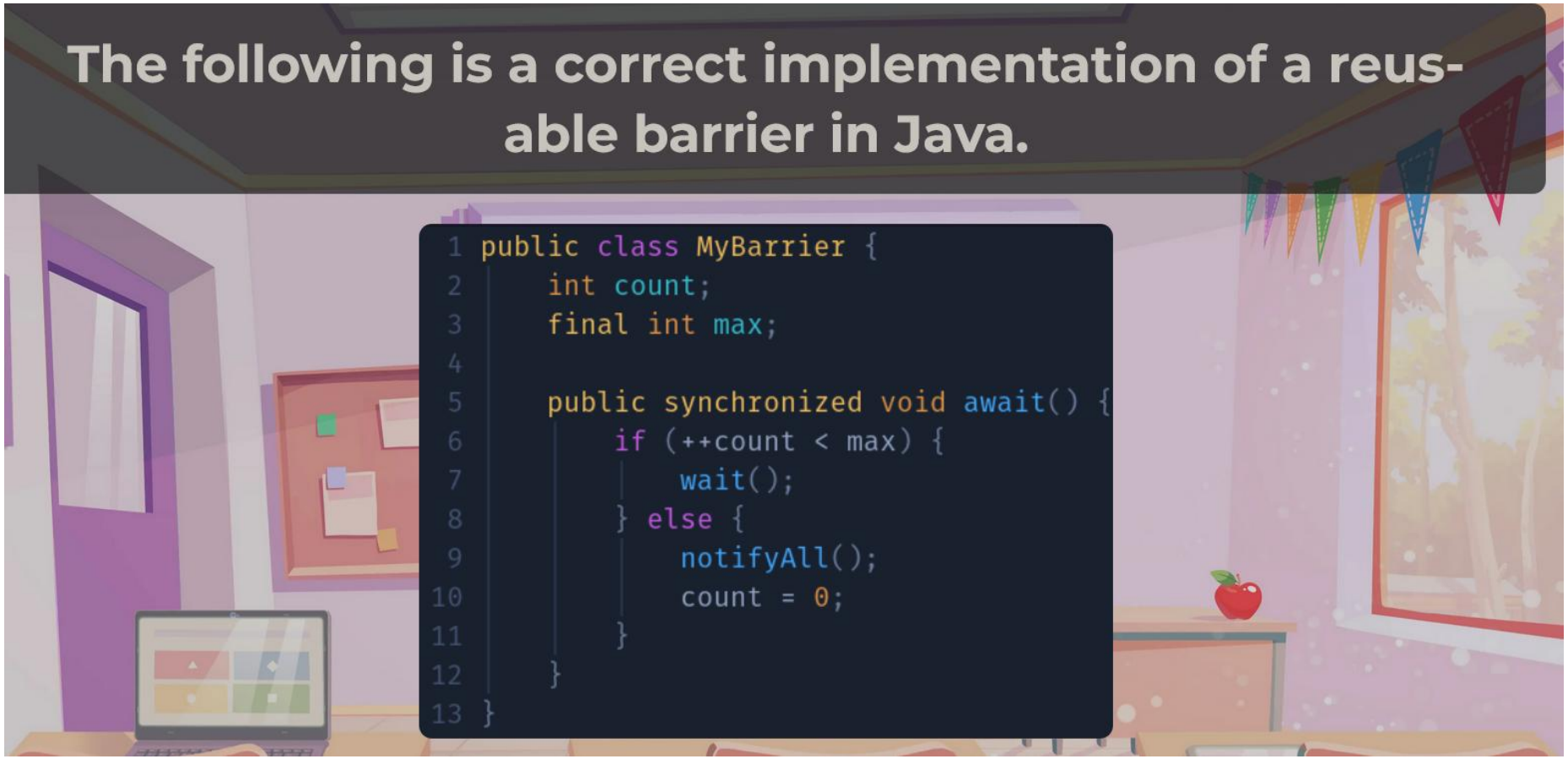
- Organisation
- Nachbesprechung Assignment 11
- Theory
- Intro Assignment 12
- **Kahoot**
- Exam questions

Kahoot!

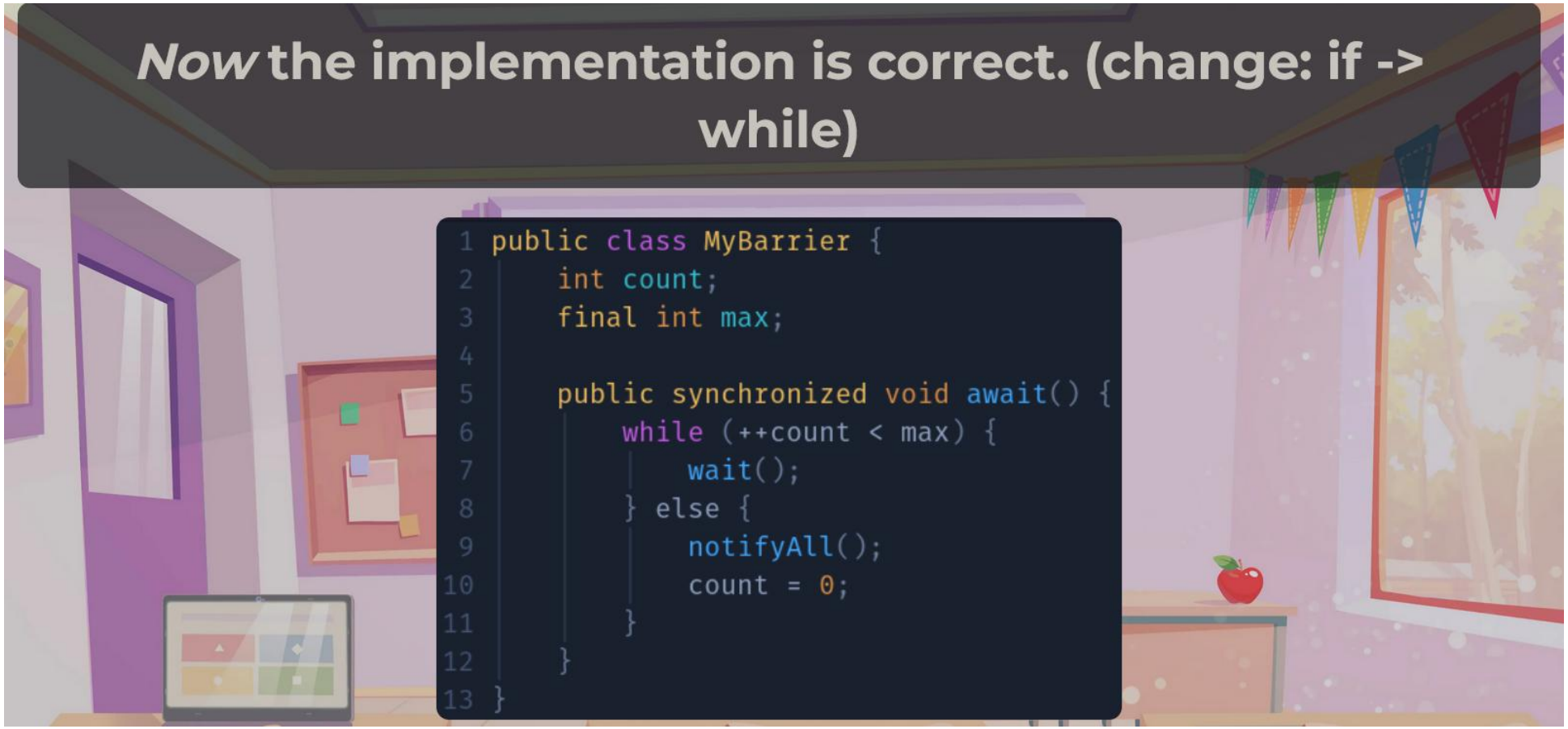
The following is a correct implementation of a reusable barrier in Java.

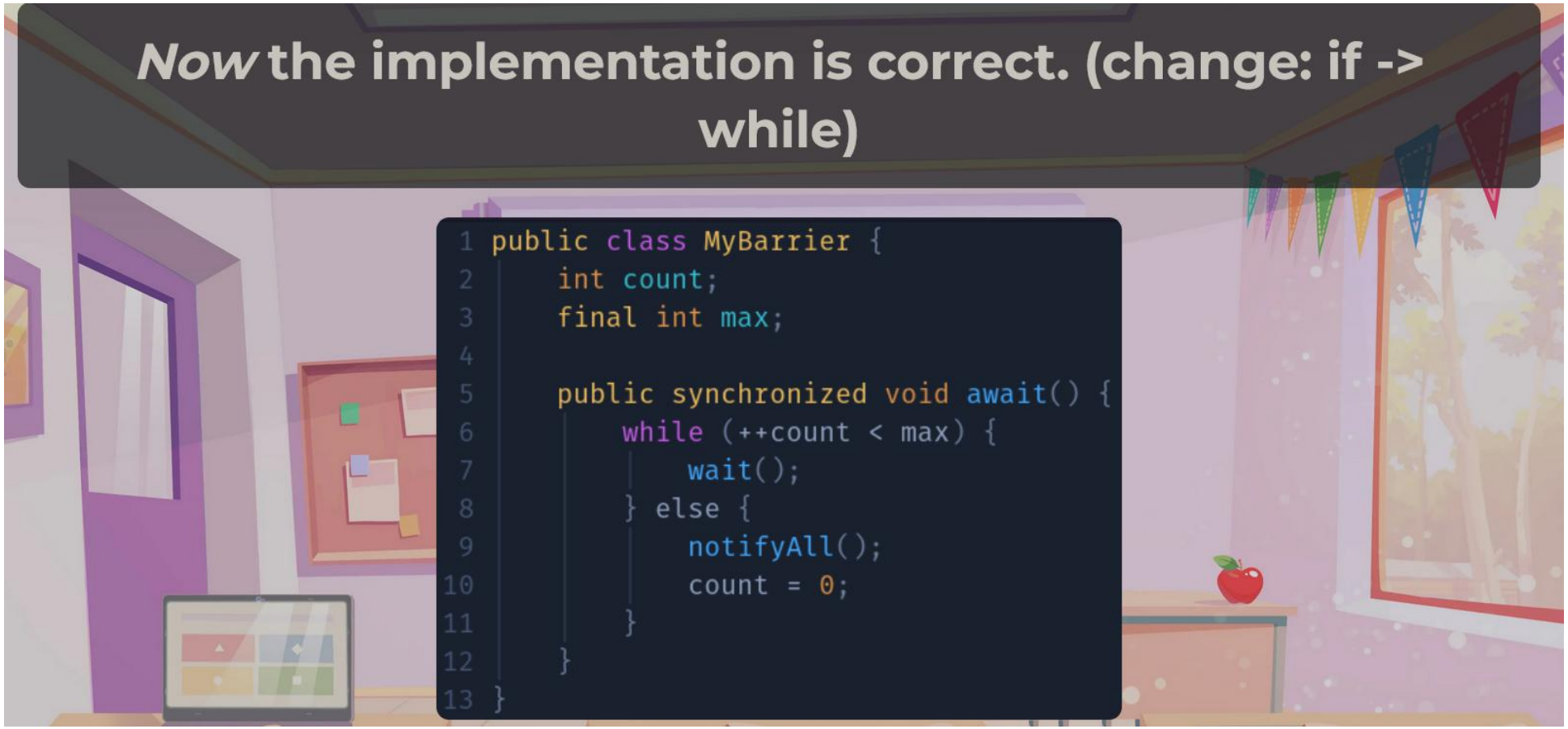


```java
public class MyBarrier {
    int count;
    final int max;

    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 11
- Theory
- Intro Assignment 12
- 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!