Parallele Programmierung FS25

Exercise Session 7

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Exercise 6
- Summary Part 1
- Intro Lecture Part 2
- Intro Exercise 7
- 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

Evaluation

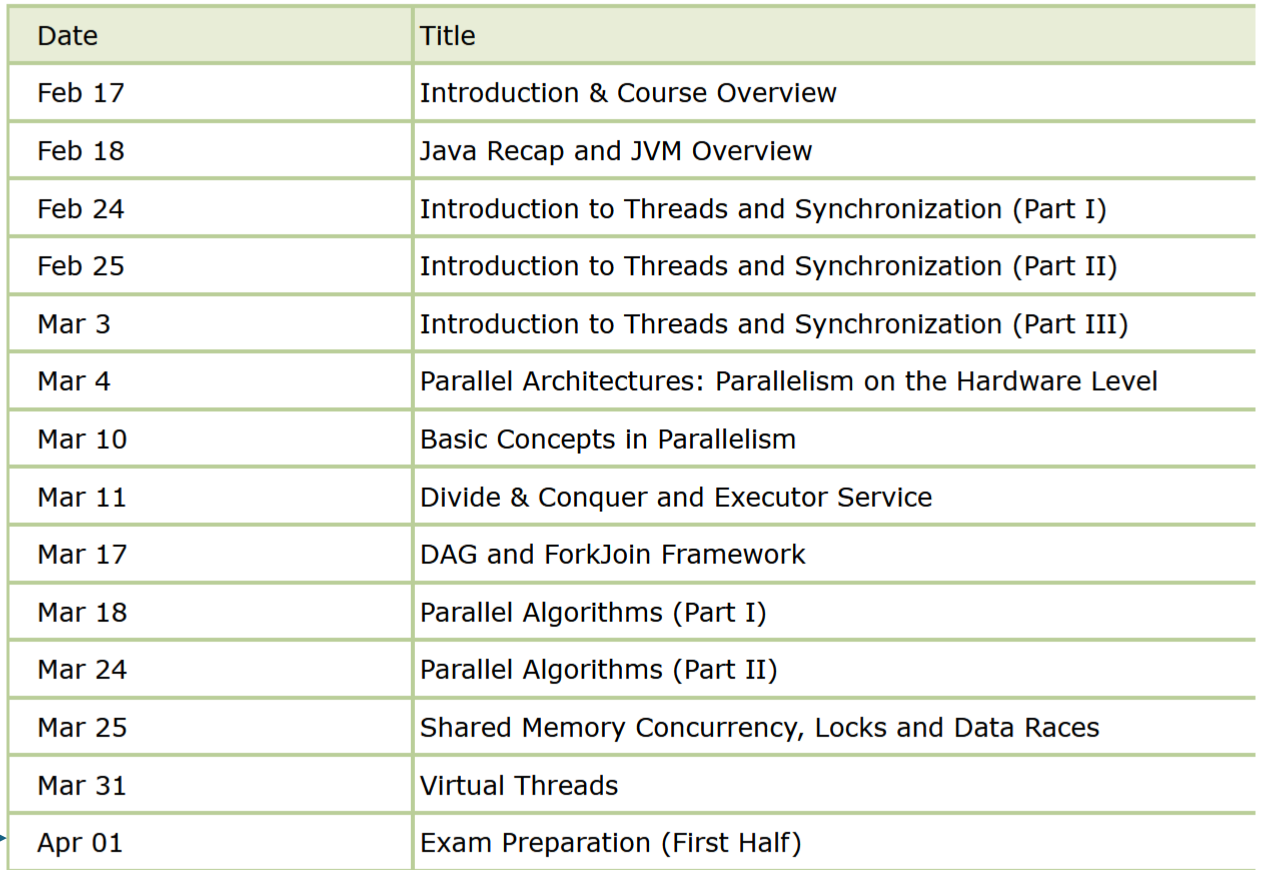
- Please fill in the evaluation (~5min)

- It is anonymous
- (you don’t need to be logged in)

- It helps us improving the exercise sessions

Organisation

- Wo sind wir jetzt?



Date	Title
Feb 17	Introduction & Course Overview
Feb 18	Java Recap and JVM Overview
Feb 24	Introduction to Threads and Synchronization (Part I)
Feb 25	Introduction to Threads and Synchronization (Part II)
Mar 3	Introduction to Threads and Synchronization (Part III)
Mar 4	Parallel Architectures: Parallelism on the Hardware Level
Mar 10	Basic Concepts in Parallelism
Mar 11	Divide & Conquer and Executor Service
Mar 17	DAG and ForkJoin Framework
Mar 18	Parallel Algorithms (Part I)
Mar 24	Parallel Algorithms (Part II)
Mar 25	Shared Memory Concurrency, Locks and Data Races
Mar 31	Virtual Threads
Apr 01	Exam Preparation (First Half)

First part done! 🥳

Plan für heute

- Organisation
- **Nachbesprechung Exercise 6**
- Summary Part 1
- Intro Lecture Part 2
- Intro Exercise 7
- Exam Questions
- Kahoot

Post-Discussion Exercise 6

Merge Sort

Discussion of solution

Longest Sequence

Given a sequence of numbers:

$$[1, 9, 4, 3, 3, 8, \boxed{7, 7, 7,} 0]$$

find the longest sequence of the same consecutive number

Longest Sequence

```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results




      // return the longest result
    }
  }
}
```

Outline almost as before, except:

Longest Sequence

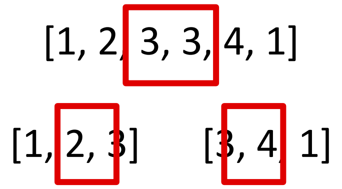
```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results

      // check that result is not in between the pieces


      // return the longest result
    }
  }
}
```

Outline almost as before, except:

Longest Sequence

```
public class LongestSequenceMulti extends RecursiveTask<Sequence> {

  protected Sequence compute() {
    if (// work is small)
      // do the work directly

    else {
      // split work into pieces

      // invoke the pieces and wait for the results

      // check that result is not in between the pieces


      // return the longest result
    }
  }
}
```

Outline almost as before, except:



Longest Sequence

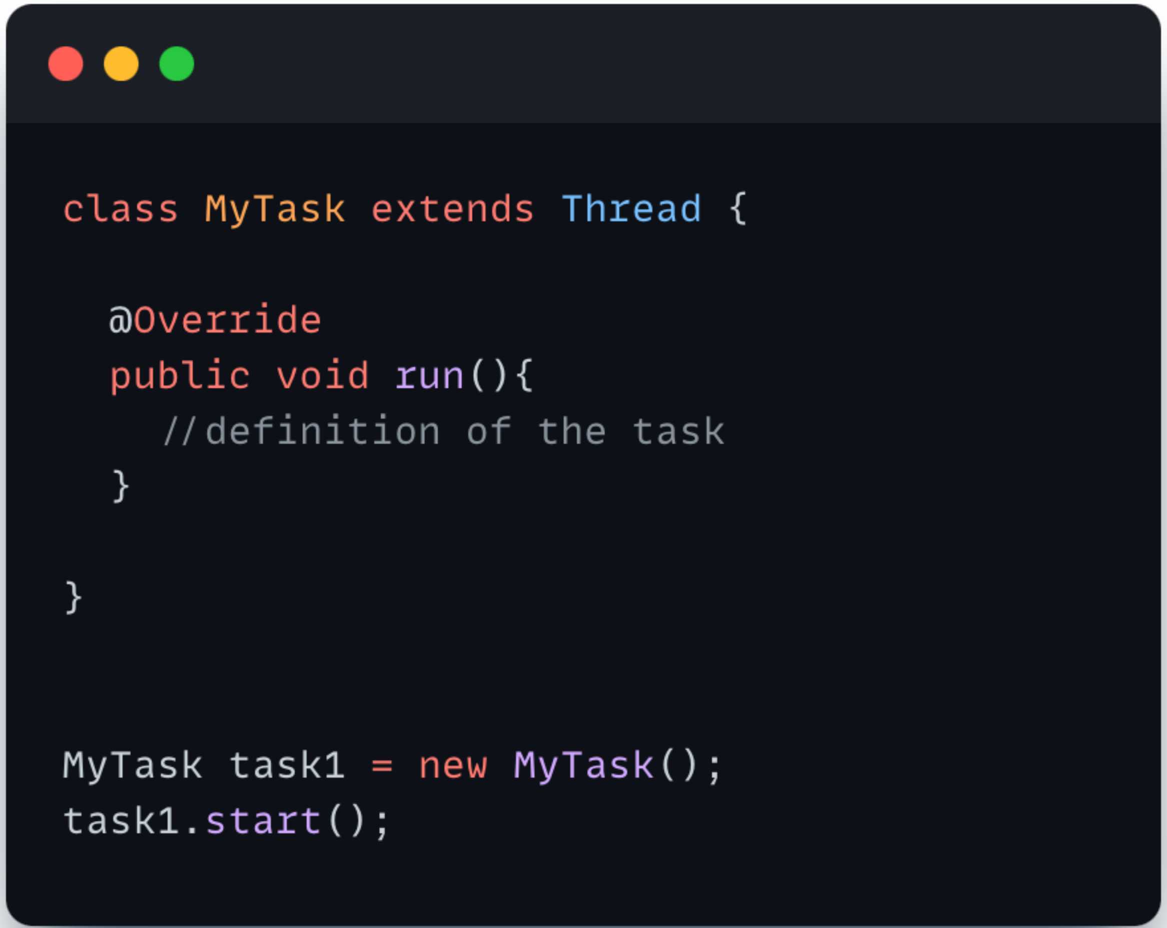
See code

Plan für heute

- Organisation
- Nachbesprechung Exercise 6
- **Summary Part 1**
- Intro Lecture Part 2
- Intro Exercise 7
- Exam Questions
- Kahoot

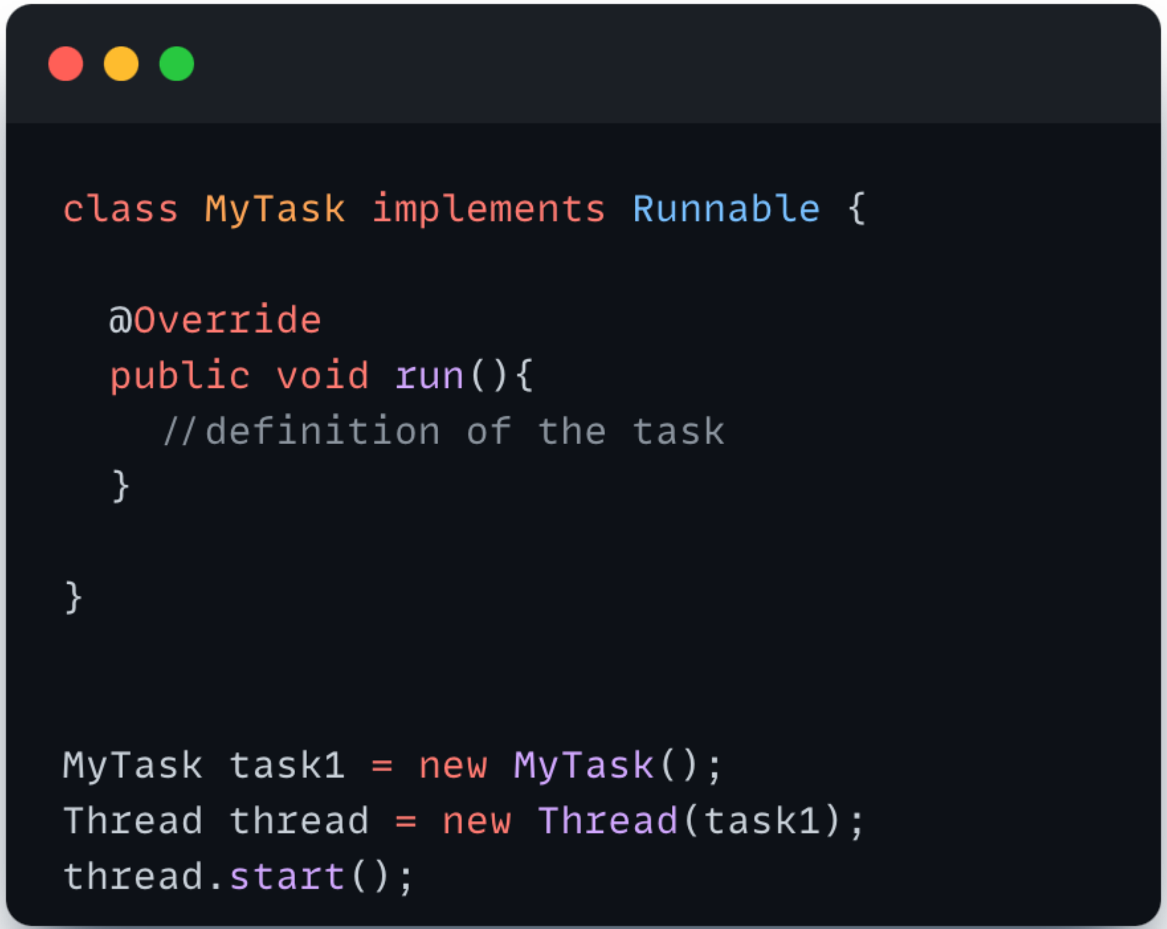
Lecture Recap

Creating Threads: extends Thread (old)



```java
class MyTask extends Thread {

  @Override
  public void run(){
    //definition of the task
  }

}


MyTask task1 = new MyTask();
task1.start();
```

Creating Threads: impl Runnable (better)



```
class MyTask implements Runnable {

  @Override
  public void run(){
    //definition of the task
  }

}


MyTask task1 = new MyTask();
Thread thread = new Thread(task1);
thread.start();
```

Java Threads: some key points

Every Java program has at least one execution thread

- First execution thread calls `main()`

Each call to **`start()`** method of a `Thread` object creates an actual execution thread

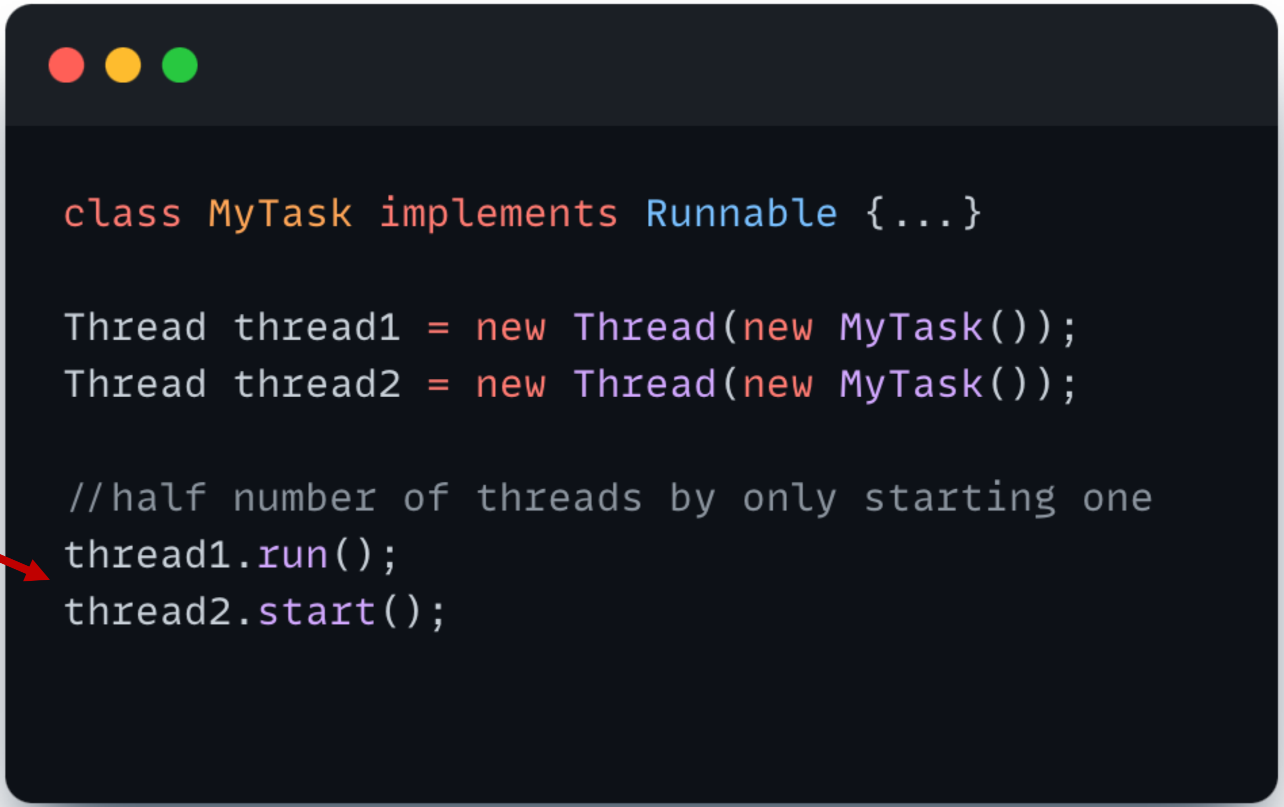
Program ends when all threads (non-daemon threads) finish.

Threads can continue to run even if `main()` returns

Creating a `Thread` object does not start a thread

Calling `run()` doesn’t start thread either (need to call start()!)

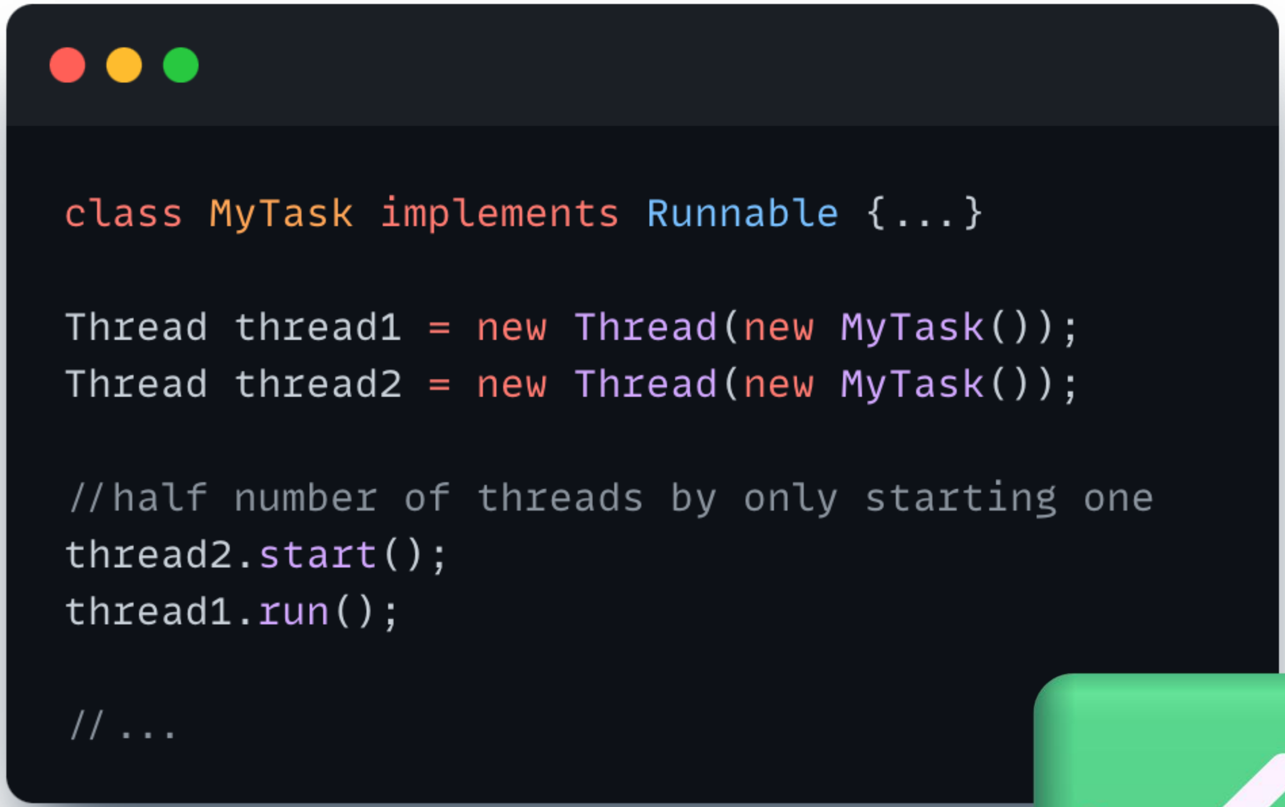
Thread Quiz: Spot the mistake



```java
class MyTask implements Runnable {...}

Thread thread1 = new Thread(new MyTask());
Thread thread2 = new Thread(new MyTask());

//half number of threads by only starting one
thread1.run();
thread2.start();
```

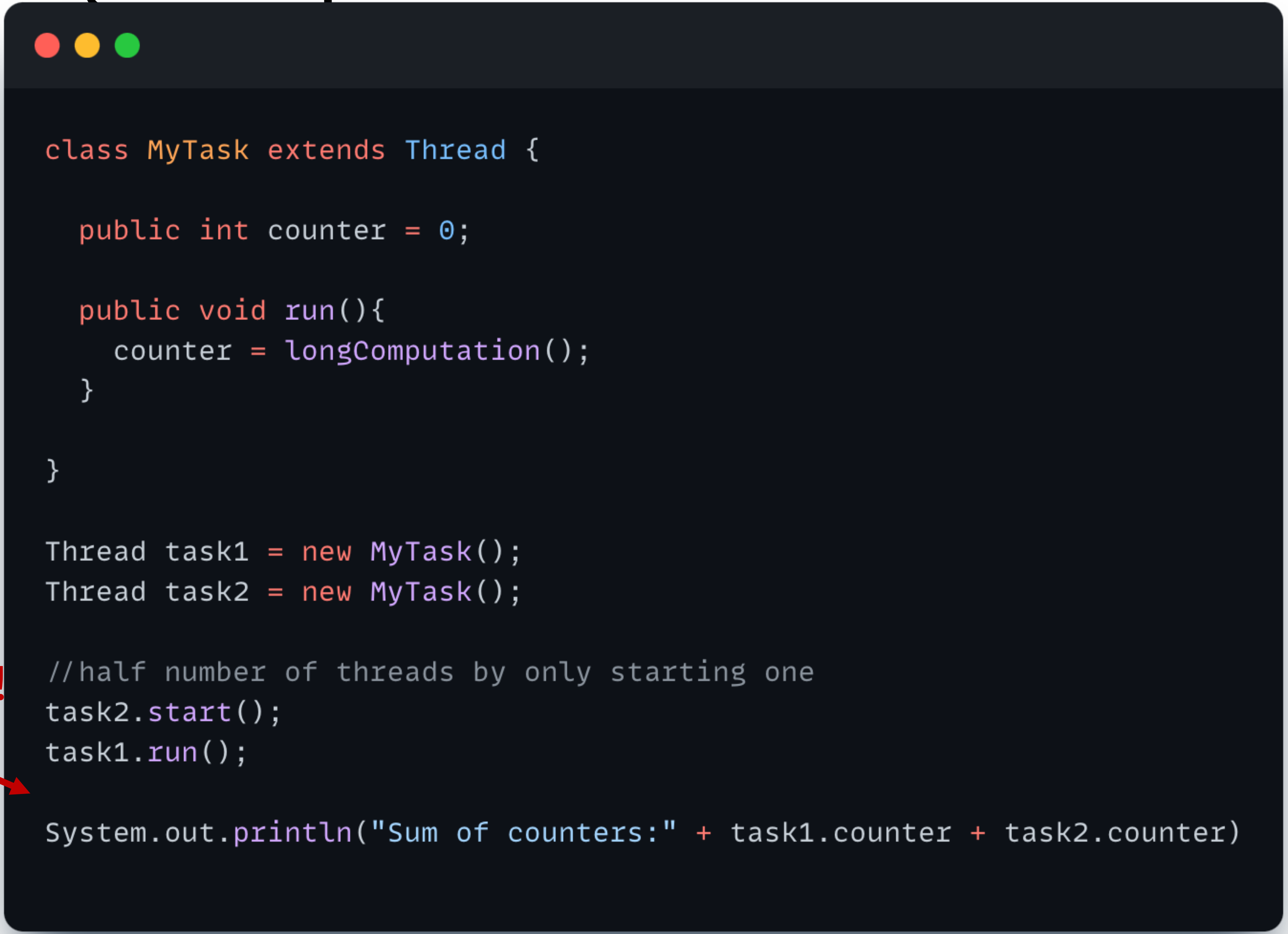
Wrong order!

Thread Quiz: Spot the mistake



```java
class MyTask implements Runnable {...}

Thread thread1 = new Thread(new MyTask());
Thread thread2 = new Thread(new MyTask());

//half number of threads by only starting one
thread2.start();
thread1.run();

//...
```

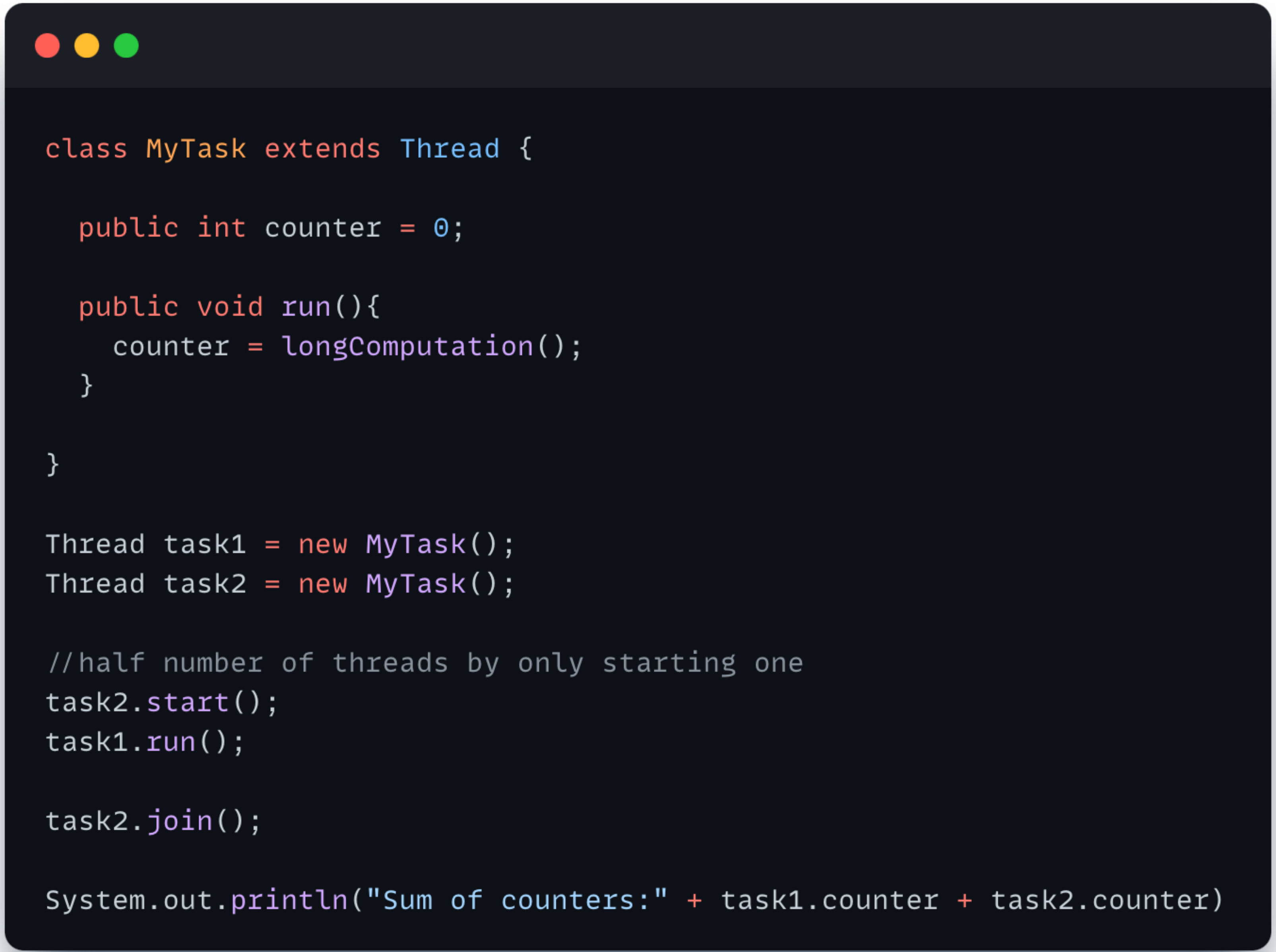
Thread Quiz: Spot the mistake



```java
class MyTask extends Thread {

  public int counter = 0;

  public void run(){
    counter = longComputation();
  }

}

Thread task1 = new MyTask();
Thread task2 = new MyTask();

//half number of threads by only starting one
task2.start();
task1.run();

System.out.println("Sum of counters:" + task1.counter + task2.counter)
```

We need to join!

Thread Quiz: Spot the mistake



```
class MyTask extends Thread {

  public int counter = 0;

  public void run(){
    counter = longComputation();
  }

}

Thread task1 = new MyTask();
Thread task2 = new MyTask();

//half number of threads by only starting one
task2.start();
task1.run();

task2.join();

System.out.println("Sum of counters:" + task1.counter + task2.counter)
```

Thread Safe Counter

```java
public class Counter {
  private int value;
  // returns a unique value

  public int getNext() {
    return value++;
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}


public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

How to implement a thread safe Counter?

Thread Safe Counter

```
public class SyncCounter {
  private int value;

  public synchronized int getNext() {
    return value++;
  }
}
```

```
public class LockCounter {
  private int value;
  private Lock = new ReentrantLock();

  public int getNext() {
    lock.lock();
    try {
      return value++;
    } finally {
      lock.unlock()
    }
  }
}
```

```
public class AtomicCounter {
  private AtomicInteger value;

  public int getNext() {
    return value.incrementAndGet();
  }
}
```

What is the difference between synchronized and a Lock?

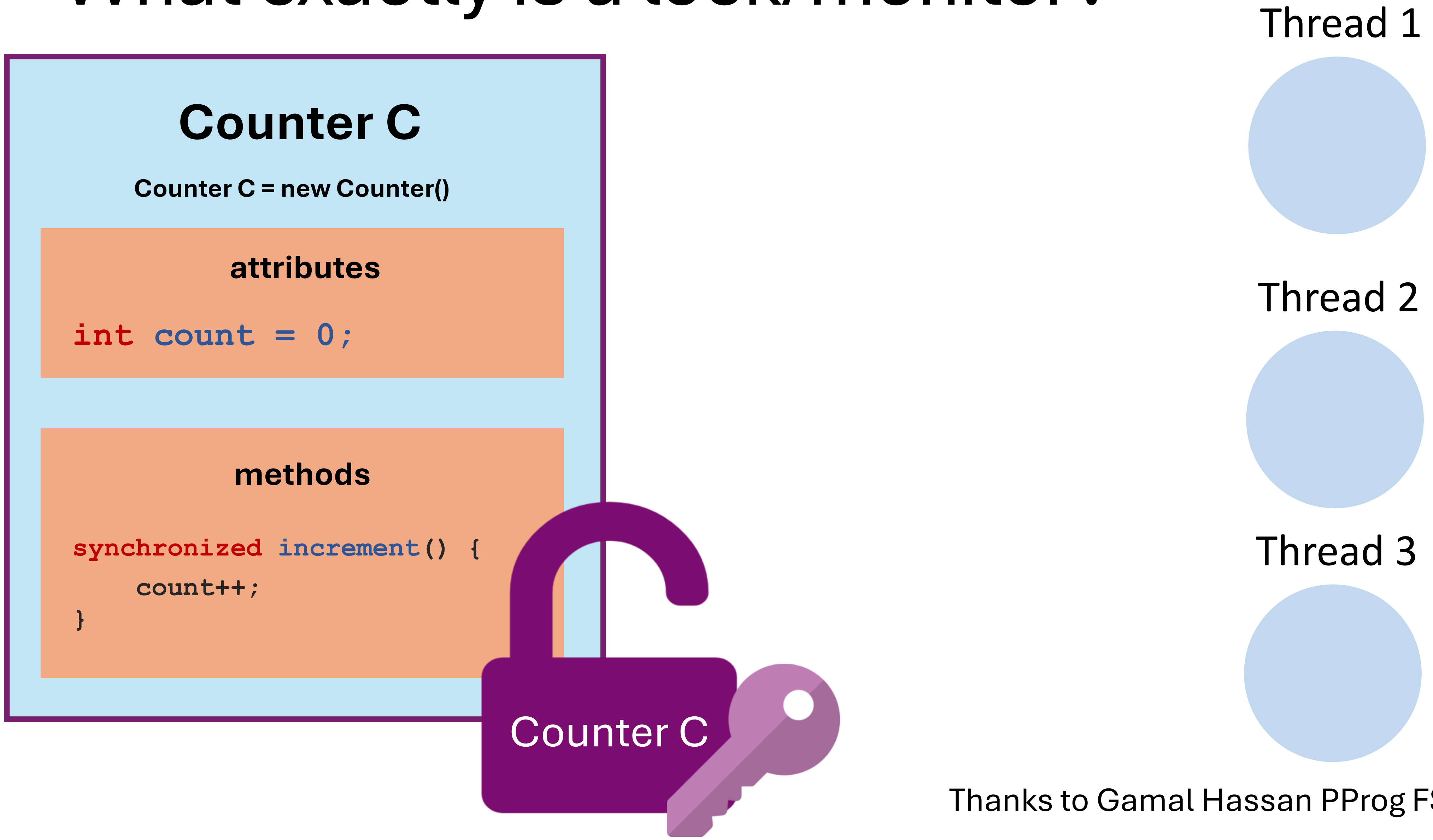
Java: The **synchronized** keyword

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock

Every intrinsic lock has an object (or class) associated with it

A thread that needs exclusive access to an object’s field has to acquire the object’s intrinsic lock before accessing them

What exactly is a lock/monitor?



Counter C

Counter C = new Counter()

attributes

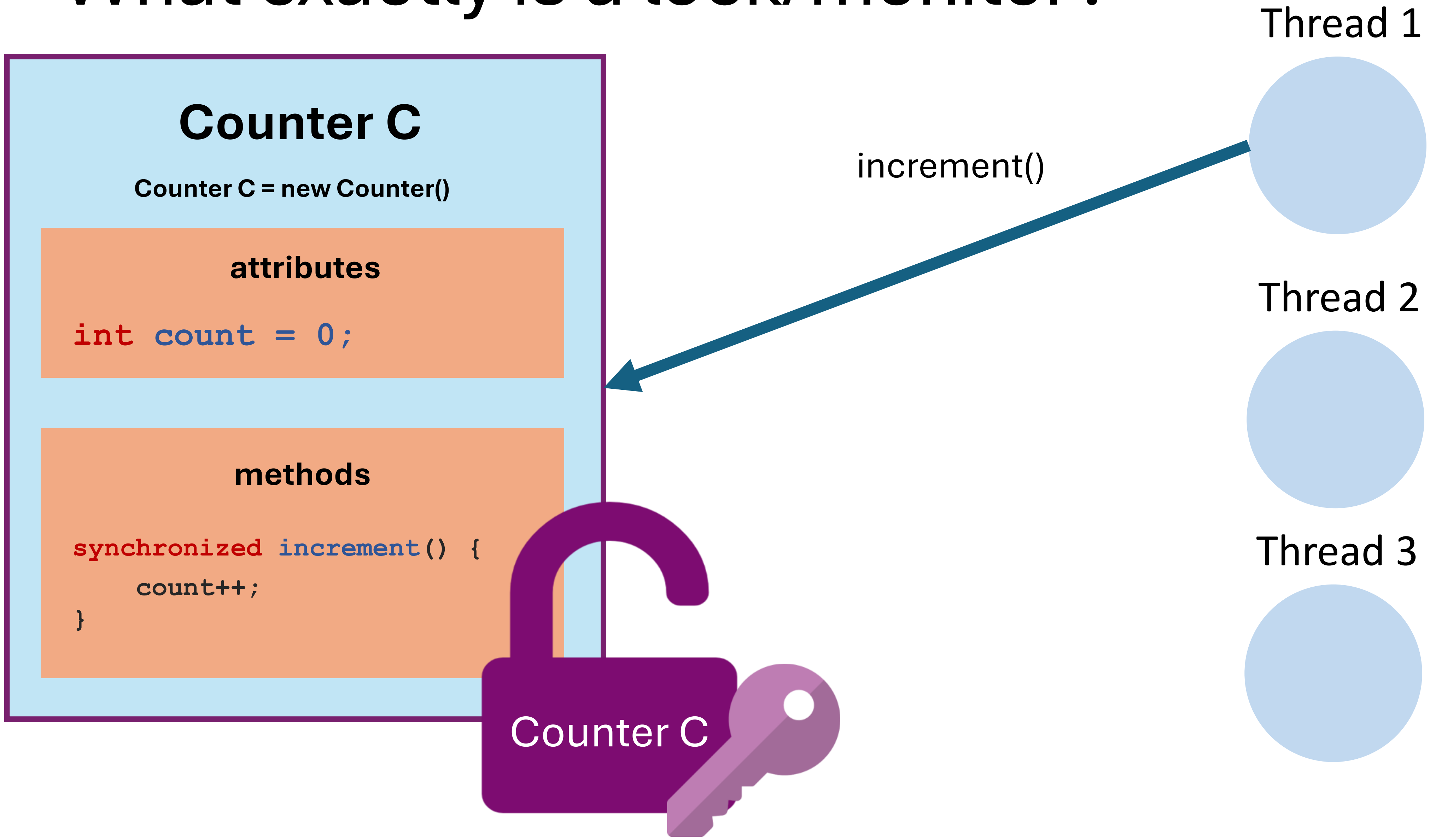
```
int count = 0;
```

methods

```
synchronized increment() {
    count++;
}
```

Thanks to Gamal Hassan PProg FS24!

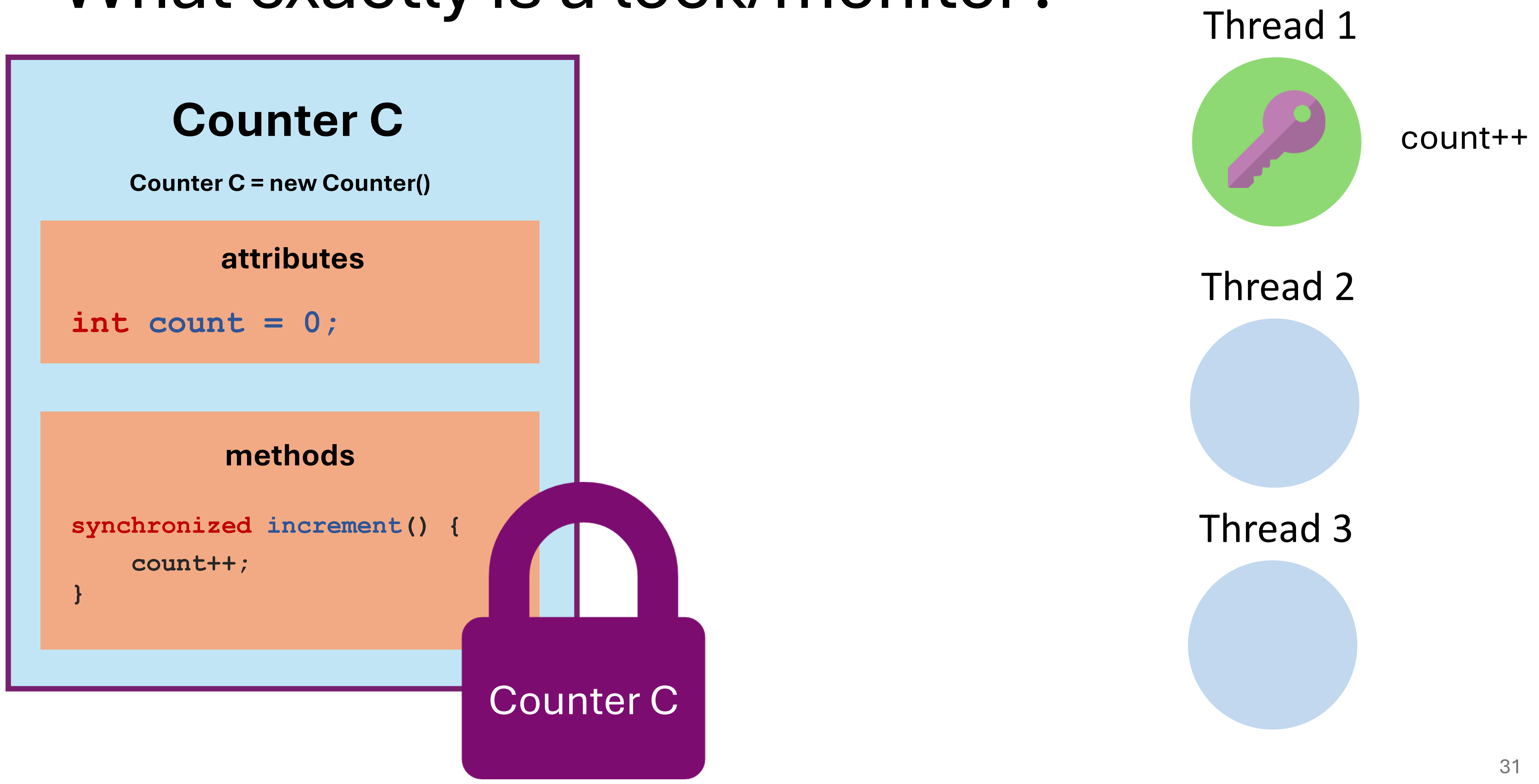
What exactly is a lock/monitor?



```
int count = 0;
```

```
synchronized increment() {
    count++;
}
```

What exactly is a lock/monitor?



Counter C

Counter C = new Counter()

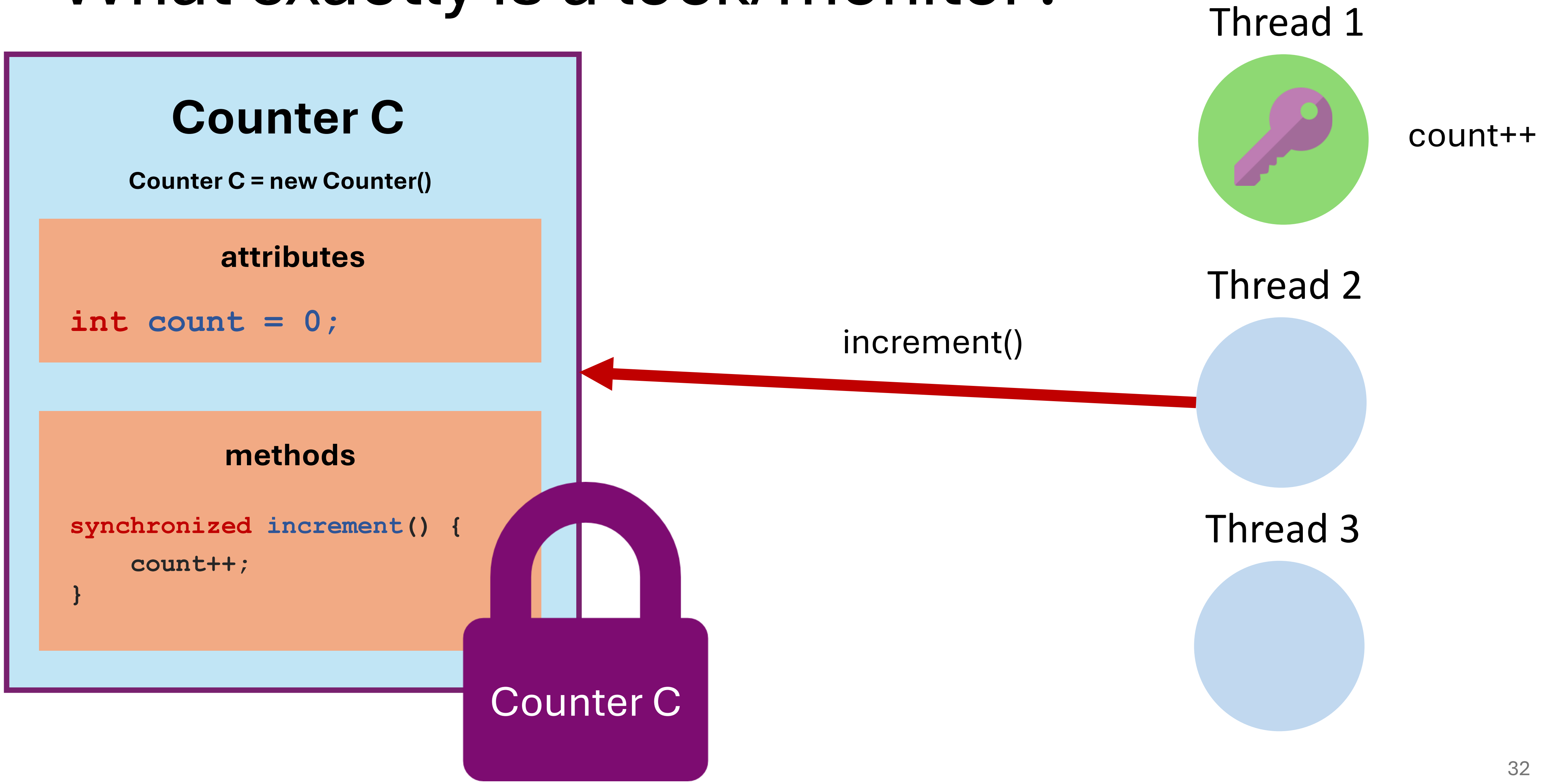
attributes

```
int count = 0;
```

methods

```
synchronized increment() {
    count++;
}
```

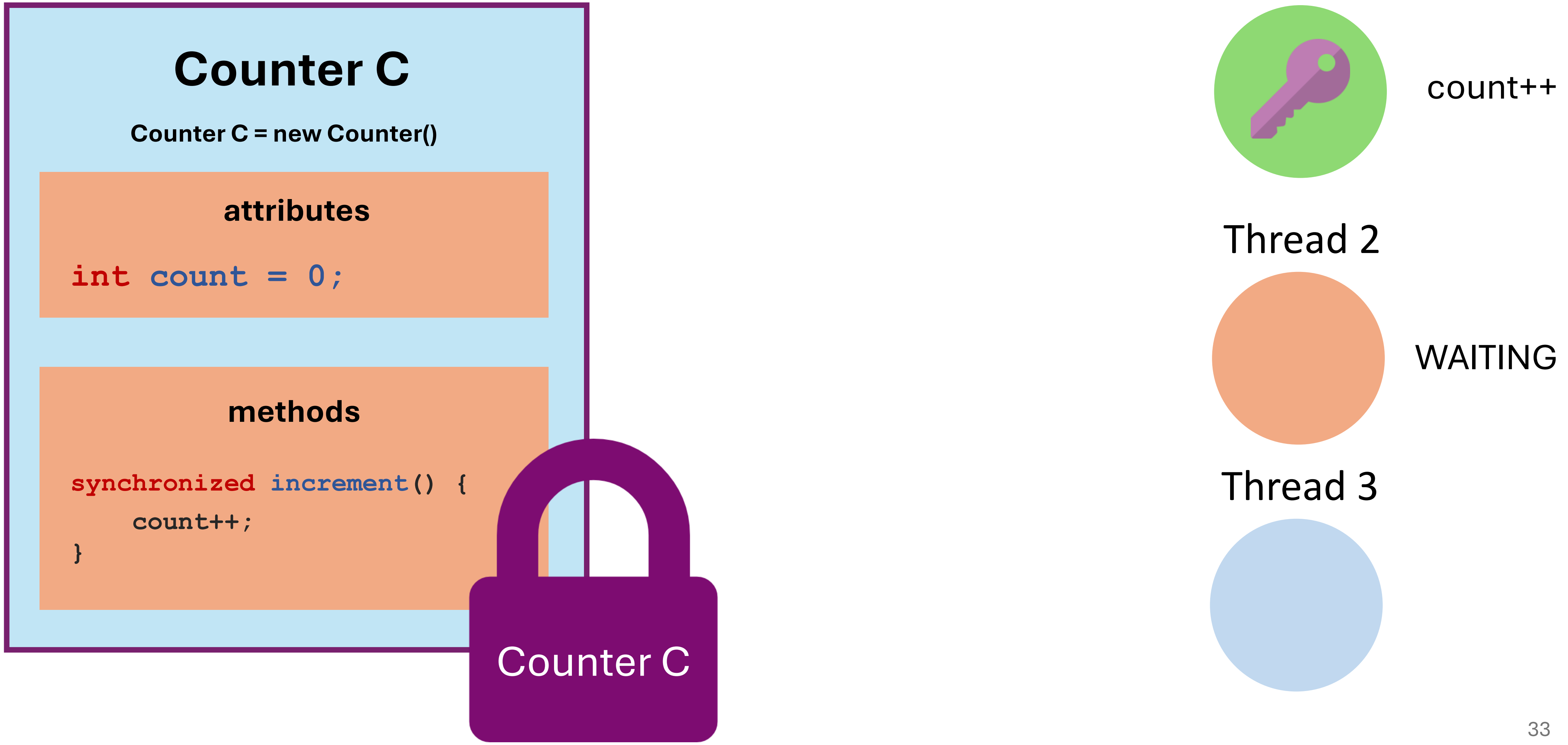
What exactly is a lock/monitor?



```
int count = 0;
```

```
synchronized increment() {
    count++;
}
```

What exactly is a lock/monitor?



```
int count = 0;
```

```
synchronized increment() {
    count++;
}
```

java.util.concurrent.Lock Interface

More low-level primitive than synchronized.

Clients need to implement:

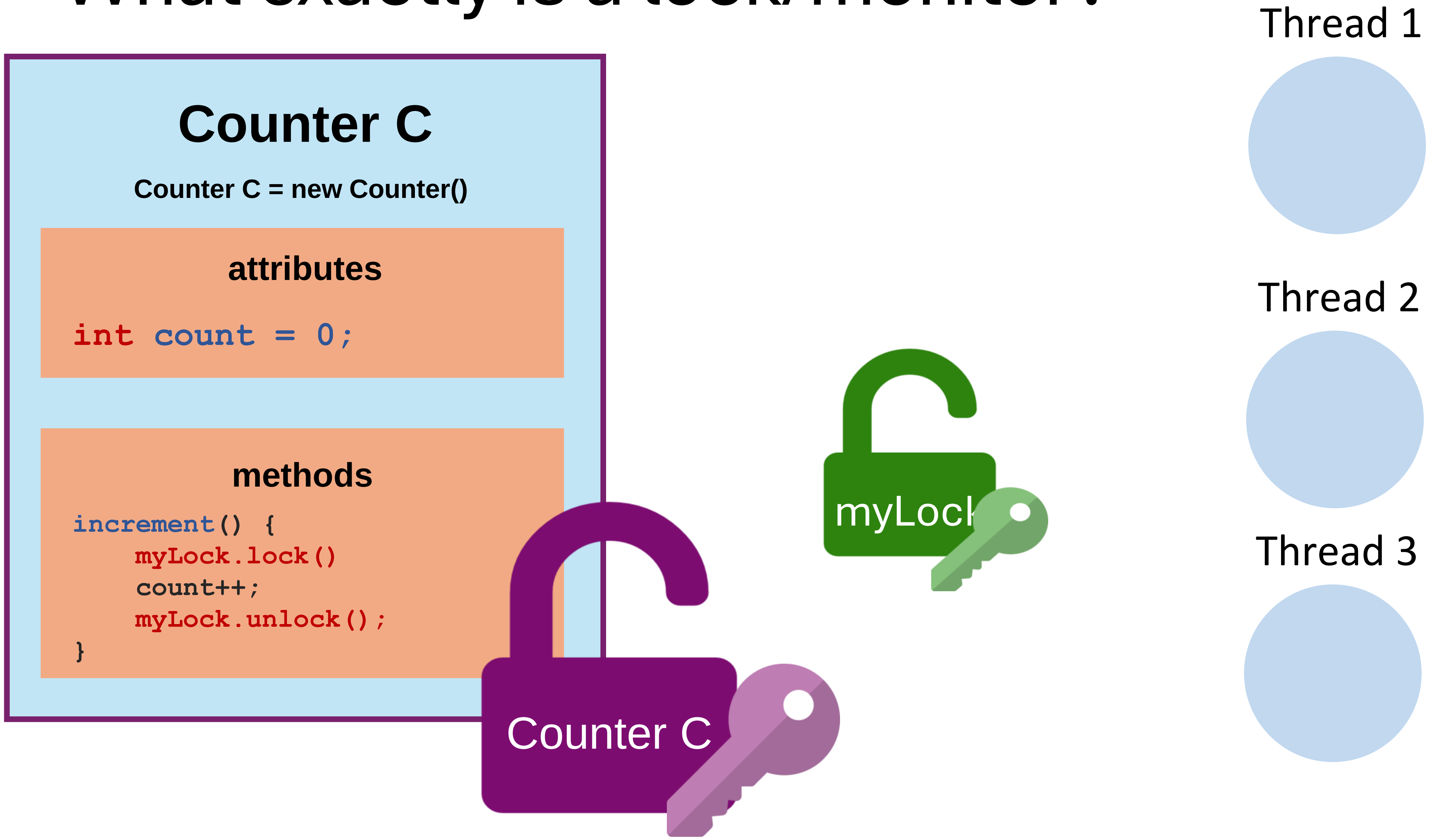
- lock(): Acquires the lock, blocks until it is acquired
- trylock(): Acquire lock only if its lock is free when function is called
- unlock(): Release the lock

Allows more flexible structuring than synchronized blocks

What does it mean to be more flexible?

Why is this useful?

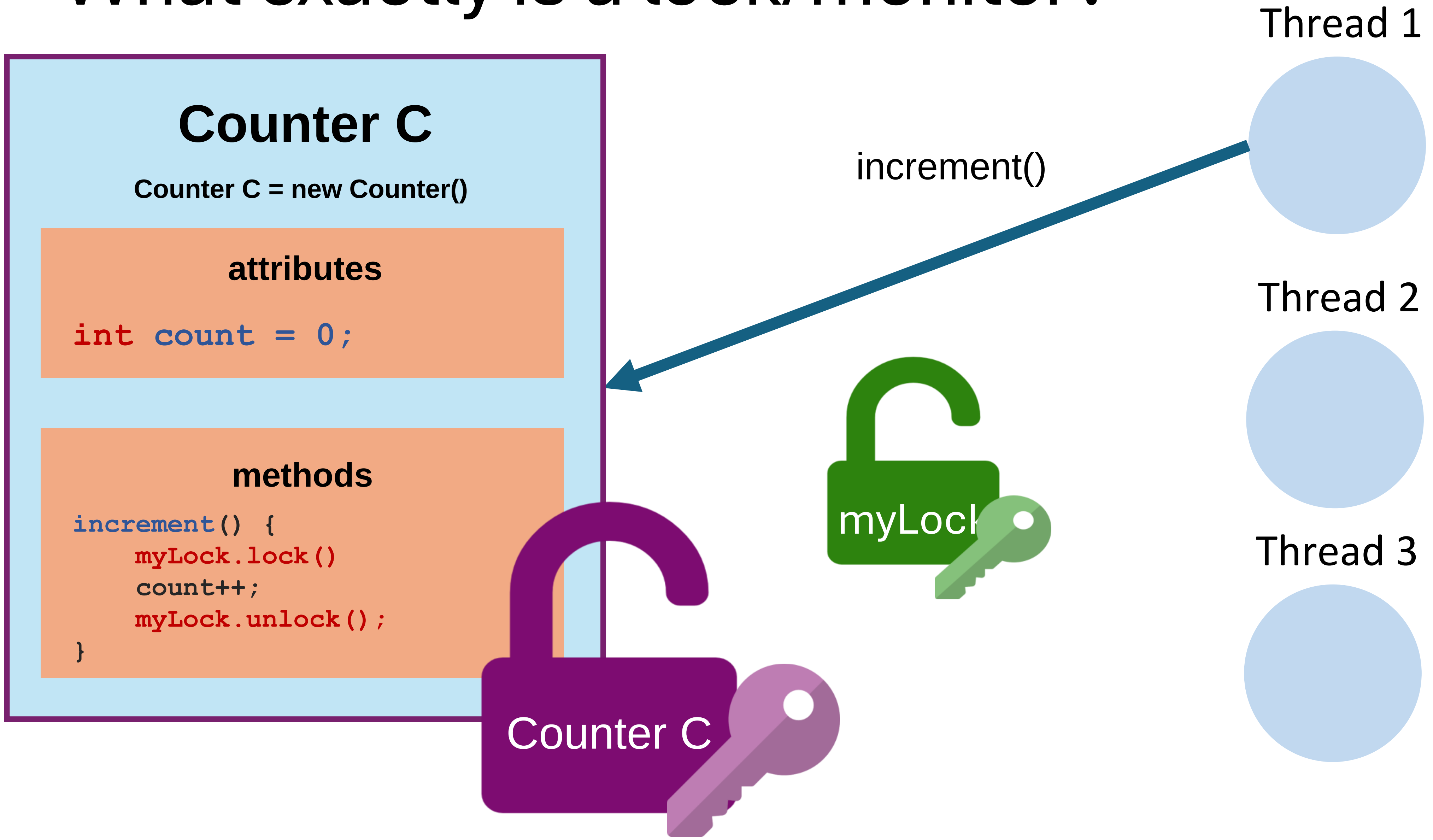
What exactly is a lock/monitor?



```
int count = 0;
```

```
increment() {
    myLock.lock()
    count++;
    myLock.unlock();
}
```

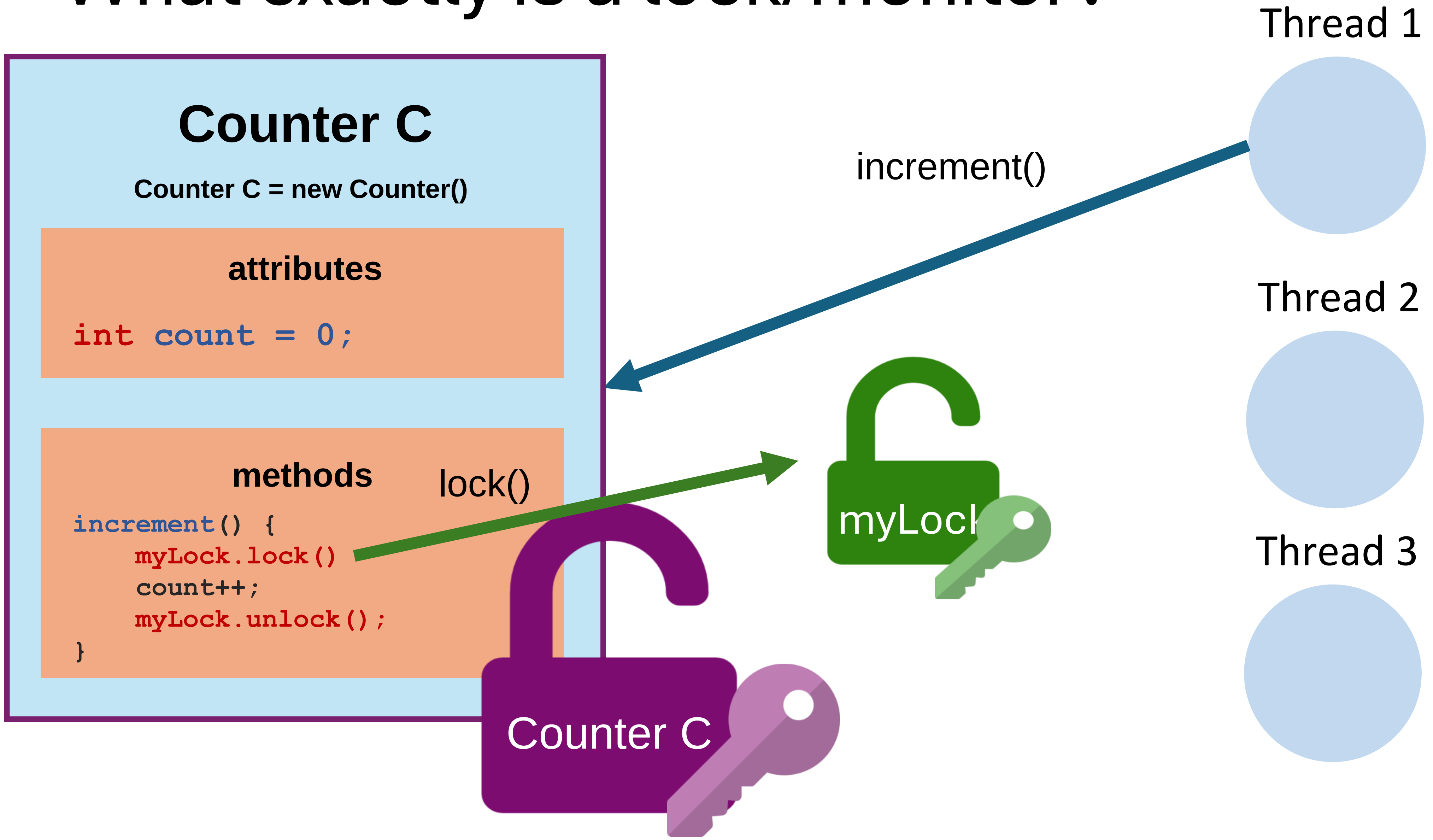
What exactly is a lock/monitor?



```
int count = 0;
```

```
increment() {
    myLock.lock()
    count++;
    myLock.unlock();
}
```

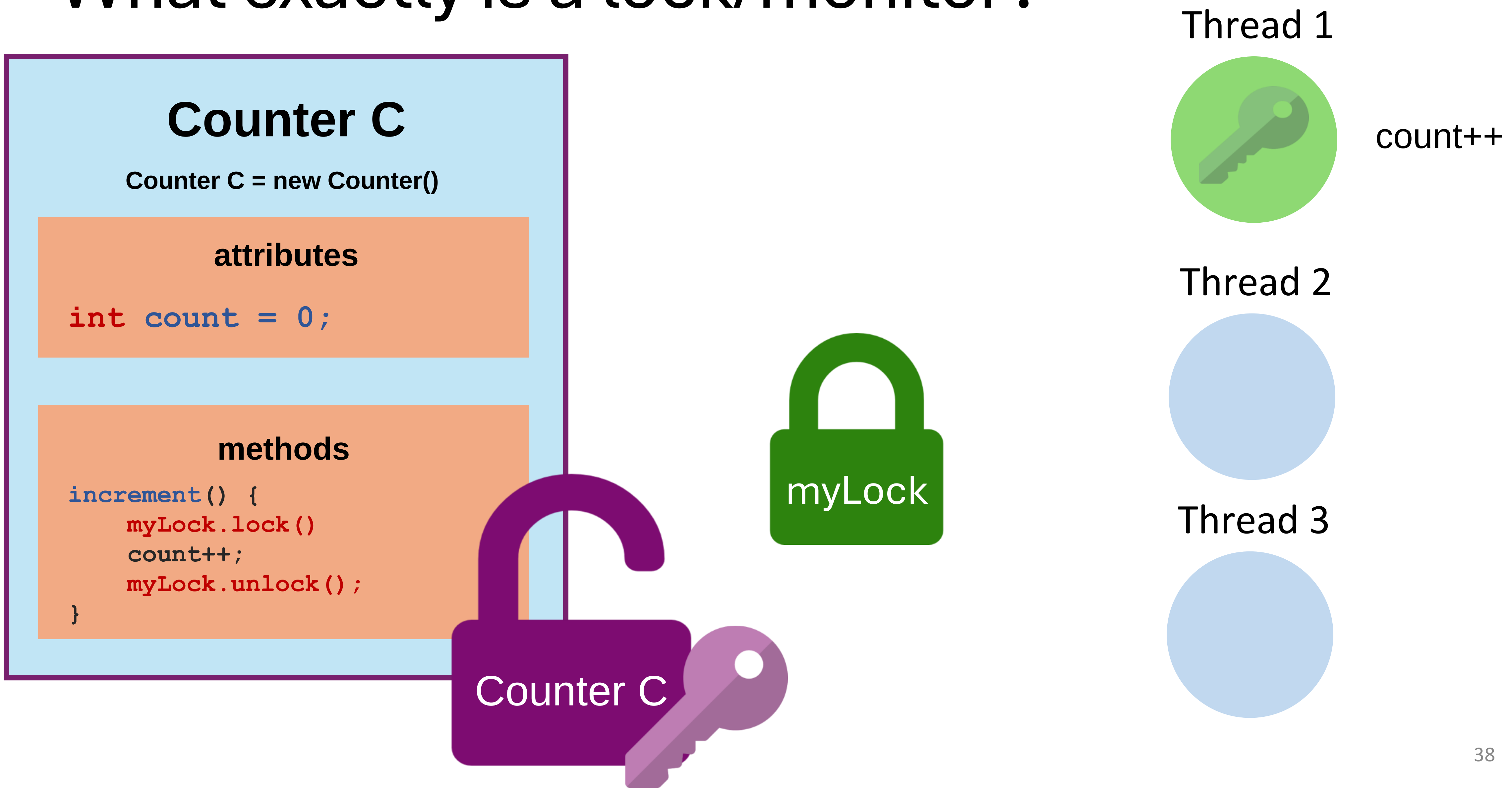
What exactly is a lock/monitor?



```
int count = 0;
```

```
increment() {
    myLock.lock()
    count++;
    myLock.unlock();
}
```

What exactly is a lock/monitor?



```
int count = 0;
```

```
increment() {
    myLock.lock()
    count++;
    myLock.unlock();
}
```

Lock Flexibility

Synchronized forces all lock acquisition and release to occur in a block-structured way

```
synchronized (A) {
  synchronized (B) {
  }
}
```

$\longrightarrow$

```
A.lock();
B.lock();
B.unlock();
A.unlock();
```

The following lock order cannot be expressed using synchronized blocks

```
A.lock();
B.lock();
A.unlock();
B.unlock();
```

Lock Flexibility

Synchronized forces all lock acquisition and release
to occur in a block-structured way

```
synchronized (A) {
  synchronized (B) {
  }
}
```

$\longrightarrow$

```
A.lock();
B.lock();
B.unlock();
A.unlock();
```

The following lock order cannot
be expressed using synchronized blocks

```
A.lock();
B.lock();
A.unlock();
B.unlock();
```

As we will see later in the course, such order is useful
for implementing concurred data structures and
referred
to as “hand-over-hand” locking (or “chain-locking”)

Lock Flexibility

Consider a list of locks that you should acquire

```
public int getNext(List<Lock> locks) {

        // acquire all locks

        // critical section

        // release all locks

}
```

Can this be achieved using synchronized?

Lock Flexibility

Is the Lock acquired?

```
lock.isLocked()
```

Is the Lock acquired by current thread?

```
lock.isHeldByCurrentThread()
```

Try acquire the Lock without blocking

```
lock.tryLock()
```

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html details

Implementing Classes of java.util.concurrent.Lock

ReentrantLock
ReentrantReadWriteLock.ReadLock
ReentrantReadWriteLock.WriteLock

Readers/Writers Lock will be covered in detail in 3 weeks

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/Lock.html

Basic Synchronization Rule

Access to **shared** and **mutable state** needs to be **always protected**!

Synchronization Issues

Data Race: ??

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Deadlock: ??

Synchronization Issues

Data Race: A program has a data race if, during any possible execution, a memory location could be written from one thread, while concurrently being read or written from another thread.

Deadlock: Circular waiting/blocking (no instructions are executed and CPU time may be used) between threads, so that the system (union of all threads) cannot make any progress anymore.

Quiz: What is wrong with this code?

```
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

Quiz: What is wrong with this code?

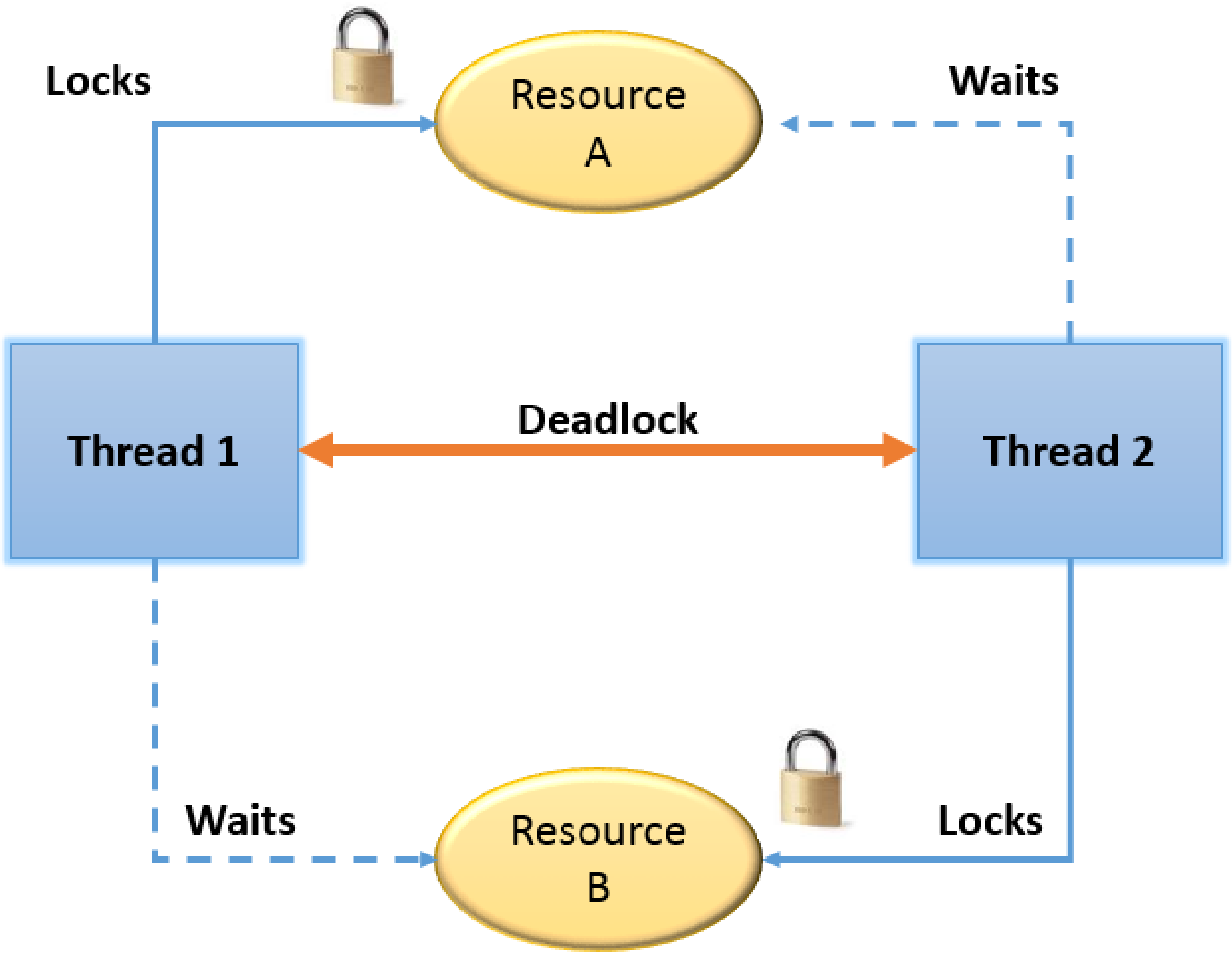
```java
void exchangeSecret(Person a, Person b) {
    a.getLock().lock();
    b.getLock().lock();
    Secret s = a.getSecret();
    b.setSecret(s);
    a.getLock().unlock();
    b.getLock().unlock()
}
```

```java
public class Person {
  private ReentrantLock mLock = new ReentrantLock();
  private String mName;

  public ReentrantLock getLock() {
    return mLock;
  }

  ...
}
```

```
Thread 1:
exchangeSecret(p1, p2)
```

Deadlock

```
Thread 2:
exchangeSecret(p2, p1)
```


Locks
Resource A
Waits
Thread 1
Deadlock
Thread 2
Waits
Resource B
Locks

Possible solution

```
void exchangeSecret(Person a, Person b) {
        ReentrantLock first, second;
        if (a.getName().compareTo(b.getName()) < 0) {
                first = a.getLock(); second = b.getLock();
        } else if (a.getName().compareTo(b.getName()) > 0) {
                first = b.getLock(); second = a.getLock();
        } else { throw new UnsupportedOperationException(); }
        first.lock();
        second.lock();
        Secret s = a.getSecret();
        b.setSecret(s);
        first.unlock();
        second.unlock();
}
```

Always acquire and release the Locks in the same order

Deadlocks and Race conditions

Not easy to spot

Hard to debug

- ➔ Might happen only very rarely
- ➔ Testing usually not good enough

  Reasoning about code is required

Lesson learned: Need to be careful when programming with locks

Wait and Notify Recap

Object (lock) provides `wait` and `notify` methods

(any object is a lock)

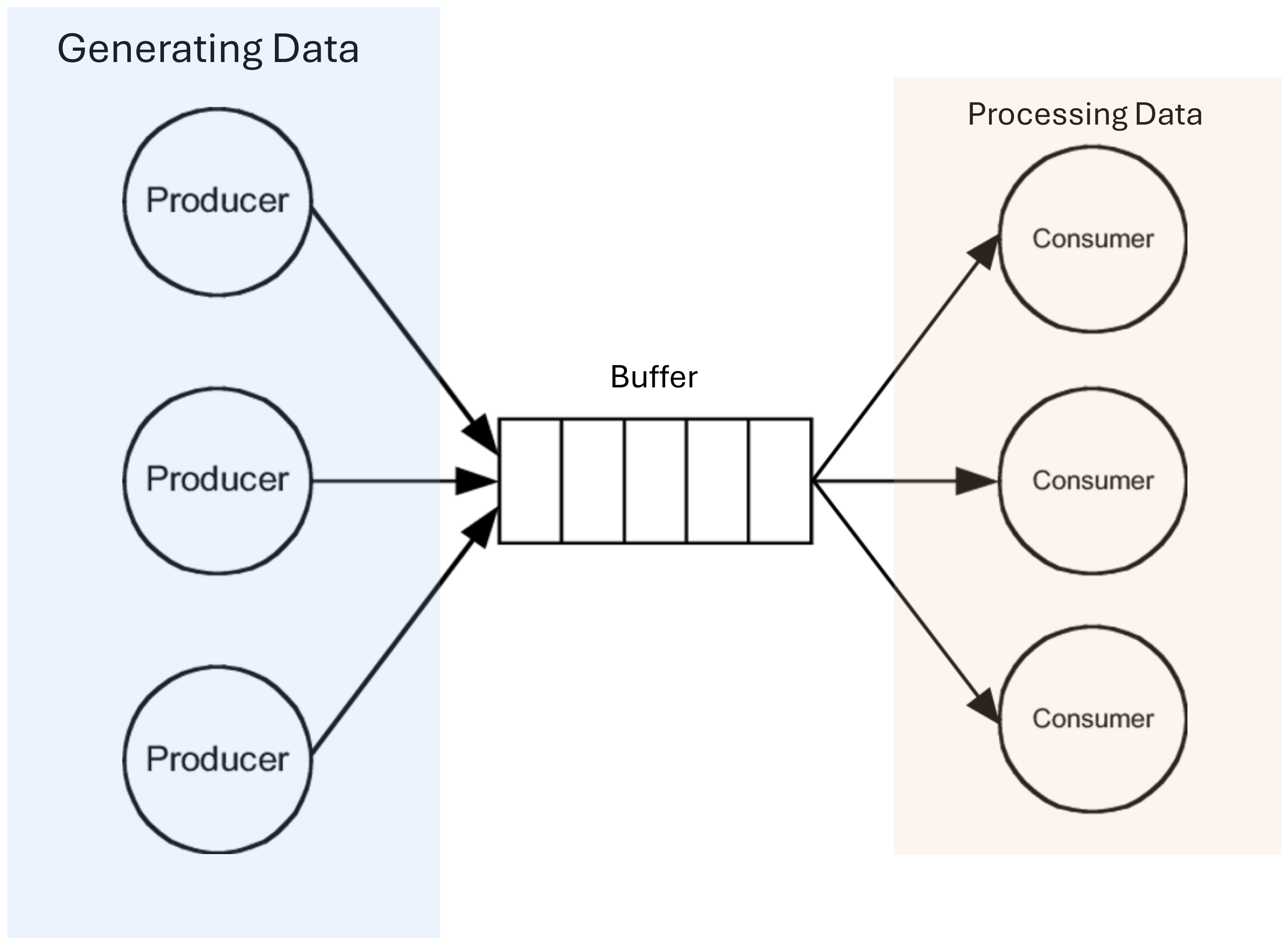
`wait`: Thread must own object's lock to call `wait`
 thread releases lock and is added to "waiting list" for that object
 thread waits until `notify` is called on the object

`notify`: Thread must own object's lock to call `notify`

`notify`: Wake one **(arbitrary)** thread from object's "waiting list"

`notifyAll`: Wake all threads

Producer-Consumer Problem



How to use wait/notify

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty())
          buffer.wait();
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
        buffer.notifyAll();
      }
    }
  }
}
```

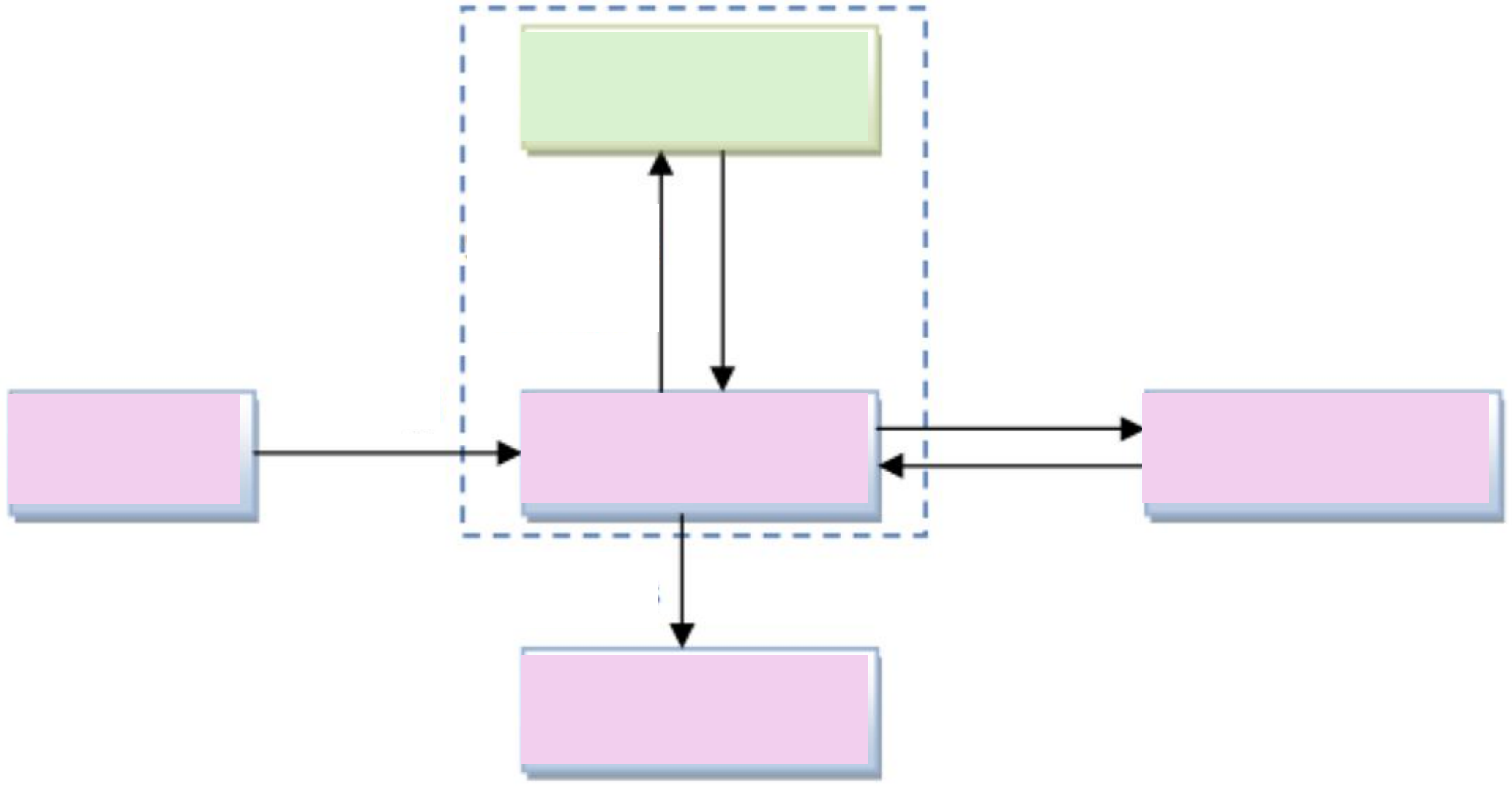
`buffer.wait():`

1. Consumer thread goes to sleep (status NOT RUNNABLE) …
2. … and gives up buffer’s lock

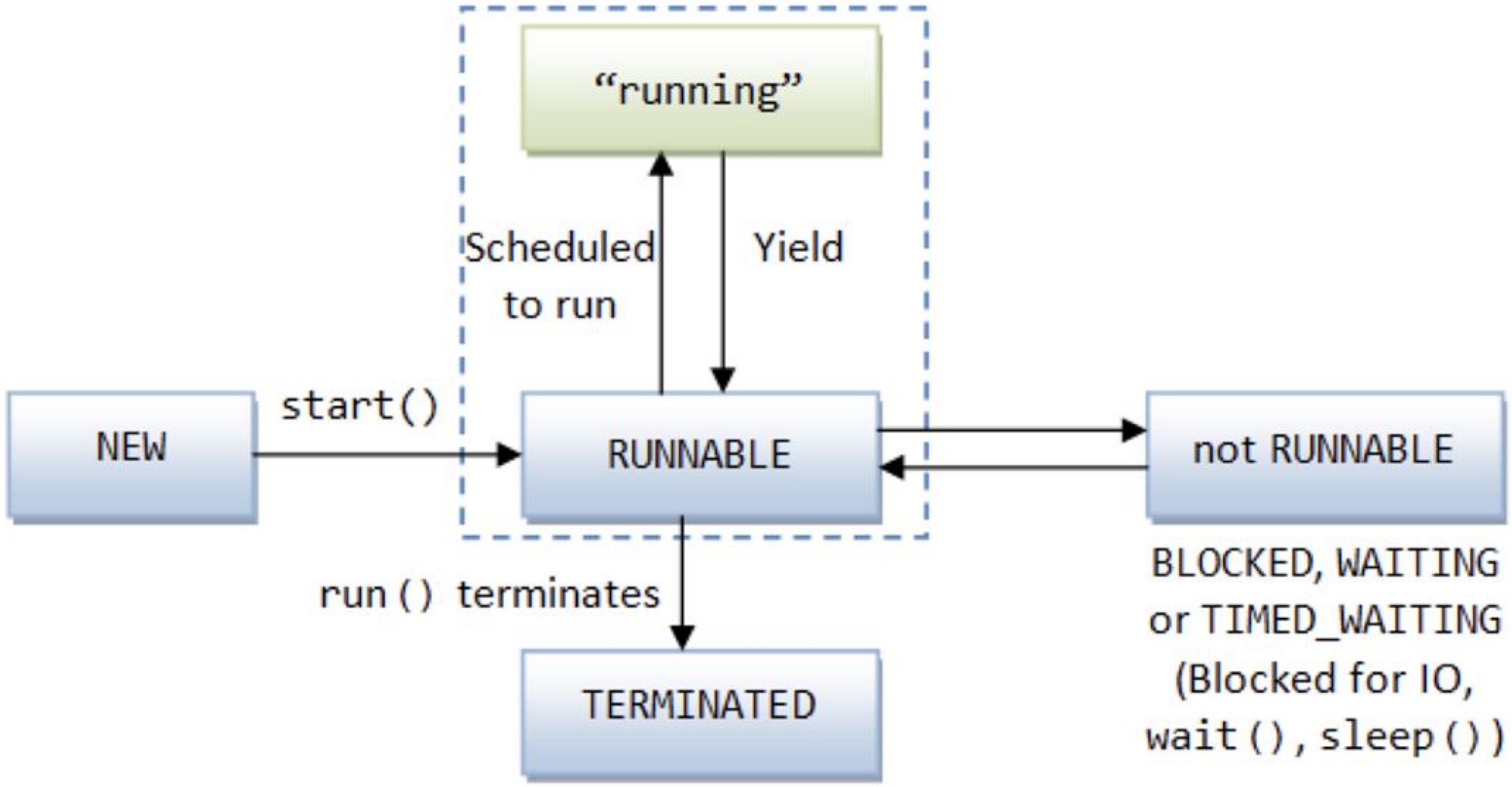
`buffer.notifyAll():`

1. All threads waiting for buffer’s lock are woken up (status RUNNABLE)

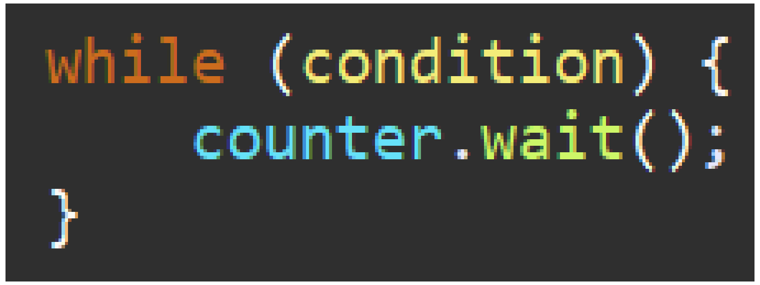
Remember? Thread State Model



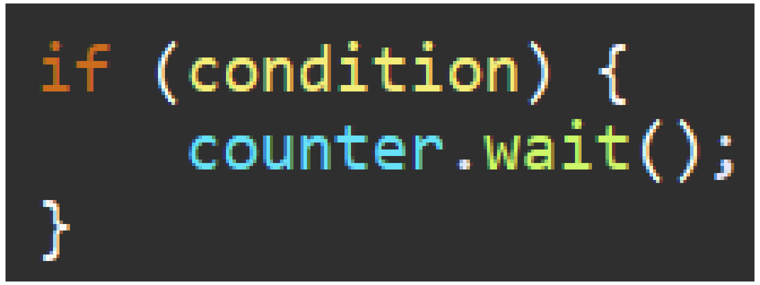
Remember? Thread State Model



Wait and Notify Recap



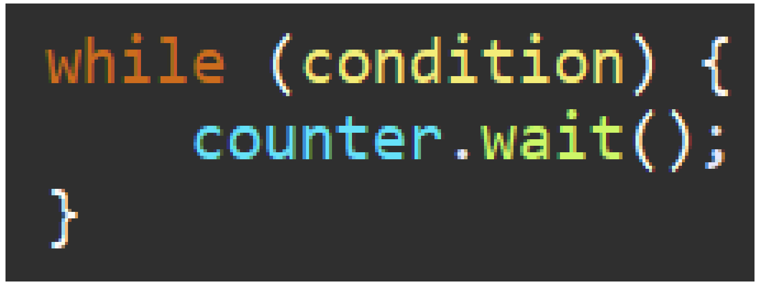
```
while (condition) {
    counter.wait();
}
```



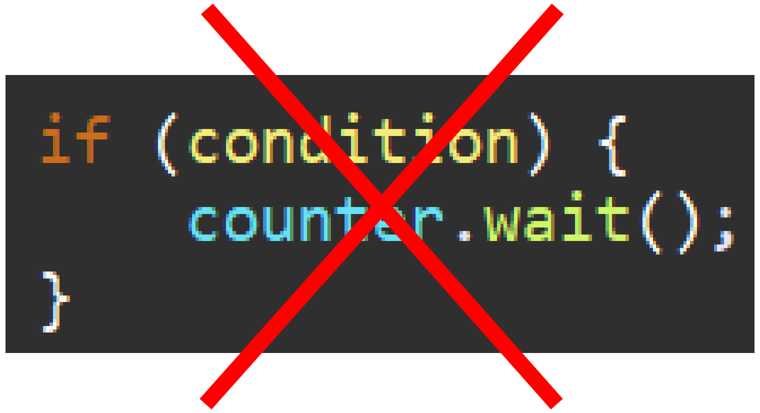
```
if (condition) {
    counter.wait();
}
```

What is the difference? Issues?

Wait and Notify Recap



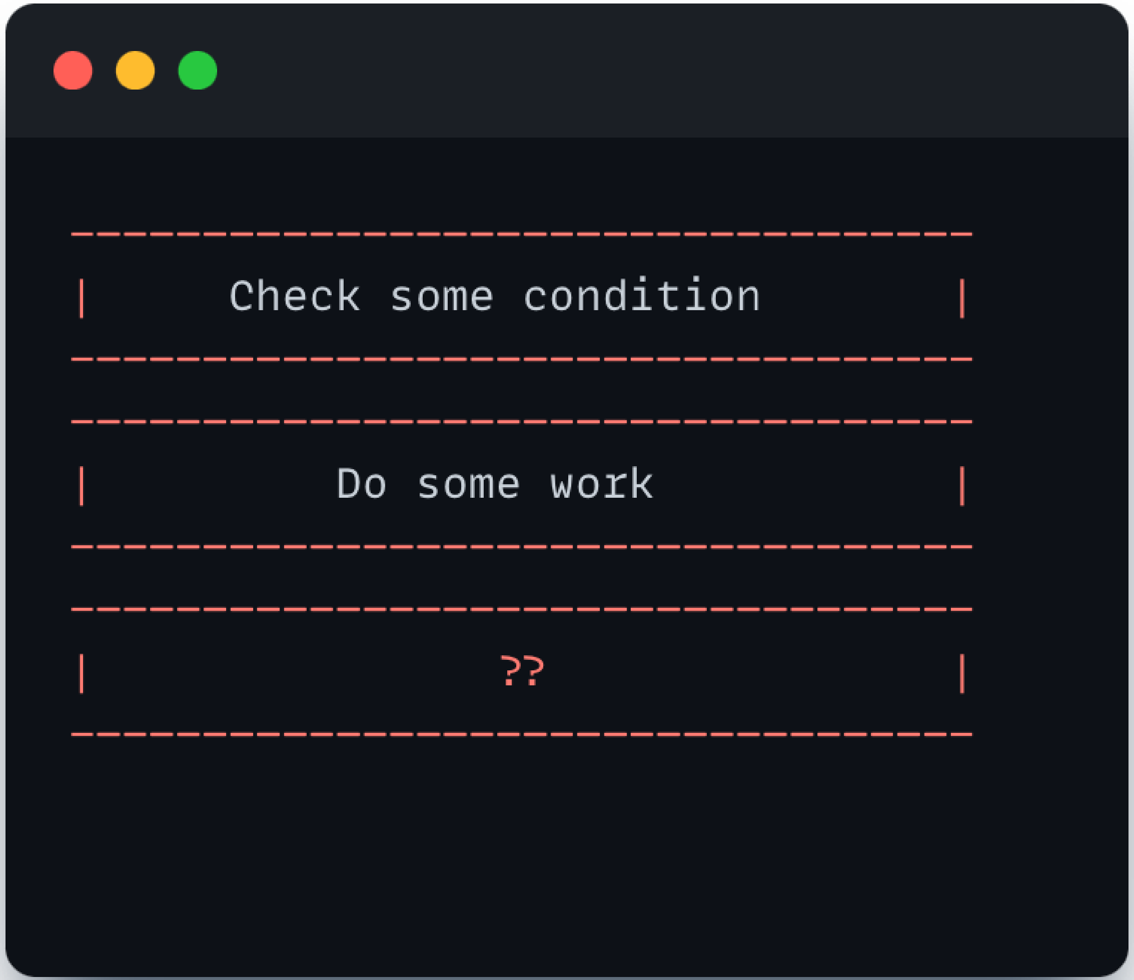
```
while (condition) {
    counter.wait();
}
```



```
if (condition) {
    counter.wait();
}
```

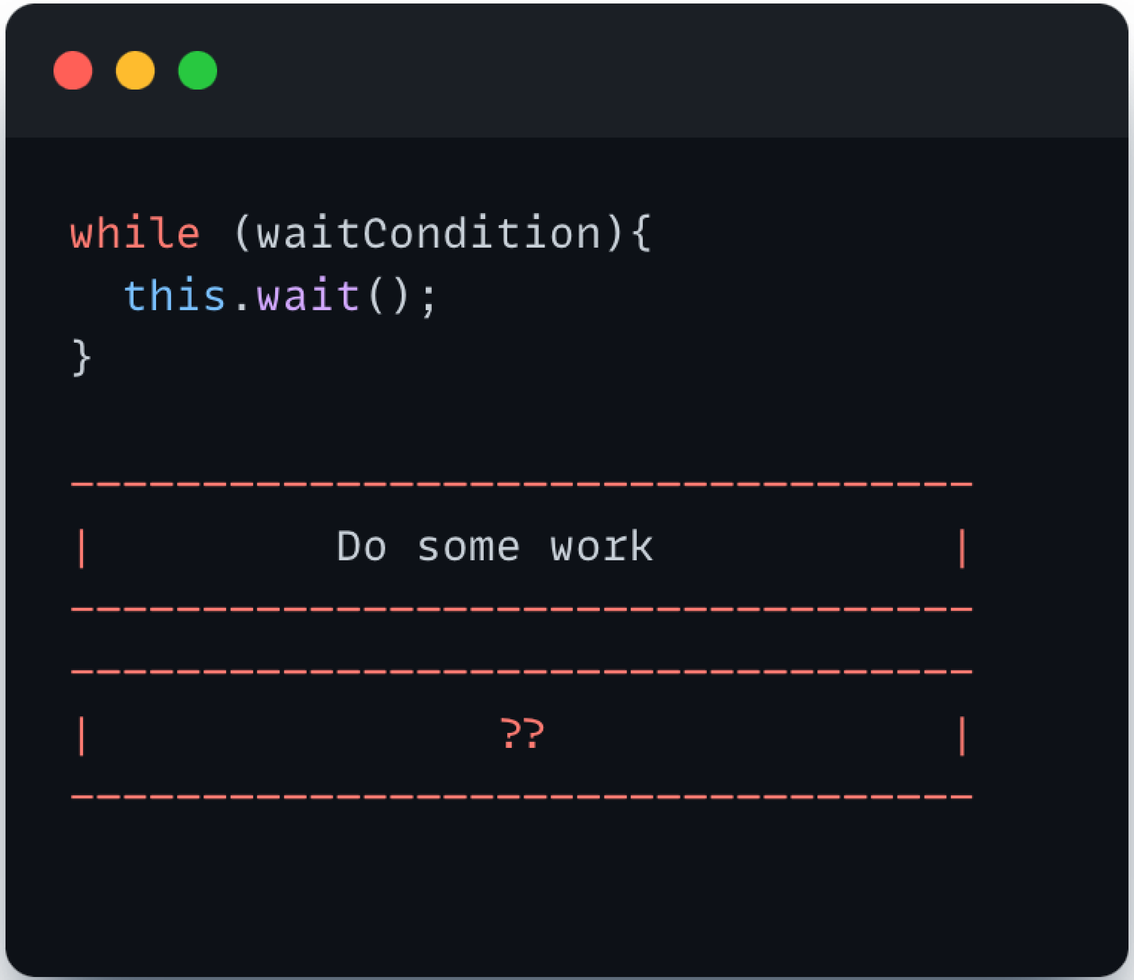
Spurious wake-ups and notifyAll()
→ `wait` has to be in a `while` loop

Wait/notify "Template"



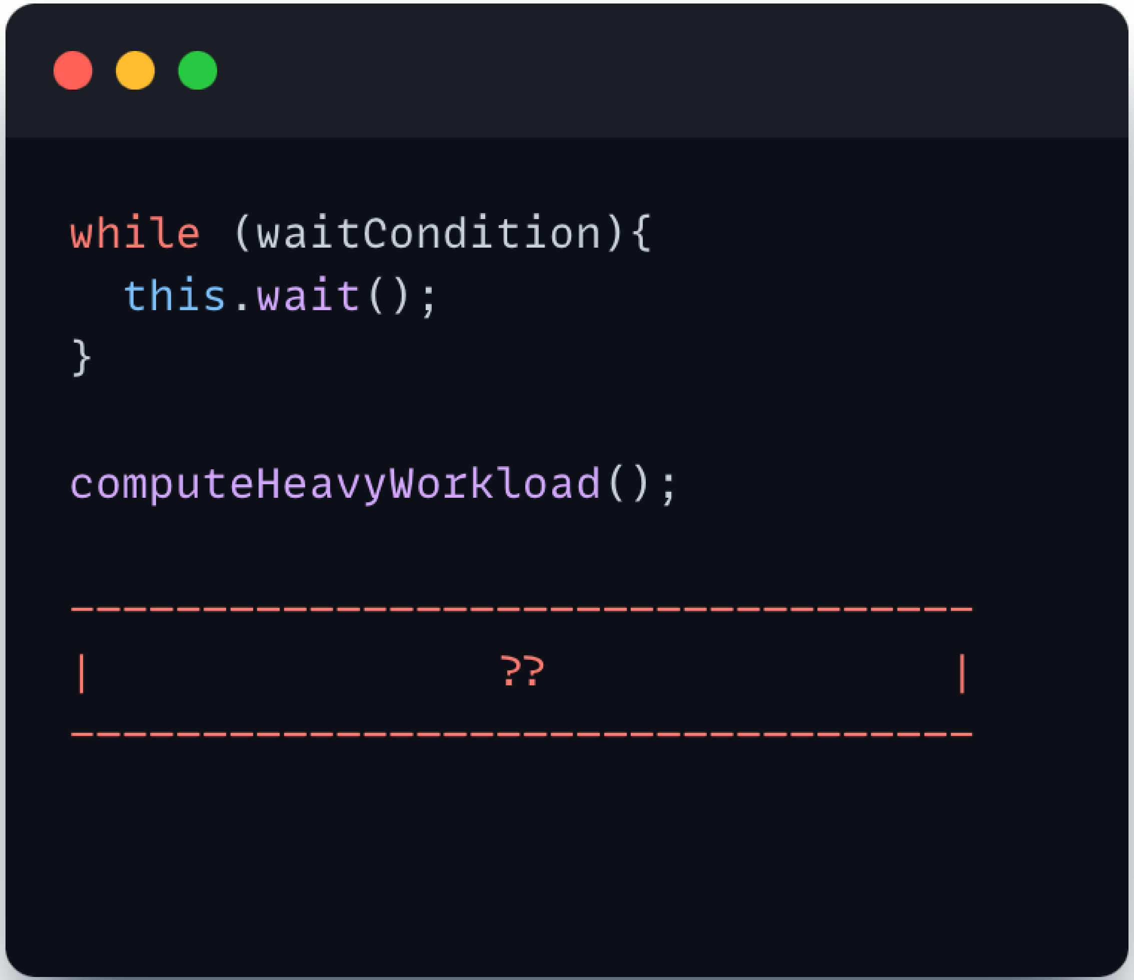
```
----------------------------------
|     Check some condition       |
----------------------------------
----------------------------------
|          Do some work          |
----------------------------------
----------------------------------
|               ??               |
----------------------------------
```

Wait/notify “Template”



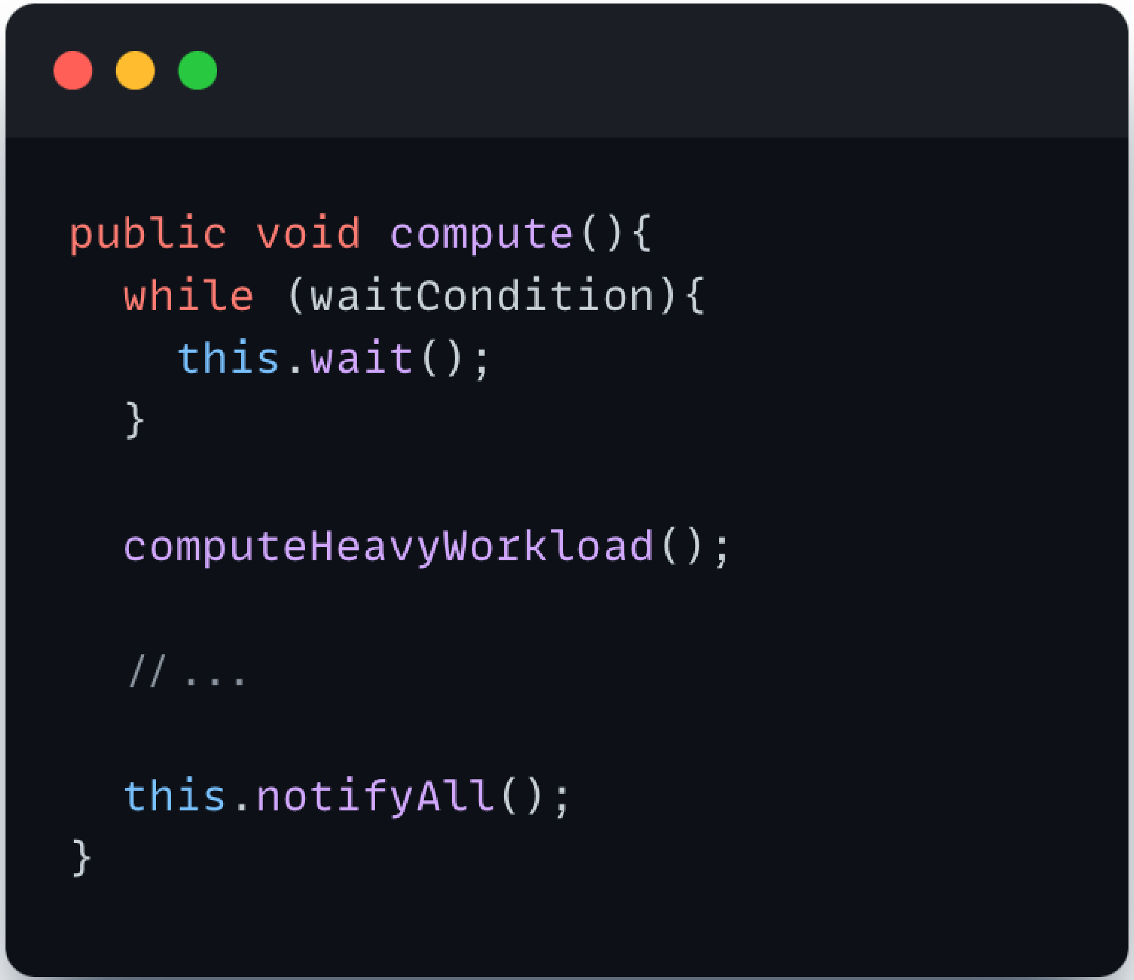
```
while (waitCondition){
  this.wait();
}

----------------------------------
|          Do some work          |
----------------------------------
----------------------------------
|               ??               |
----------------------------------
```

Wait/notify “Template”



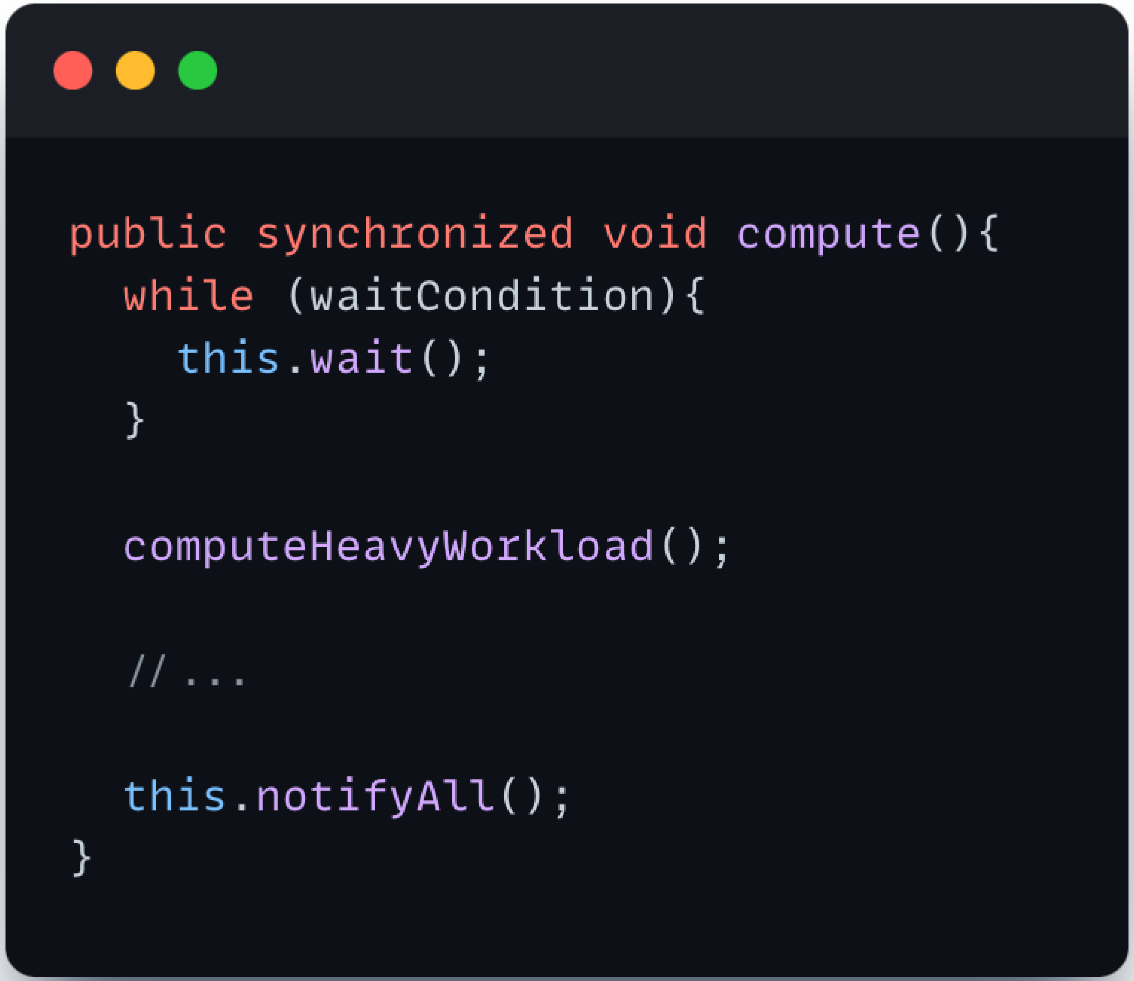
```
while (waitCondition){
  this.wait();
}

computeHeavyWorkload();

----------------------------------
|               ??               |
----------------------------------
```

Is that all?



```
public void compute(){
  while (waitCondition){
    this.wait();
  }

  computeHeavyWorkload();

  // ...

  this.notifyAll();
}
```

No, it must be inside a synchronized block!



```java
public synchronized void compute(){
  while (waitCondition){
    this.wait();
  }

  computeHeavyWorkload();

  // ...

  this.notifyAll();
}
```

Pipelining: Main Concepts Recap

- **Latency**

- **Throughput**
-

- **Balanced/Unbalanced Pipeline**

Pipelining: Main Concepts Recap

- **Latency**
  time needed to perform a given computation
  (e.g., process a customer)
- **Throughput**
-
- **Balanced/Unbalanced Pipeline**

Pipelining: Main Concepts Recap

- **Latency**

time needed to perform a given computation
(e.g., process a customer)

- **Throughput**

- amount of work that can be done by a system in a given period of time
(e.g., how many customers can be processed in one minute)

- **Balanced/Unbalanced Pipeline**

Pipelining: Main Concepts Recap

- **Latency**

time needed to perform a given computation
(e.g., process a customer)

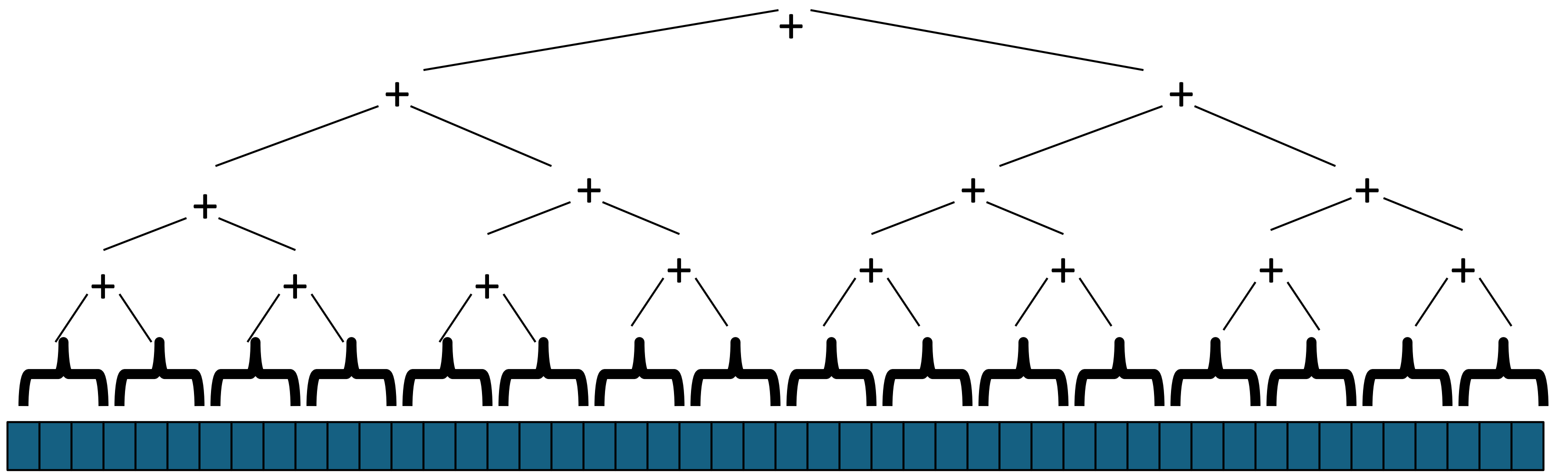
- **Throughput**

- amount of work that can be done by a system in a given period of time
(e.g., how many customers can be processed in one minute)

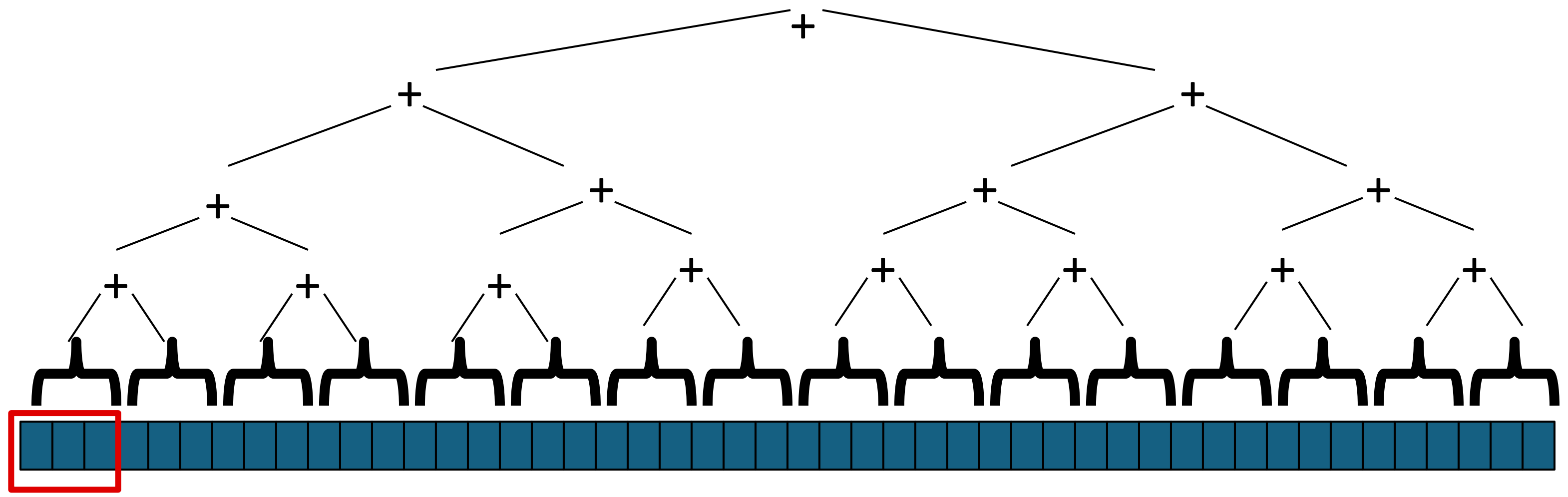
- **Balanced/Unbalanced Pipeline**

- a pipeline is balanced if each stage takes the same length of time

Divide and Conquer



Divide and Conquer

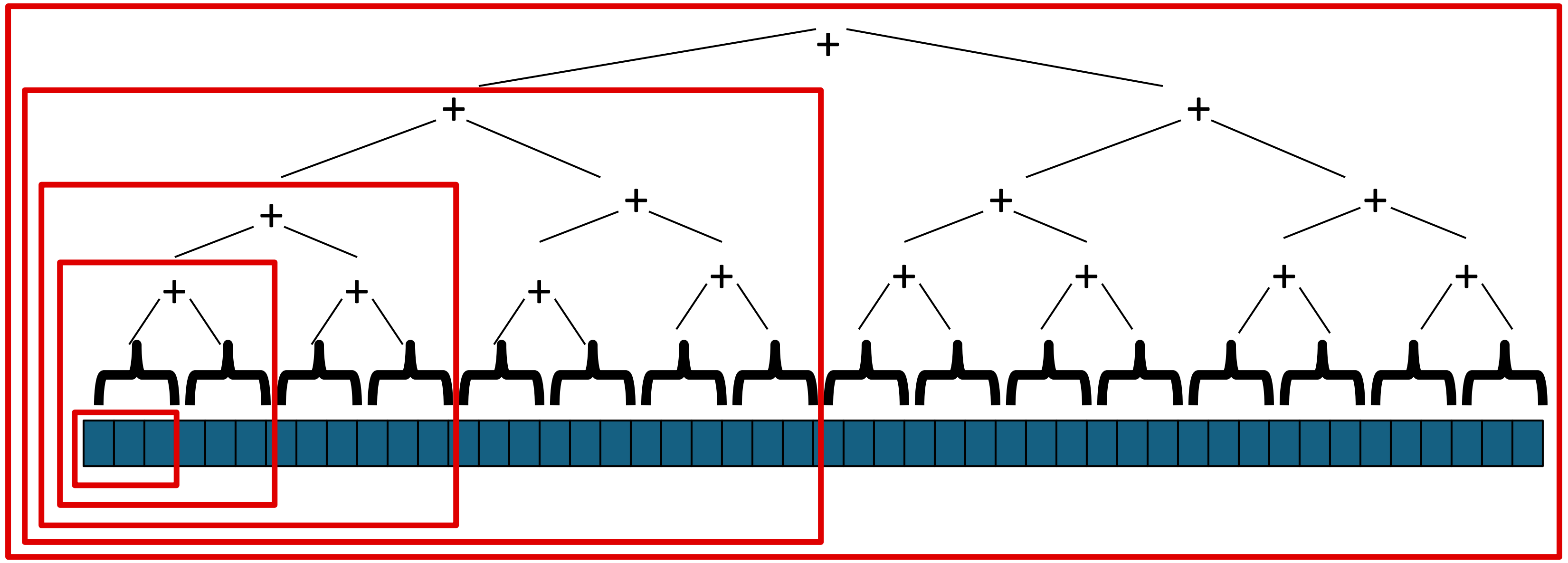


base case

no further split

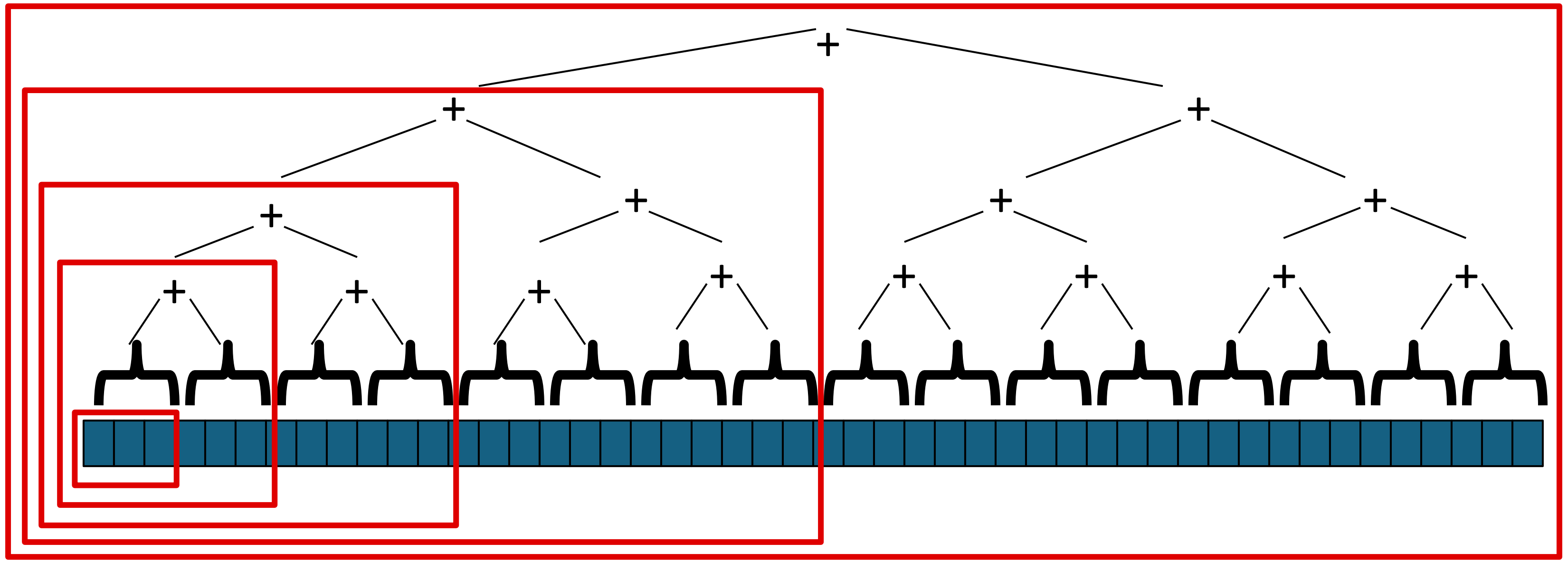
Divide and Conquer

Tasks at different
levels of granularity



Divide and Conquer

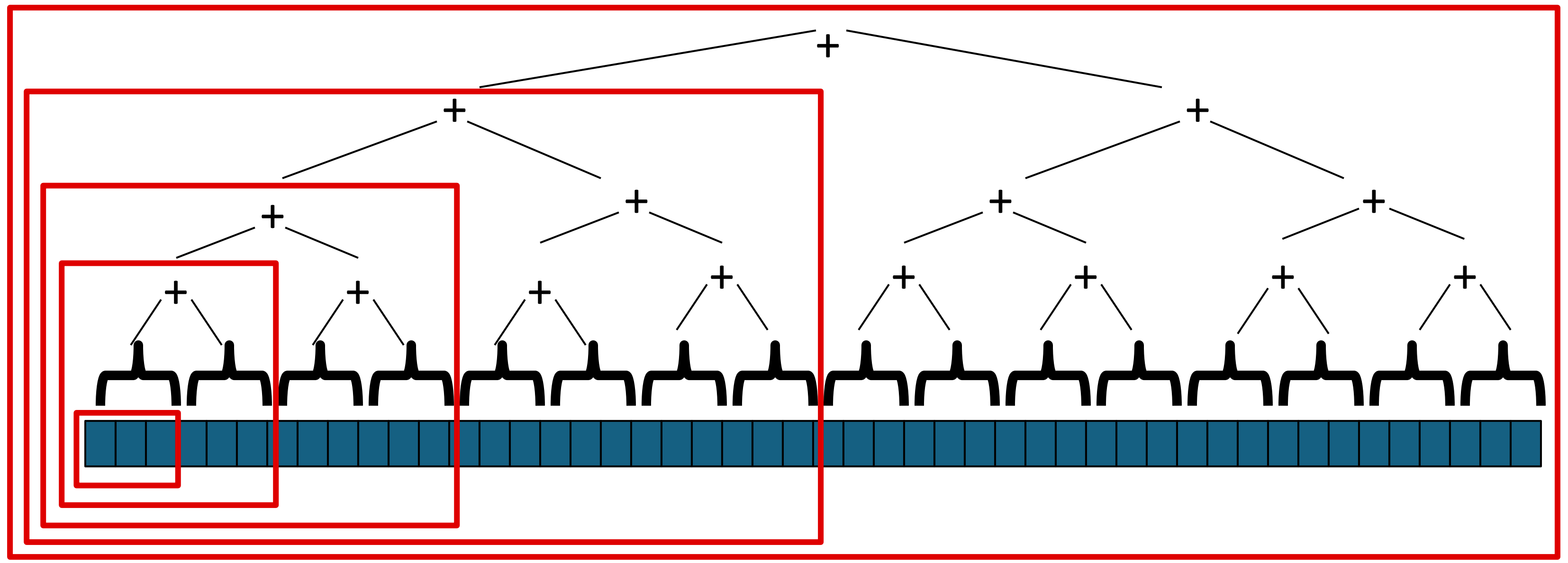
Tasks at different
levels of granularity



What determines a task?

Divide and Conquer

Tasks at different
levels of granularity



What determines a task?

i) input array ii) start index iii) length/end index

These are fields we want to store in the task

Will this work?

```
public void run(){
    int size = h-l;
    if (size == 1) {
           result = xs[l];
           return;
    }
    int mid = size / 2;
    SumThread t1 = new SumThread(xs, l, l + mid);
    SumThread t2 = new SumThread(xs, l + mid, h);

    t1.start();
    t2.start();

    t1.join();
    t2.join();

    result = t1.result + t2.result;
    return;
}
```

Result

Java.lang.OutOfMemoryError: unable to create new native thread

Divide-and-conquer – with manual fixes (Pt. I)

```
public void run(){
        int size = h-l;
        if (size < SEQ_CUTOFF)
                for (int i=l; i<h; i++)
                        result += xs[i];
        else {
           int mid = size / 2;
           SumThread t1 = new SumThread(xs, l, l + mid);
           SumThread t2 = new SumThread(xs, l + mid, h);
           t1.start();
           t2.start();
           t1.join();
           t2.join();
           result = t1.result + t2.result;
        }
}
```

Half the threads

```
// wasteful: don't
SumThread t1 = …
SumThread t2 = …
t1.start();
t2.start();
t1.join();
t2.join();
result=t1.result+t2.result;
```

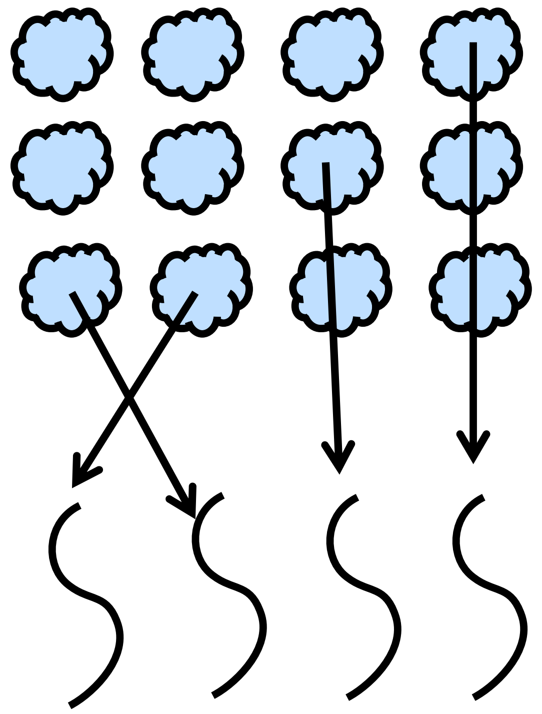
```
// better: do
// order of next 4 lines
// essential – why?
t1.start();
t2.run();
t1.join();
result=t1.result+t2.result;
```

Does this fix our issue?

No. Java Threads are too heavyweight. What should we do?

ExecutorService!

Alternative approach: schedule tasks on threads



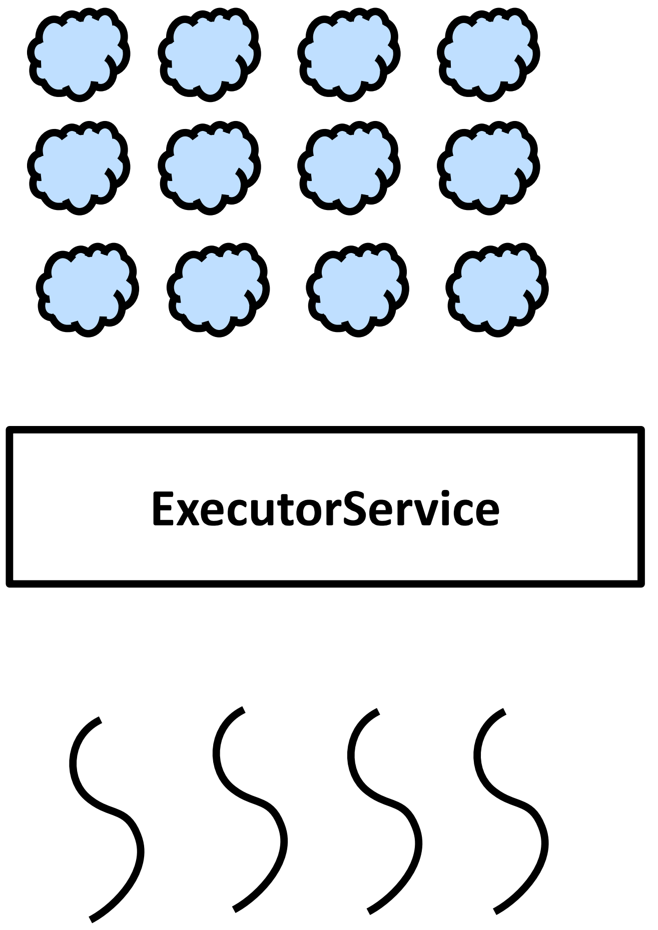
Tasks

(Thread pool)

Threads

How many threads would you use?

Java's executor service: managing asynchronous tasks



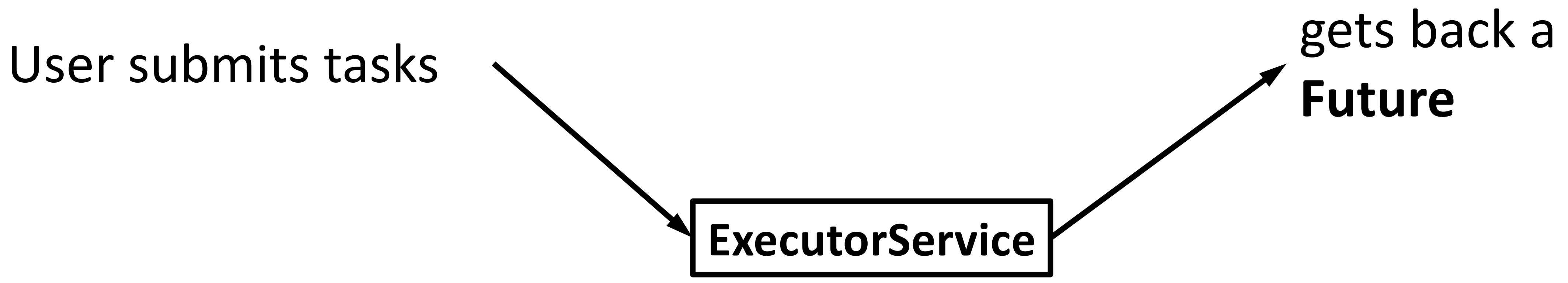
Tasks

Interface

(Thread pool)

Implementation
e.g.: **ThreadPoolExecutor**

Java's executor service:managing asynchronous tasks



```
.submit(Callable<T> task) → Future<T>
.submit(Runnable task)    → Future<?>
```

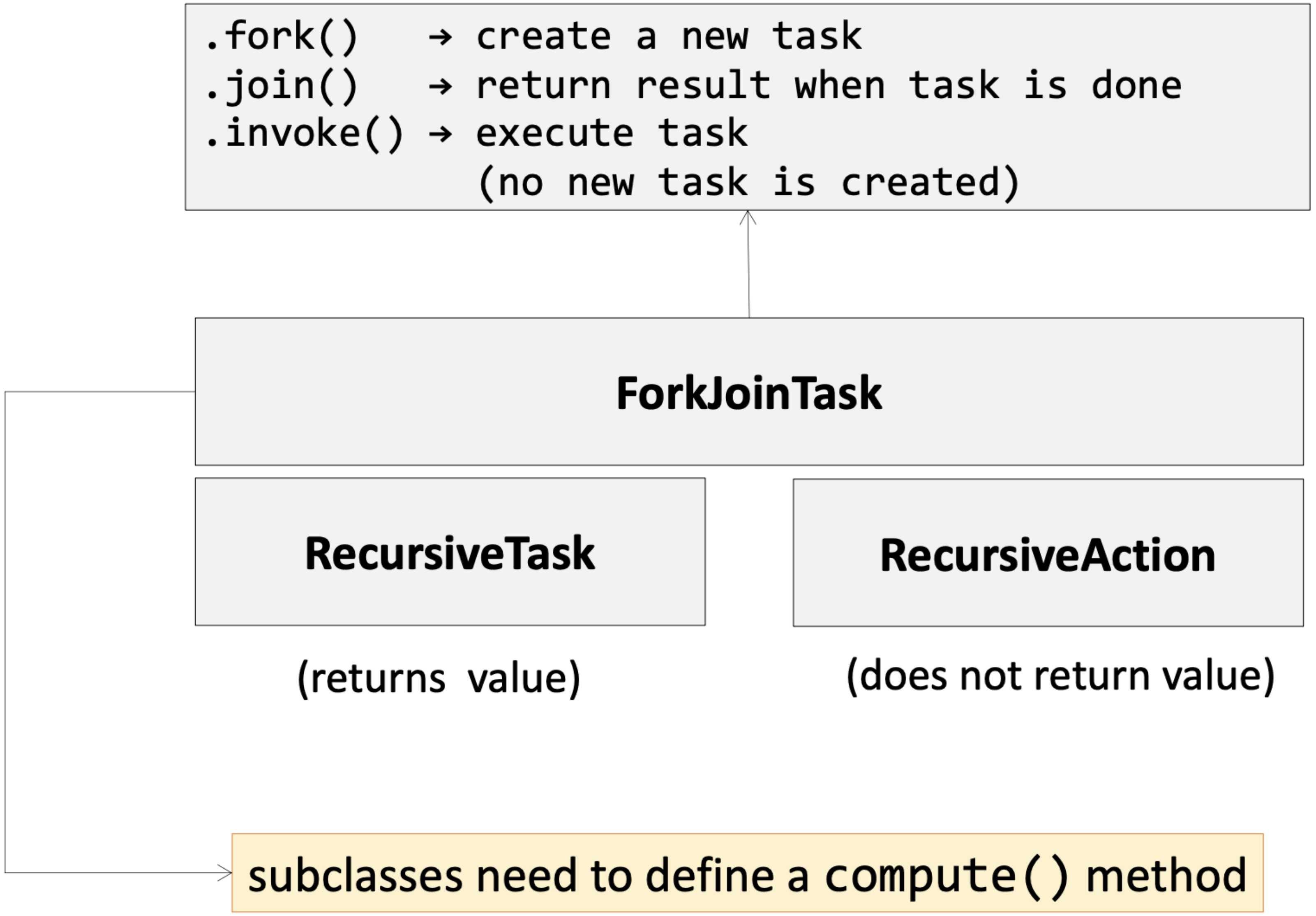
Recursive Sum with ExecutorService

```
public Integer call() throws Exception {
  int size = h - l;
  if (size == 1)
    return xs[l];

  int mid = size / 2;
  sumRecCall c1 = new sumRecCall(ex, xs, l, l + mid);
  sumRecCall c2 = new sumRecCall(ex, xs, l + mid, h);

  Future<Integer> f1 = ex.submit(c1);
  Future<Integer> f2 = ex.submit(c2);

  return f1.get() + f2.get();
}
```

Does this fix our issue?

No. We have dependencies between tasks

Tasks in Fork/Join Framework



```
.fork()   → create a new task
.join()   → return result when task is done
.invoke() → execute task
            (no new task is created)
```

Recursive sum with ForkJoin (compute)

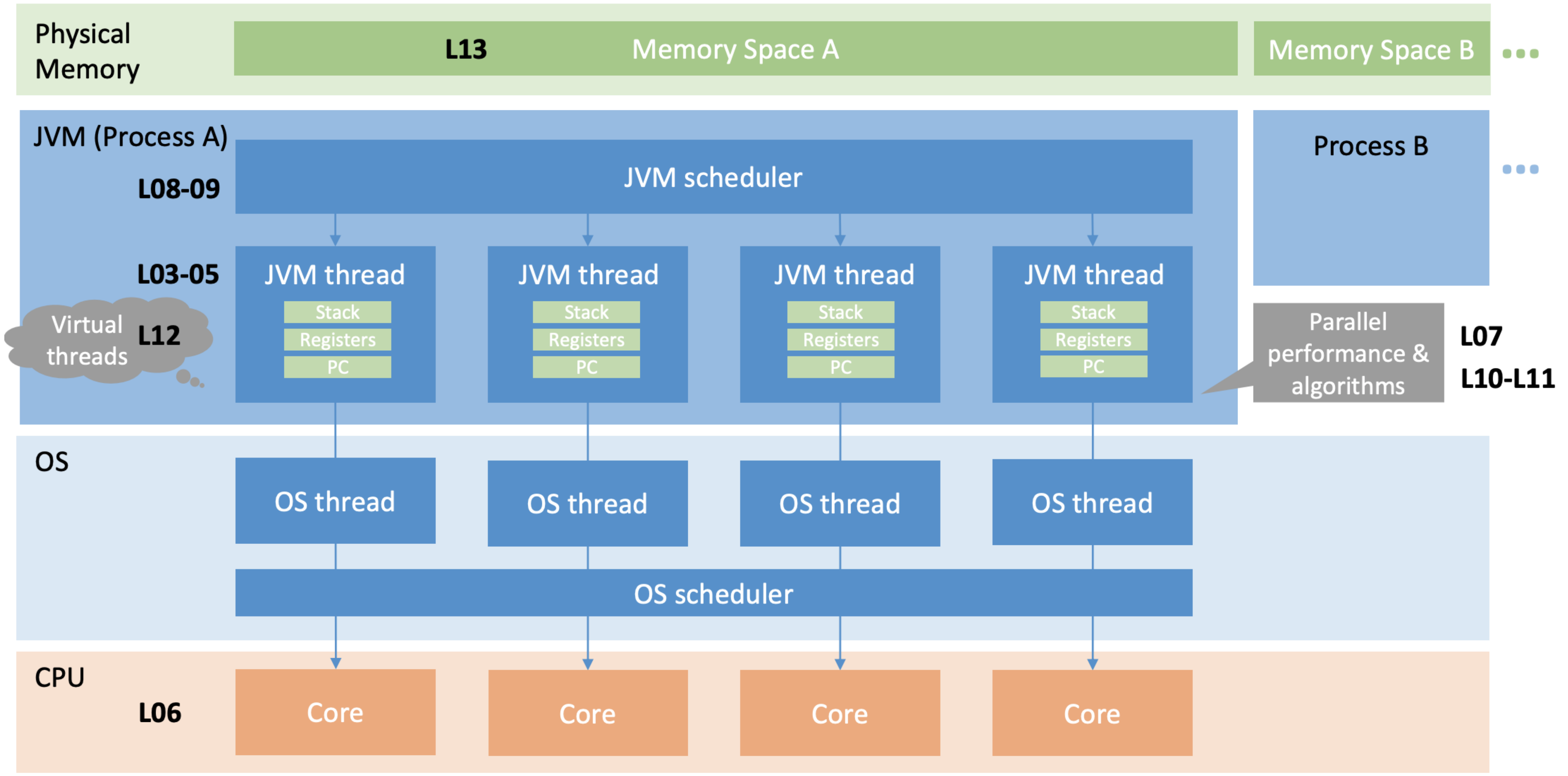
```
protected Long compute() {
       if(high - low <= 1)
           return array[high];
       else {
           int mid = low + (high - low) / 2;
           SumForkJoin left  = new SumForkJoin(array, low, mid);
           SumForkJoin right = new SumForkJoin(array, mid, high);
           left.fork();
           right.fork();
           return left.join() + right.join();
       }
    }
```

Fixes/Work-Around – cont'd

```
protected Long compute() {
       if(high - low <= SEQUENTIAL_THRESHOLD) {
           long sum = 0;
           for(int i=low; i < high; ++i)
               sum += array[i];
           return sum;
        } else {
           int mid = low + (high - low) / 2;
           SumForkJoin left  = new SumForkJoin(array, low, mid);
           SumForkJoin right = new SumForkJoin(array, mid, high);
           left.fork();
           long rightAns = right.compute();
           long leftAns  = left.join();
           return leftAns + rightAns;
        }
    }
```

Make sure each task has 'enough' to do!

Big Picture


Physical Memory
L13
Memory Space A
Memory Space B
JVM (Process A)
L08-09
JVM scheduler
Process B
L03-05
JVM thread
Stack
Registers
PC
Virtual threads
L12
Parallel performance & algorithms
L07
L10-L11
OS
OS thread
OS scheduler
CPU
L06
Core

Credits to Erxuan Li PProg FS25

The prefix-sum problem

Given **`int[] input`**,

produce **`int[] output`** where:

`output[i]` = **`input[0]+input[1]+…+input[i]`**

Sequential prefix-sum

```
int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];

    for(int i = 1; i < input.length; i++)
        output[i] = output[i-1] + input[i];

    return output;
}
```

Does not seem parallelizable

- Work: O(n), Span: O(n)
- This algorithm is sequential, but a **different algorithm** has: Work O(n), Span O(`log` n)

Parallel prefix-sum

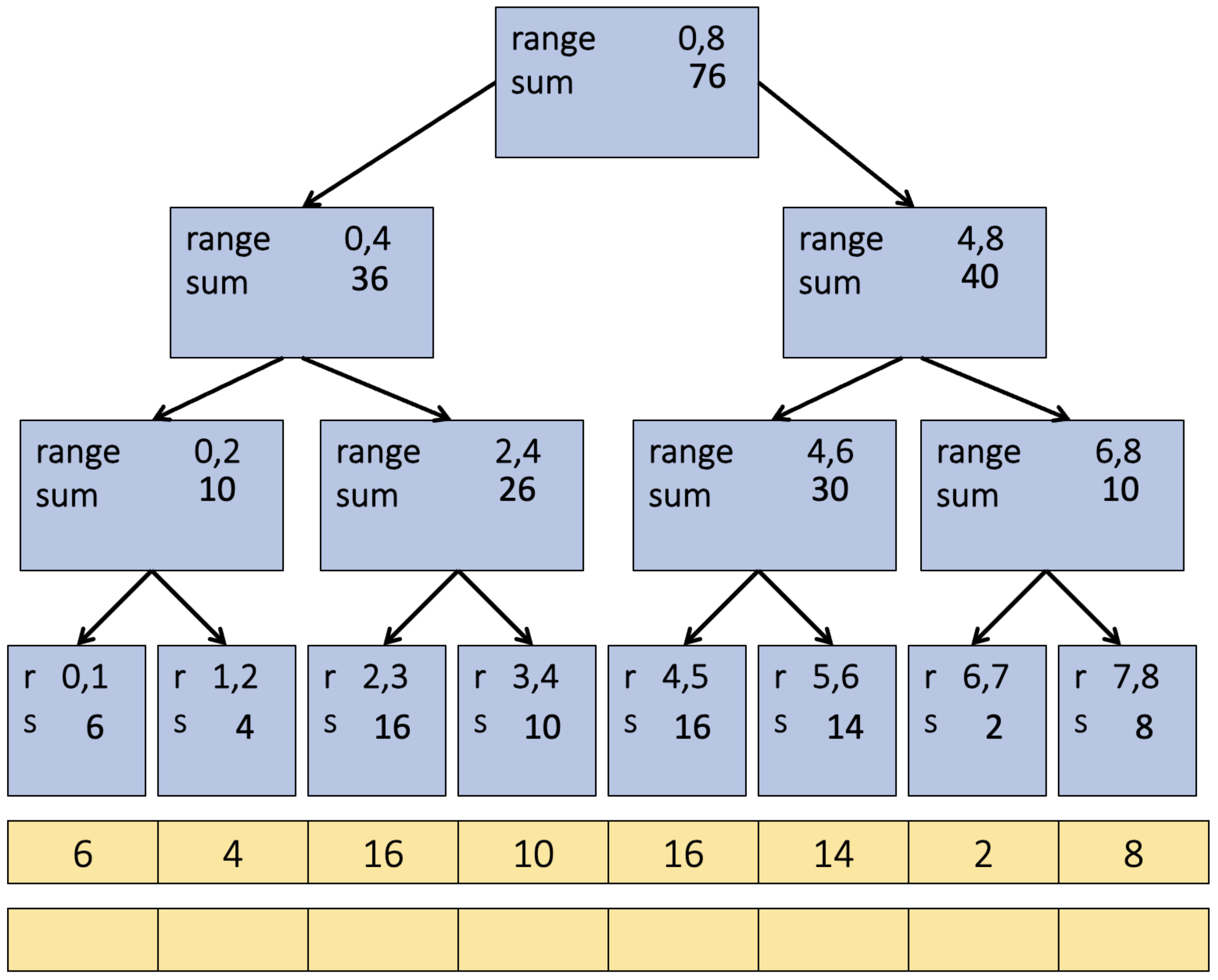
The parallel-prefix algorithm does two passes

- Each pass has O(n) work and O(`log` n) span
- So in total there is O(n) work and O(`log` n) span
- So like with array summing, parallelism is n/`log` n

First pass builds a tree bottom-up: the “up” pass

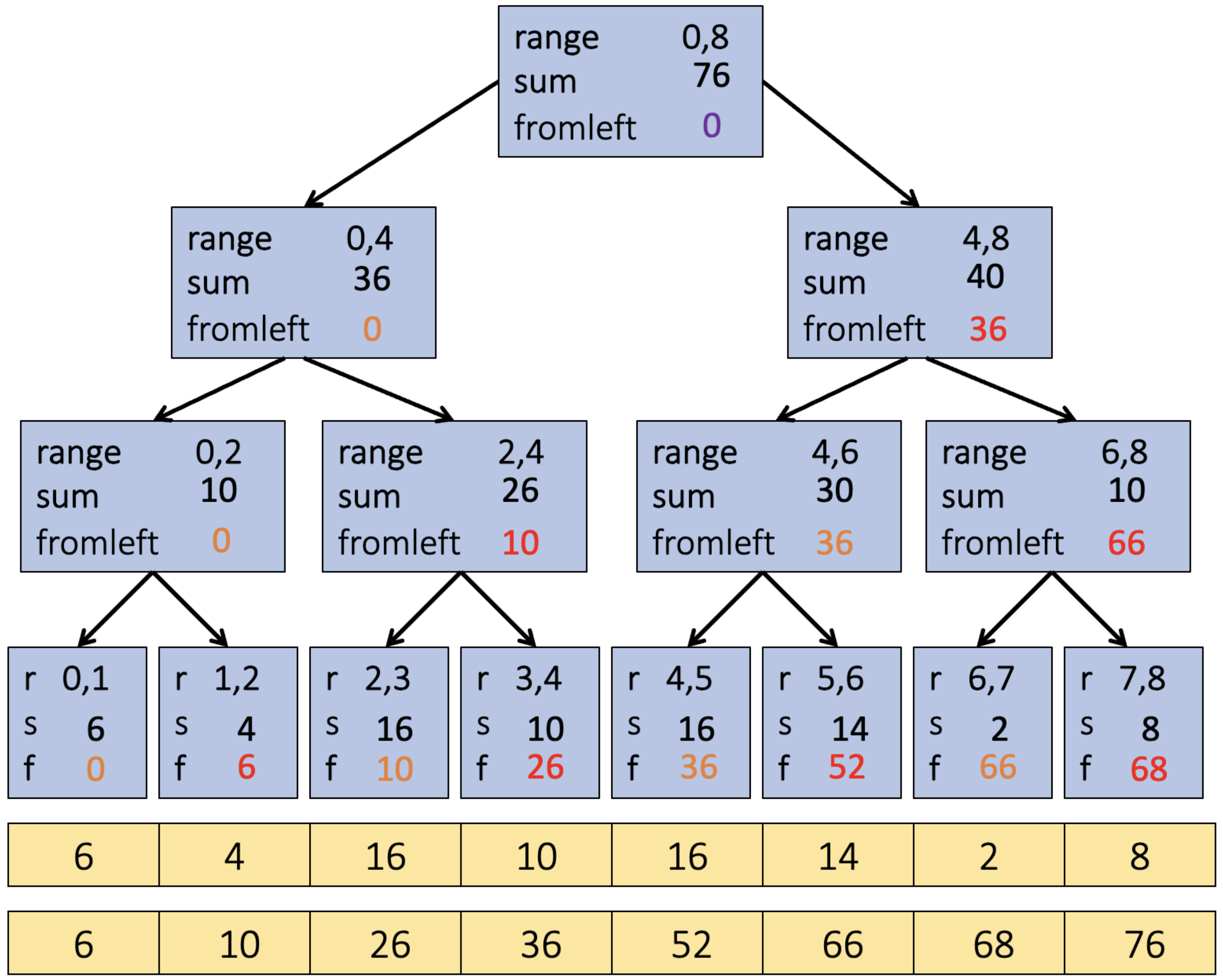
Second pass traverses the tree top-down: the “down” pass

Example



`input`	6	4	16	10	16	14	2	8
`output`								

Example



input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

Plan für heute

- Organisation
- Nachbesprechung Exercise 6
- Summary Part 1
- **Intro Lecture Part 2**
- Intro Exercise 7
- Exam Questions
- Kahoot

The Future

- Prof. Höfler hat den ACM Prize gewonnen

Torsten Hoefler, Professor at ETH Zürich, Chief Architect for AI and Machine Learning at CSCS, and recipient of the ACM Prize in Computing 2024. (Image: Massimo Piccoli, CSCS)

Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?

Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?
  - Atomics geben uns eine Möglichkeit
  - Was wenn wir keine atomics haben?

Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?
  - Atomics geben uns eine Möglichkeit
  - Was wenn wir keine atomics haben?
- Volatile!

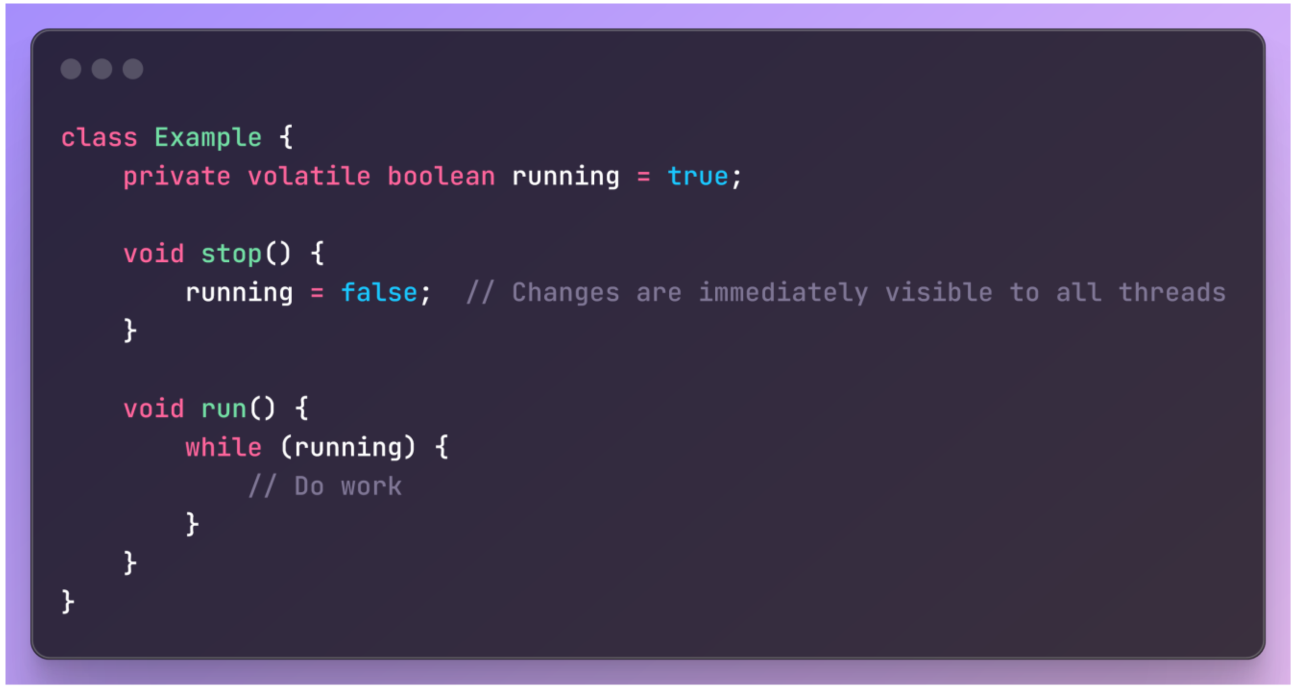
Volatile Keyword

- When multiple threads access a shared variable, each thread may keep local copy in its CPU cache, updates might not be immediately visible to other threads

- Volatile gives us visibility guarantee!

Volatile Keyword

- Volatile variable ensures that any read or write operation always happens directly in main memory, so all threads see latest value on next read

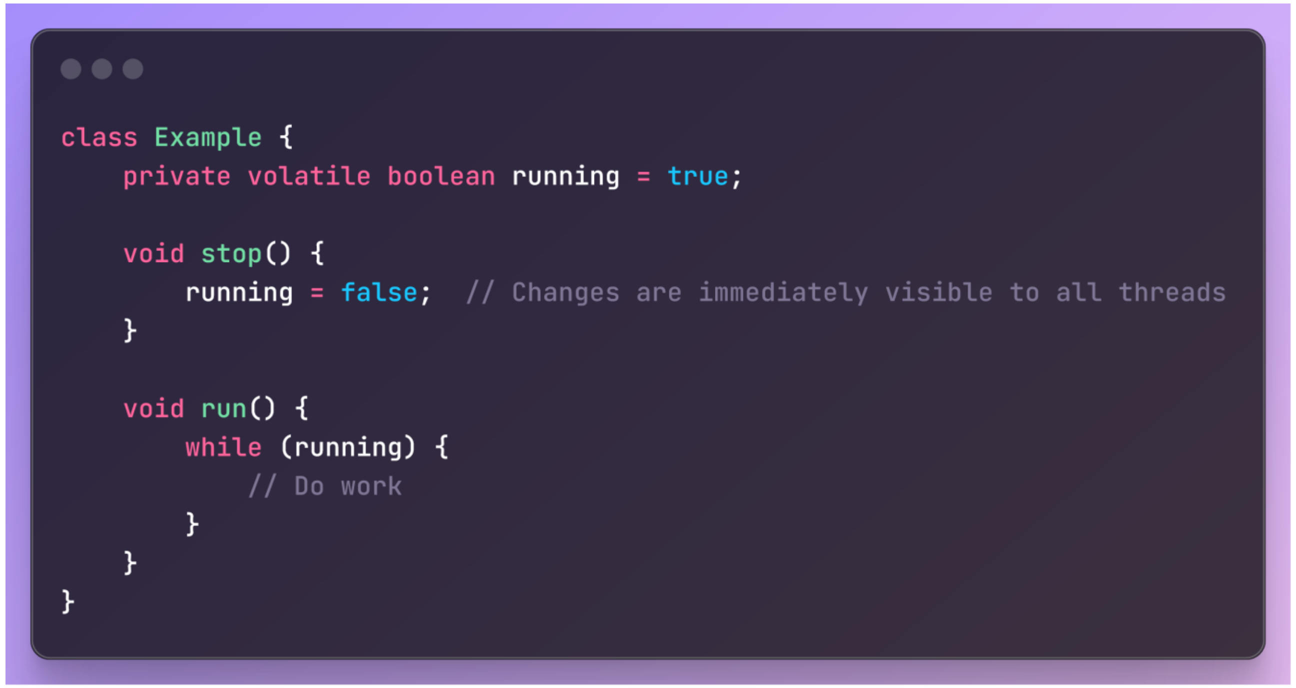


```java
class Example {
    private volatile boolean running = true;

    void stop() {
        running = false;  // Changes are immediately visible to all threads
    }

    void run() {
        while (running) {
            // Do work
        }
    }
}
```

Volatile Keyword

- Without volatile, another thread calling stop() might not be seen by the run() method because the CPU might cache running locally

- With volatile, the change to running is guaranteed to be visible to all threads



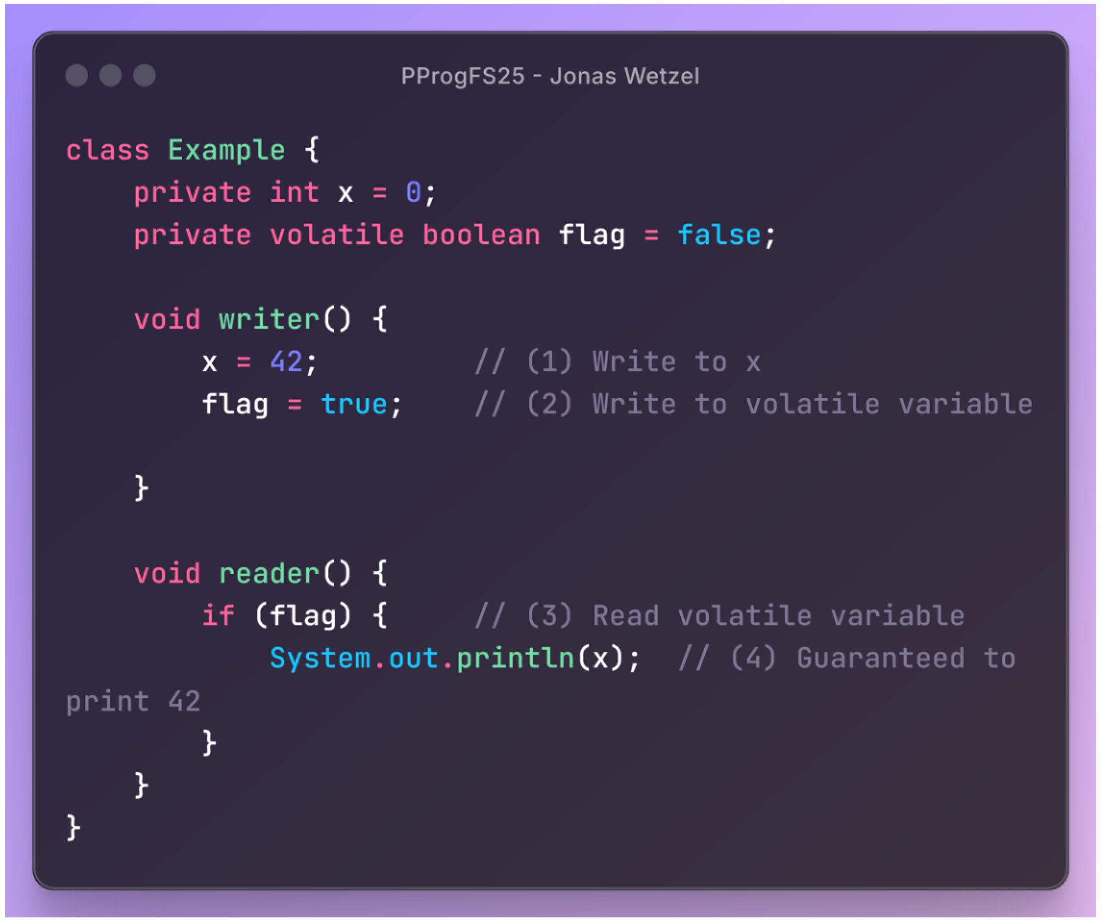
```java
class Example {
    private volatile boolean running = true;

    void stop() {
        running = false;  // Changes are immediately visible to all threads
    }

    void run() {
        while (running) {
            // Do work
        }
    }
}
```

Volatile Keyword

- It also prevents optimizations like out of order execution from happening!
- Java Memory Model allows JVM and CPU to reorder instructions for optimization

Volatile Keyword

- Without volatile, compiler or CPU might reorder (1) and (2), leading reader() to see flag == true but still read old value of x
- With volatile, (2) happens after (1), ensuring x = 42 is visible before flag = true is read.



PProgFS25 - Jonas Wetzel

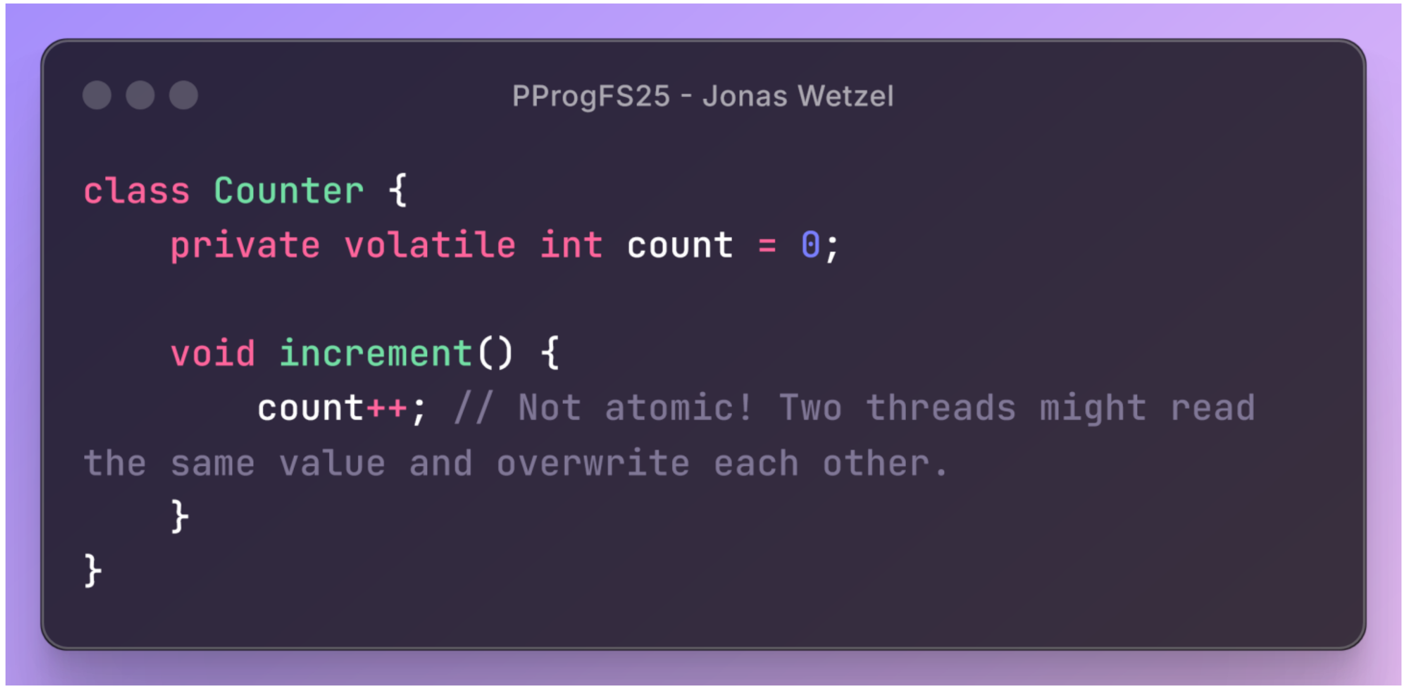
```
class Example {
    private int x = 0;
    private volatile boolean flag = false;

    void writer() {
        x = 42;          // (1) Write to x
        flag = true;     // (2) Write to volatile variable

    }

    void reader() {
        if (flag) {      // (3) Read volatile variable
            System.out.println(x);  // (4) Guaranteed to
print 42
        }
    }
}
```

Volatile Keyword

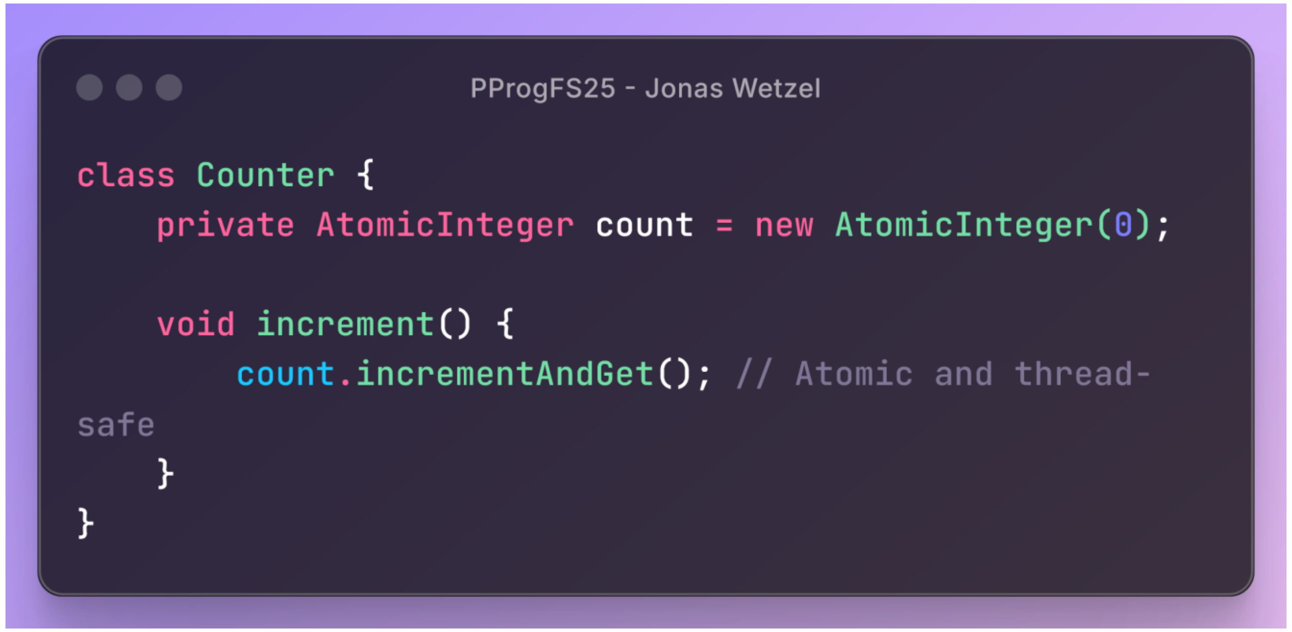
- Volatile has limitations
- Does not prevent race conditions: volatile ensures visibility but not atomicity for compound actions like count++
- Not a replacement for synchronization: It doesn’t provide mutual exclusion (locking)

Volatile Keyword

- Volatile has limitations
- Does not prevent race conditions: volatile ensures visibility but not atomicity for compound actions like count++
- Not a replacement for synchronization: It doesn’t provide mutual exclusion (locking)


PProgFS25 - Jonas Wetzel

```
class Counter {
    private volatile int count = 0;

    void increment() {
        count++; // Not atomic! Two threads might read
the same value and overwrite each other.
    }
}
```


PProgFS25 - Jonas Wetzel

```
class Counter {
    private AtomicInteger count = new AtomicInteger(0);

    void increment() {
        count.incrementAndGet(); // Atomic and thread-
safe
    }
}
```

Volatile Summary

- Volatile ensures that all reads and writes go directly to main memory, preventing stale values
- It prevents instruction reordering
- It does NOT provide atomicity or mutual exclusion
- Suitable for simple flags and state indicators, but not for counters or complex data structures

Volatile

- Suitable for simple flags and state indicators, but not for counters or complex data structures
- Can we build a lock with that?

Volatile

- Suitable for simple flags and state indicators, but not for counters or complex data structures
- Can we build a lock with that?
- Yes, we can!

Deckers Lock

- Each thread sets its flag[id] = true to indicate that it wants access to critical section
- If other thread also wants access, they use turn variable to decide who goes first
- If it's not thread's turn, it backs off, resets its flag, and waits for its turn
- After exiting critical section, thread gives turn to the other thread and resets its flag

A combination of the tries 2 and 3: Decker’s Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

A combination of the tries 2 and 3: Decker’s Algorithm

```
volatile boolean wantp=false, wantq=false, integer turn= 1
```

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

A combination of the tries 2 and 3: Decker’s Algorithm

```
volatile boolean wantp=false, wantq=false, integer turn= 1
```

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

A combination of the tries 2 and 3: Decker’s Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

Deckers Lock

- ✅ Ensures mutual exclusion (only one thread enters the critical section at a time)
- ✅ Avoids deadlock by using the turn variable
- ✅ Provides fairness (both threads get their turn)
- ❌ Limited to two threads (doesn’t scale well)

Petersons Lock

- Each thread sets flag[id] = true to indicate it wants access to the critical section.
- The thread gives priority to the other thread by setting turn = other.
- If the other thread also wants access (flag[other] == true) and it’s still its turn, the thread waits.
- Once it gets access, it enters the critical section.
- After exiting, the thread resets flag[id] = false so the other thread can proceed.

More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

Peterson Lock

- ✅ **Ensures mutual exclusion** (only one thread enters the critical section at a time)
  ✅ **Prevents deadlock** (always allows progress)
  ✅ **Fair (bounded waiting)** (no thread is starved forever)
- ✅ **simpler** than Deckers Lock
  ❌ **Works only for two threads** (not scalable)

Peterson Lock

- ✅ **Ensures mutual exclusion** (only one thread enters the critical section at a time)
  ✅ **Prevents deadlock** (always allows progress)
  ✅ **Fair (bounded waiting)** (no thread is starved forever)
- ✅ **simpler** than Deckers Lock
  ❌ **Works only for two threads** (not scalable)

- Bakery Lock allows us to extend Peterson Lock Idea to n Threads!
  - We'll see it in like 3 weeks

Deckers Lock & Peterson Lock

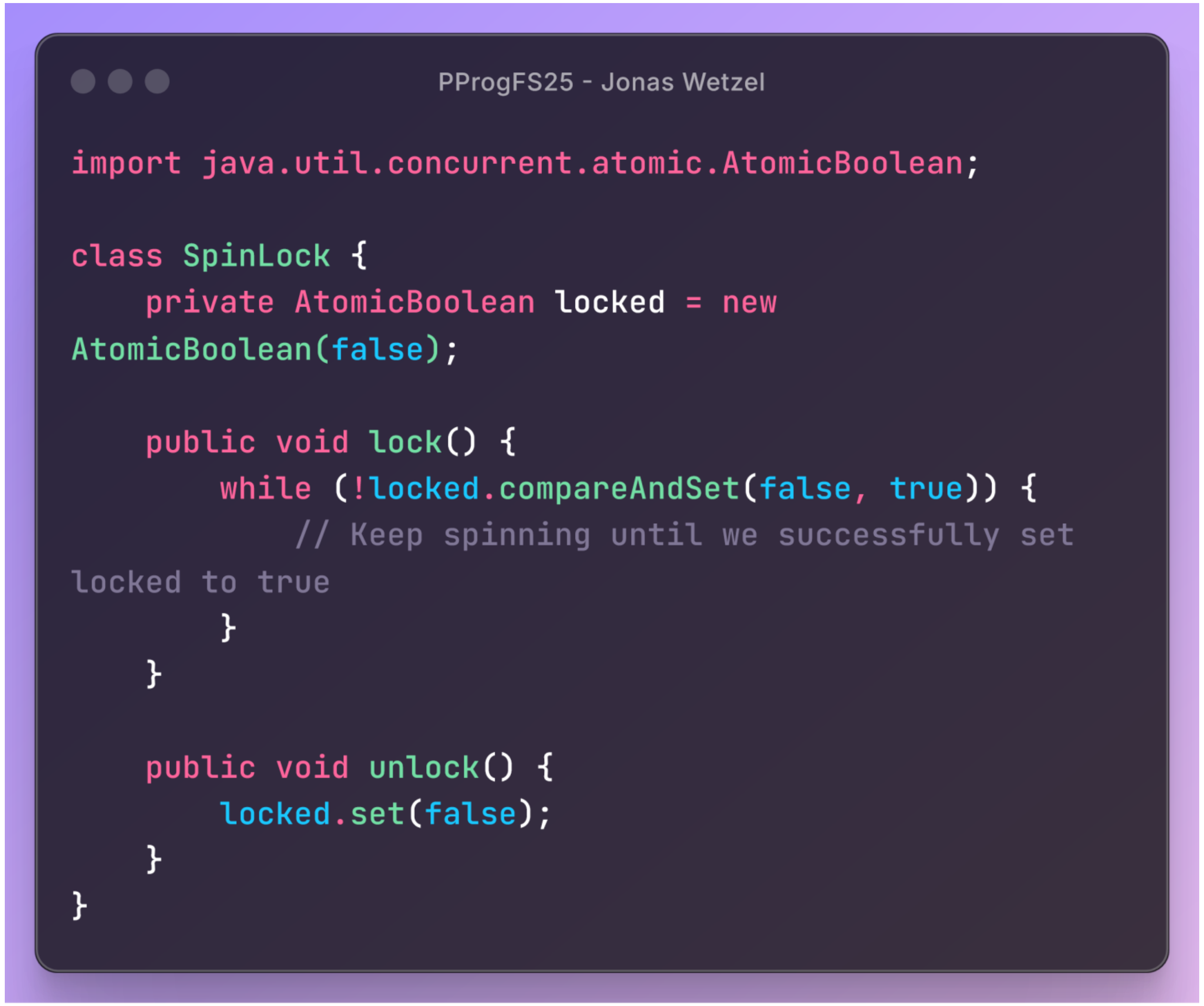
- See code

Can we build a lock with atomics?

- How?

Can we build a lock with atomics?

- Now you see why we like atomics so much. It's much simpler!



```java
import java.util.concurrent.atomic.AtomicBoolean;

class SpinLock {
    private AtomicBoolean locked = new
AtomicBoolean(false);

    public void lock() {
        while (!locked.compareAndSet(false, true)) {
            // Keep spinning until we successfully set
locked to true
        }
    }

    public void unlock() {
        locked.set(false);
    }
}
```

Plan für heute

- Organisation
- Nachbesprechung Exercise 6
- Theory Recap
- **Intro Exercise 7**
- Exam Questions
- Kahoot

Pre-Discussion Exercise 7

Exercise 7

Banking System

- Multi-Threaded Implementation
- Coding exercise: Use **synchronized** and/or **Locks**
  - Might have to make additions to existing classes
- Reason about Performance
- Reason about Deadlocks
- Run Tests

Multi-threaded Implementation

Task 1 – Problem Identification:

The methods of the classes **Account** and **BankingSystem** must be thread-safe.

You should understand why the current implementation does not work for more than one thread.

Thread-Safe – transferMoney()

Task 2 – Synchronized:

A simple solution to make the *transferMoney()* thread-safe is to use the **synchronized** keyword:
public synchronized boolean transferMoney(…)

Even though the code works as expected, the performance is poor.

The performance of the multi-threaded implementation is worse than the single-threaded. Why does this happen?

Performance of transferMoney()

Task 3 – Locking:

Since the solution with the synchronized keyword does not perform well, you should find a better strategy to achieve the thread-safe implementation.

- *Does your proposed solution work if a transaction happens from and to the same account?*
- *How do you know that your proposed solution does not suffer from deadlocks?*

ThreadSafe - sumAccounts()

Task 4 – Summing Up

With a fine-grained synchronization on the transfer method, the method sumAccounts() may return incorrect results when a transaction takes place at the same time.

- Explain why the current implementation of the sumAccounts() method is not thread-safe any more.
- You should provide a thread-safe implementation.
- Is there any way to parallelize this method?

Testing

You should run the provided tests for your implementation.

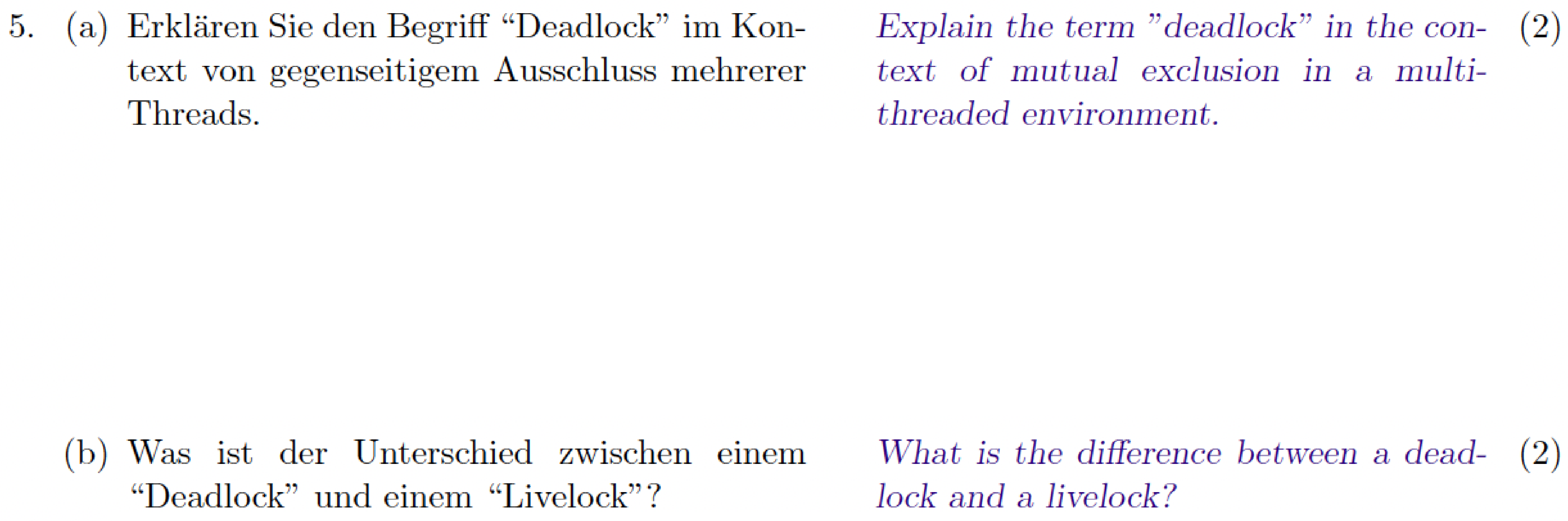
If the test succeeds, your code is not necessarily correct.

It is hard to reproduce a bad interleaving.

Plan für heute

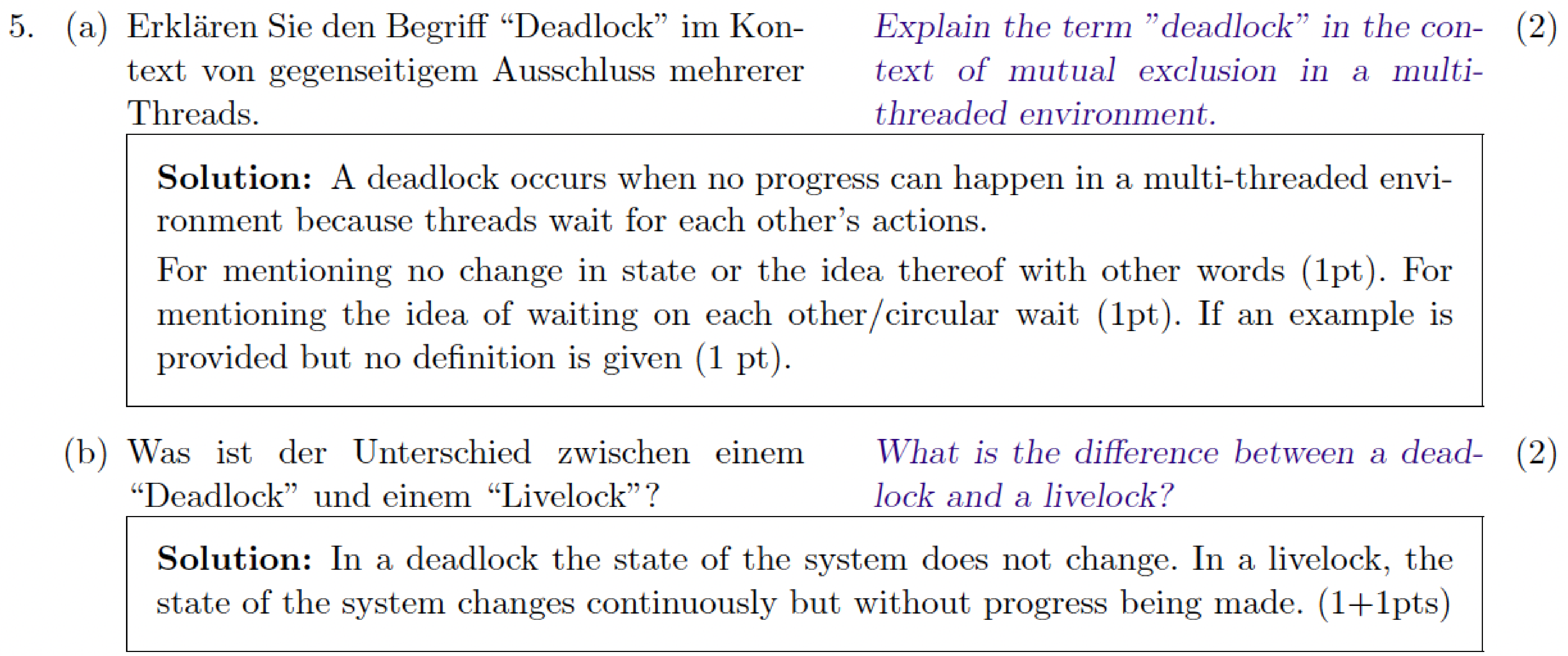
- Organisation
- Nachbesprechung Exercise 6
- Theory Recap
- Intro Exercise 7
- **Exam Questions**
- Kahoot

Old Exam Task (FS 2023)



5.	(a)	Erklären Sie den Begriff “Deadlock” im Kontext von gegenseitigem Ausschluss mehrerer Threads.	*Explain the term ”deadlock” in the context of mutual exclusion in a multi-threaded environment.*	(2)
	(b)	Was ist der Unterschied zwischen einem “Deadlock” und einem “Livelock”?	*What is the difference between a deadlock and a livelock?*	(2)

Old Exam Task (FS 2023)



5. (a) Erklären Sie den Begriff “Deadlock” im Kontext von gegenseitigem Ausschluss mehrerer Threads.

Explain the term ”deadlock” in the context of mutual exclusion in a multi-threaded environment. (2)

Solution: A deadlock occurs when no progress can happen in a multi-threaded environment because threads wait for each other’s actions.

For mentioning no change in state or the idea thereof with other words (1pt). For mentioning the idea of waiting on each other/circular wait (1pt). If an example is provided but no definition is given (1 pt).

(b) Was ist der Unterschied zwischen einem “Deadlock” und einem “Livelock”?

What is the difference between a deadlock and a livelock? (2)

Solution: In a deadlock the state of the system does not change. In a livelock, the state of the system changes continuously but without progress being made. (1+1pts)

```java
public class Main {
  public static Thread CreateThread(int start) {
    return new Thread(new Runnable() {

      @Override
      public void run() {
        for (int i = start; i < 7; i+=2) {
            System.out.println("Number " + i);
        }
      }
    });
  }

  public static void main(String[] args) throws InterruptedException {
    CreateThread(1).start();
    CreateThread(2).start();
  }
}
```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

Mark all the print sequences that can be produced by running the program shown above.

☐

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
```

☐

```
Number 1
Number 6
Number 3
Number 4
Number 5
Number 2
```

☐

```
Number 6
Number 5
Number 4
Number 3
Number 2
Number 1
```

☐

```
Number 2
Number 4
Number 6
Number 1
Number 3
Number 5
```

```java
public class Main {
  public static Thread CreateThread(int start) {
    return new Thread(new Runnable() {

      @Override
      public void run() {
        for (int i = start; i < 7; i+=2) {
            System.out.println("Number " + i);
        }
      }
    });
  }

  public static void main(String[] args) throws InterruptedException {
    CreateThread(1).start();
    CreateThread(2).start();
  }
}
```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

Mark all the print sequences that can be produced by running the program shown above.

[✓ ☐]

```
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
```

☐

```
Number 1
Number 6
Number 3
Number 4
Number 5
Number 2
```

☐

```
Number 6
Number 5
Number 4
Number 3
Number 2
Number 1
```

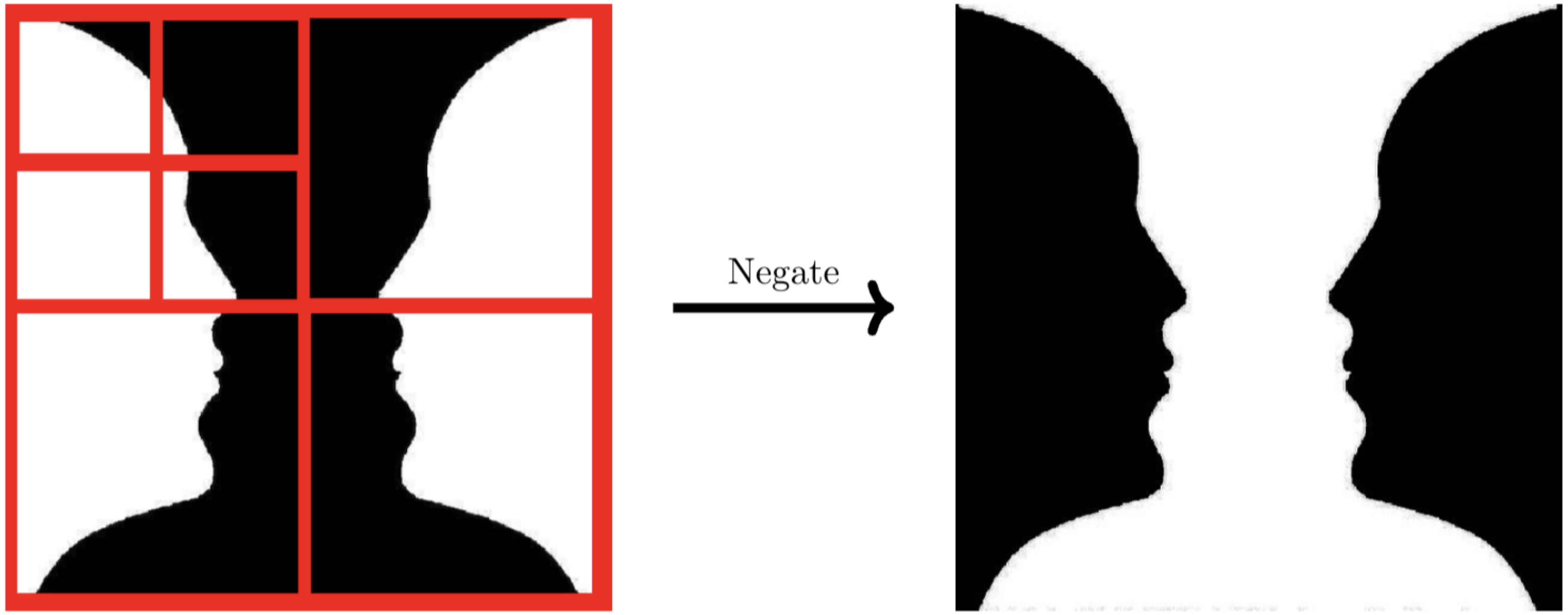
[✓ ☐]

```
Number 2
Number 4
Number 6
Number 1
Number 3
Number 5
```

Fork/Join Framework (16 points)

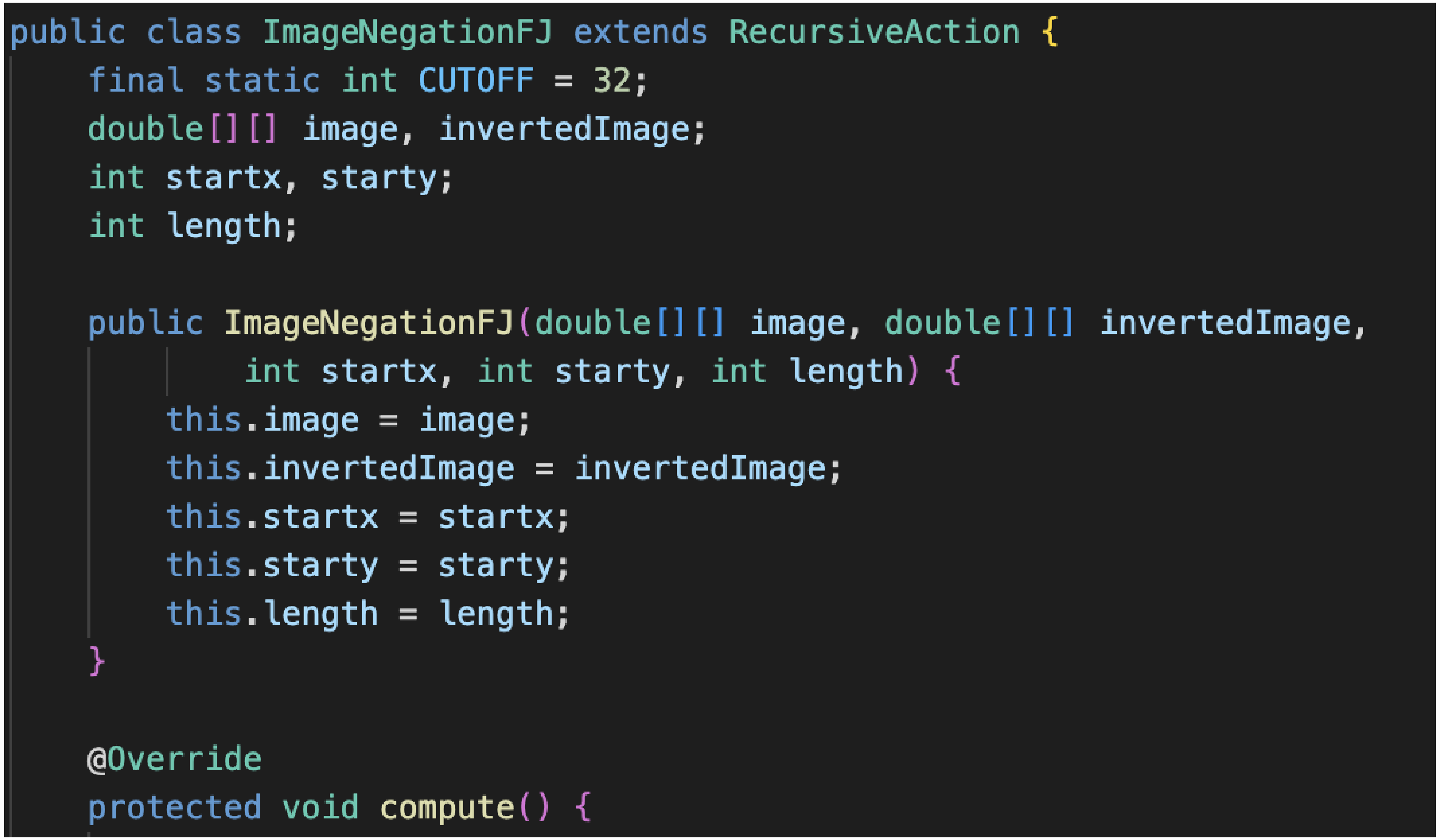
3. Der folgende Code zielt darauf ab, ein Bild zu negieren, indem es mithilfe des Fork/Join-Frameworks rekursiv in mehrere Unterfenster (vier pro Rekursionsschritt) unterteilt wird. Die Unterfenster können dann parallel negiert werden. Das folgende Beispiel verdeutlicht die Unterteilung des Bildes und die Negierung der einzelnen Unterfenster.

The following code aims to negate an image by recursively subdividing it into multiple subwindows (four per recursion step) using the Fork/Join framework. The subwindows can then be negated in parallel. The example below illustrates the subdivision of the image and negation of the individual subwindows.

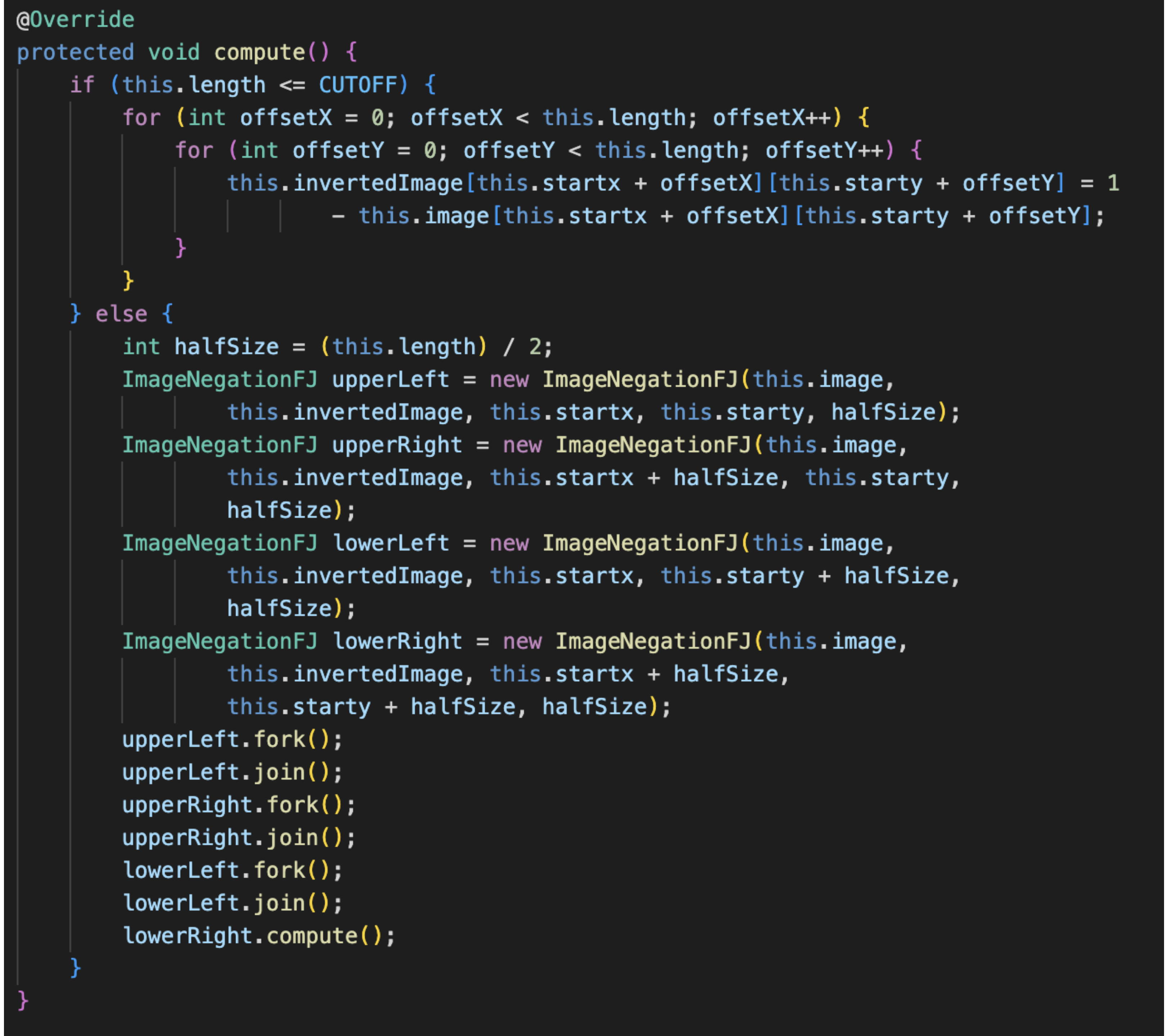


Bitte lesen Sie den Code sorgfältig durch und beantworten Sie dann die Fragen zum Code:

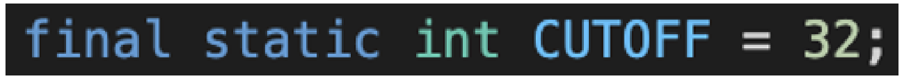
Please read the code carefully and then answer the questions regarding the code:



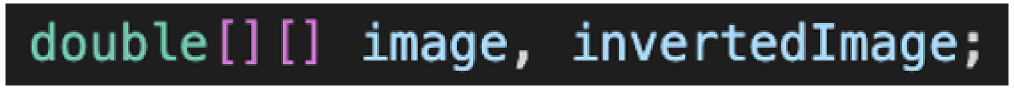
```
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```



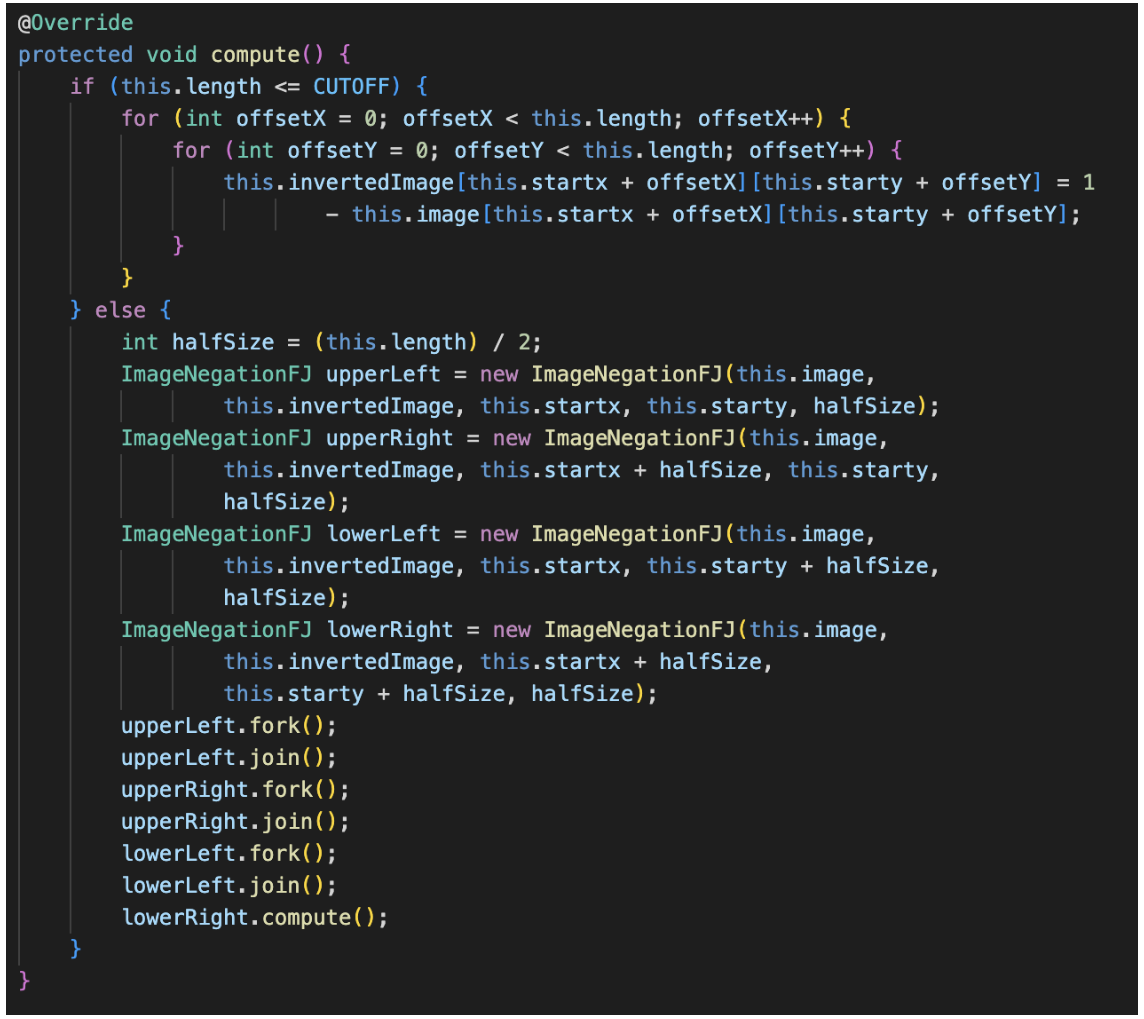
```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```



```java
final static int CUTOFF = 32;
```



```java
double[][] image, invertedImage;
```



```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

(a) Welche Annahme trifft der Code bezüglich der Abmessungen des Arrays, das das Eingabebild darstellt?

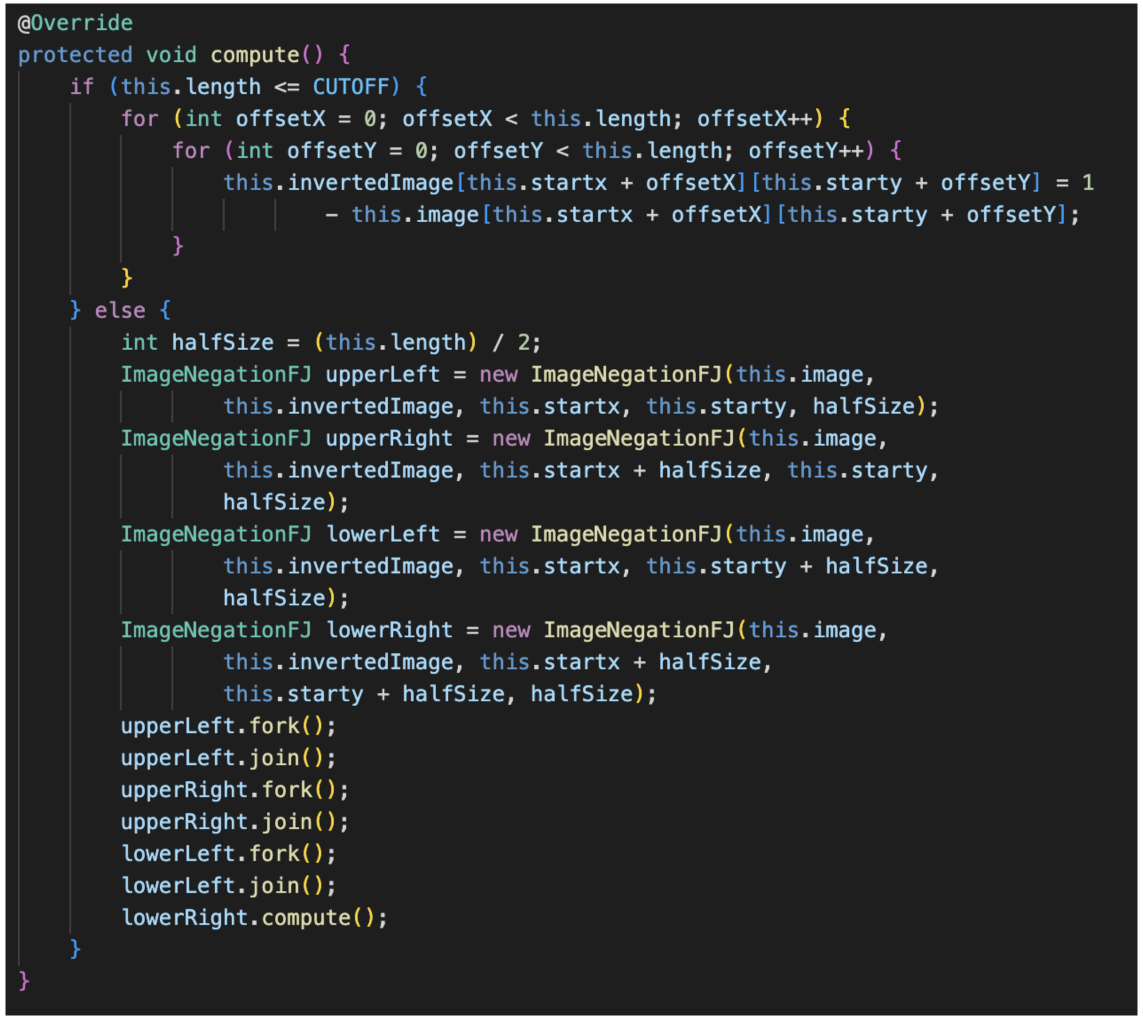
What assumption does the code make concerning the dimensions of the array representing the input image? (2)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

13

The image should be square $s \times s$ and we should have $s = d2^k$, where $d \leq 32$. This is necessary, because we want `length` to be divisible by 2 in the case `length > 32`. If this would not be the case, we would do floor division and leave pixels unprocessed.

+ Add Comment ··· More

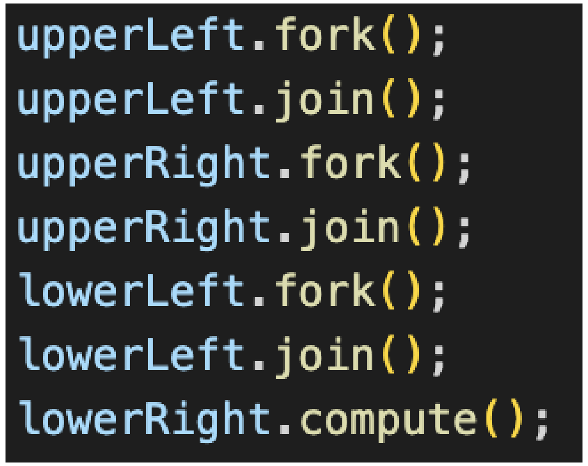


```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

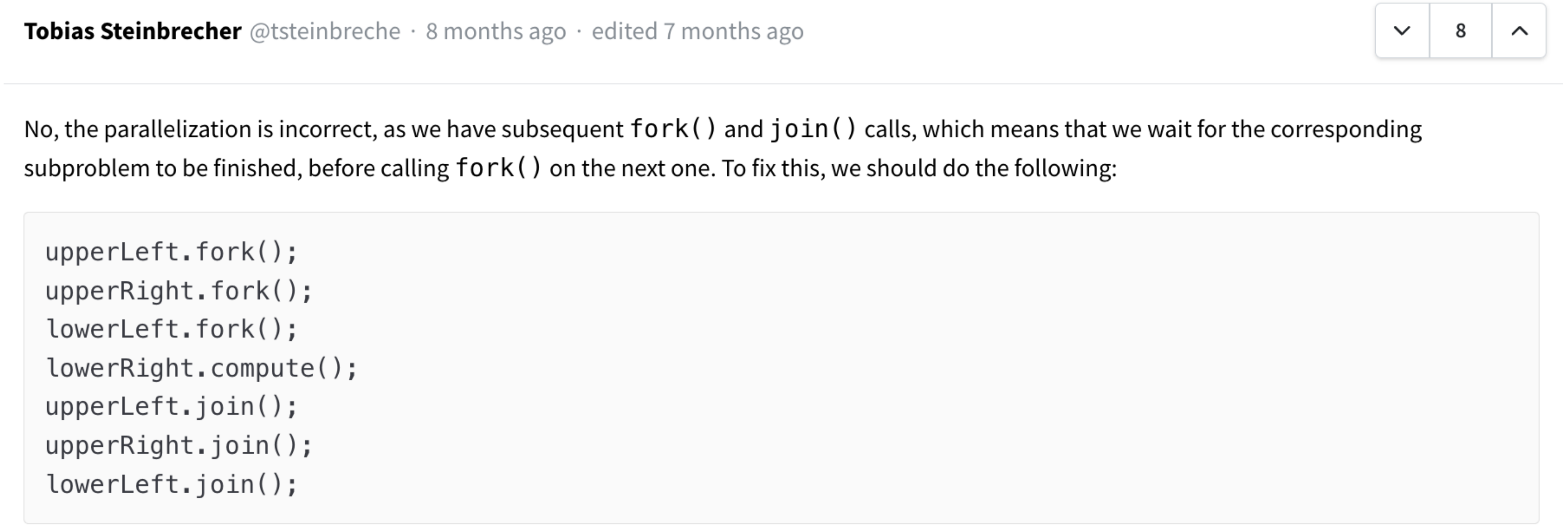
(b) Parallelisiert der Code die beabsichtigte Aufgabe korrekt oder gibt es weitere Optimierungsmöglichkeiten? Wenn ja, welche Optimierung würden Sie vorschlagen und warum?

Does the code correctly parallelize the intended task or is there further optimization that could be done? If so, which optimization would you propose and why? (4)

Not good:



```
upperLeft.fork();
upperLeft.join();
upperRight.fork();
upperRight.join();
lowerLeft.fork();
lowerLeft.join();
lowerRight.compute();
```

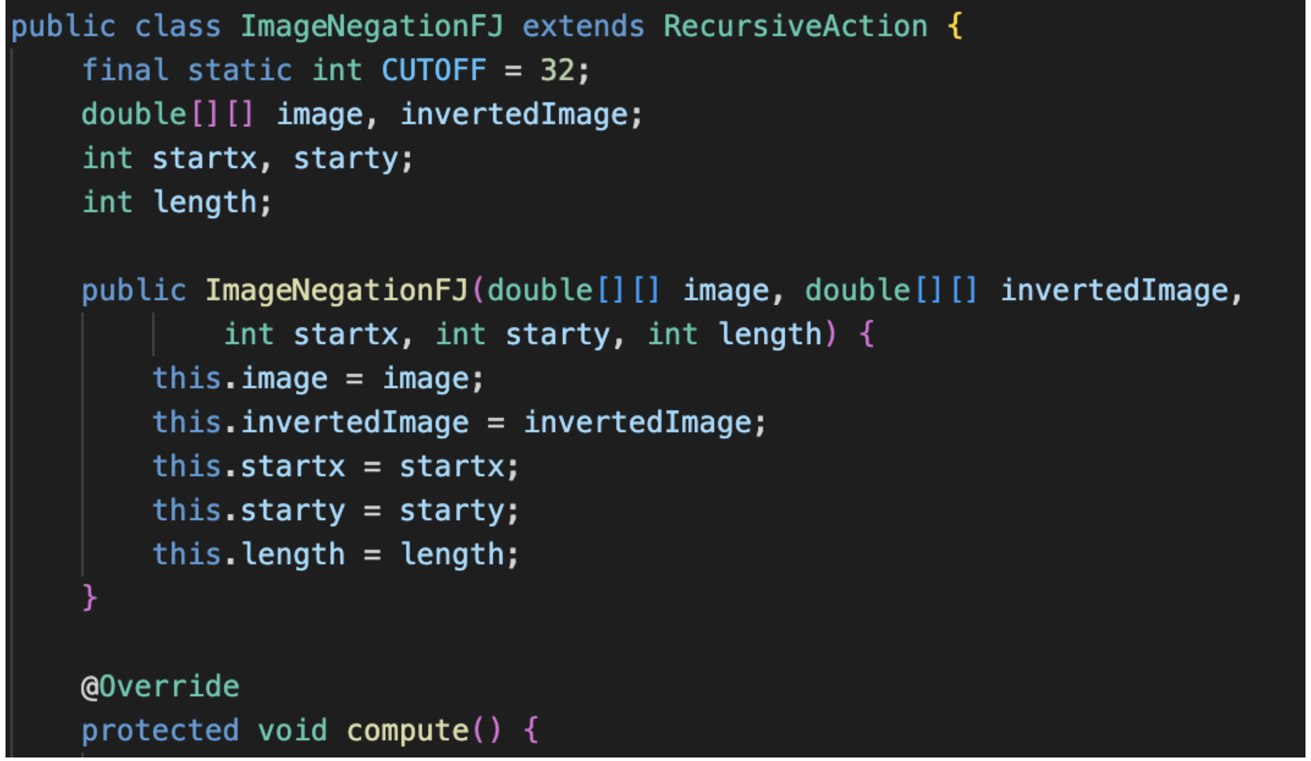


Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 7 months ago

8

No, the parallelization is incorrect, as we have subsequent `fork()` and `join()` calls, which means that we wait for the corresponding subproblem to be finished, before calling `fork()` on the next one. To fix this, we should do the following:

```
upperLeft.fork();
upperRight.fork();
lowerLeft.fork();
lowerRight.compute();
upperLeft.join();
upperRight.join();
lowerLeft.join();
```



```java
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```

(c) Vervollständigen Sie das folgende Codegerüst, indem Sie die oben implementierte `ImageNegationFJ` Klasse und die `ForkJoinPool` Klasse verwenden, um die Variable `negatedImage` mit den negierten Werten zu füllen.

Complete the following code skeleton by using the above implemented `ImageNegationFJ` class and the `ForkJoinPool` class to fill the variable `negatedImage` with the negated values. (4)

```java
double[][] image = {{0, 1}, {1, 0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
.......................................................
.......................................................
.......................................................
.......................................................
.......................................................
.......................................................
```

Tobias Steinbrecher @tsteinbreche · 8 months ago

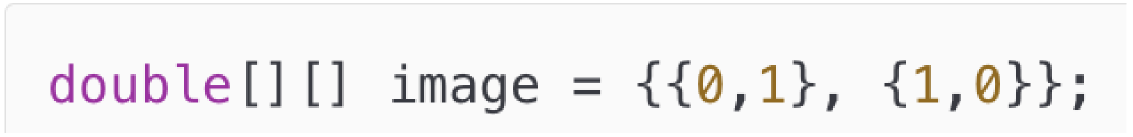
10

```java
double[][] image = {{0,1}, {1,0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
ForkJoinPool fjp = new ForkJoinPool();
ForkJoinTask t = new ImageNegationFJ(image, negatedImage, 0, 0 ,imageSize);
fjp.invoke(t);
```

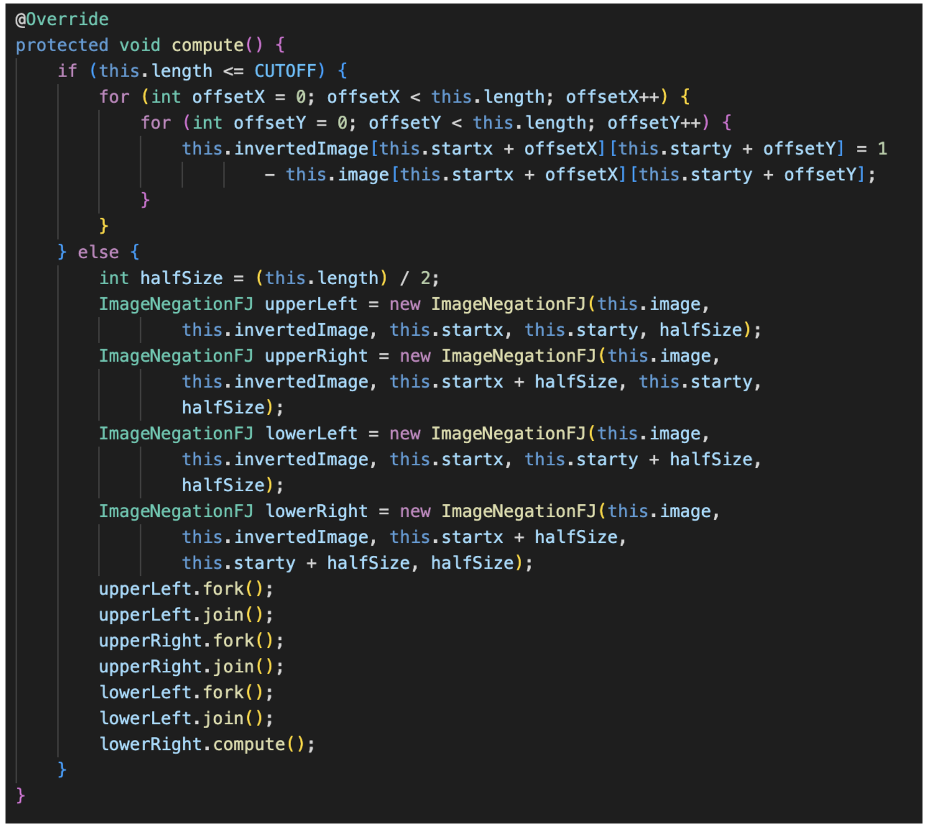
+ Add Comment ··· More

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoin-Pool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

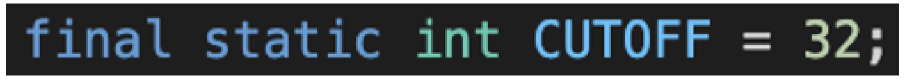
Assuming that the `ImageNegationFJ` *class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the* 2×2 `negatedImage` *array from task 3c)?* (2)



```java
double[][] image = {{0,1}, {1,0}};
```



```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

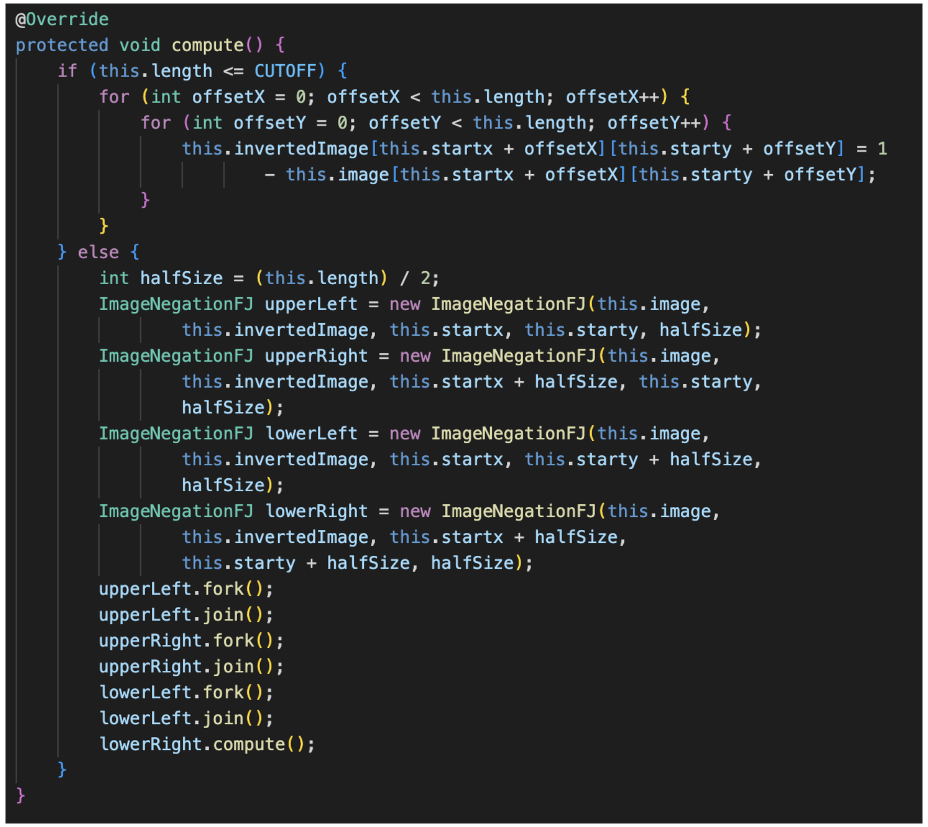


```java
final static int CUTOFF = 32;
```

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoin-Pool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

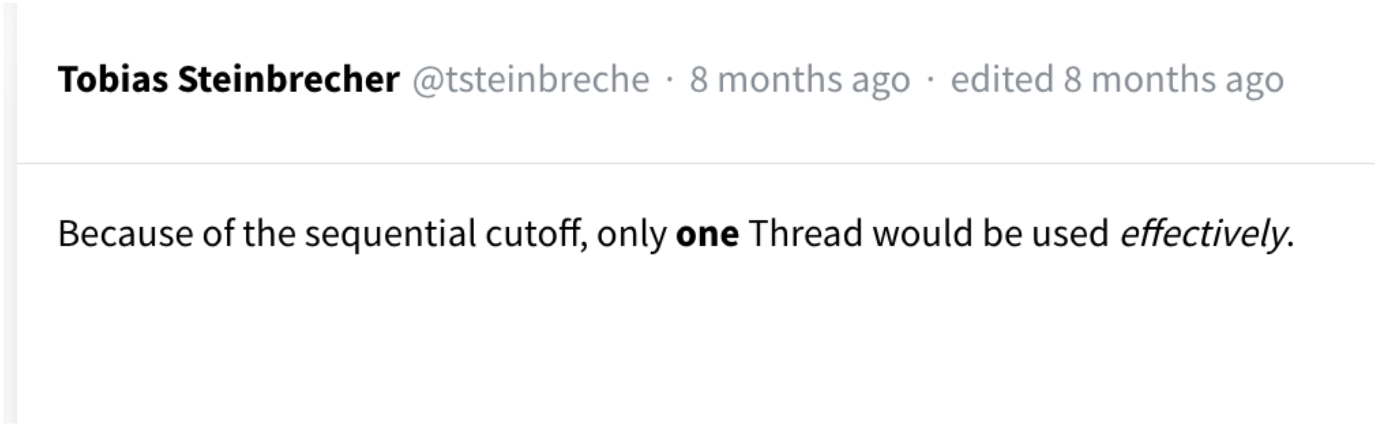
Assuming that the `ImageNegationFJ` class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the 2×2 `negatedImage` array from task 3c)? (2)

```java
double[][] image = {{0,1}, {1,0}};
```



```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```java
final static int CUTOFF = 32;
```



Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

Because of the sequential cutoff, only **one** Thread would be used *effectively*.

(e) Gehen Sie von einem konstanten Overhead von $16\,\text{MB} = 2^4\,\text{MB}$ pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe $10\,\text{GB}$ der Speicher ausgeht? Hinweis: $1\,\text{GB} = 2^{10}\,\text{MB}$.

Assume a fixed overhead of $16\,\text{MB} = 2^4\,\text{MB}$ per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for `CUTOFF` if you input an image of size 4000×4000 before you run out of memory using a RAM of size $10\,\text{GB}$? Hint: $1\,\text{GB} = 2^{10}\,\text{MB}$. (4)

(e) Gehen Sie von einem konstanten Overhead von 16 MB = 2^4 MB pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe 10 GB der Speicher ausgeht? Hinweis: 1 GB = 2^{10} MB.

Assume a fixed overhead of 16 MB = 2^4 MB per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for `CUTOFF` *if you input an image of size 4000×4000 before you run out of memory using a RAM of size 10 GB? Hint: 1 GB = 2^{10} MB.* (4)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago 14

Number of threads, which we can use:

$$N = \frac{10 \cdot 2^{10}}{2^4} = 10 \cdot 2^6 = 10 \cdot 4^3$$

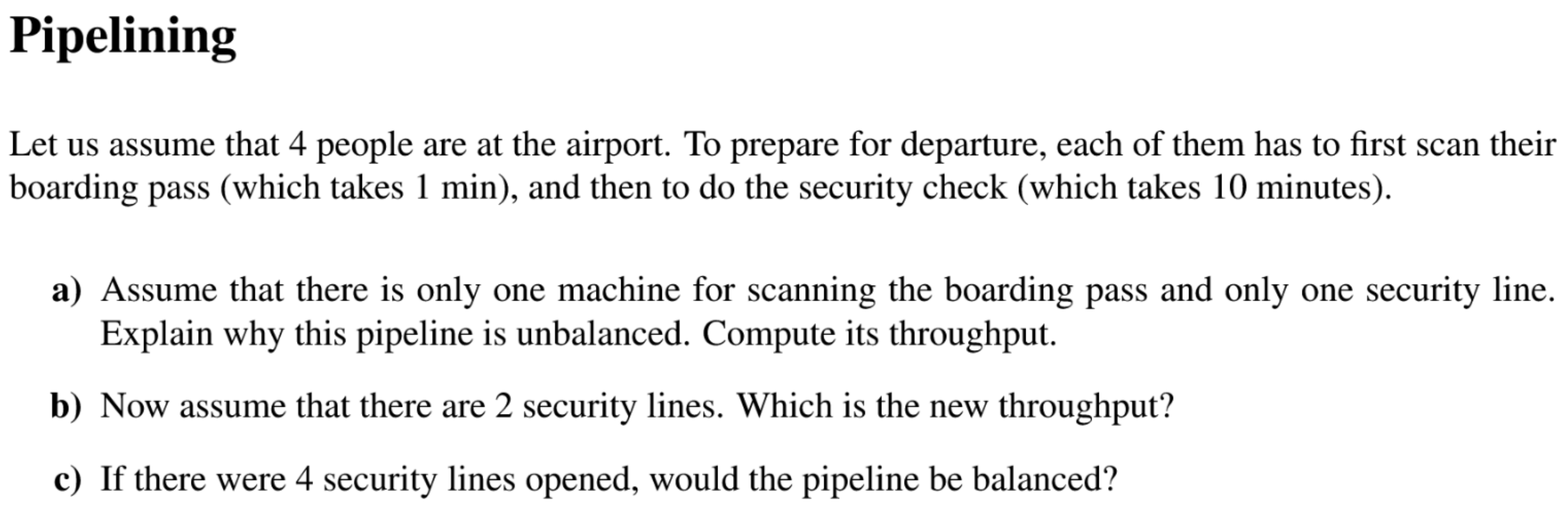
In each recursive call, we will use 4 new threads (under given assumptions). Thereby, we have the constraint ($i :=$ number of divisions)

$$4^i \leq 10 \cdot 4^3 \iff i \leq \log_4(10) + 3 \iff i \leq 4$$

and the smallest possible value is CUTOFF $= 4000/2^4 = \mathbf{250}$ to avoid a fifth division.

+ Add Comment ··· More

Extra Tasks



Pipelining

Let us assume that 4 people are at the airport. To prepare for departure, each of them has to first scan their boarding pass (which takes 1 min), and then to do the security check (which takes 10 minutes).

a) Assume that there is only one machine for scanning the boarding pass and only one security line. Explain why this pipeline is unbalanced. Compute its throughput.

b) Now assume that there are 2 security lines. Which is the new throughput?

c) If there were 4 security lines opened, would the pipeline be balanced?

Pipelining

Let us assume that 4 people are at the airport. To prepare for departure, each of them has to first scan their boarding pass (which takes 1 min), and then to do the security check (which takes 10 minutes).

a) Assume that there is only one machine for scanning the boarding pass and only one security line. Explain why this pipeline is unbalanced. Compute its throughput.

b) Now assume that there are 2 security lines. Which is the new throughput?

c) If there were 4 security lines opened, would the pipeline be balanced?

Solution

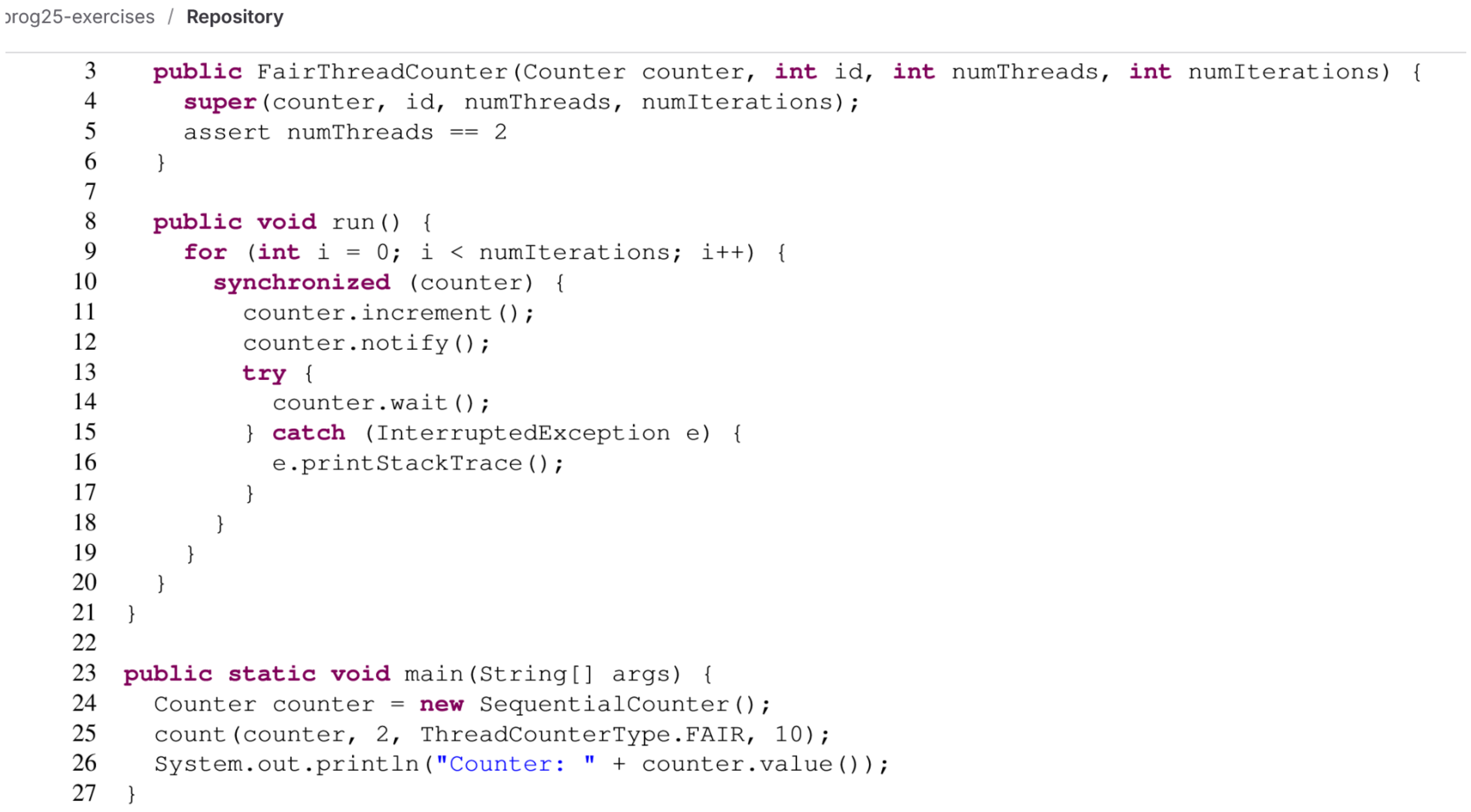
a) The pipeline is unbalanced, because the latency is not constant. Person 1 has a latency of 11 minutes, whereas person 2 has latency of 20 minutes – i) scan boarding pass (1 min), ii) wait for the first person to finish security check (9 min), iii) pass trough security check (10 min). The throughput is 1 person per 10 minutes.

b) The new throughput is 2 persons per 10 minutes. Note that even though the pipeline is unbalanced, the throughput is constant.

c) No. For the first 4 people the latency will be constant, but the 5th one would still have to wait. A pipeline is balanced only when the latency is constant for all its inputs.

Wait and Notify

Consider the following implementation of a FairThreadCounter which implements the Round Robin policy for 2 threads (as described in exercise 3).



rog25-exercises / **Repository**

```
  public FairThreadCounter(Counter counter, int id, int numThreads, int numIterations) {
    super(counter, id, numThreads, numIterations);
    assert numThreads == 2
  }

  public void run() {
    for (int i = 0; i < numIterations; i++) {
      synchronized (counter) {
        counter.increment();
        counter.notify();
        try {
          counter.wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
}

public static void main(String[] args) {
  Counter counter = new SequentialCounter();
  count(counter, 2, ThreadCounterType.FAIR, 10);
  System.out.println("Counter: " + counter.value());
}
```

a) What will be printed in the console after running the program?

b) Does the solution behave as expected? If not, explain why and fix the errors.

a) What will be printed in the console after running the program?

b) Does the solution behave as expected? If not, explain why and fix the errors.

Solution

a) Nothing – the program does not terminate, so nothing will be printed. The problem is that in the last iteration, the second thread will be stuck at line 14 waiting to be notified. As the first thread already finished incrementing, the second thread will never be notified.

a) What will be printed in the console after running the program?

b) Does the solution behave as expected? If not, explain why and fix the errors.

Solution

b) There are several mistakes in this solution:

- *Non-deterministic thread order.* Even though both threads increment within a synchronized block, which one starts is not specified. To fix this one should first check whether the thread is supposed to increment (and `wait` if necessary) and increment only after this condition is true.
- *Unhandled spurious wakeups.* After calling `wait` the counter does not check whether it was woken up spuriously or it is indeed its turn to perform the increment.
- *Non-termination.* Even without spurious wake-ups, the program will not terminate as in the last iteration, the second thread will be stuck at line 14 waiting to be notified. A tempting solution is to put an additional `counter.notify()` statement after the for loop. However, this does not solve the problem as there is not guarantee that the notify will be executed after the second thread called `wait`. To fix this issue properly, the code needs to be changed such that it includes an explicit condition that denotes whether the thread should wait or continue. This can be a boolean flag (if there are only two threads) or a thread id as used in exercise 3.

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- Intro Exercise 6
- Exam Questions
- **Kahoot**

Kahoot!!

- omg

QUIZ

Feedback

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

Danke

- Bis nächste Woche!