Parallele Programmierung FS25

Exercise Session 3

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Exercise 2
- Theory Recap
- Intro Exercise 3
- 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/b8nw9v32ChDcN3XR9

Organisation

- Feedback zur Session: https://forms.gle/b8nw9v32ChDcN3XR9
- Falls ihr Feedback möchtet kommt bitte zu mir

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)

Plan für heute

- Organisation
- **Nachbesprechung Exercise 2**
- Theory Recap
- Intro Exercise 3
- Exam Questions
- Kahoot

Common Mistakes Exercise 2

Task A

Using Thread as Runnable

```
Thread a2 = new newThread2();
Thread aa = new Thread(a2);
```

or

```
Thread aa = new Thread(new newThread2());
```

Common Mistakes Exercise 2

Forgetting to join

```
public static long taskC() {
        long startTime = System.nanoTime();
        Thread t = new Thread(new Runnable() {
                @Override
                public void run() {}
        });
        t.start();
        long endTime = System.nanoTime();
        return endTime - startTime;
}
```

Task C: Thread with no Task

```
public static class EmptyTask implements Runnable{

  @Override
  public void run() {}

}

public static long taskC() {
  long start = System.nanoTime();
  Thread t = new Thread(new EmptyTask());
  t.start();
  long end = System.nanoTime();
  return (end-start);
 }
```

Task C: Thread with no Task

```
public static long taskC() {
  long start = System.nanoTime();
  Thread t = new Thread(); //no Task!
  t.start();
  long end = System.nanoTime();
  return (end-start);
 }
```

Common Mistakes Exercise 2

Task E

Unnecessary input copy

All the threads can operate on the same input array since it's read only

Forgetting to join

All the threads were started, but never joined.

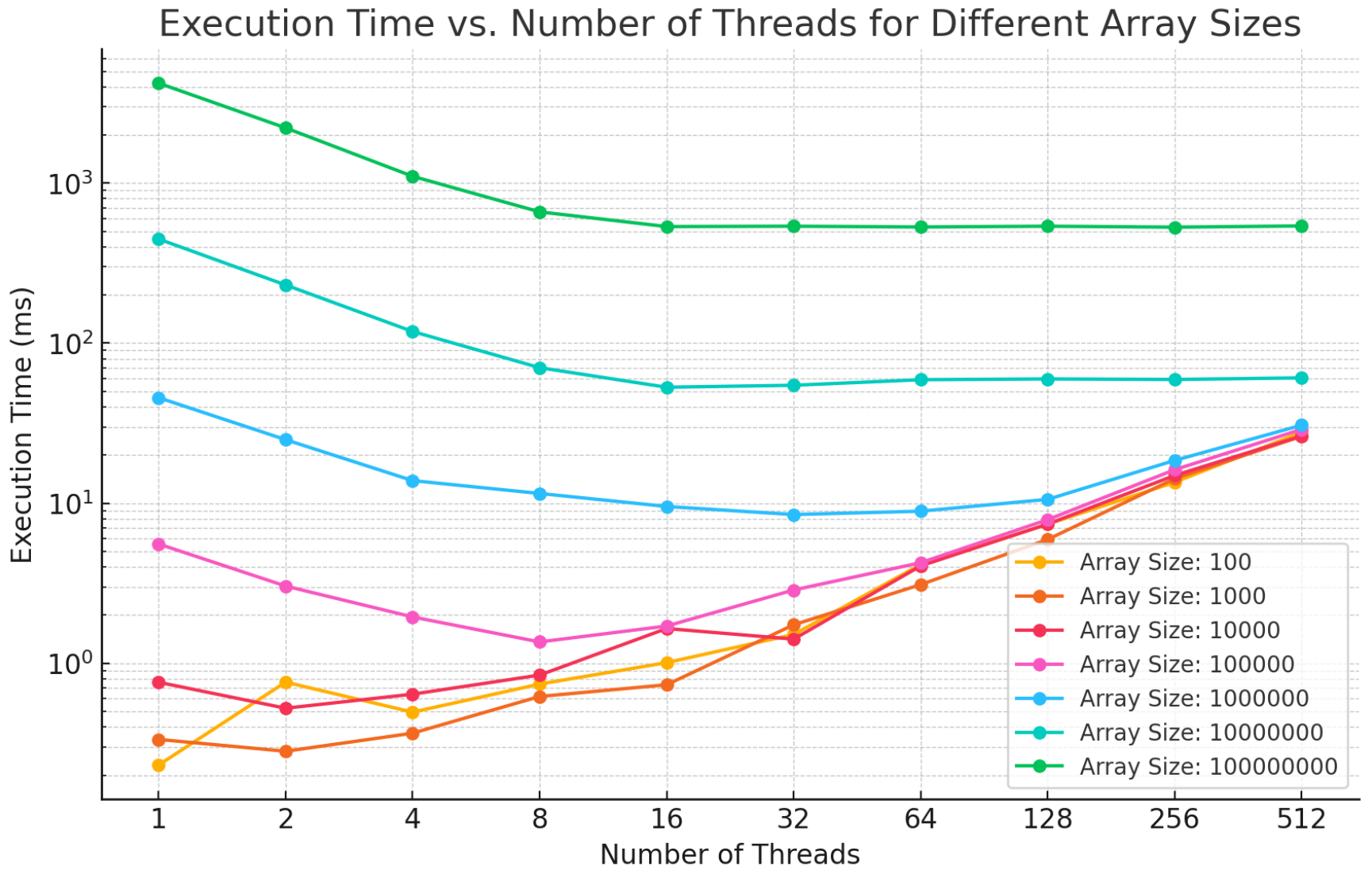
Solving the task sequentially

```
for (int i = 0; i < numThreads; i++) {
    Thread t = new MyThread(...);
    t.start();
    try {
        t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

Common Mistakes Exercise 2

- code ex2Sol

Benchmark



16 cores available

Benchmark

- For small arrays, increasing threads does not improve performance and may even degrade it due to thread management overhead
- For larger arrays, execution time decreases significantly up to a certain number of threads, beyond which performance gains diminish
- At very high thread counts, overhead dominates, causing execution time to increase

Benchmark

- We don’t want to create many threads due to overhead
- Solution is the ExecutorService which handles a fixed number of threads
- See code example

Why Use Thread Pools?

- Efficient Resource Management: Creating new threads for every task can be expensive. A thread pool manages a fixed number of threads that are reused.
- Task Scheduling: Executors handle scheduling and execution of tasks efficiently.

Task D: PartitionData

Static partitioning vs. other: dynamic, guided, etc.

In real world: use existing libraries. well tested, concise, fast (e.g. parallel streams for Java)

Task E: Sharing Data Across Threads

code `SharedData`

Plan für heute

- Organisation
- Nachbesprechung Exercise 2
- **Theory Recap**
- Intro Exercise 3
- Exam Questions
- Kahoot

Counter

Let’s count the number of times a given event occurs

```
public interface Counter {
  public void increment();
  public int value();
}
```

Counter

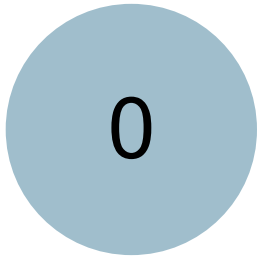
Let’s count the number of times a given event occurs

```
public interface Counter {
  public void increment();
  public int value();
}
```

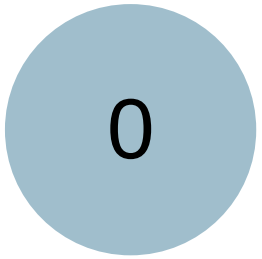
```
// background threads
for (int i = 0; i < numIterations; i++) {
  // perform some work

  counter.increment();
}

// progress thread
while (isWorking) {
  System.out.println(counter.value());
}
```

10 iterations each

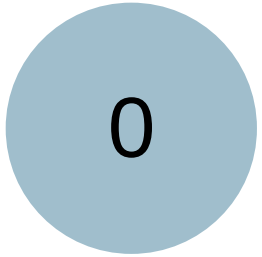
Thread 1



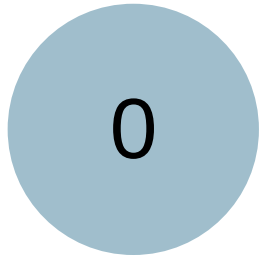
Counter



Thread 2



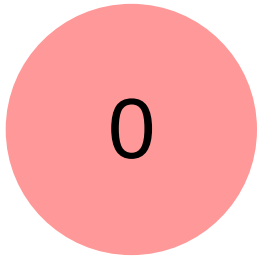
Thread 3



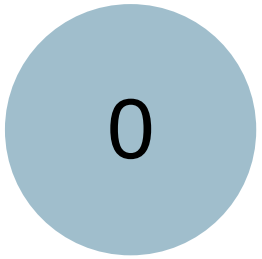
value of the
shared Counter

number of times
`increment()` is called

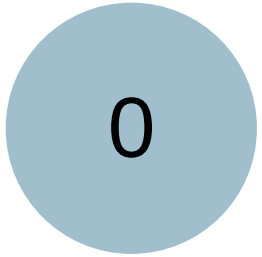
Thread 1



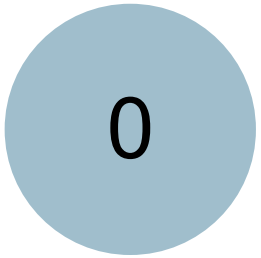
Counter



Thread 2

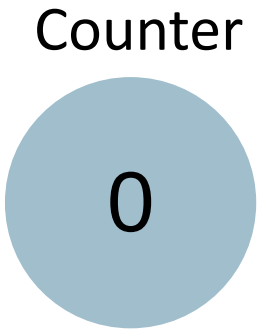


Thread 3

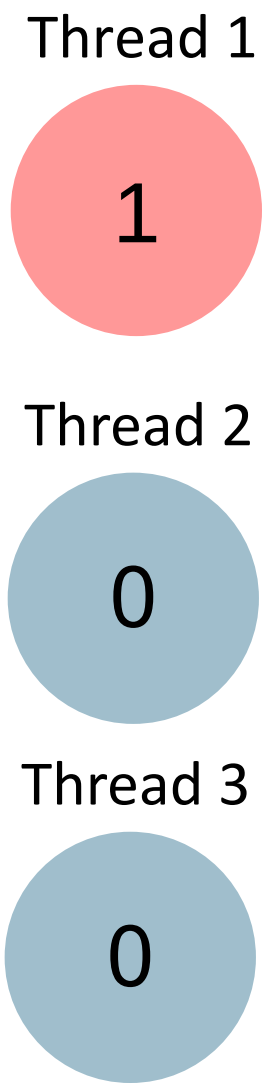


value of the
shared Counter

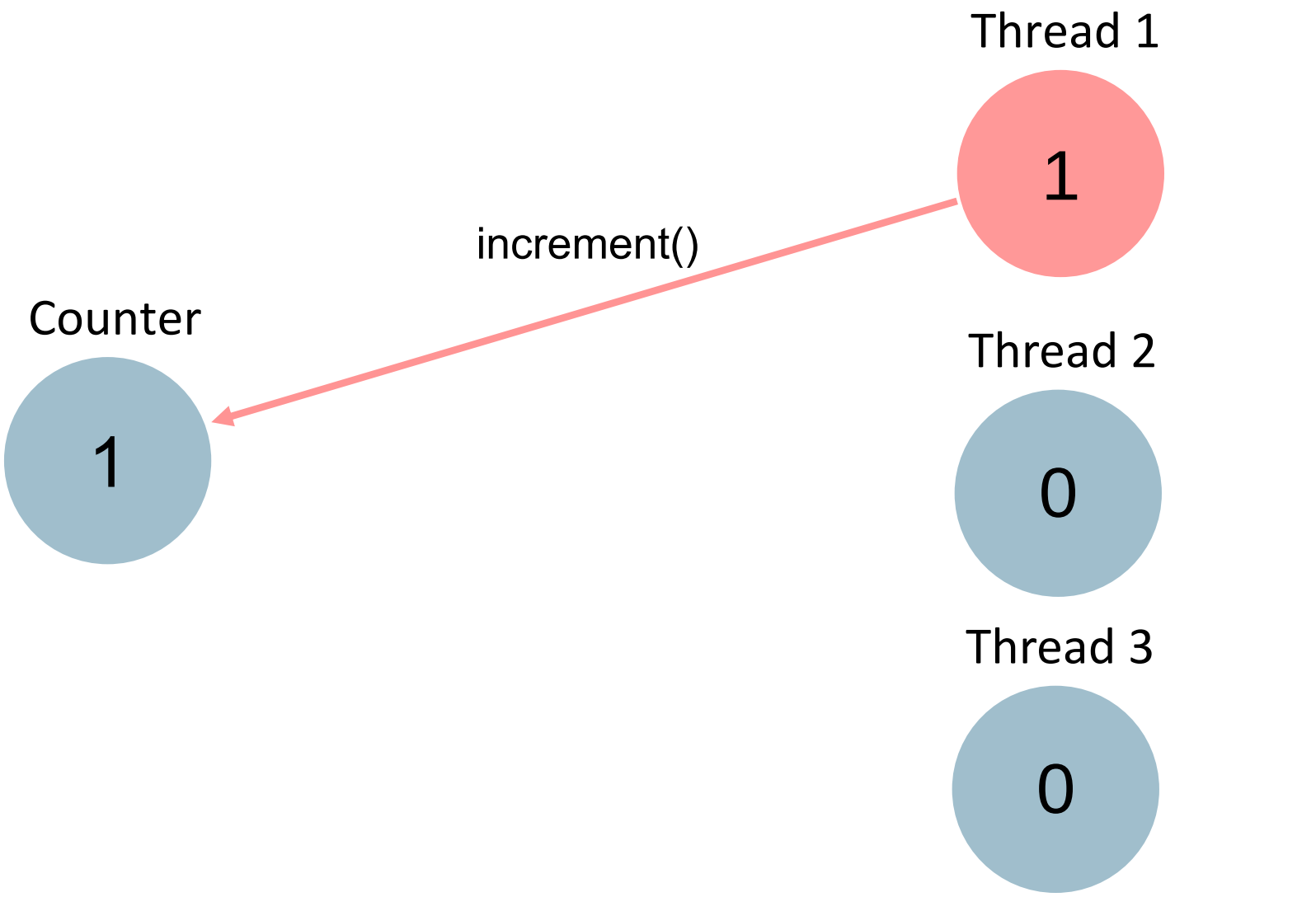
number of times
`increment()` is called



value of the
shared Counter

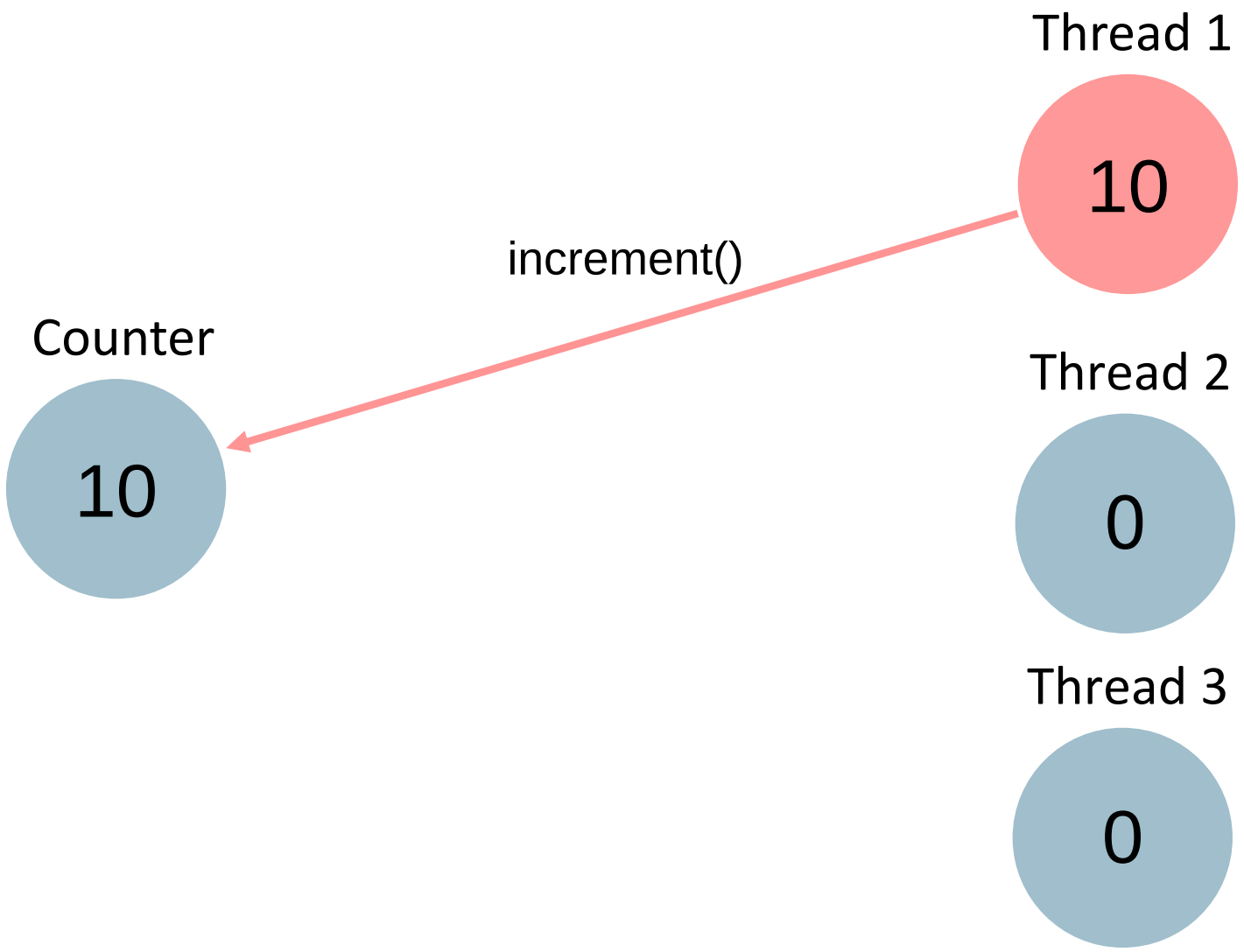


number of times
`increment()` is called



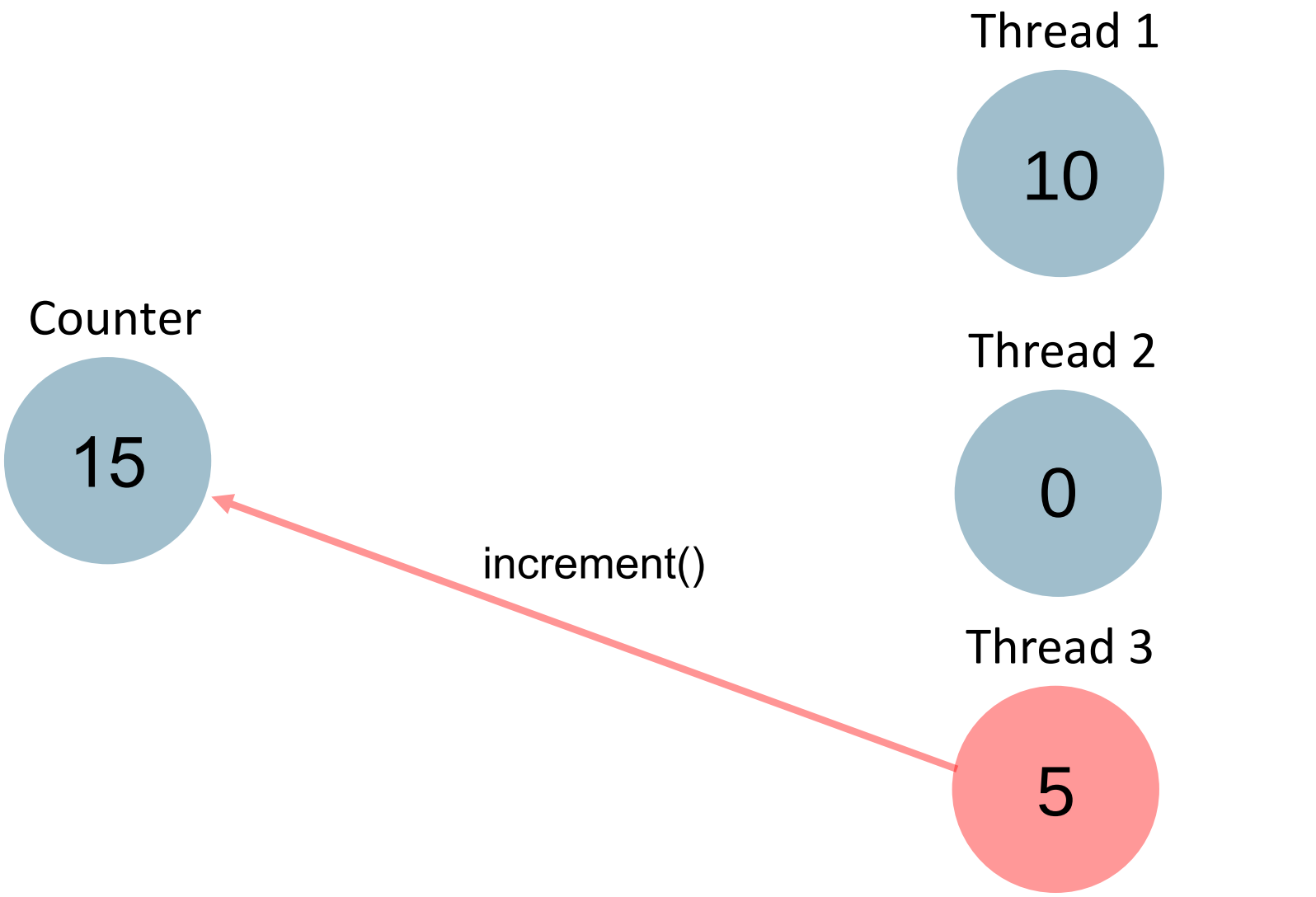
value of the
shared Counter

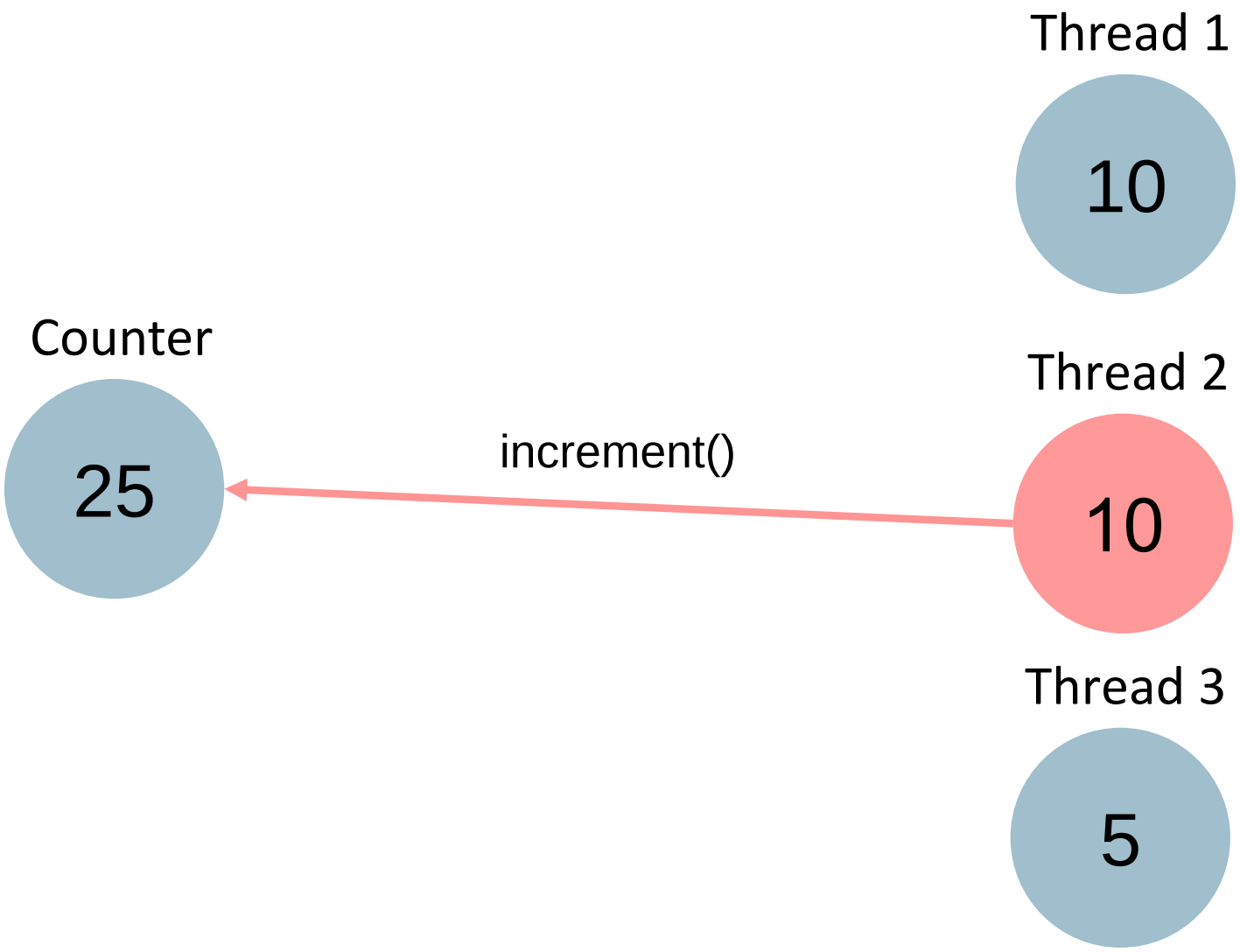
number of times
`increment()` is called



value of the shared Counter

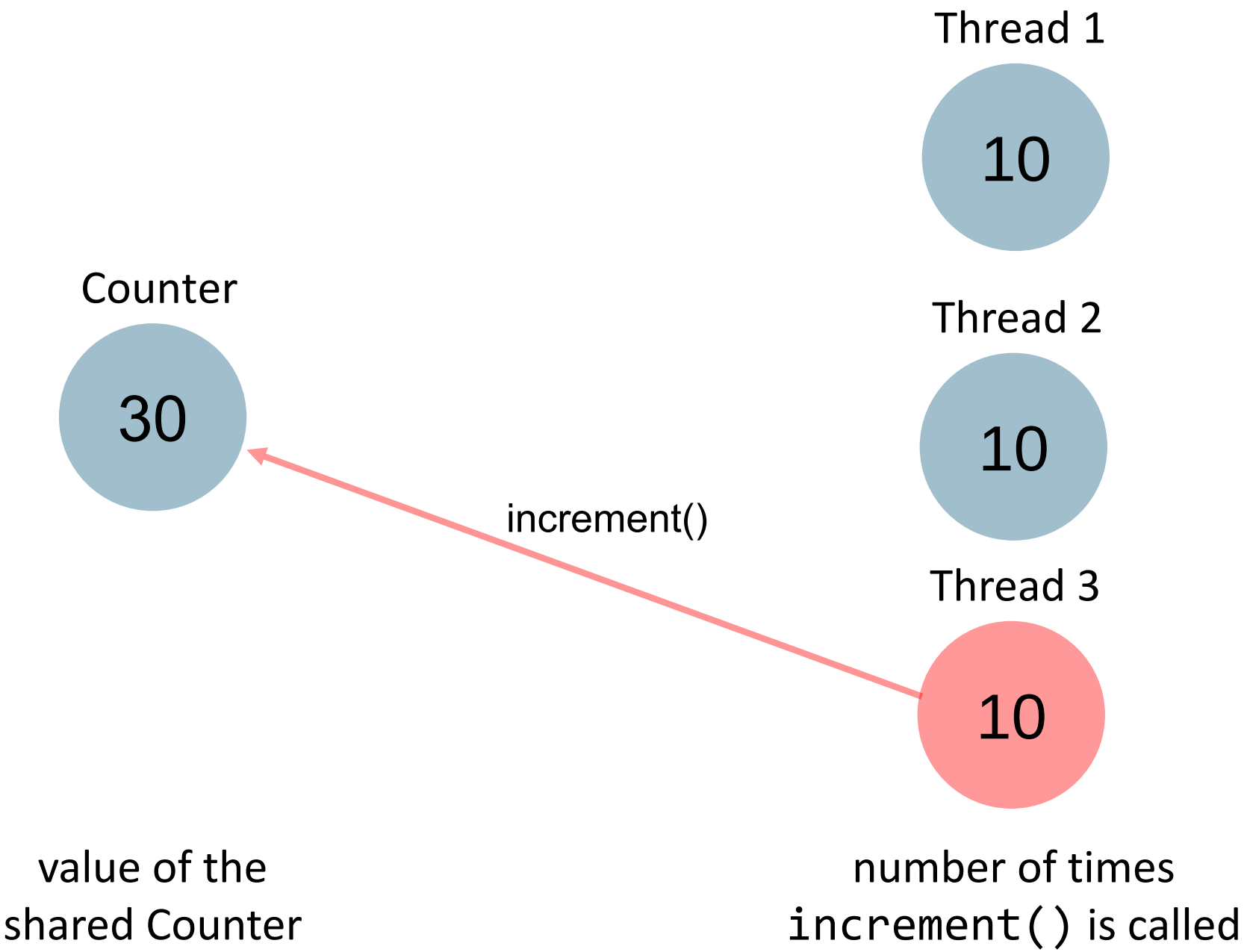
number of times `increment()` is called

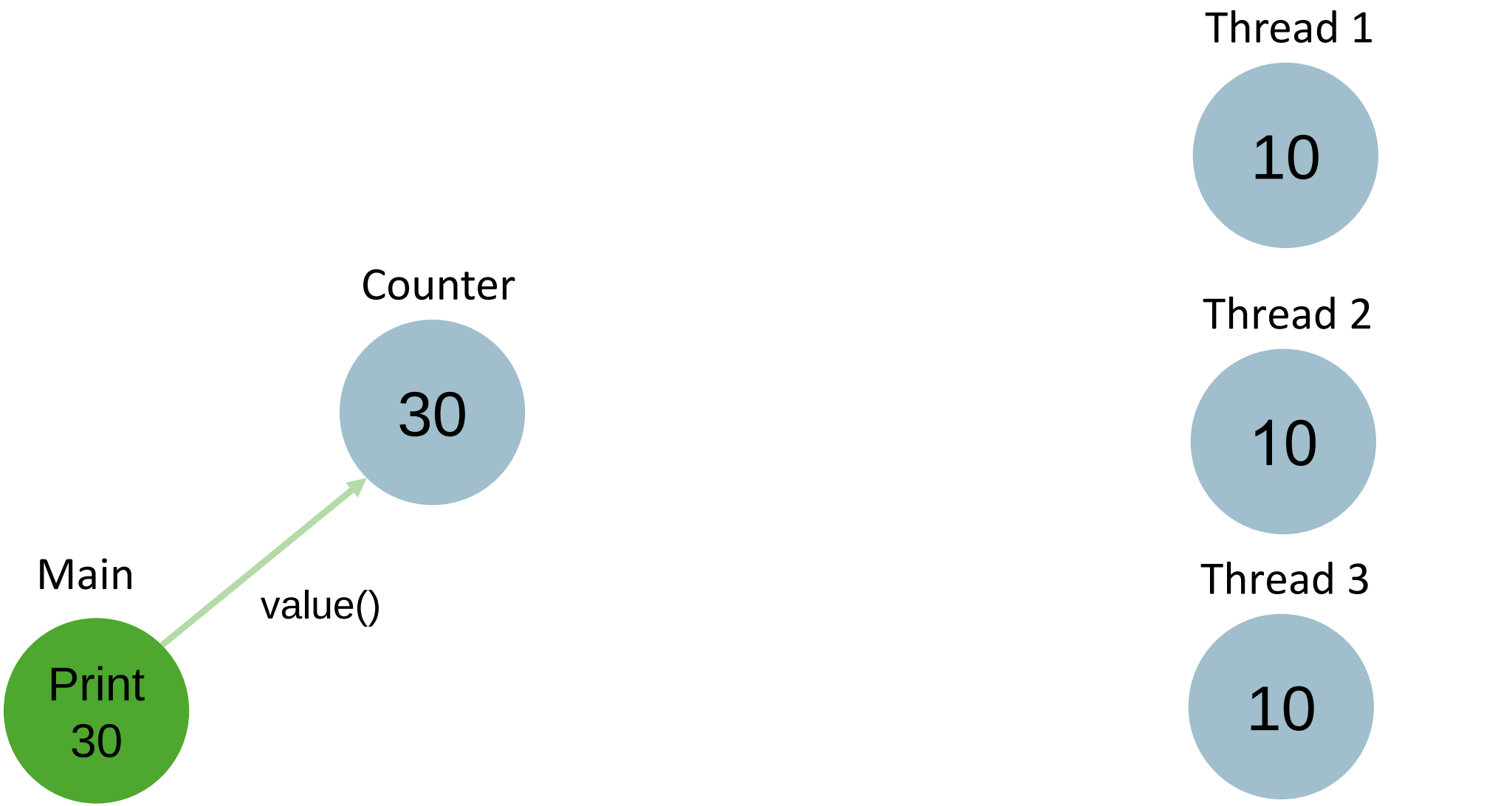

Thread 1
10
Counter
15
Thread 2
0
increment()
Thread 3
5
value of the
shared Counter
number of times
increment() is called



value of the shared Counter

number of times `increment()` is called


Thread 1
10
Counter
30
Thread 2
10
increment()
Thread 3
10
value of the
shared Counter
number of times
increment() is called



read the Counter value

value of the shared Counter

number of times `increment()` is called

Counter

Why will what we just saw probably not work?

Remember: (Bad) Interleavings

Assume we have two threads executing increment() n-times concurrently.

```
public class Counter {
    int count = 0;

    public void increment() {
        count++;
    }
}
```

Synchronization

- ➔ Every reference type contains a lock inherited from the Object class
- ➔ Primitive fields can be locked only via their enclosing objects
- ➔ Locking arrays does not lock their elements
- ➔ A lock is *automatically* acquired when entering and released when exiting a synchronized block
- ➔ ***Locks will be covered in more detail later in the course***

Synchronization

```
public synchronized void xMethod() {
  // method body
}
```

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

➔ Synchronized method locks the object owning the method

```
foo.xMethod() //lock on foo
```

➔ Synchronized keyword obtains a lock on the parameter object

```
synchronized (bar) { … } //lock on bar
```

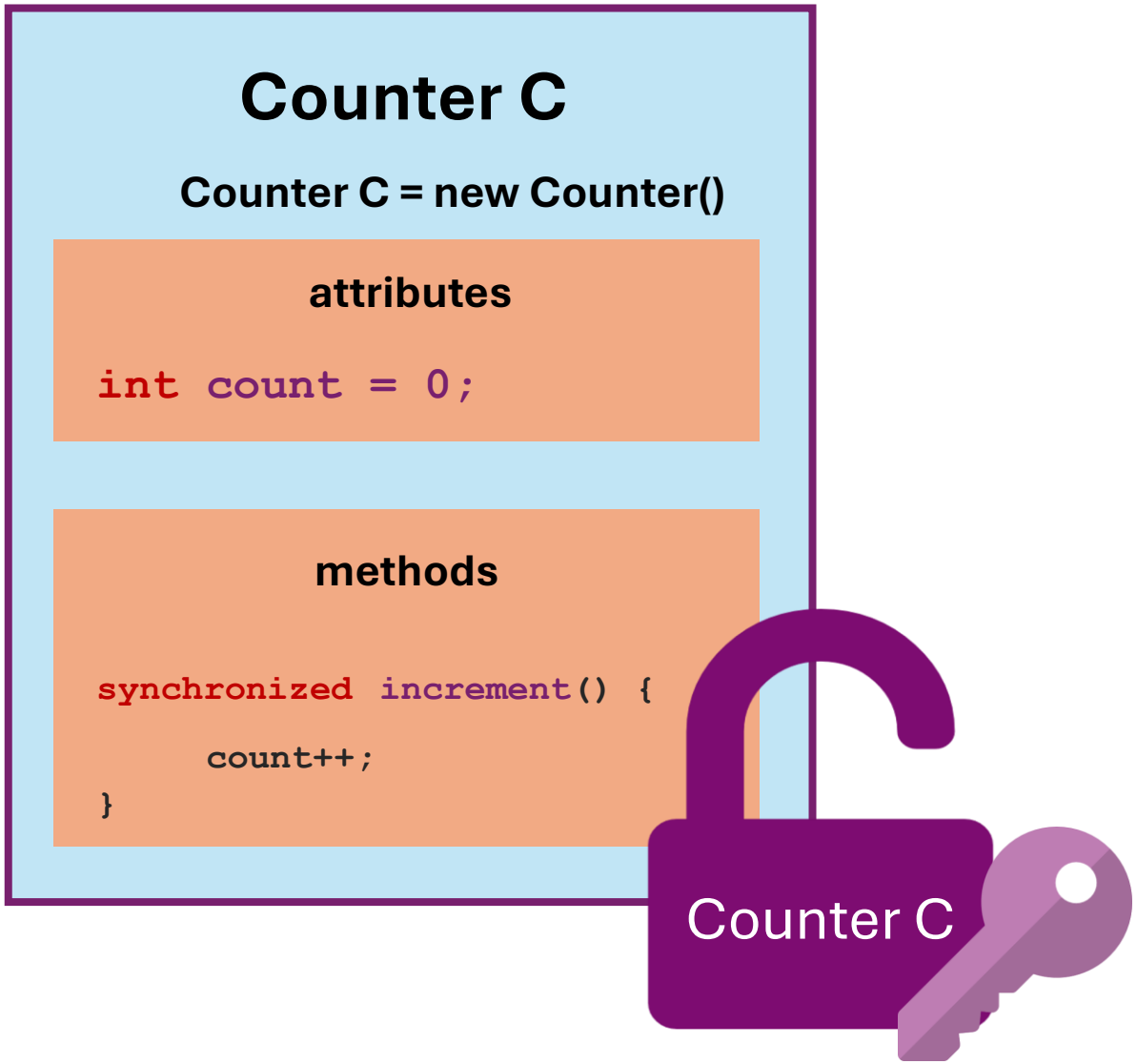
➔ A thread can obtain multiple locks (by nesting the synchronized blocks)

Using `synchronized`

Now only one thread at a time can enter the increment() method ☺

```
public class Counter {
    int count = 0;

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

What exactly is a lock/monitor?



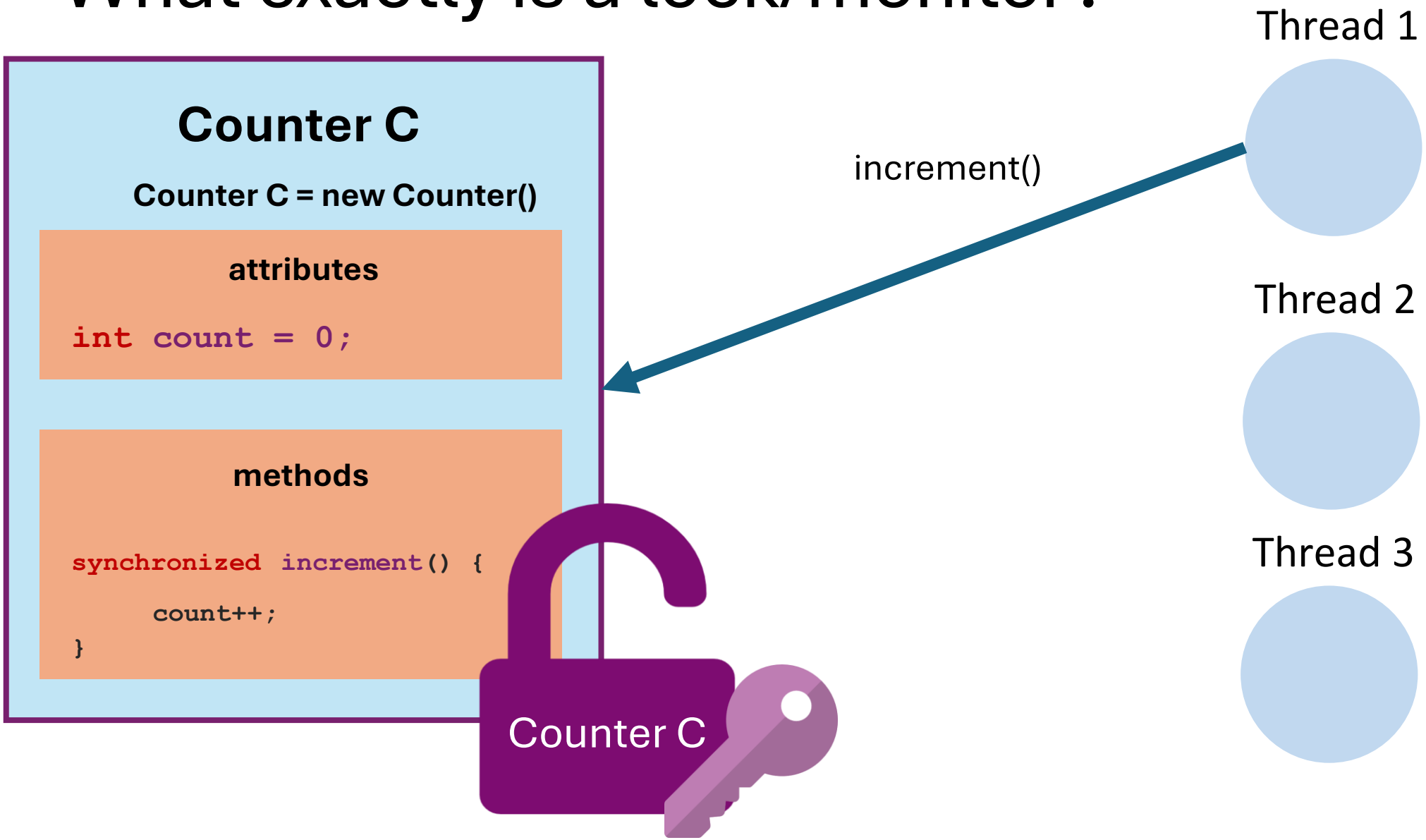
Thread 1

Thread 2

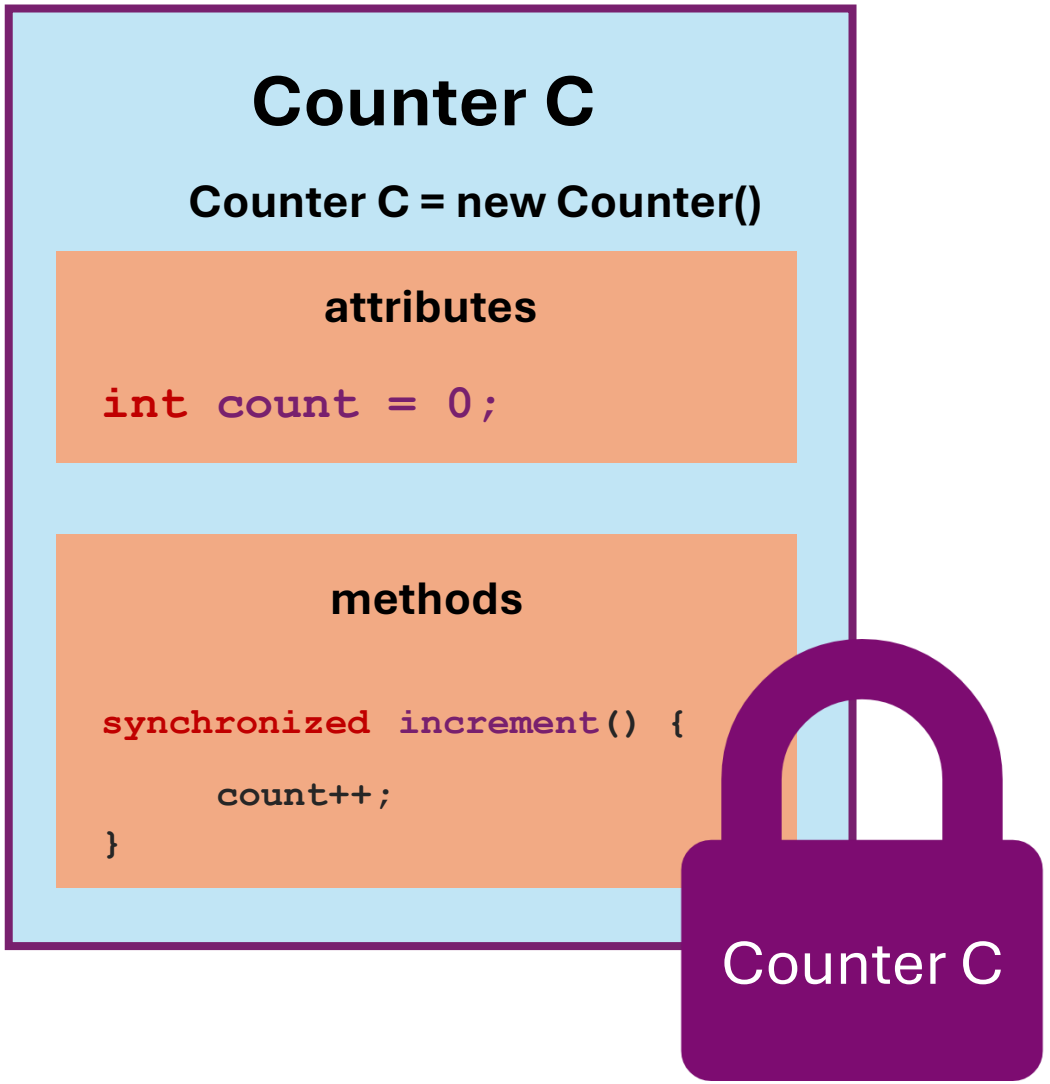
Thread 3

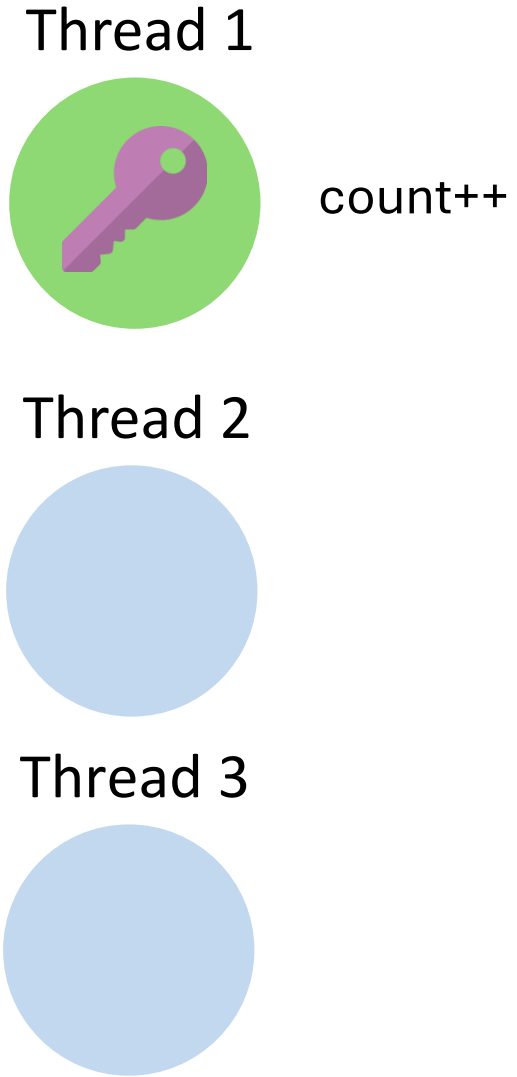
Credits to Gamal Hassan PProg FS24

What exactly is a lock/monitor?

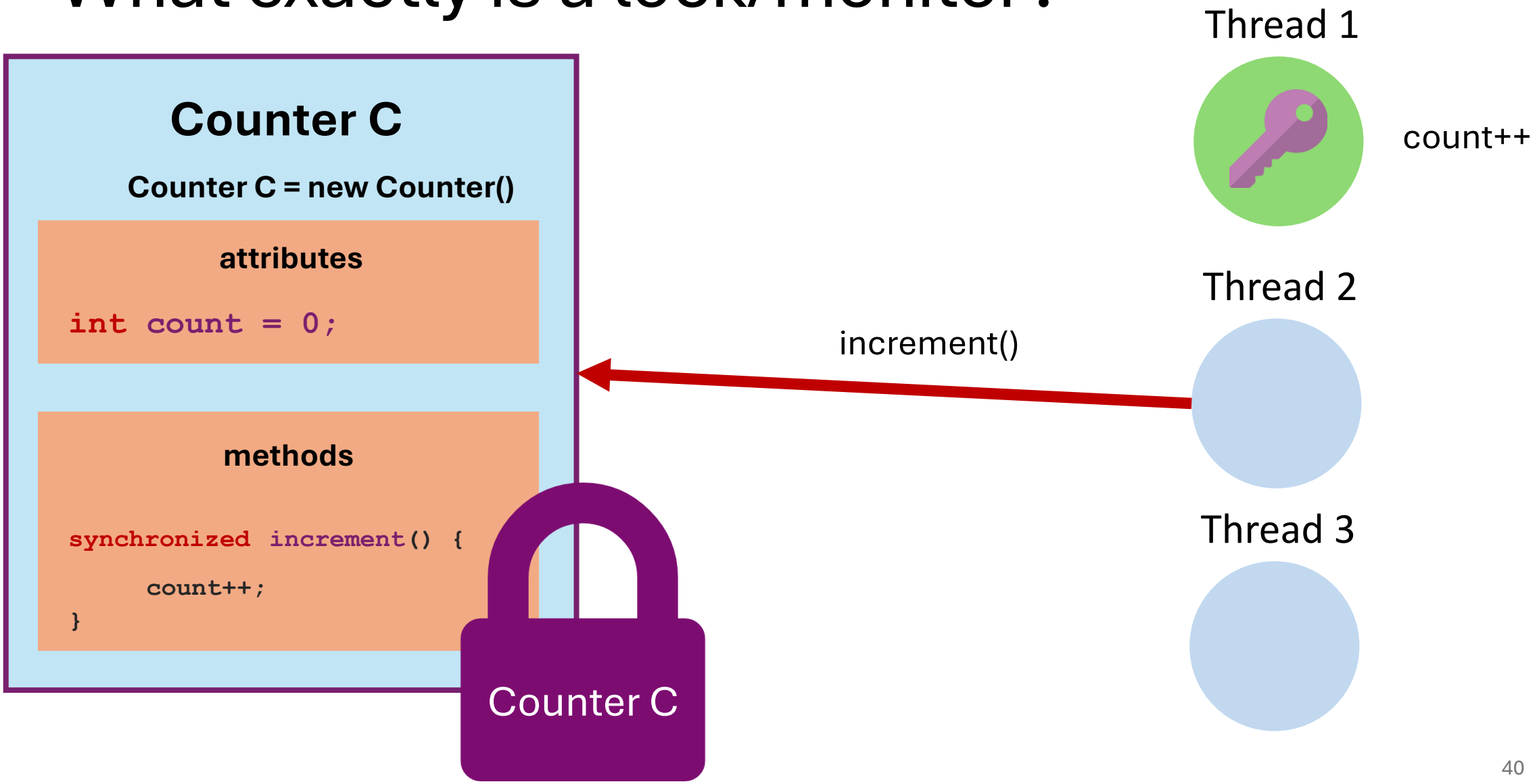


What exactly is a lock/monitor?

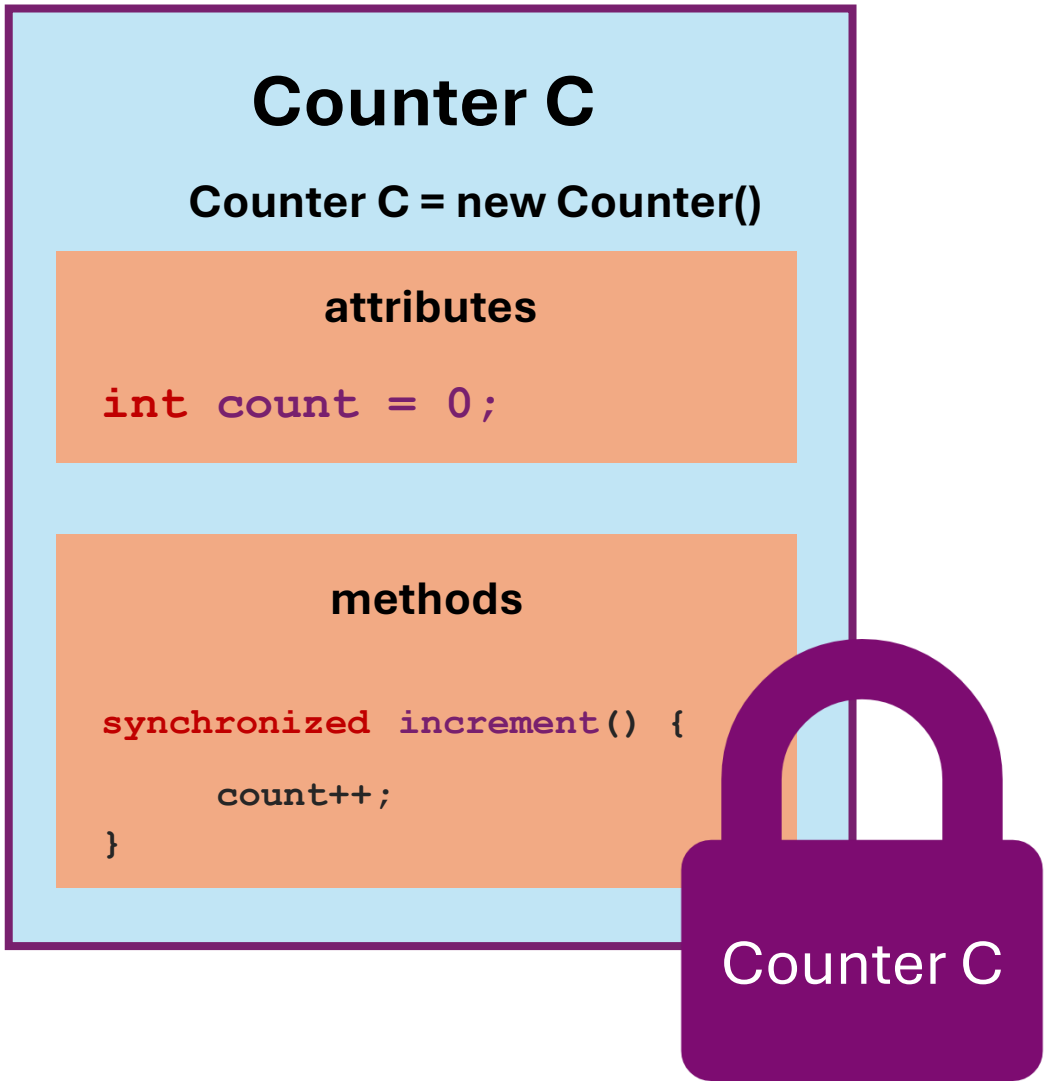


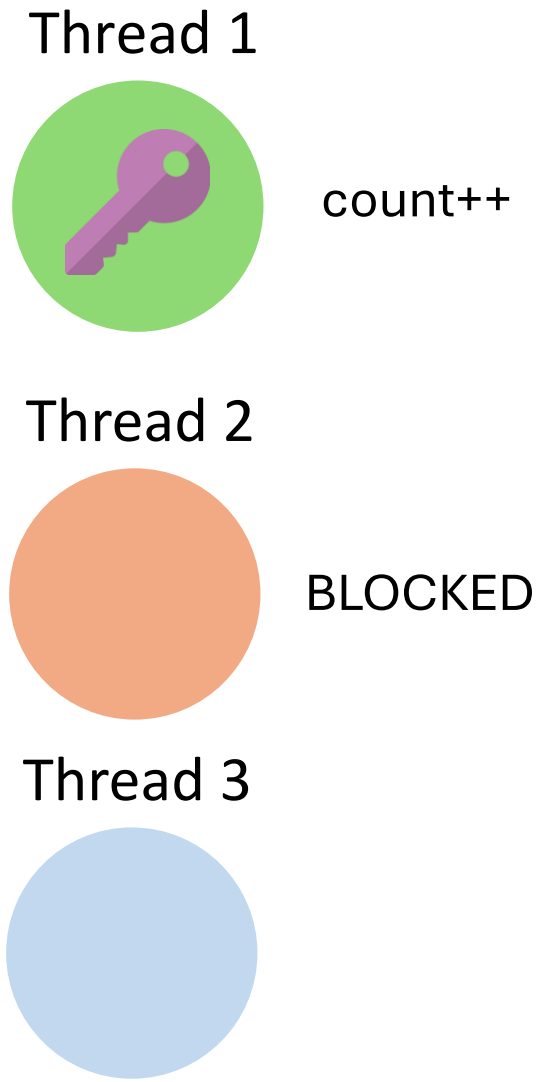


What exactly is a lock/monitor?



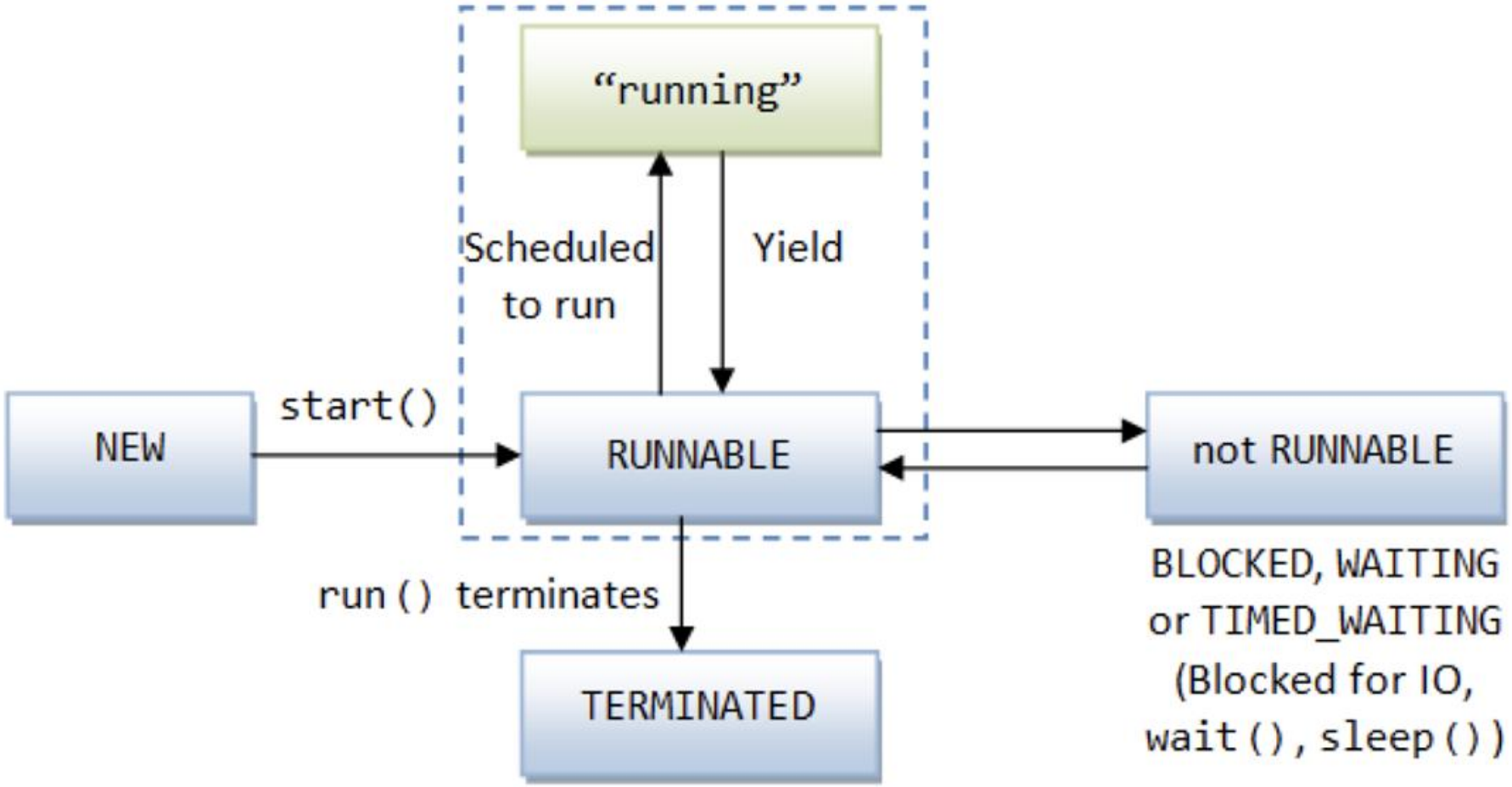
What exactly is a lock/monitor?



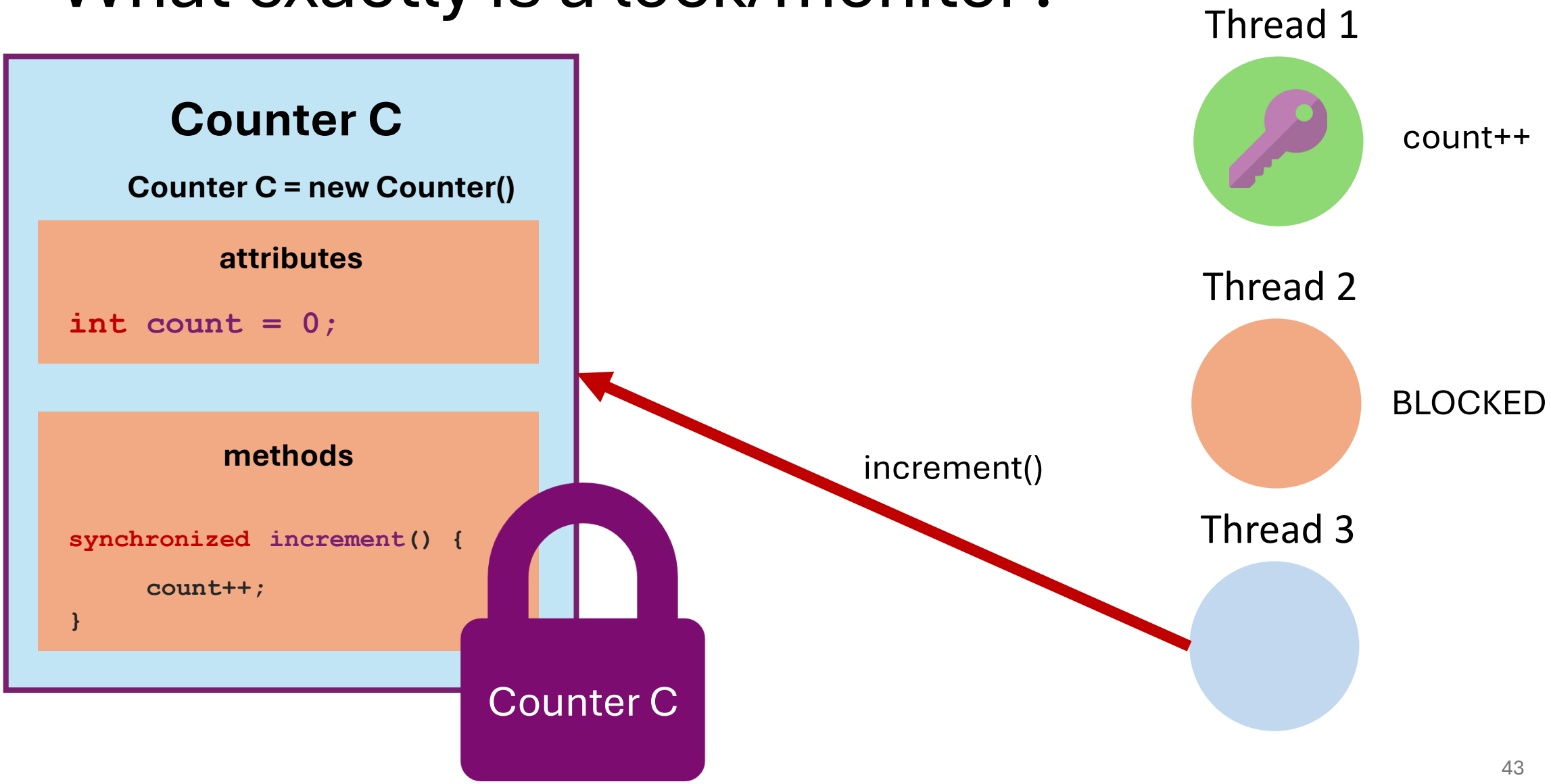


details

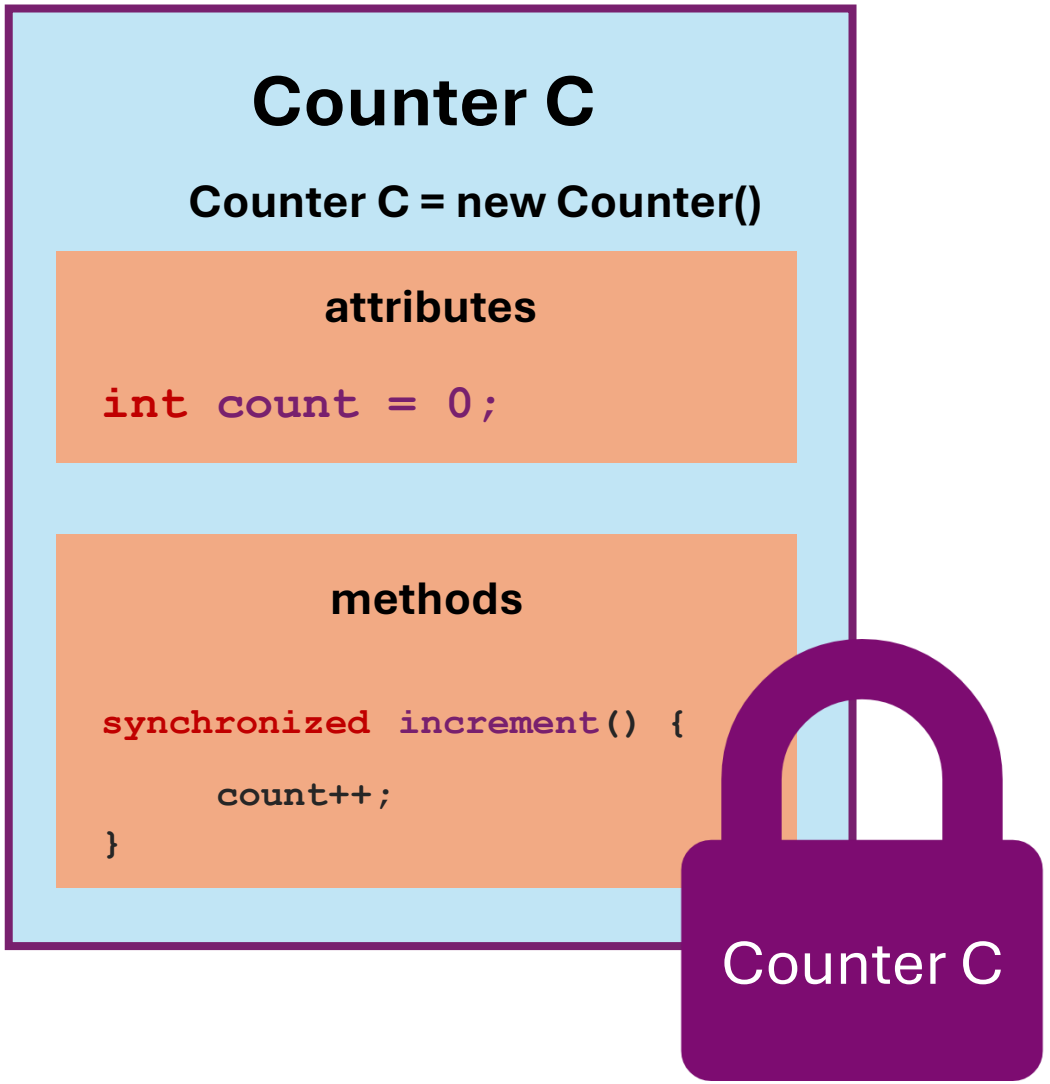
Remember? Thread State Model

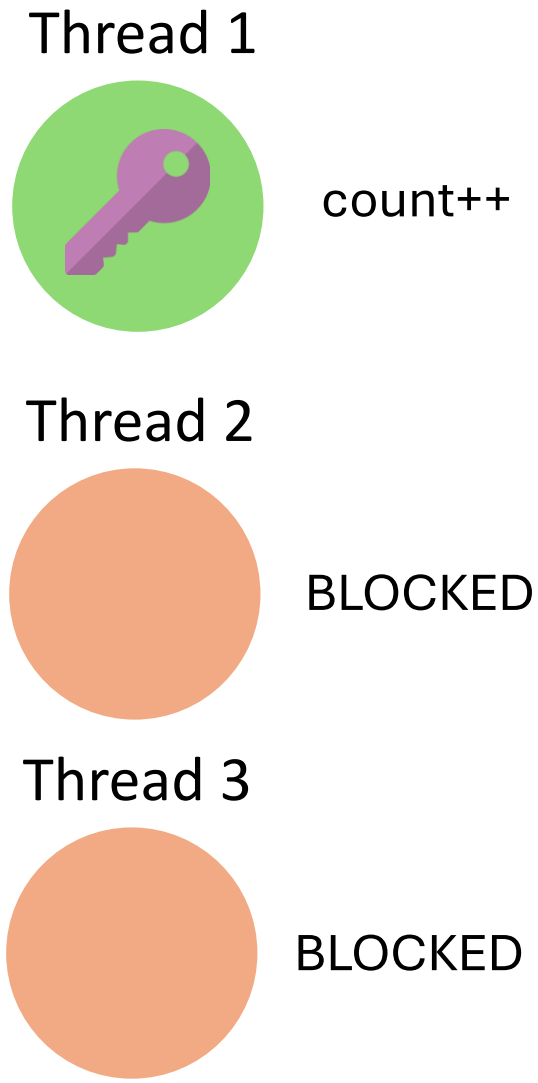


What exactly is a lock/monitor?

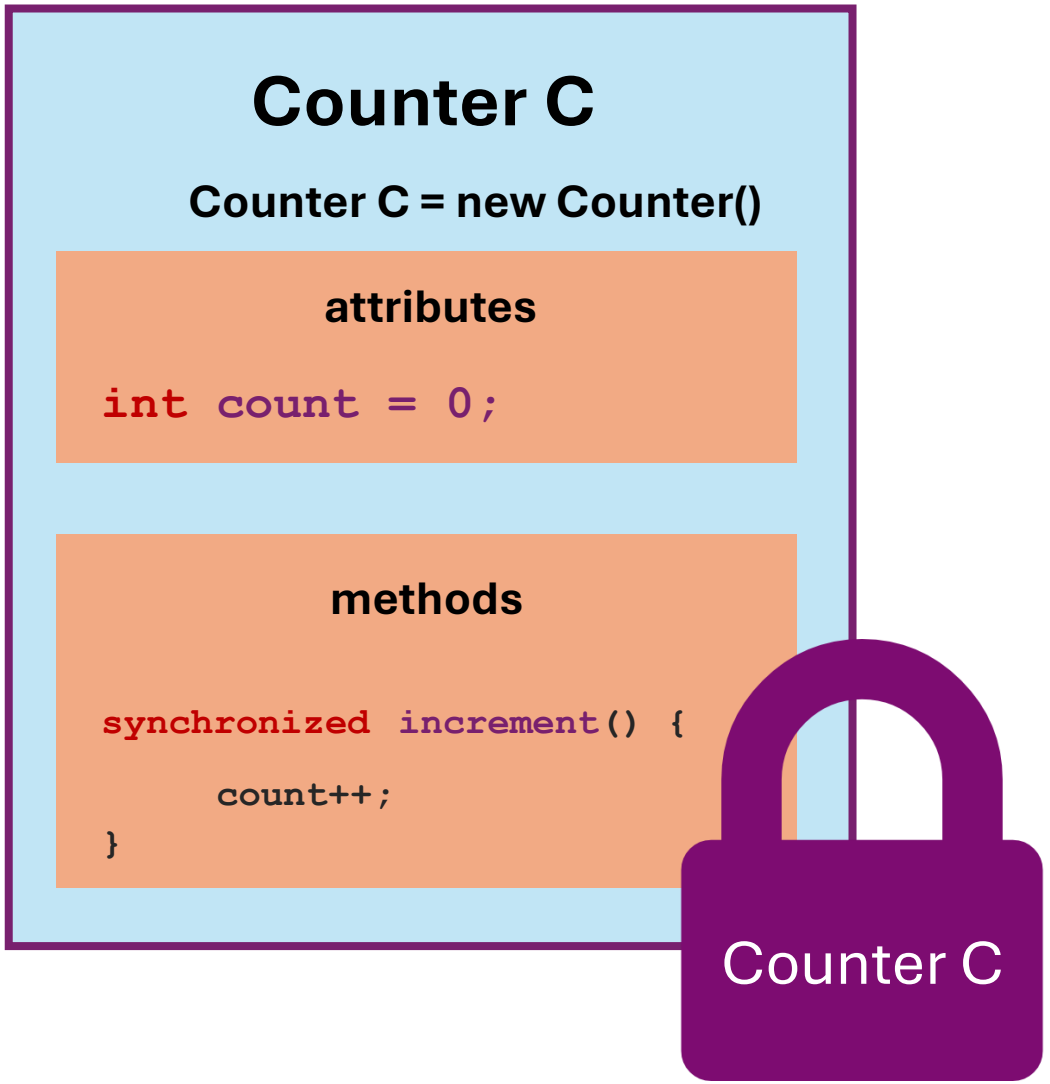


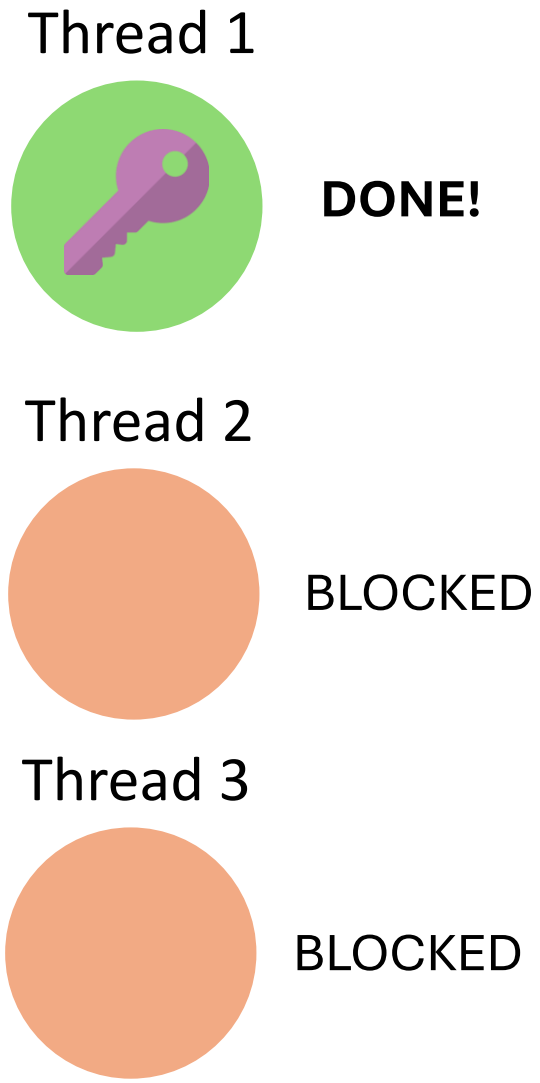
What exactly is a lock/monitor?



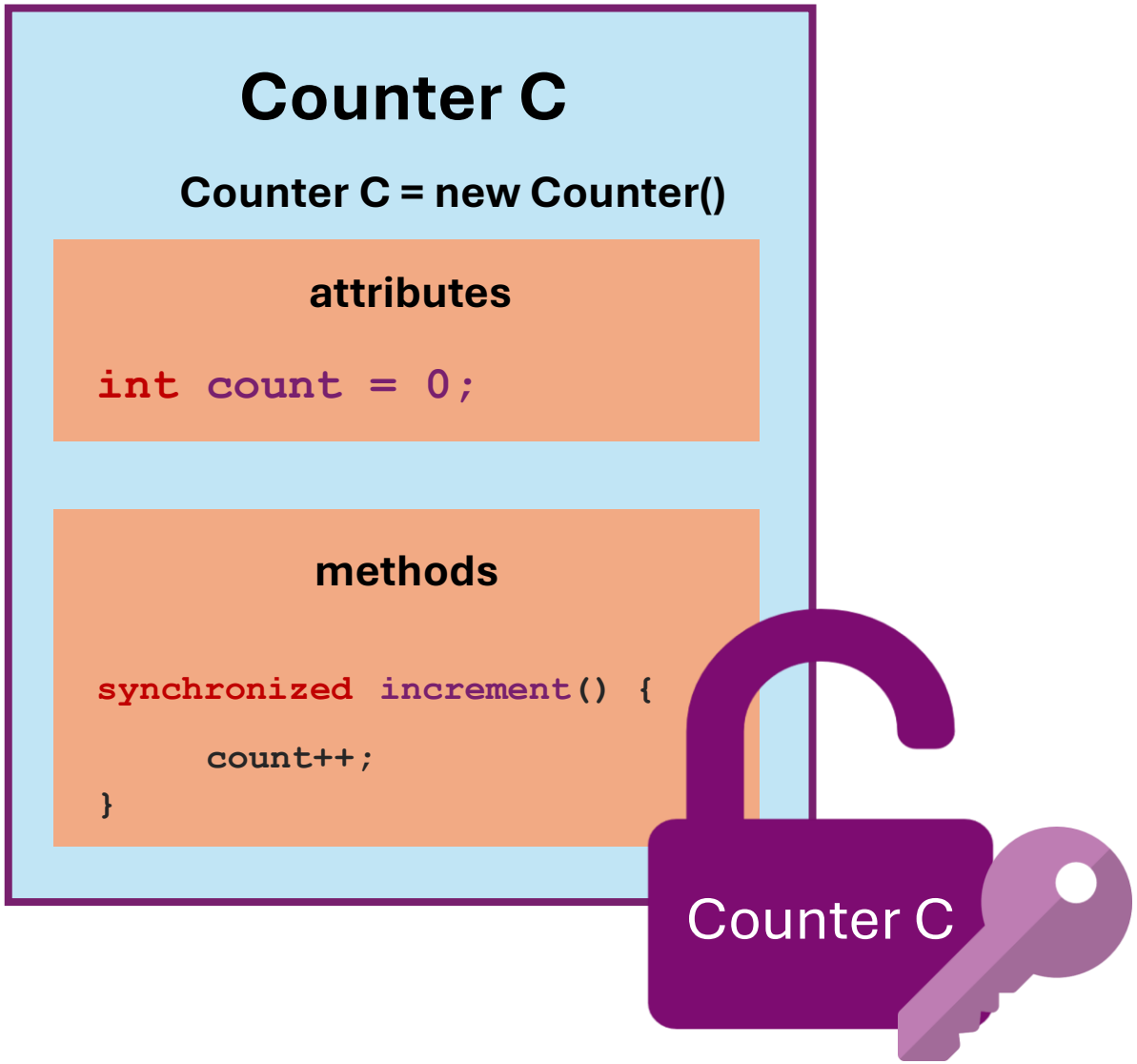


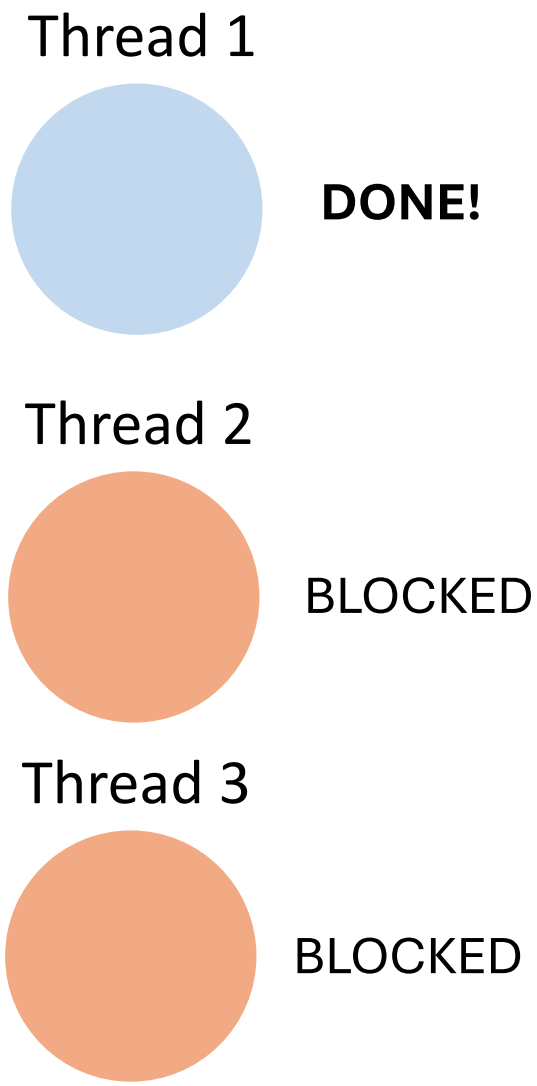
What exactly is a lock/monitor?





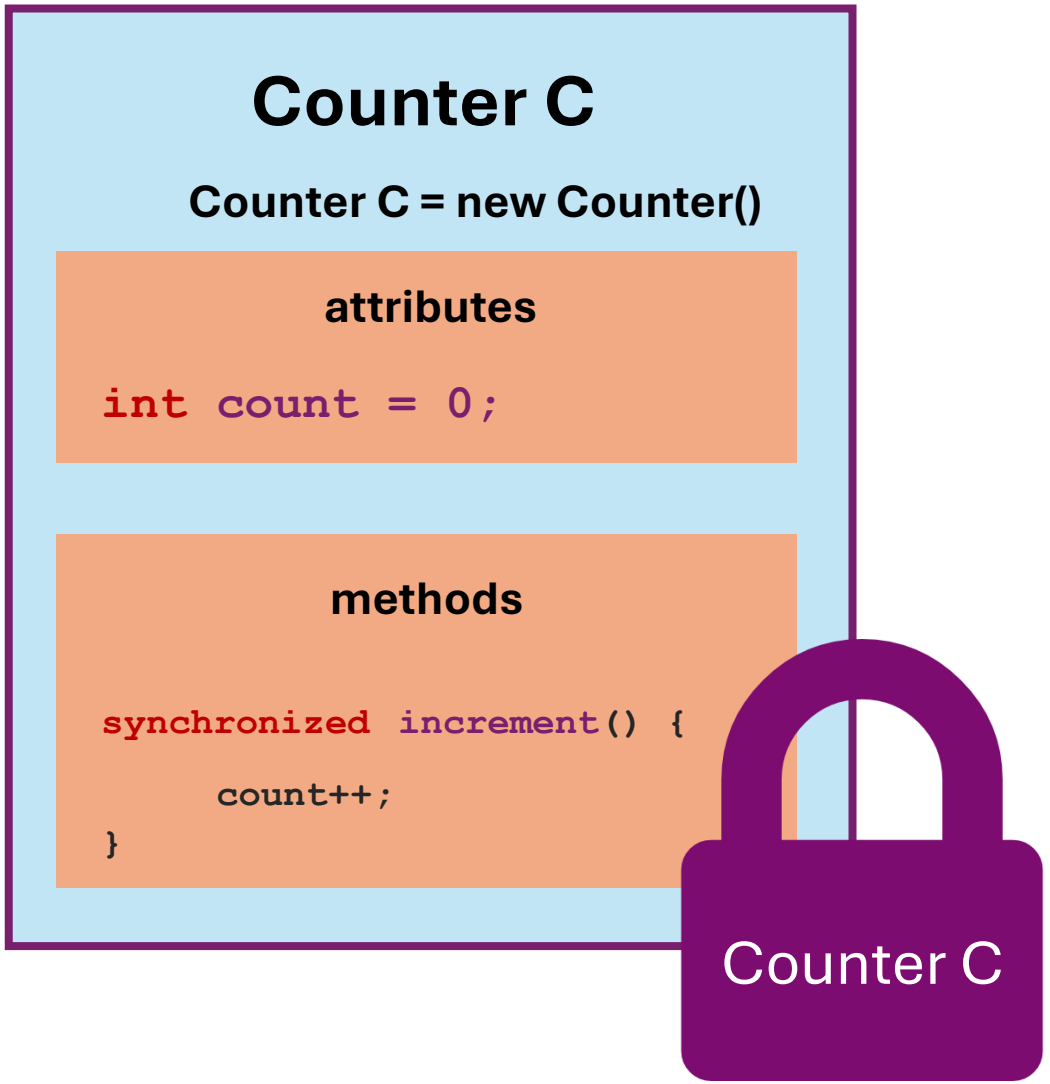
What exactly is a lock/monitor?

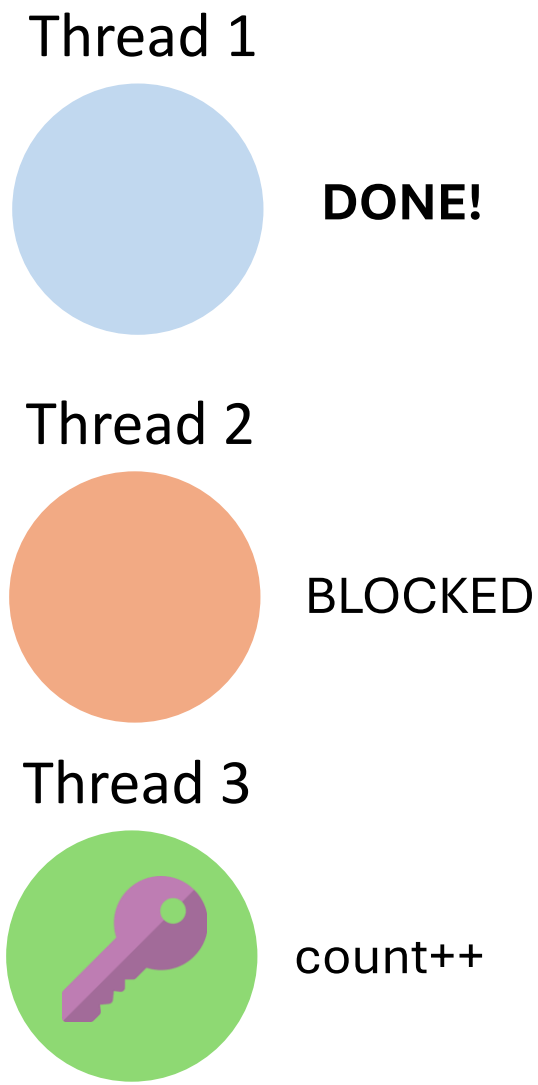




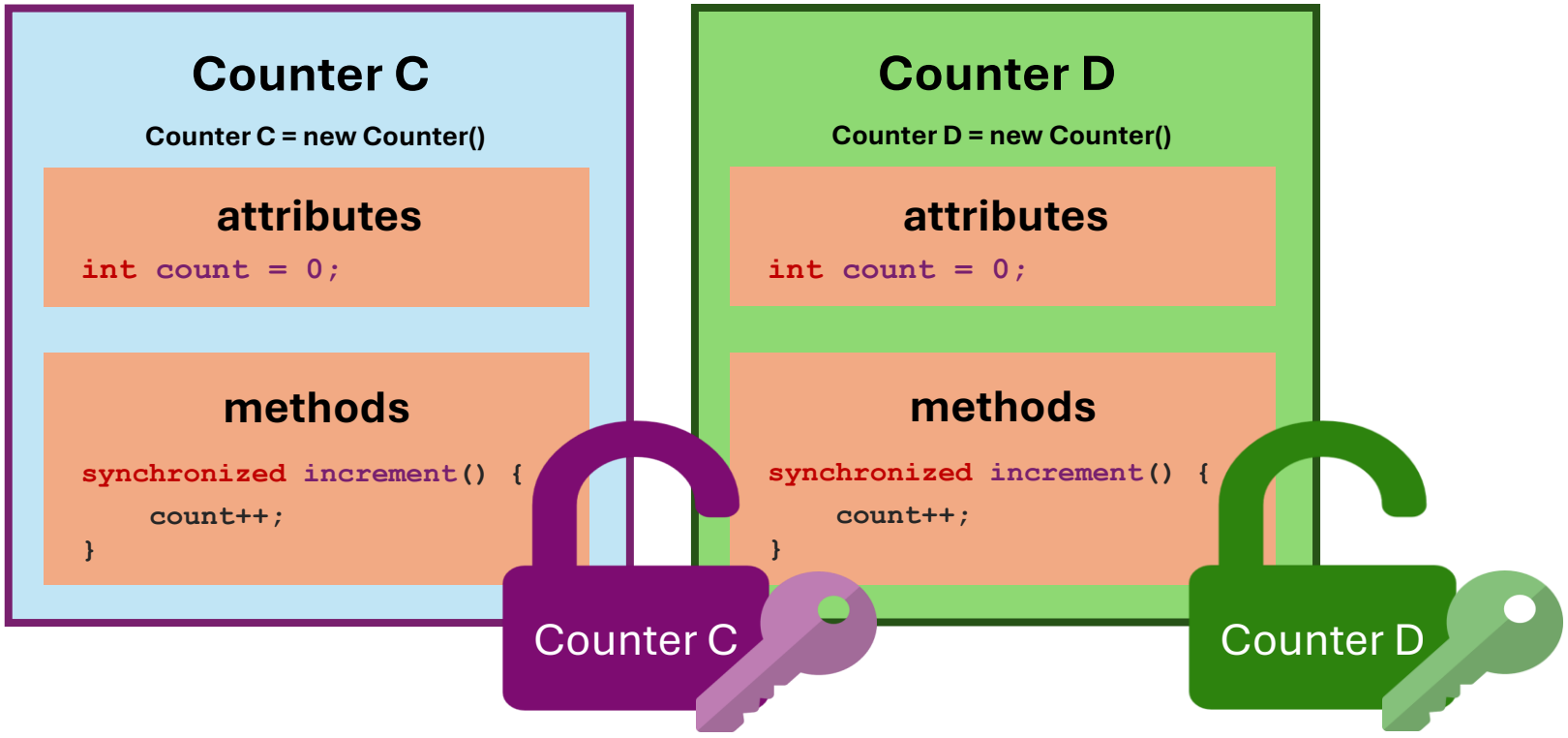
What exactly is a lock/monitor?

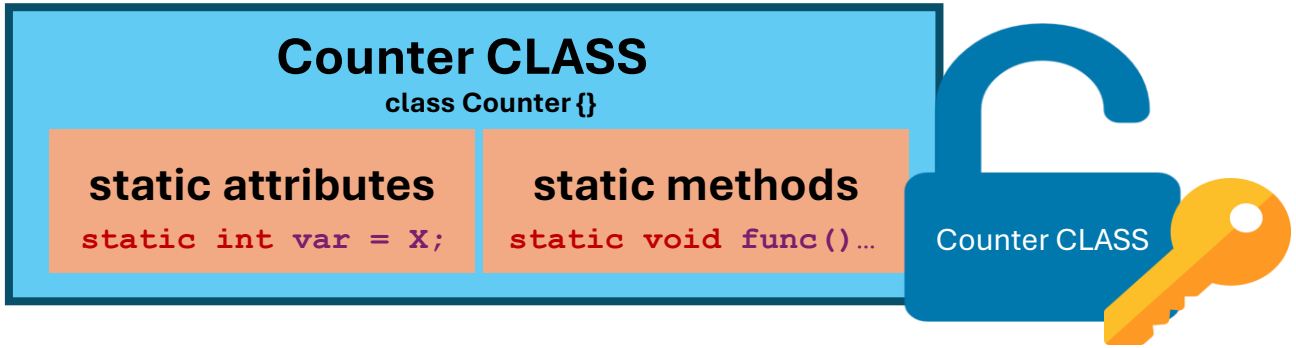
THIS LOCK IS SPECIFIC TO THE OBJECT!

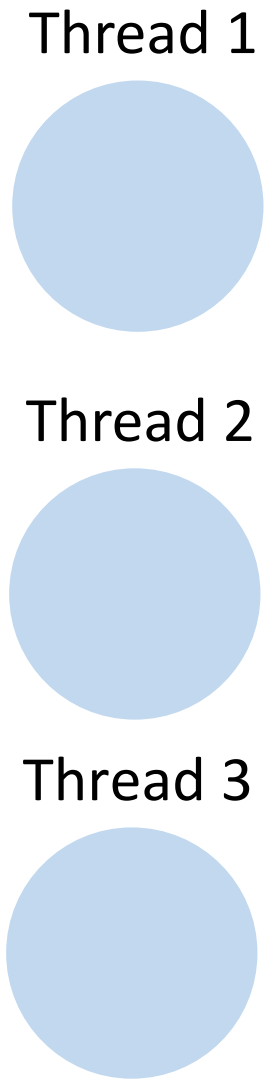




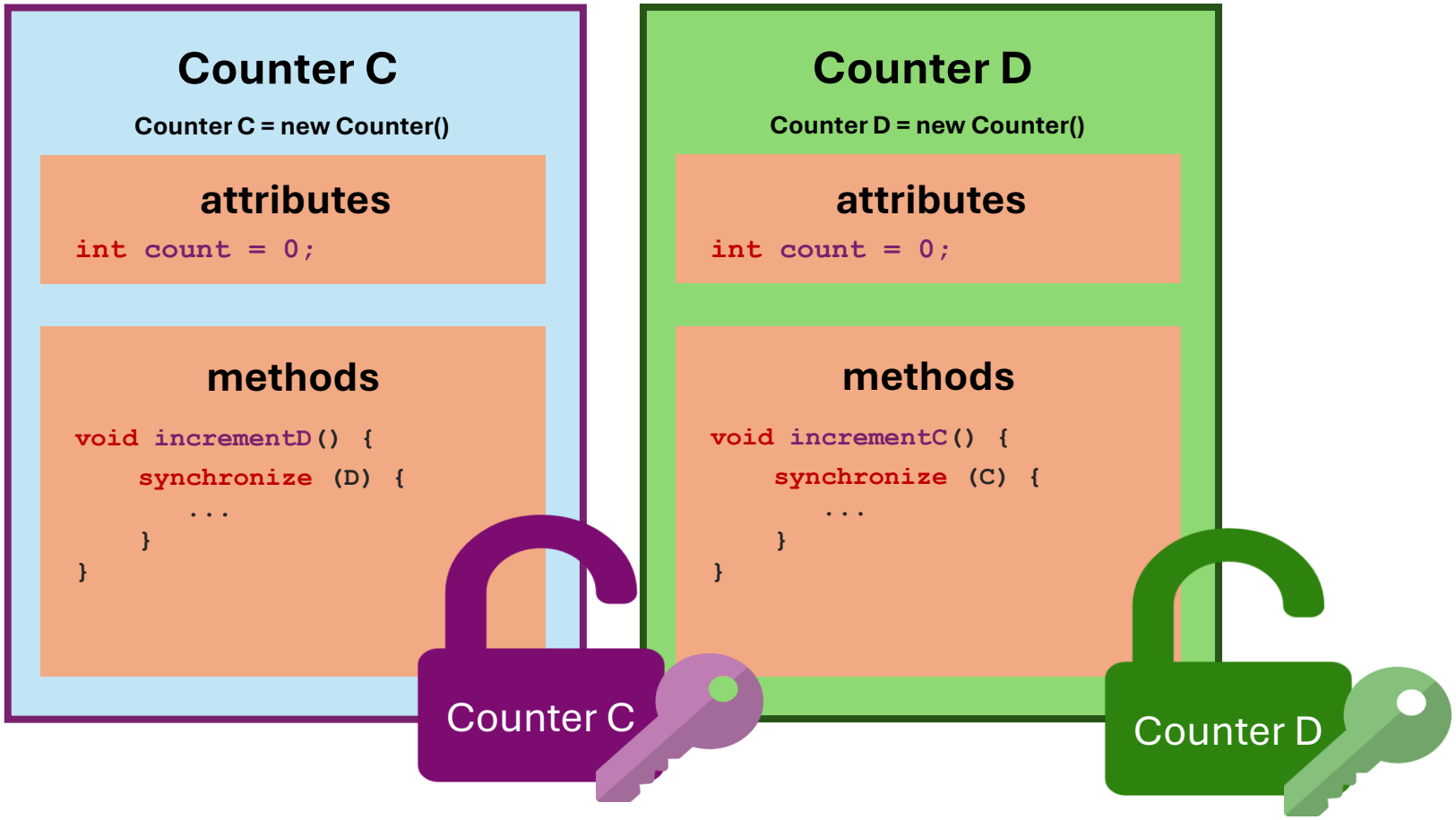
Locks are specific to Object/Class







Locks are specific to Object/Class

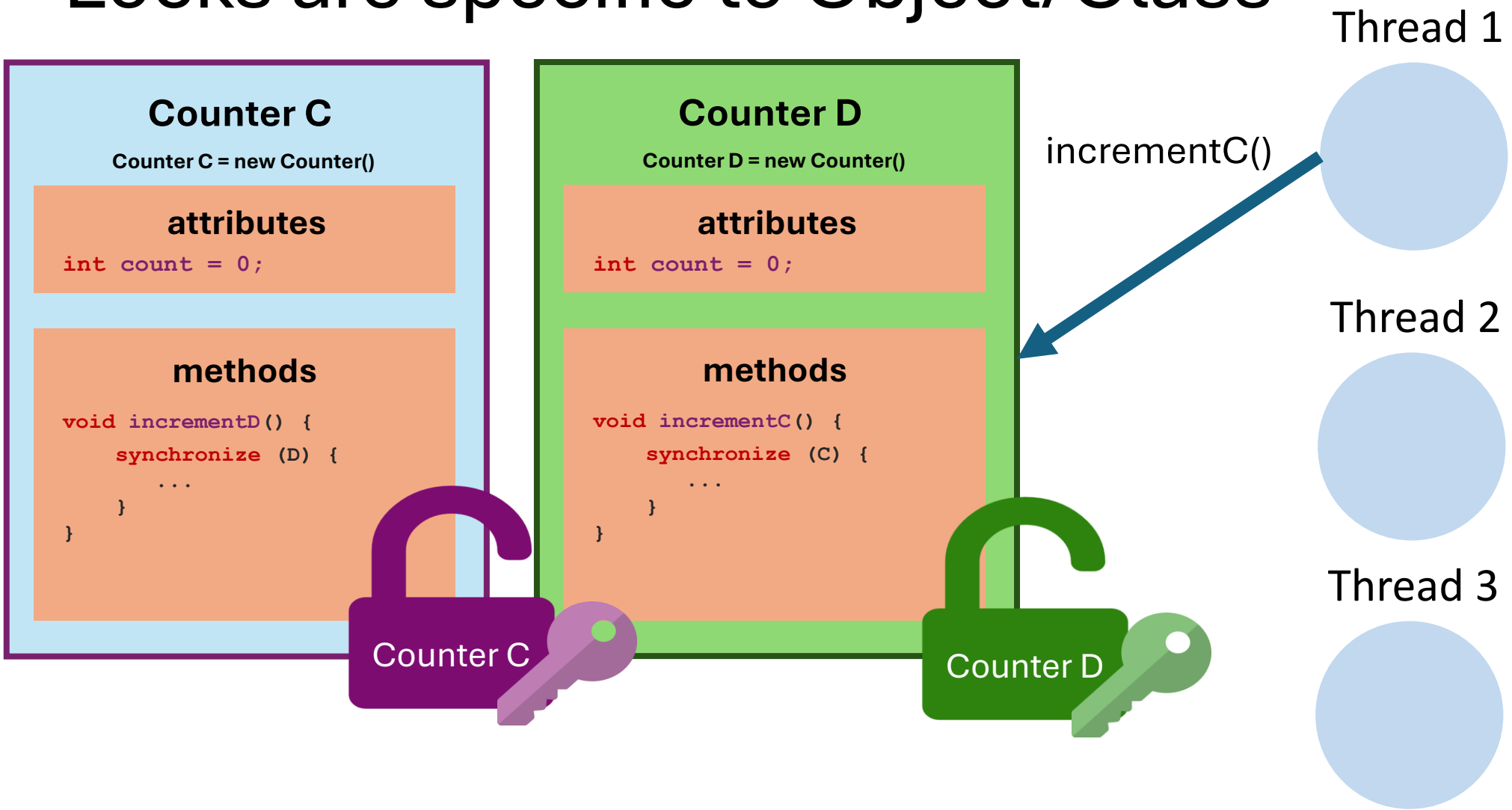


Thread 1

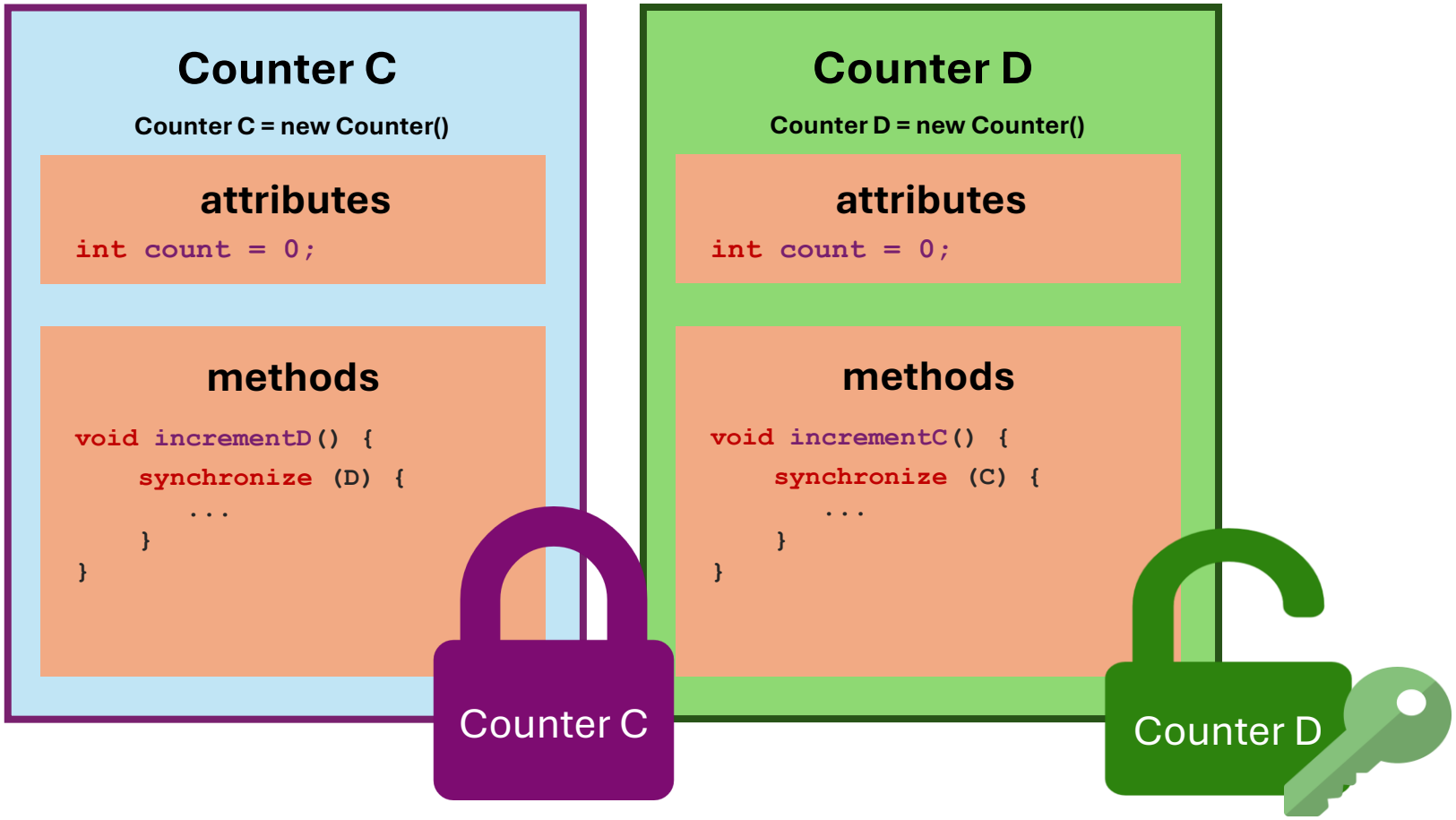
Thread 2

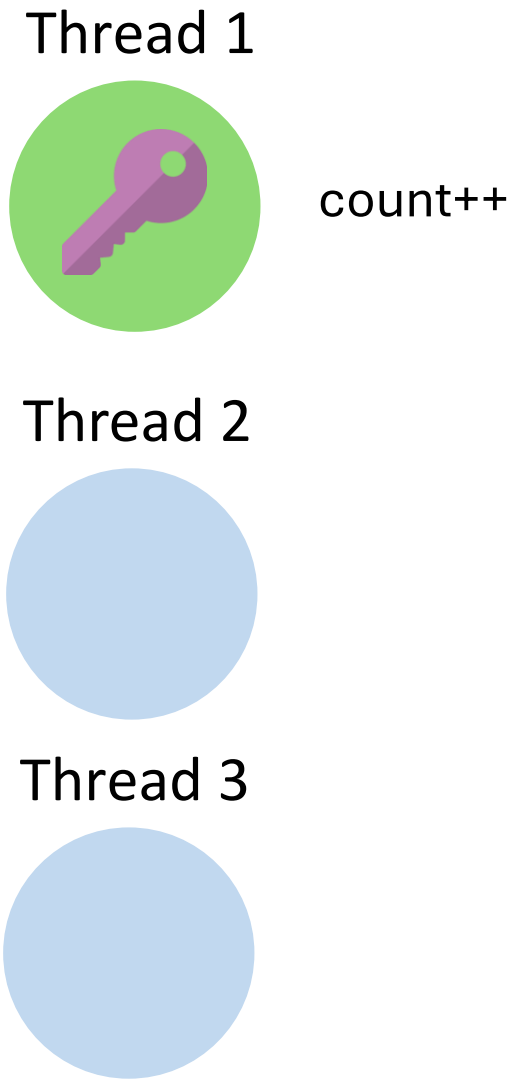
Thread 3

Locks are specific to Object/Class



Locks are specific to Object/Class





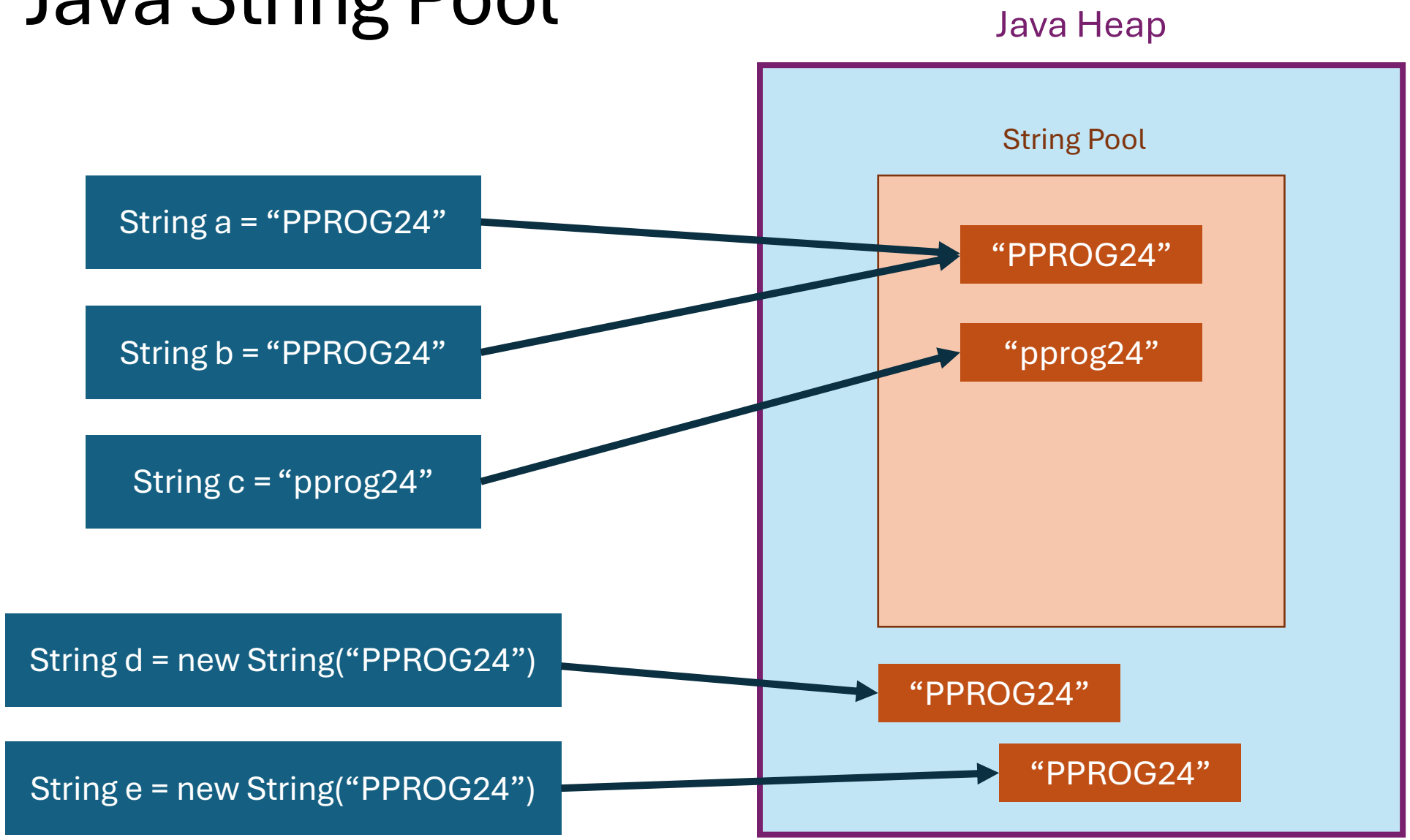
Bad Practices With Synchronization

- Do NOT synchronize on:
  - Literals
  - Boxed Primitives

Good or not good?

```
String stringLock = "LOCK_STRING";

public void badOrGood() {
    synchronized (stringLock) {
        // ...
    }
}
```

Java String Pool



Good or not good?

Good or not good?

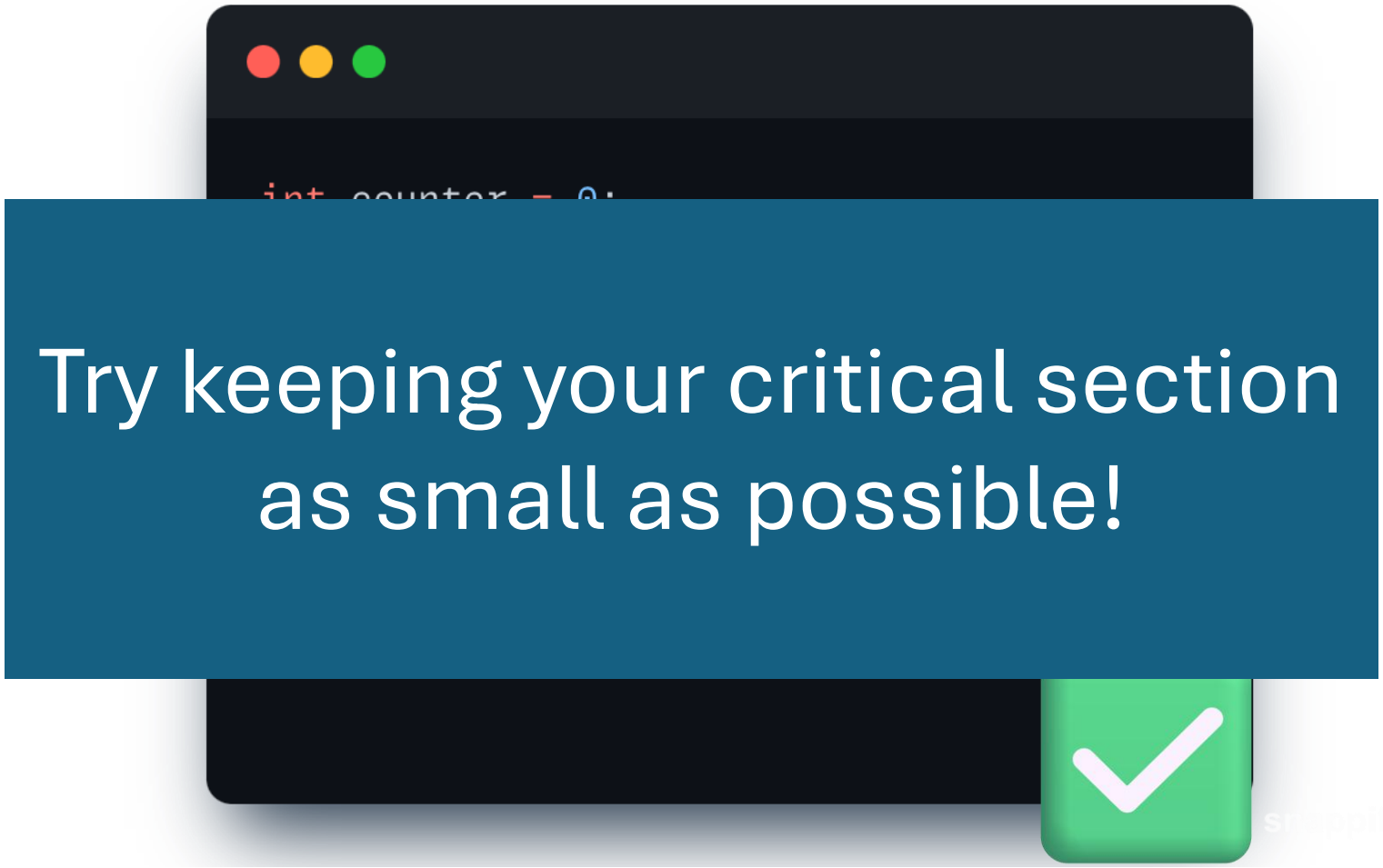
```
Integer intLock = 7;

public void badOrGood() {
    synchronized (intLock) {
        // ...
    }
}
```

Good or not good?

```
int counter = 0;

public void badOrGood() {
    synchronized (this) {
        Result r = someHeavyComputation();
        counter += r.value();
    }
}
```

Assume this computation takes *a lot* of time

Good or not good?



Using locks in Java

- Code example

Reentrant

Java locks are reentrant

A thread can hold a lock more than once

Also have to release multiple times

details

Wait and Notify

- What’s that?

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

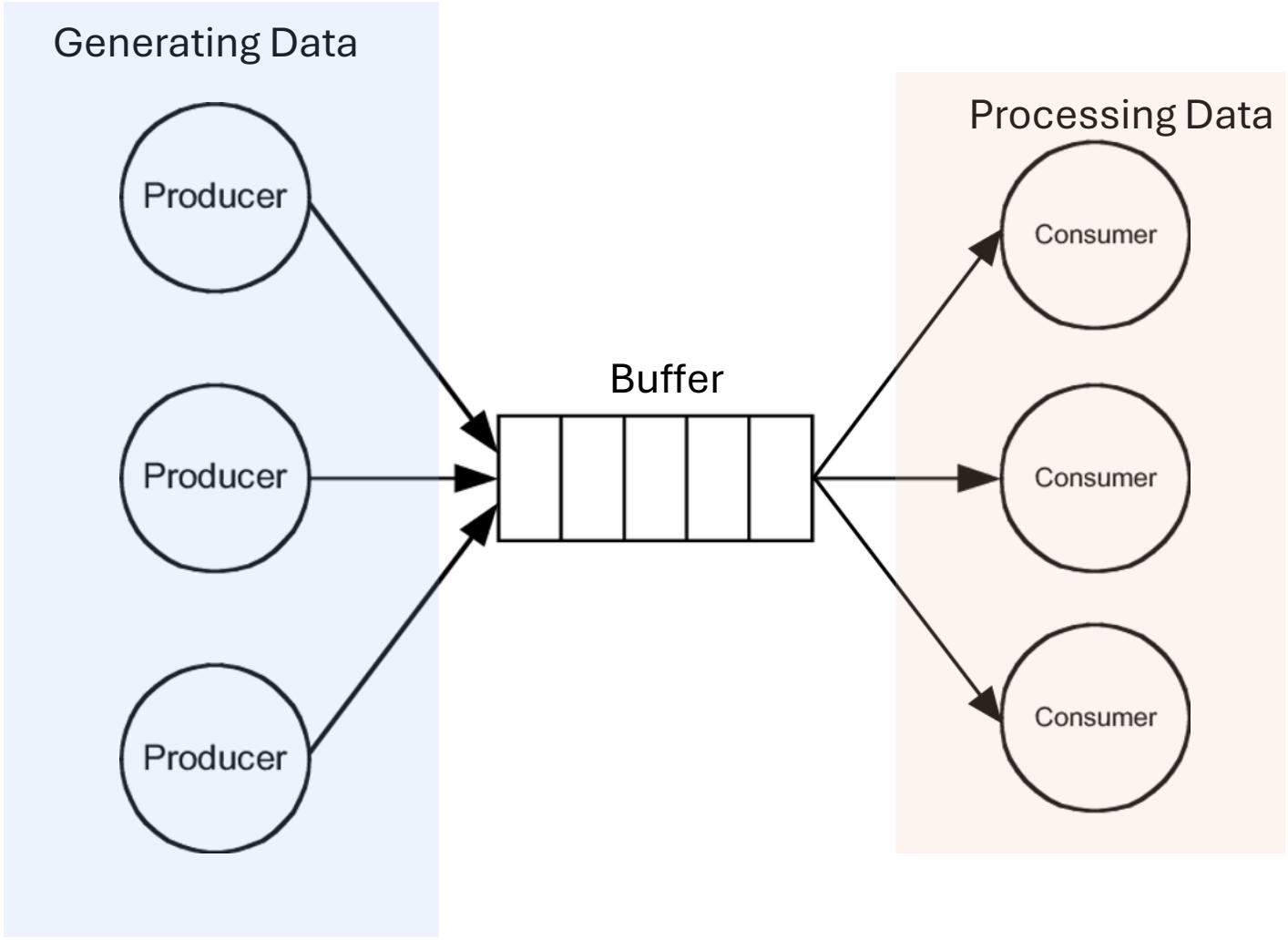
`notify`: Thread must own object's lock to call `notify`

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

`notifyAll`: Wake all threads

Why do we need this?

Producer-Consumer Problem



The Buffer

```
public class UnboundedBuffer {
  // Internal implementation could be a standard collection,
  // or a manually-maintained array or linked-list

  public boolean isEmpty() { ... }
  public void add(long value) { ... }
  public long remove() { ... }
}
```

The Producer

```
public class Producer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    ...

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

The Consumer

```
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

Where is the problem?

The Consumer

```
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

Bad Interleaving!

How about now?

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

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

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

  public void run() {
    ...

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

How about now?

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

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

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

  public void run() {
    ...

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

Problem:

1. Consumer locks buffer (`synchronize (buffer)`)
2. Consumer spins on `isEmpty()`, i.e. waits for producer to add item
3. Producer waits for lock to become available (`synchronize (buffer)`)
4. → Deadlock! Consumer and producer wait for each other; no progress

Solution? 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)

Wait/Notify für Producer Consumer Problem

- Code example

Wait and Notify Recap

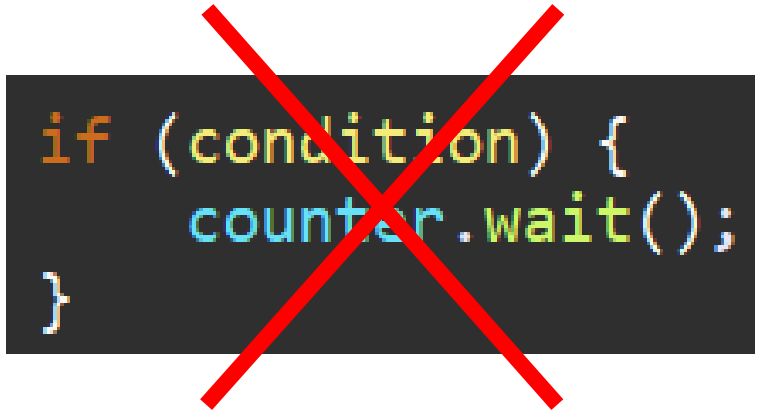
```
while (condition) {
    counter.wait();
}
```

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

What is the difference? Issues?

Wait and Notify Recap

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



Spurious wake-ups and notifyAll()

→ `wait` has to be in a `while` loop

Wait and Notify Recap

```
public class Object {
      ...
      public final native void notify();
      public final native void notifyAll();

      public final native void wait(long timeout) throws InterruptedException;
      public final void wait() throws InterruptedException { wait(0); }
      public final void wait(long timeout, int nanos)
         throws InterruptedException { ... }
}
```

wait() releases object lock, thread waits on internal queue

notify() wakes the highest-priority thread closest to front of object's internal queue

notifyAll() wakes up all waiting threads

- Threads non-deterministically compete for access to object
- May not be fair (low-priority threads may never get access)

May only be called when object is locked (e.g. inside `synchronize`)

details

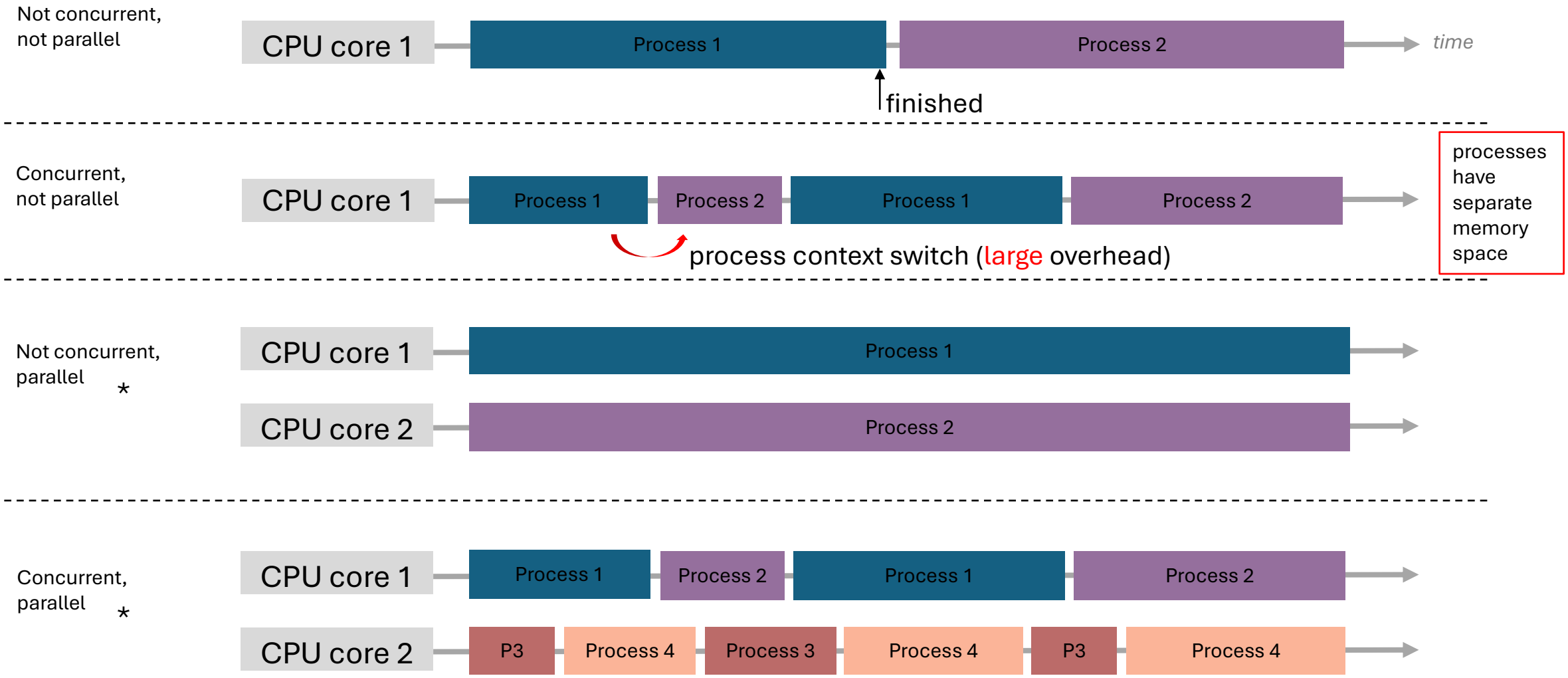
Concurrency vs Parallelism

Concurrency
Dealing with multiple things at the same time (ETH experience)
Reasoning about and managing shared resources. Often used interchangeably with parallelism.

Parallelism
Doing multiple things at the same time
Performing computations simultaneously; either actually, if sufficient computations units (CPUs, cores, ...) are available, or virtually, via some form of alternation. Often used interchangeably with concurrency. Parallelism can be specified explicitly by manually assigning tasks to threads or implicitly by using a framework that takes care of distributing tasks to threads.

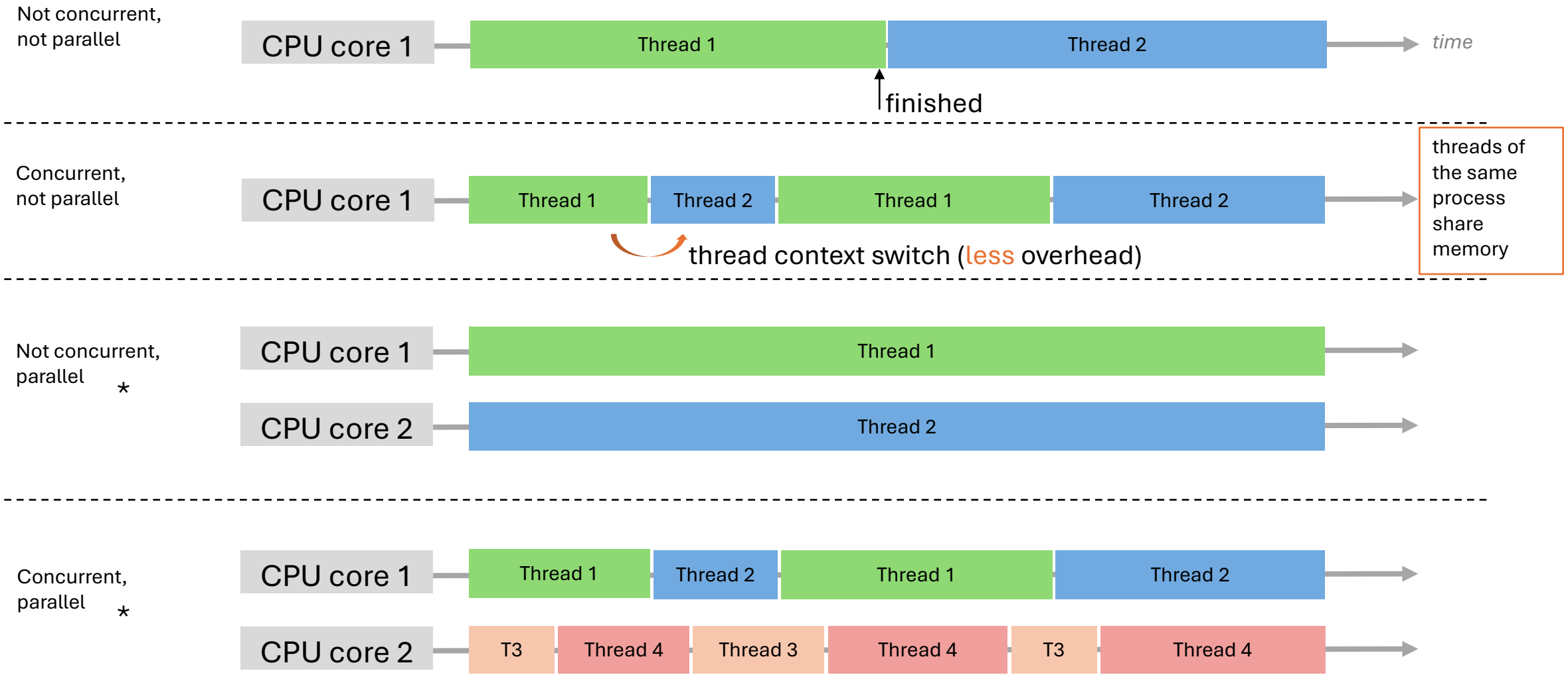
Source: https://cgl.ethz.ch/teaching/parallelprog23/pages/terminology.html

Sequential, Concurrent, Parallel



*multi-core and multi-processor systems

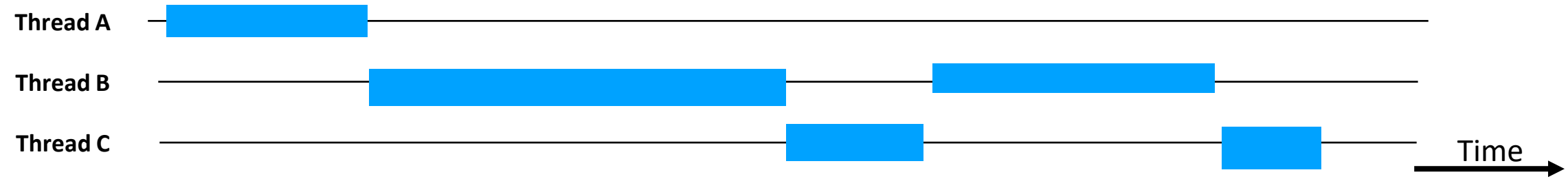
Sequential, Concurrent, Parallel



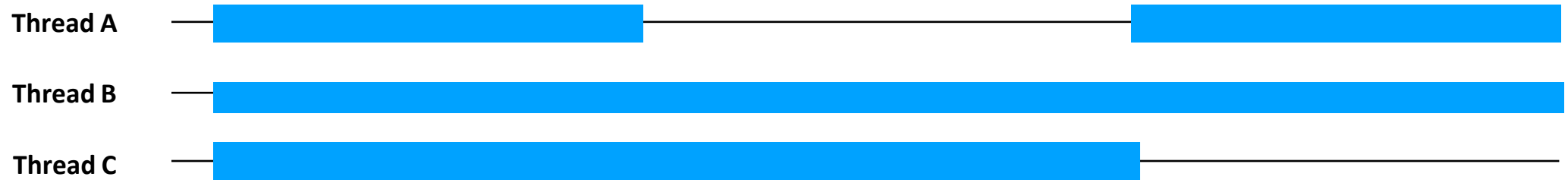
*multi-core and multi-processor systems

Concurrency vs Parallelism

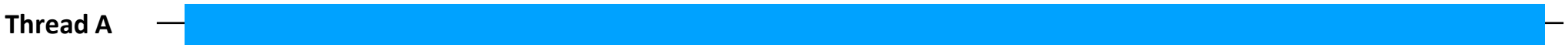
Concurrent, not parallel



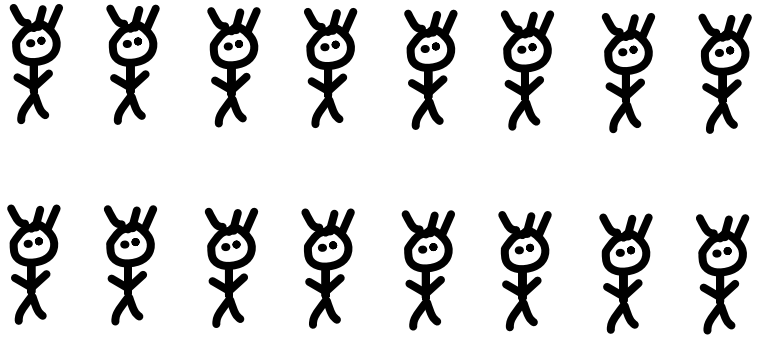
Concurrent, parallel

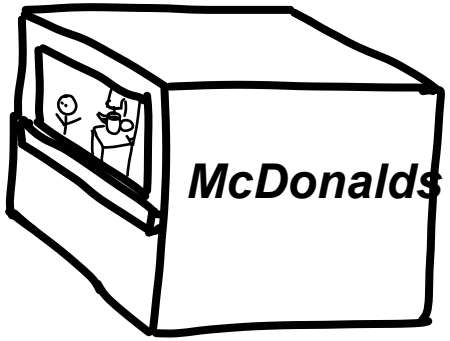


Not concurrent, not parallel

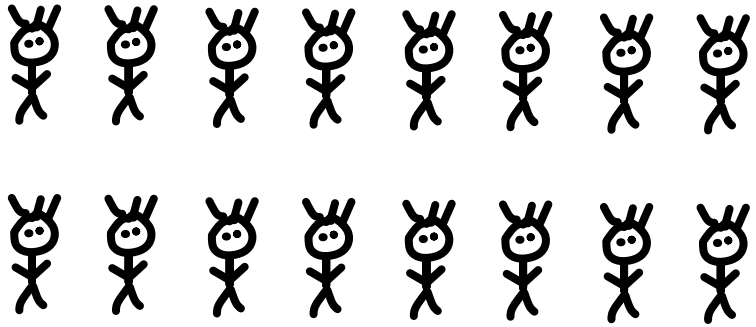


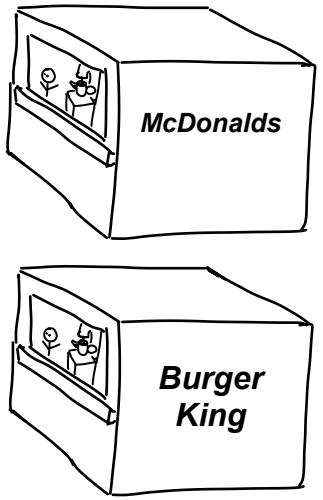
Concurrency vs Parallelism





Concurrency
No parallelism





Concurrency
Parallelism

Plan für heute

- Organisation
- Nachbesprechung Exercise 2
- Theory Recap
- **Intro Exercise 3**
- Exam Questions
- Kahoot

Counter

There are many threads accessing the counter at the same time.
How should we implement it such that there are no conflicts?
You will try different solutions including:

- ➔ *Task A: SequentialCounter*
- ➔ *Task B: SynchronizedCounter*
- ➔ *Task E (optional): AtomicCounter*

Task A – Sequential counter

- ➔ Implement a sequential version of the Counter in SequentialCounter class that does not use any synchronization.
- ➔ In taskASequential we provide a method that runs a single thread which increments the counter. Inspect the code and understand how it works.
- ➔ Verify that the SequentialCounter works properly when used with a single thread (the test testSequentialCounter should pass).

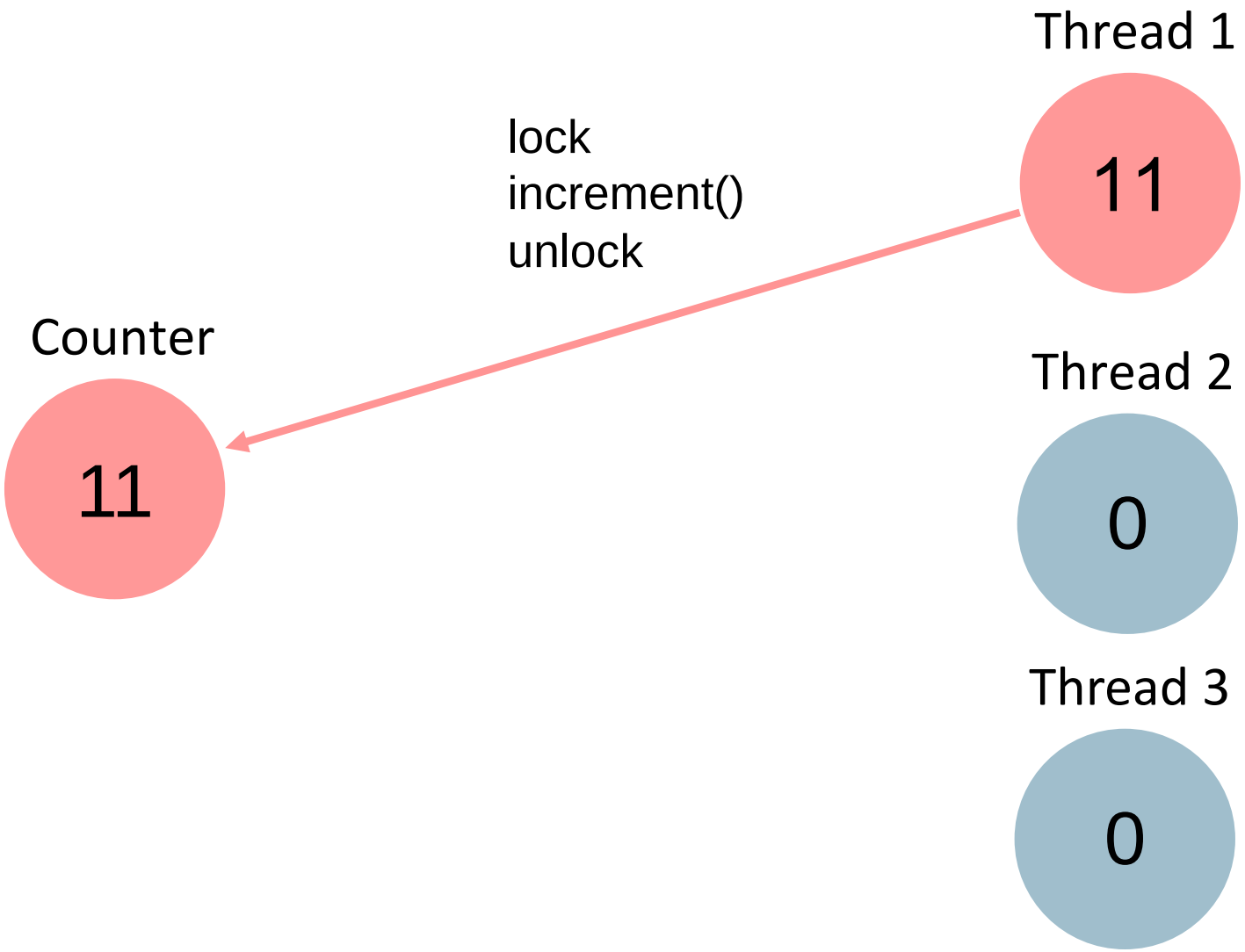
Task A – Parallel counter

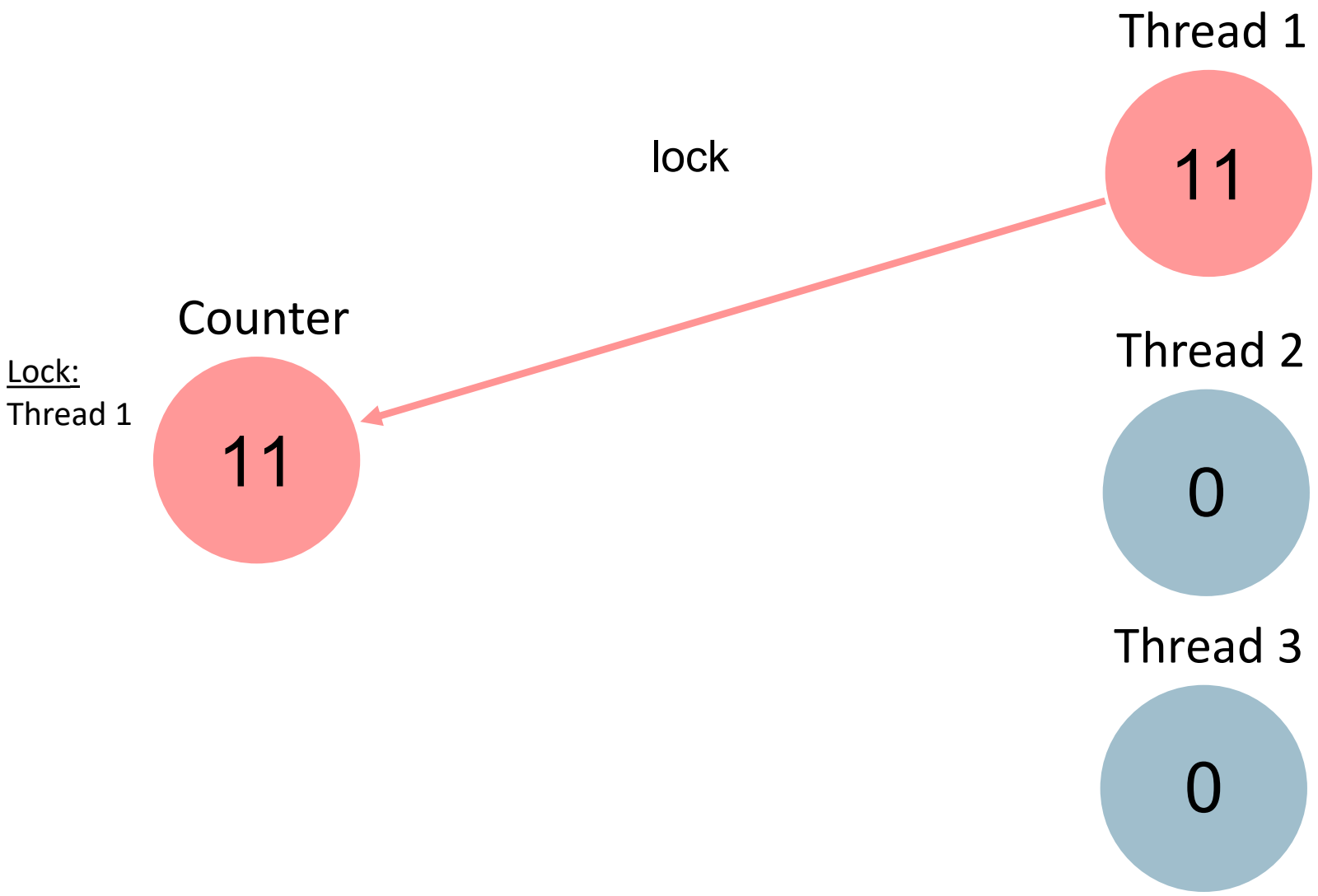
- ➔ Run the code in taskAParallel which creates several threads that all try to increment the counter at the same time.
- ➔ Will this work? What will happen?

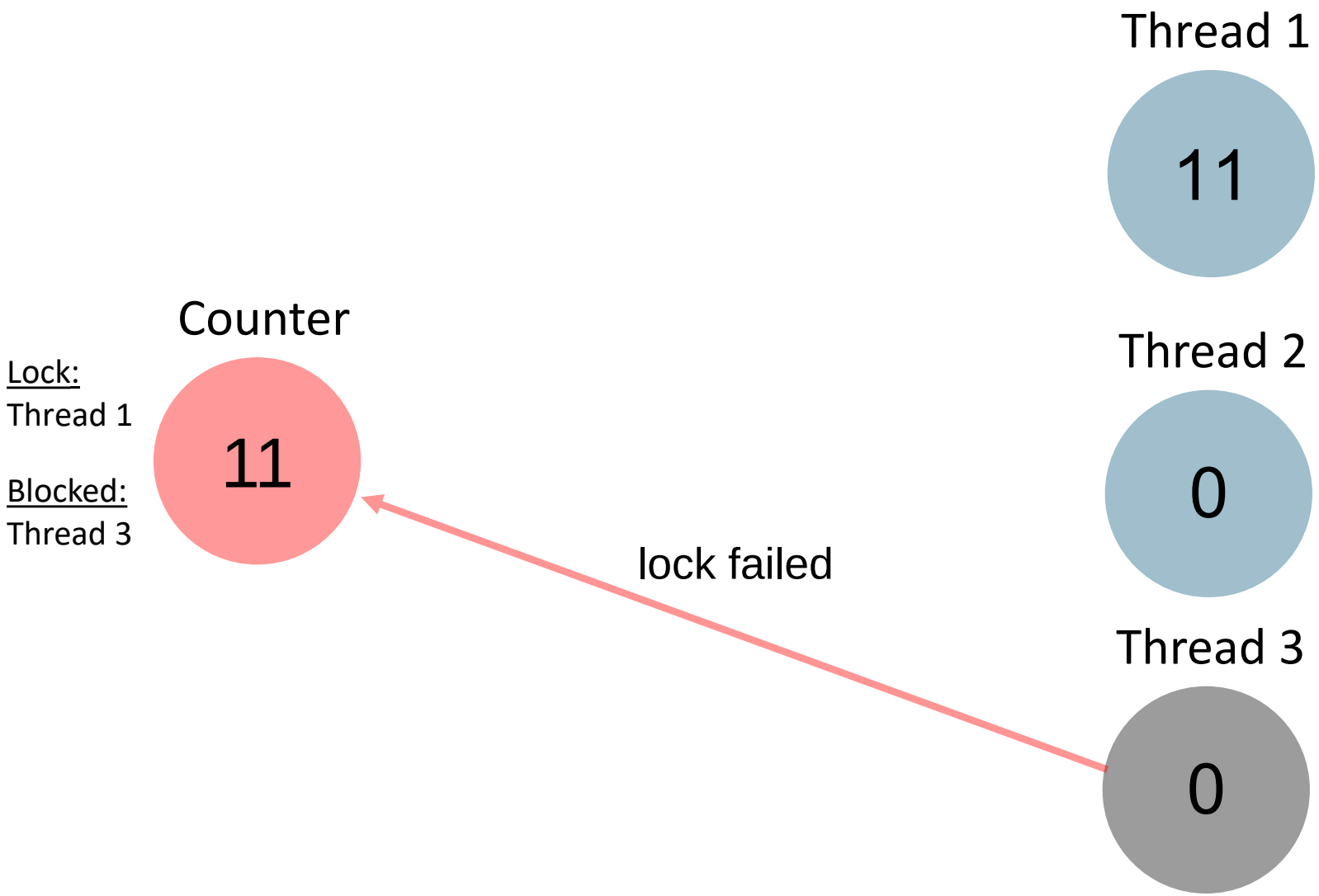
Task B – Synchronized counter

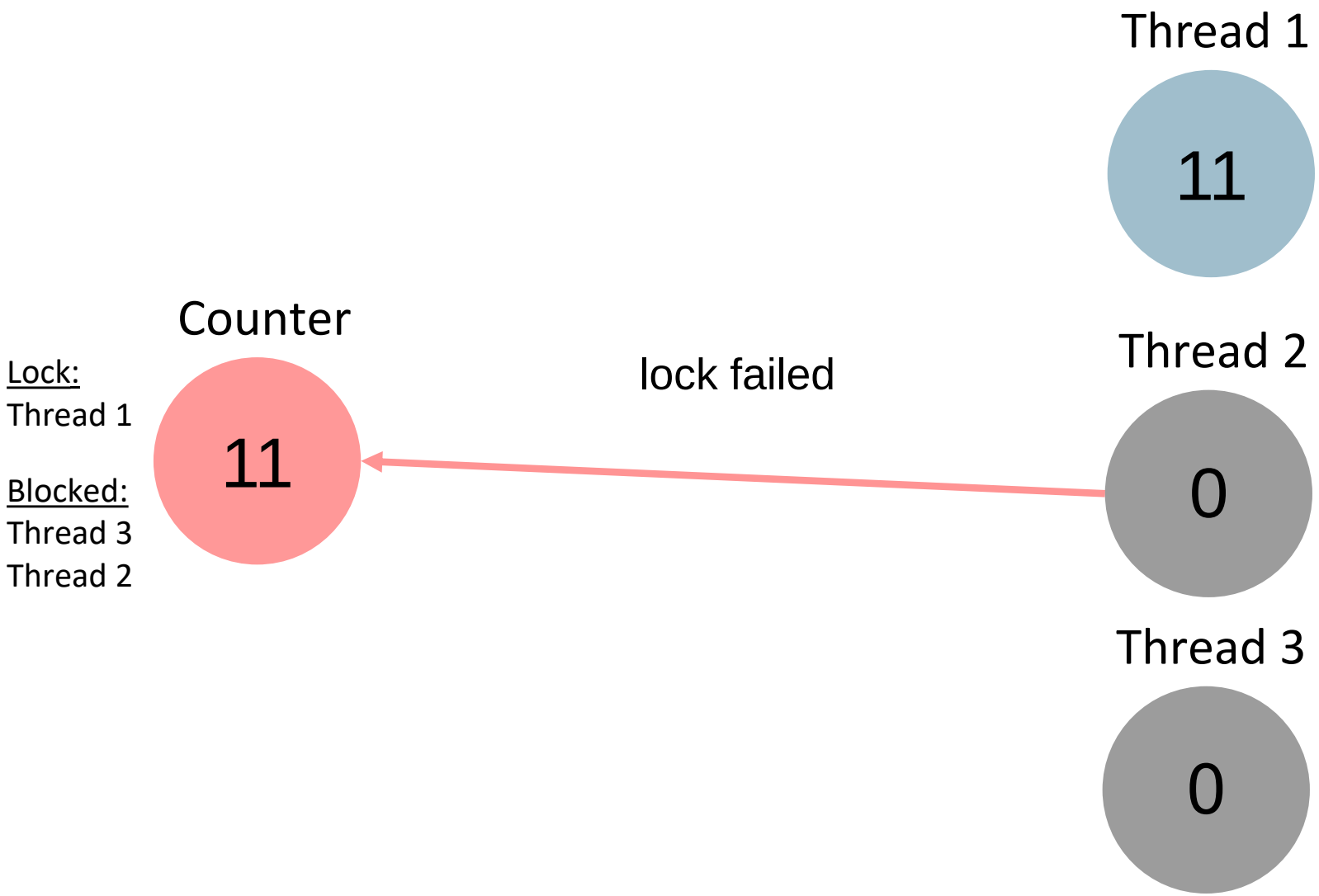
- ➔ Implement a different thread safe version of the Counter in SynchronizedCounter. In this version use the standard primitive type int but synchronize the access to the variable by inserting synchronized blocks.
- ➔ Run the code in taskB.

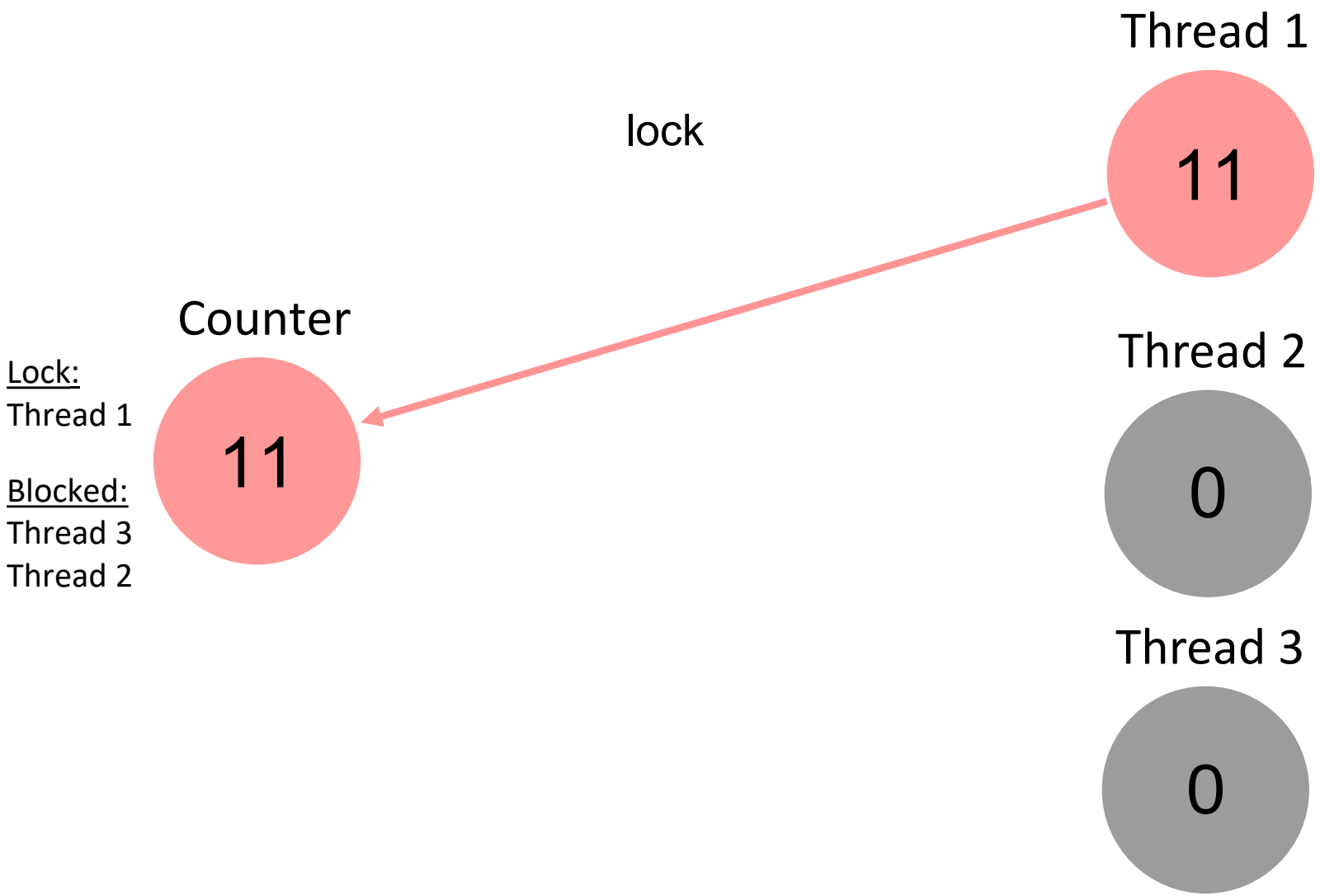

Thread 1
10
Counter
10
Thread 2
0
Thread 3
0

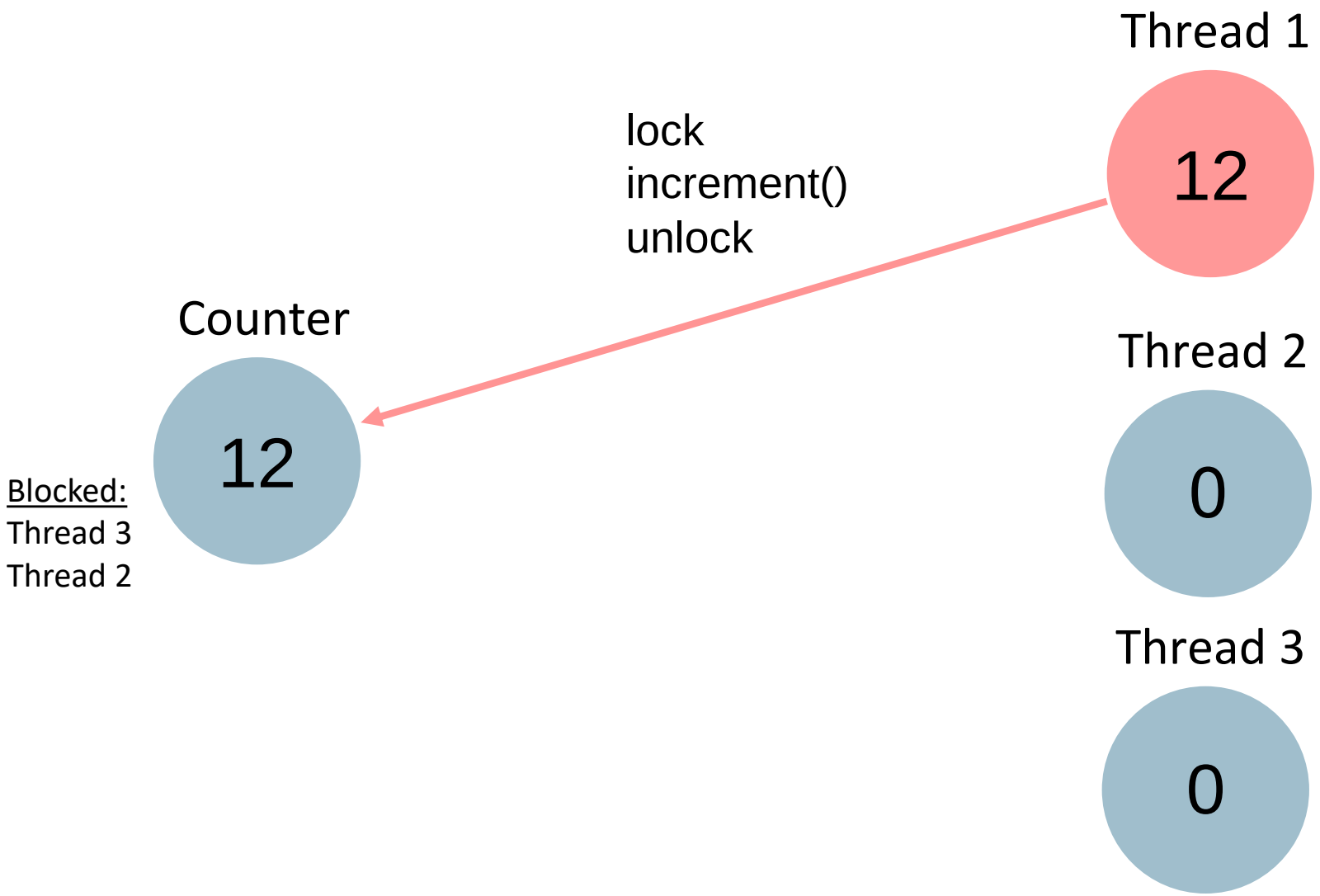

Thread 1
11
lock
increment()
unlock
Counter
11
Thread 2
0
Thread 3
0


Thread 1
11
lock
Counter
Lock:
Thread 1
11
Thread 2
0
Thread 3
0


Thread 1
11
Counter
Lock:
Thread 1
Blocked:
Thread 3
11
Thread 2
0
lock failed
Thread 3
0


Thread 1
11
Counter
Lock:
Thread 1
Blocked:
Thread 3
Thread 2
11
lock failed
Thread 2
0
Thread 3
0


Thread 1
lock
11
Counter
Lock:
Thread 1
11
Thread 2
Blocked:
Thread 3
Thread 2
0
Thread 3
0


Thread 1
12
lock
increment()
unlock
Counter
12
Thread 2
0
Blocked:
Thread 3
Thread 2
Thread 3
0

- See code LockExample

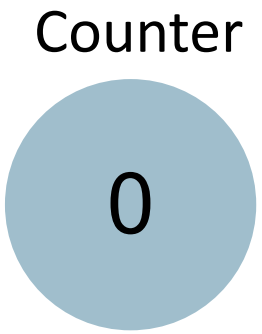
Task C

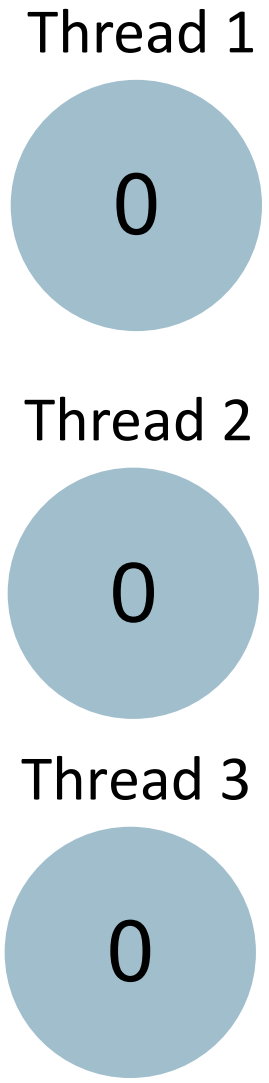
Whenever the Counter is incremented, keep track which thread performed the increment (you can print out the thread-id to the console). Can you see a pattern in how the threads are scheduled? Discuss what might be the reason for this behaviour.

Task D

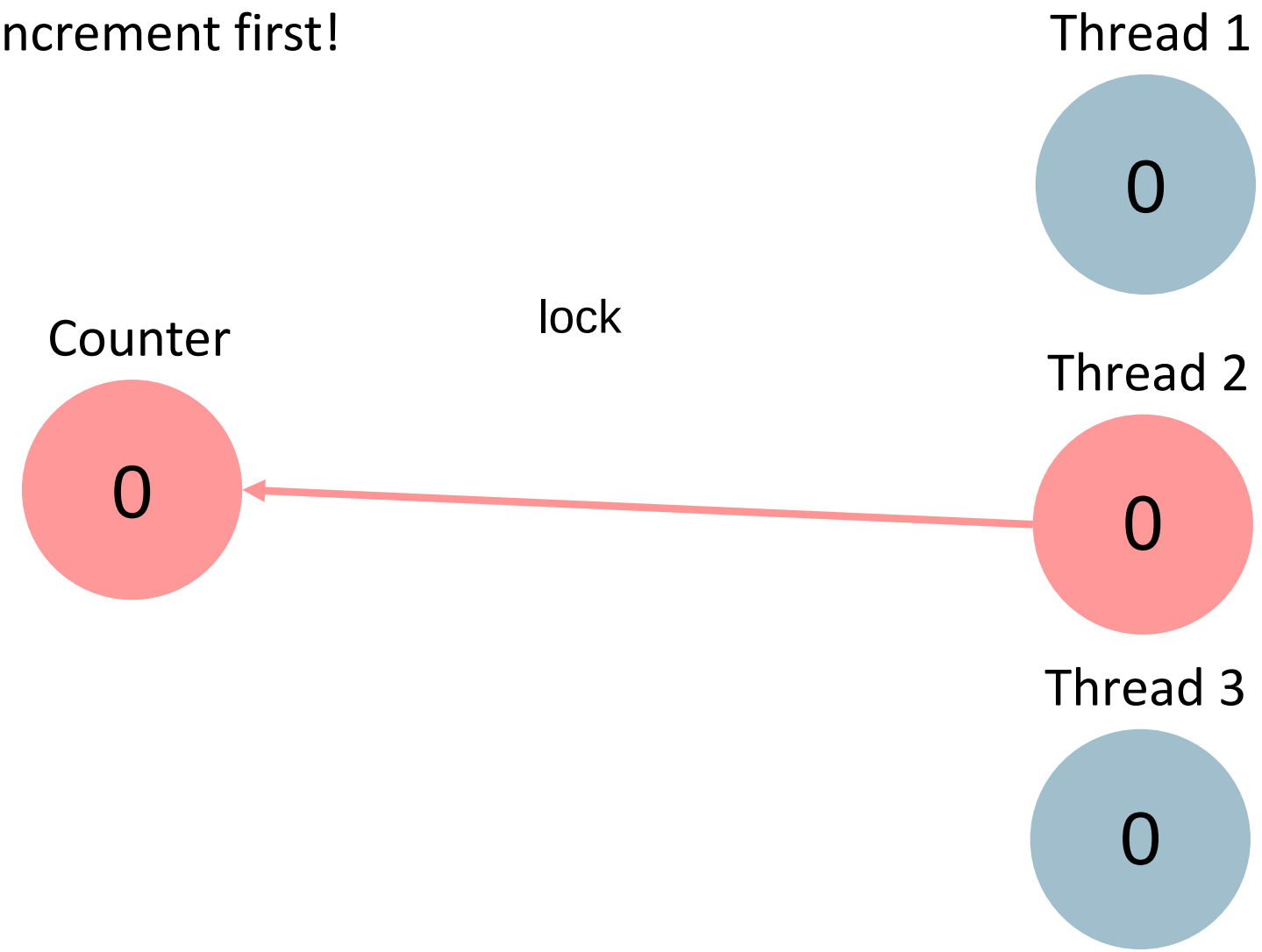
- ➔ Implement a FairThreadCounter that ensures that different threads increment the Counter in a round-robin fashion. In round-robin scheduling the threads perform the increments in circular order. That is, two threads with ids 1 and 2 would increment the value in the following order 1, 2, 1, 2, 1, 2, etc.
- ➔ You should implement the scheduling using the **wait** and **notify** methods.
- ➔ Can you think of implementation that does not use wait and notify methods?

Thread 1 must increment first!

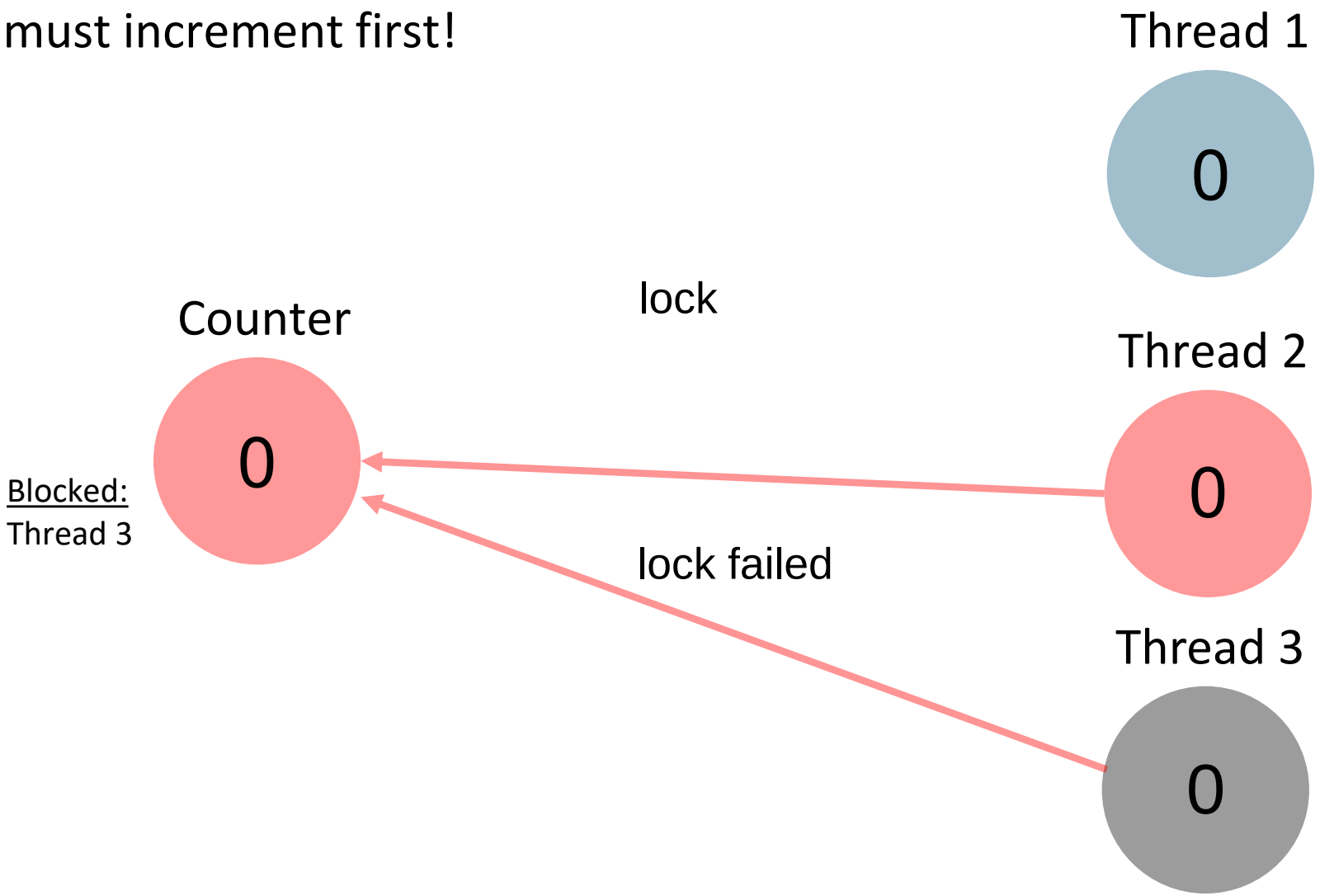




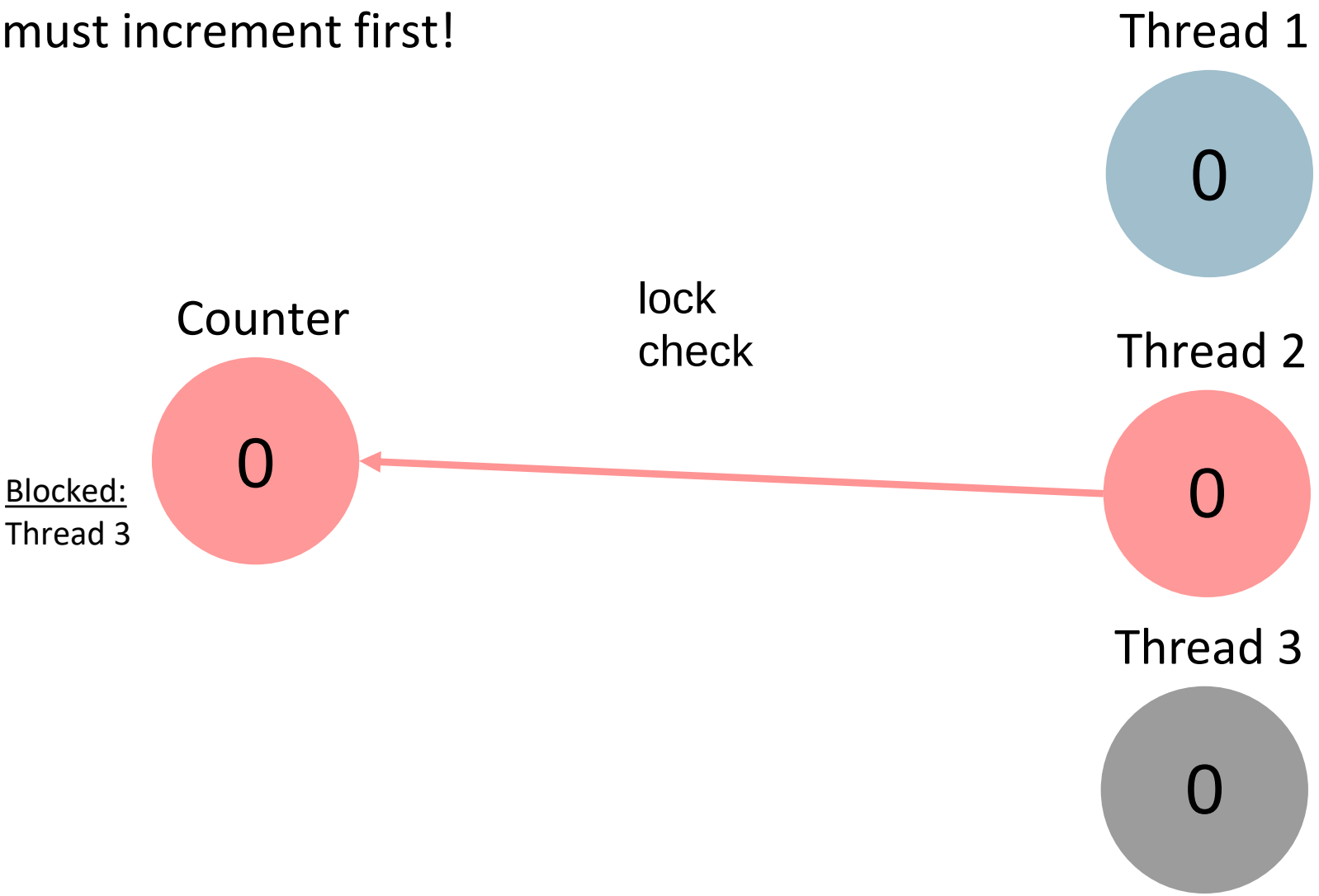
Thread 1 must increment first!



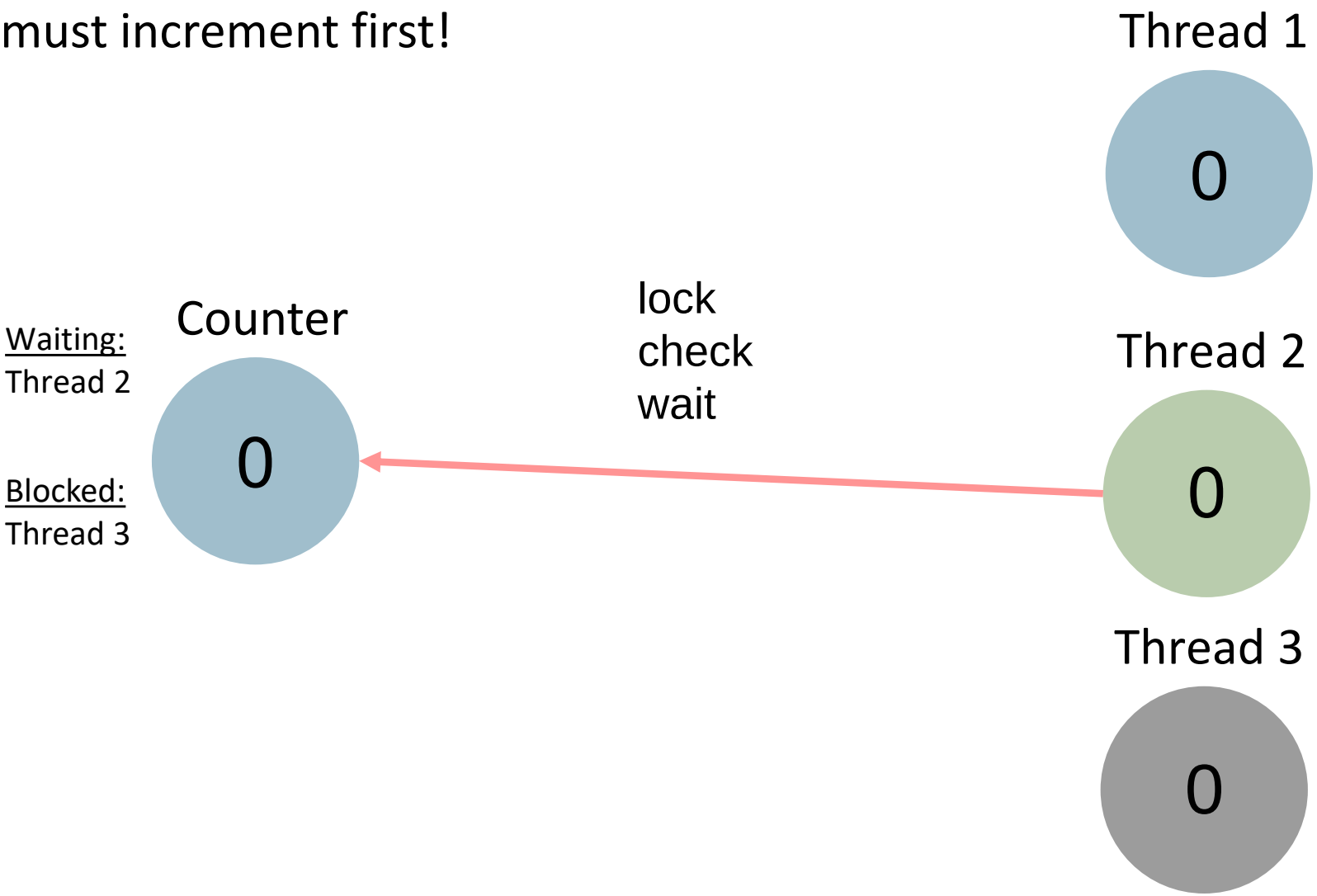
Thread 1 must increment first!



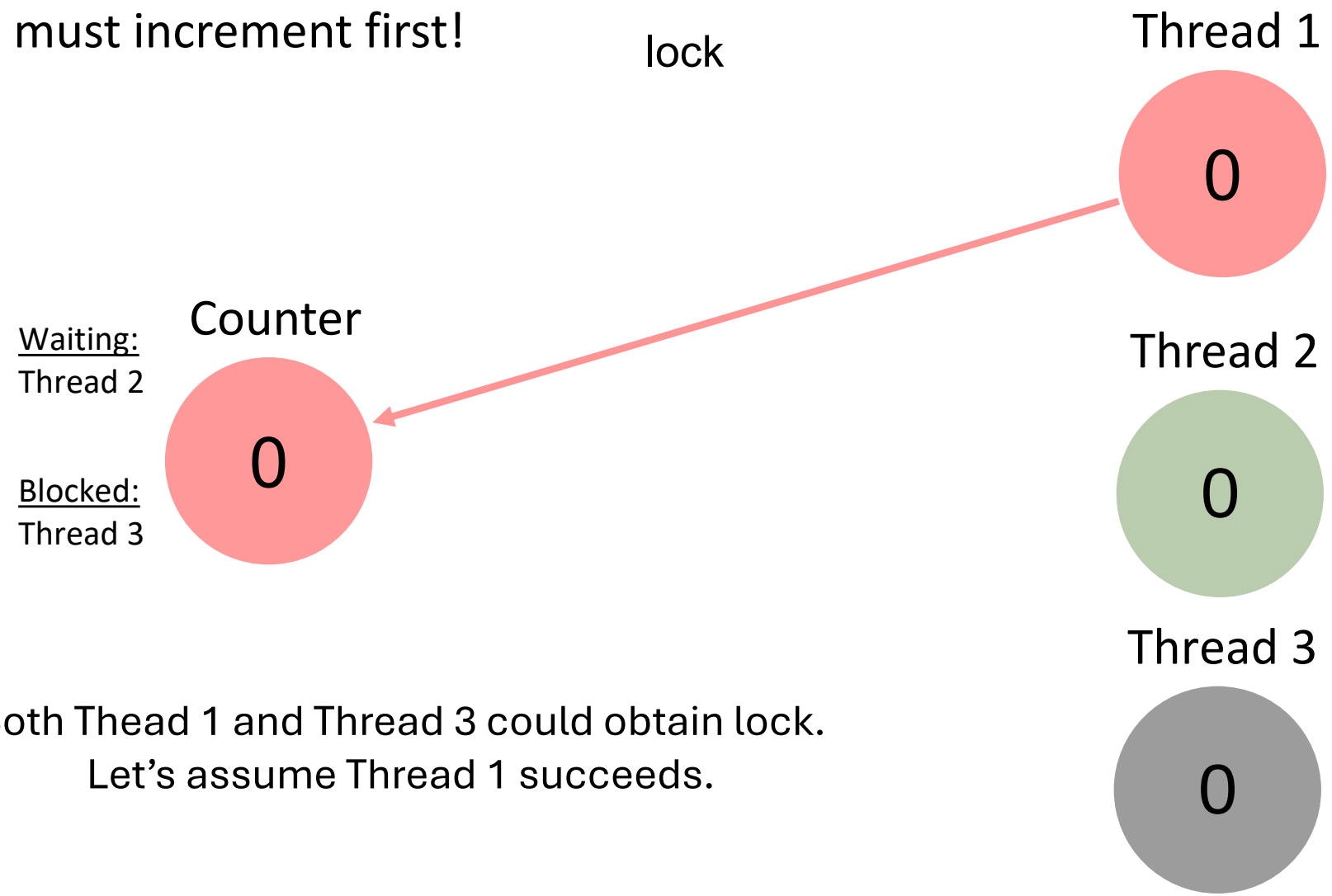
Thread 1 must increment first!



Thread 1 must increment first!

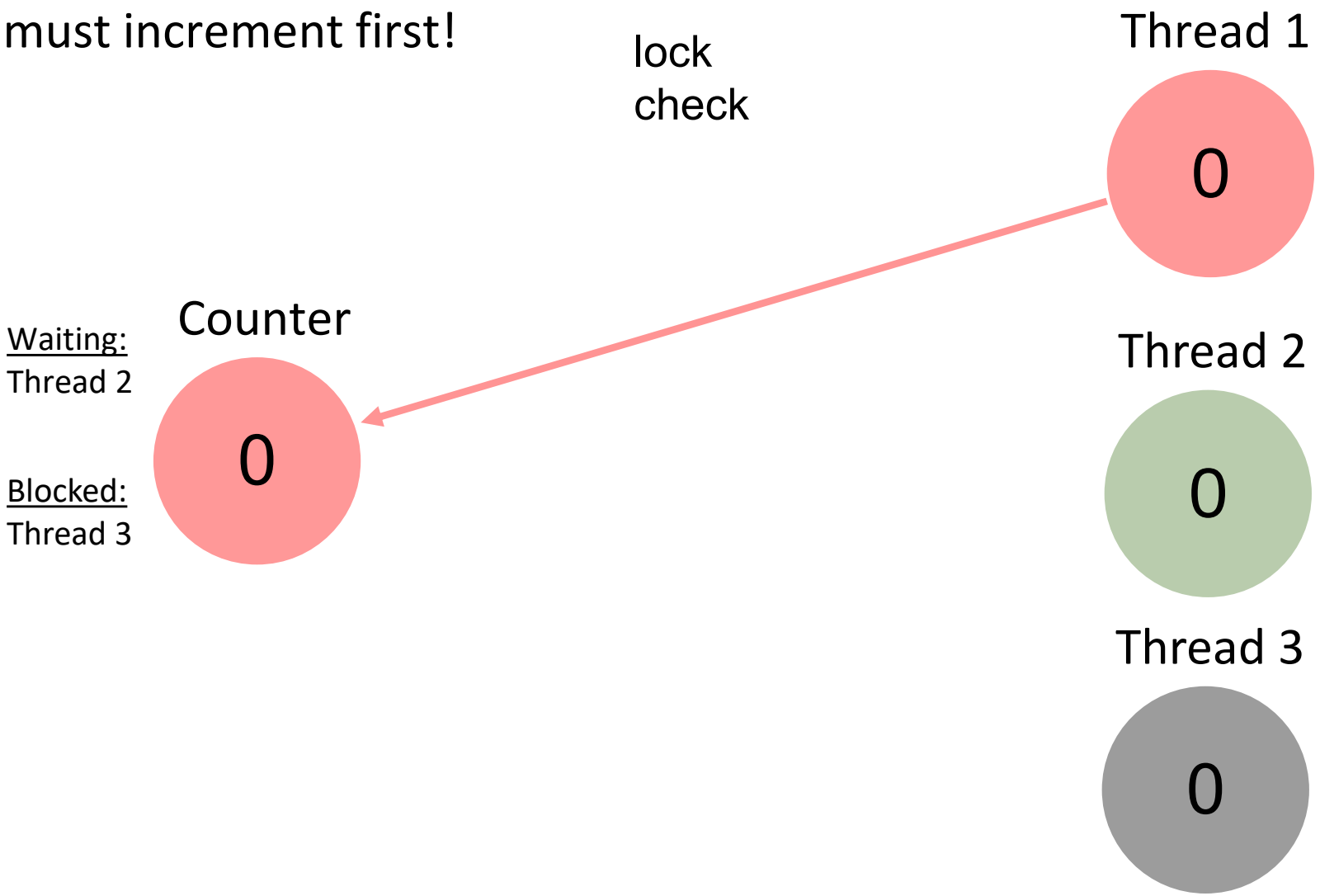


Thread 1 must increment first!

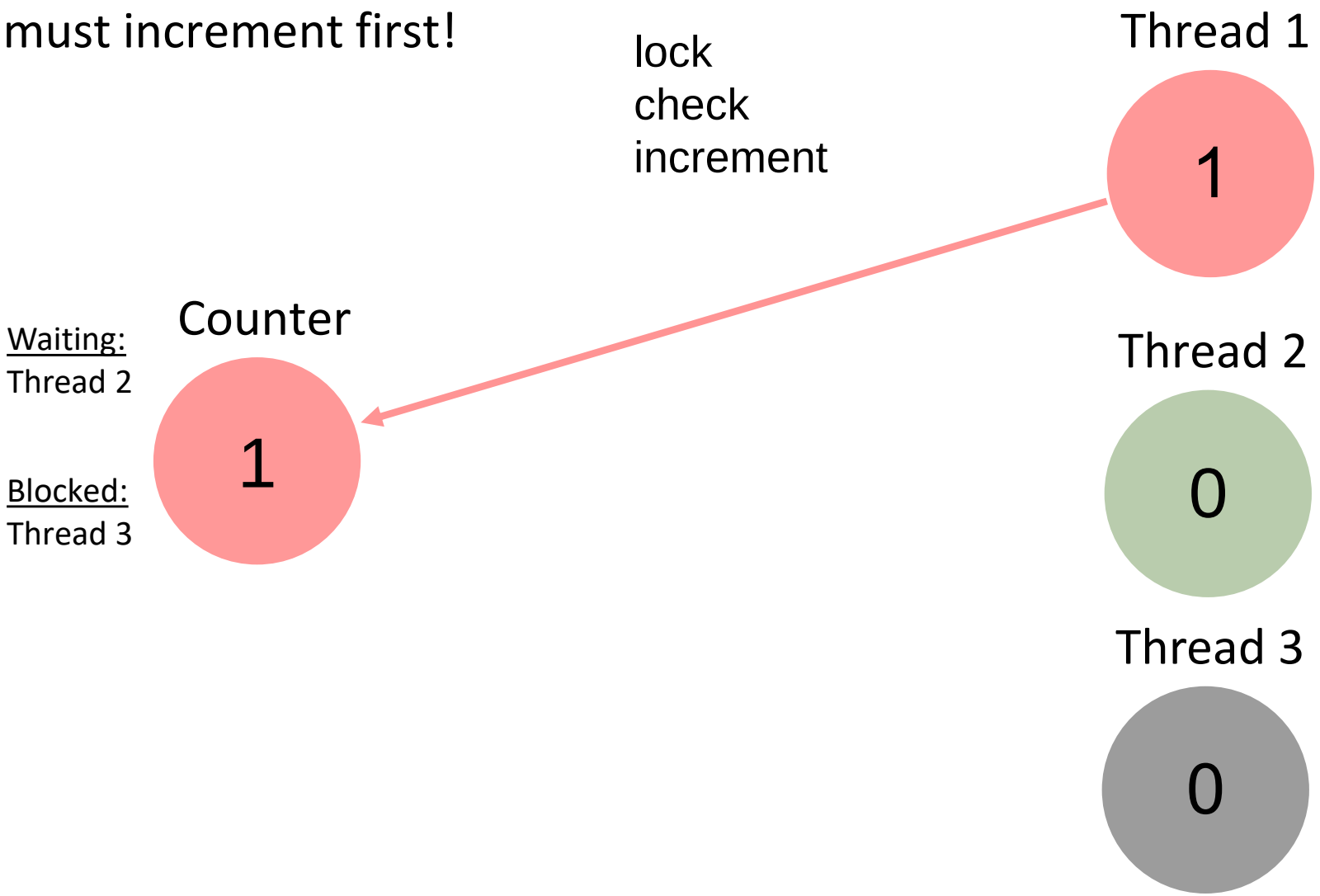


Both Thead 1 and Thread 3 could obtain lock.
Let's assume Thread 1 succeeds.

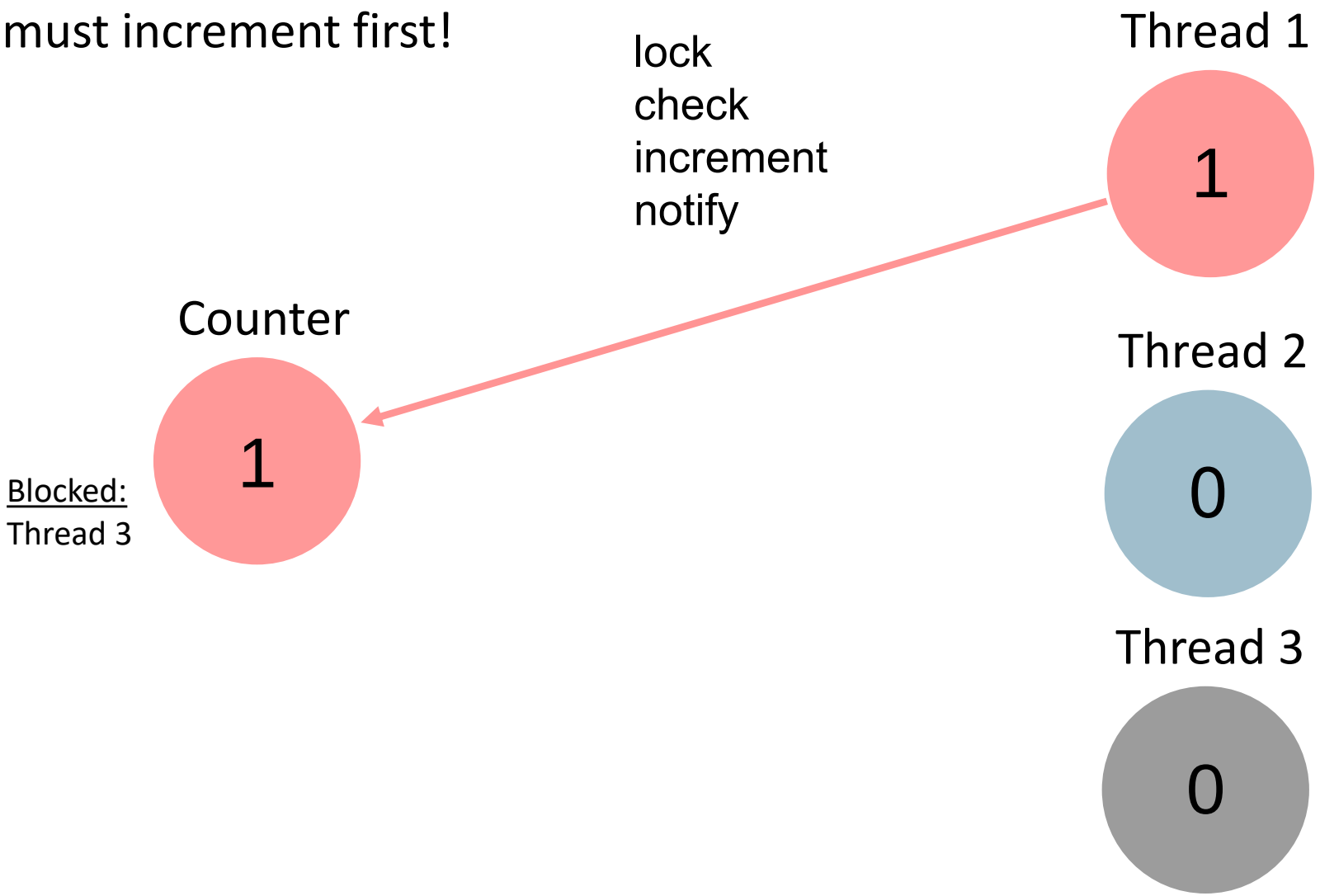
Thread 1 must increment first!



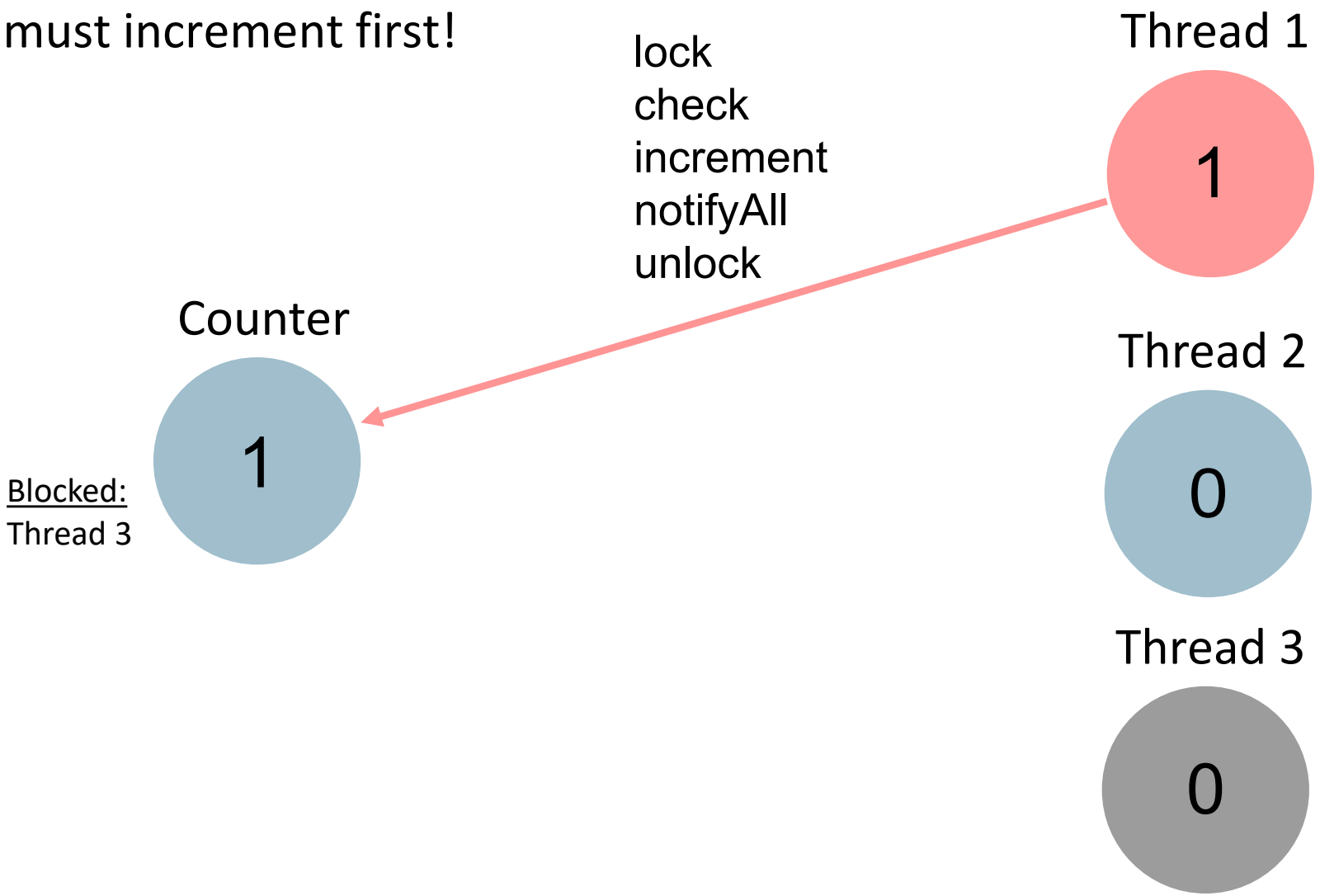
Thread 1 must increment first!



Thread 1 must increment first!



Thread 1 must increment first!



Task E – Atomic counter

Implement a thread safe version of the Counter in AtomicCounter. In this version we will use an implementation of the int primitive value, called AtomicInteger, that can be safely used from multiple threads.

Atomic Variables

- ➔ Set of classes providing implementation of atomic variables in Java, e.g., AtomicInteger, AtomicLong, …
- ➔ An operation is atomic if no other thread can see it partially executed. Atomic as in “appears indivisible”.
- ➔ Implemented using special hardware primitives (instructions) for concurrency. *Will be covered in detail later in the course.*

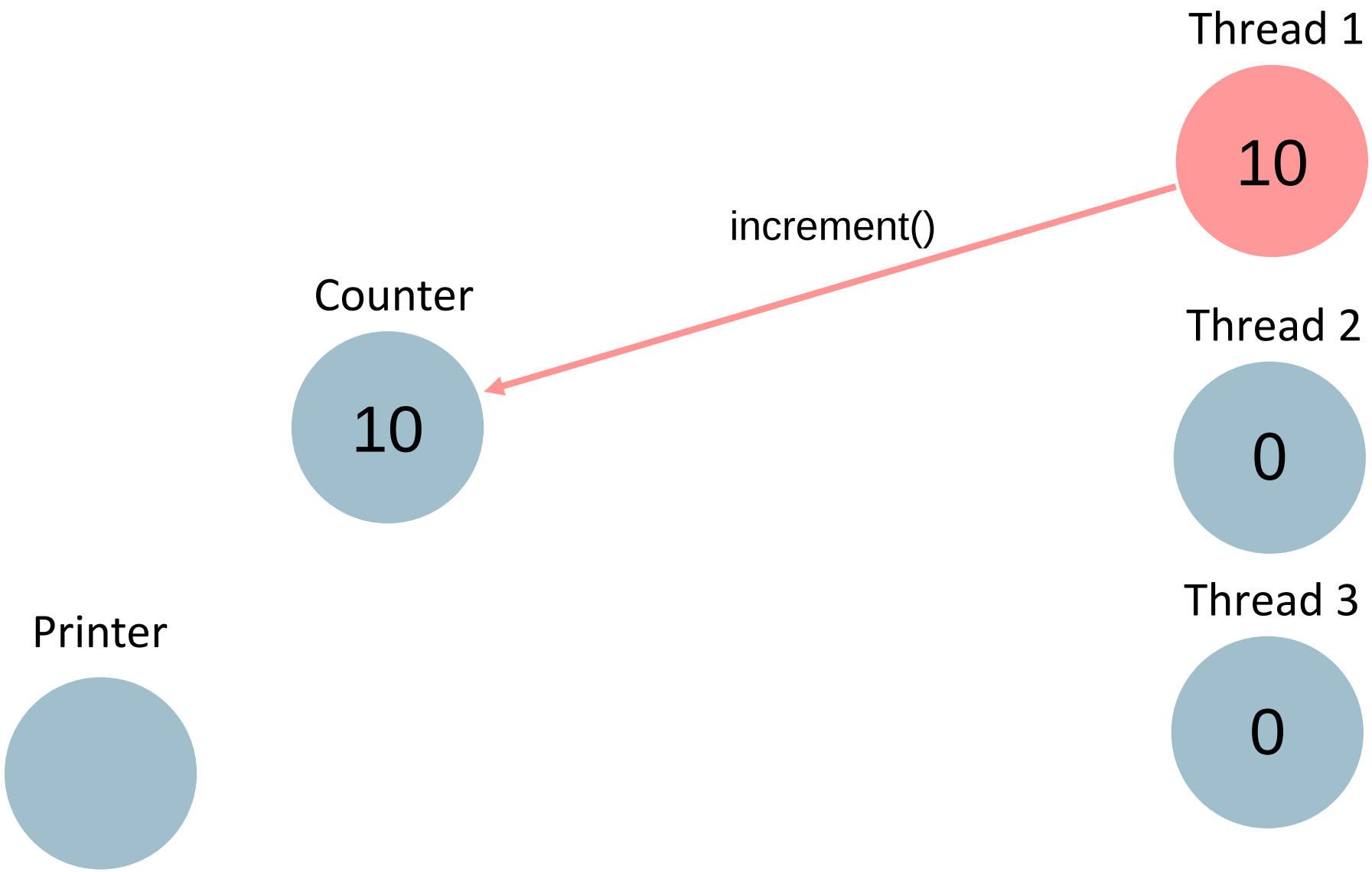
Task F – Atomic vs Synchronized counter

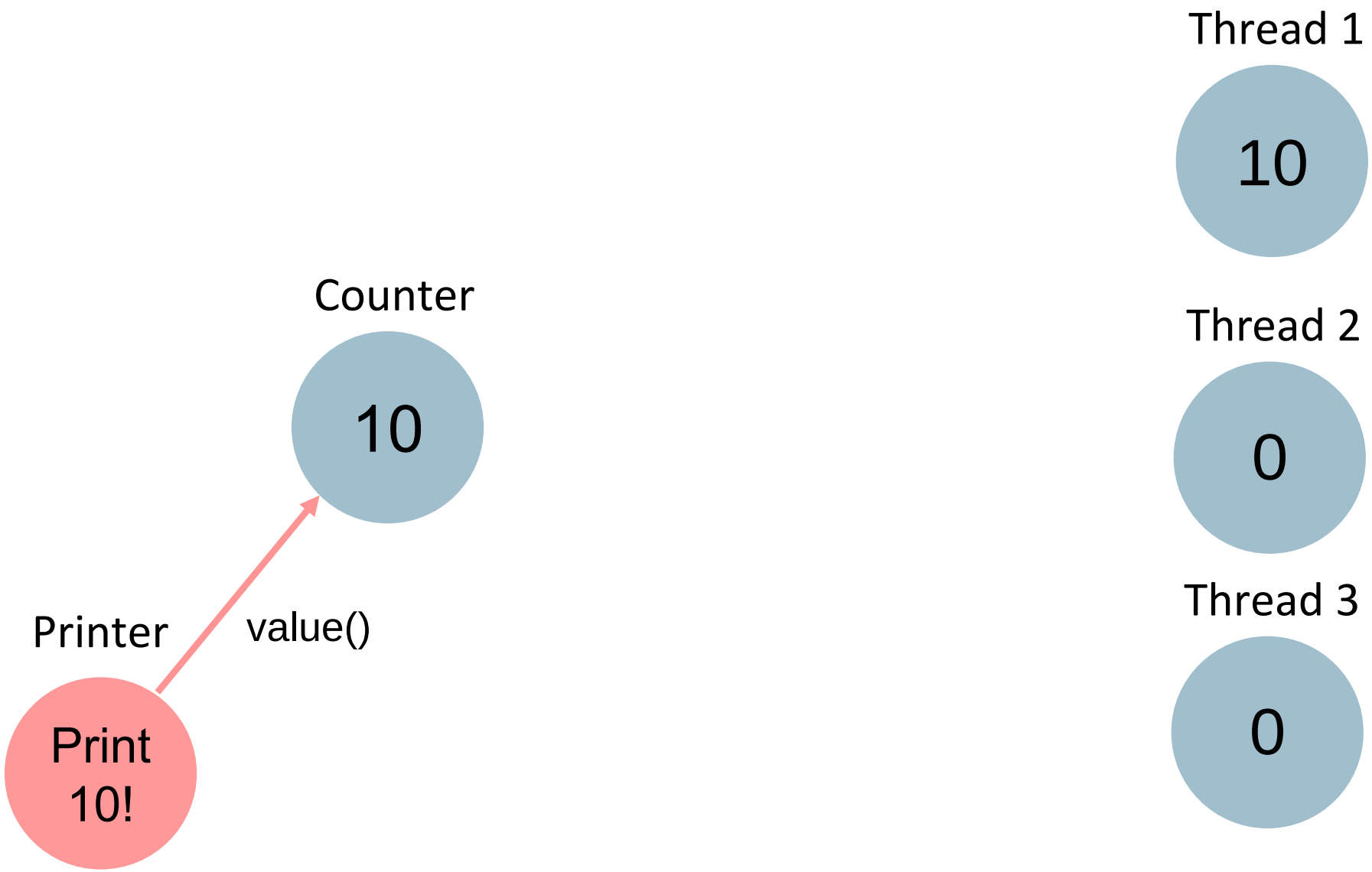
Experimentally compare the AtomicCounter and SynchronizedCounter implementations by measuring which one is faster. Observe the differences in the CPU load between the two versions. Can you explain what is the cause of different performance characteristics?

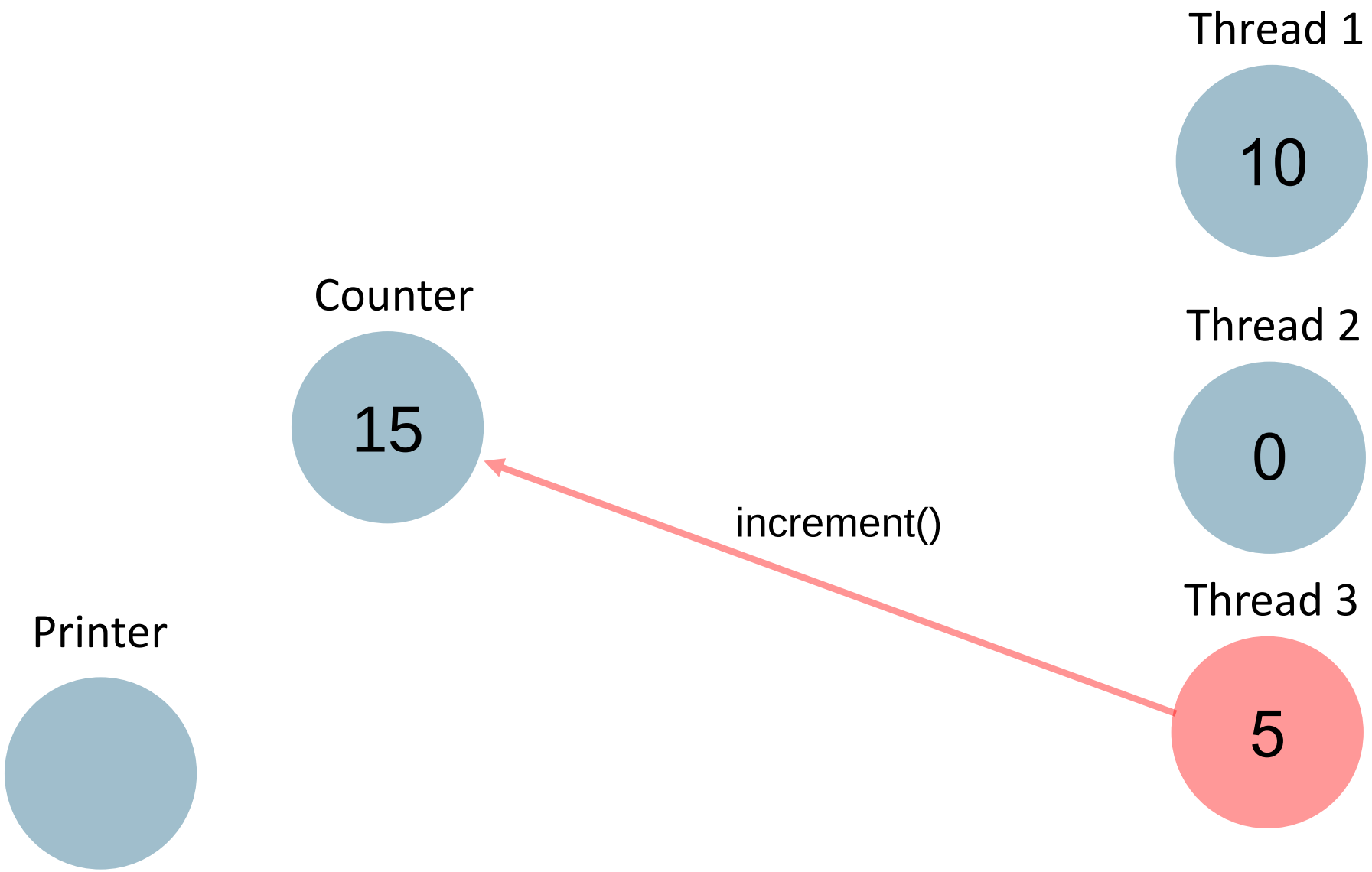
- Vary the load per thread
- Vary the number of threads

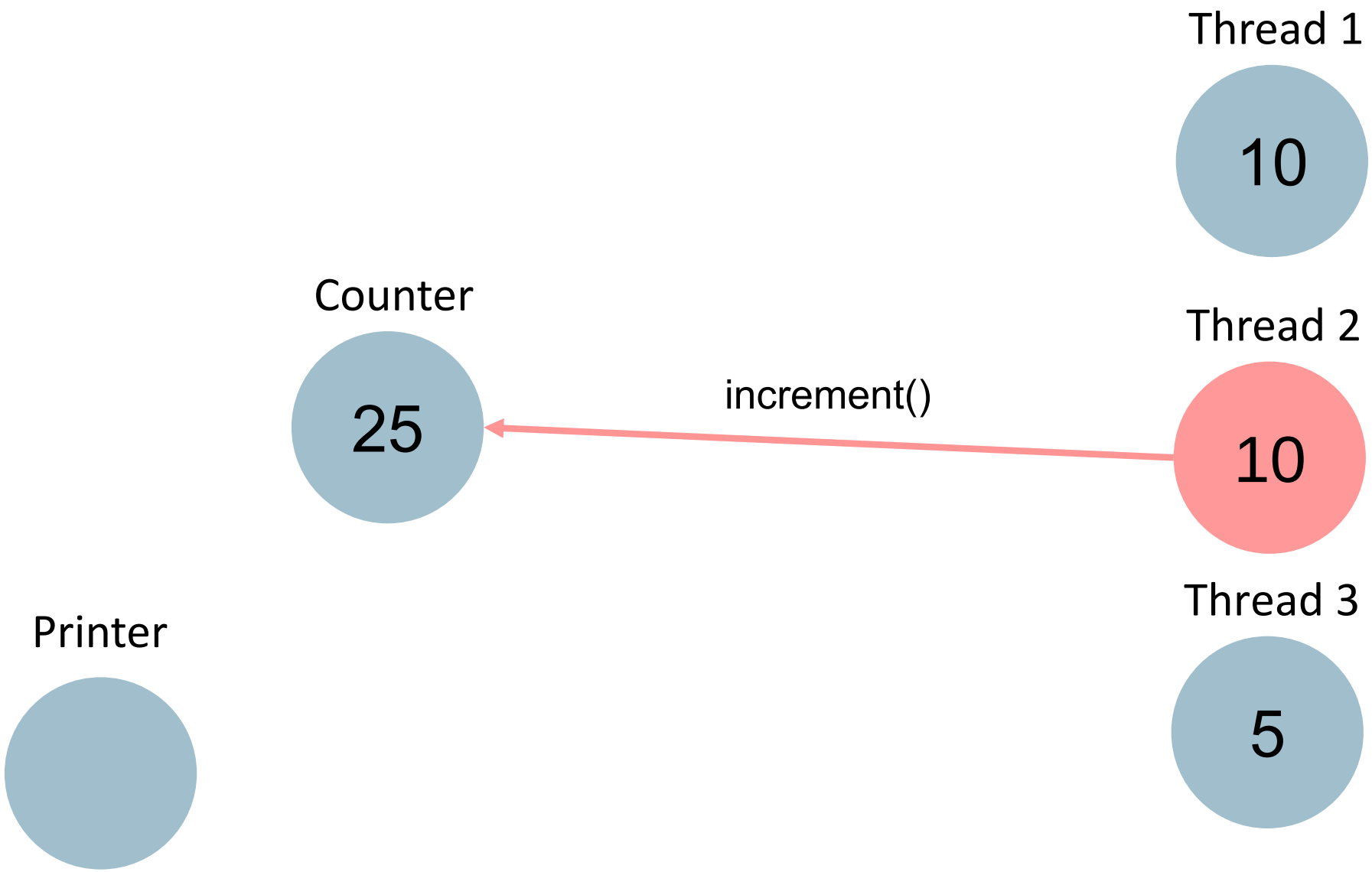
Task G

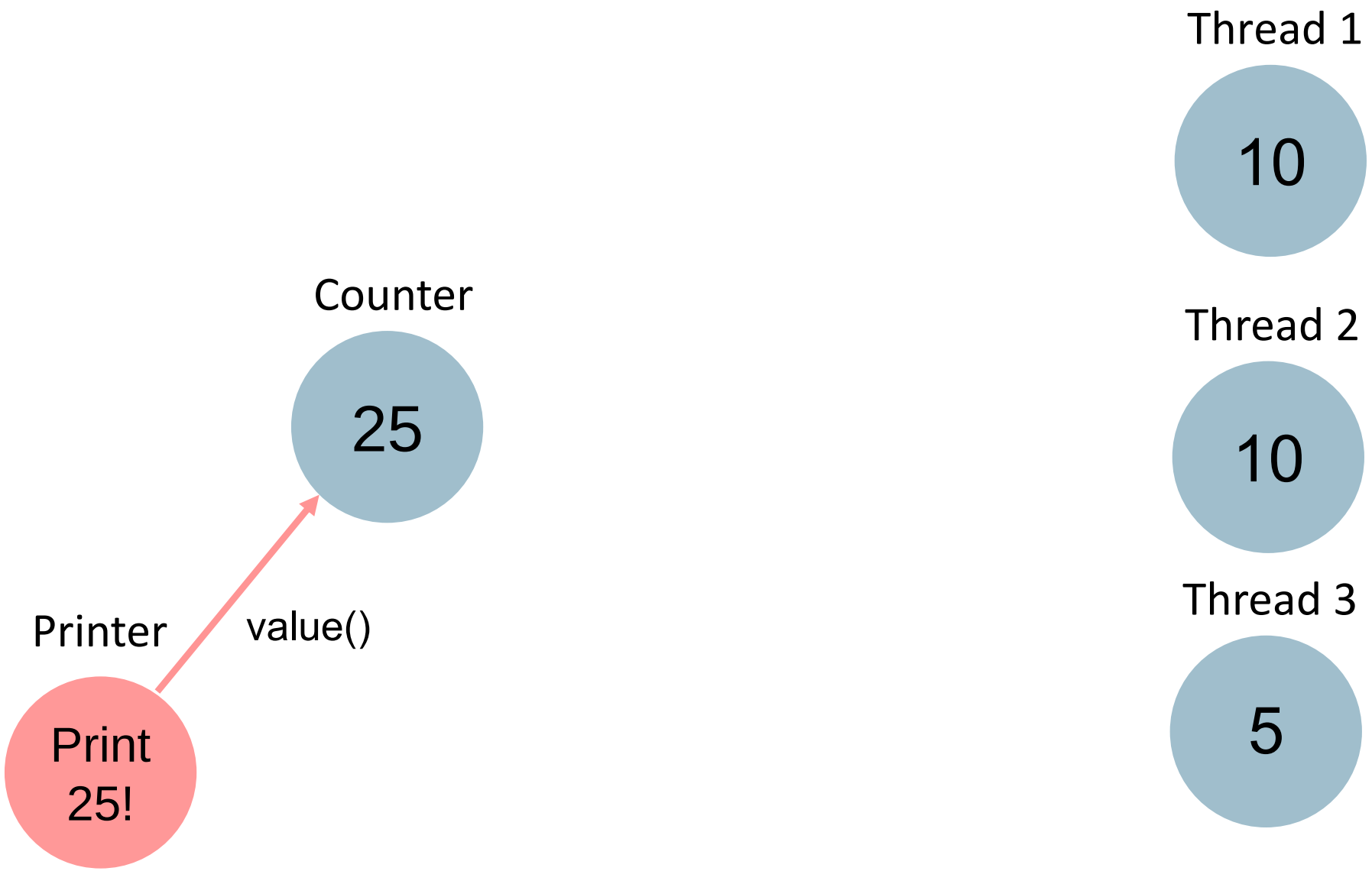
Implement a thread that measures execution progress. That is, create a thread that observes the values of the Counter during the execution and prints them to the console. Make sure that the thread is properly terminated once all the work is done [thread.interrupt()].

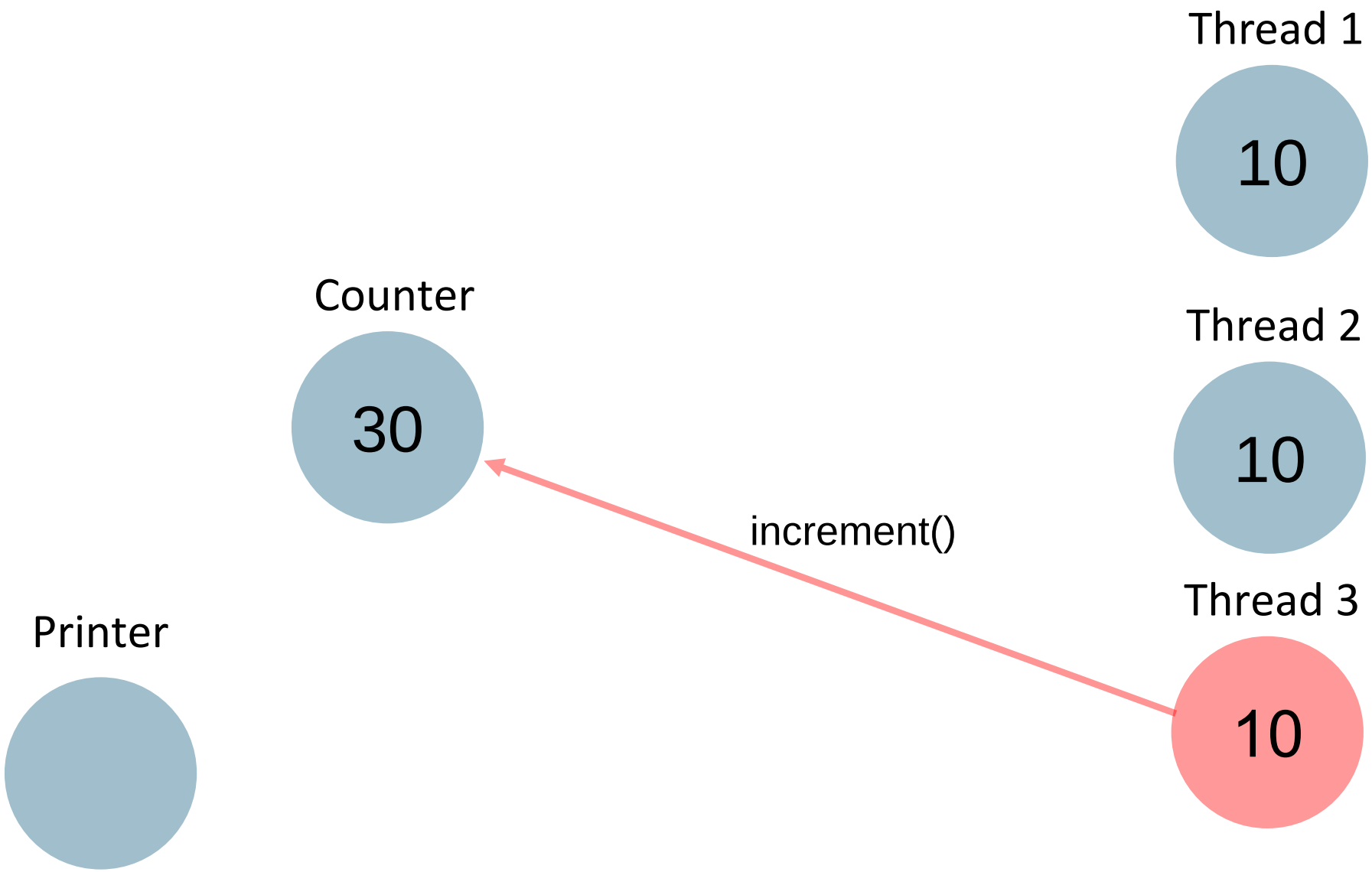

Thread 1
10
increment()
Counter
10
Thread 2
0
Thread 3
0
Printer

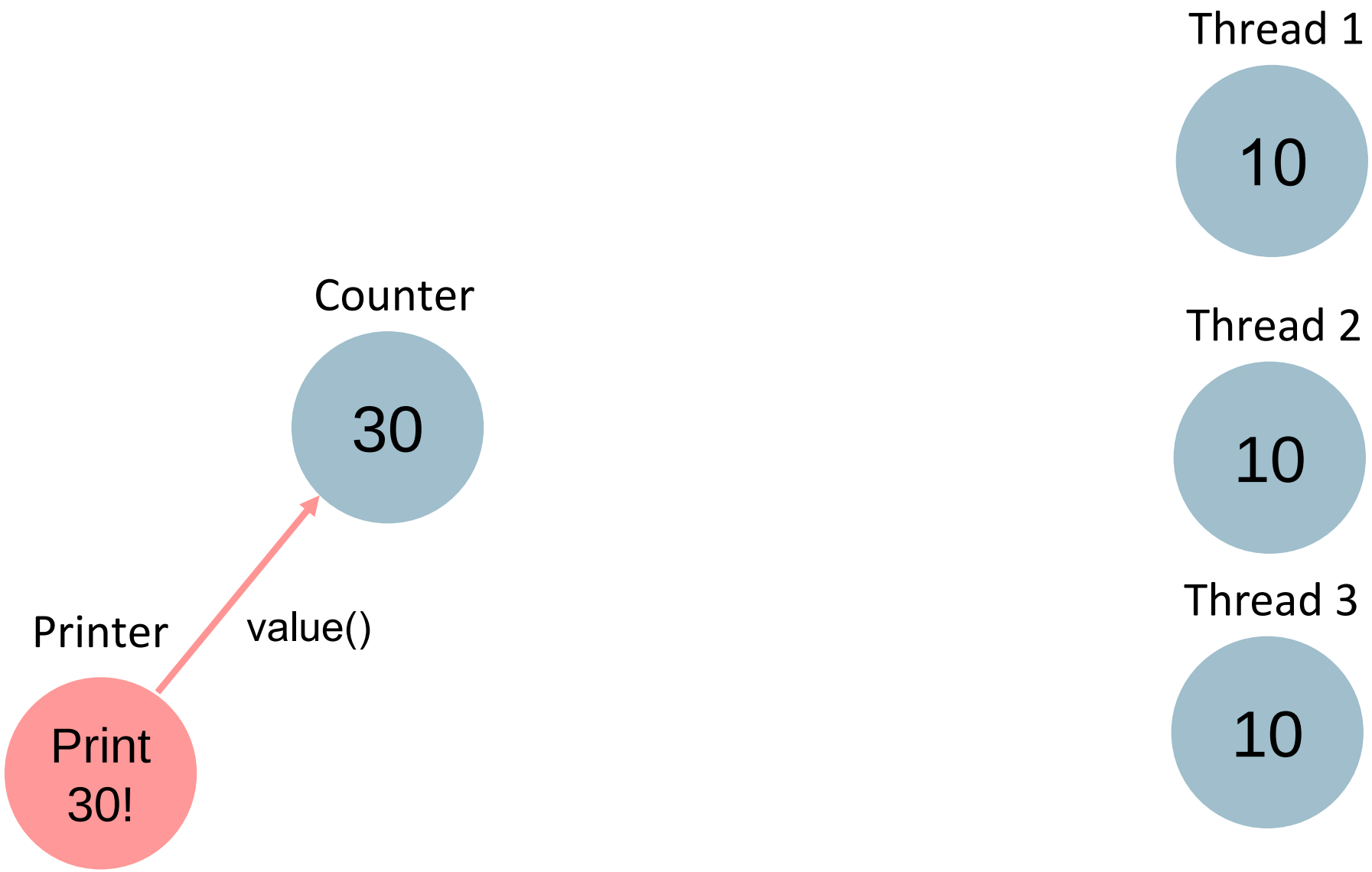

Thread 1
10
Thread 2
0
Thread 3
0
Counter
10
Printer
value()
Print
10!


Thread 1
10
Counter
15
Thread 2
0
increment()
Thread 3
5
Printer


Thread 1
10
Counter
25
increment()
Thread 2
10
Thread 3
5
Printer


Thread 1
10
Thread 2
10
Thread 3
5
Counter
25
Printer
value()
Print
25!


Thread 1
10
Counter
30
Thread 2
10
increment()
Thread 3
10
Printer


Thread 1
10
Thread 2
10
Thread 3
10
Counter
30
Printer
value()
Print
30!

Plan für heute

- Organisation
- Nachbesprechung Exercise 2
- Theory Recap
- Intro Exercise 3
- **Exam Questions**
- Kahoot

Past Exam Task

Kreuzen Sie alle korrekten Aussagen über die Ausführung von Java `Threads` an.

Mark all correct statements regarding the execution of Java `Threads`.

- ○ Die start() Methode in `t = new Thread(); t.start()` ruft automatisch auch die run() methode auf.

  The start() method in `t = new Thread(); t.start()` automatically also calls the run() method.

- ○ Die run() Methode in `t = new Thread(); t.run()` erzeugt einen neuen Thread und führt diesen aus.

  The run() method in `t = new Thread(); t.run()` creates a new thread and executes the thread.

- ○ Ein Codeblock mit mehreren Threads wird immer deterministisch ausgeführt. D.h. der Output ist immer exakt der gleiche.

  A codeblock with several threads is always executed deterministically. That means the output is always the same.

- ○ Ein komplett serieller Codeblock kann zur Beschleunigung auf mehreren Prozessoren ausgeführt werden.

  A fully serial block of code can be run on multiple processors to speedup execution.

Rep. Exam, FS 2023

Past Exam Task

Kreuzen Sie alle korrekten Aussagen über die Ausführung von Java `Threads` an.

Mark all correct statements regarding the execution of Java `Threads`.

- √ **Die start() Methode in `t = new Thread(); t.start()` ruft automatisch auch die run() methode auf.**

  The start() method in `t = new Thread(); t.start()` automatically also calls the run() method.

- ○ Die run() Methode in `t = new Thread(); t.run()` erzeugt einen neuen Thread und führt diesen aus.

  The run() method in `t = new Thread(); t.run()` creates a new thread and executes the thread.

- ○ Ein Codeblock mit mehreren Threads wird immer deterministisch ausgeführt. D.h. der Output ist immer exakt der gleiche.

  A codeblock with several threads is always executed deterministically. That means the output is always the same.

- ○ Ein komplett serieller Codeblock kann zur Beschleunigung auf mehreren Prozessoren ausgeführt werden.

  A fully serial block of code can be run on multiple processors to speedup execution.

Rep. Exam, FS 2023

Past Exam Task

(c) Wozu dient die `join()` Methode in Java `Threads`? *What is the purpose of the `join()` method in Java `Threads`?.* (2)

- ◯ Um eine Prioritätenreihenfolge zwischen mehreren Threads zu erzwingen. *To enforce a priority order among multiple threads.*
- ◯ Um das von dem aktuellen Thread gehaltene `Lock` freizugeben. *To release the lock held by the current thread.*
- ◯ Um die Ausführung des aktuellen Threads anzuhalten, bis der Thread, den er `joined`, abgeschlossen ist. *To pause the current thread's execution until the thread it `joins` completes.*
- ◯ Um die Kontrolle an einen anderen Thread zu übergeben, ohne auf dessen Abschluss zu warten. *To transfer control to another thread without waiting for its completion.*

Rep. Exam, FS 2023

Past Exam Task

(c) Wozu dient die `join()` Methode in Java `Threads`? *What is the purpose of the `join()` method in Java `Threads`?.* (2)

- ○ Um eine Prioritätenreihenfolge zwischen mehreren Threads zu erzwingen. *To enforce a priority order among multiple threads.*
- ○ Um das von dem aktuellen Thread gehaltene `Lock` freizugeben. *To release the lock held by the current thread.*
- √ **Um die Ausführung des aktuellen Threads anzuhalten, bis der Thread, den er `joined`, abgeschlossen ist.** *To pause the current thread's execution until the thread it `joins` completes.*
- ○ Um die Kontrolle an einen anderen Thread zu übergeben, ohne auf dessen Abschluss zu warten. *To transfer control to another thread without waiting for its completion.*

Rep. Exam, FS 2023

iv.	In Java kann ein Thread sich ein Monitor Lock eines Objekts nur einmal erwerben.	*In Java, a thread can acquire the monitor lock of an object only once.*	(1)

True False

iv. In Java kann ein Thread sich ein Monitor Lock eines Objekts nur einmal erwerben.

In Java, a thread can acquire the monitor lock of an object only once. (1)

True False

Locks sind reentrant!

vi. Wenn ein Java Thread versucht in einen `synchronized` Block einzutreten, welcher bereits von einem anderen Thread ausgeführt wird, geht der Thread in den `BLOCKED` Zustand.

When a Java thread tries to enter a ***`synchronized`*** *block which is already being executed by another thread, the thread enters the* ***`BLOCKED`*** *state.*

(1)

True False

vi. Wenn ein Java Thread versucht in einen `synchronized` Block einzutreten, welcher bereits von einem anderen Thread ausgeführt wird, geht der Thread in den `BLOCKED` Zustand.

When a Java thread tries to enter a ***`synchronized`*** *block which is already being executed by another thread, the thread enters the* ***`BLOCKED`*** *state.*

(1)

True False

viii. Die `notify()` Methode gibt dem Aufwecken von Threads, die am längsten warten, **keine** Priorität.

*The `notify()` method does **not** prioritise waking up threads which have waited the longest.* (1)

True False

viii. Die `notify()` Methode gibt dem Aufwecken von Threads, die am längsten warten, **keine** Priorität.

*The `notify()` method does **not** prioritise waking up threads which have waited the longest.* (1)

True False

vii. Ein Thread, der auf `Object obj` wartet, kann **nur** aufgeweckt werden, wenn ein anderer Therad `obj.notify()` oder `obj.notifyAll()` aufruft.

*A thread waiting on `Object obj` can **only** be woken up when another thread calls the `obj.notify()` or `obj.notifyAll()` methods.*

(1)

True False

vii. Ein Thread, der auf `Object obj` wartet, kann **nur** aufgeweckt werden, wenn ein anderer Therad `obj.notify()` oder `obj.notifyAll()` aufruft.

*A thread waiting on `Object obj` can **only** be woken up when another thread calls the `obj.notify()` or `obj.notifyAll()` methods.* (1)

True False

Spurious wake-ups! Deswegen ist wait() immer in einer while loop.


QUIZ

Feedback

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

Danke

- Bis nächste Woche!