Parallel Programming
Exercise Session 3

Spring 2026

anguhl@ethz.ch
annikaguhl.com

Today

Post-Discussion Exercise 2 15’

Pre-Discussion Exercise 3 30’

Break 5’

Theory Recap 10’

Old exam question 10’

Quiz 10’

Past Exam Task

Kreuzen Sie alle korrekten Aussagen über das Erstellen von Java `Threads` an.	*Mark all correct statements regarding the creation of Java `Threads`.*
○ Beim Aufteilen eines Workloads sollte man soviele Threads erstellen wie möglich, bis nur noch elementare Operationen pro Thread ausgeführt werden.	*When splitting a workload, as many threads as possible should be created until only elementary operations are performed per thread.*
√ **Um eine eigene Thread-Klasse in Java zu definieren kann man das Runnable-Interface implementieren.**	*To define a custom thread class in Java, one can implement the Runnable interface.*
√ **Um eine eigene Thread-Klasse in Java zu definieren kann man die Thread-Klasse erweitern.**	*To define a custom thread class in Java, one can extend the Thread class.*
○ Threads werden fast ausschliesslich genutzt um eine rekursive Implementation zu beschleunigen.	*Threads are used almost exclusively to speed up a recursive implementation.*

Exam, HS 2023

Past Exam Task

Kreuzen Sie alle korrekten Aussagen über das Erstellen von Java `Threads` an.	*Mark all correct statements regarding the creation of Java `Threads`.*
○ Beim Aufteilen eines Workloads sollte man soviele Threads erstellen wie möglich, bis nur noch elementare Operationen pro Thread ausgeführt werden.	*When splitting a workload, as many threads as possible should be created until only elementary operations are performed per thread.*
√ **Um eine eigene Thread-Klasse in Java zu definieren kann man das Runnable-Interface implementieren.**	*To define a custom thread class in Java, one can implement the Runnable interface.*
√ **Um eine eigene Thread-Klasse in Java zu definieren kann man die Thread-Klasse erweitern.**	*To define a custom thread class in Java, one can extend the Thread class.*
○ Threads werden fast ausschliesslich genutzt um eine rekursive Implementation zu beschleunigen.	*Threads are used almost exclusively to speed up a recursive implementation.*

Unklar formuliert, man kann für wahr oder falsch argumentieren

In der Prüfung gibt es keine Aufgaben, die nur auf kleinen Details/Formulierungen basieren

Exam, HS 2023

Post-Discussion Exercise 2

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)

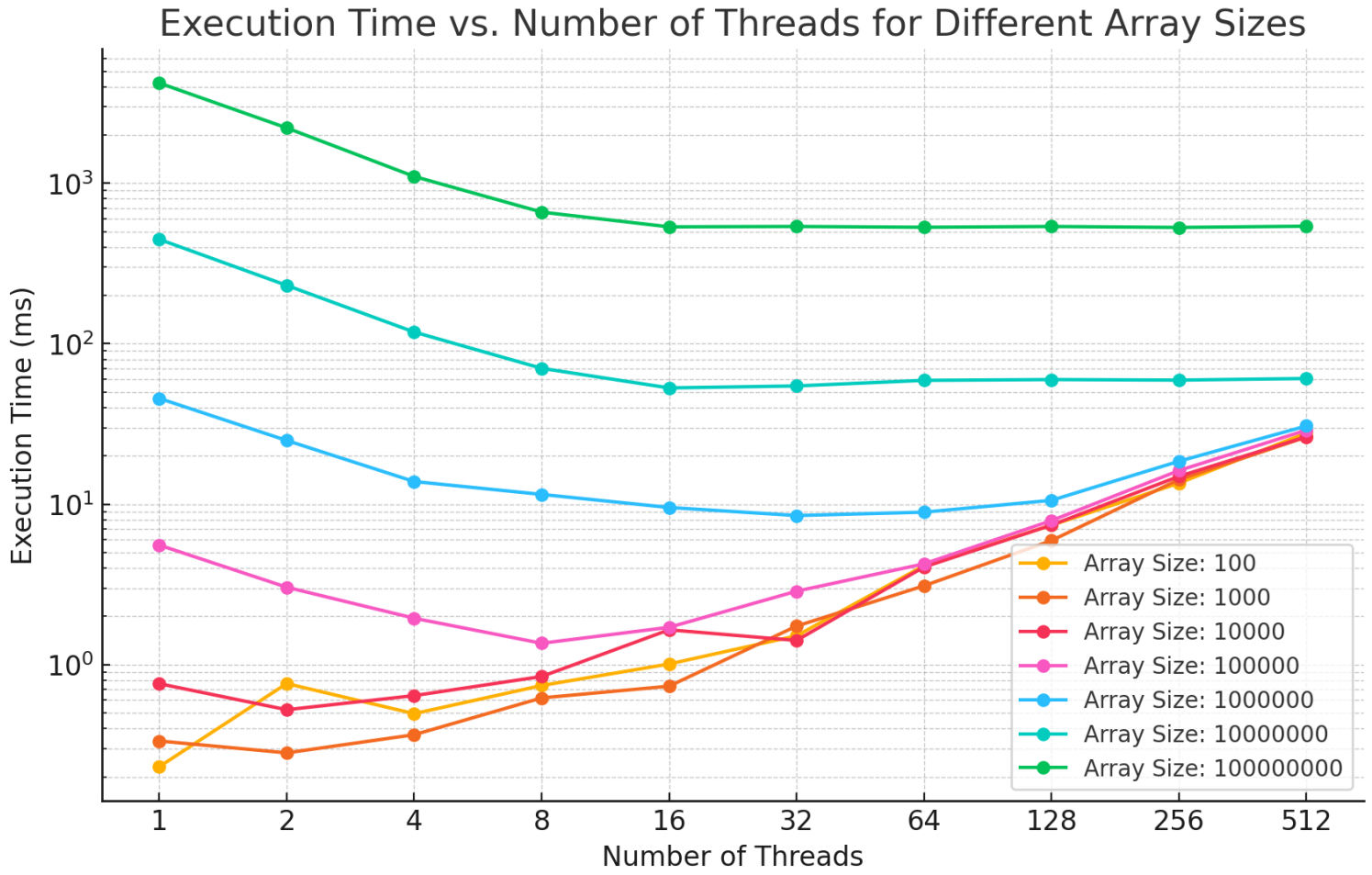
Task E: Sharing Data Across Threads

demo `SharedData`

(need this for E)

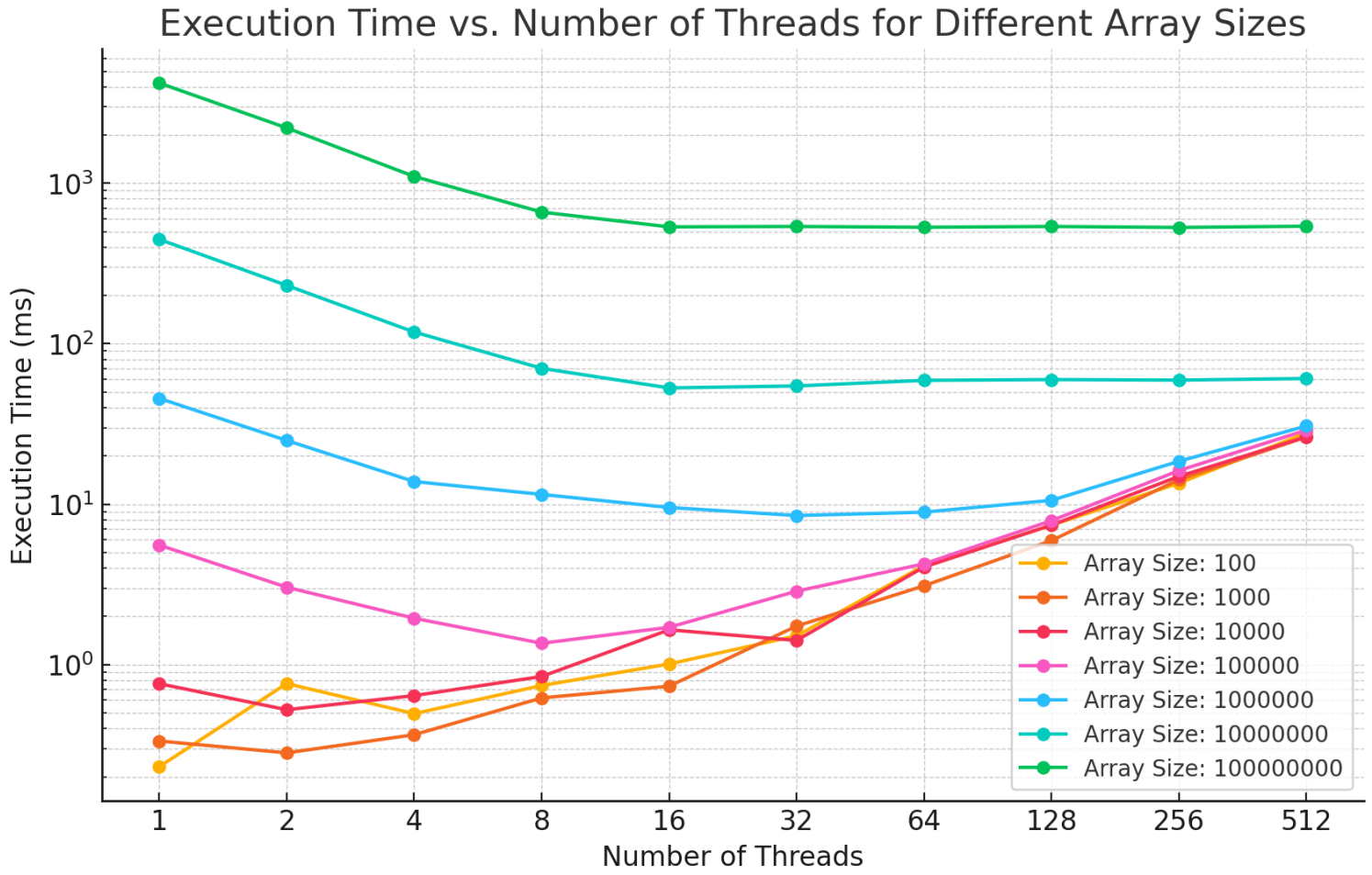
- Pass shared variable as argument to thread
- Inline thread creation with final variables

Task F: Execution Speed-Up



Experiment done on CPU with 16 cores available.

Task F: Execution Speed-Up



Experiment done on CPU with 16 cores available.

Task F: Execution Speed-Up

Small arrays: increasing number of threads does not improve performance due to thread management overhead.

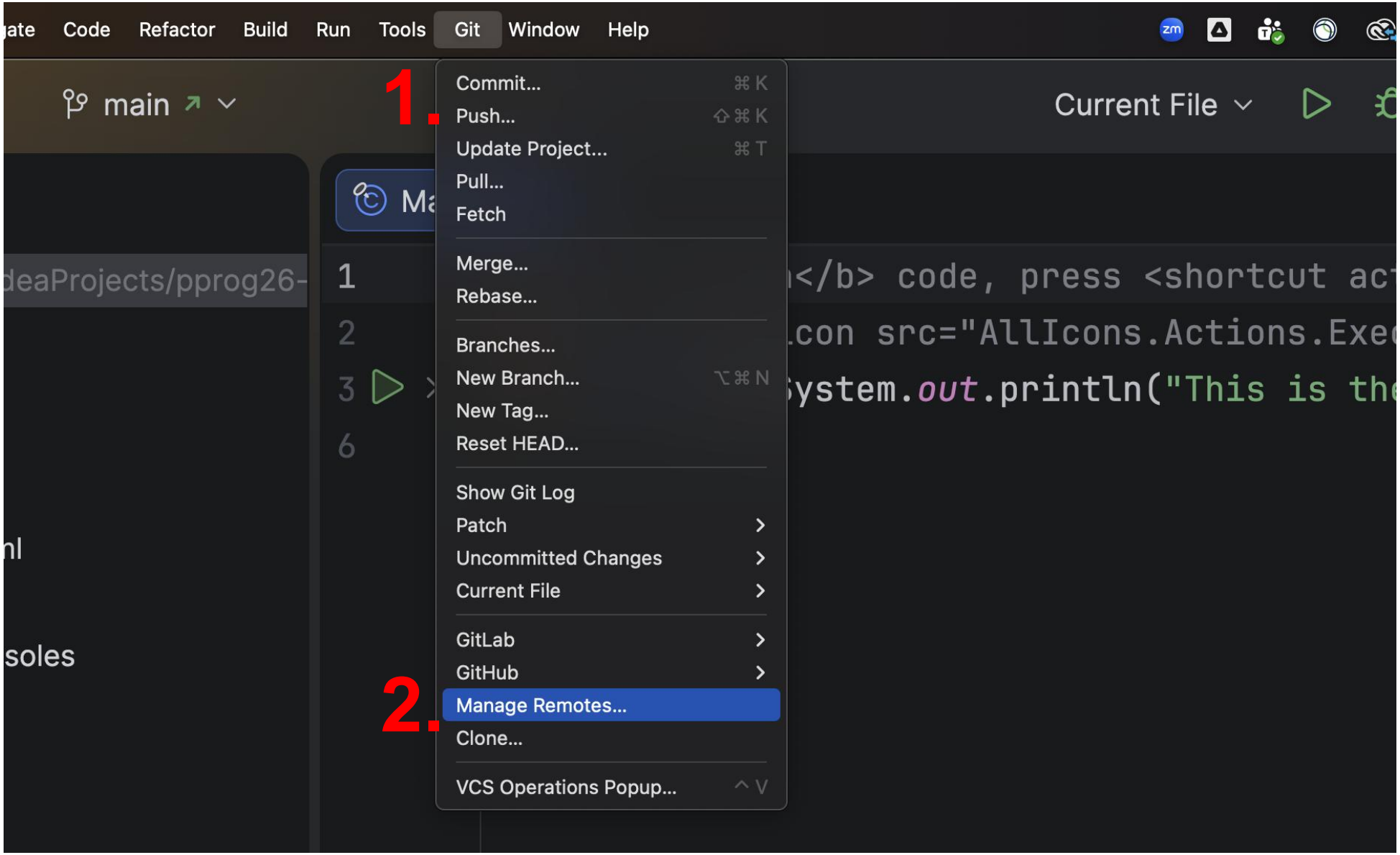
Large arrays: speed-up converges with a certain number of threads.

At very high thread counts, overhead dominates, causing execution time to increase.

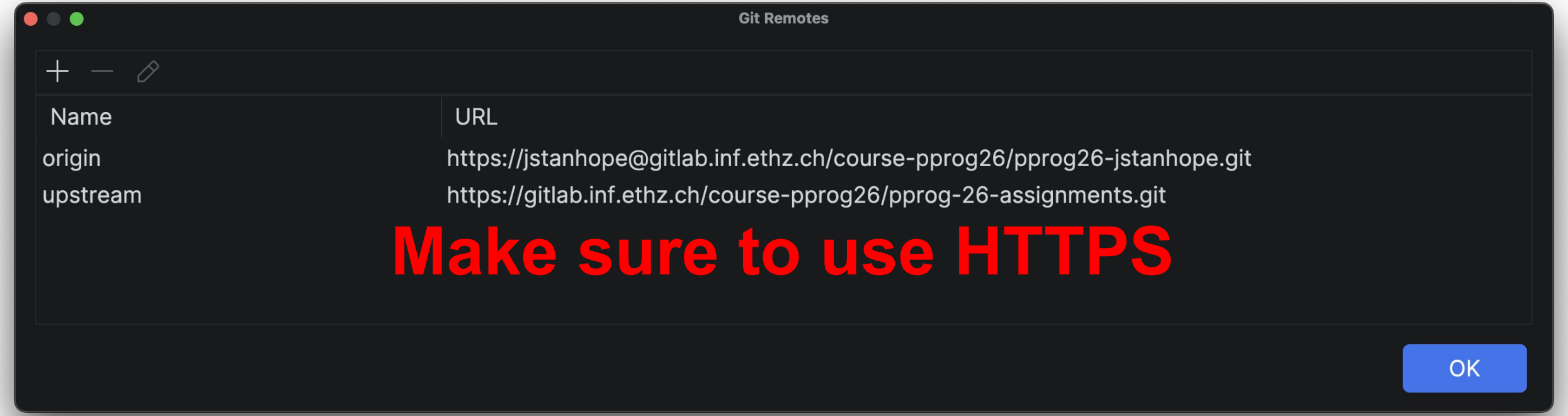
Exercise 2

Which task(s) do you want to discuss?

Managing Git Remotes



Managing Git Remotes



Pre-Discussion Exercise 3

Counter

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

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

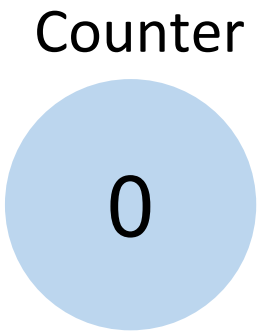
Counter

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

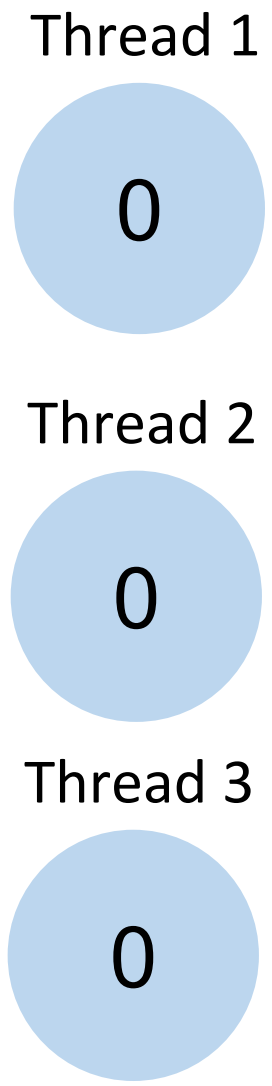
```
public interface Counter {
  public void increment();
  public int value();
}
```

```
// 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



value of the
shared Counter



number of times
`increment()` is called

Thread 1

0

Counter

0

Thread 2

0

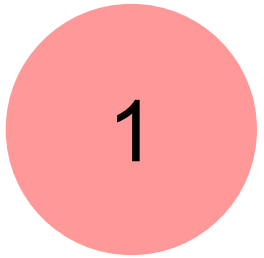
Thread 3

0

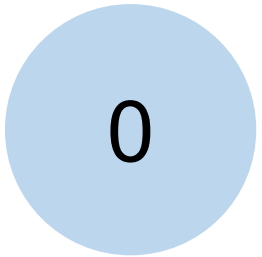
value of the
shared Counter

number of times
`increment()` is called

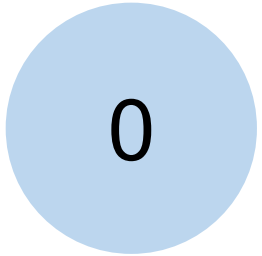
Thread 1



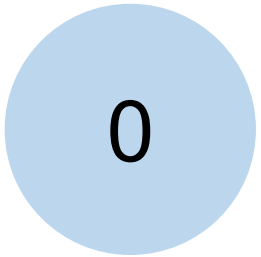
Counter



Thread 2

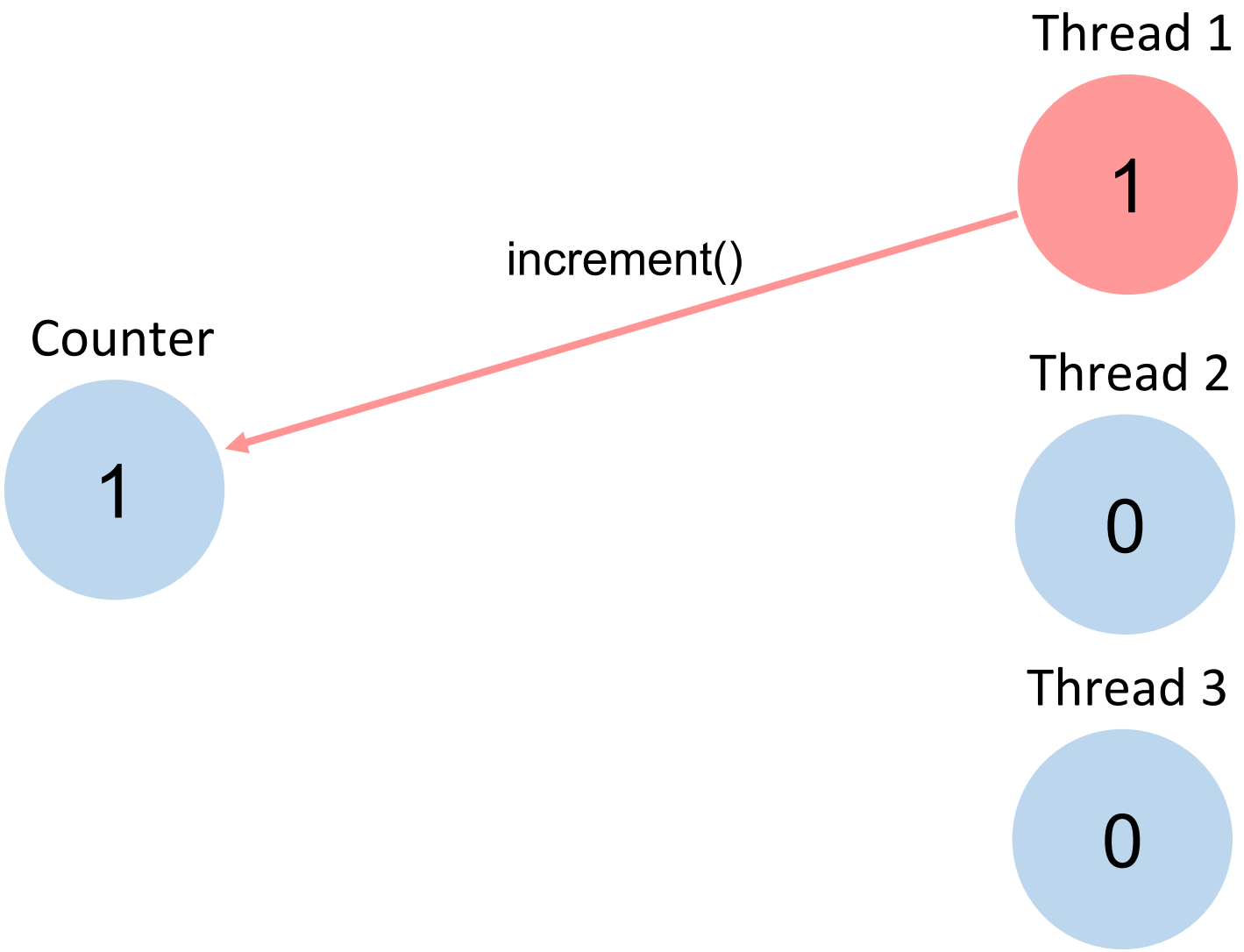


Thread 3



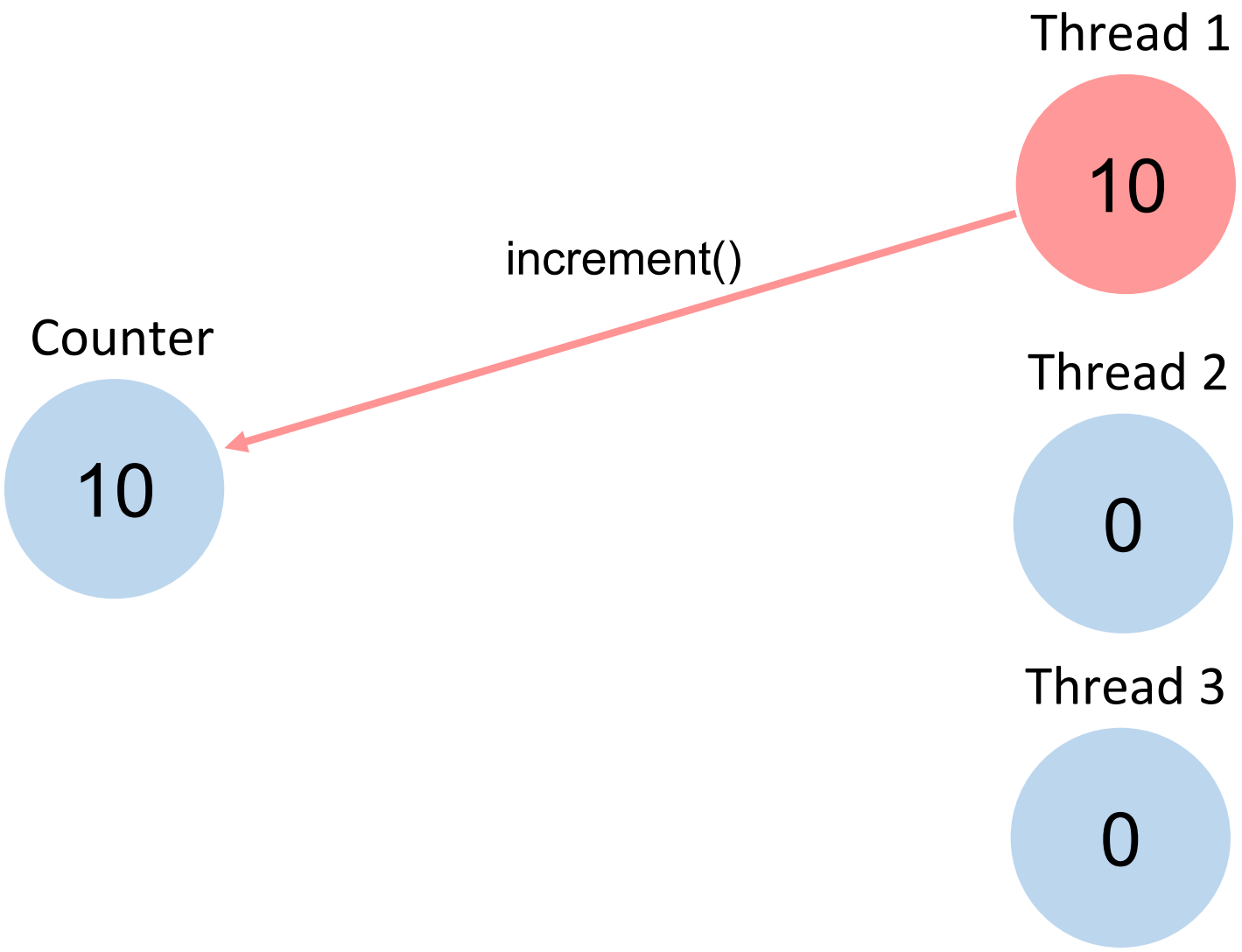
value of the
shared Counter

number of times
`increment()` is called



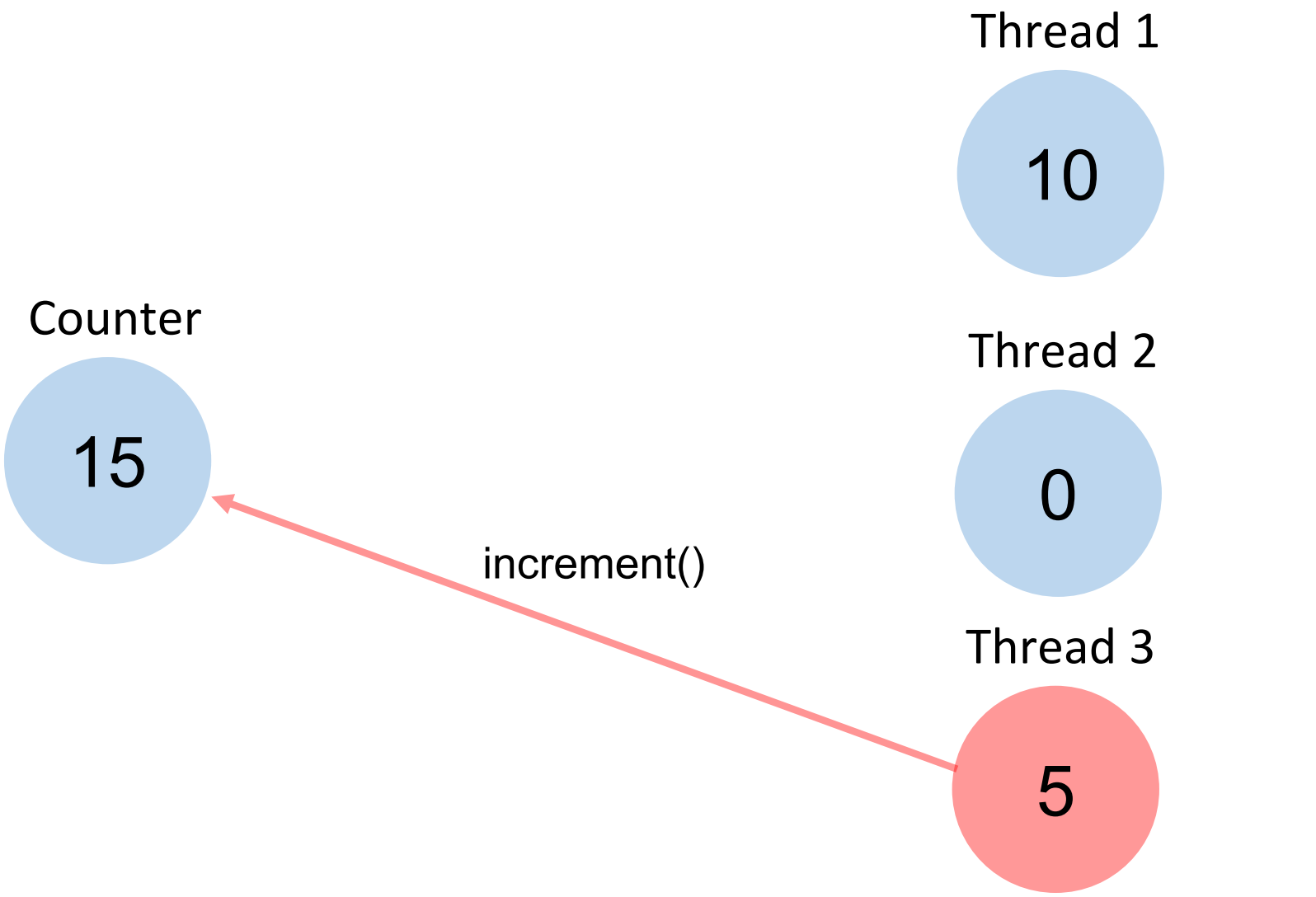
value of the
shared Counter

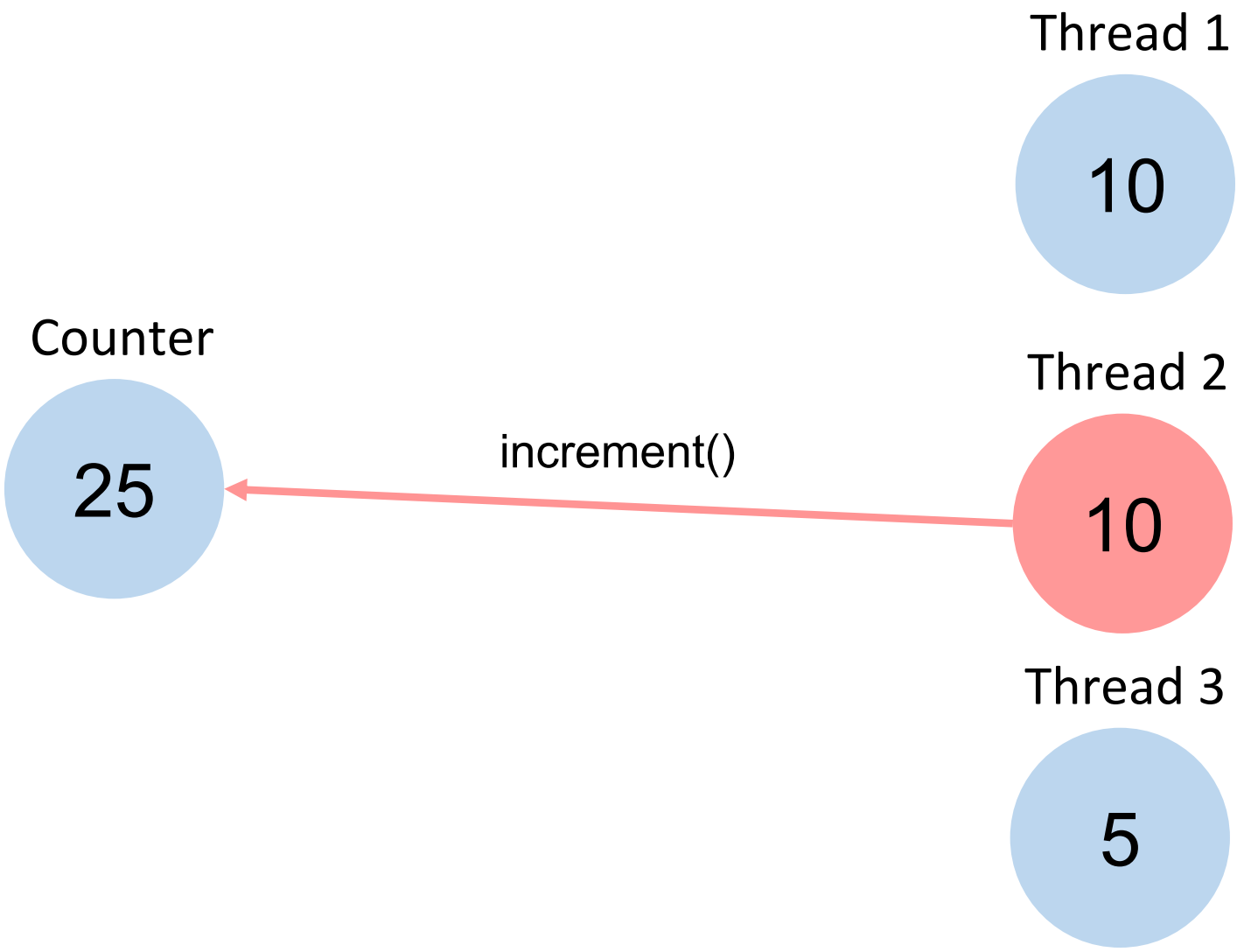
number of times
`increment()` is called



value of the
shared Counter

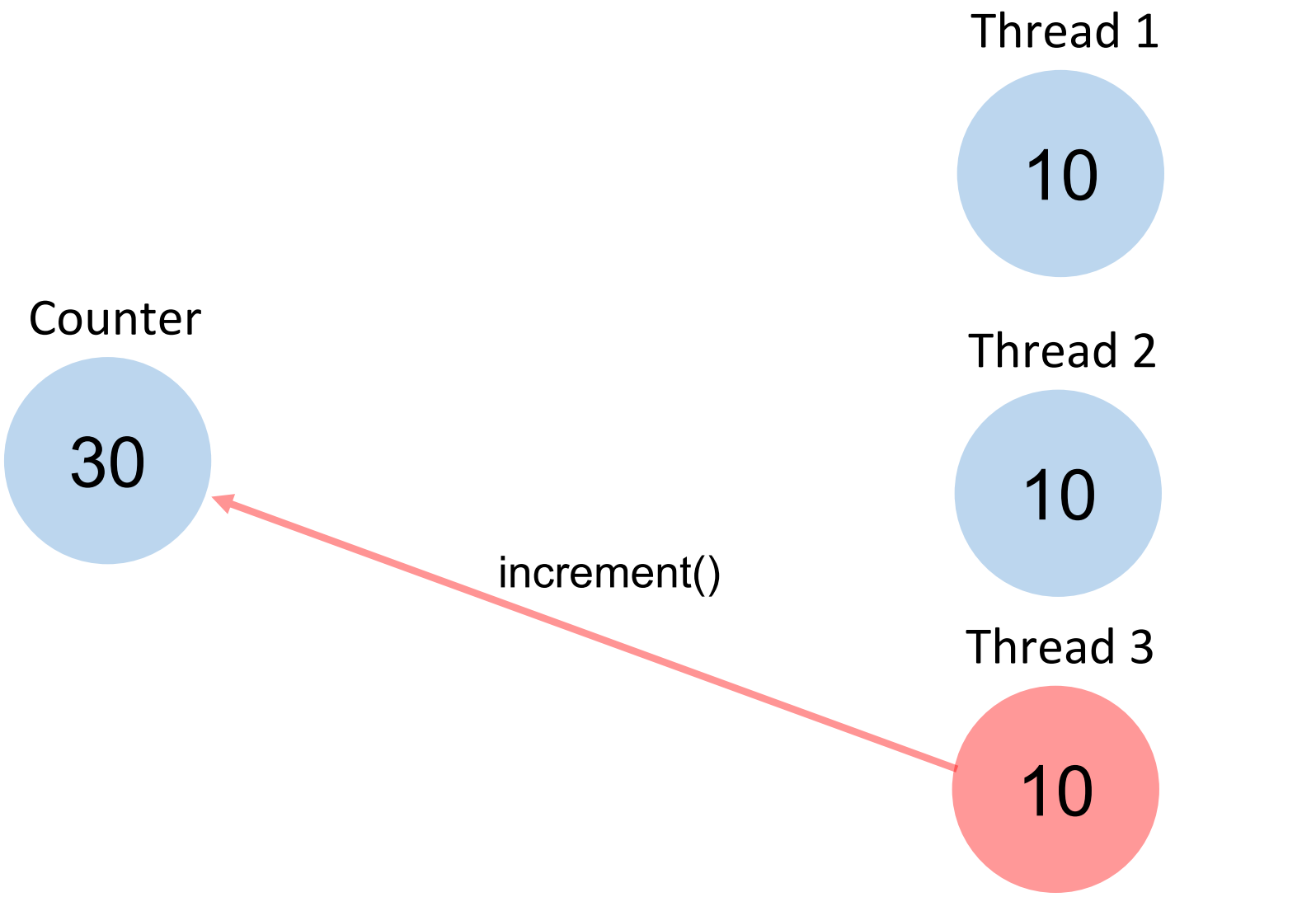
number of times
`increment()` is called

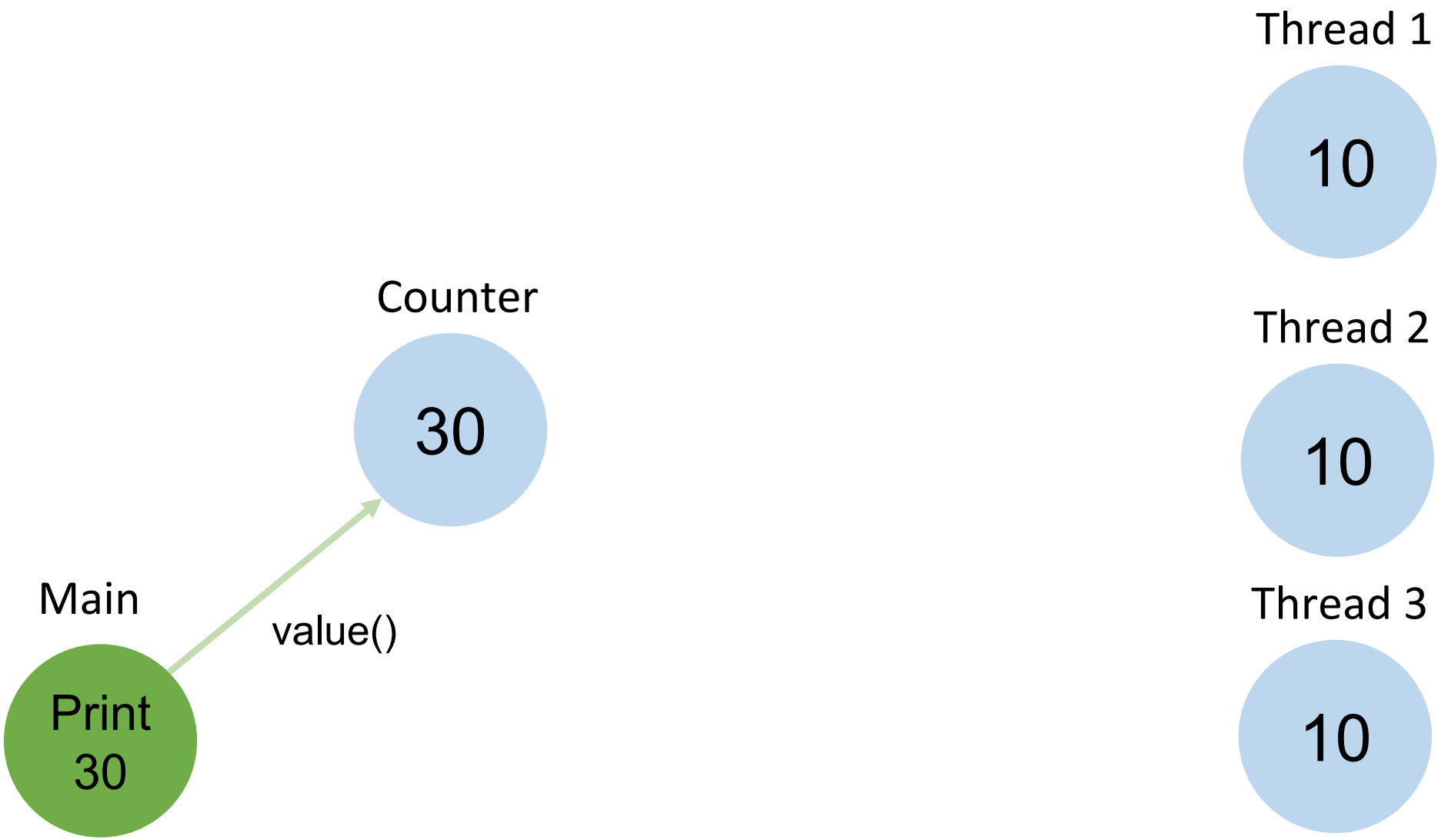

Thread 1
10
Counter
15
Thread 2
0
increment()
Thread 3
5
value of the
shared Counter
number of times
increment() is called



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?

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.

Synchronization

- ➔ Every reference type contains a lock inherited from the Object class
- ➔ Primitive fields can be locked only via their enclosing objects
  - ➔ Verwendet nicht die enclosing objects als locks! Es kann Probleme mit caching geben, und falls ihr den Wert überschreibt, ist es ein neues Objekt.
  - ➔ Verwendet stattdessen final Object lock = new Object()
- ➔ 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

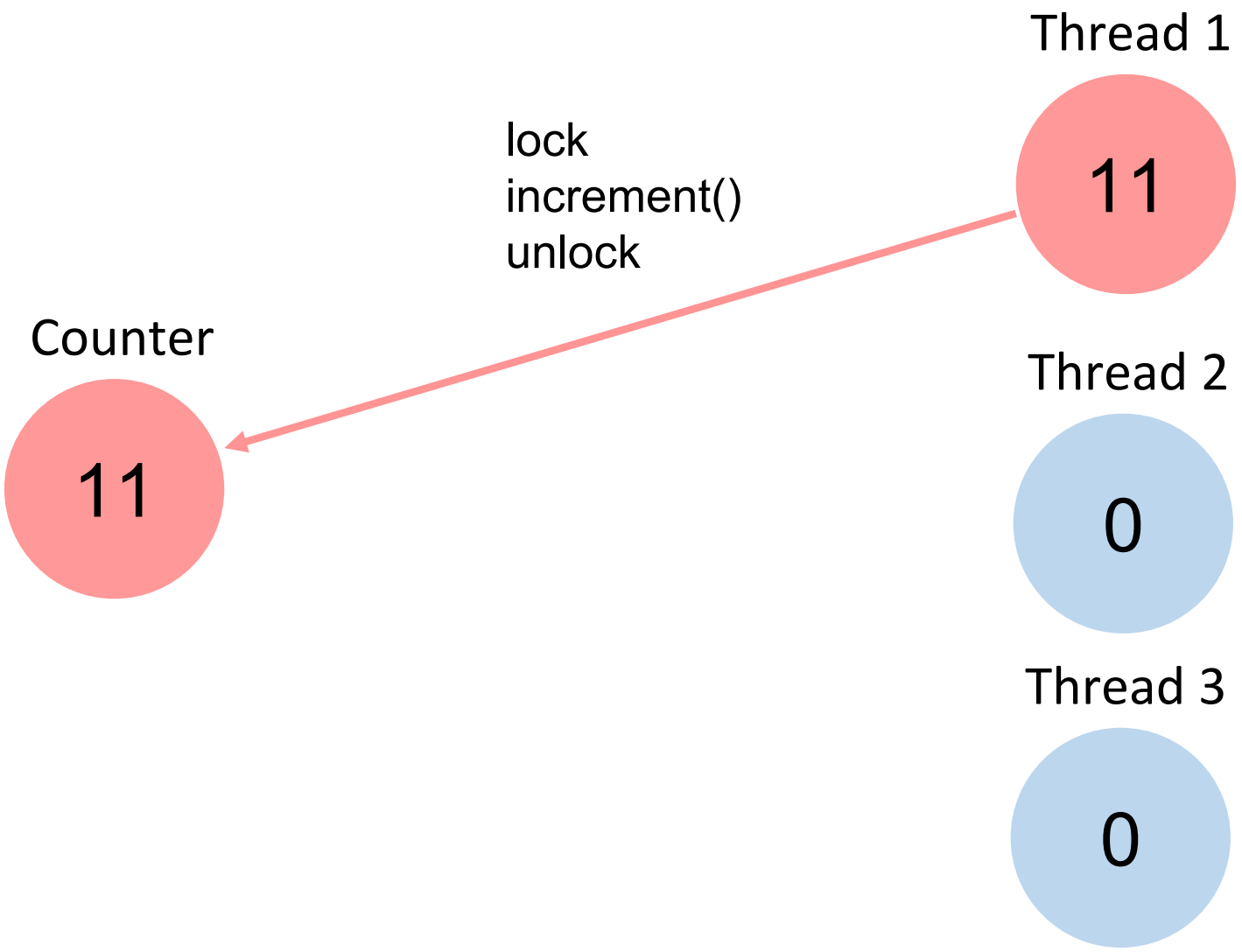
`foo.xMethod() //lock on foo`

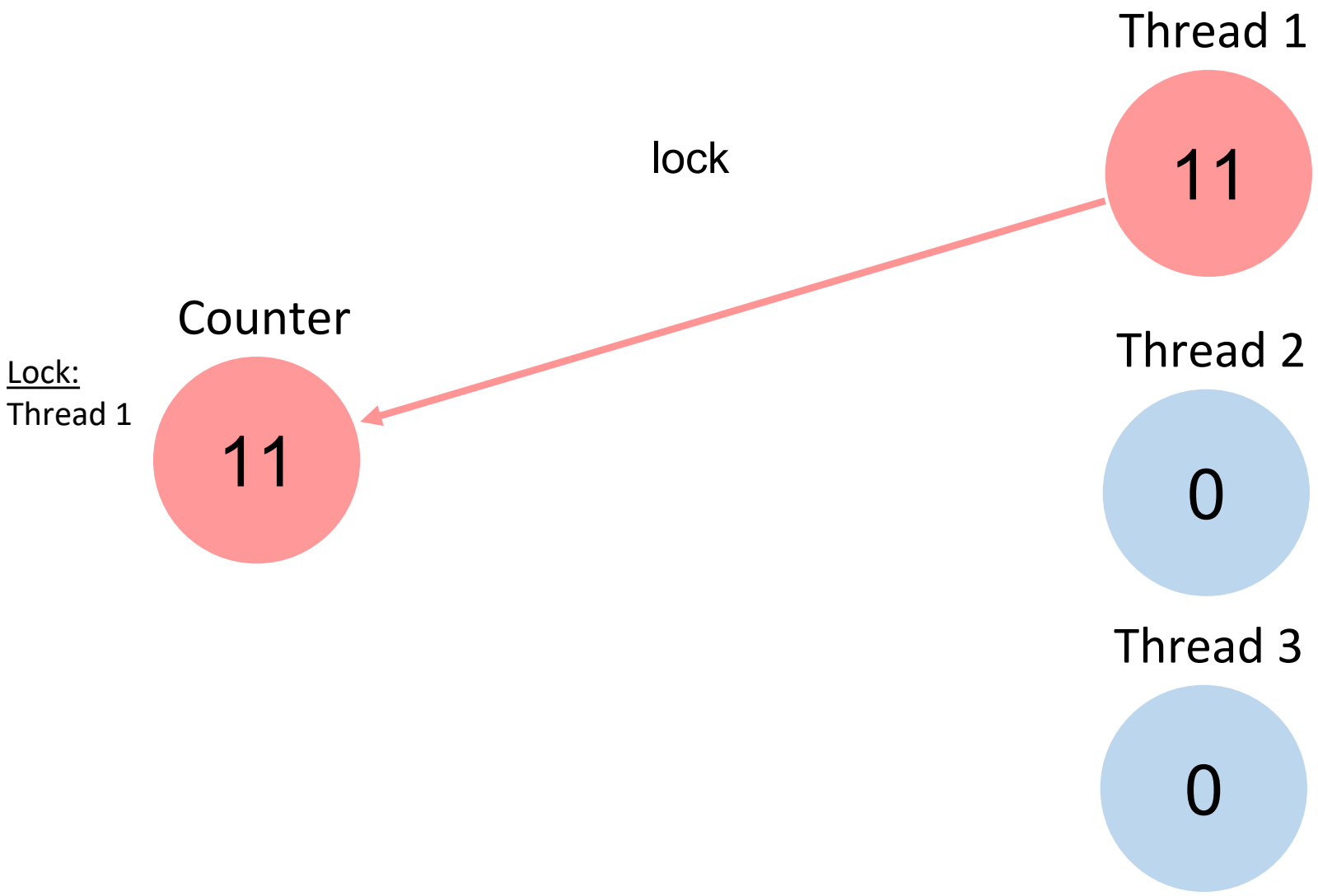
➔ Synchronized keyword obtains a lock on the parameter object

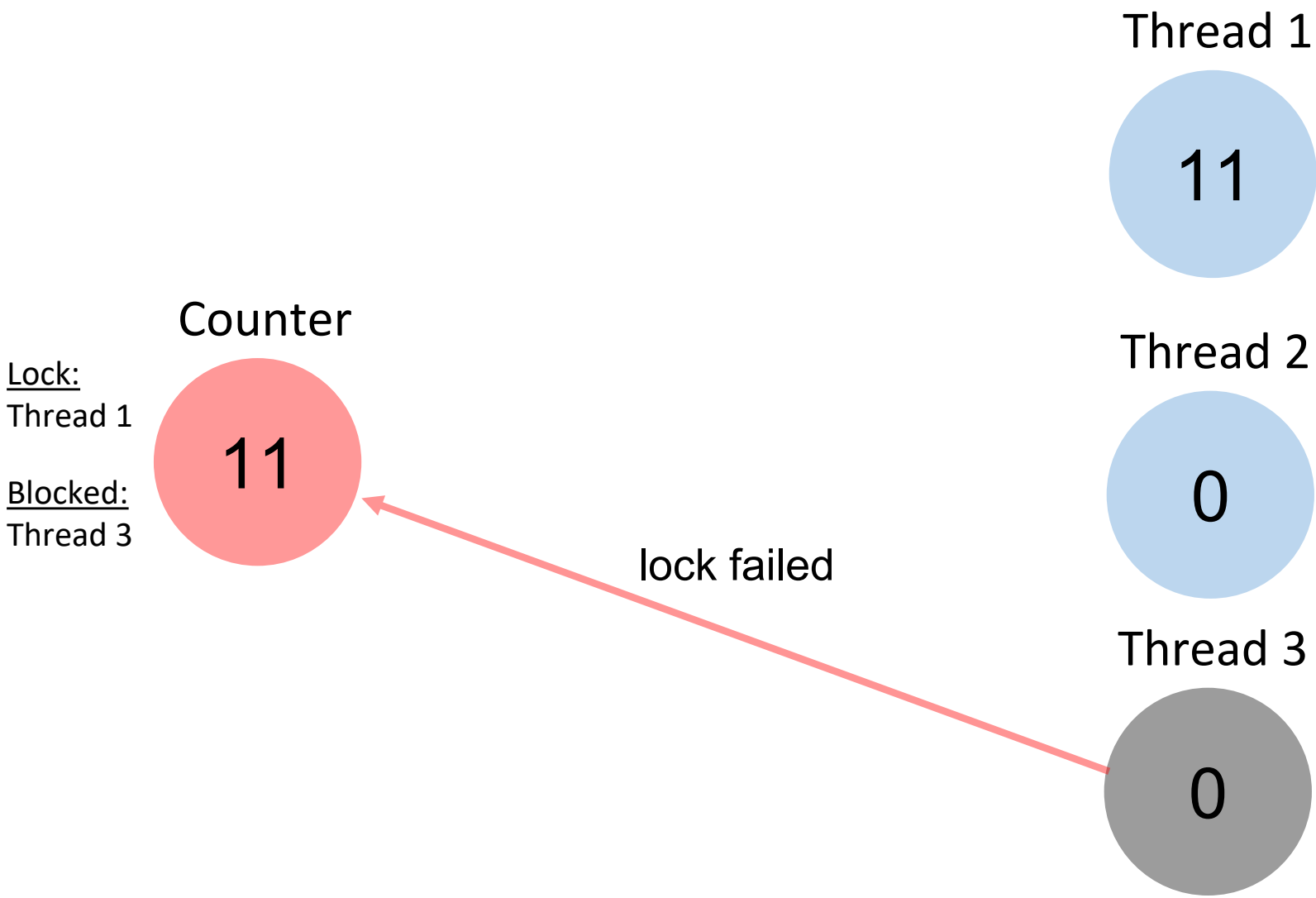
`synchronized (bar) { … } //lock on bar`

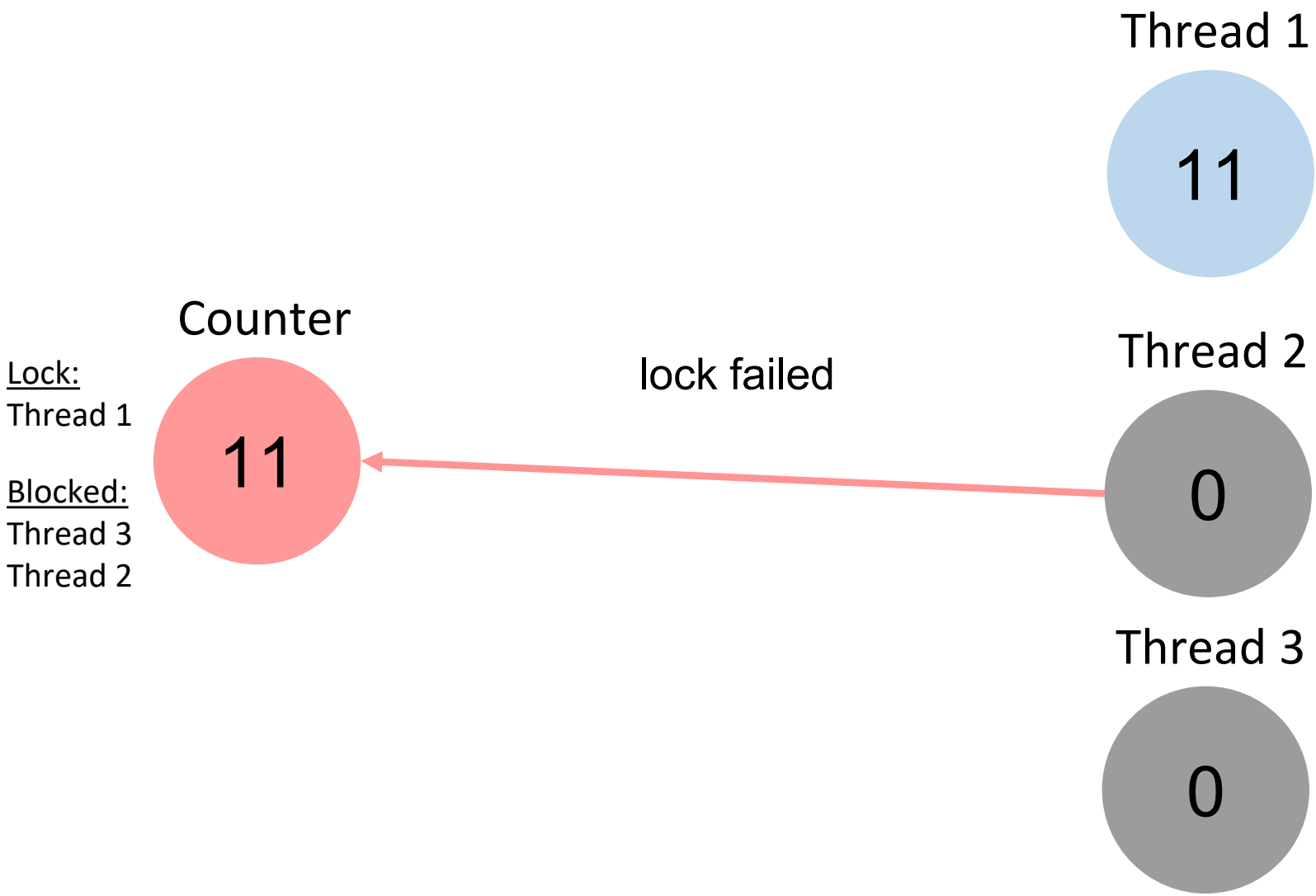
➔ A thread can obtain multiple locks (by nesting the synchronized blocks)

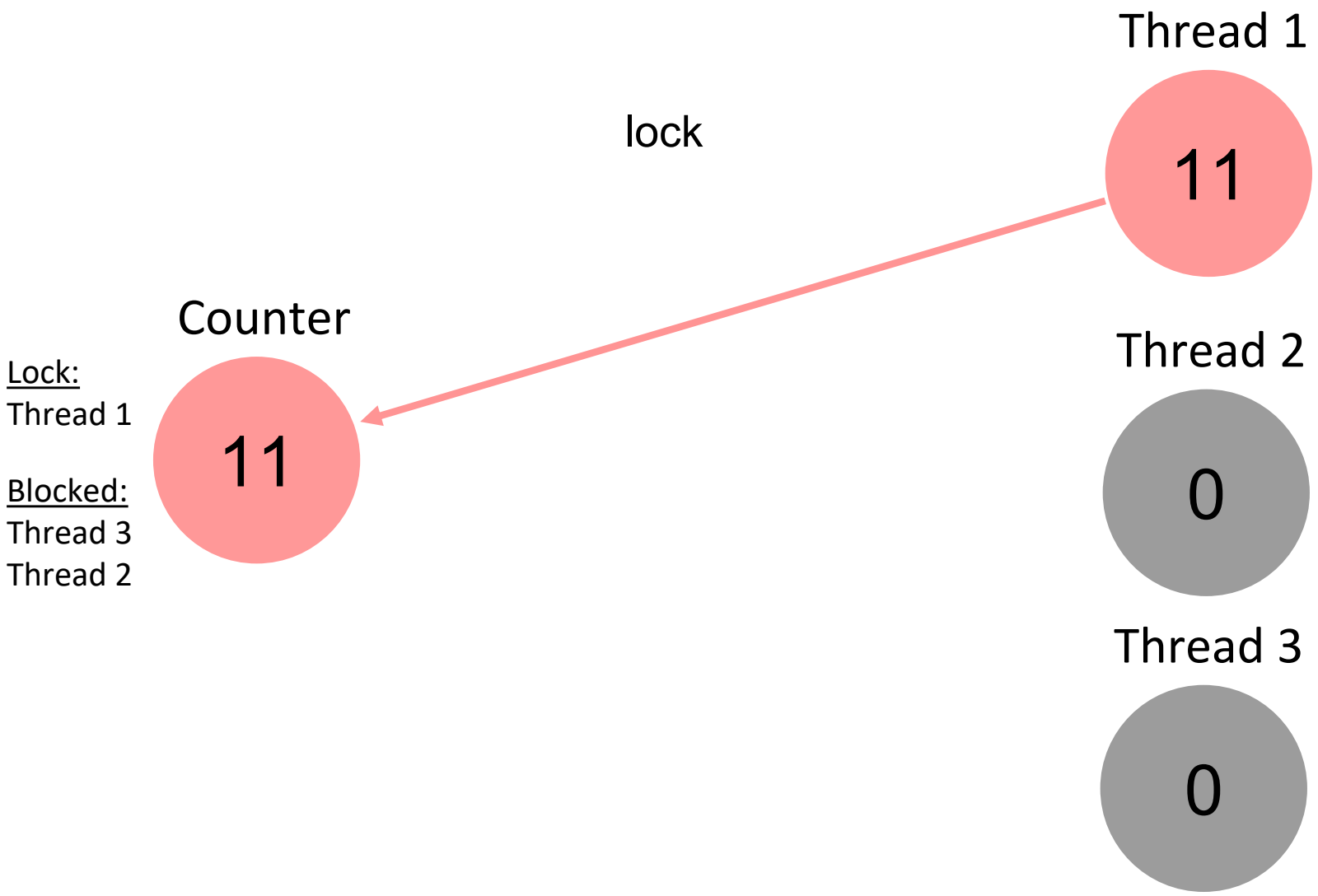

Thread 1
10
Counter
10
Thread 2
0
Thread 3
0

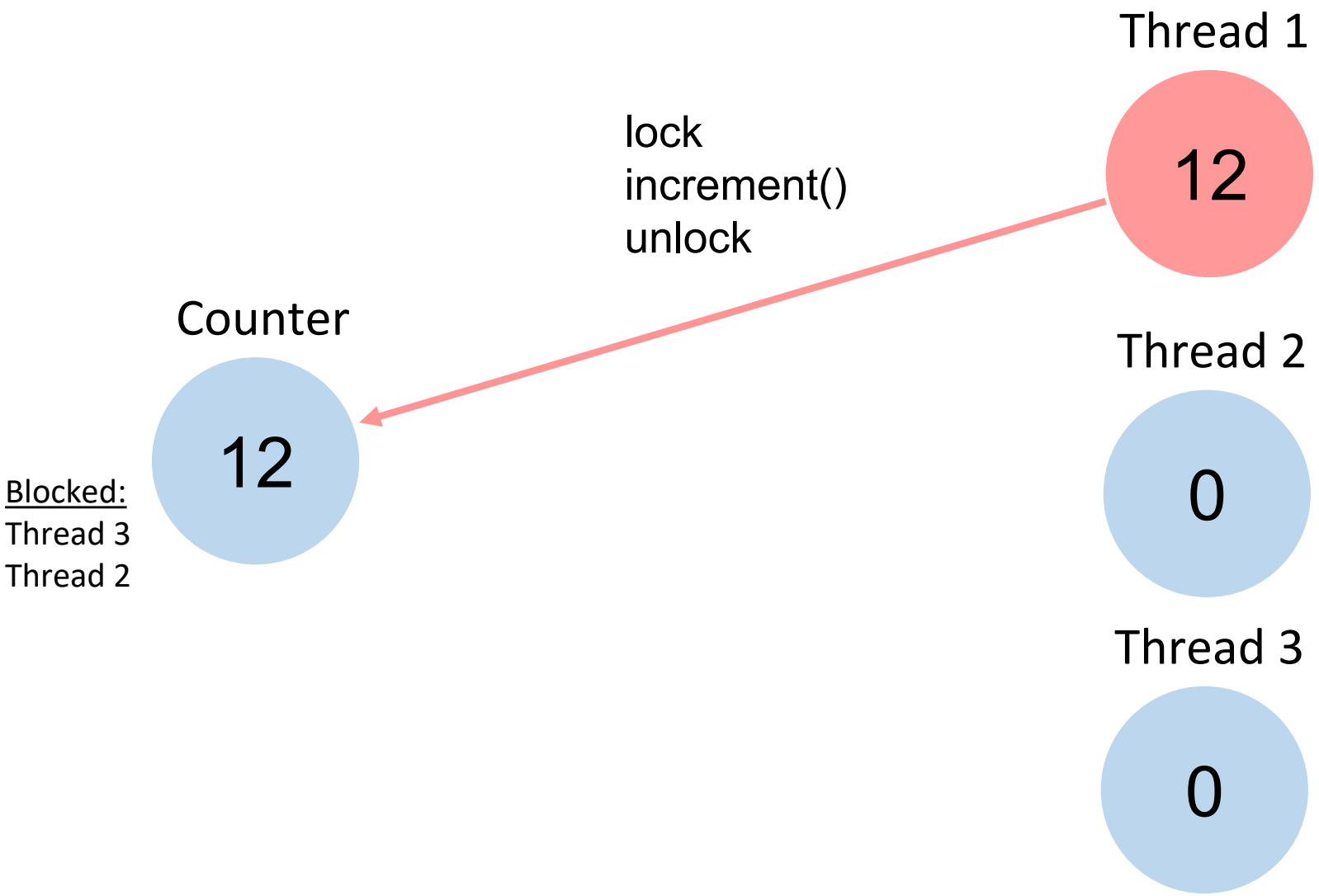

Thread 1
11
lock
increment()
unlock
Counter
11
Thread 2
0
Thread 3
0


Thread 1
lock
11
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

Task C

Whenever the Counter is incremented, keep track which thread performed the increment (you can print out the thread-id to the console). Observe how the threads are scheduled and discuss the factors that might influence this behavior.

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?

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

Wait and Notify Recap

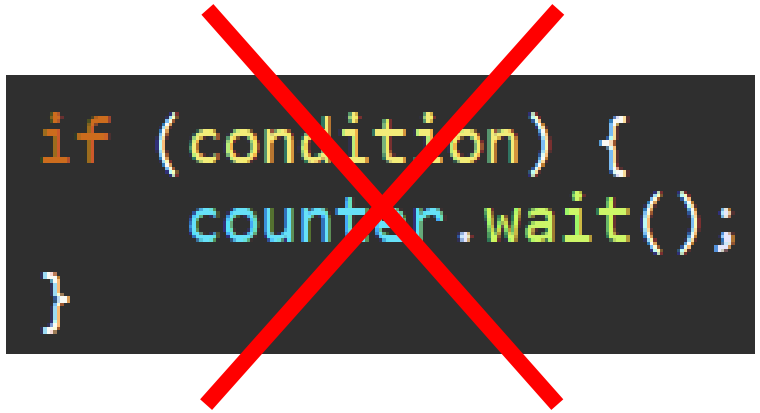
```
while (condition) {
    counter.wait();
}
```

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

What is the difference? Issues?

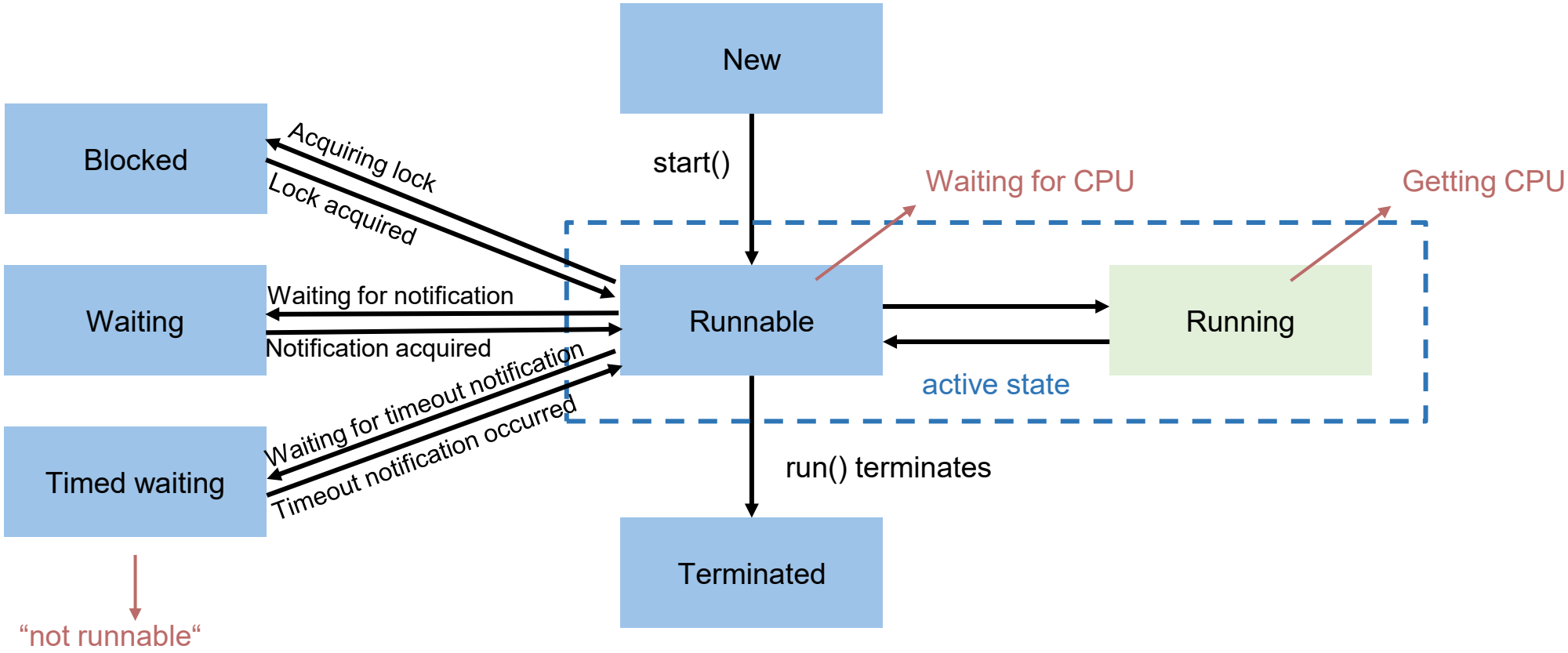
Wait and Notify Recap

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

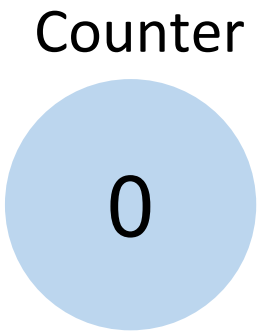


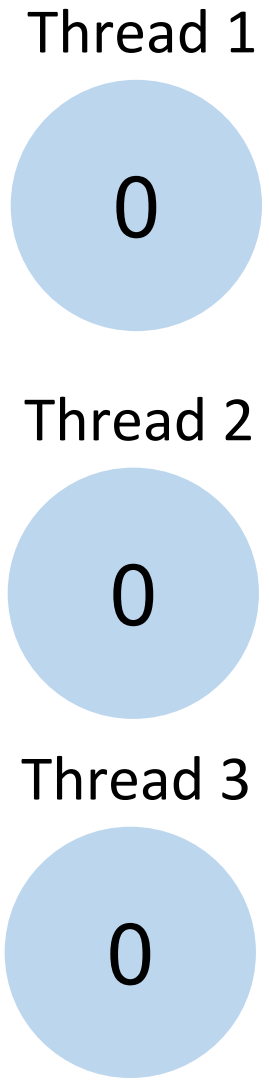
Spurious wake-ups and notifyAll()
→ `wait` has to be in a `while` loop

Life cycle of a Thread

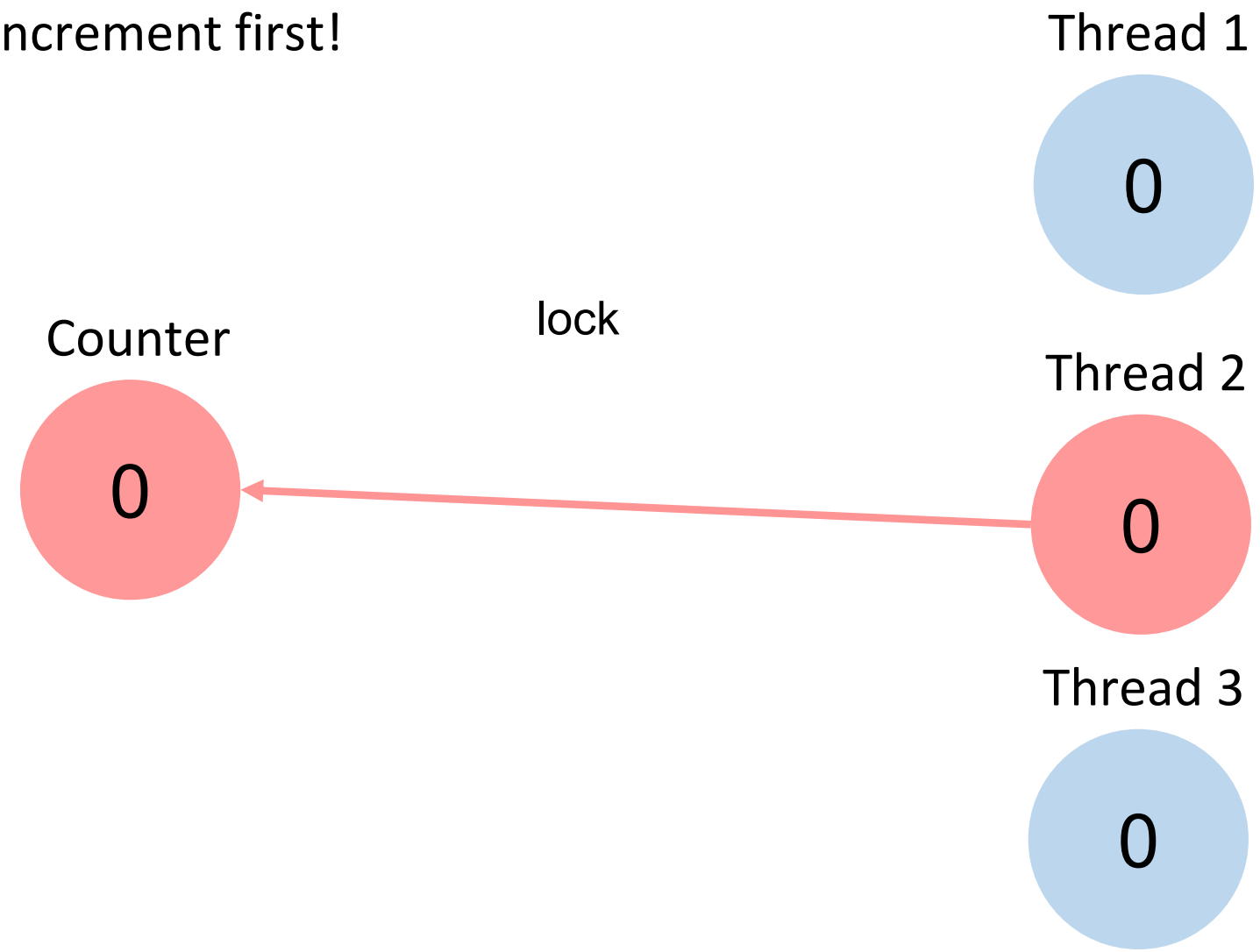


Thread 1 must increment first!

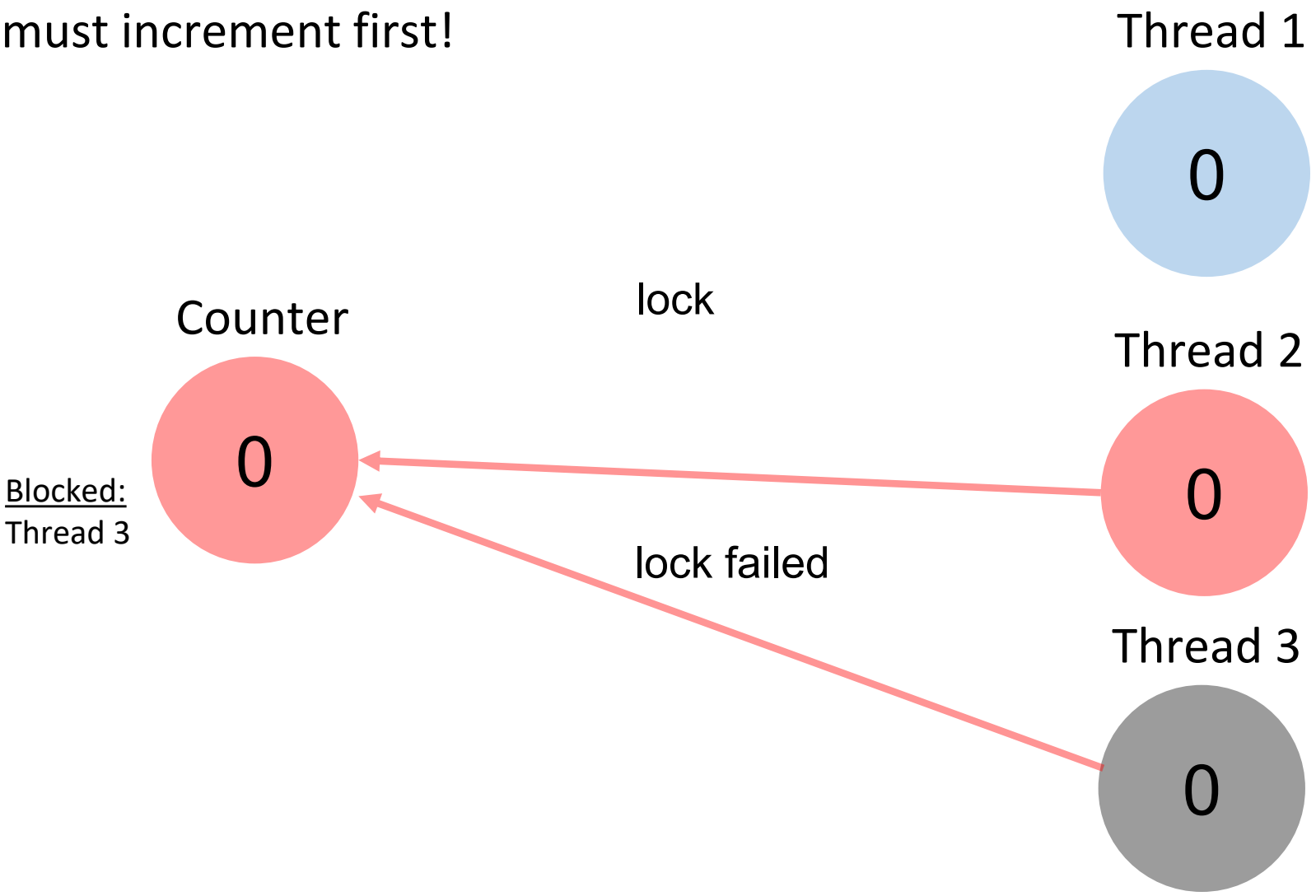




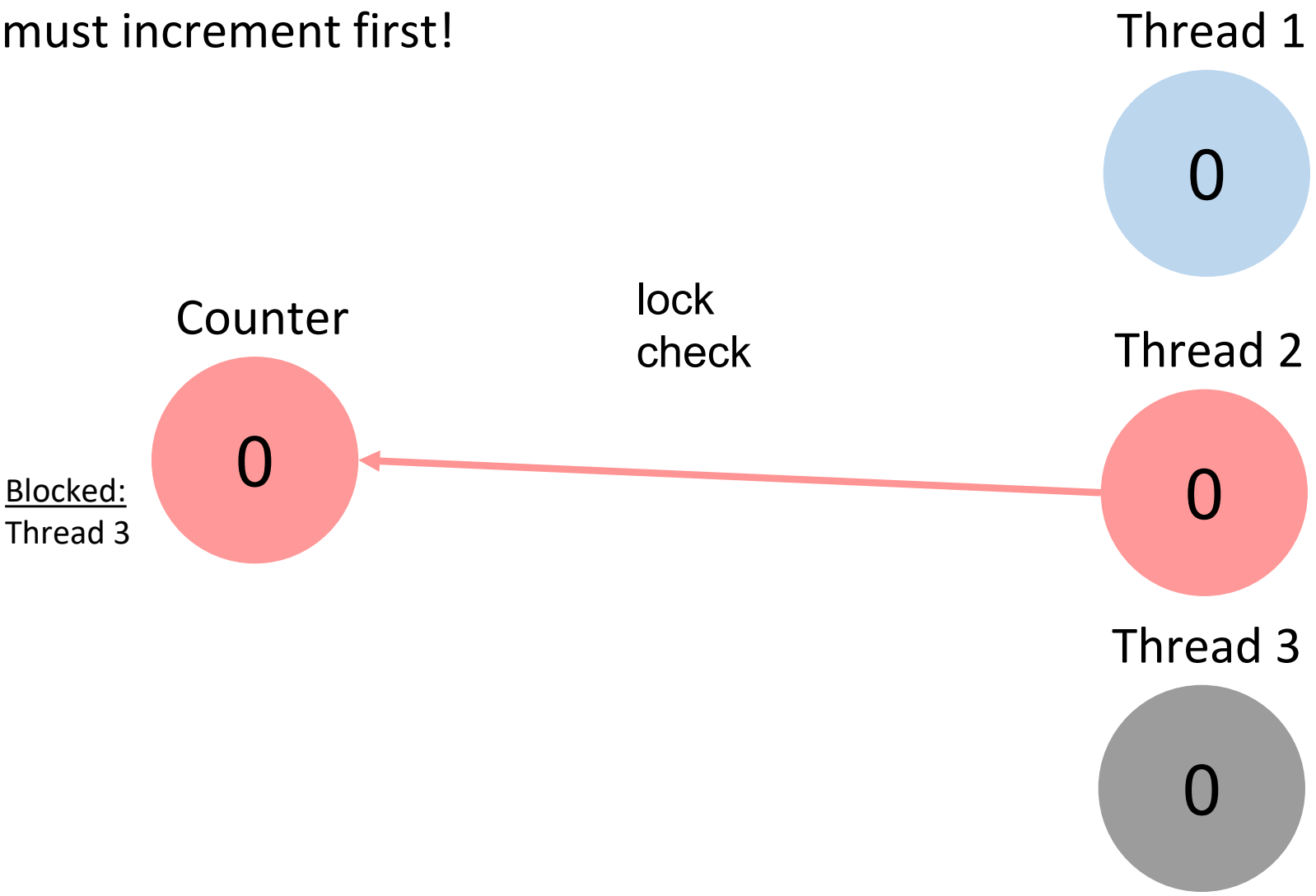
Thread 1 must increment first!



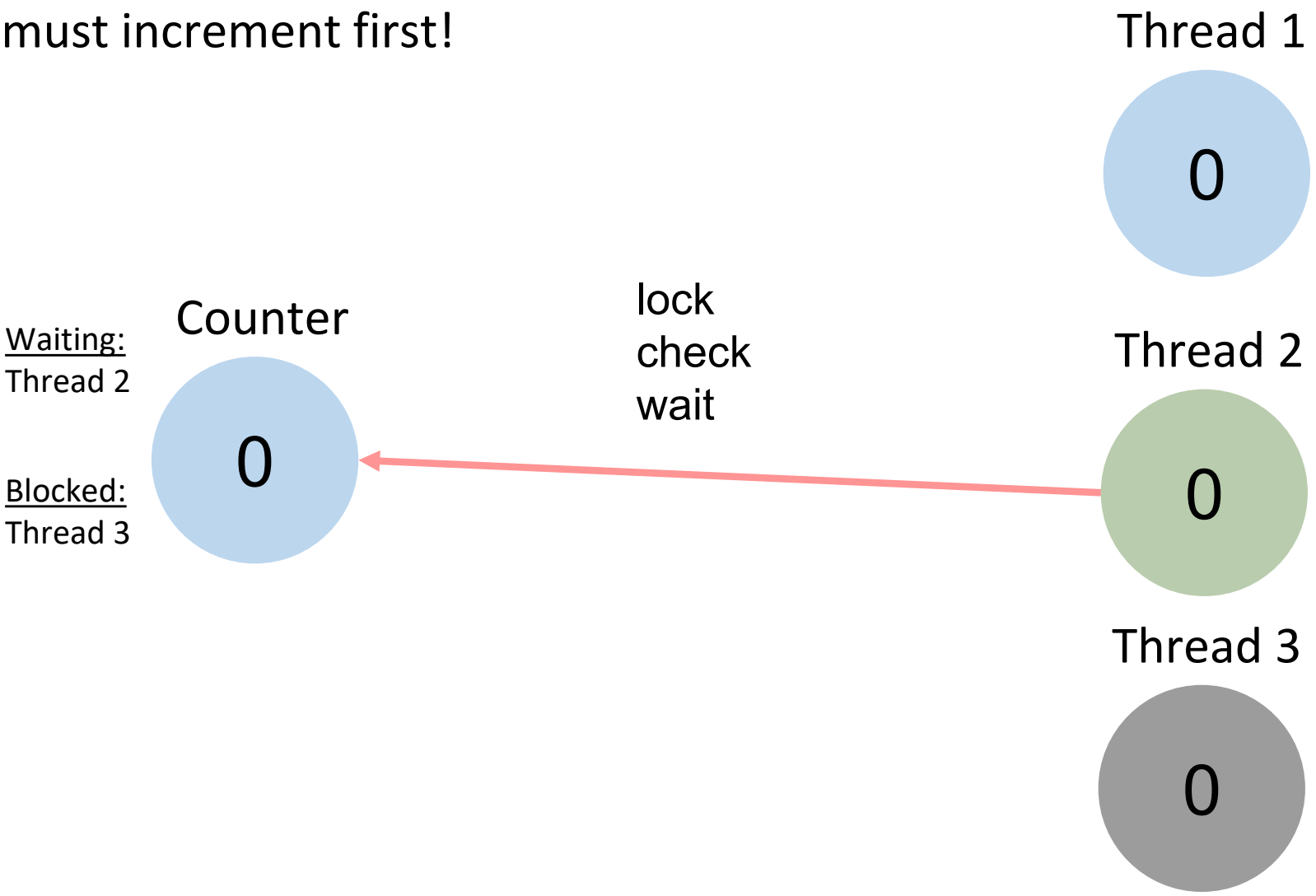
Thread 1 must increment first!



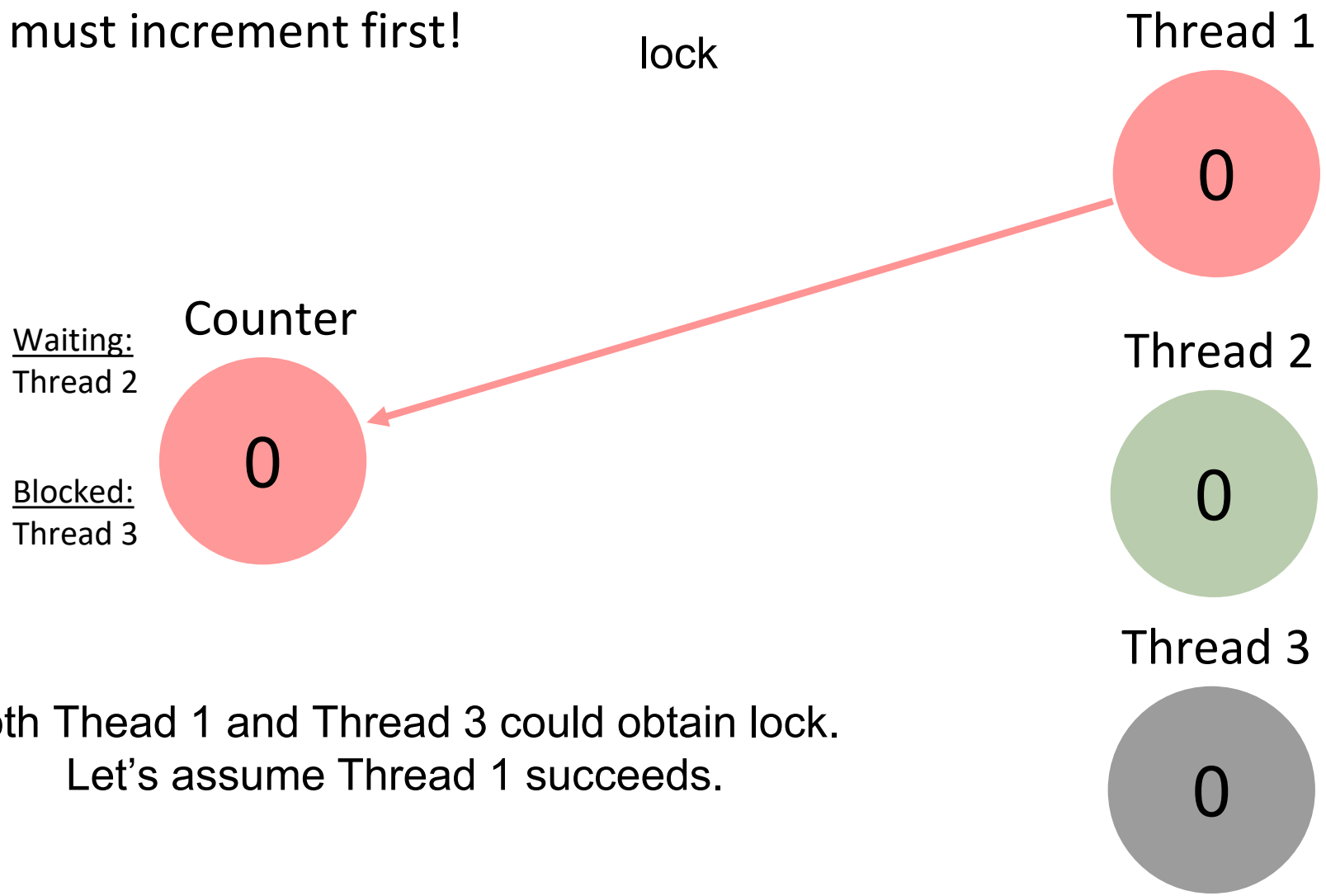
Thread 1 must increment first!



Thread 1 must increment first!

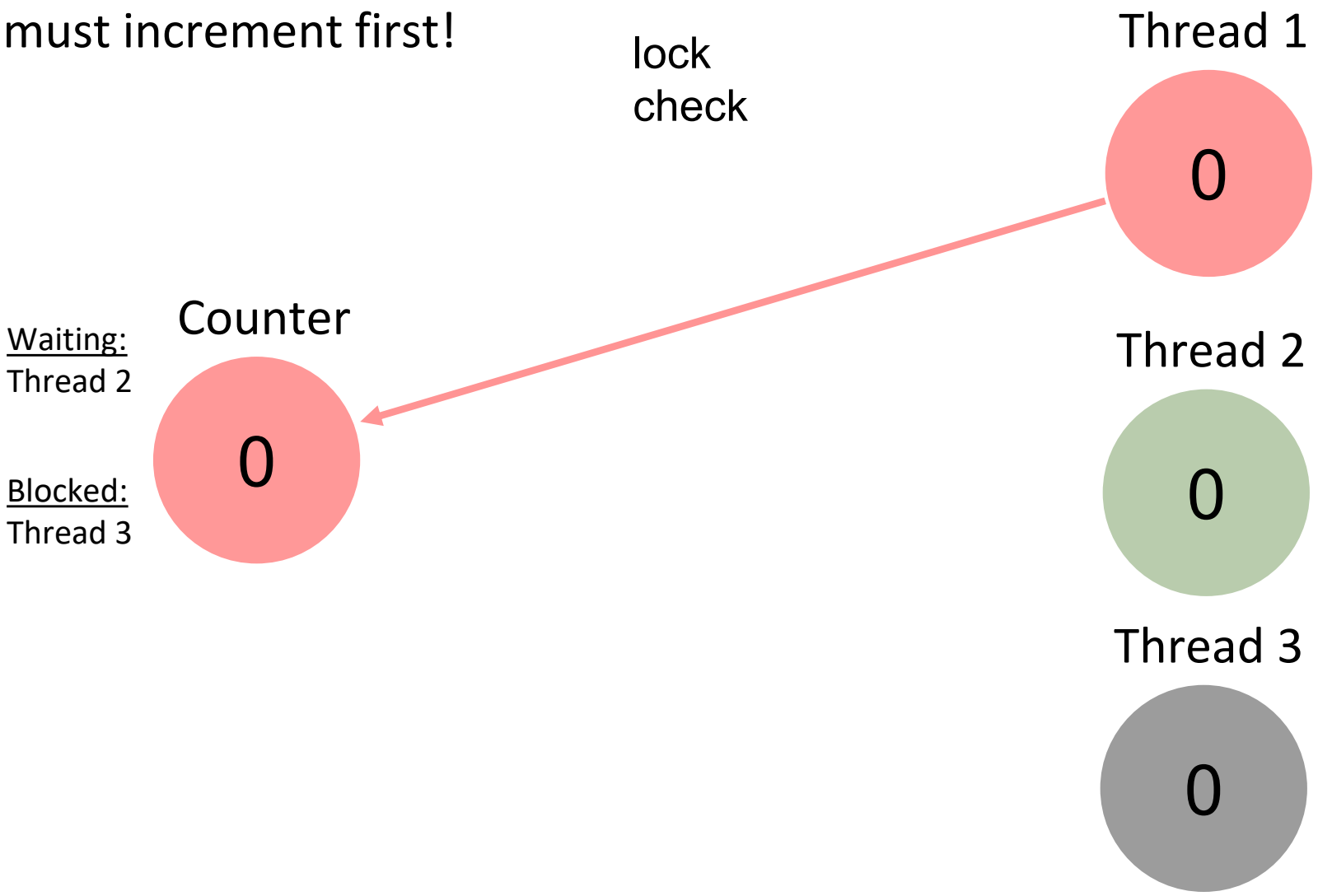


Thread 1 must increment first!

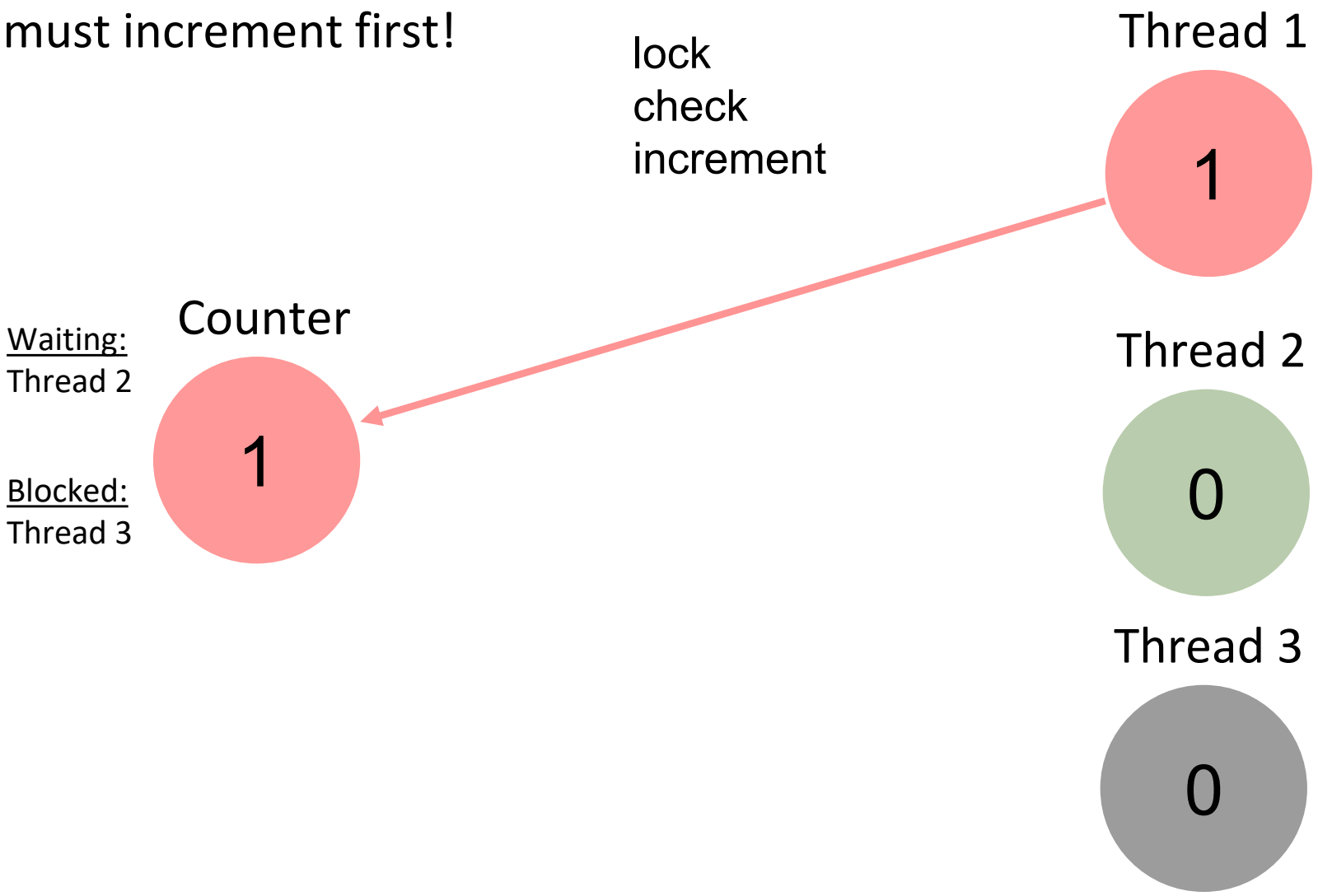


Both Thead 1 and Thread 3 could obtain lock.
Let's assume Thread 1 succeeds.

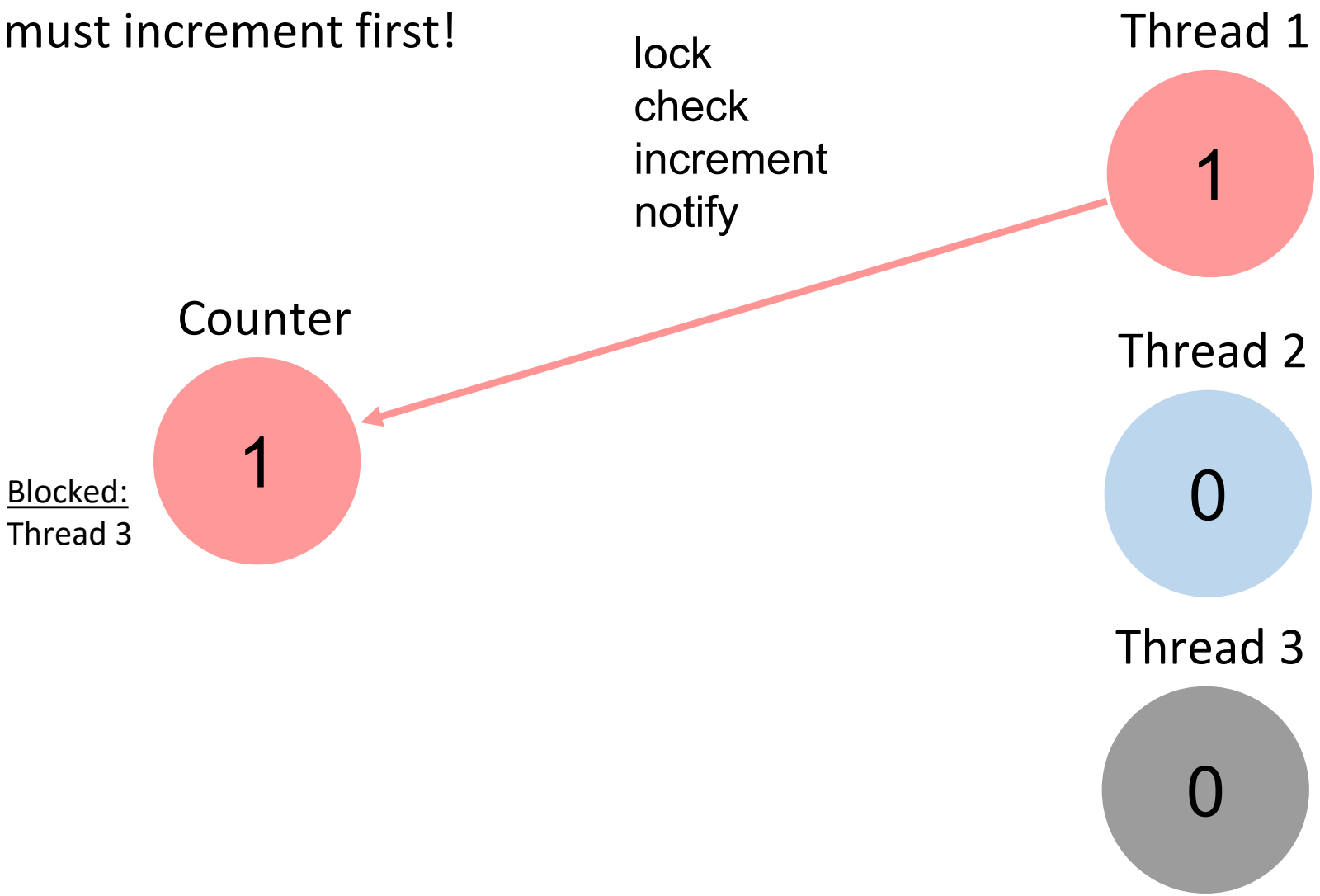
Thread 1 must increment first!



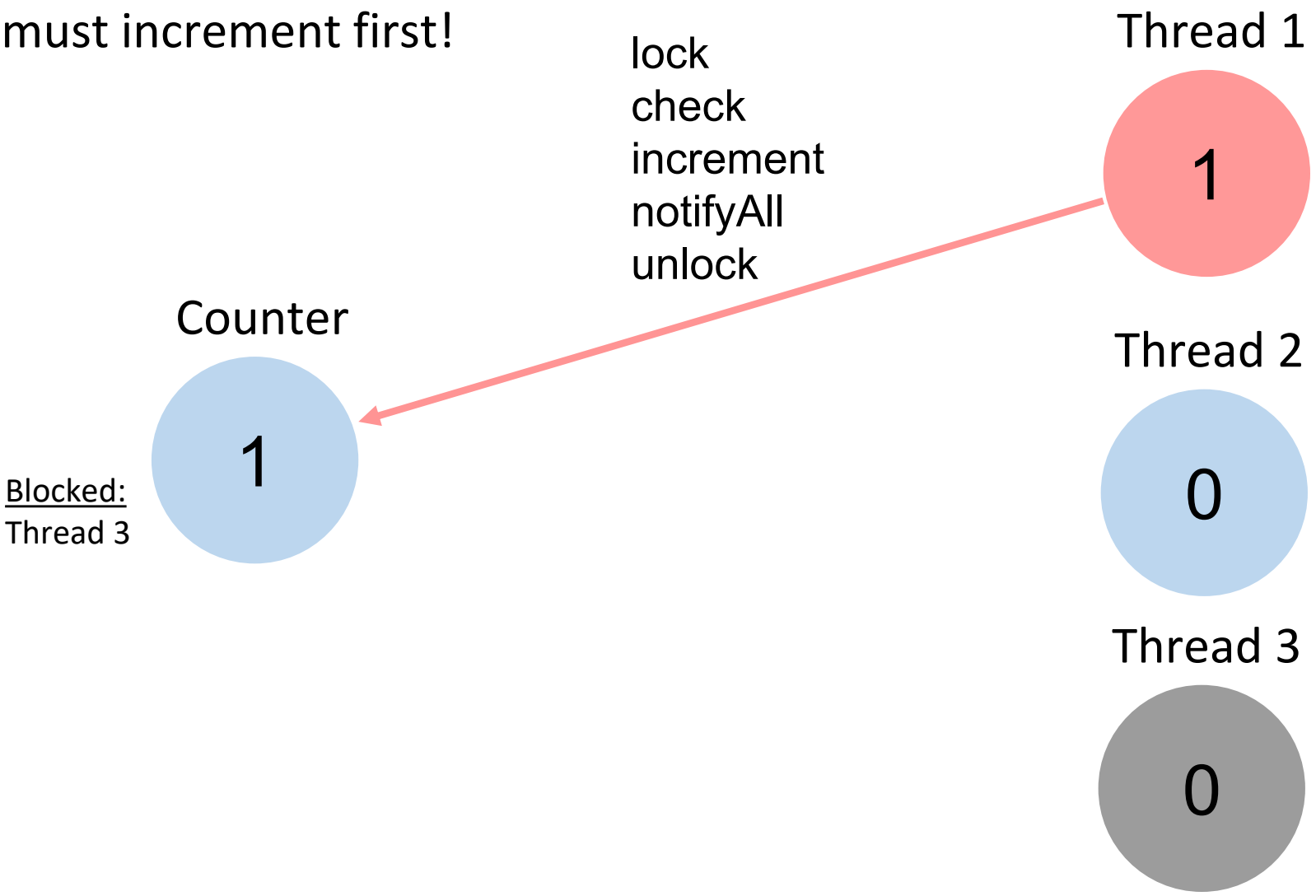
Thread 1 must increment first!



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.*

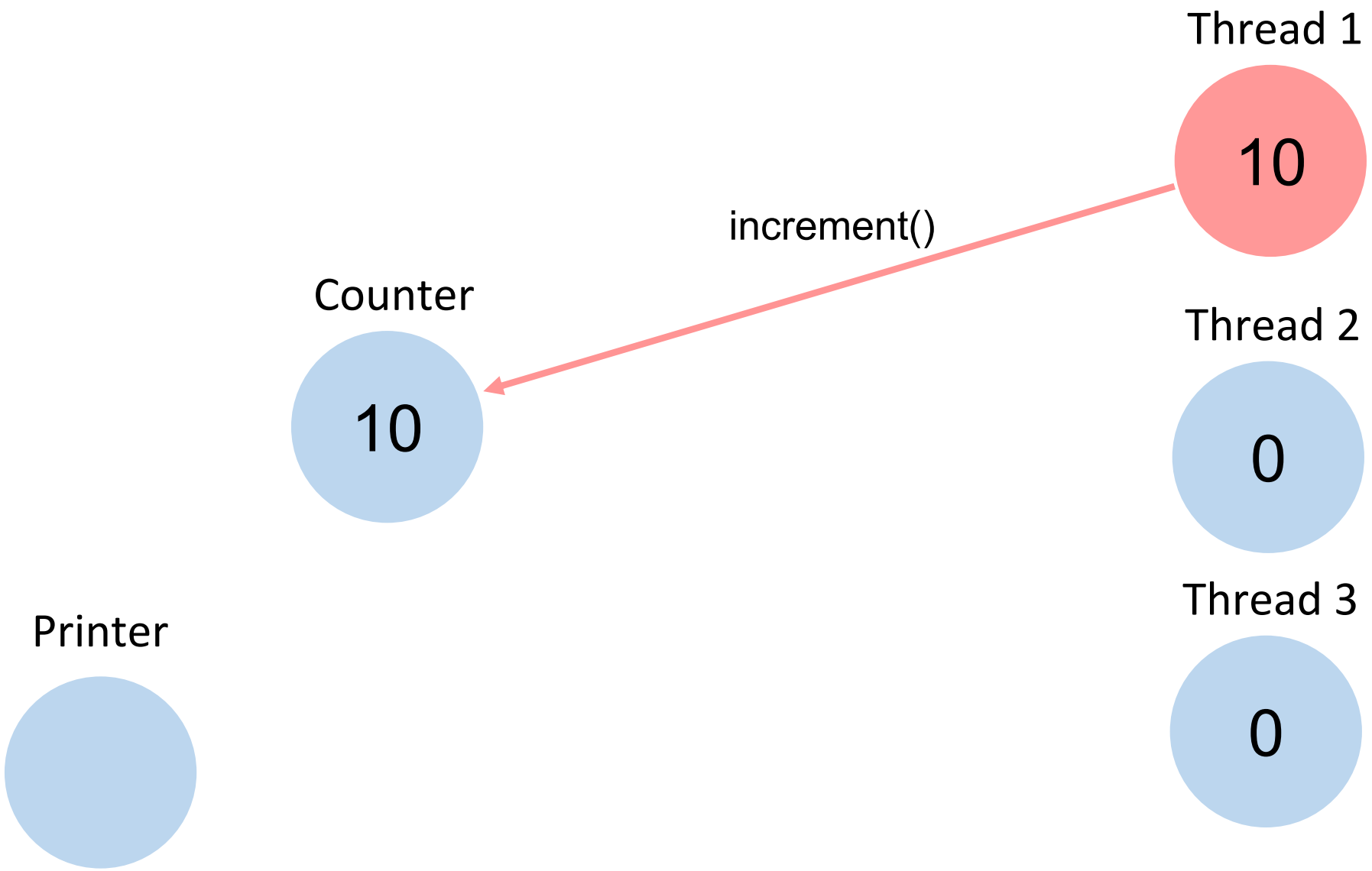
Task F – Atomic vs Synchronized counter

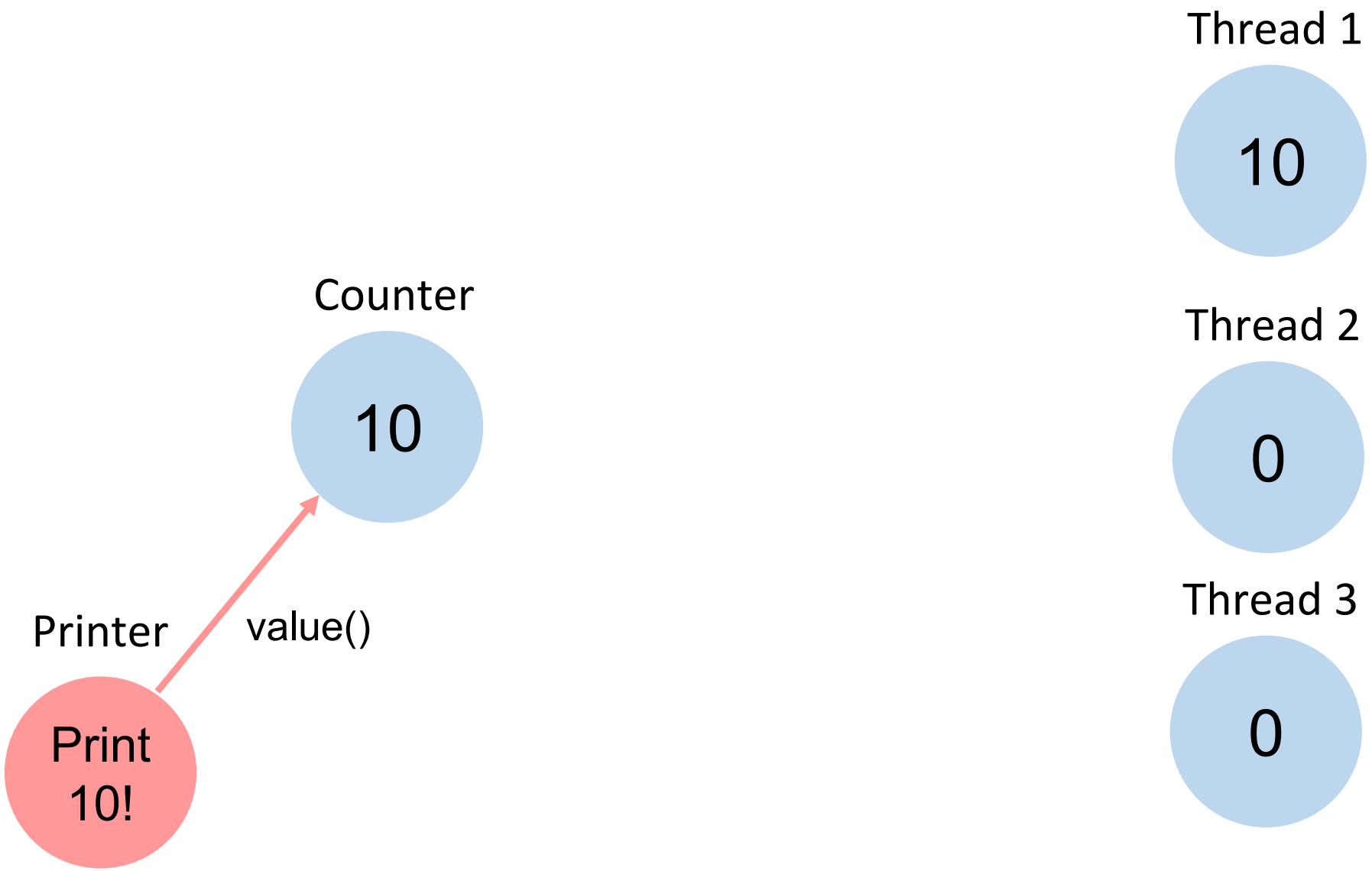
Experimentally compare the AtomicCounter and SynchronizedCounter implementations by measuring which one is faster. Observe the differences in the CPU load between the two versions. Can you explain what is the cause of different performance characteristics?

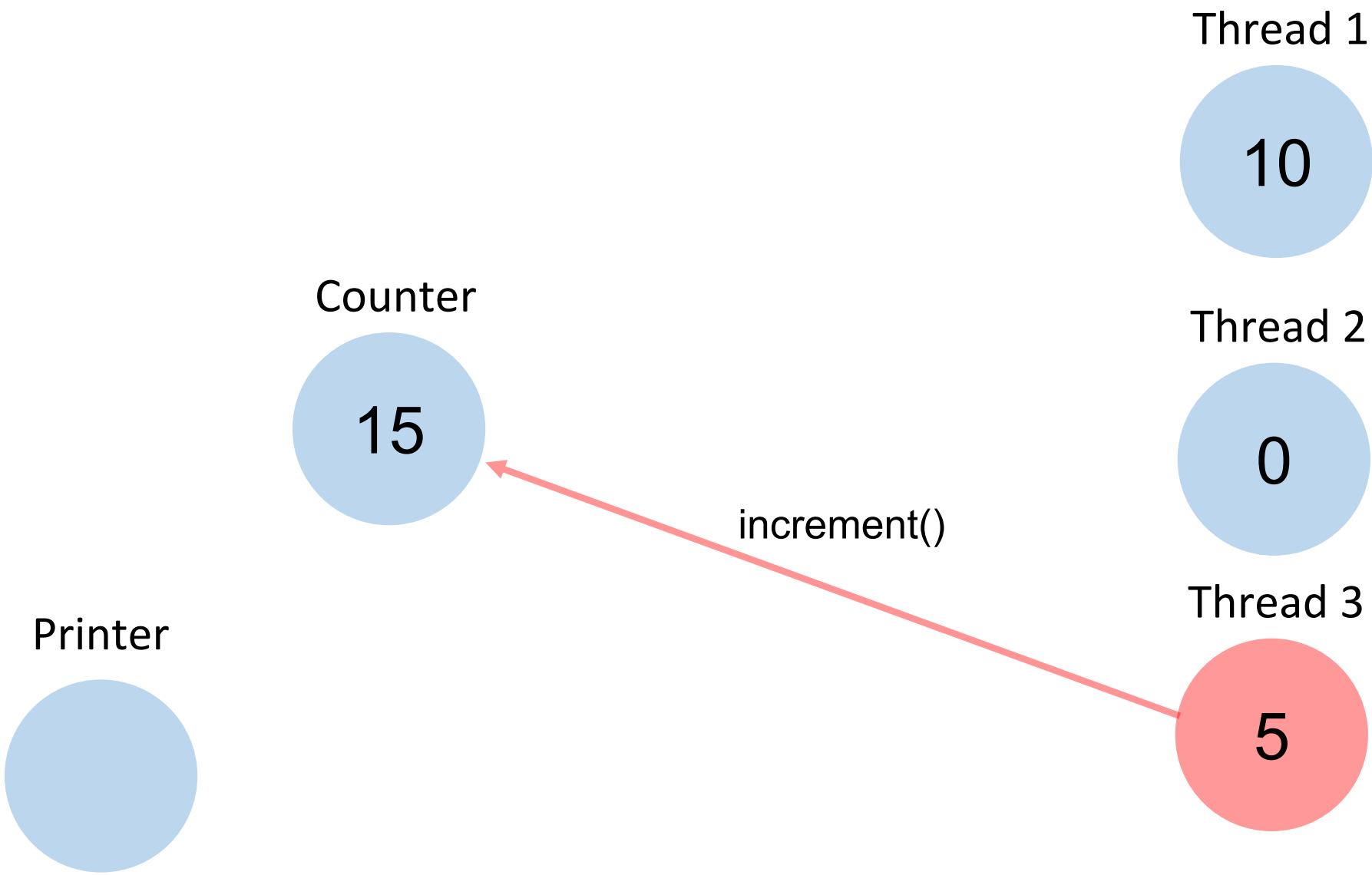
- Vary the load per thread
- Vary the number of threads

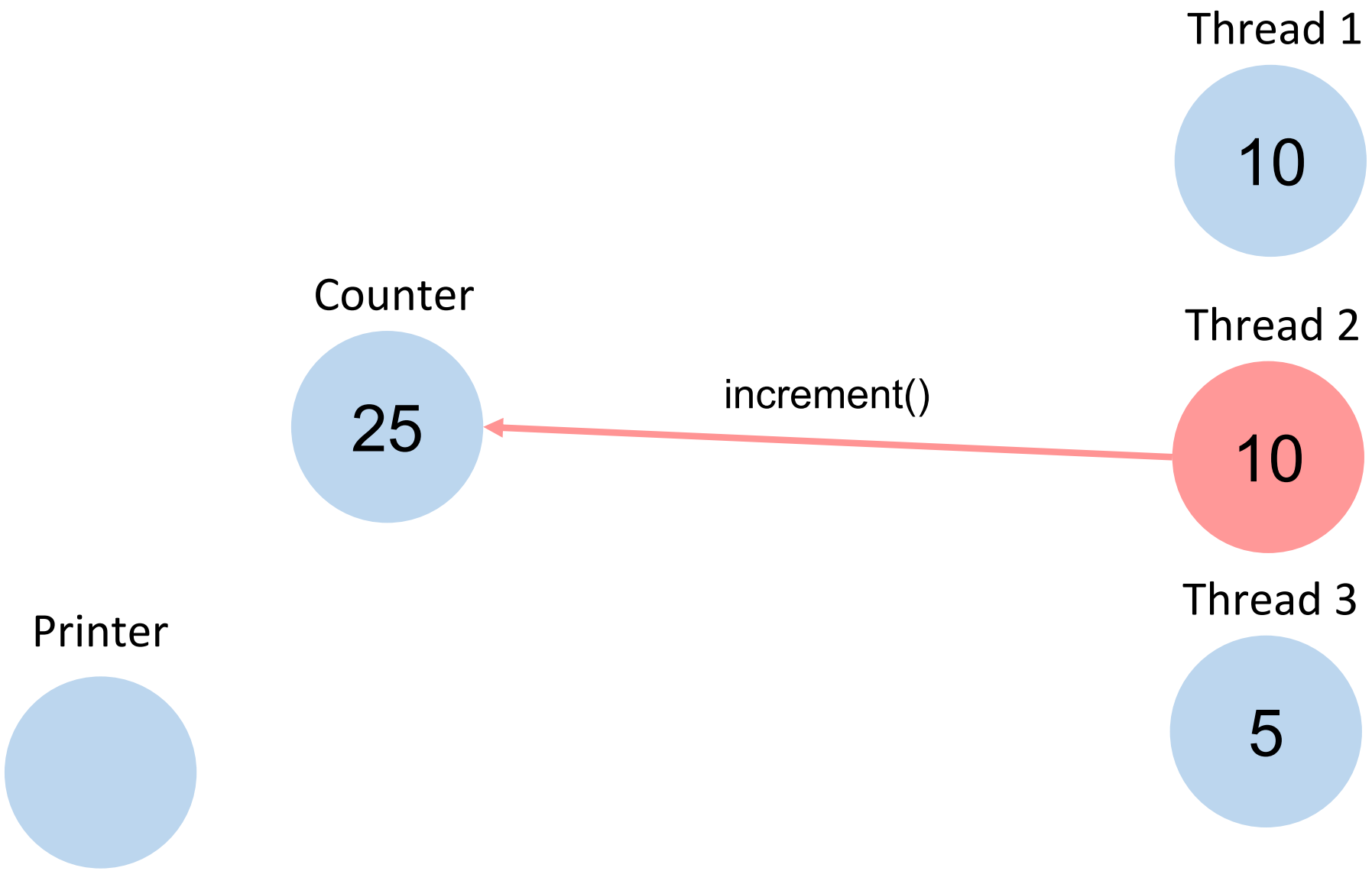
Task G

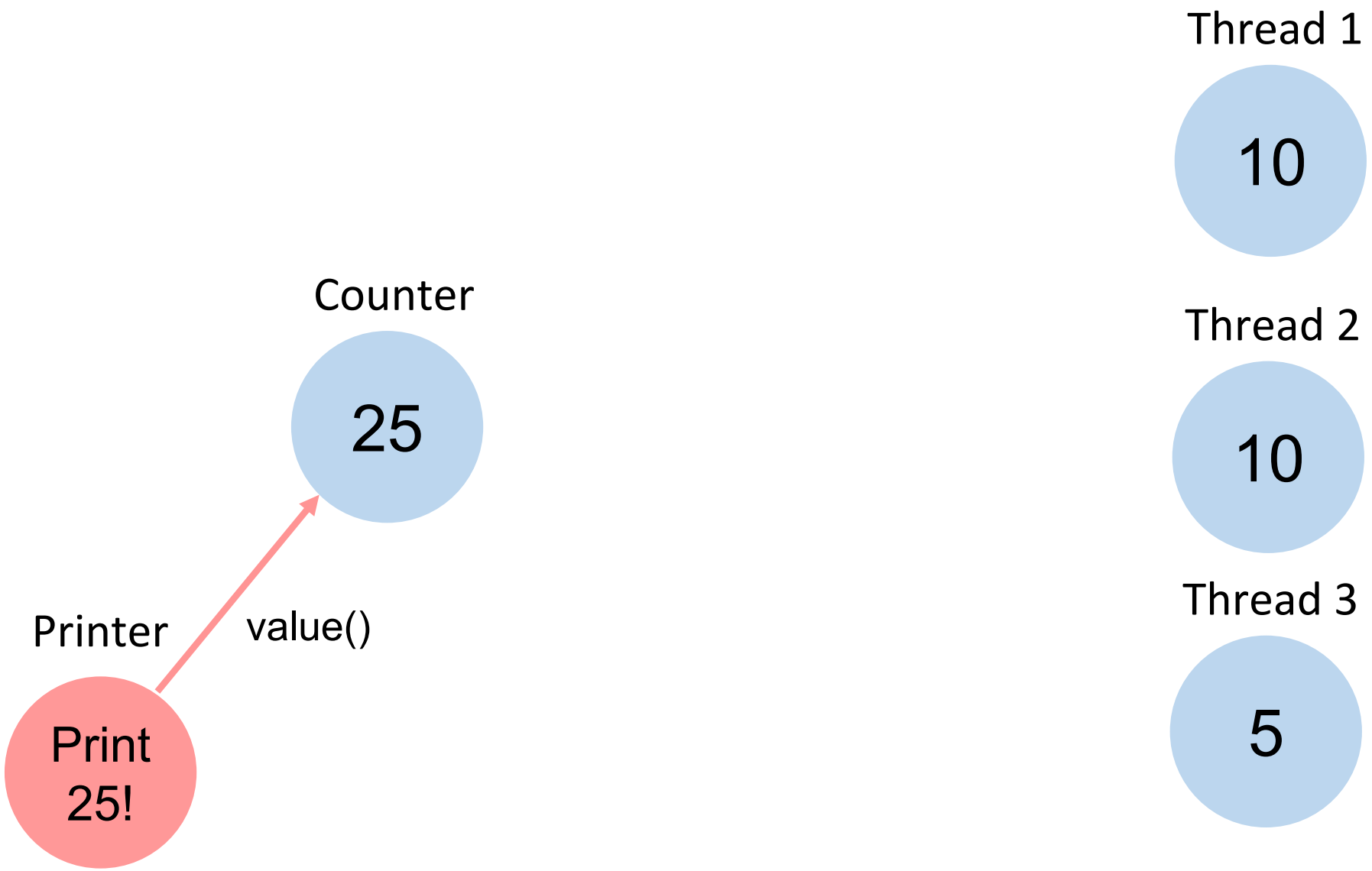
Implement a thread that measures execution progress. That is, create a thread that observes the values of the Counter during the execution and prints them to the console. Make sure that the thread is properly terminated once all the work is done [thread.interrupt()].

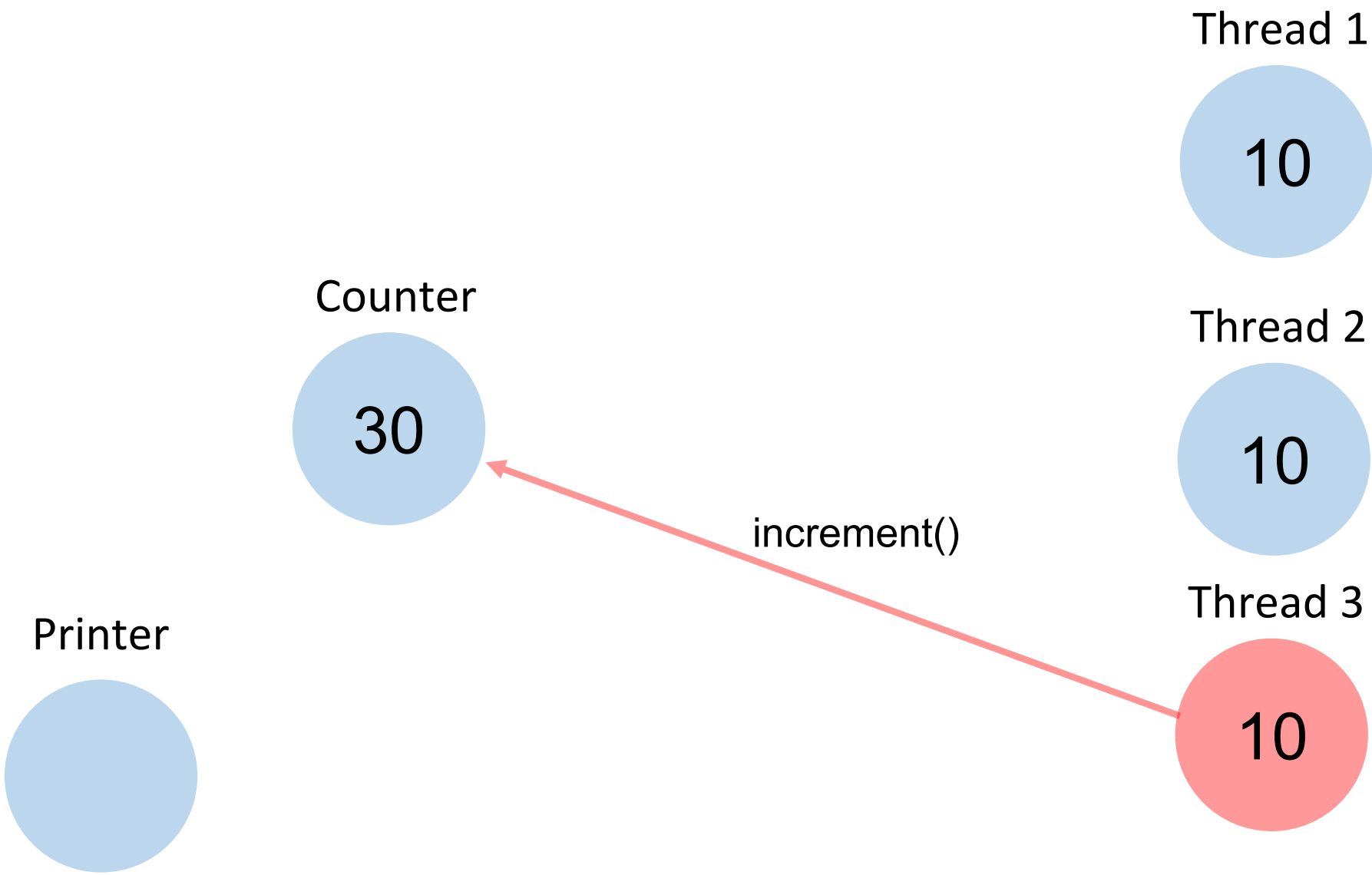

Thread 1
10
increment()
Counter
10
Thread 2
0
Thread 3
0
Printer

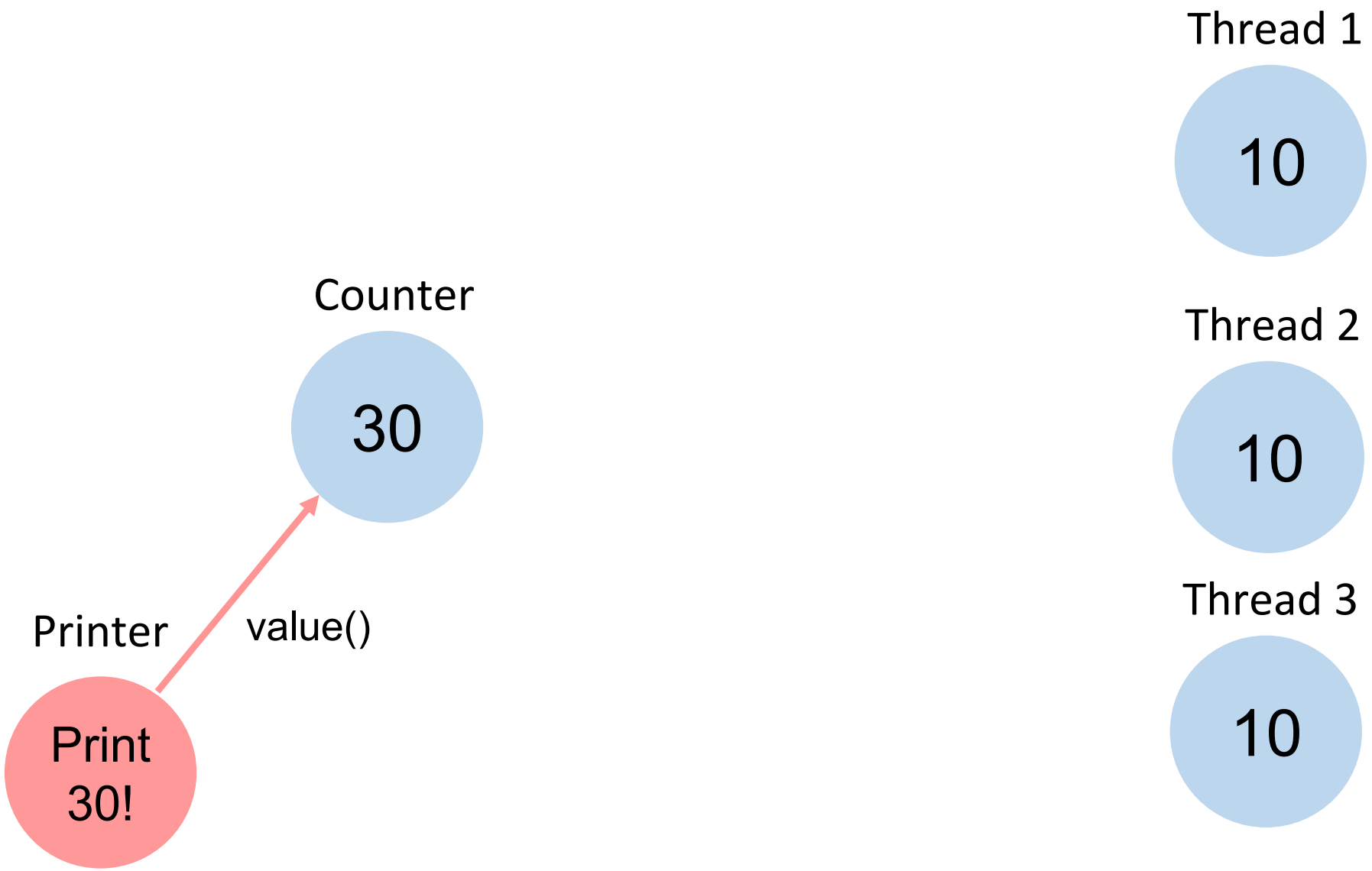

Thread 1
10
Thread 2
0
Thread 3
0
Counter
10
Printer
value()
Print
10!


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


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


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!

Theory Recap

Reentrant Property

Java locks are reentrant

=> A thread can hold a lock more than once. Also have to release multiple times!

```
public class Foo {
     public void synchronized f() { … }
     public void synchronized g() { … f(); … }
}

Foo foo = new Foo();

synchronized(foo) { … synchronized(foo) { … } … }
```

share the same lock if called on the same instance

synchronized for static methods

```
public synchronized void instanceMethod() { }
 => /* locks this */

public static synchronized void staticMethod() { }
 => /* locks ClassName.class */
```

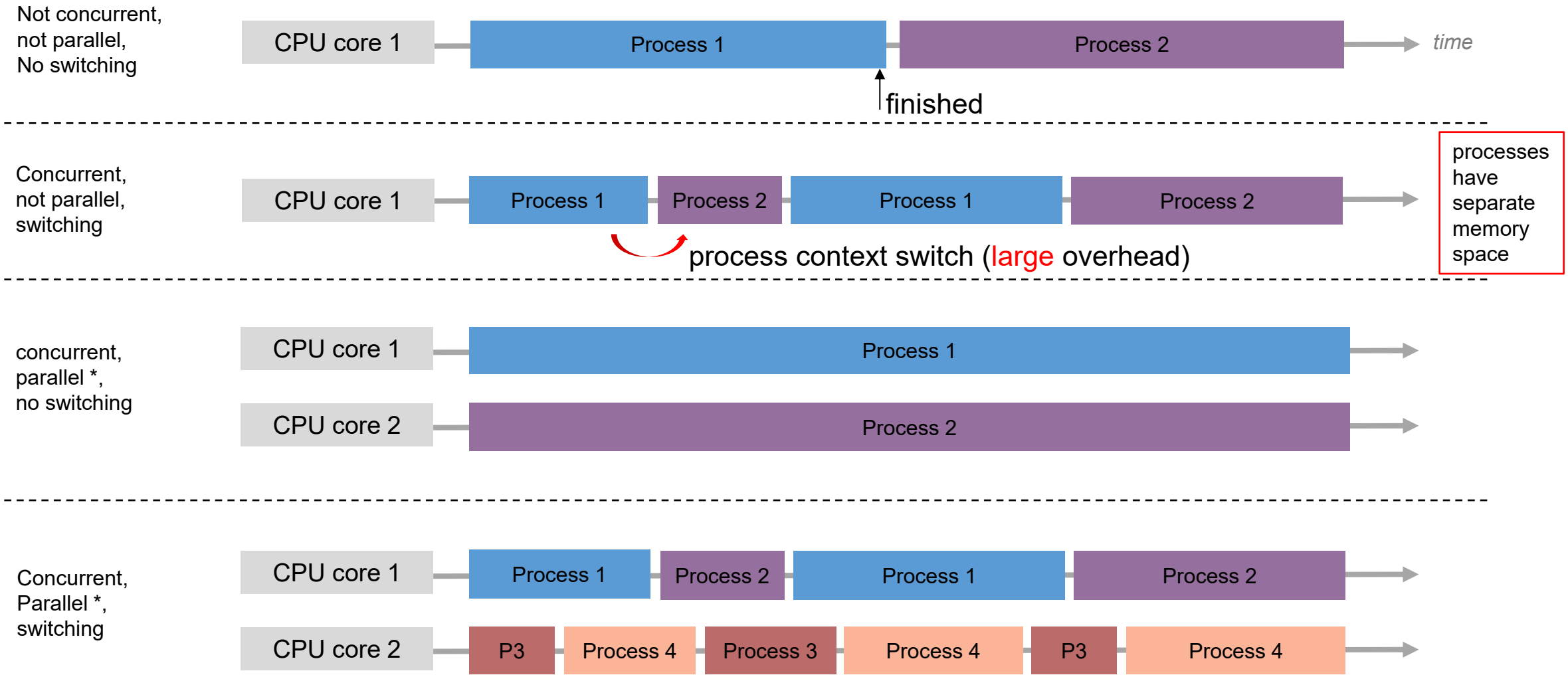
Concurrency vs Parallelism

Concurrency
Dealing with multiple things at the same time
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.

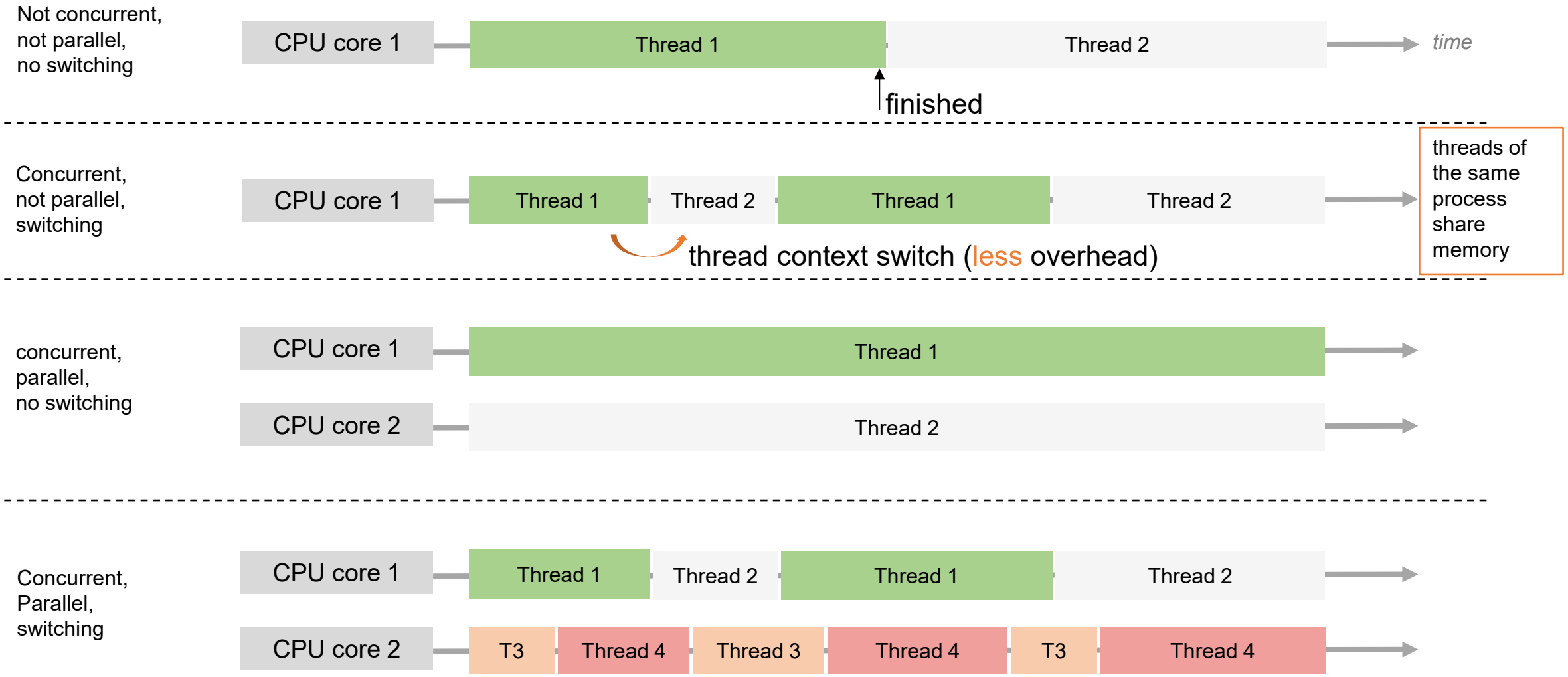
Source: https://cgl.ethz.ch/teaching/parallelprog26/pages/terminology.html

Sequential, Concurrent, Parallel



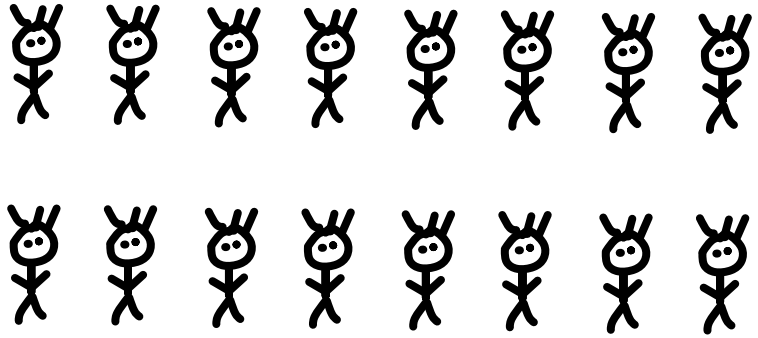
*multi-core and multi-processor systems

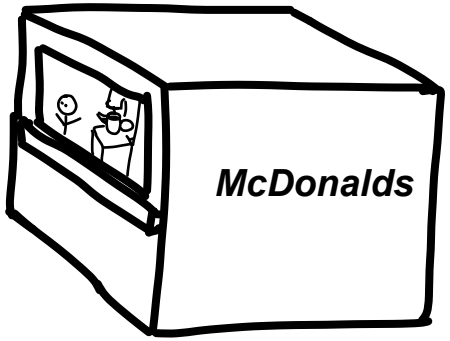
Sequential, Concurrent, Parallel



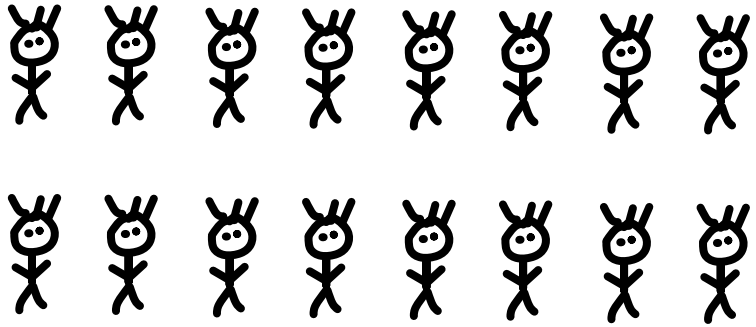
*multi-core and multi-processor systems

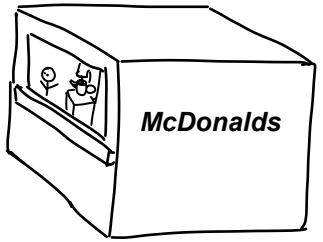
Concurrency vs Parallelism

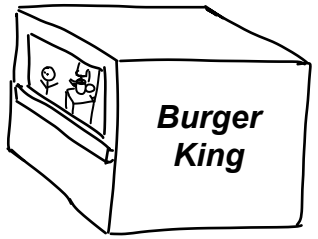




Concurrency
No parallelism







Concurrency
Parallelism

Important Terminology

Data race (low-level race condition): Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g., simultaneous read/write or write/write of the same memory location.

=> hardware / Java memory model problem

Bad interleaving (high-level race condition): Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm.

=> logical correctness problem

Layer 1 – Hardware

Two cores touching same memory line.

Layer 2 – Language Memory Model

Rules defining when behavior is well-defined.

Layer 3 – Algorithm

Logical correctness (bad interleaving).

Important Terminology

Critical section: Part of a program where shared resources are accessed by multiple threads, and only one thread should execute it at a time to prevent data races and inconsistencies.

Mutual exclusion: Ensures that only one process or thread enters the critical section at a time, preventing data races and bad interleavings that could lead to inconsistent or incorrect program behavior.

Atomicity: In the context of synchronized, atomicity means that a critical section (protected by synchronized) is executed as an indivisible unit, preventing other threads from interrupting or seeing partial updates.

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.

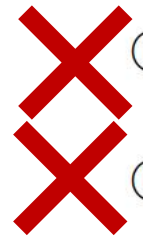
A fully serial block of code can be run on multiple processors to speedup execution.

Rep. Exam, FS 2023

Past Exam Task

(c) Wozu dient die `join()` Methode in Java `Threads`? (2)

What is the purpose of the `join()` method in Java `Threads`?.



○ 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

https://quizizz.com/admin/quiz/65e58677360c000d5287d49b?source=quiz_share

Replace link with link to quiz