Parallel Programming
Exercise Session 3

Spring 2020

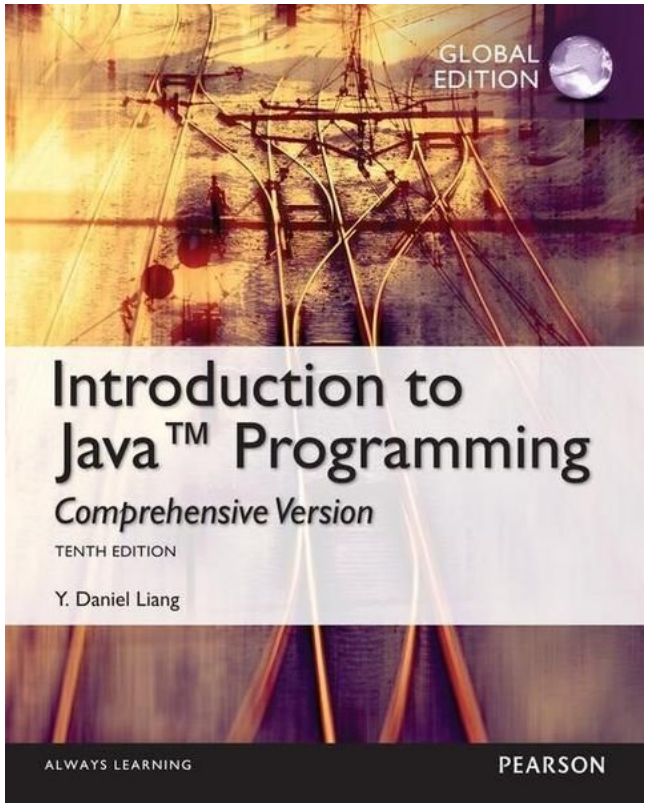
Java Review

Java packages/access modifiers

(Chapter 9.8) Visibility Modifiers
(Chapter 11.14) Protected Data and Methods

Try/Catch

(Chapter 12.1 - 12.7) Exception Handling



Feedback: Exercise 2

Task D

- We covered static partitioning but other types are possible, e.g., dynamic, guided, etc. See list of options provided by OpenMP
- We implemented parallel loop as part of our exercise — in practice use existing libraries that are well tested, concise and faster than your implementation, e.g. OpenMP for C++ or parallel streams for Java 8

Exercise 3

Counter

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

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

Counter

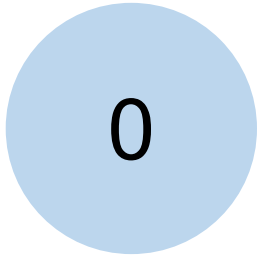
Let’s count the number of times a given event occurs

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

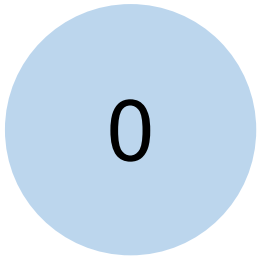
```
// background threads
for (int i = 0; i < numIterations; i++) {
  // perform some work

  counter.increment();
}

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

10 iterations each

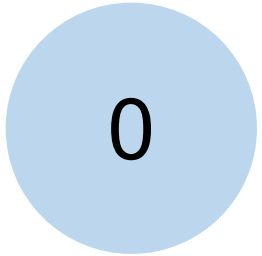
Thread 1



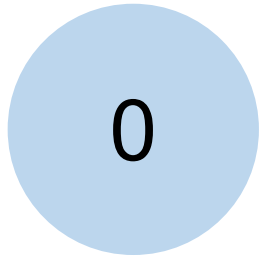
Counter



Thread 2



Thread 3



value of the
shared Counter

number of times
`increment()` is called

Thread 1

0

Counter

0

Thread 2

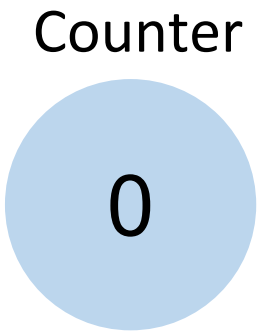
0

Thread 3

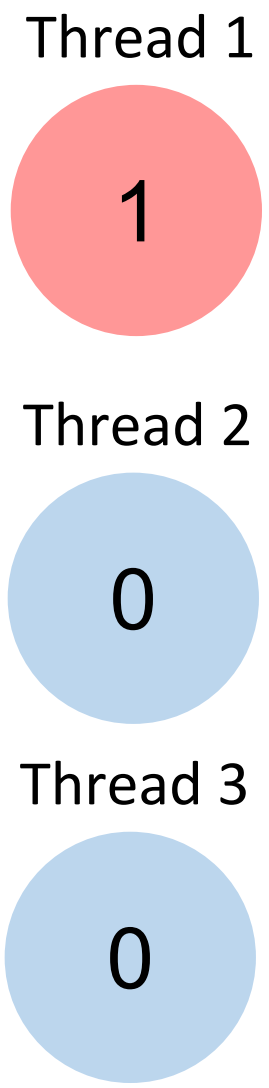
0

value of the
shared Counter

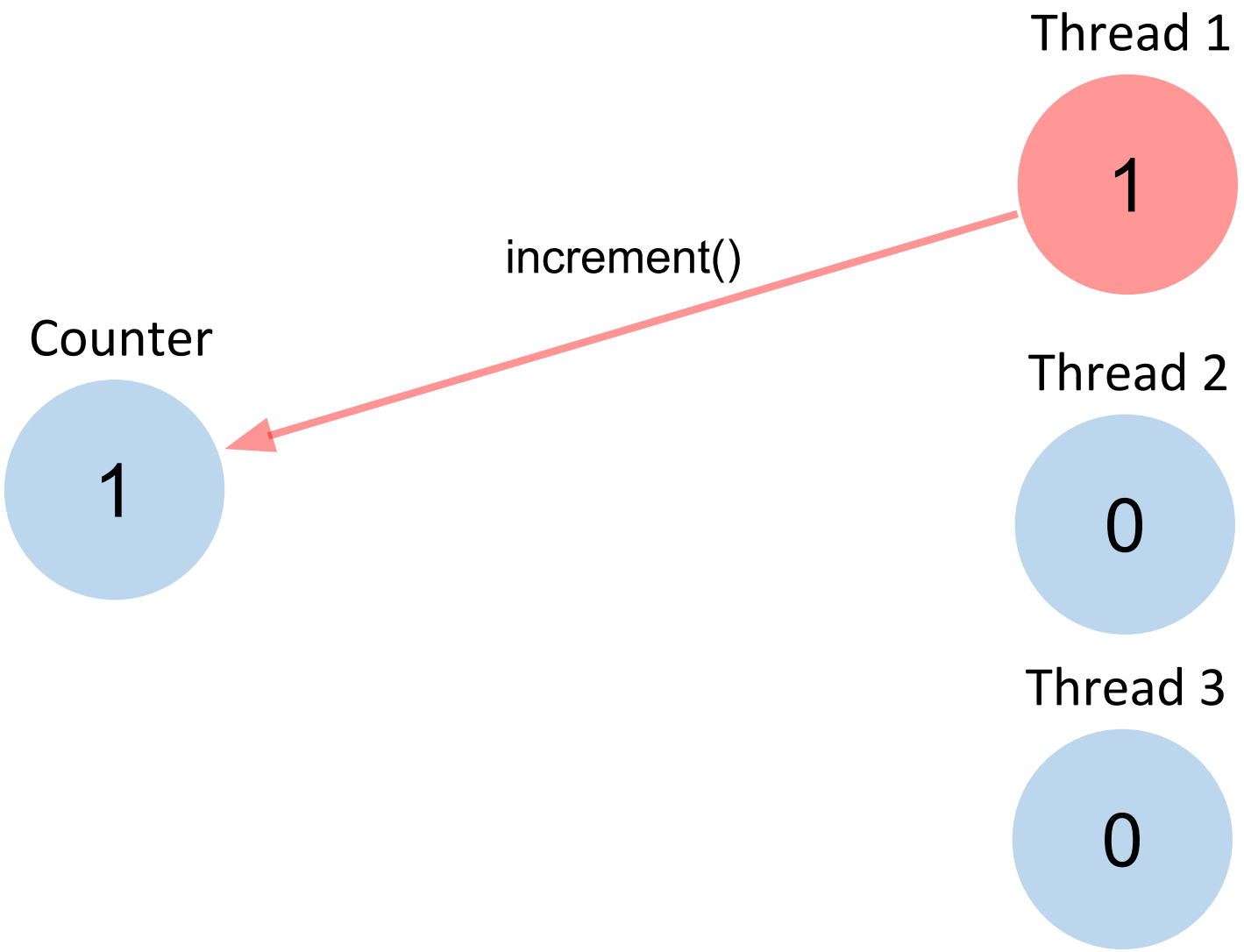
number of times
`increment()` is called



value of the
shared Counter

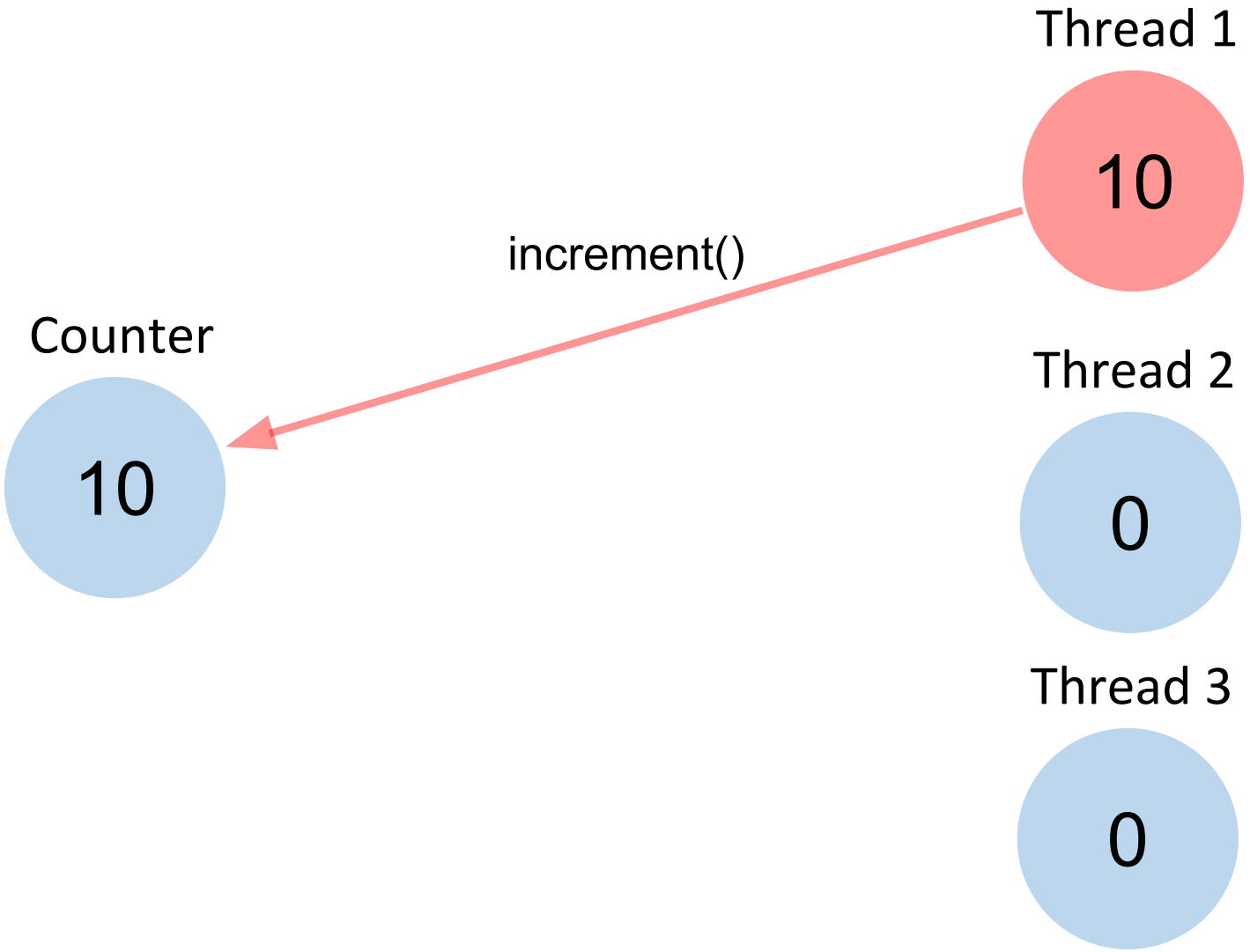


number of times
`increment()` is called



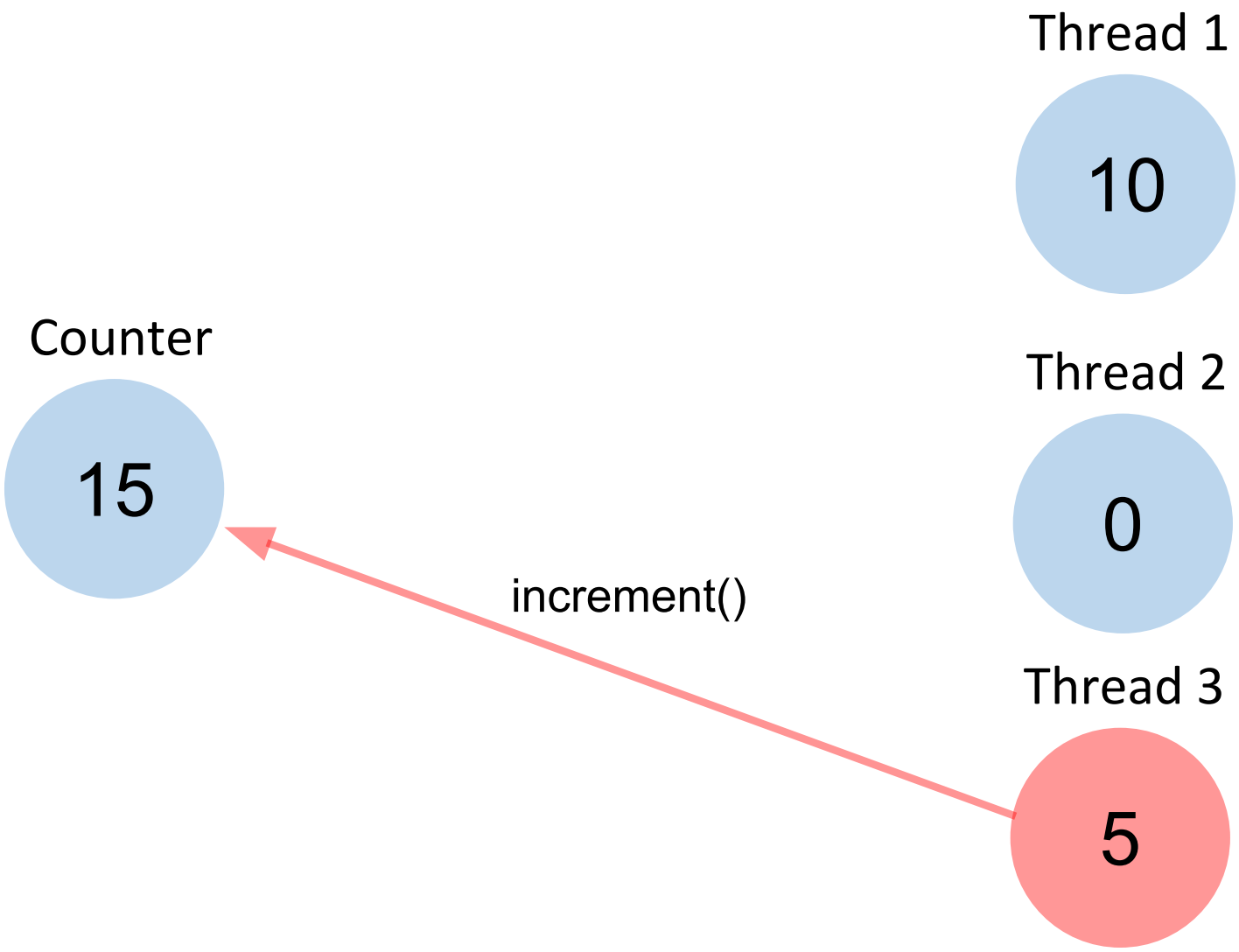
value of the shared Counter

number of times `increment()` is called



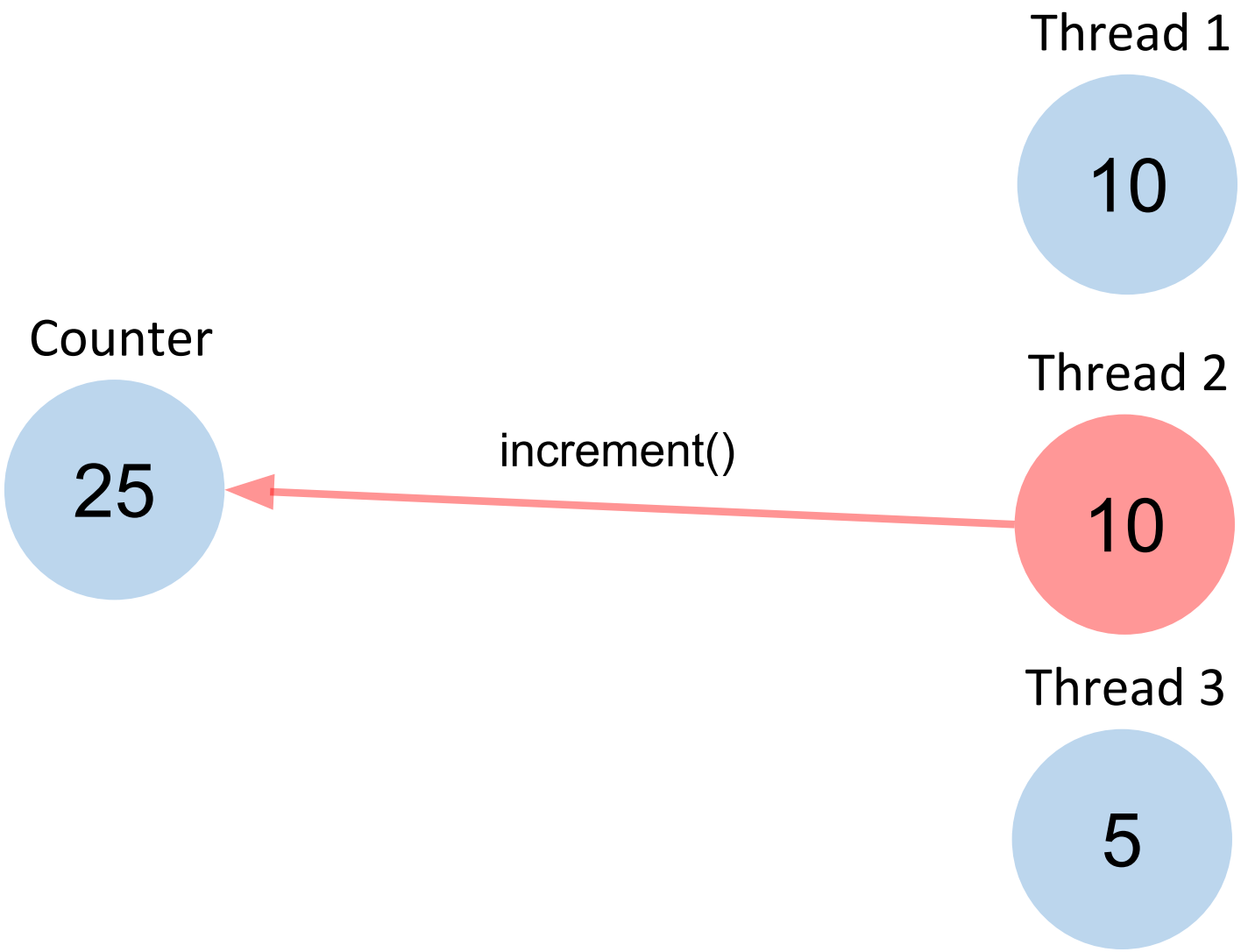
value of the shared Counter

number of times `increment()` is called



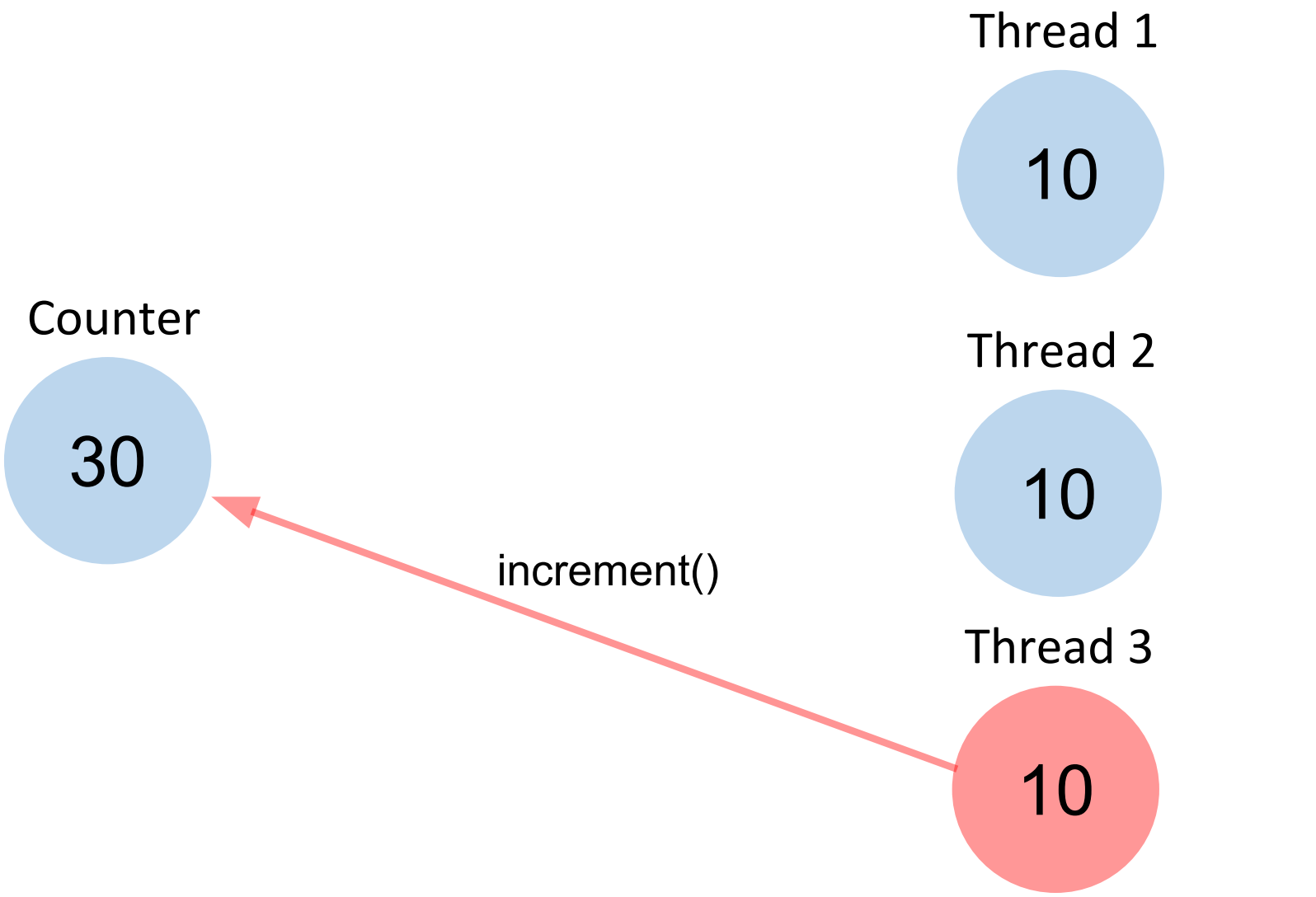
value of the shared Counter

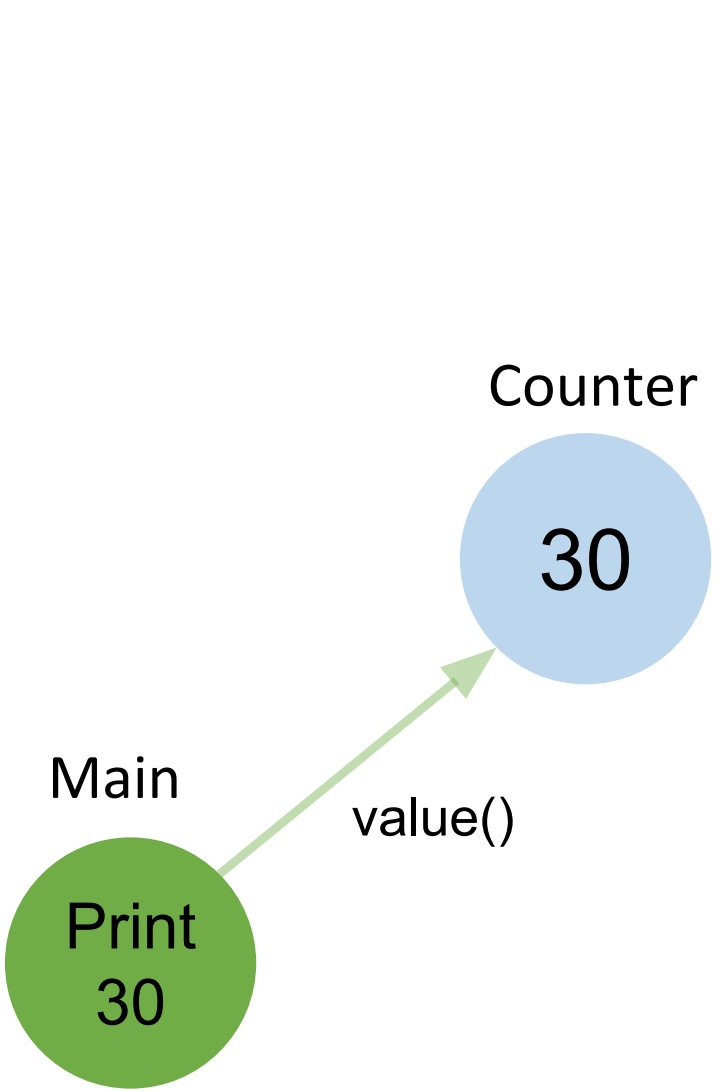
number of times `increment()` is called



value of the
shared Counter

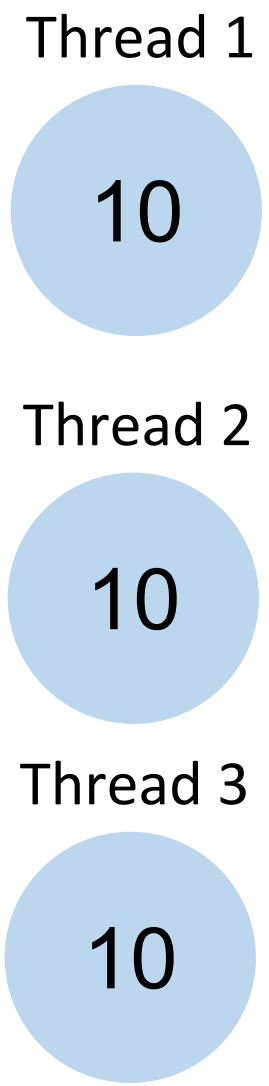
number of times
`increment()` is called


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

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.
- ➔ Notice how the expected value of counter at the end of execution is not what we would expect. Discuss why this is the case.

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
- ➔ 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

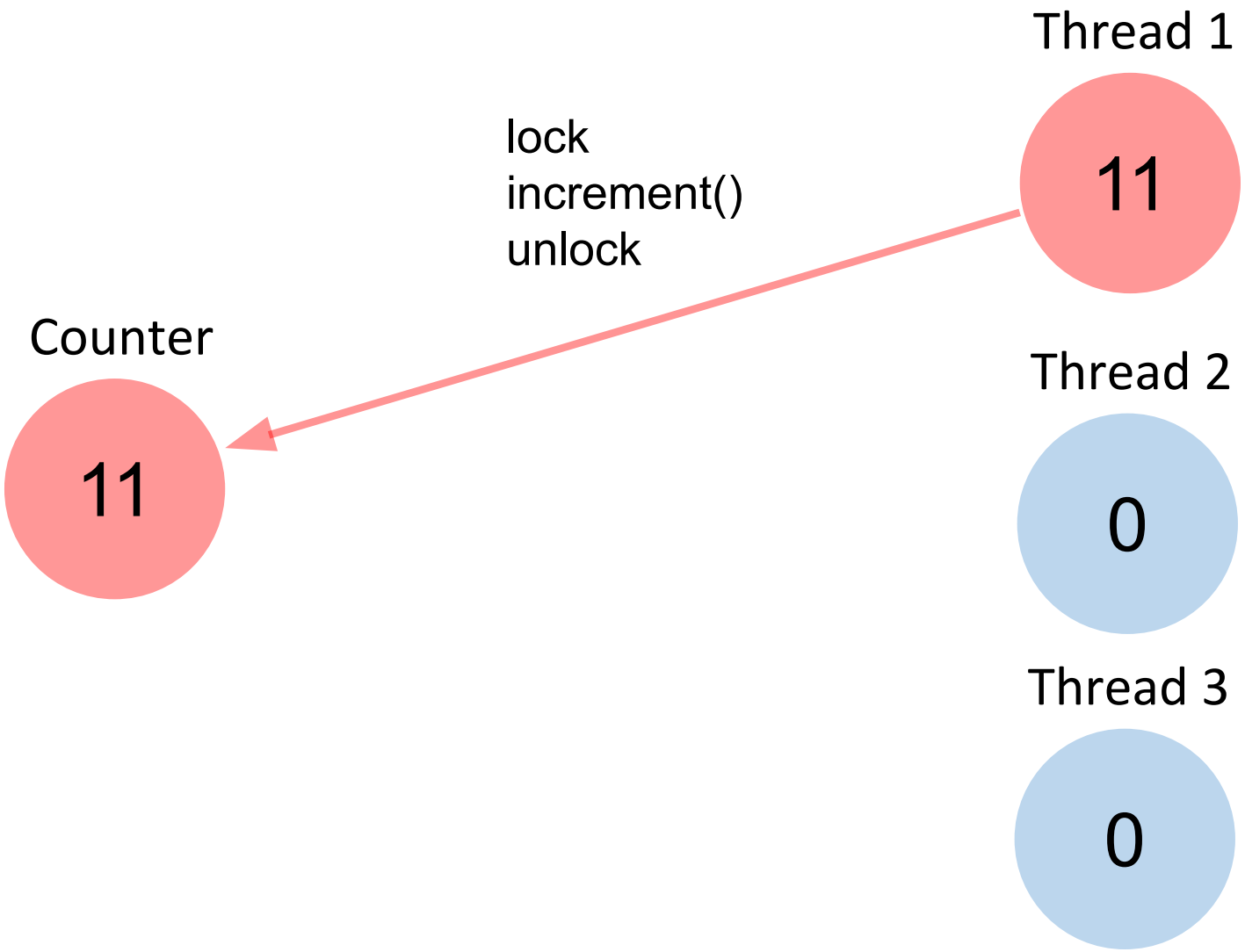
`foo.xMethod() //lock on foo`

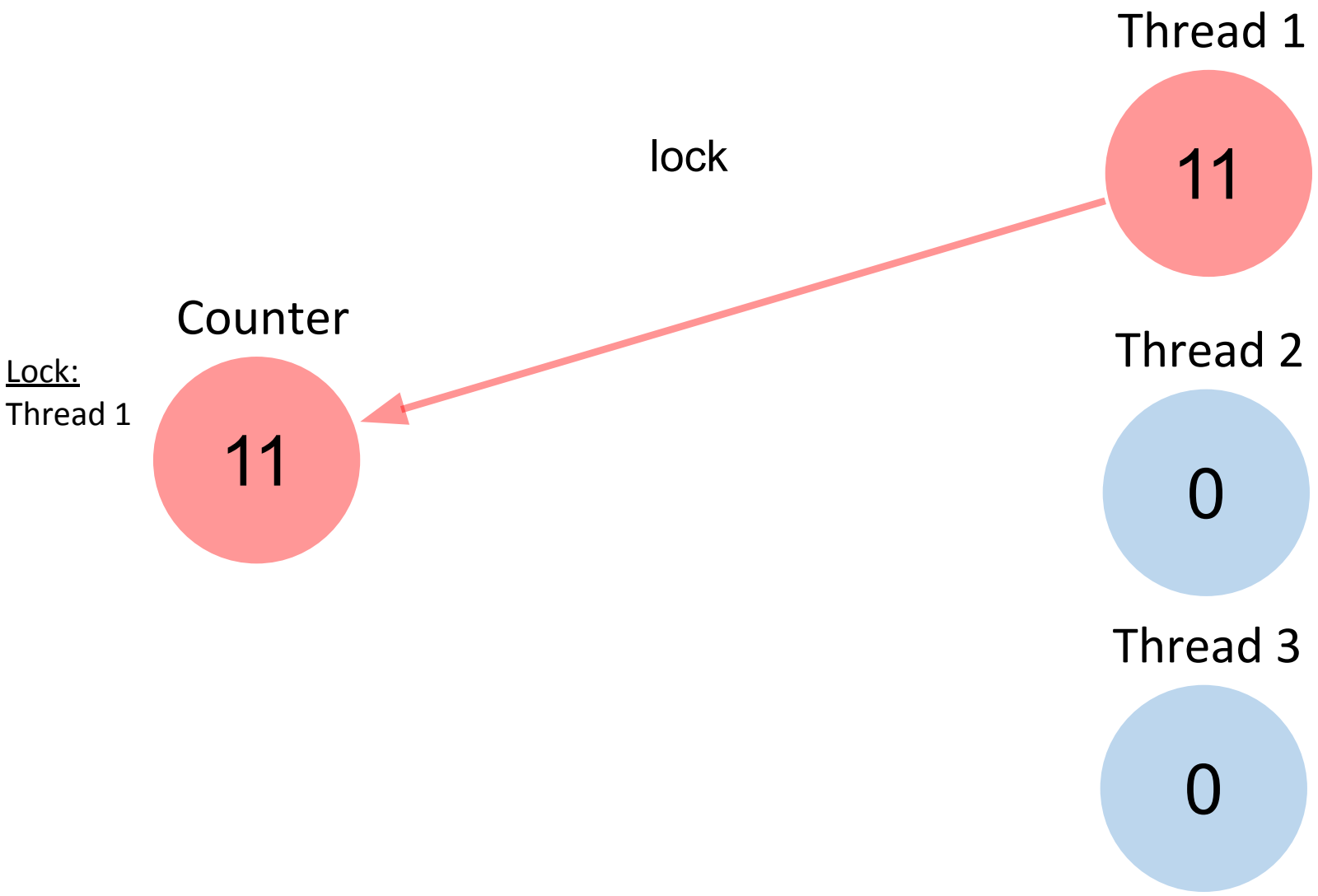
➔ Synchronized keyword obtains a lock on the parameter object

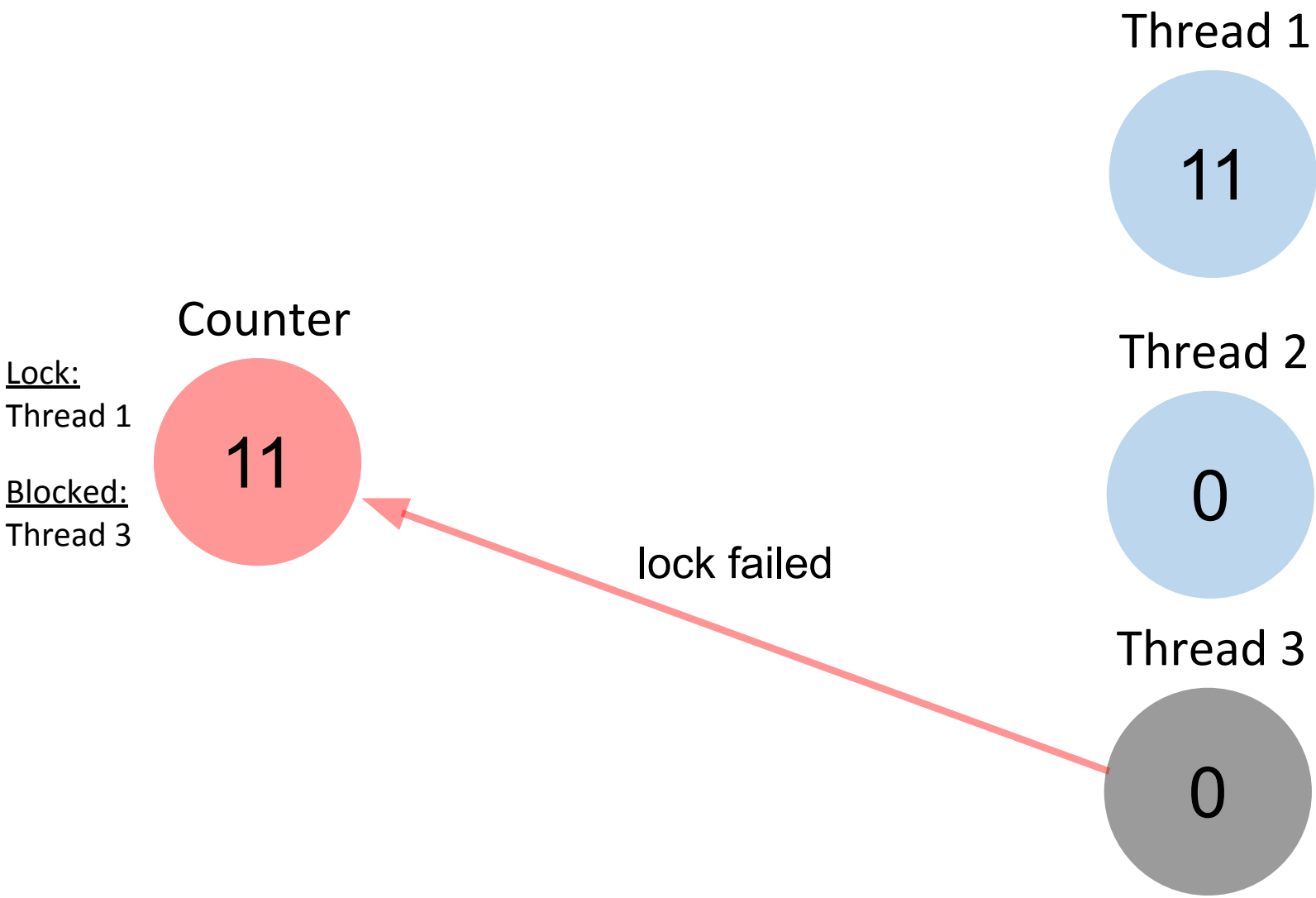
`synchronized (bar) { ... } //lock on bar`

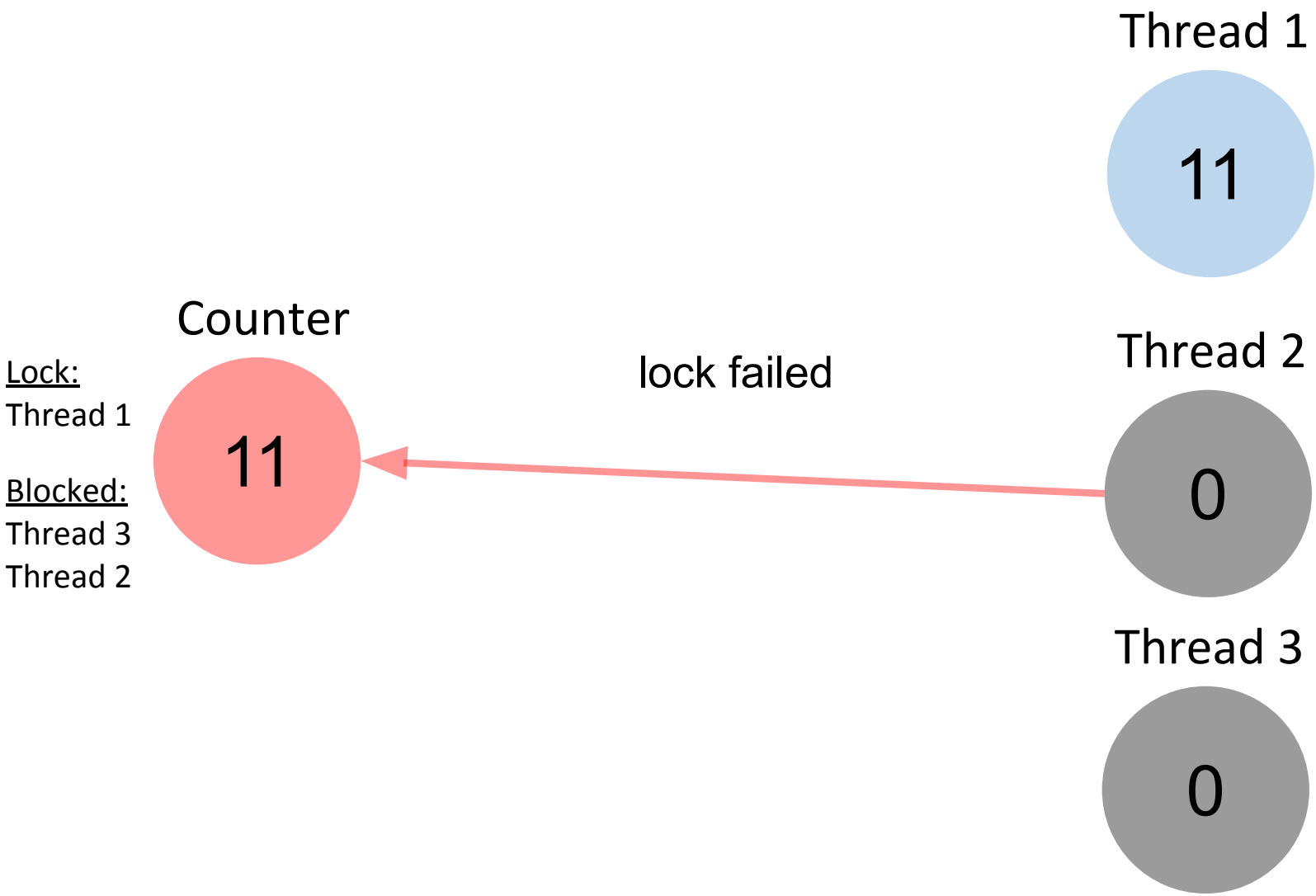
➔ A thread can obtain multiple locks (by nesting the synchronized blocks)

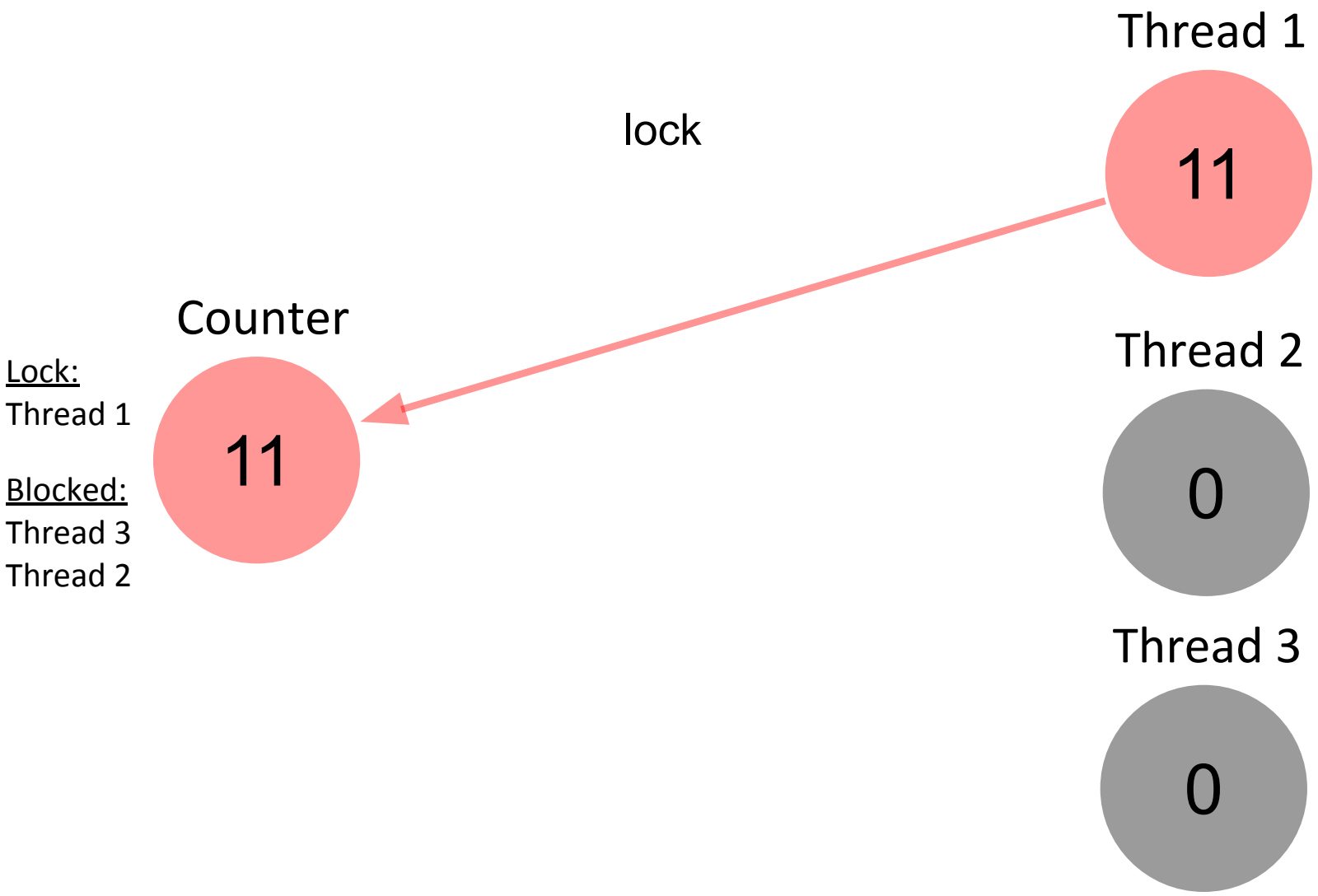

Thread 1
10
Counter
10
Thread 2
0
Thread 3
0

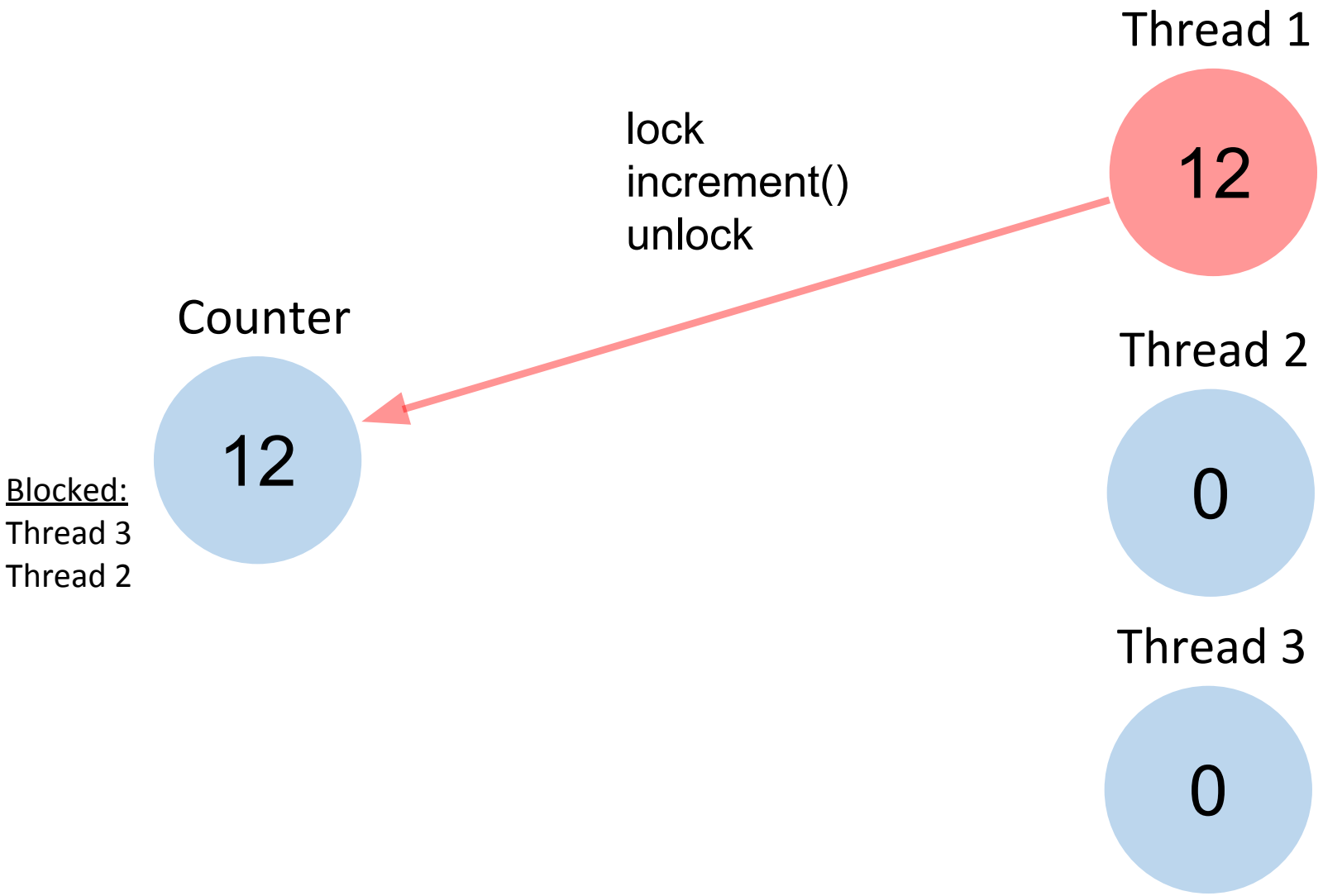

Thread 1
11
lock
increment()
unlock
Counter
11
Thread 2
0
Thread 3
0


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). Can you see a pattern in how the threads are scheduled? Discuss what might be the reason for this behaviour.

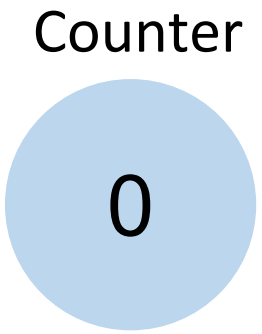
Task D

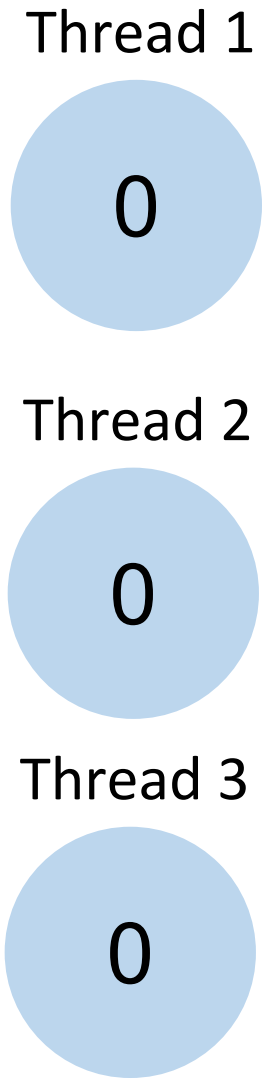
- ➔ Implement a FairThreadCounter that ensures that different threads increment the Counter in an round-robin fashion. In round-robin scheduling the threads perform the increments in circular order. That is, two threads with ids 1 and 2 would increment the value in the following order 1, 2, 1, 2, 1, 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

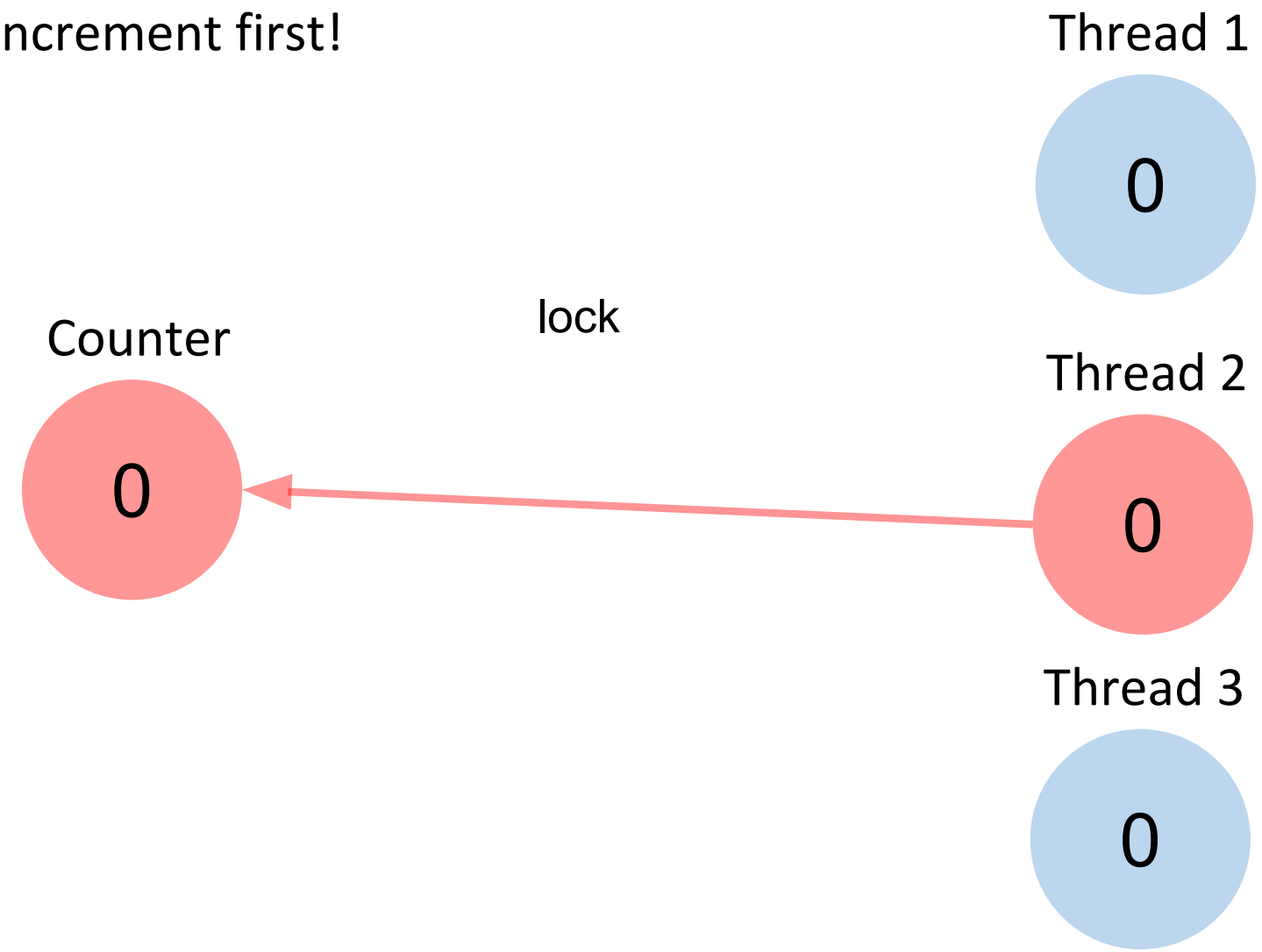
- ➔ Object provides wait() and notify() methods
- ➔ To call wait() on an object thread must own its lock
- ➔ Thread releases the lock and is added to the “waiting list” for that object
- ➔ Thread waits until a notify method is called on the object
- ➔ notify() removes one (arbitrary) thread from the object’s “waiting list”
- ➔ notifyAll() removes all the threads

Thread 1 must increment first!

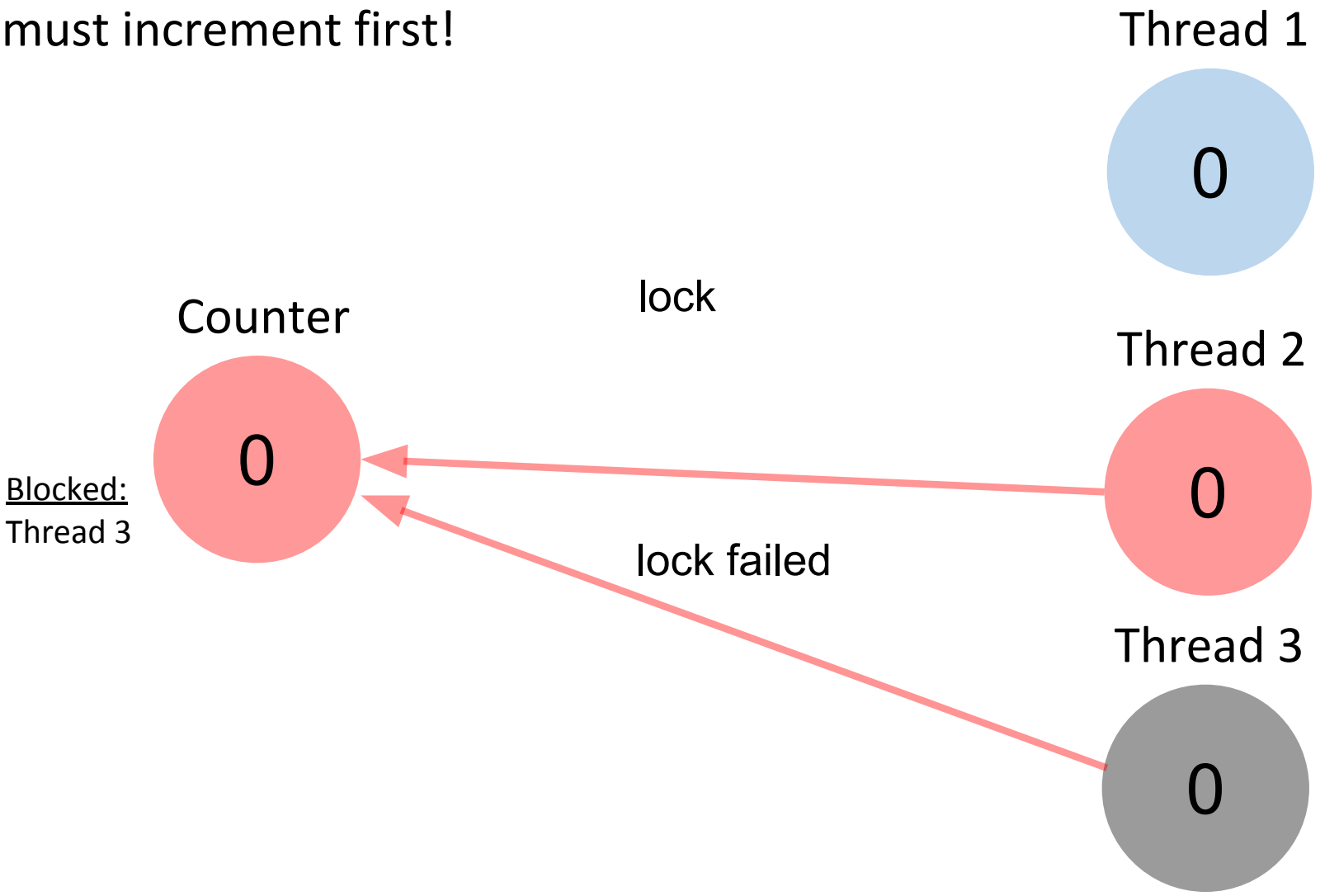




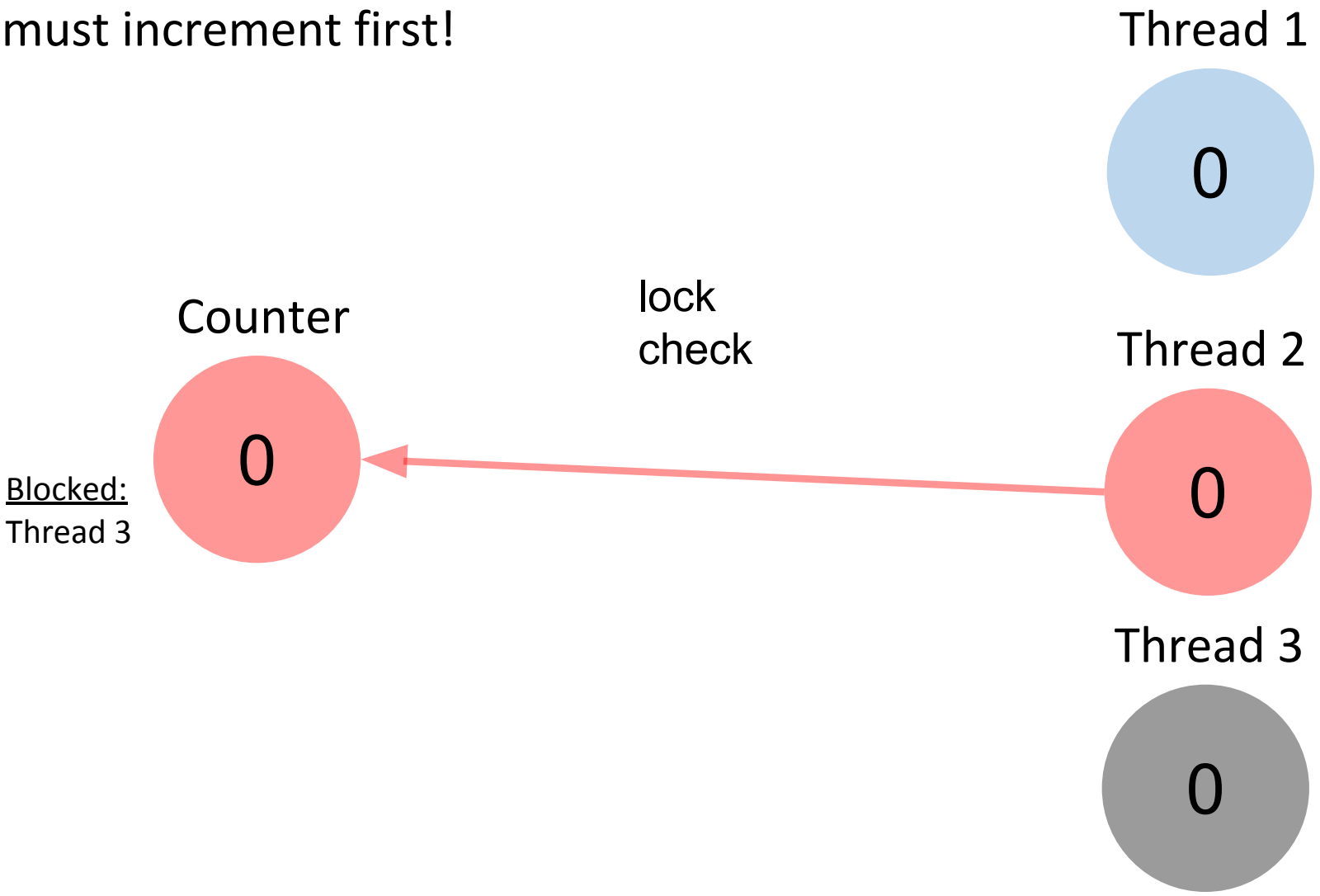
Thread 1 must increment first!



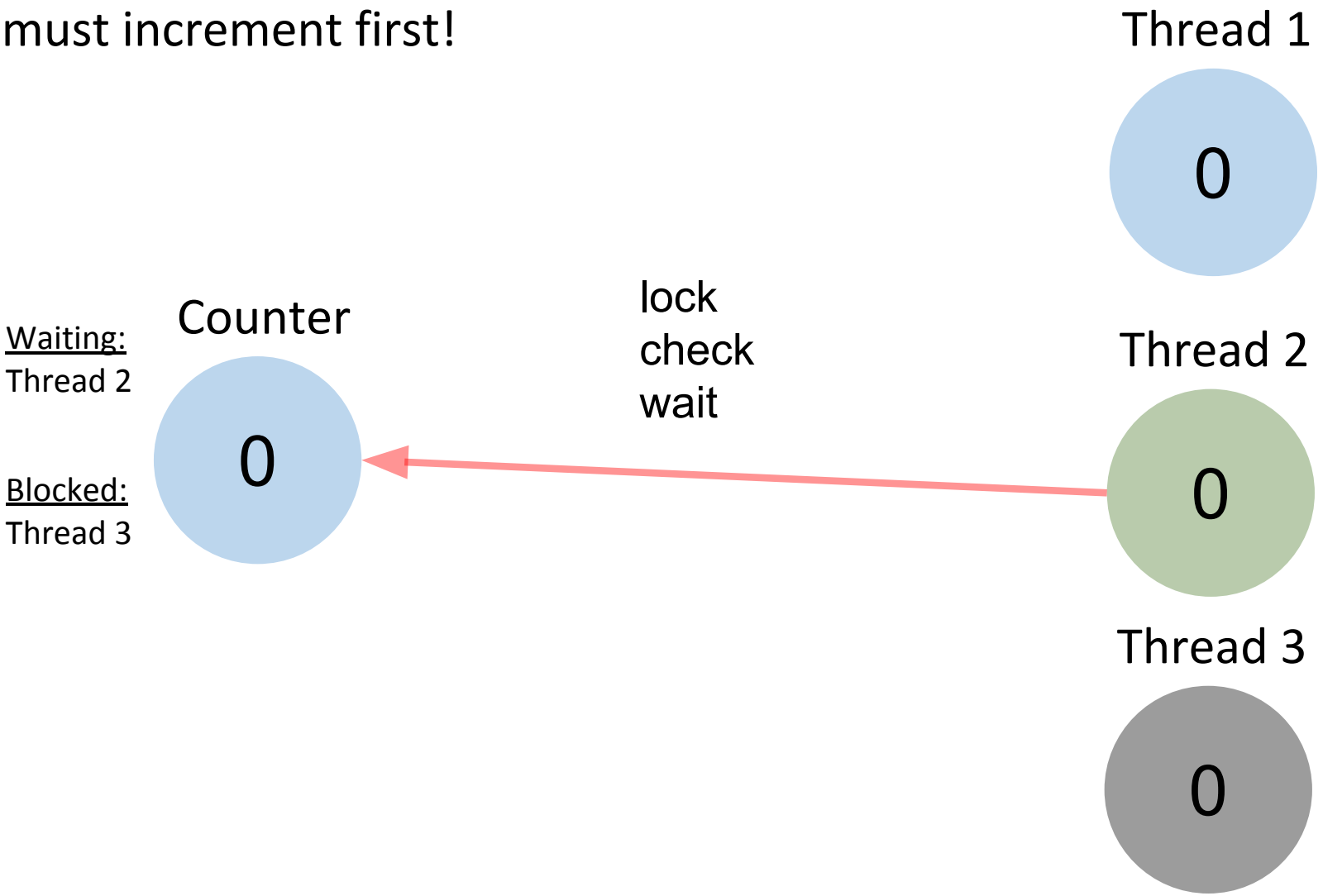
Thread 1 must increment first!



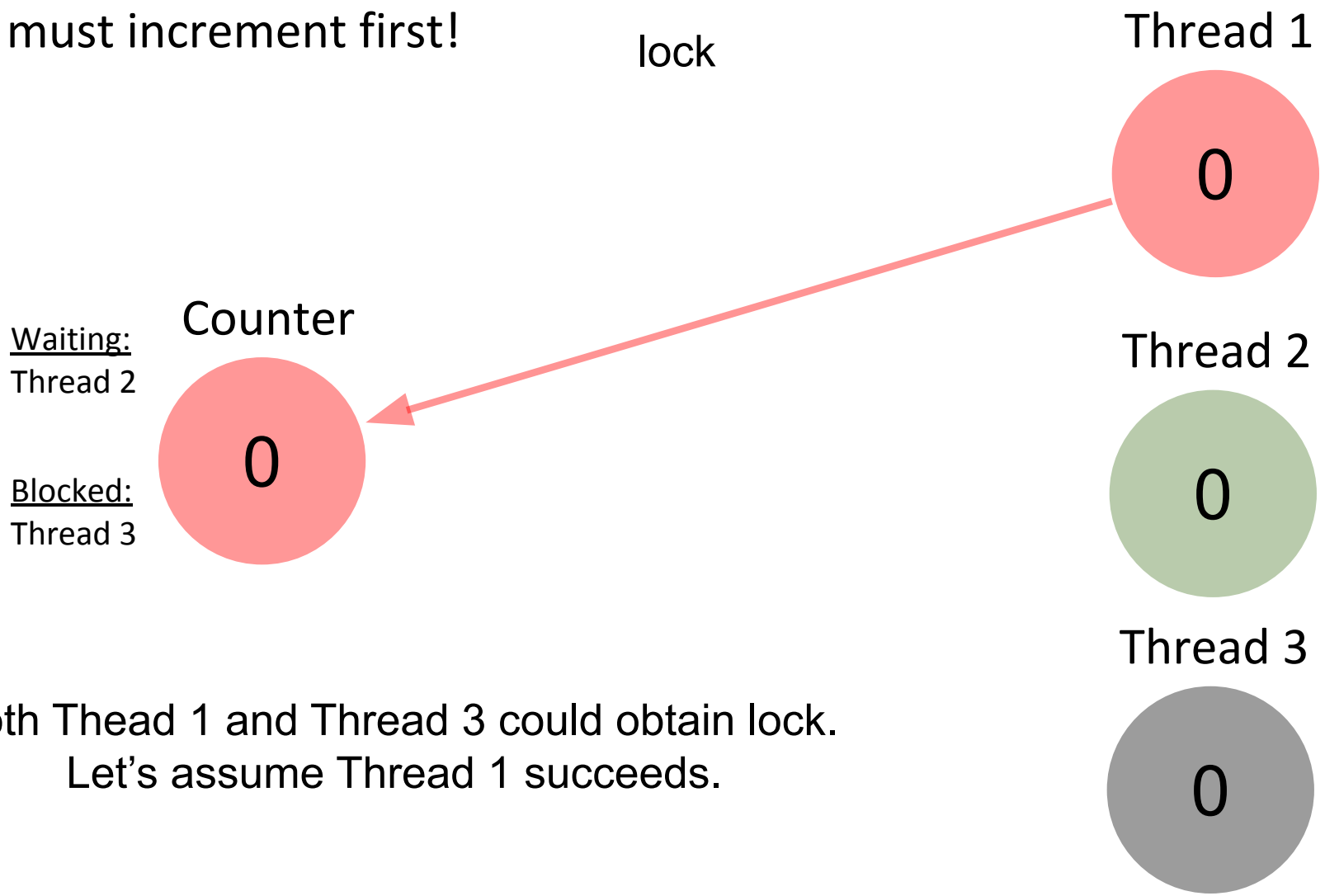
Thread 1 must increment first!



Thread 1 must increment first!

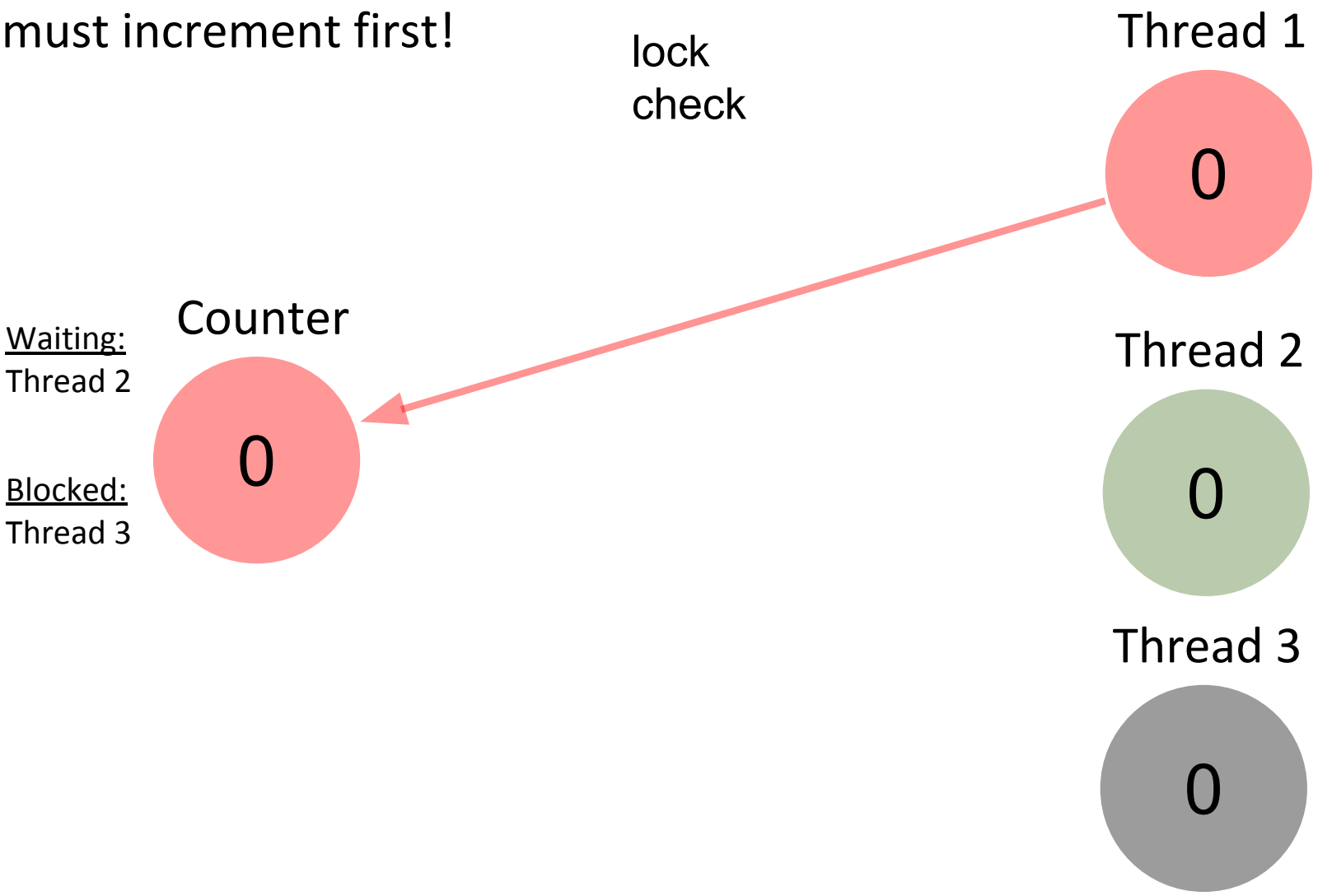


Thread 1 must increment first!

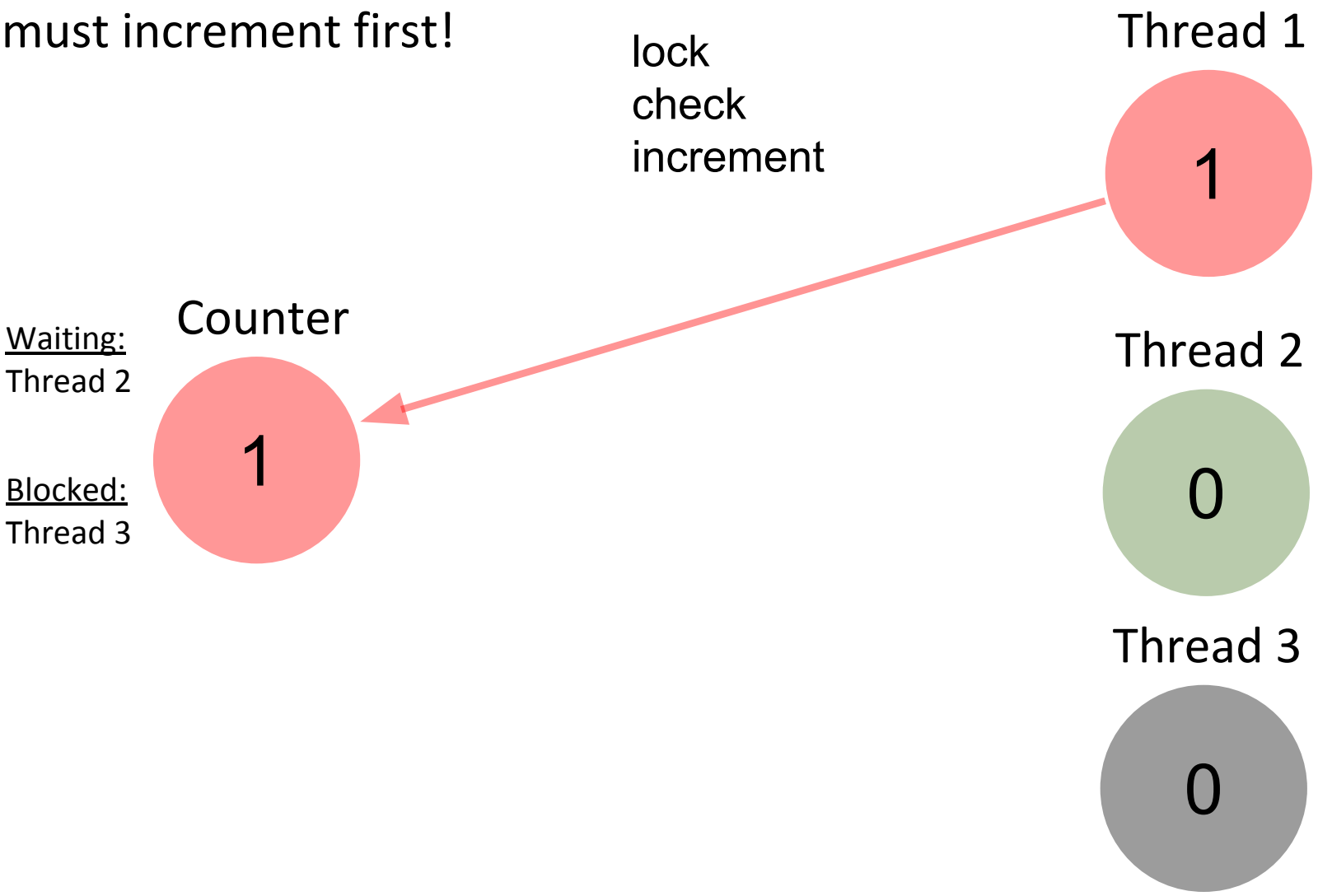


Both Thead 1 and Thread 3 could obtain lock.
Let's assume Thread 1 succeeds.

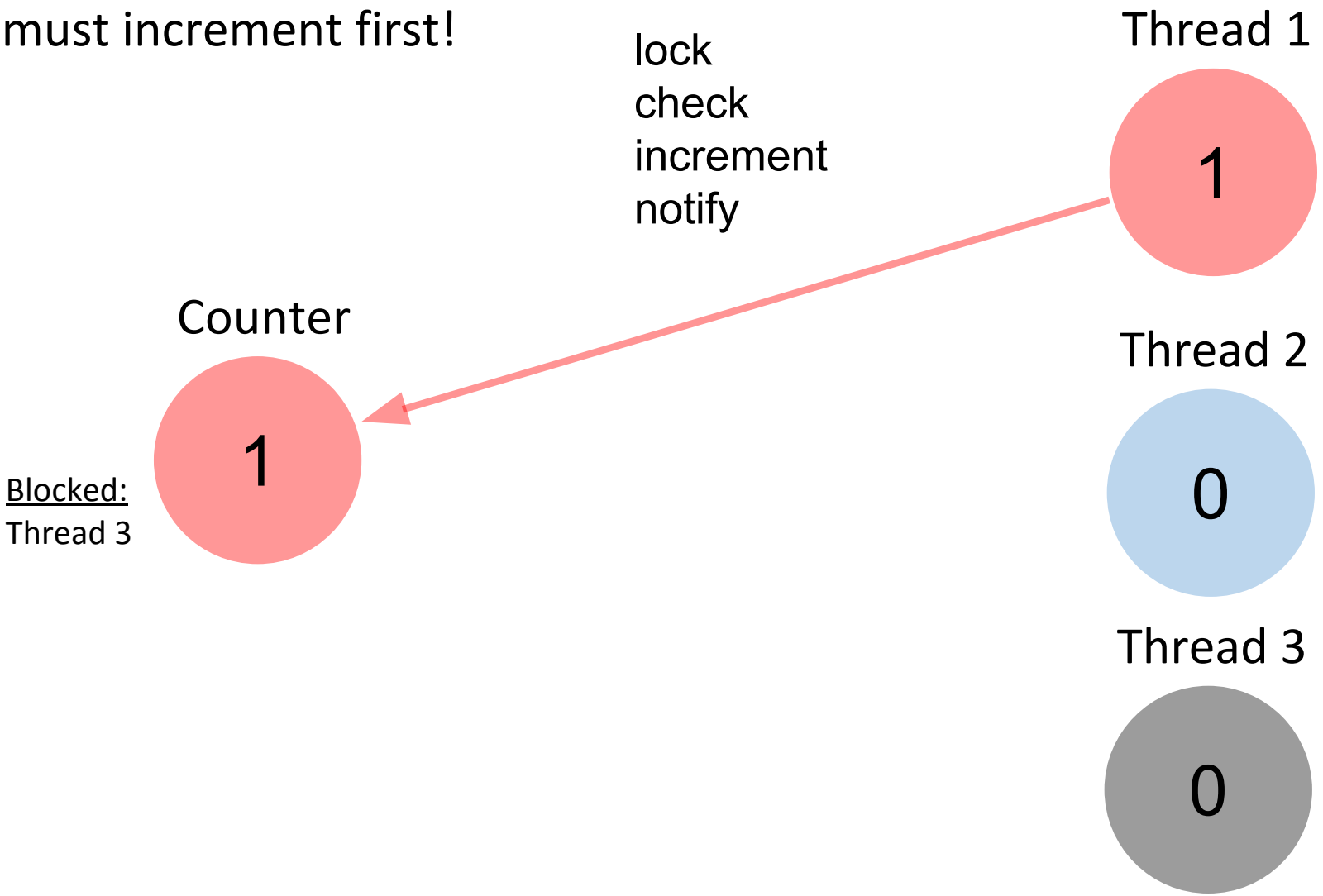
Thread 1 must increment first!



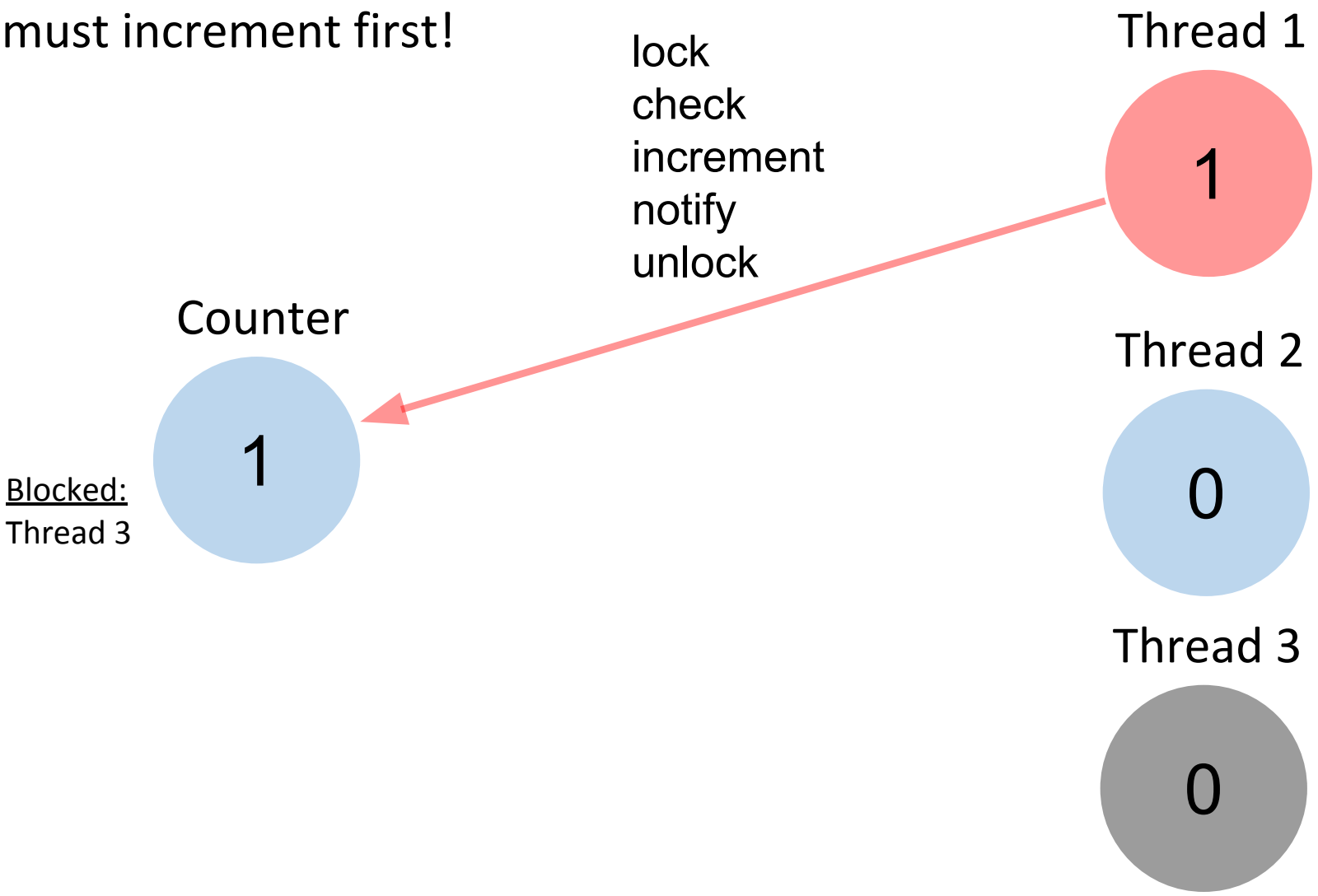
Thread 1 must increment first!



Thread 1 must increment first!



Thread 1 must increment first!



Task E (Optional) – Atomic counter

Implement a thread safe version of the Counter in AtomicCounter. In this version we will use and 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*

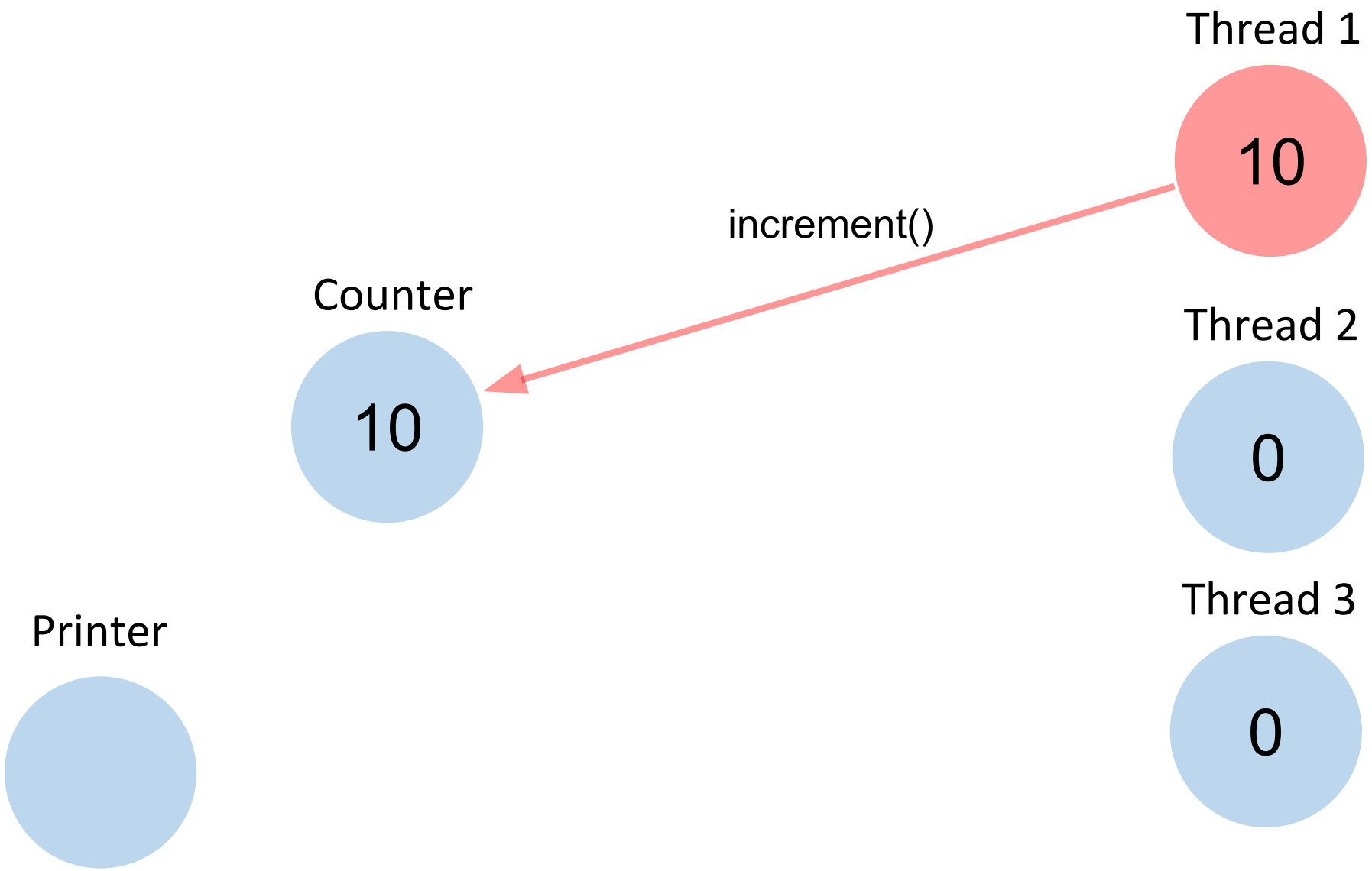
Task F (Optional) – Atomic vs Synchronized counter

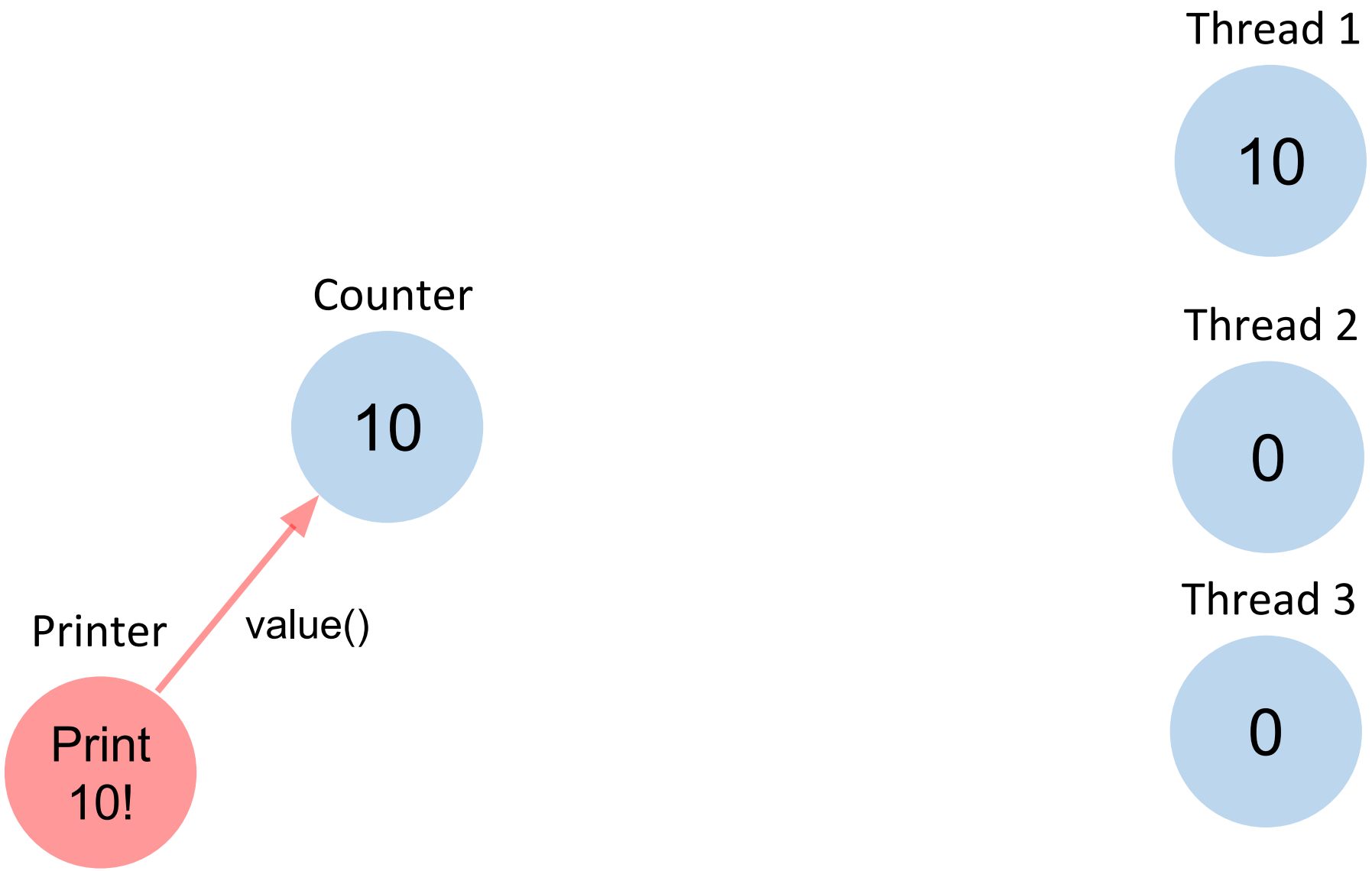
Experimentally compare the AtomicCounter and SynchronizedCounter implementations by measuring which one is faster. Observe the differences in the CPU load between the two versions. Can you explain what is the cause of different performance characteristics?

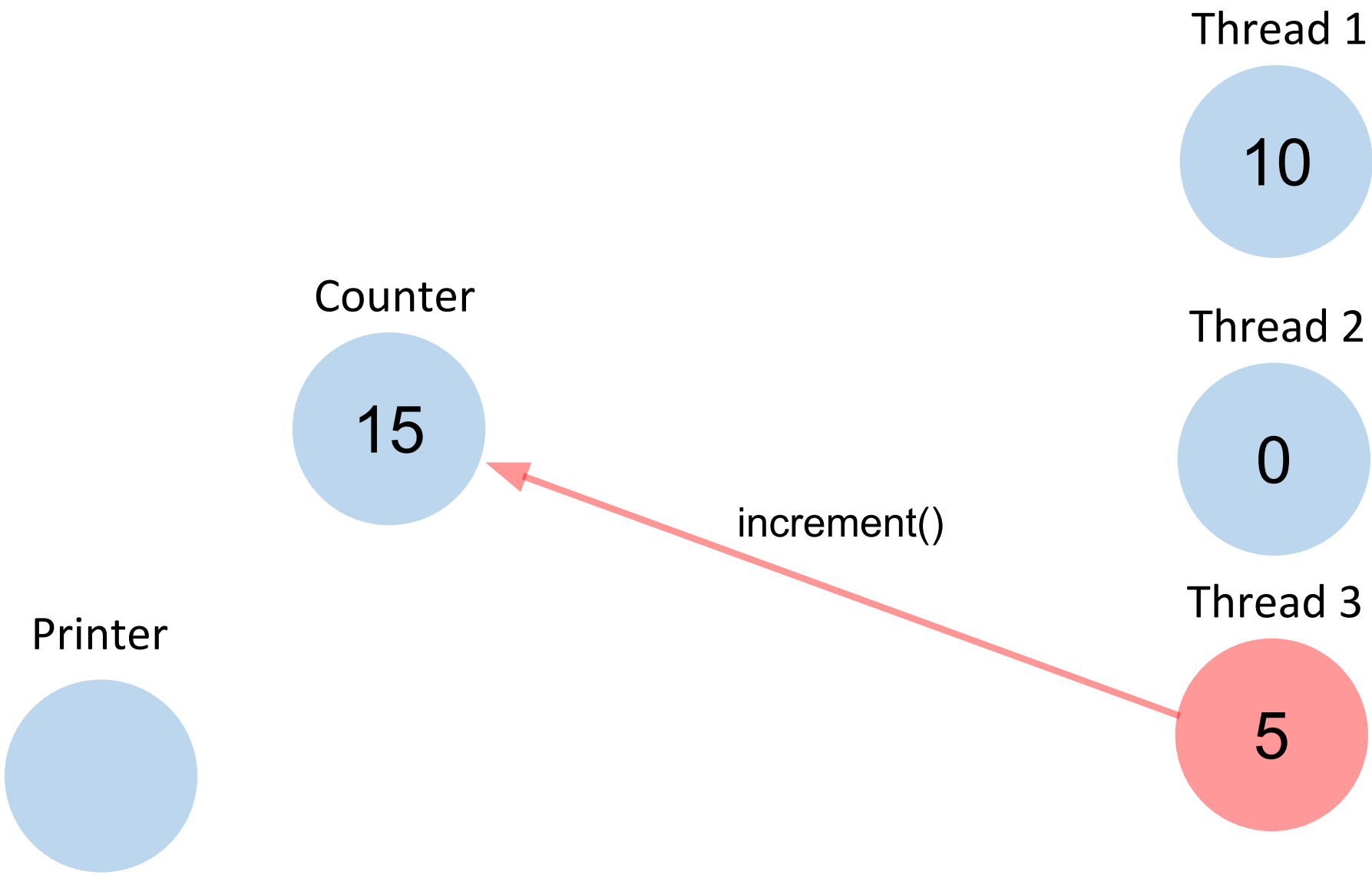
- Vary the load per thread
- Vary the number of threads

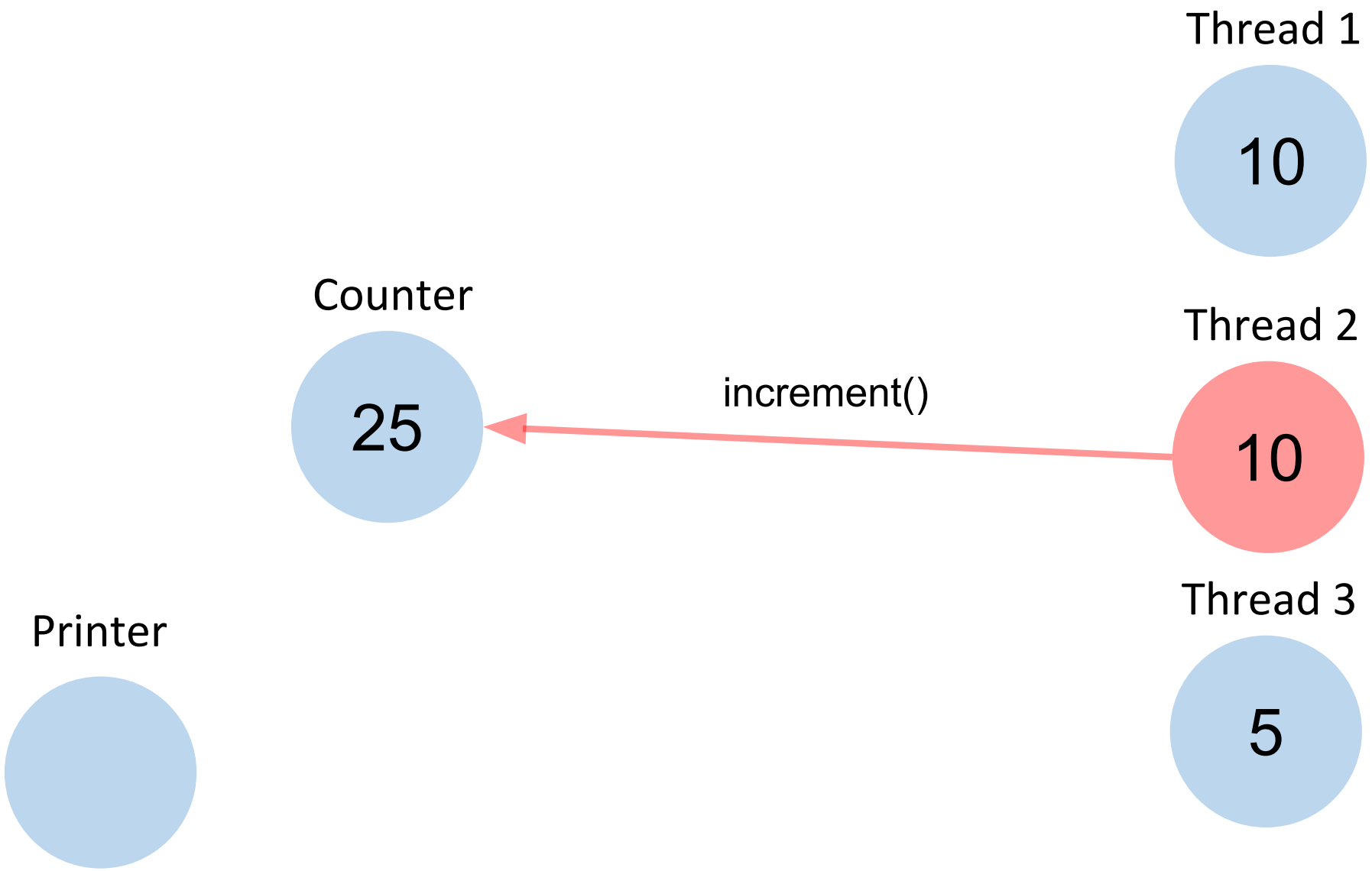
Task G (Optional)

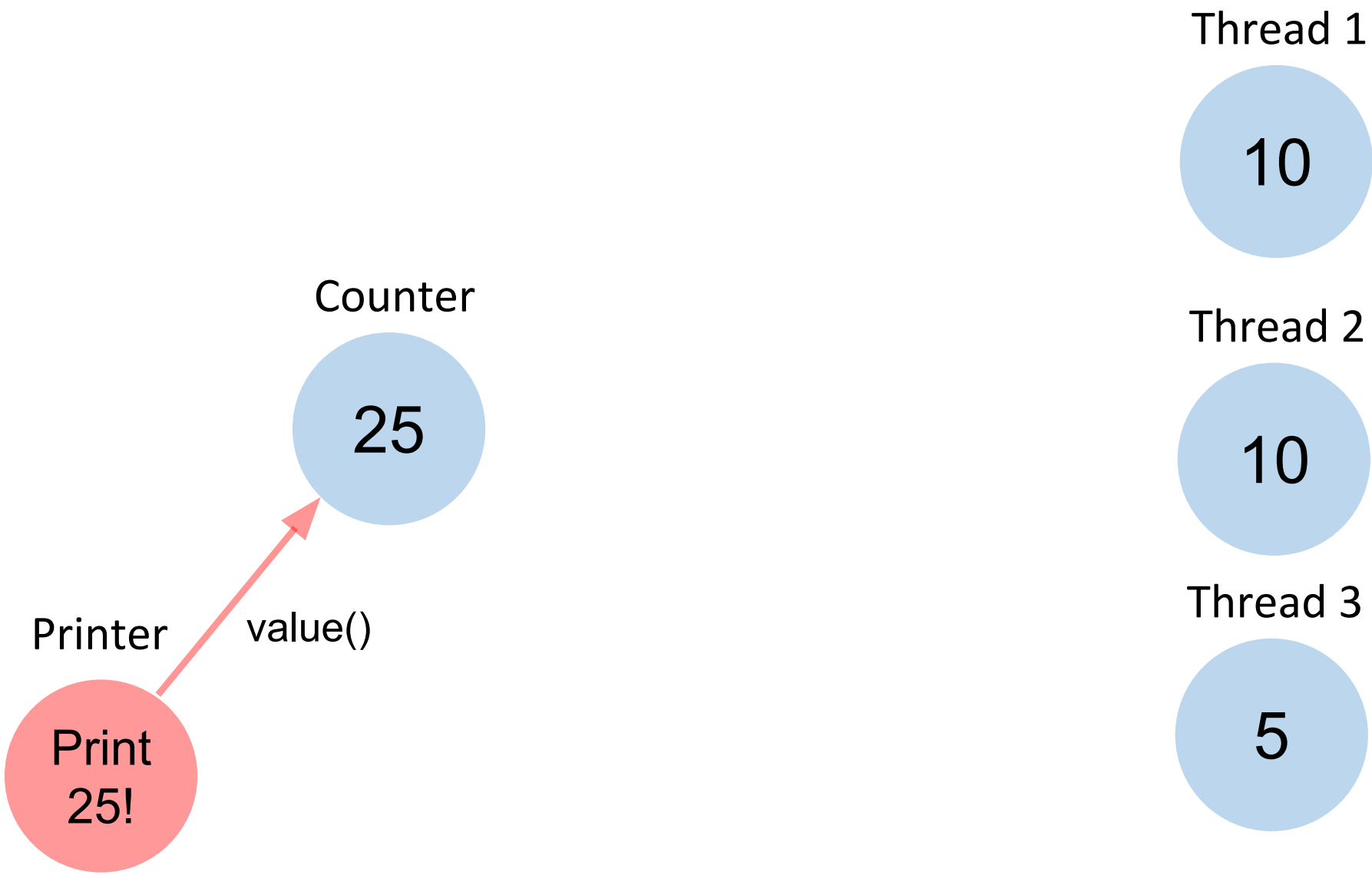
Implement a thread that measures execution progress. That is, create a thread that observes the values of the Counter during the execution and prints them to the console. Make sure that the thread is properly terminated once all the work is done.

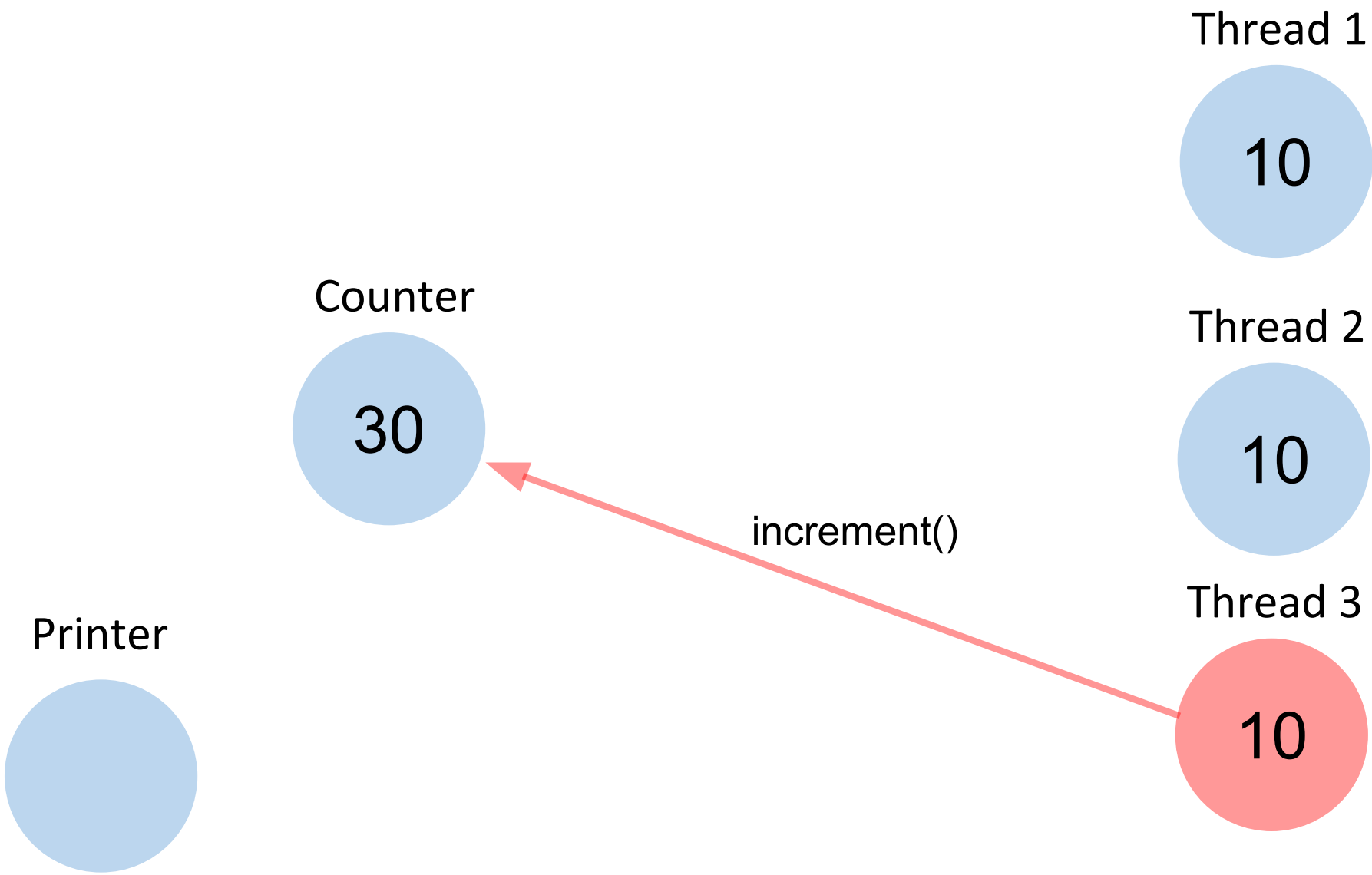

Thread 1
10
increment()
Counter
10
Thread 2
0
Printer
Thread 3
0

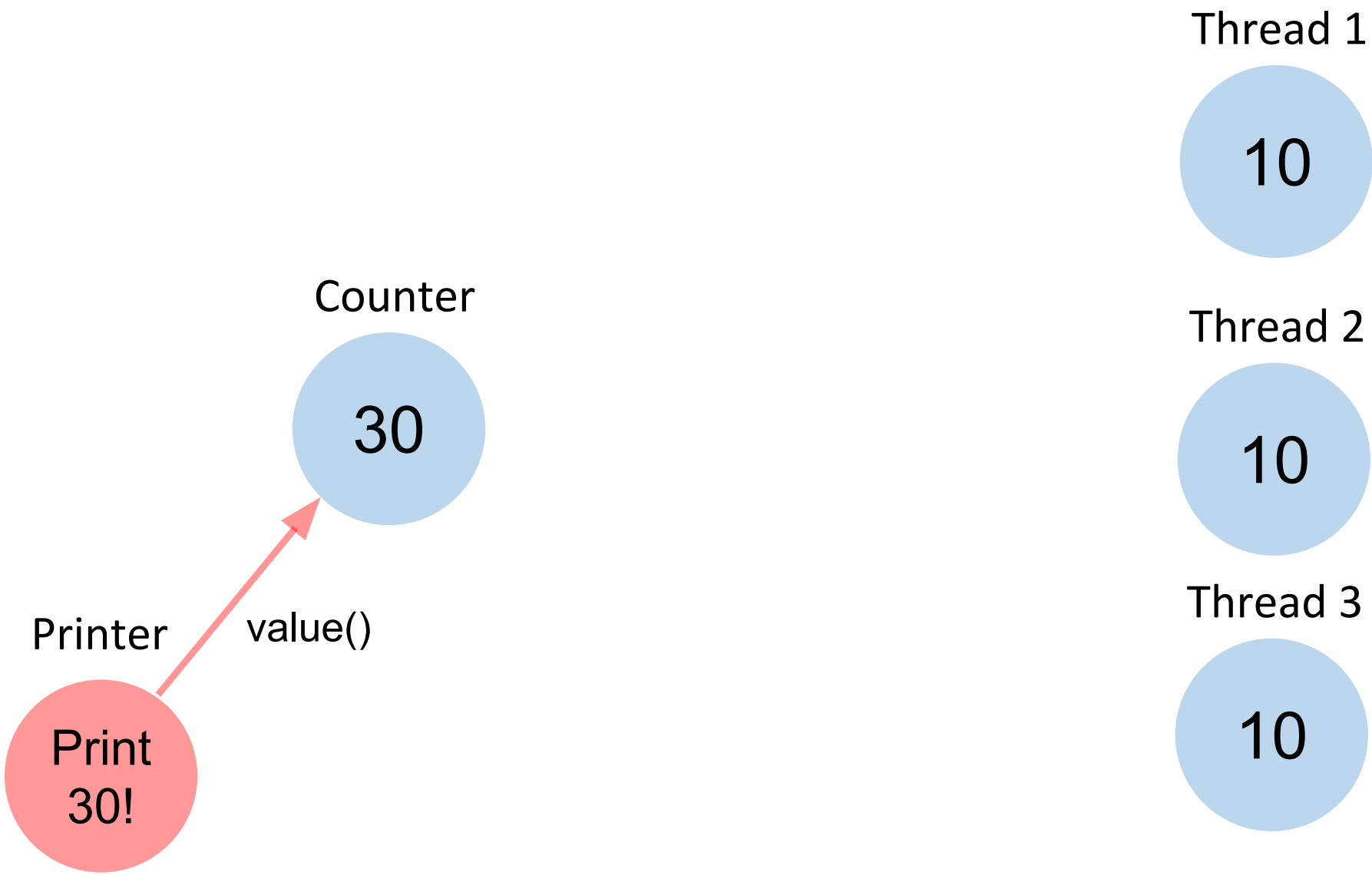

Thread 1
10
Counter
10
Thread 2
0
Printer
value()
Thread 3
0
Print
10!


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!