

Parallel Programming Exercise Session 3

Spring 2024

Plan für heute

Nachbesprechung
Übung 2

Vorbesprechung
Übung 3

Theorie

Demo

Alte
Prüfungsaufgaben

Quiz

Nachbesprechung Übung 2

Nachbesprechung Übung 2

- Task A: Hello Thread
- Task B: Single Thread `computePrimeFactors()`
- Task C: Overhead von Thread messen
- Task D: `PartitionData()` implementieren
- Task E: `computePrimeFactors()` parallelisieren
- Task F: Über Speedup nachdenken
- Task G: Messung für versch. Anz. Threads und Längen

Task E: Teilen von Daten zwischen Threads

Idee: Überlegt jedem Thread eine Referenz auf das geteilte Objekt.

Aufpassen: Bad Interleavings möglich, wenn mehrere Threads das Objekt gleichzeitig modifizieren.

Gleichzeitiges lesen geht immer.

-> Musterlösung von Task E anschauen

Vorbesprechung Übung 3

Counter

Wir wollen zählen, wie oft ein Ereignis eingetreten ist.

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

Counter

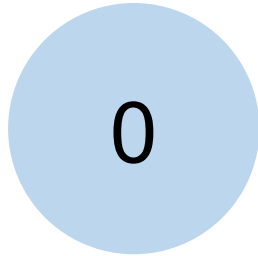
Let's count the number of times a given event occurs

```
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());  
}
```

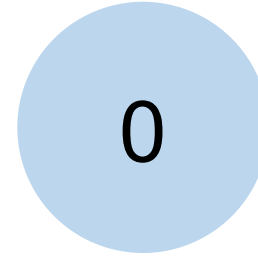

10 iterations each

Counter

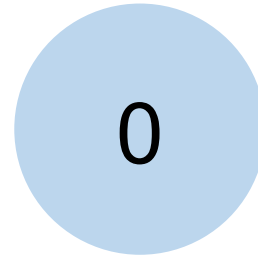


value of the
shared Counter

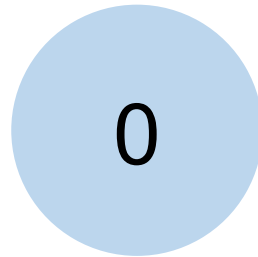
Thread 1



Thread 2

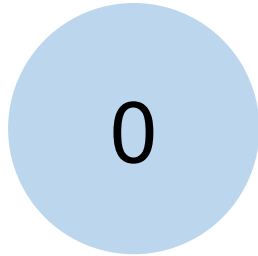


Thread 3



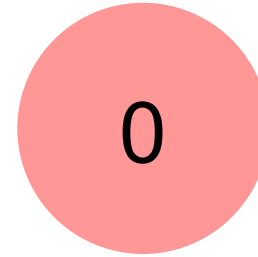
number of times
`increment()` is called

Counter

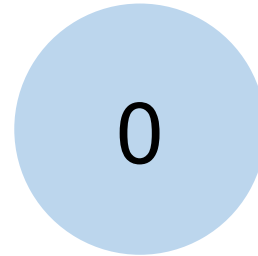


value of the
shared Counter

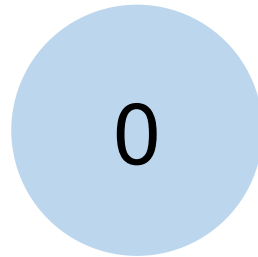
Thread 1



Thread 2

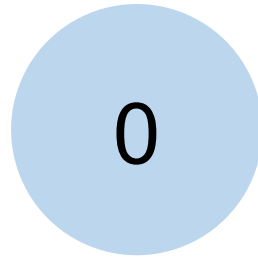


Thread 3



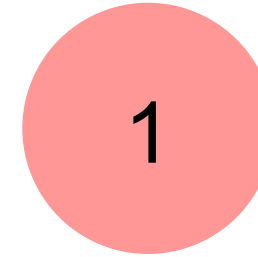
number of times
`increment()` is called

Counter

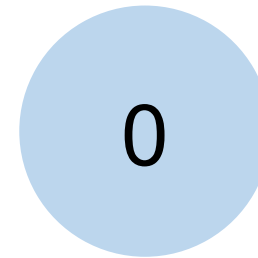


value of the
shared Counter

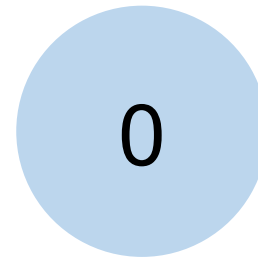
Thread 1



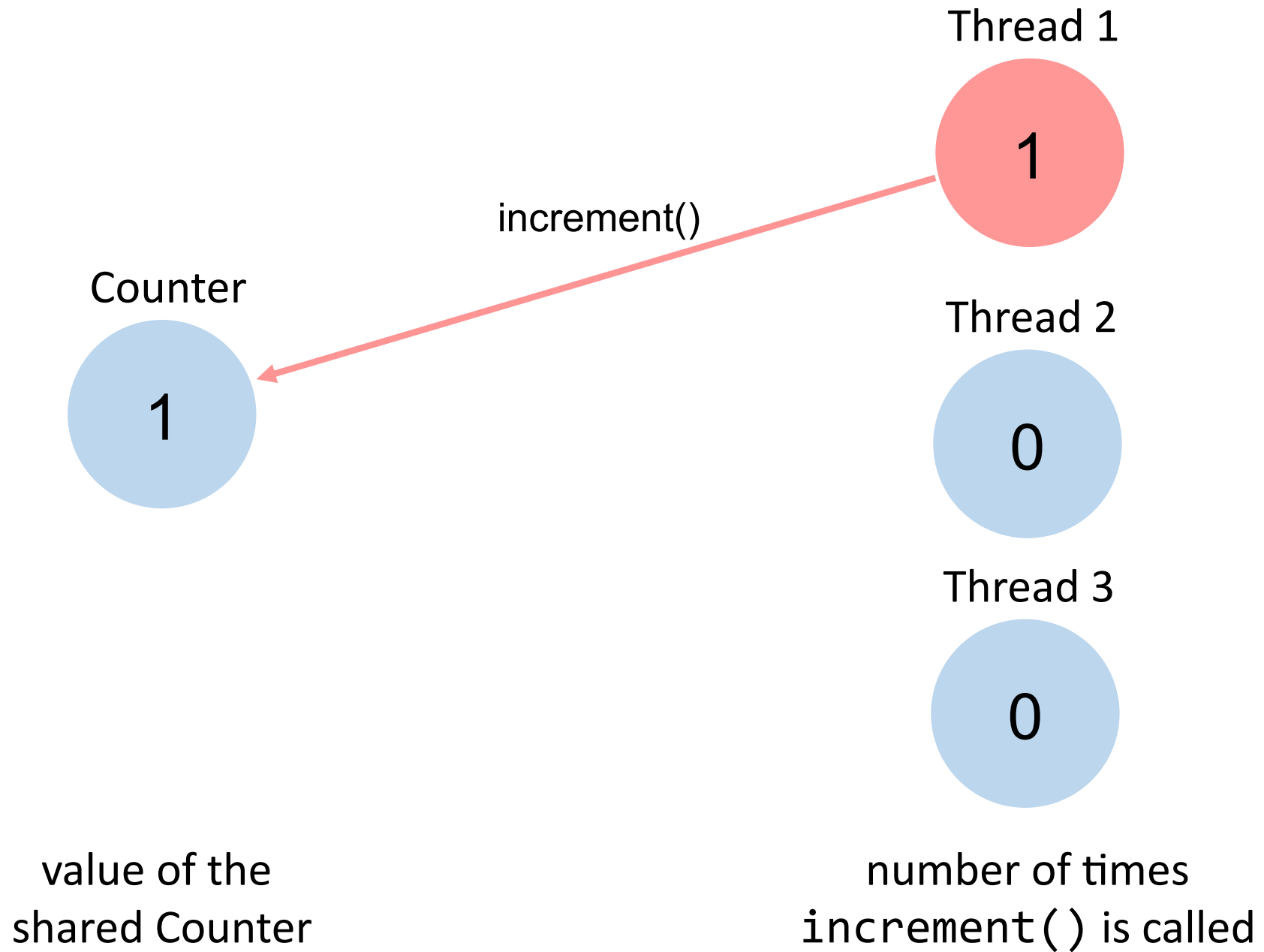
Thread 2

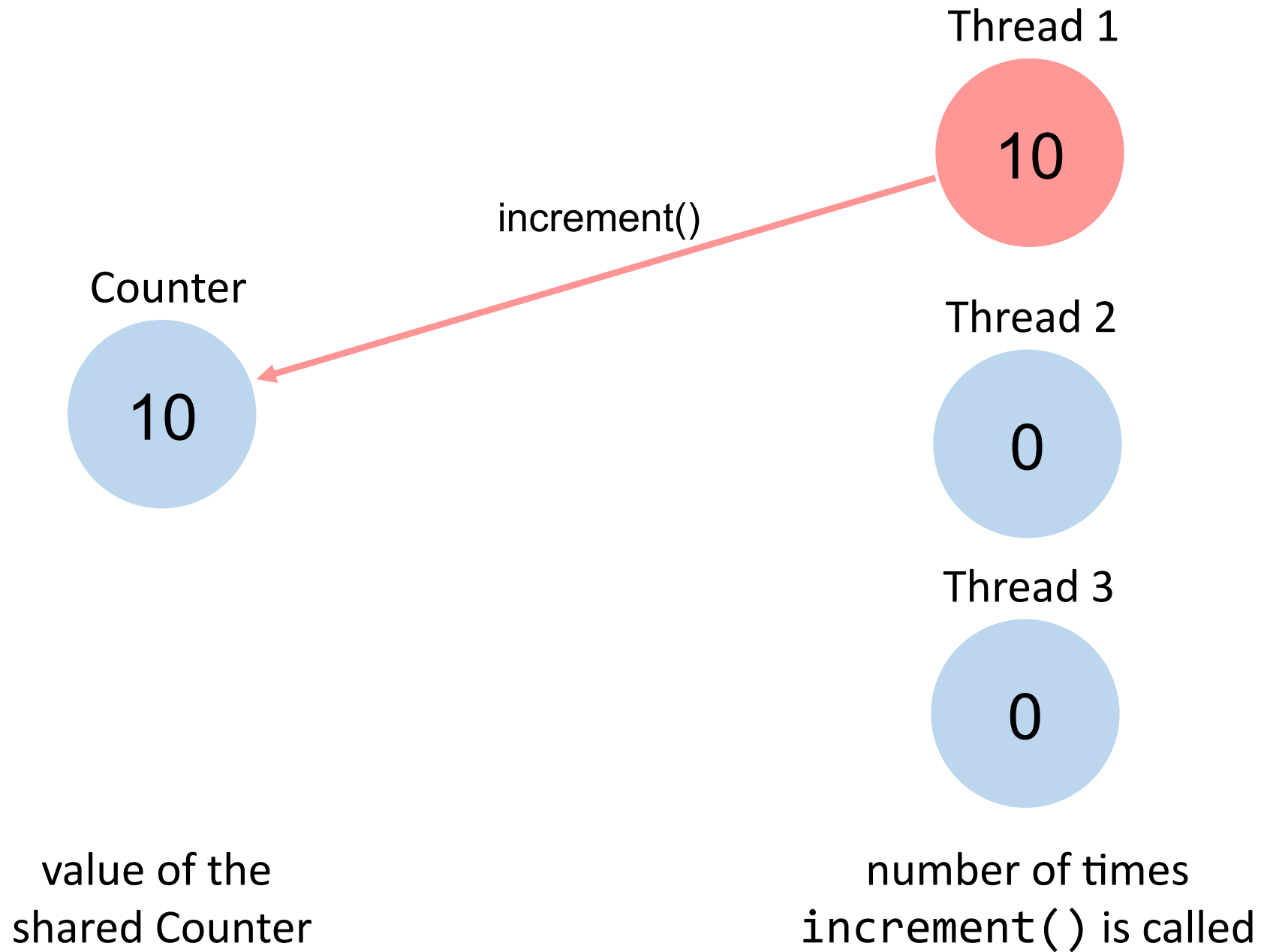


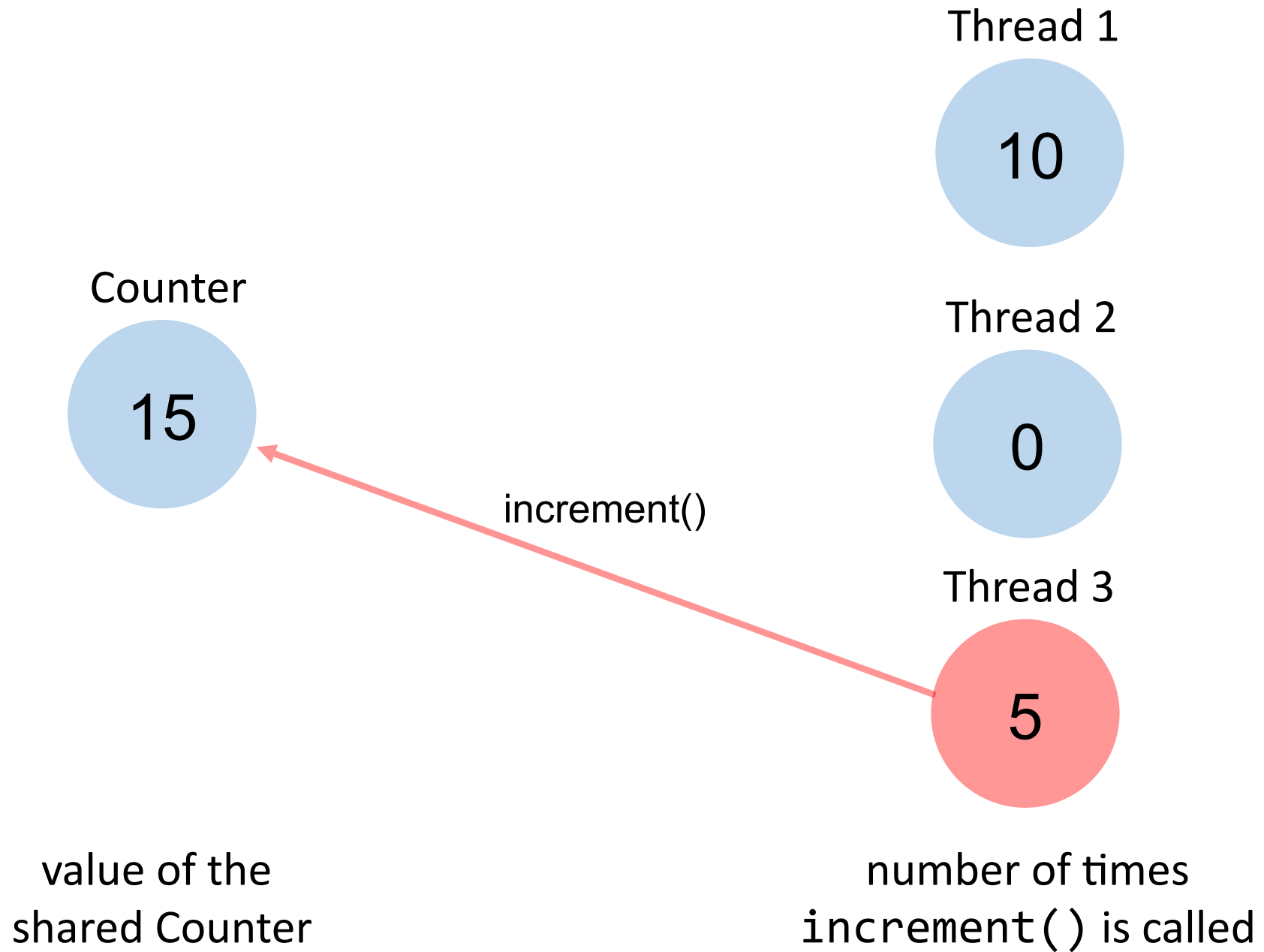
Thread 3

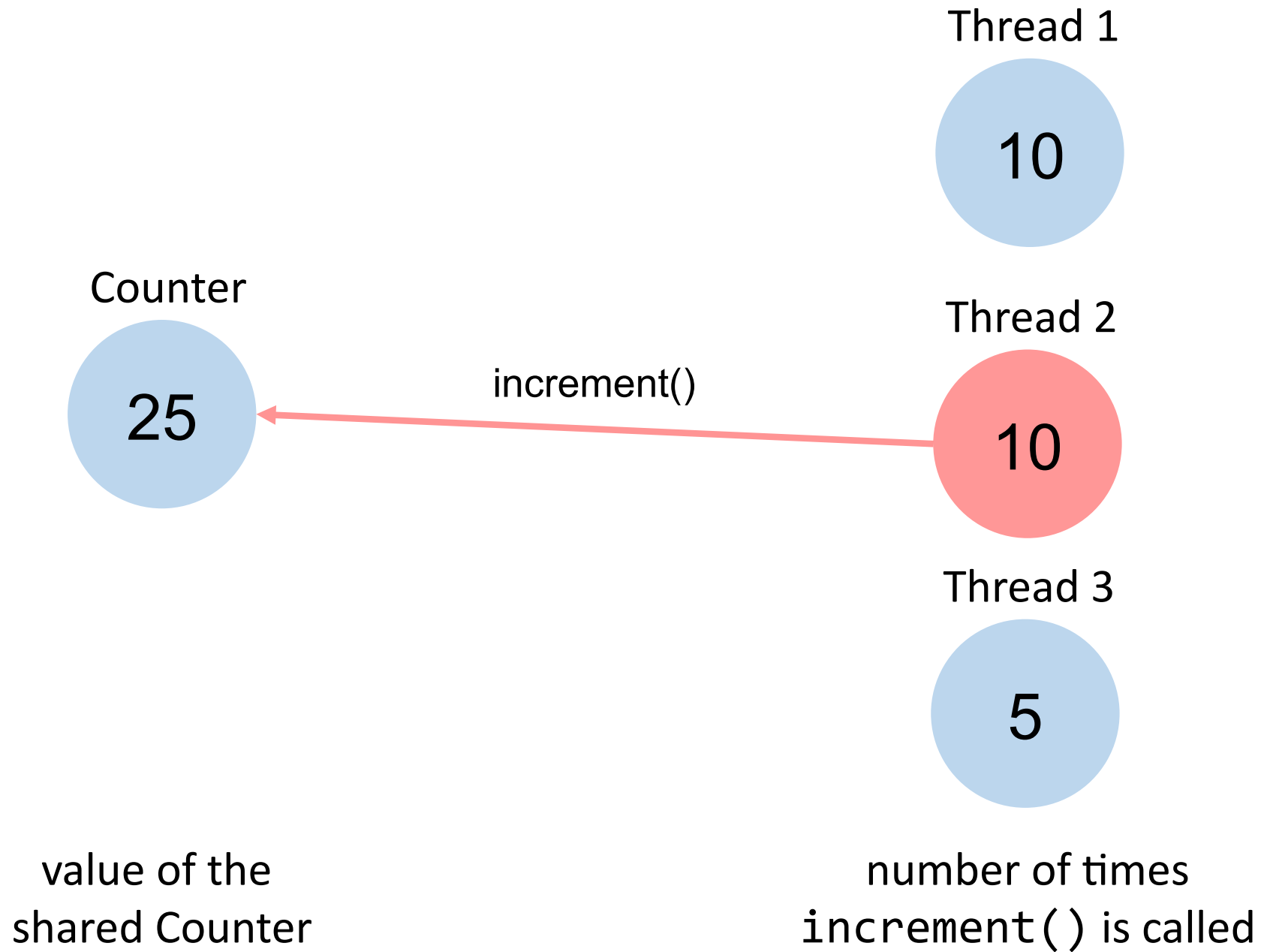


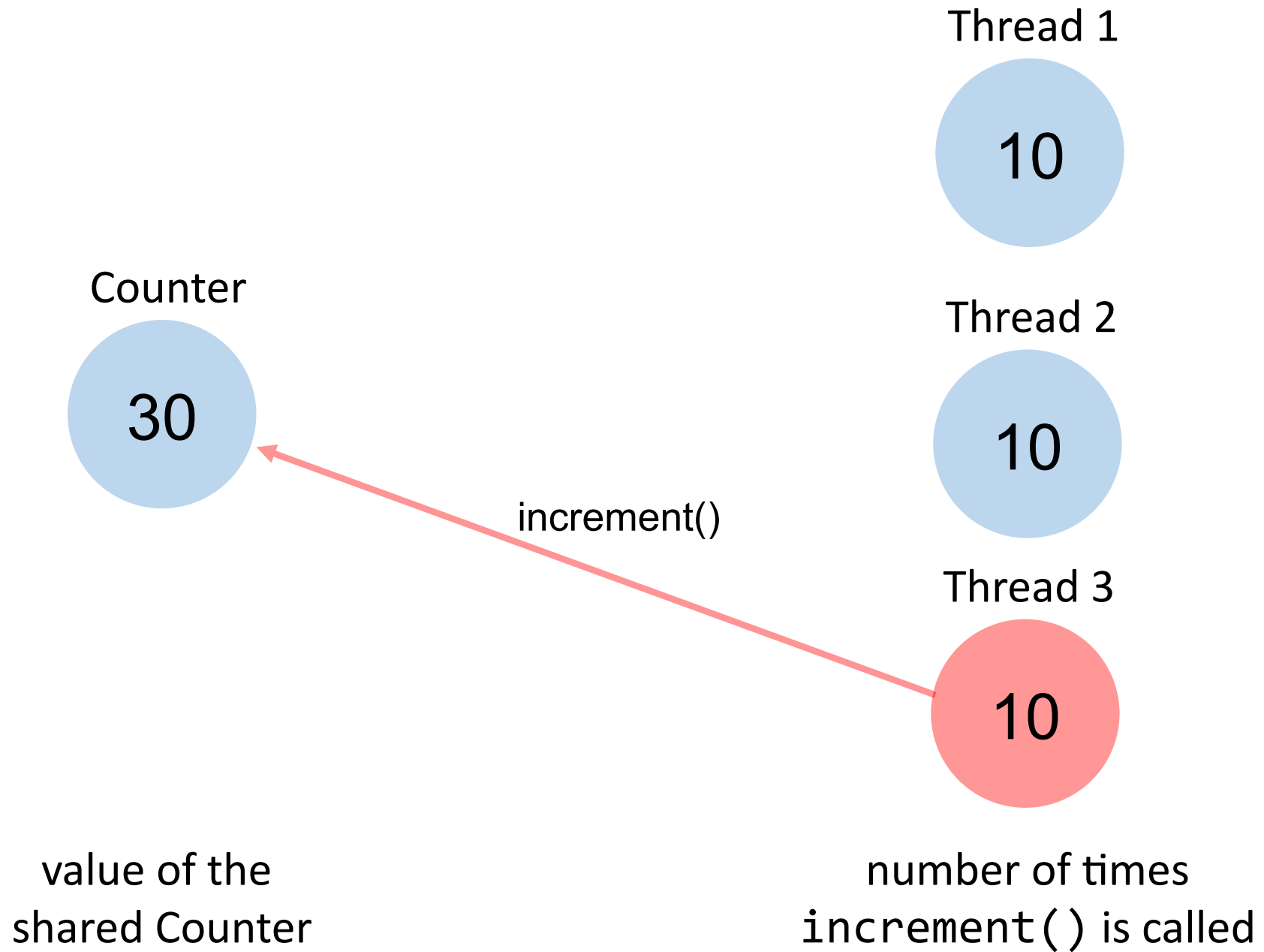
number of times
`increment()` is called

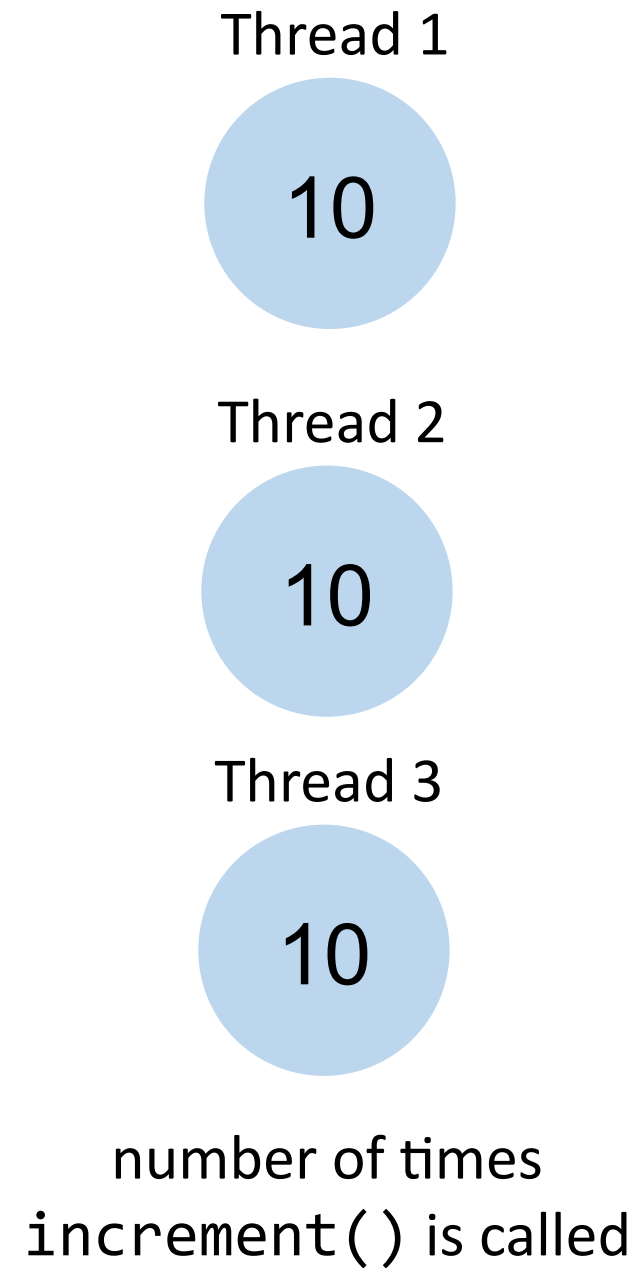
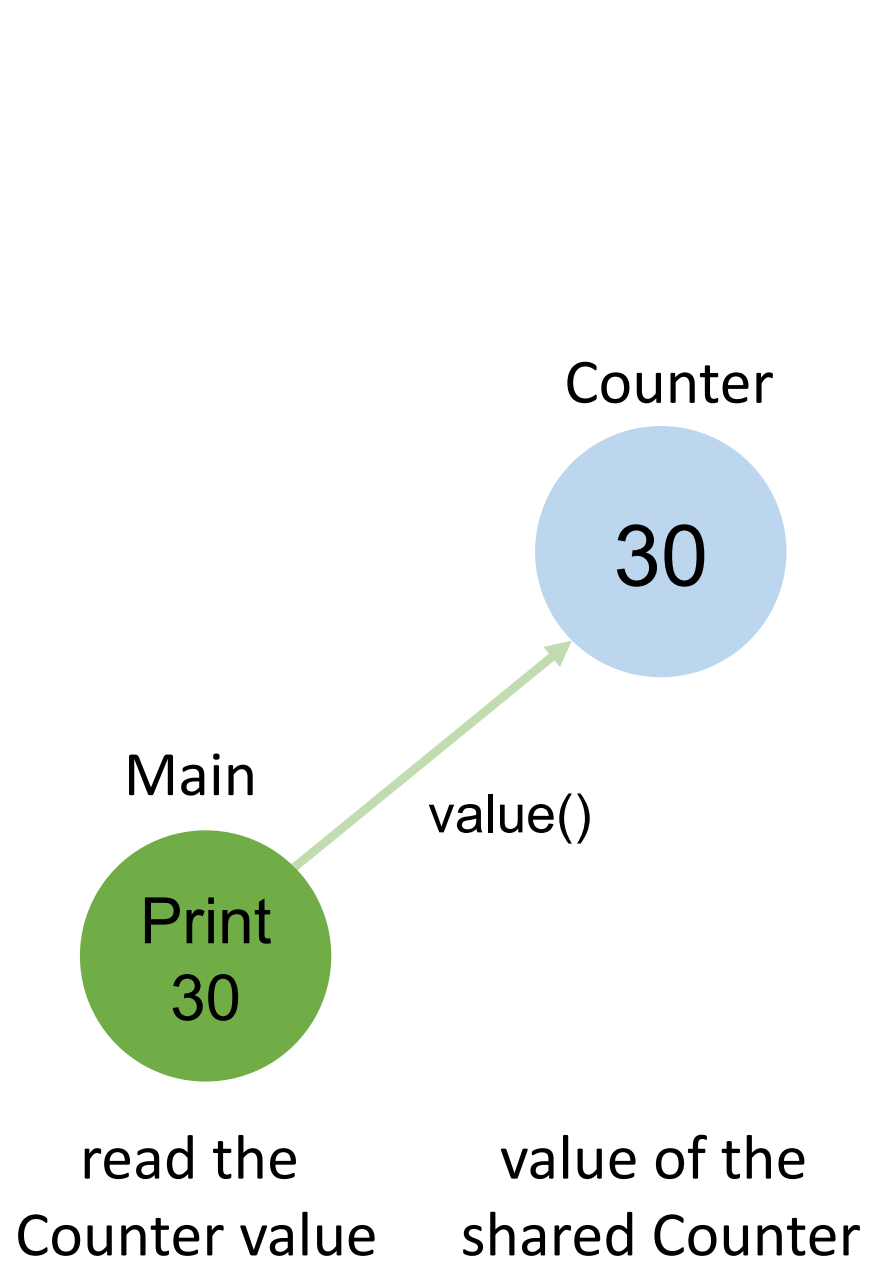












Counter

Warum wird das wahrscheinlich nicht funktionieren?

Lösung: Bad Interleavings

Counter

Mehrere Threads greifen gleichzeitig auf den Counter zu.
Wie wollen wir dieses Problem beheben?
Ihr werdet verschiedene Lösungen implementieren.

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

Task A – Sequential counter

- Implementiert eine sequentielle Version des Counters, die keine Synchronisierung benutzt.
- Die `taskASequential()` Methode benötigt eure sequentielle Version. Stellt sicher, dass ihr die Methode versteht.
- Verifiziert die Korrektheit in dem ihr die Tests laufen lässt. `testSequentialCounter` sollte bestehen.

Task A – Parallel counter

- Führt `taskAParallel()` aus. Diese Methode erstellt mehrere Threads, die auf euren sequentiellen Counter zugreifen.
- Wird diese Methode korrekt sein und warum ja/nein?

Task B – Synchronized counter

- Implementiert eine Thread-sichere (thread-safe) Version des Counters. Benutzt dafür **synchronized**.
- Führt den Code von taskB() aus.

Synchronization

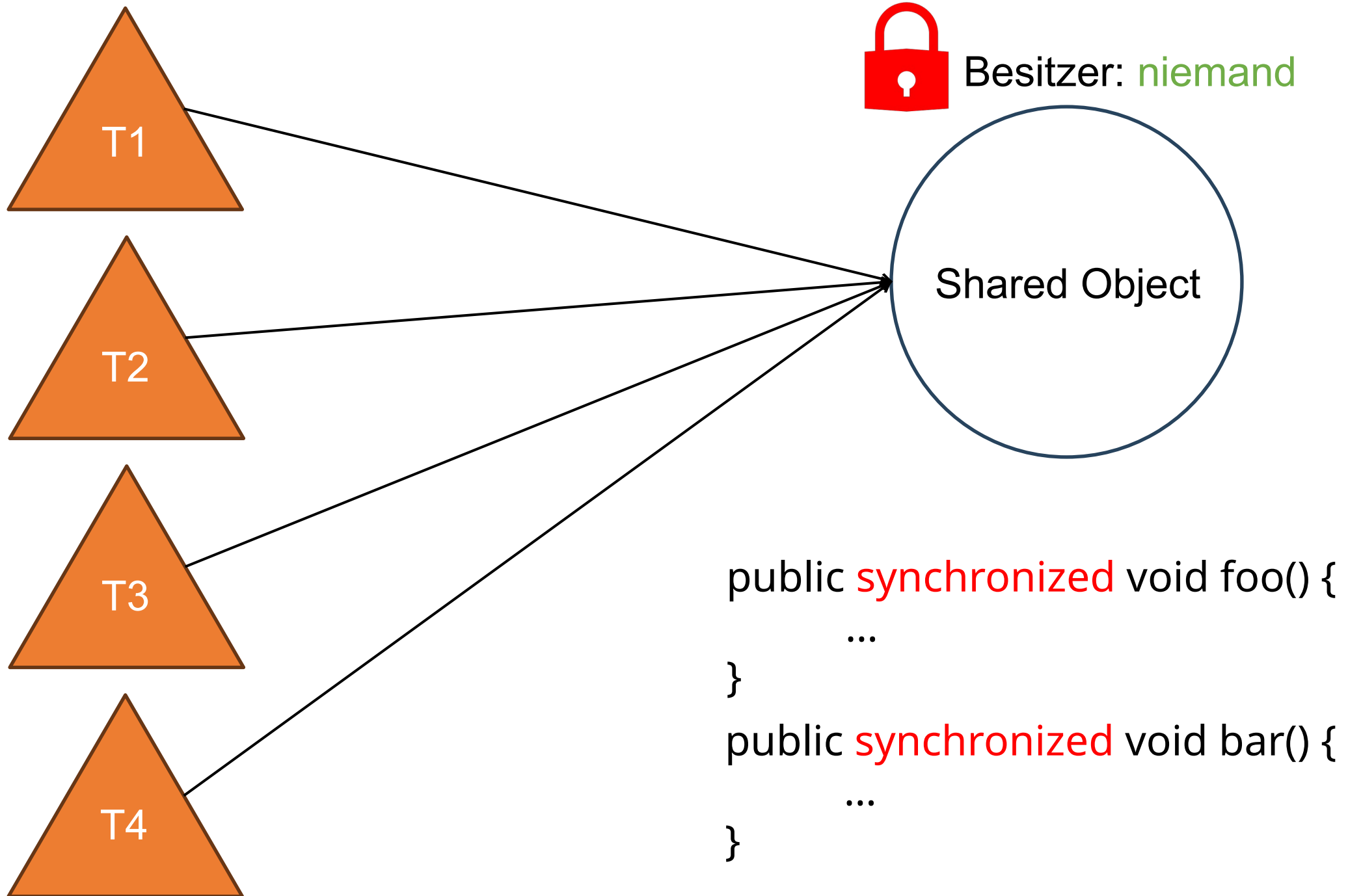
- Ziel: Bad Interleavings verhindern
- Jedes Java Object hat ein Schloss (Lock/Monitor), dass von der Object class vererbt wird.
- Ein Lock wird beim eintreten in einen synchronized Block automatisch erworben und am Schluss auch wieder released.

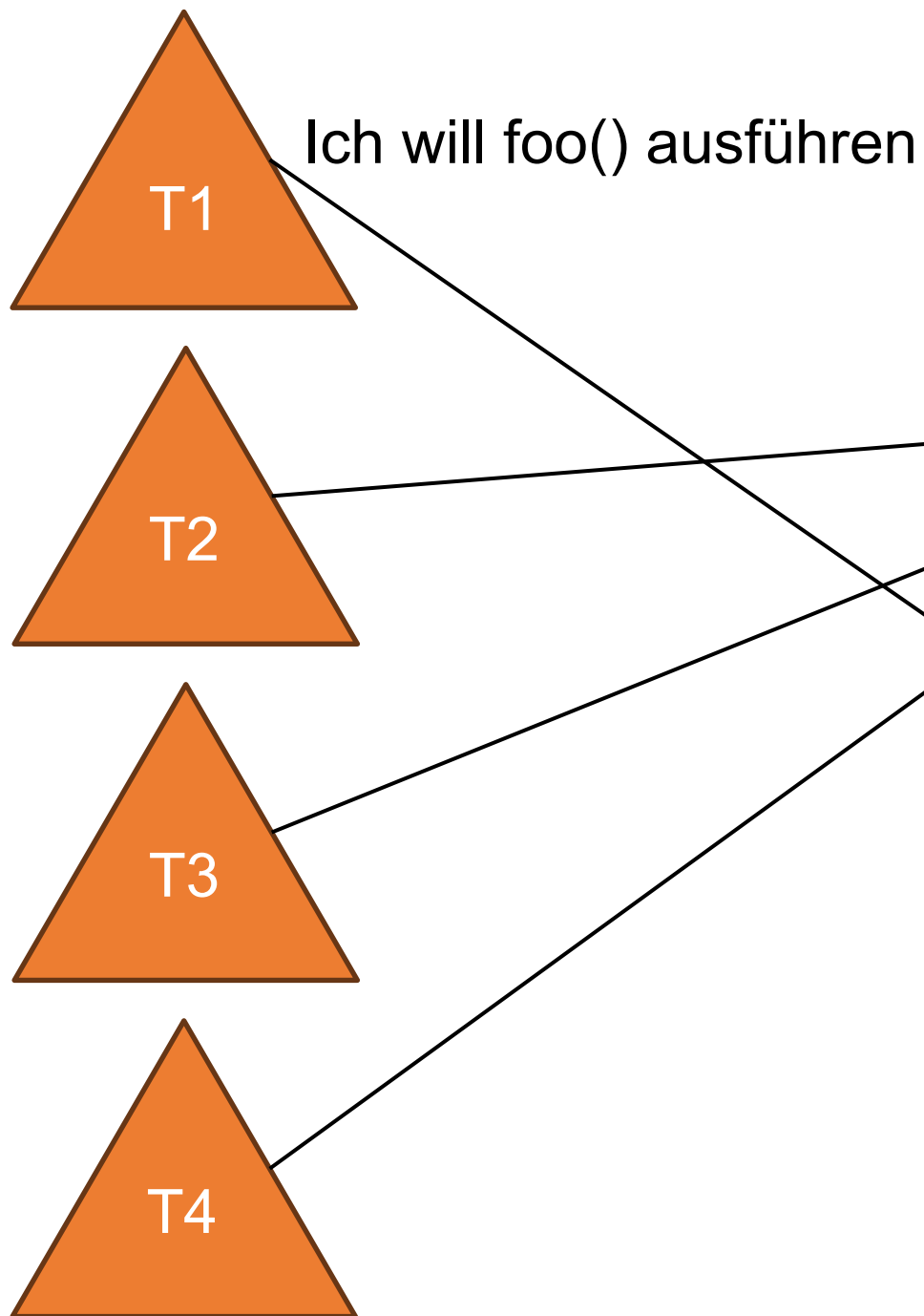


Besitzer: **niemand**

Shared Object

```
public synchronized void foo() {  
    ...  
}  
public synchronized void bar() {  
    ...  
}
```

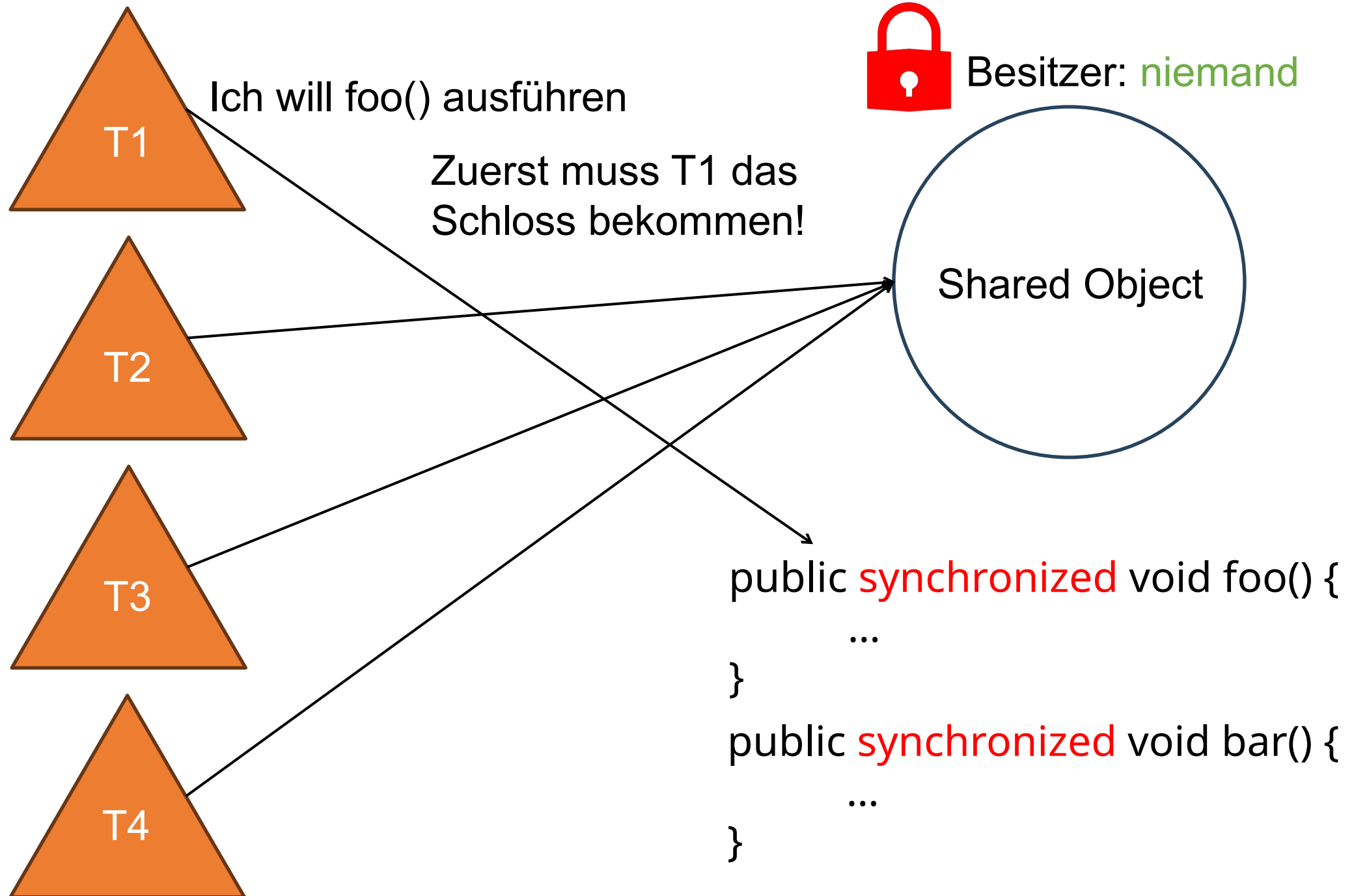



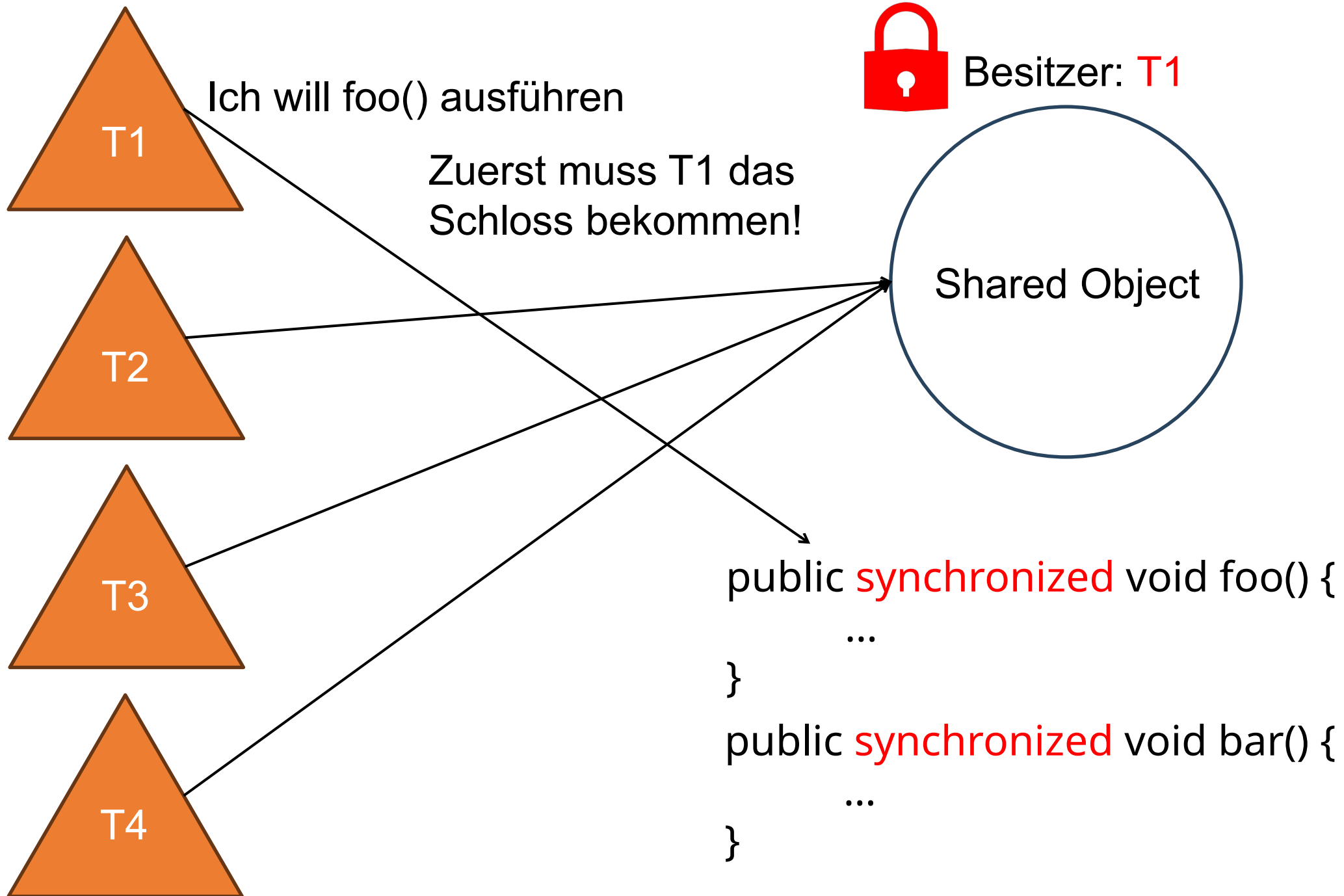


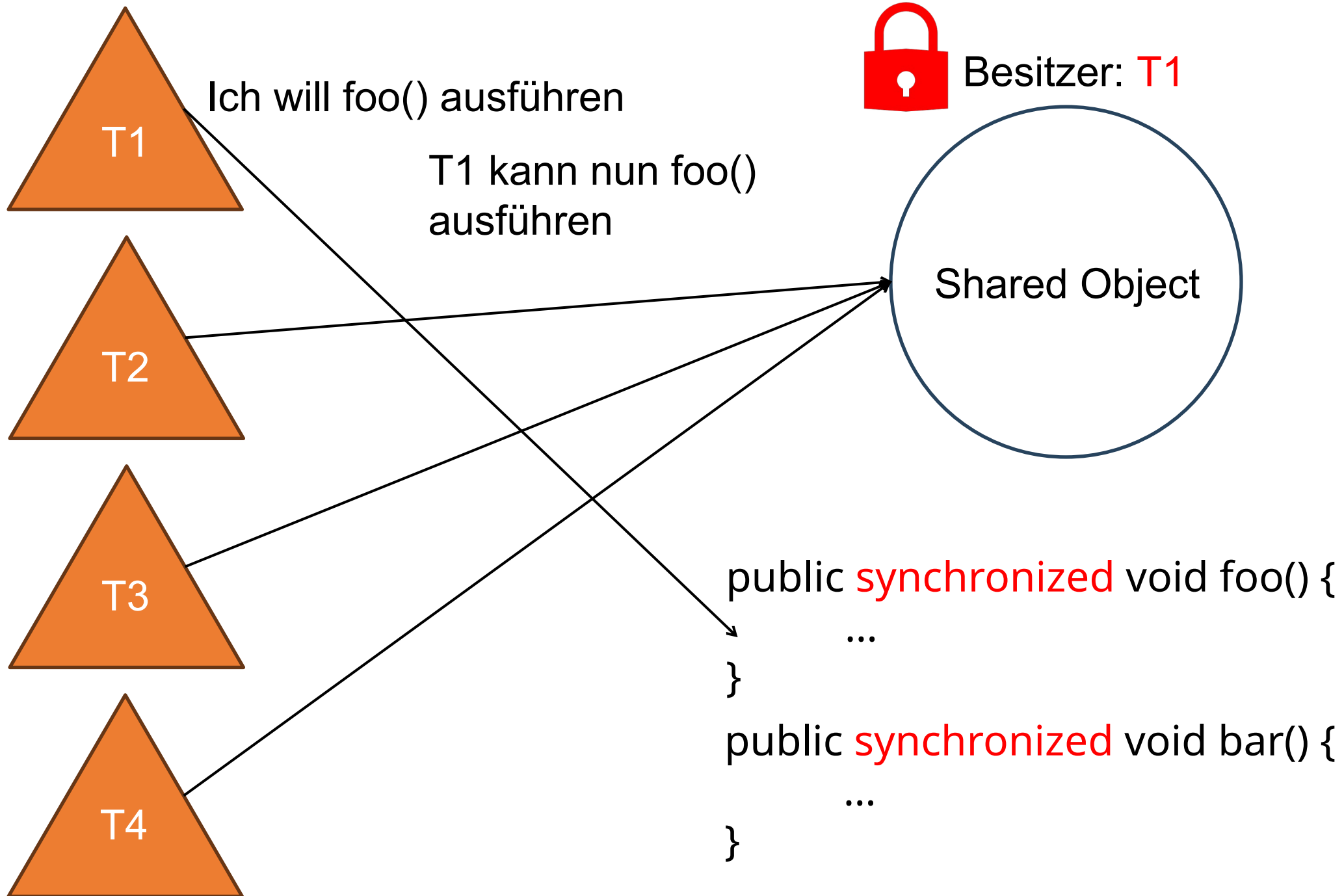
Besitzer: niemand

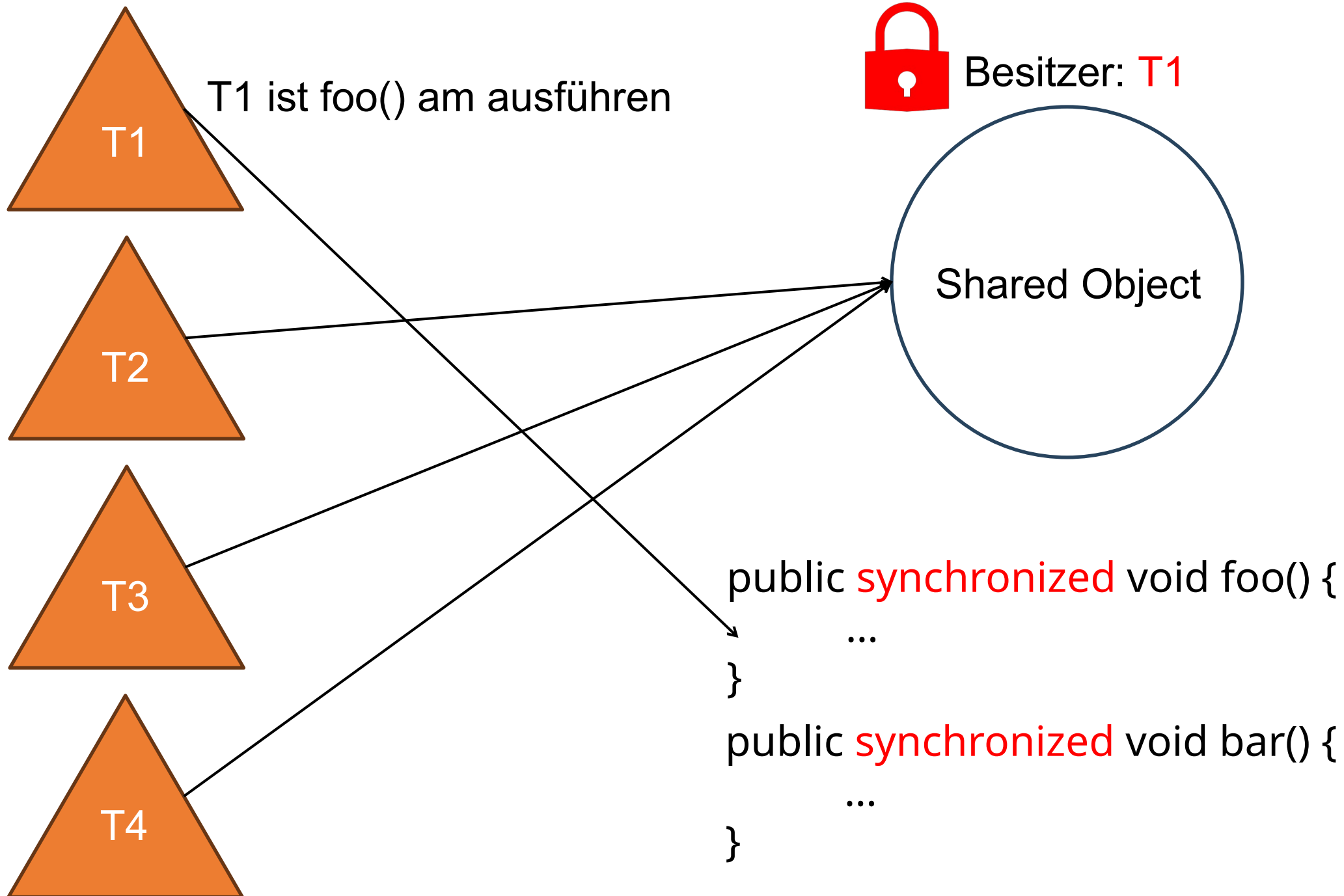
Shared Object

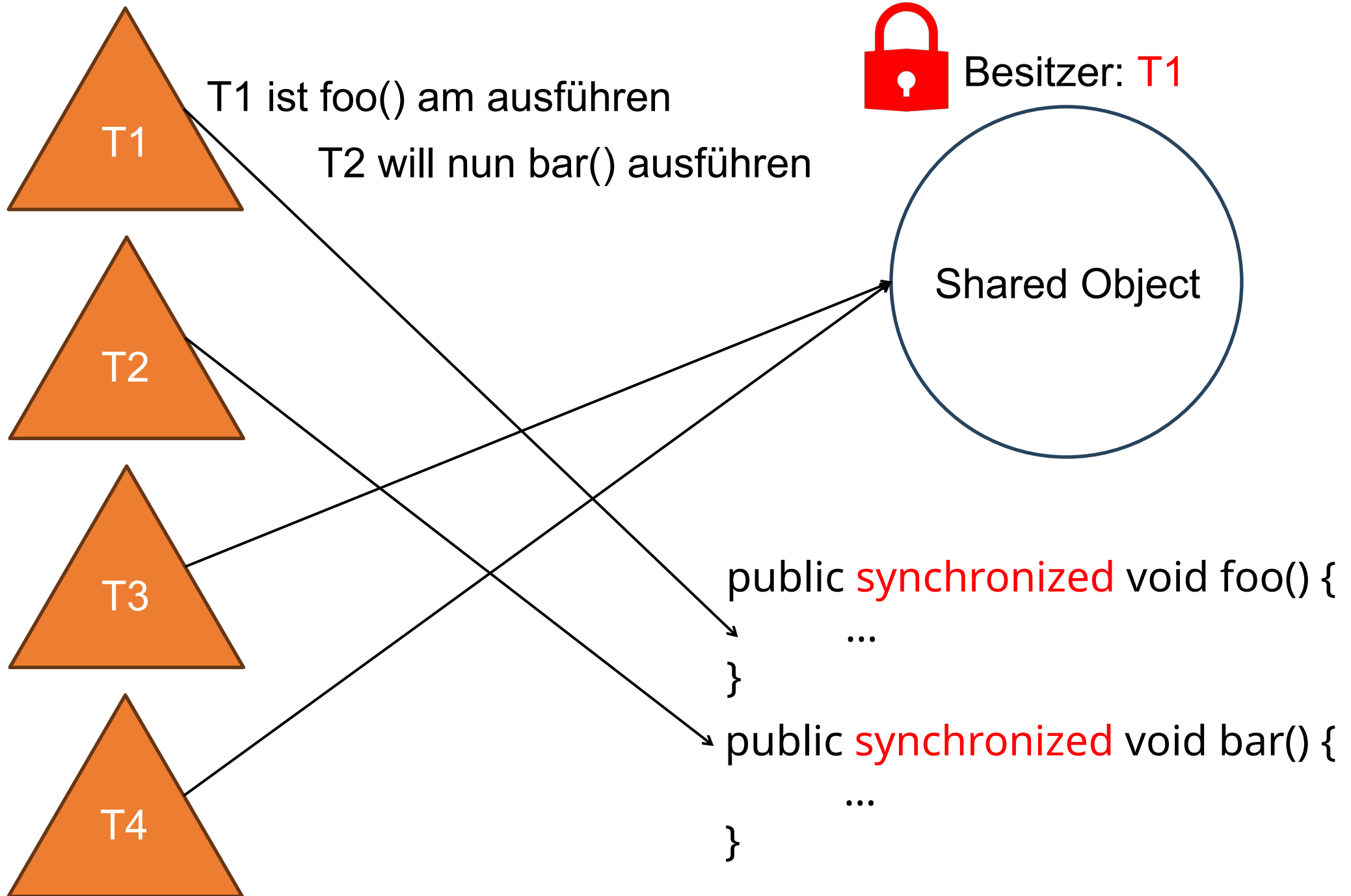
```
public synchronized void foo() {  
    ...  
}  
public synchronized void bar() {  
    ...  
}
```

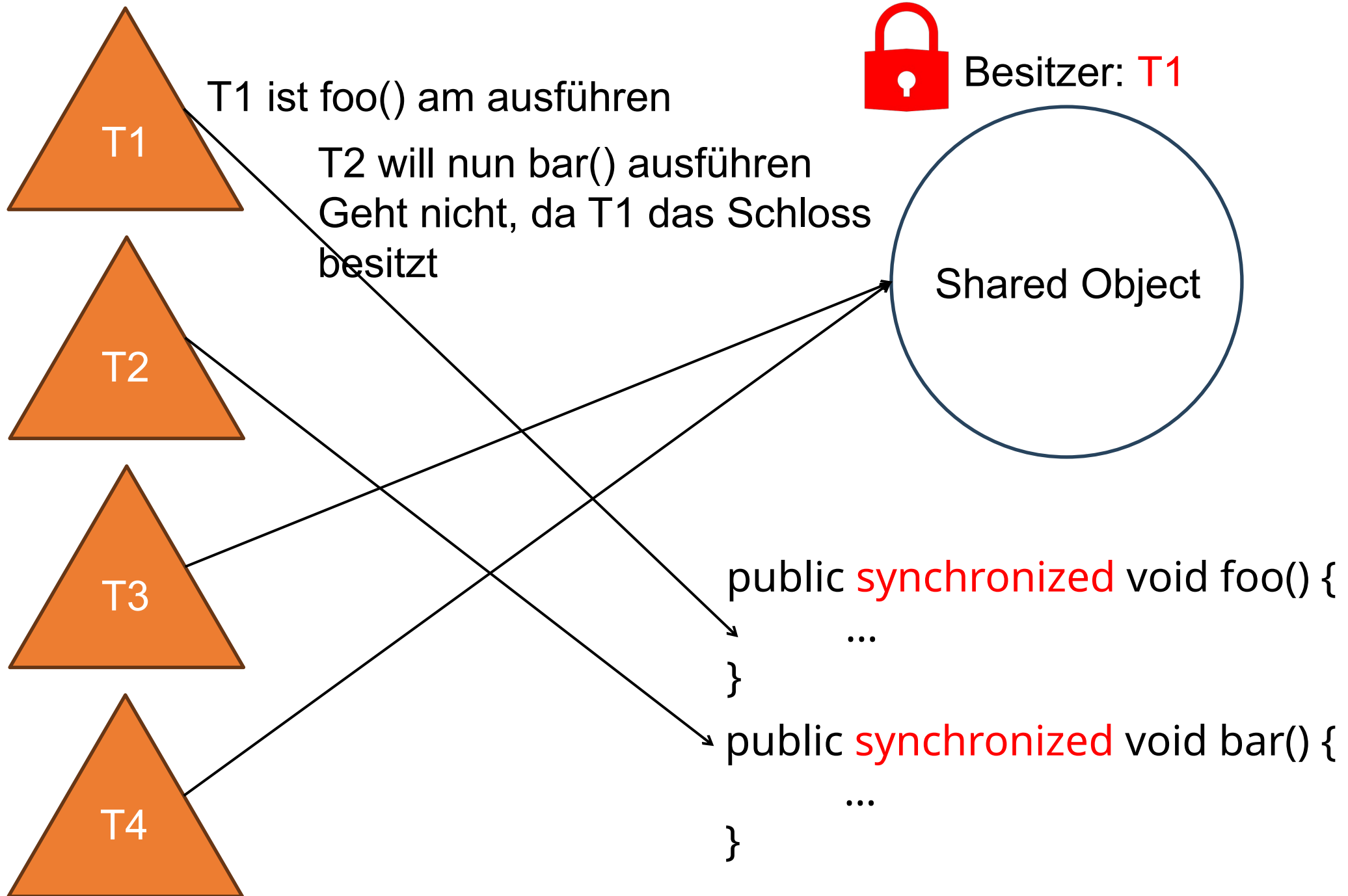


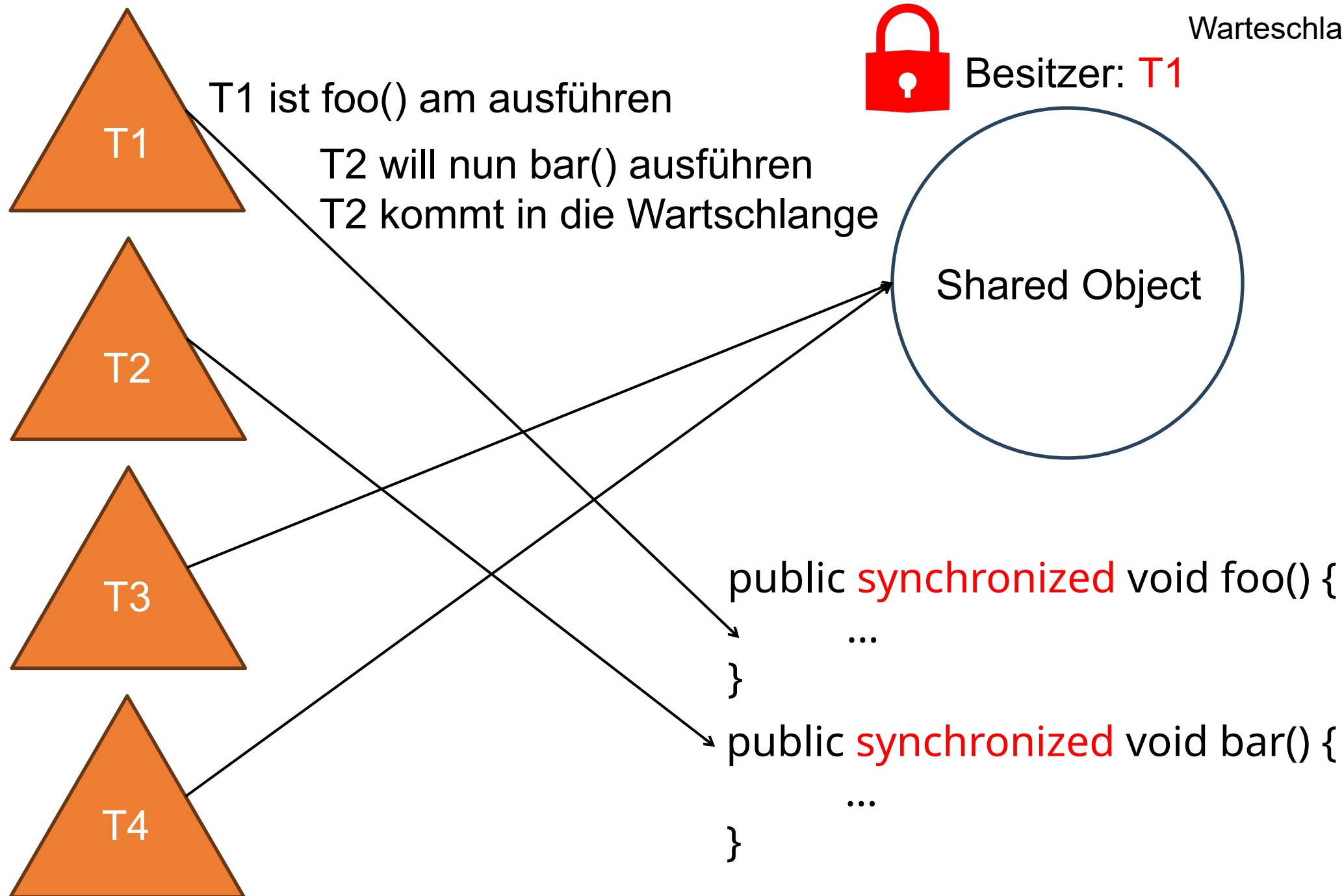


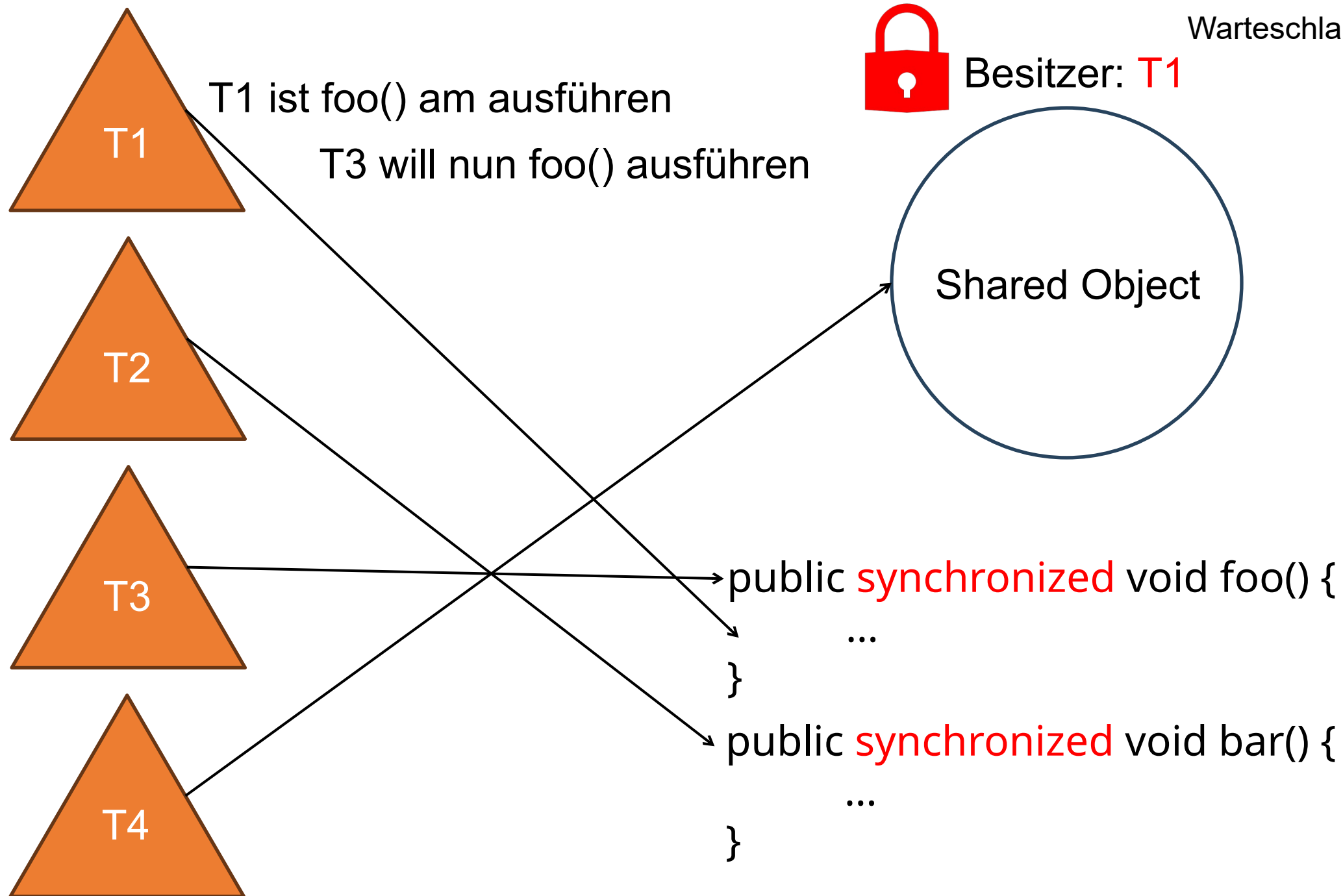


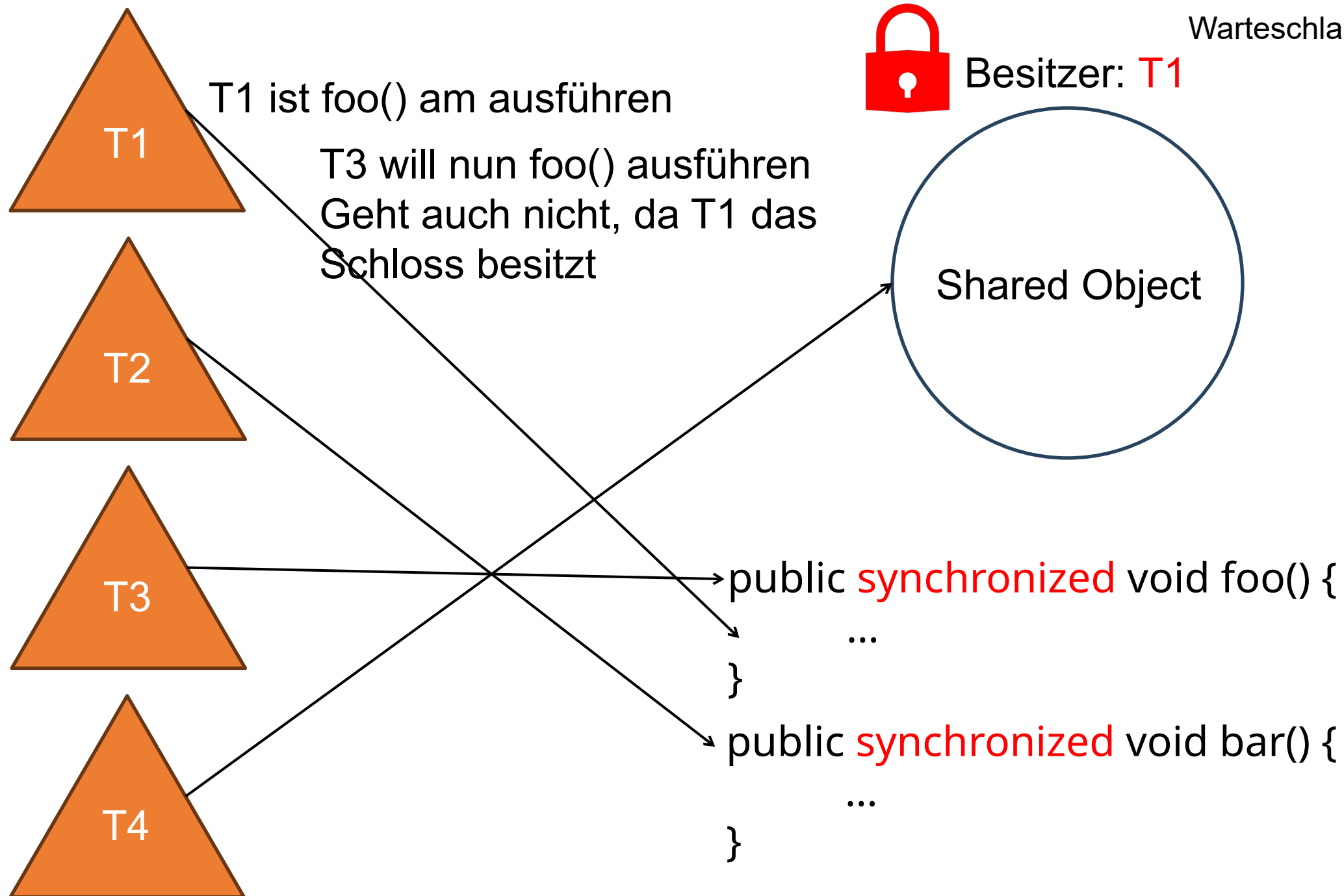


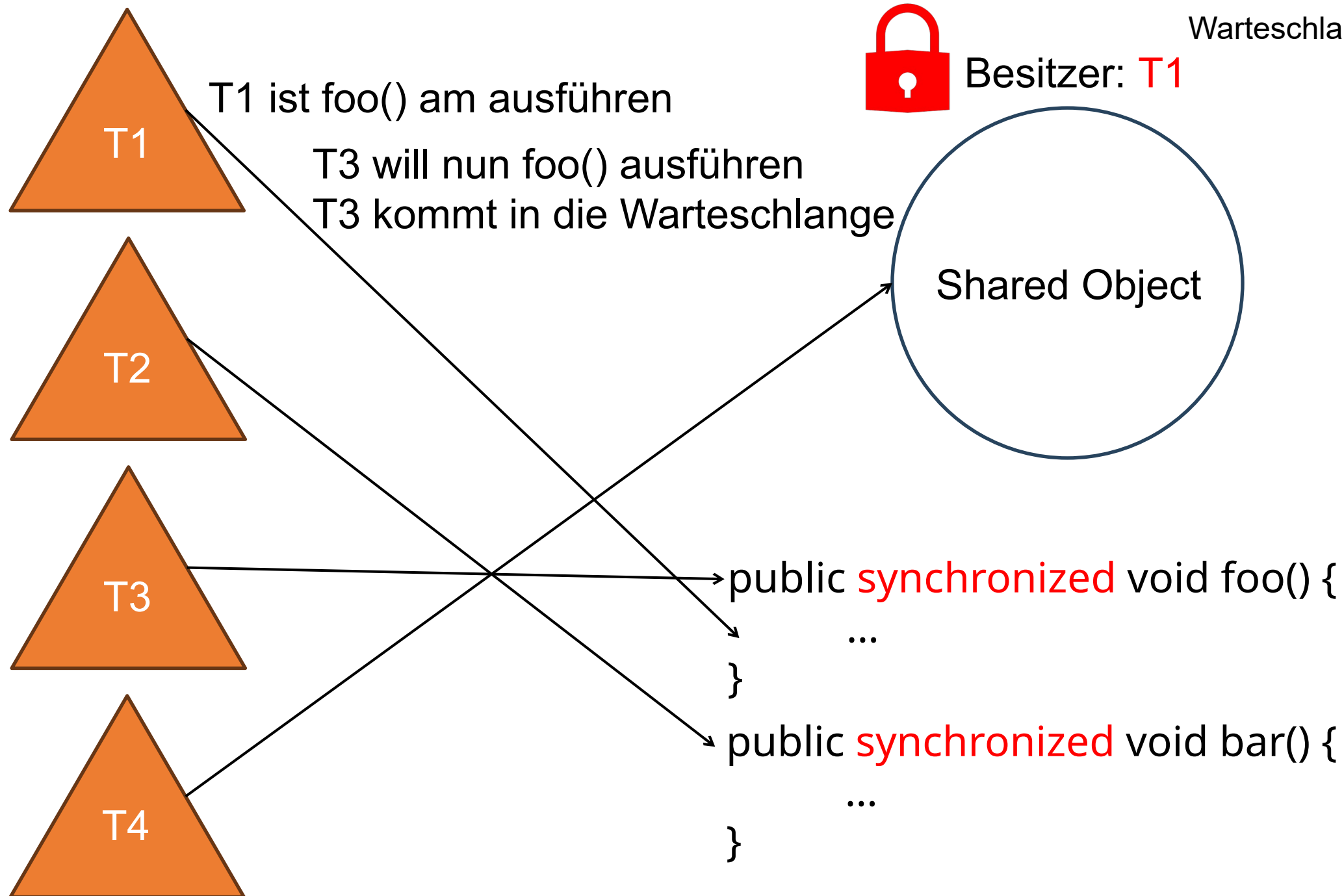


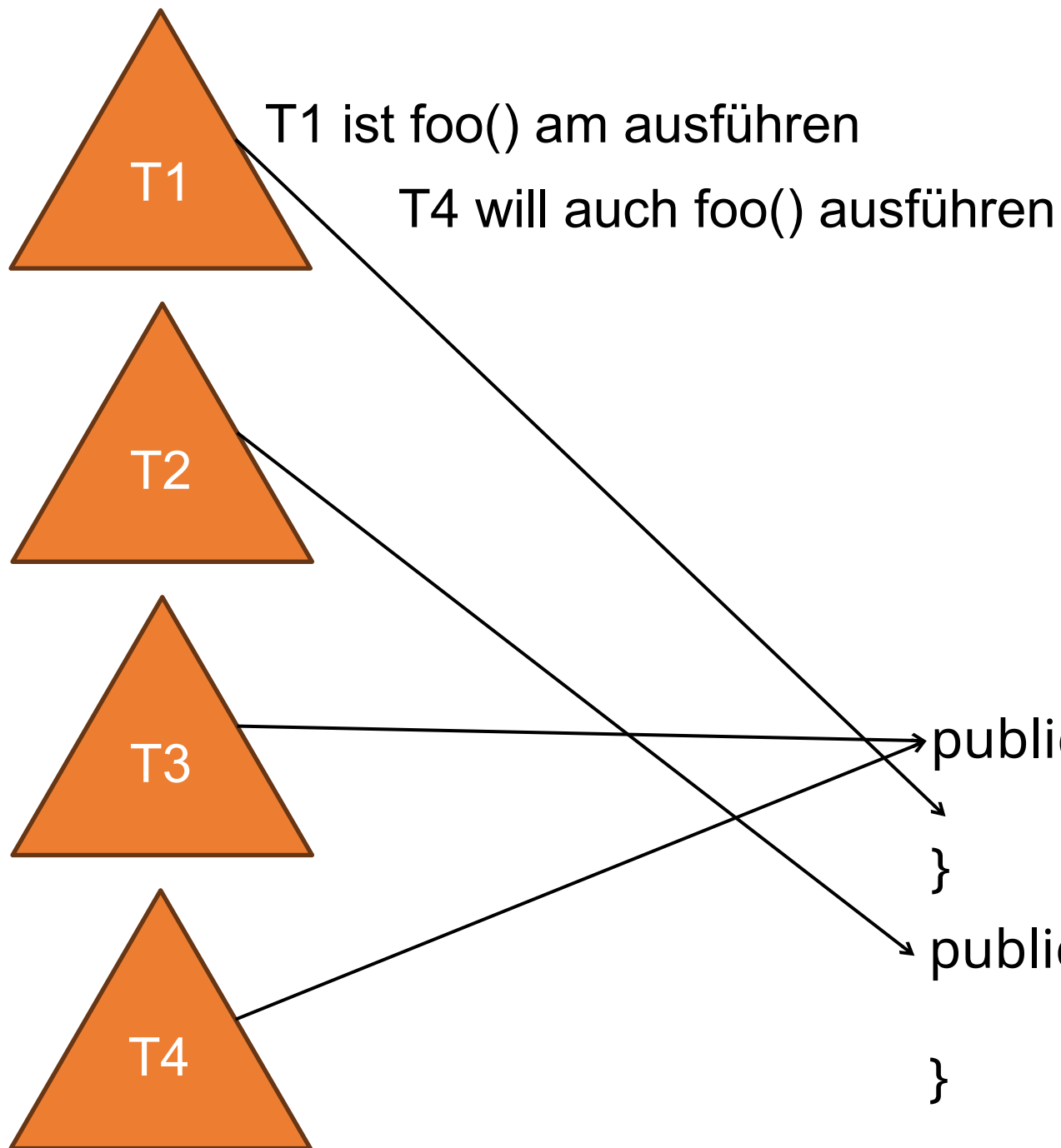








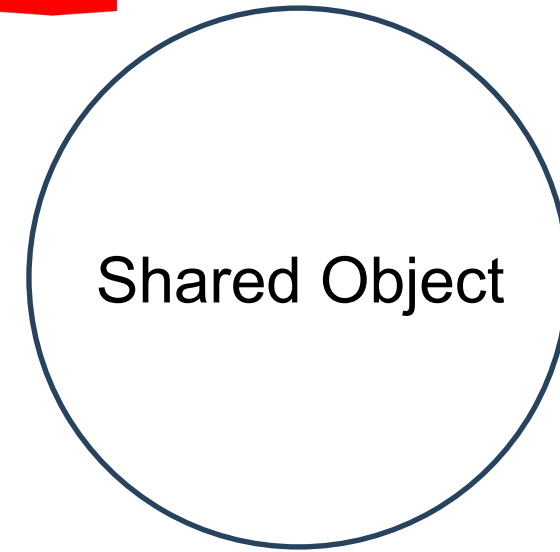




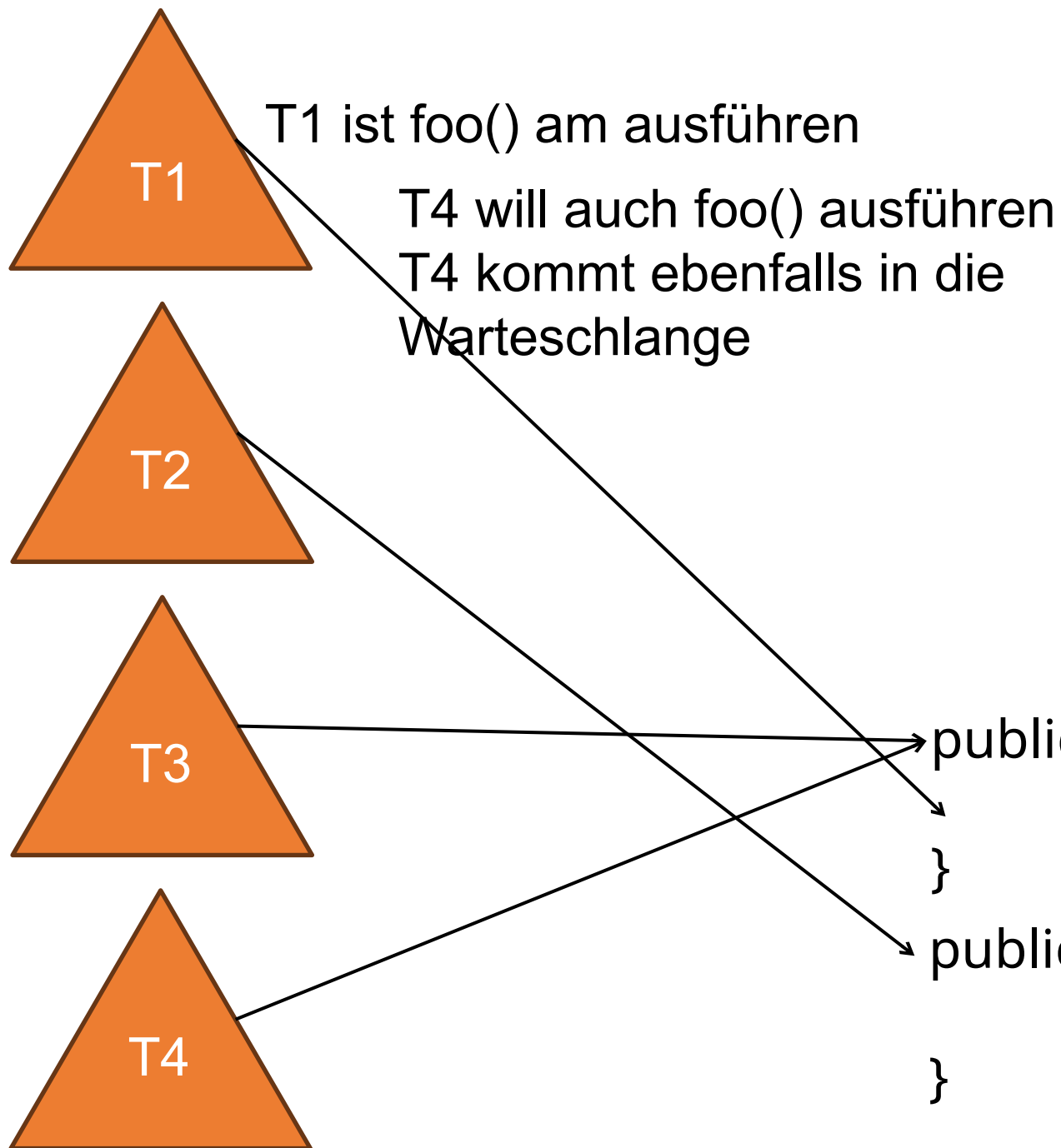
Besitzer: **T1**

Warteschlange für 

- T2
- T3



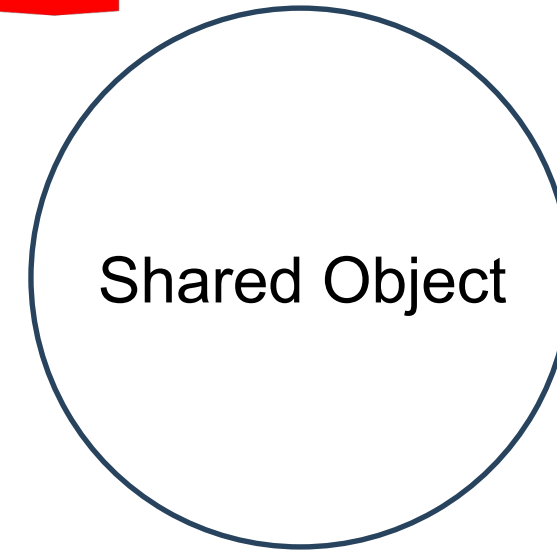
```
public synchronized void foo() {  
    ...  
}  
public synchronized void bar() {  
    ...  
}
```



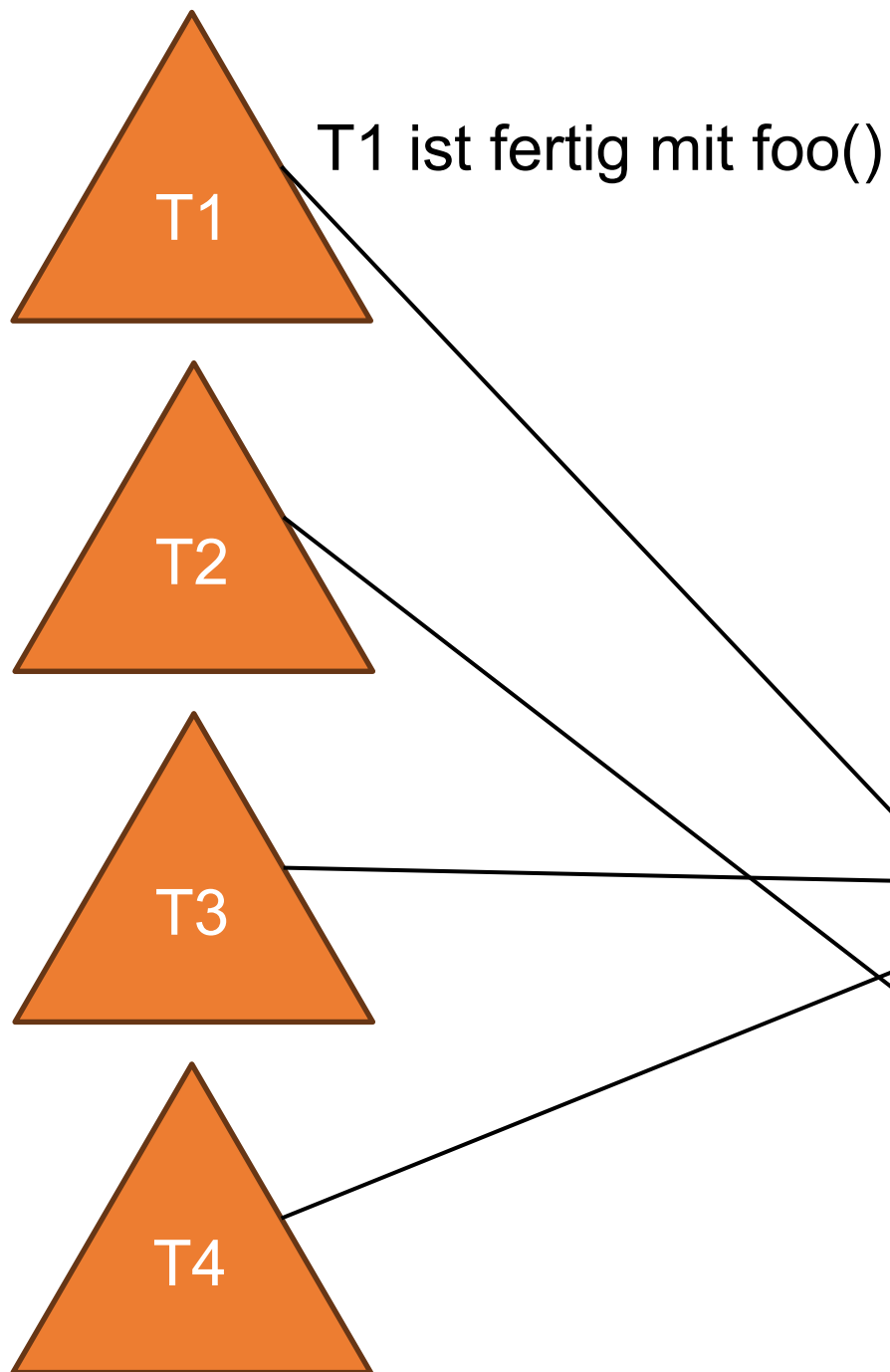
Besitzer: **T1**

Warteschlange für 

- T2
- T3
- T4



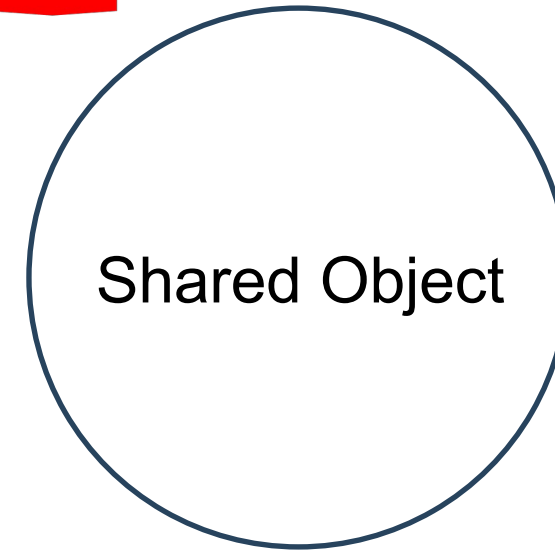
```
public synchronized void foo() {  
    ...  
}  
public synchronized void bar() {  
    ...  
}
```



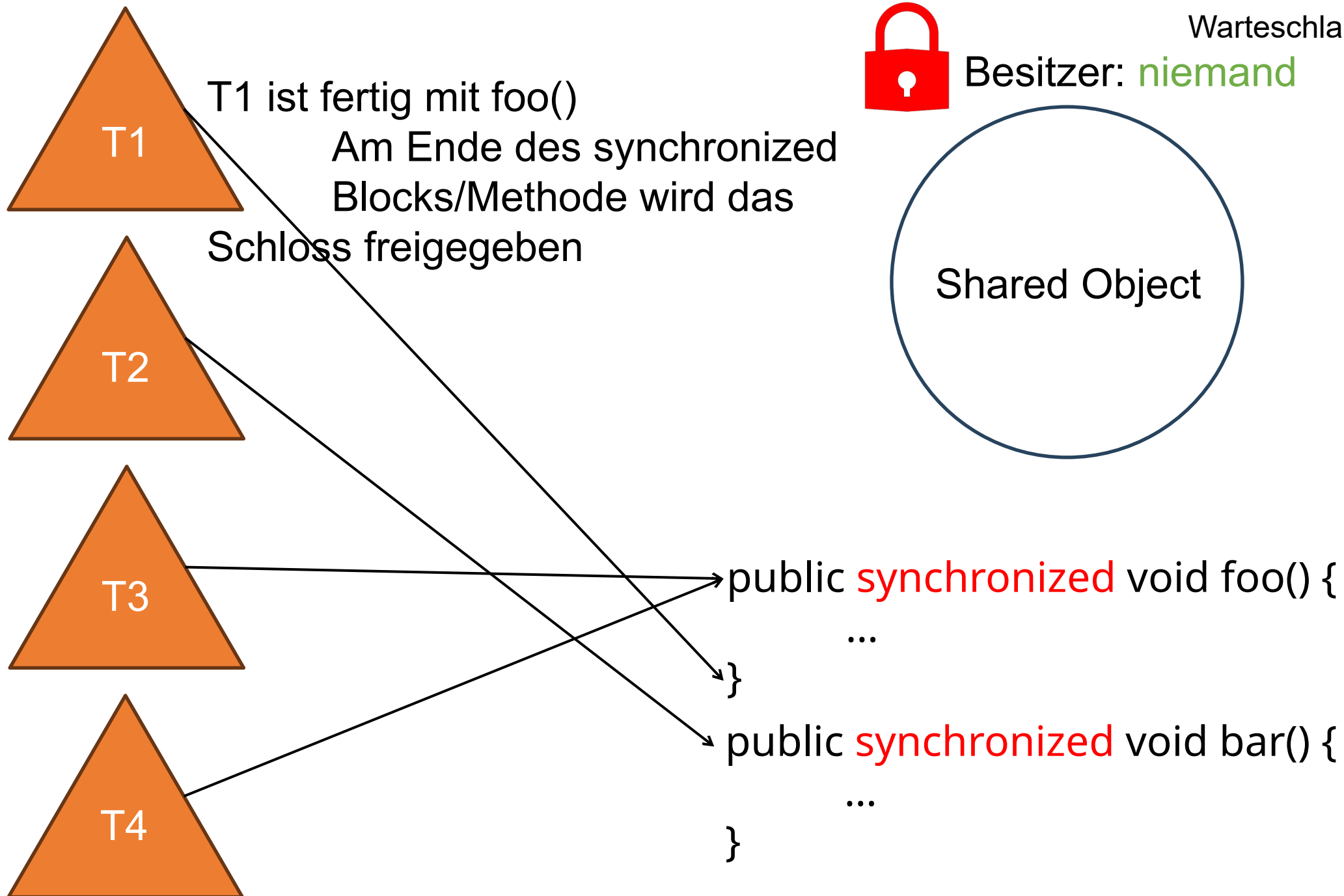
Besitzer: T1

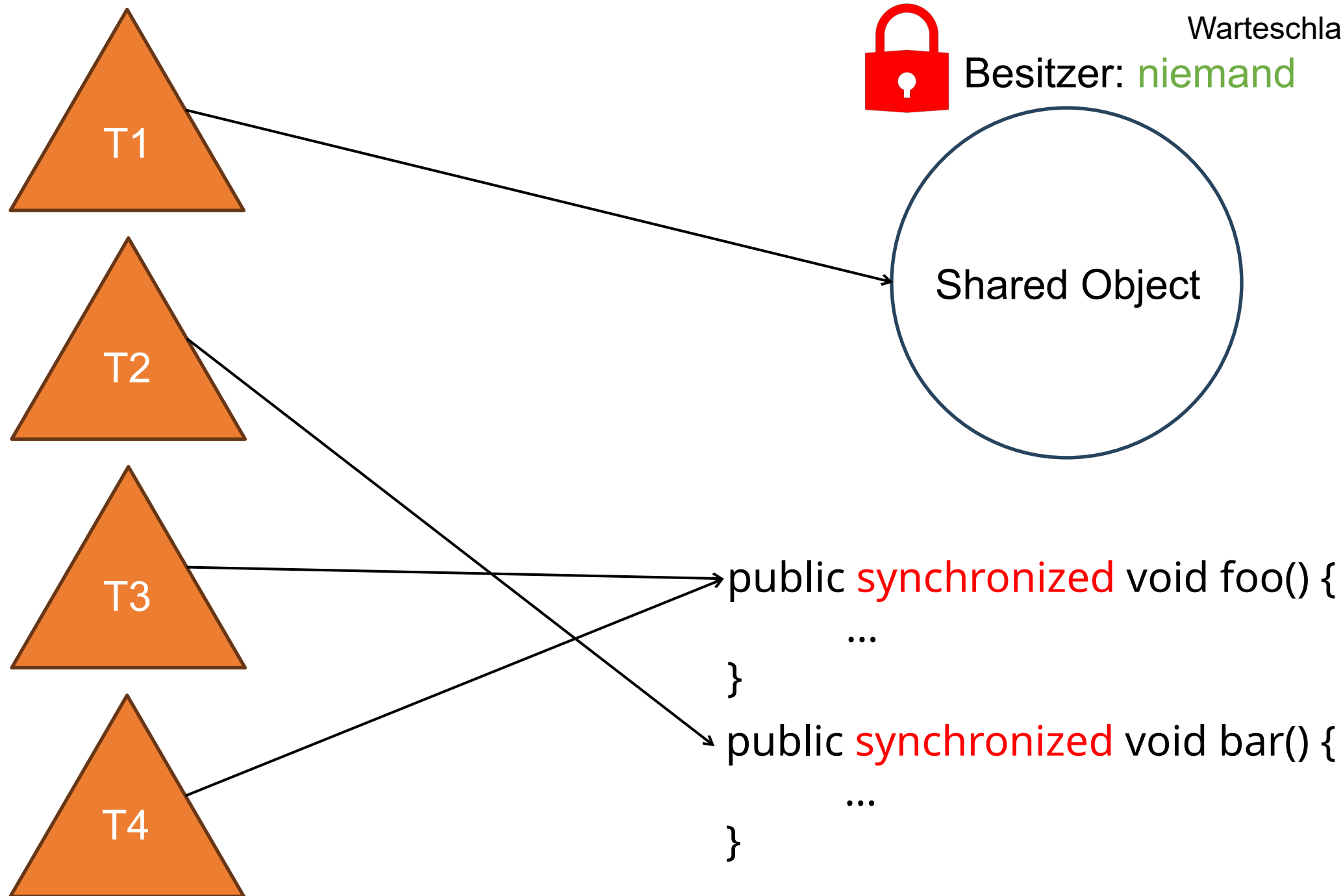
Warteschlange für 

- T2
- T3
- T4



```
public synchronized void foo() {  
    ...  
}  
public synchronized void bar() {  
    ...  
}
```

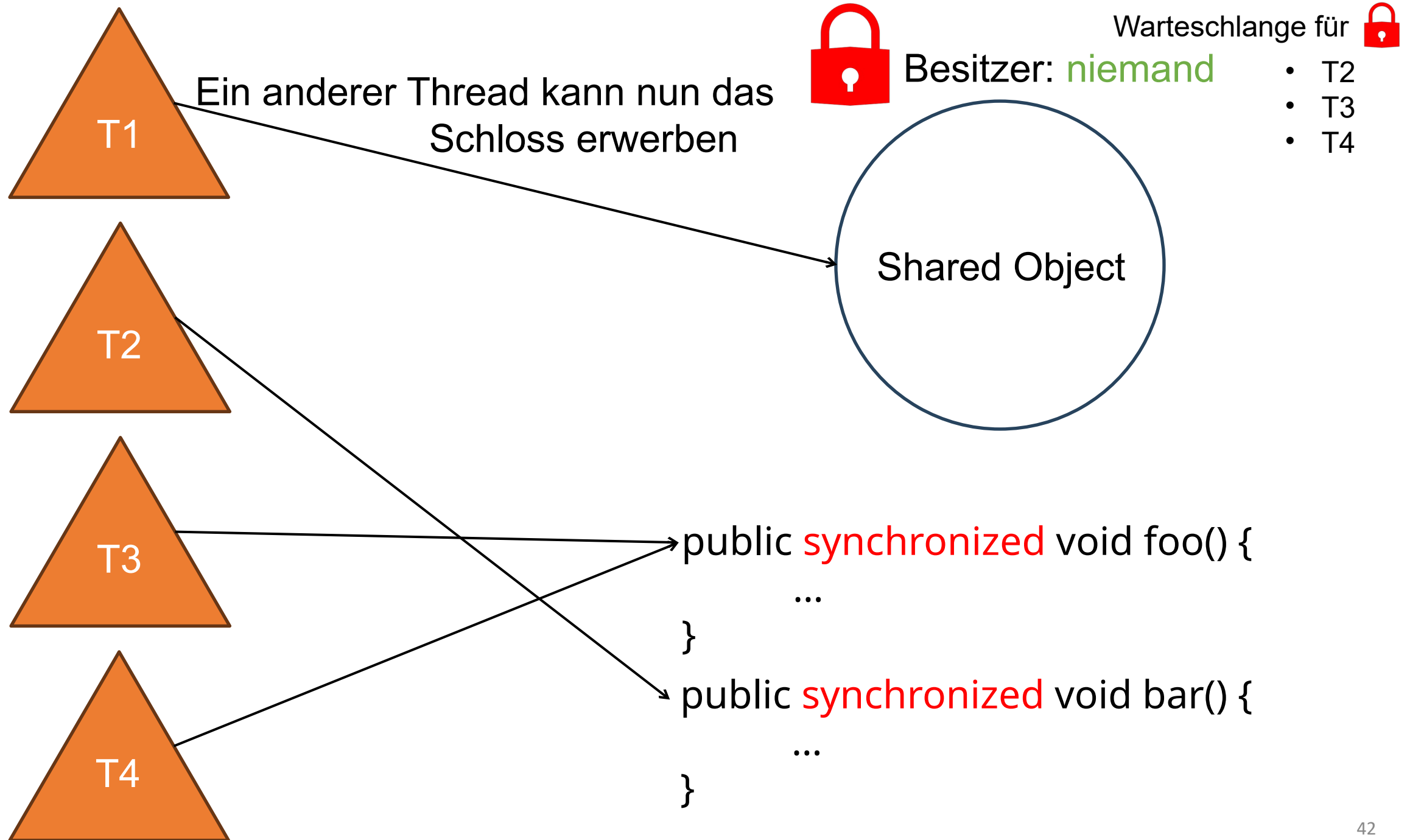


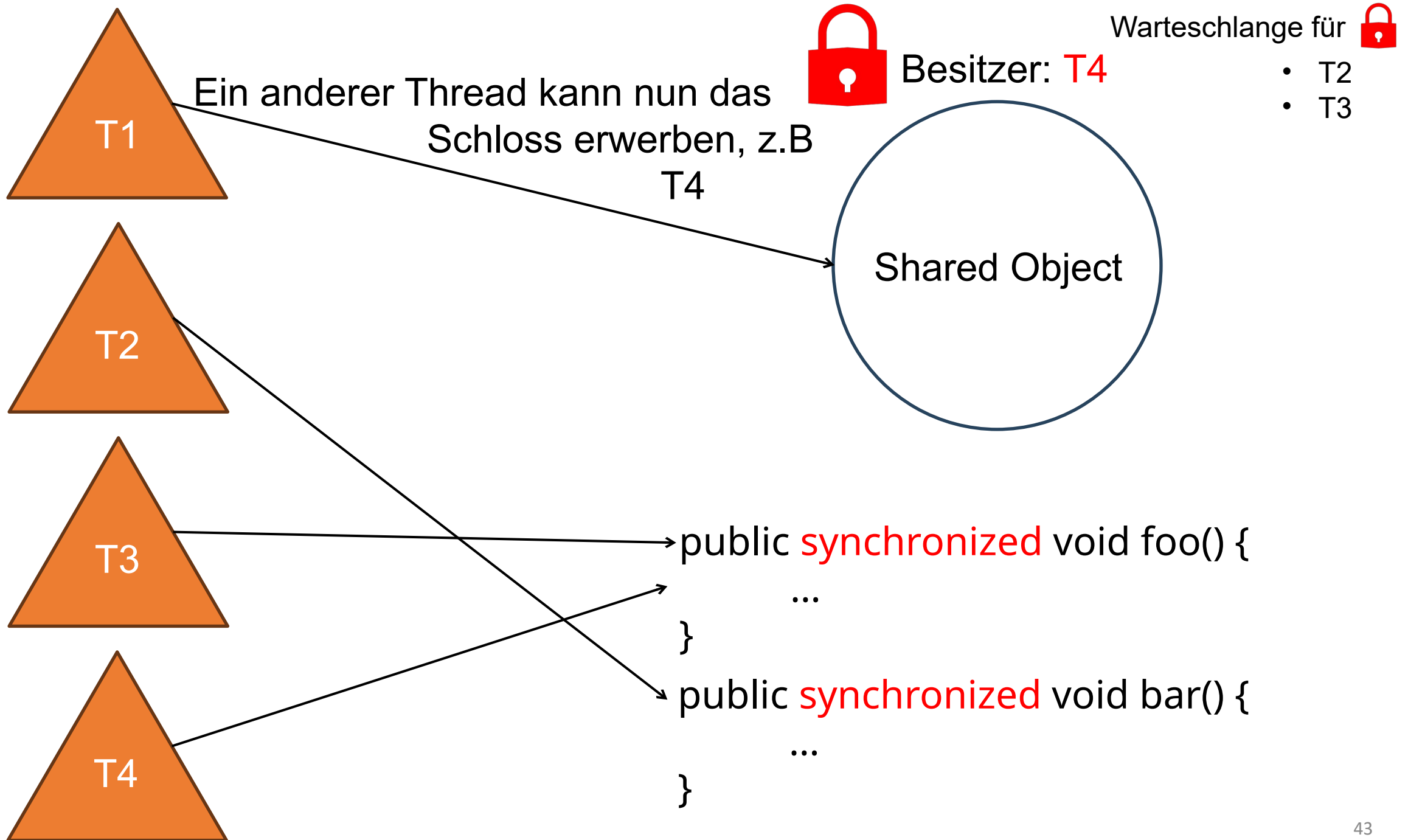


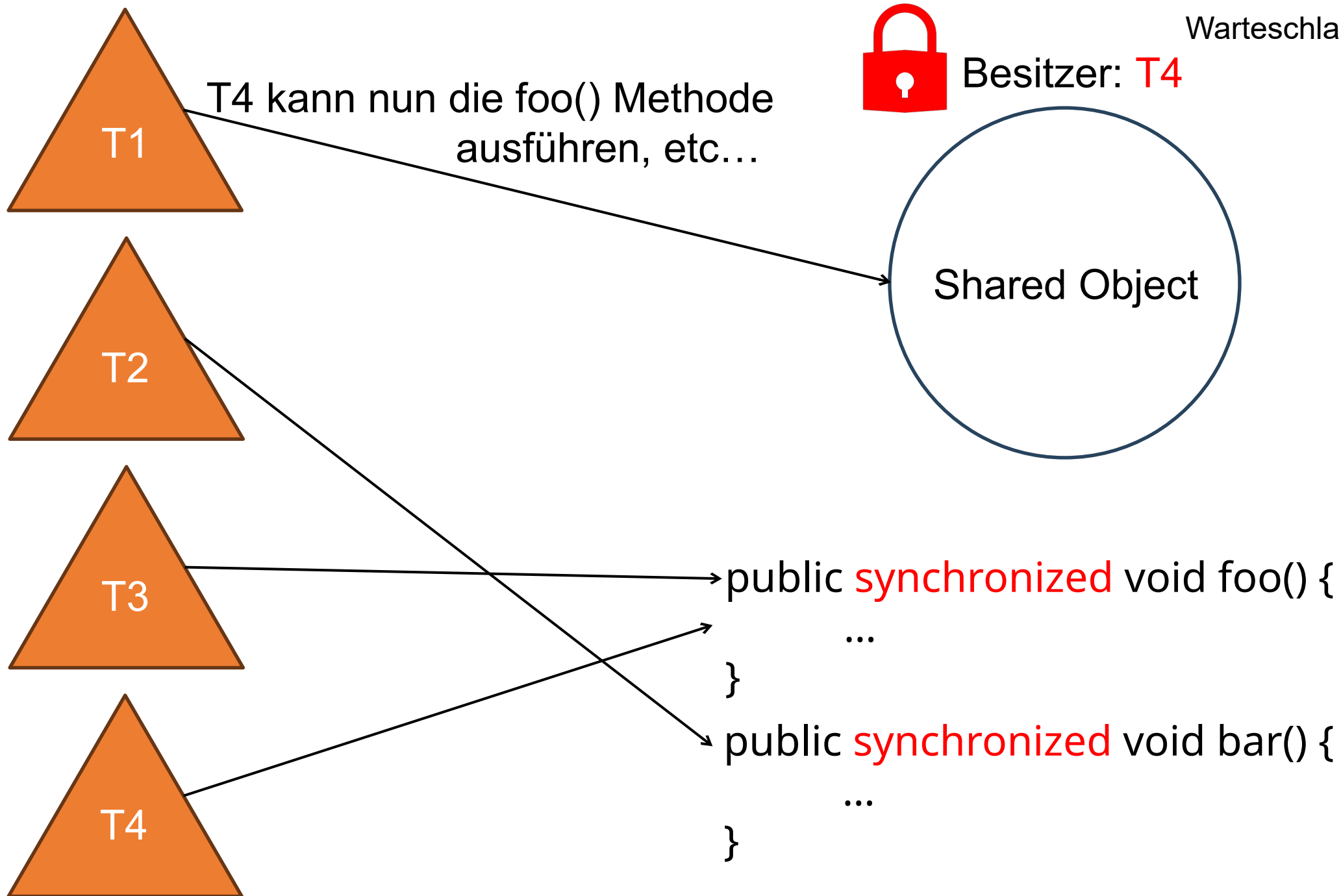
Besitzer: **niemand**

Warteschlange für 

- T2
- T3
- T4







Synchronization: Wichtig

Threads synchronisieren immer in Bezug auf ein Objekt!

D.h. Es werden keine Threads gelocked, sondern Objekte werden von den Threads gelocked!

Synchronization: Zusatz

- Primitive types haben kein Lock, dafür aber ihre Wrapper-Classes
- Das Locken eines Arrays locked nicht die einzelnen Elemente
- ***Locks werden wir später in der Vorlesung genauer analysieren.***

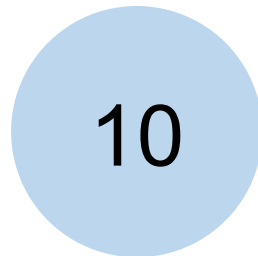
Synchronization

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

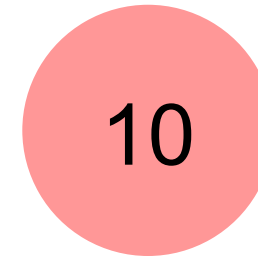
```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

- Synchronized Methode locked das “this” Objekt auf dem die Methode aufgerufen wird.
- Synchronized Block erwirbt das Lock des Objekts in Klammern.
- Ein Thread kann mehrere Locks gleichzeitig erwerben (z.B. durch verschachteln von synchronized Blocks).

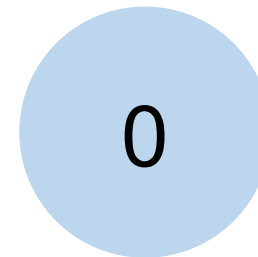
Counter



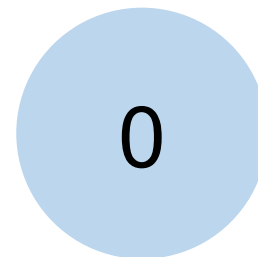
Thread 1

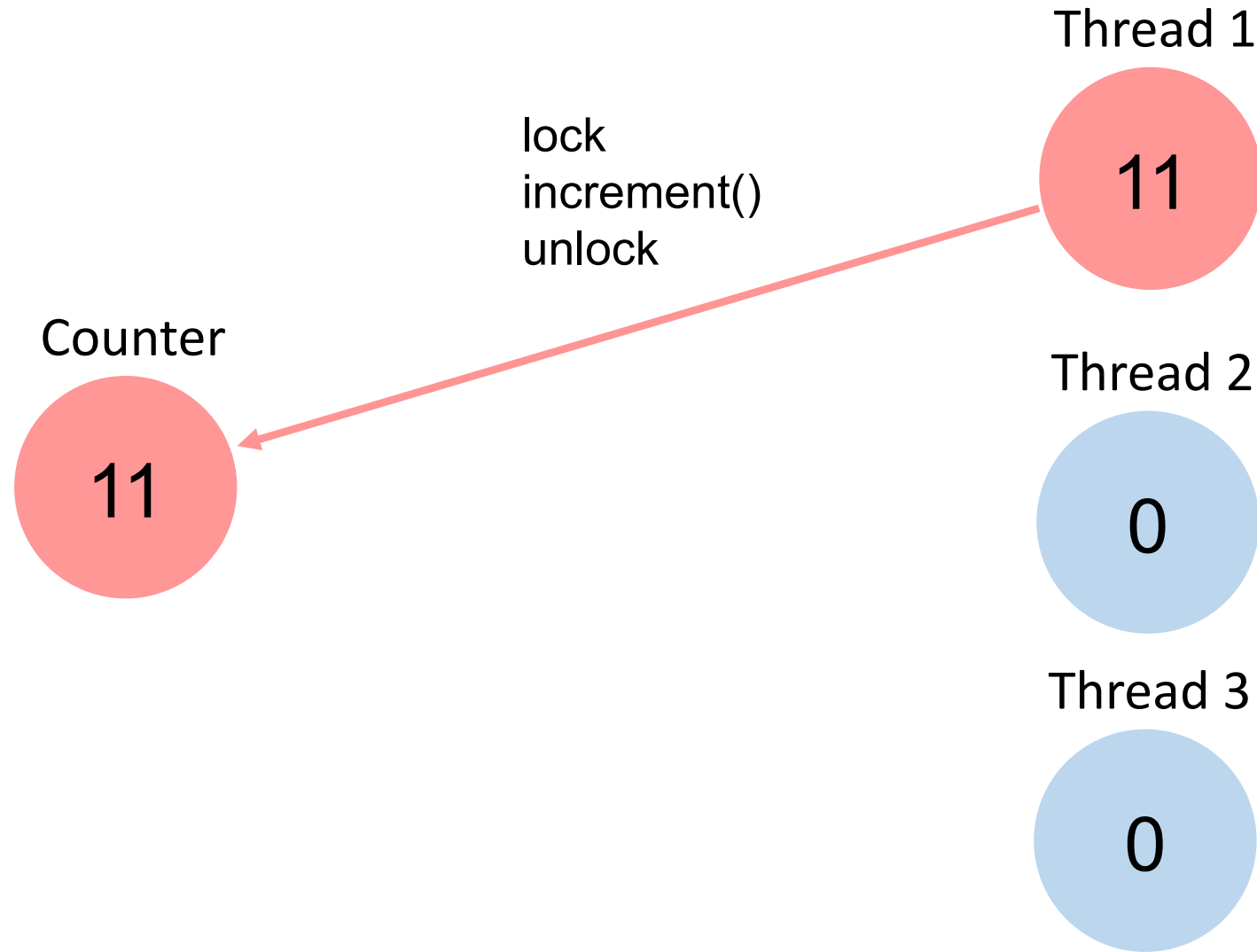


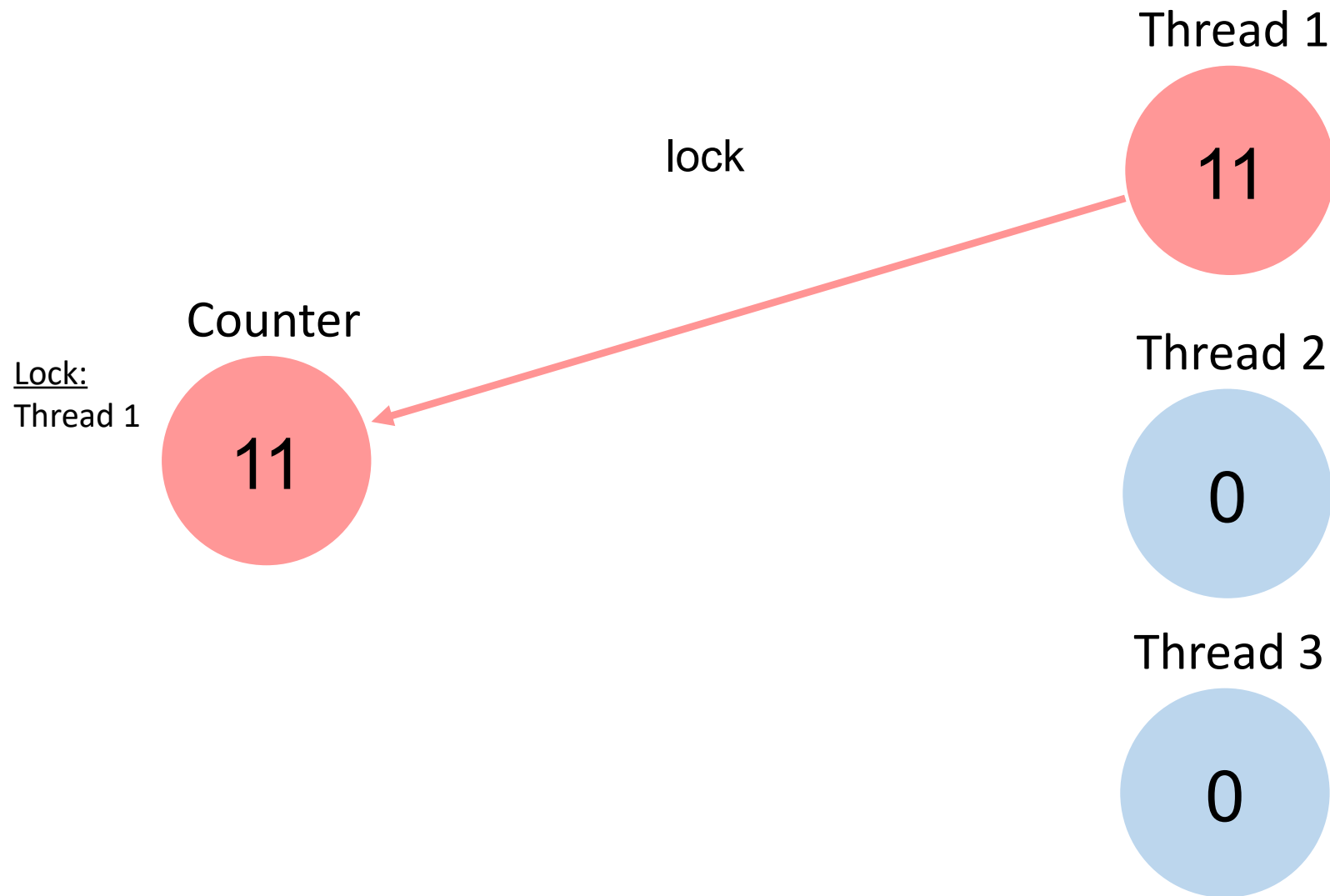
Thread 2

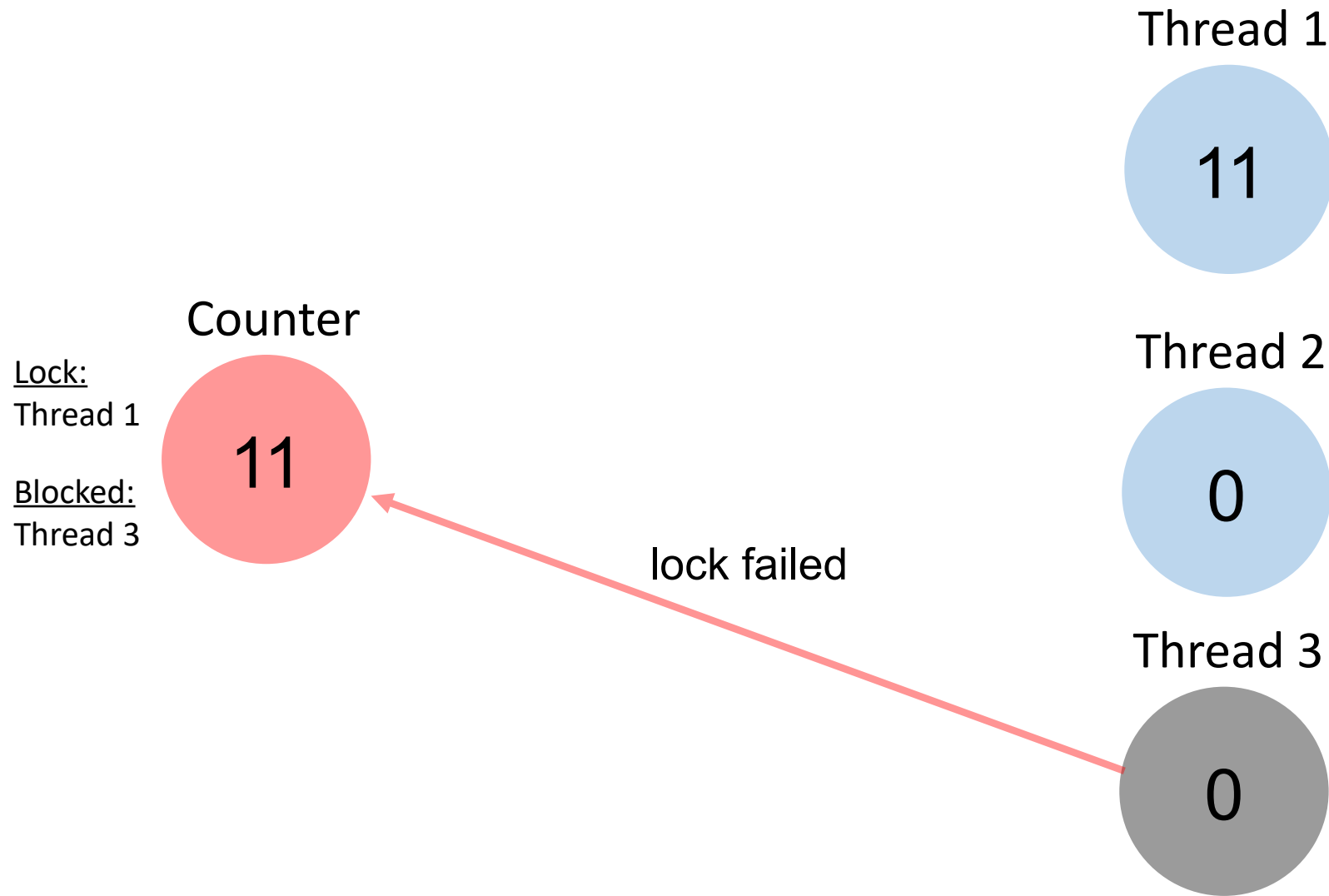


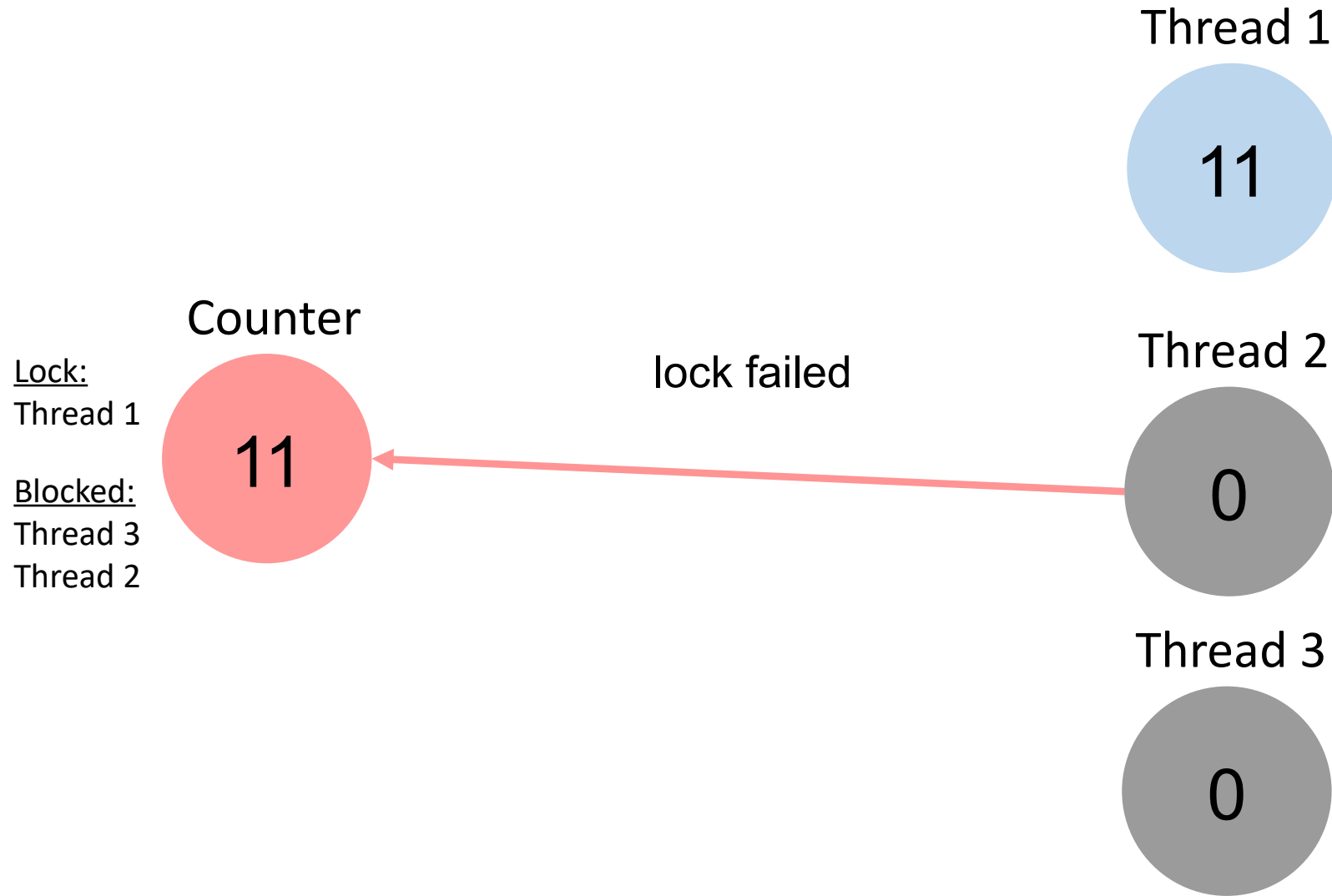
Thread 3

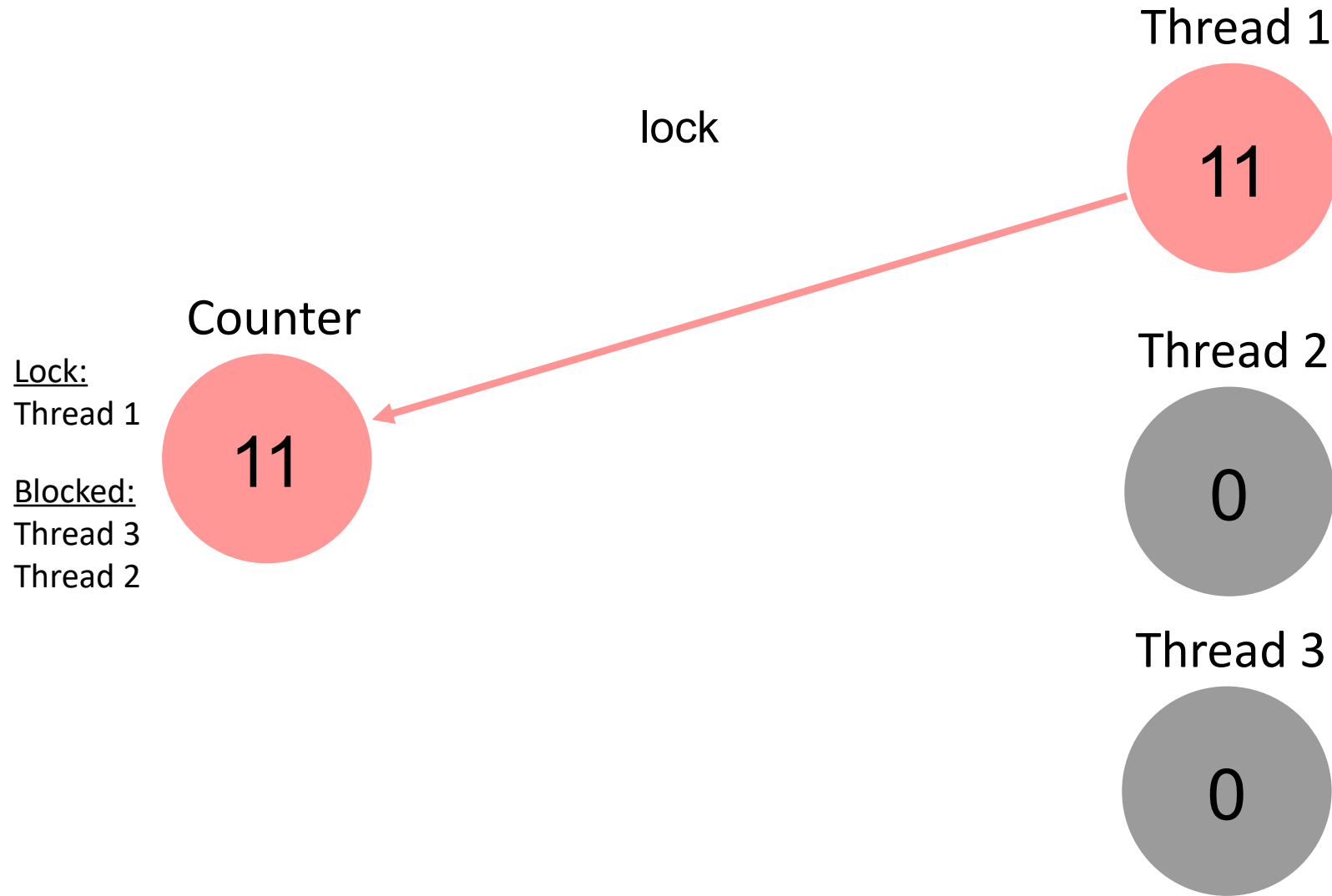


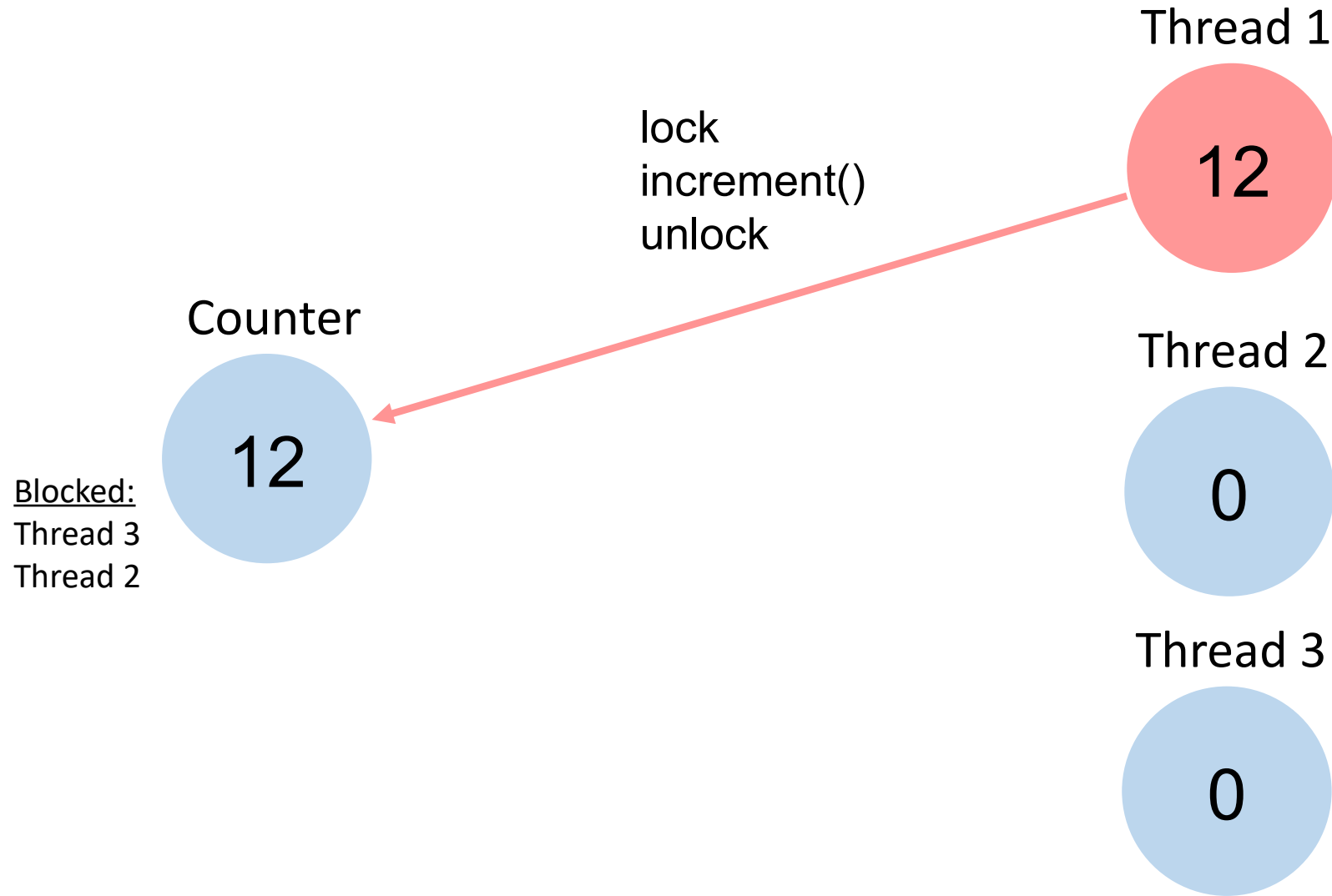












Task C

Analysiert die Reihenfolge, in der der Counter erhöht wird (printed die Thread-ID). Könnt ihr ein Muster erkennen? Was könnten mögliche Gründe für dieses Verhalten sein?

Task D

- Implementiert einen “FairThreadCounter”, der die Threads nacheinander (im Kreis) den Counter erhöhen lässt.
- Für 4 Threads: 1,2,3,4,1,2,3,4,... Englisch: Round-Robin
- Braucht dafür **wait** und **notify**.
- Überlegt euch auch, wie ihr das ohne wait und notify lösen würdet.

Wait and Notify

Ziel: Koordination zwischen Threads

Alle Java Objekte besitzen wait() und notify() Methoden (vererbt von Object Class)

Ein Thread muss das Lock des Objektes besitzen, um wait() oder notify() aufzurufen.

Wait and Notify

wait: Der Thread gibt das Lock wieder ab und wird zum “waiting set” für dieses Objekt hinzugefügt. Der Thread wartet, bis notify auf dem Objekt aufgerufen wird.

notify: Weckt einen zufälligen Thread aus dem «waiting set» für dieses Objekt auf.

notifyAll: Weckt alle Threads aus dem “waiting set” auf.

Wait and Notify Recap

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

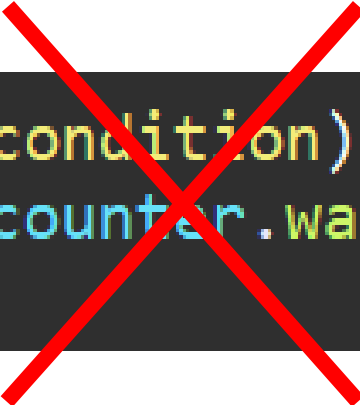
```
if (condition) {  
    counter.wait();  
}
```

Was ist der Unterschied? Gibt es Probleme?

Wait and Notify Recap

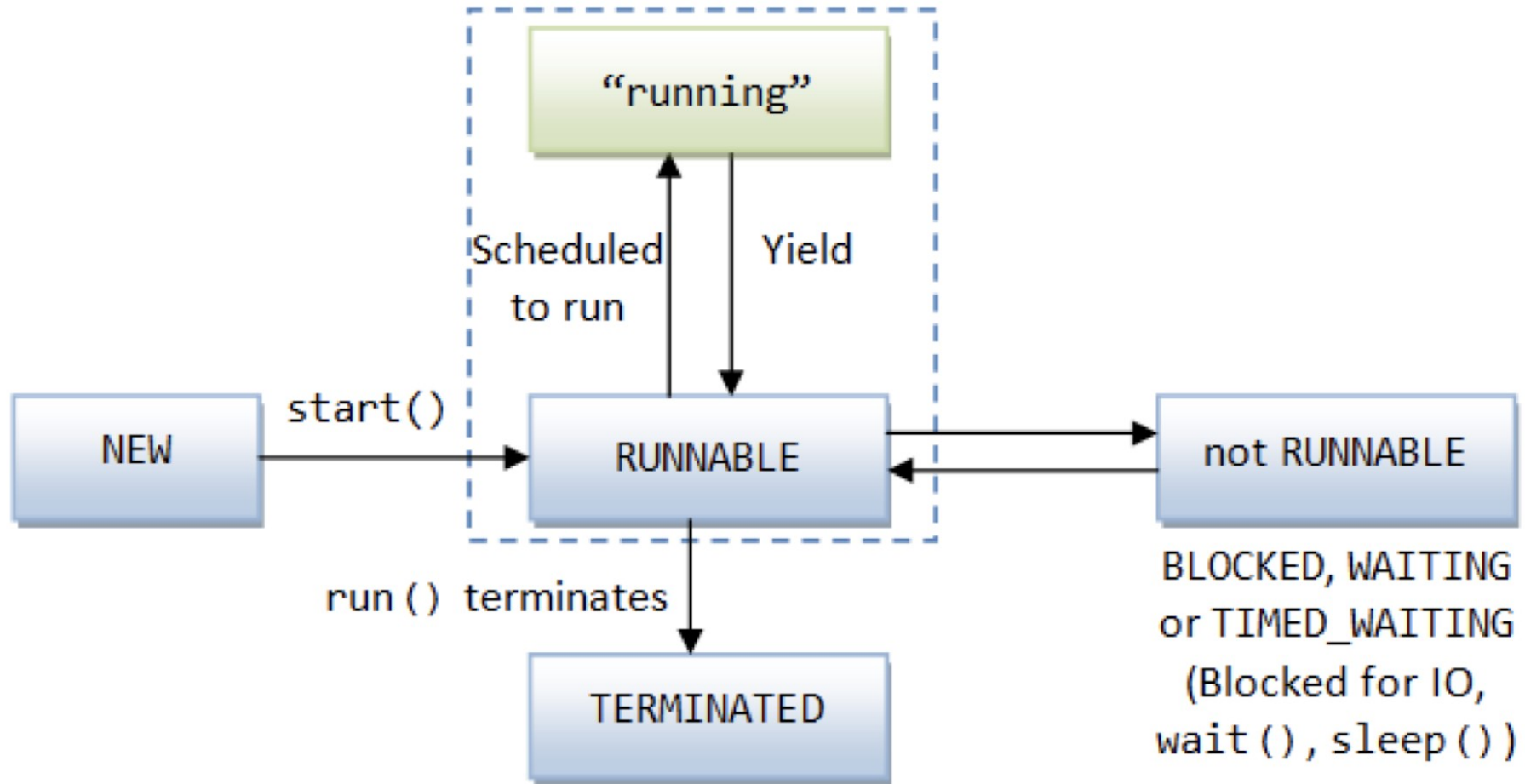
```
while (condition) {  
    counter.wait();  
}
```

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



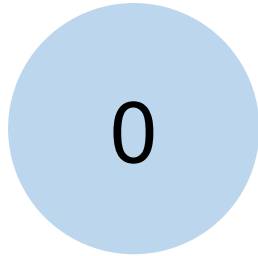
Problem: “Spurious Wakeups” und der Fall, dass `notifyAll()` aufgerufen wird.

Thread State Model

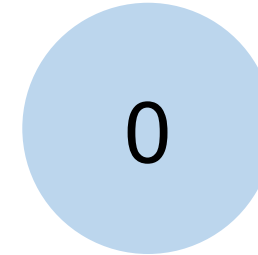


Thread 1 muss als erstes den Counter erhöhen.

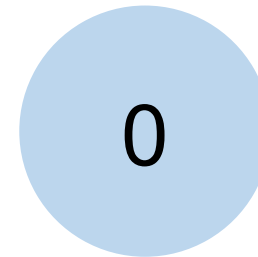
Counter



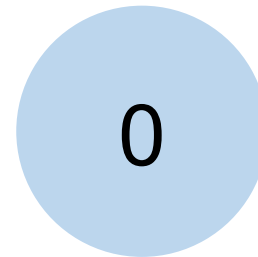
Thread 1



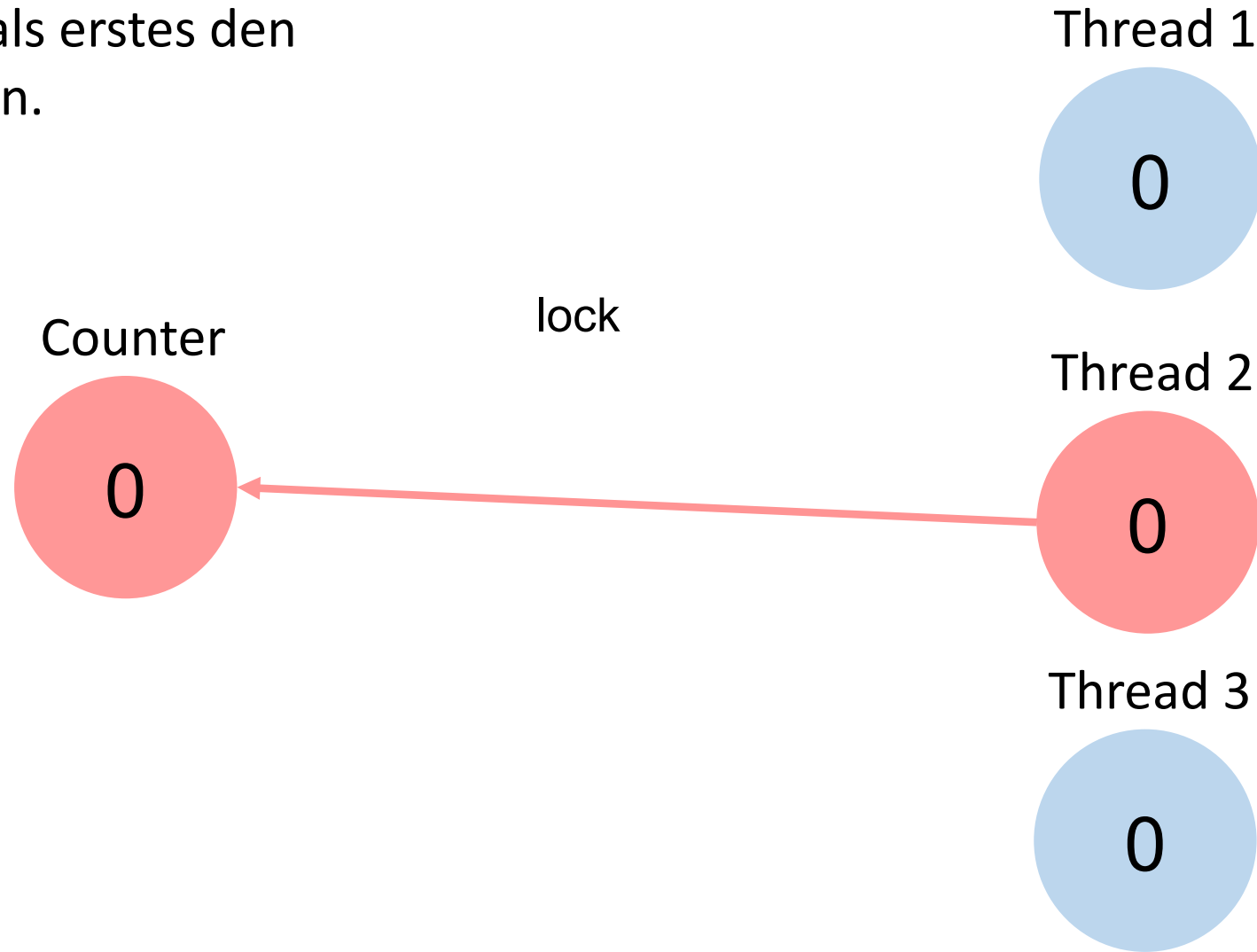
Thread 2



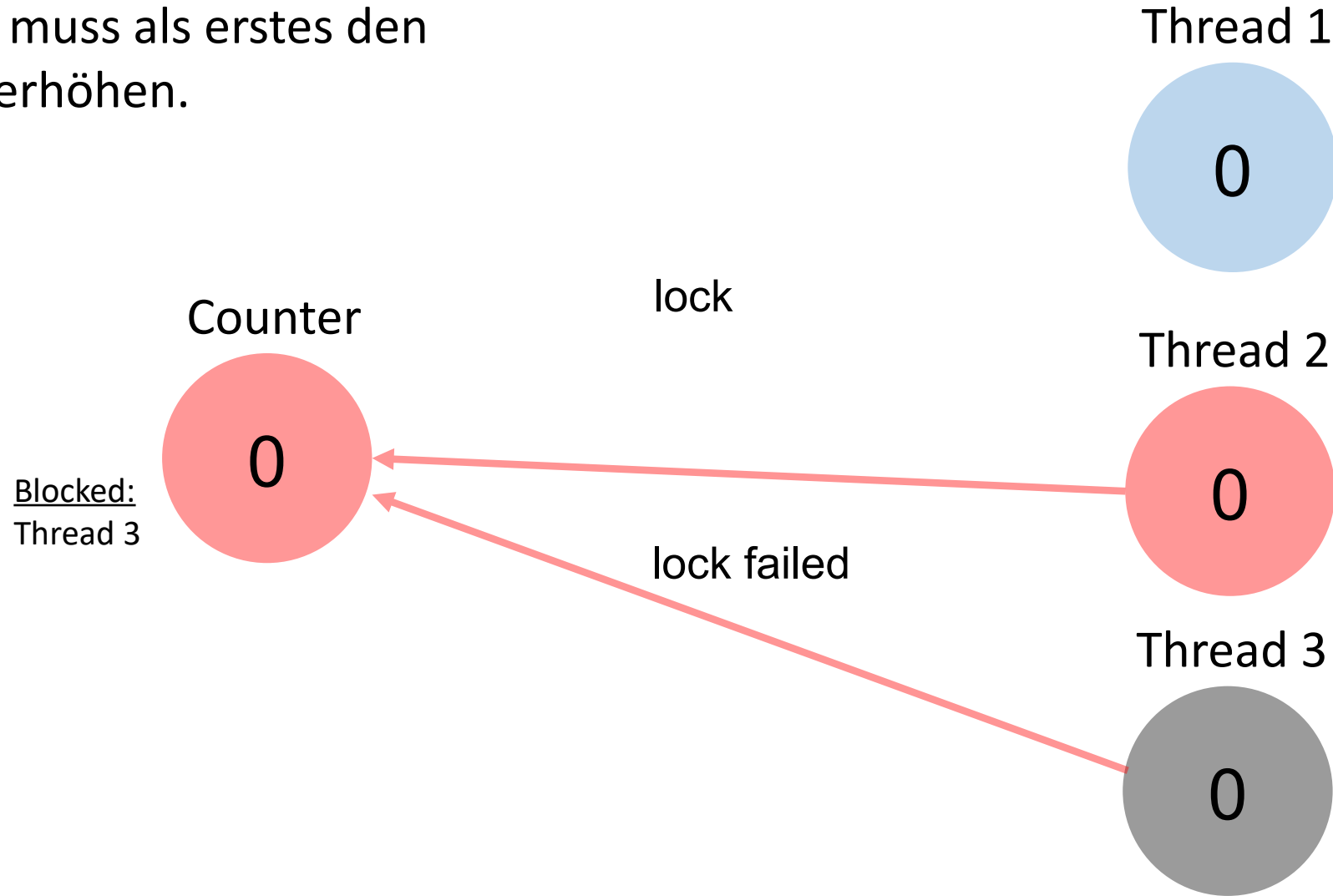
Thread 3



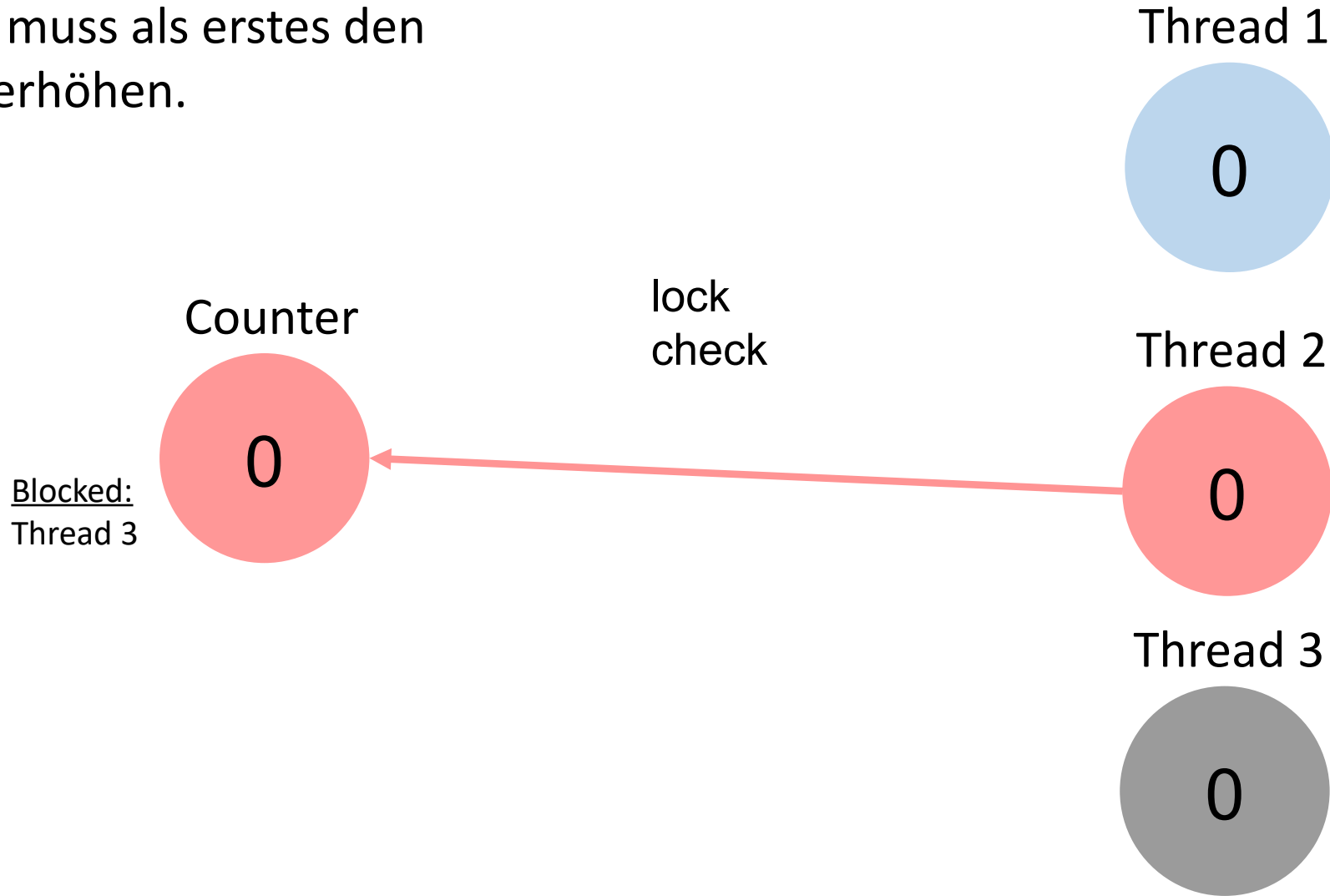
Thread 1 muss als erstes den Counter erhöhen.



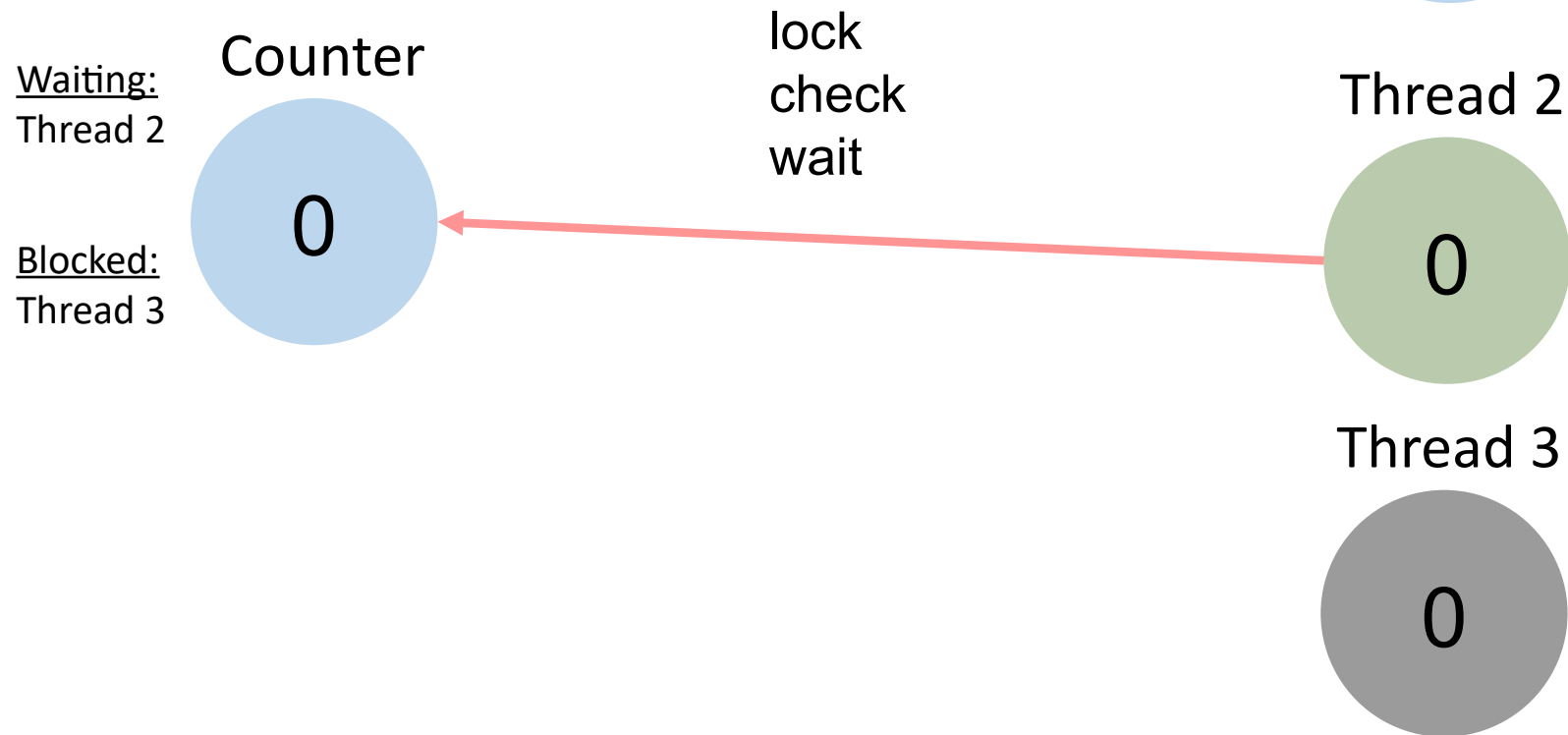
Thread 1 muss als erstes den Counter erhöhen.



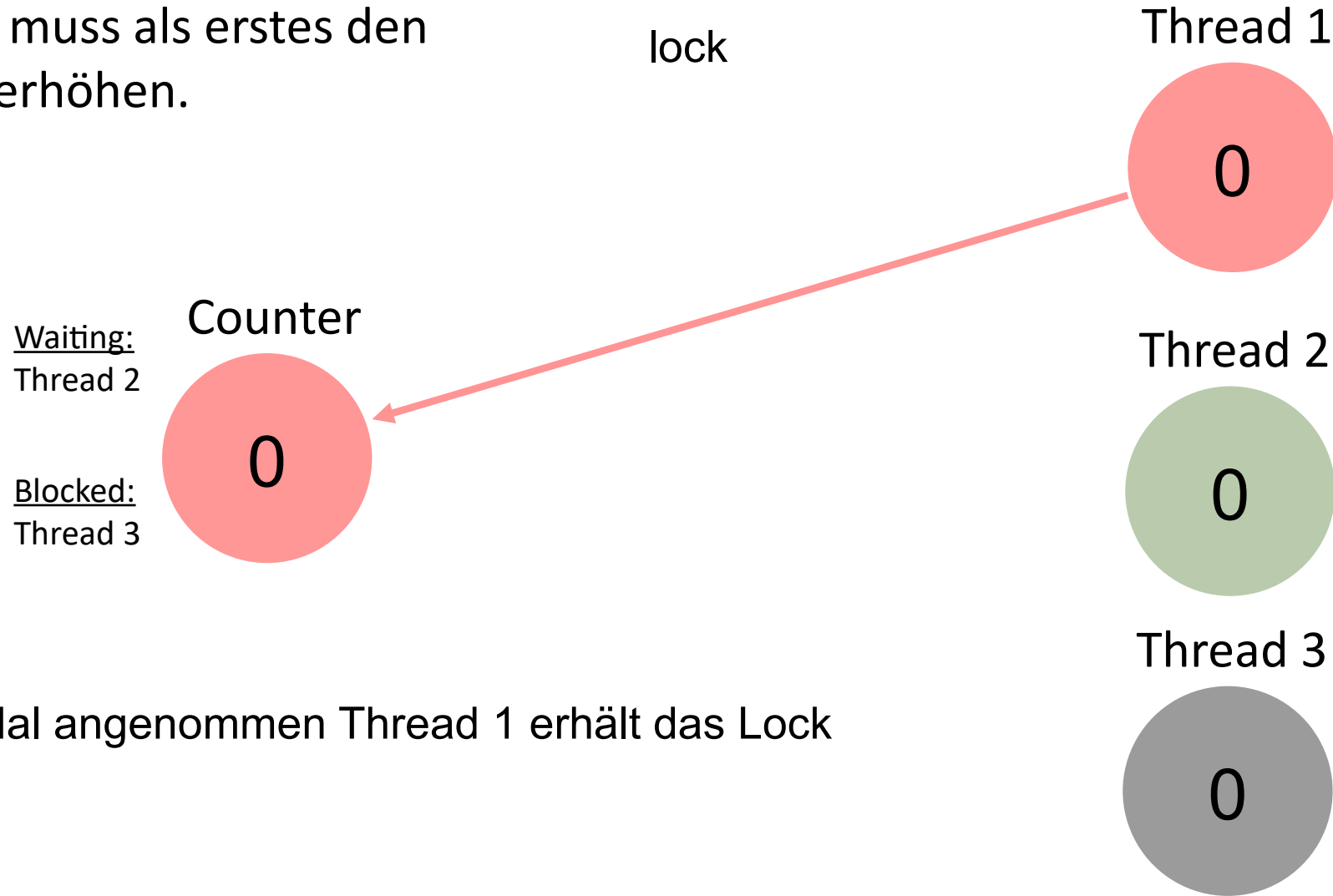
Thread 1 muss als erstes den Counter erhöhen.



Thread 1 muss als erstes den Counter erhöhen.

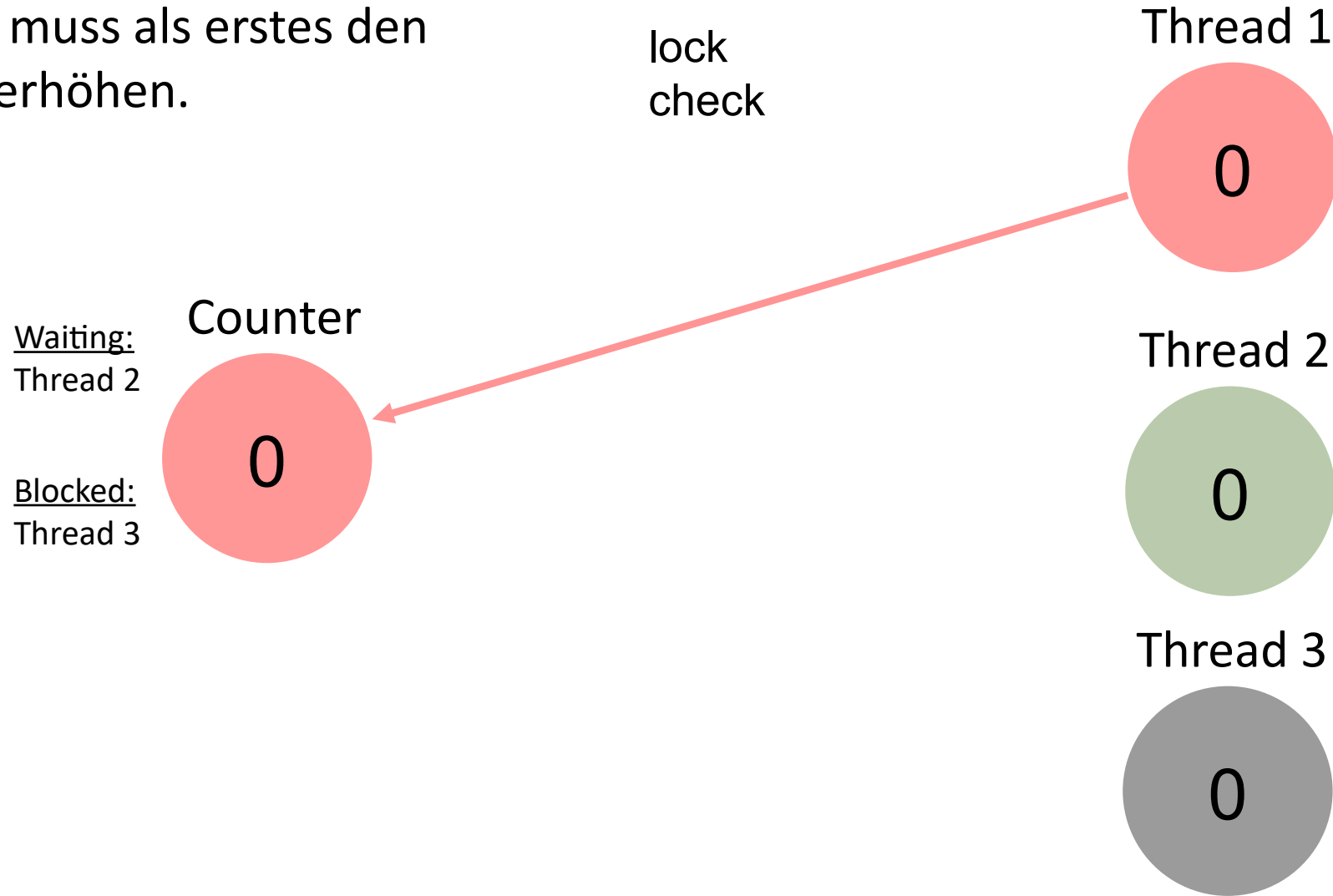


Thread 1 muss als erstes den Counter erhöhen.

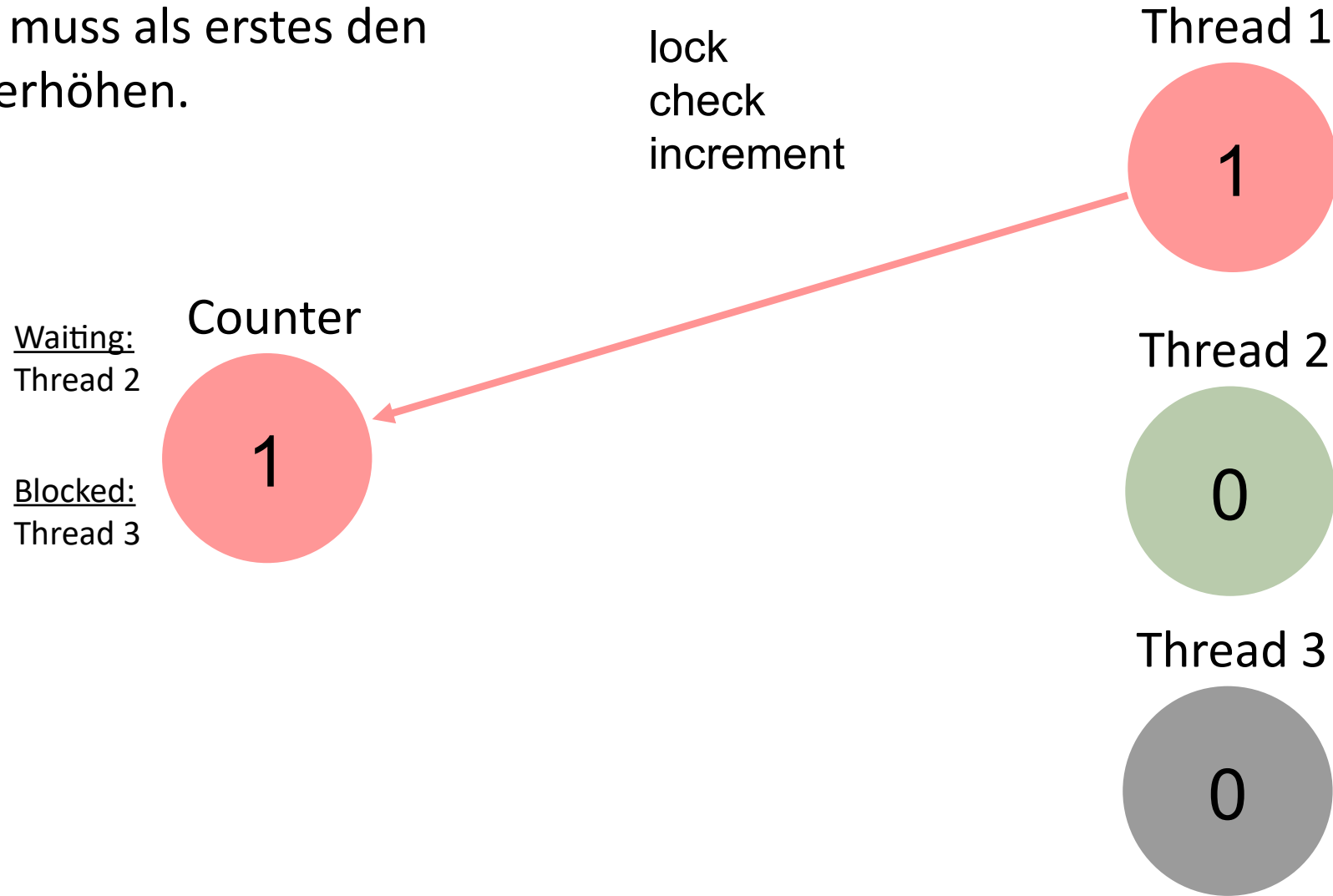


Mal angenommen Thread 1 erhält das Lock

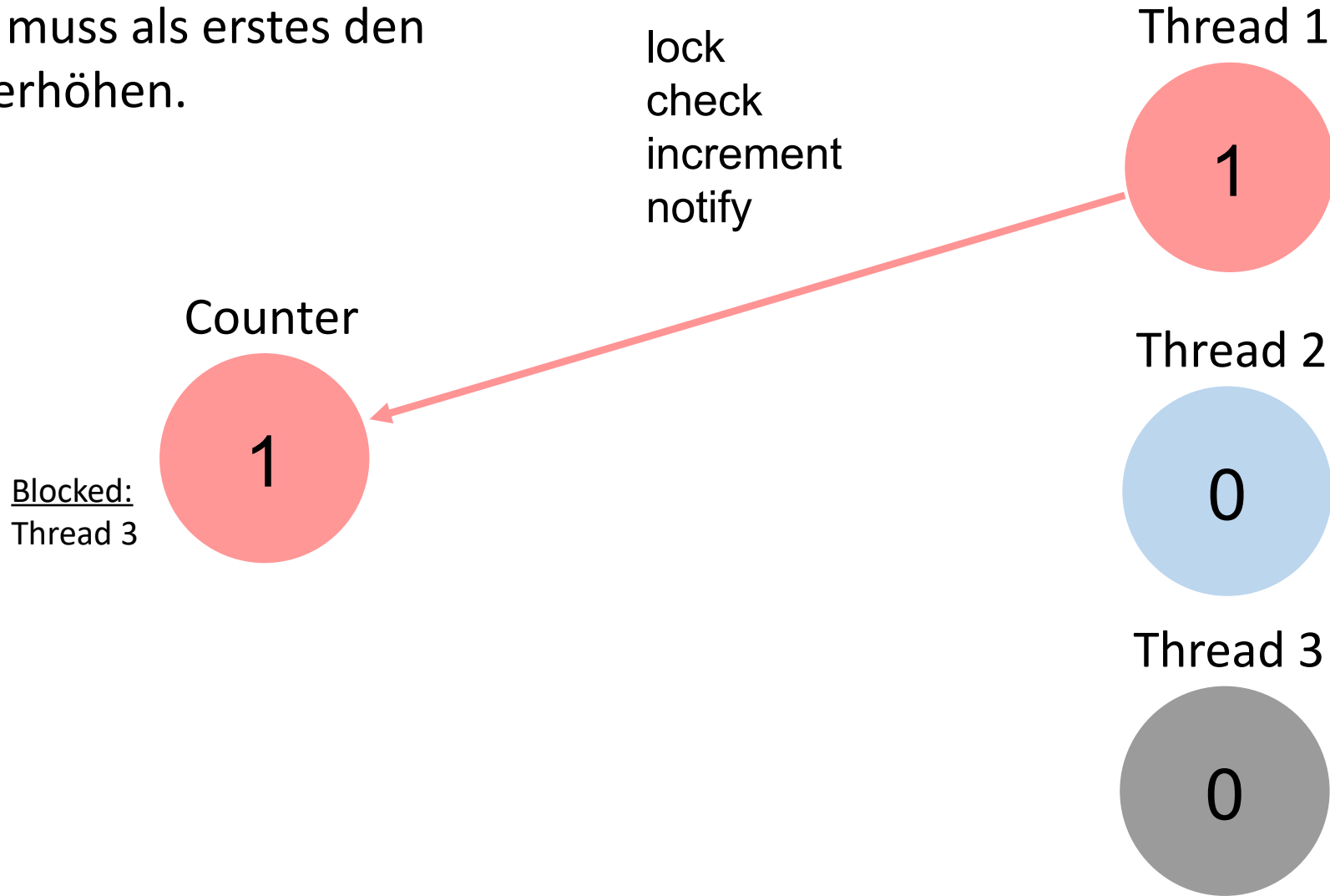
Thread 1 muss als erstes den Counter erhöhen.



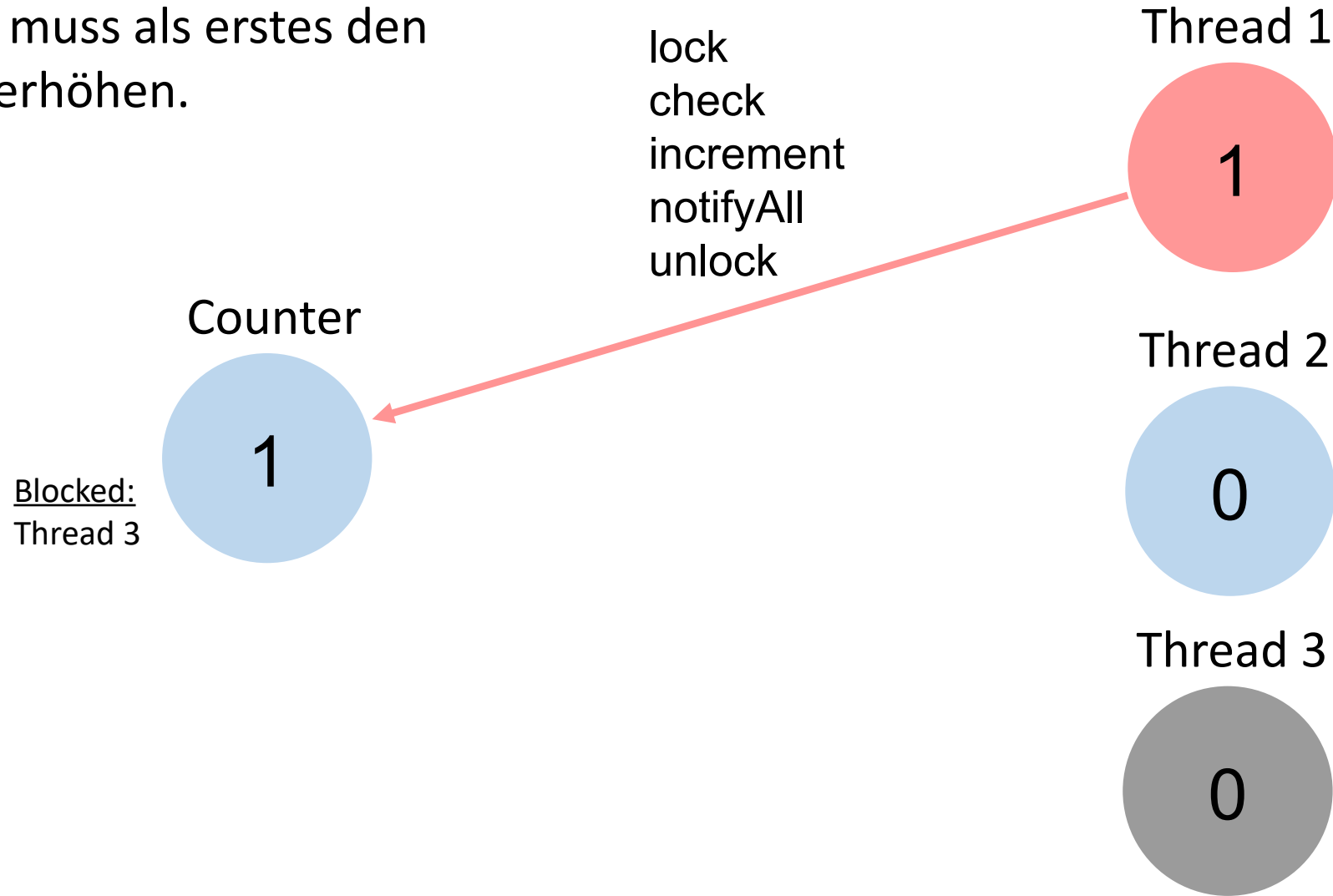
Thread 1 muss als erstes den Counter erhöhen.



Thread 1 muss als erstes den Counter erhöhen.



Thread 1 muss als erstes den Counter erhöhen.



Task E – Atomic counter

Implementiert eine weitere Thread-sichere Version. Benutzt dafür einen `AtomicCounter`. Dieser kann sicher von mehreren Threads gleichzeitig gebraucht werden.

Atomic Variables

- Eine Menge von Java Klassen z.B. AtomicInteger, AtomicLong, ...
- Eine Operation ist **atomic**, wenn sie von den anderen Threads als untrennbar wahrgenommen wird.
- *Wird durch spezielle Hardware Instruktionen implementiert. Mehr dazu später in der Vorlesung.*
- *Mehr Infos in der Dokumentation*

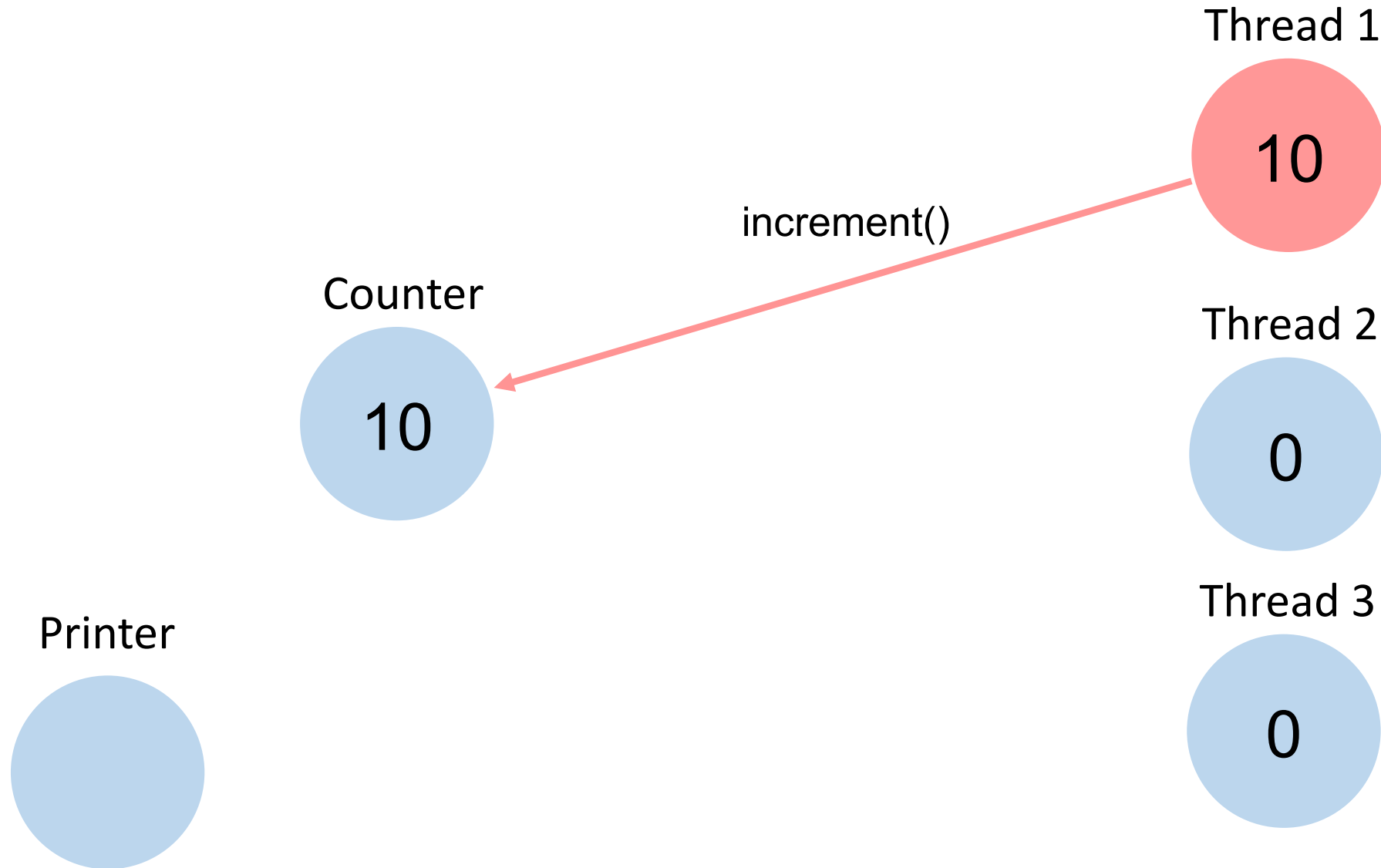
Task F – Atomic vs Synchronized counter

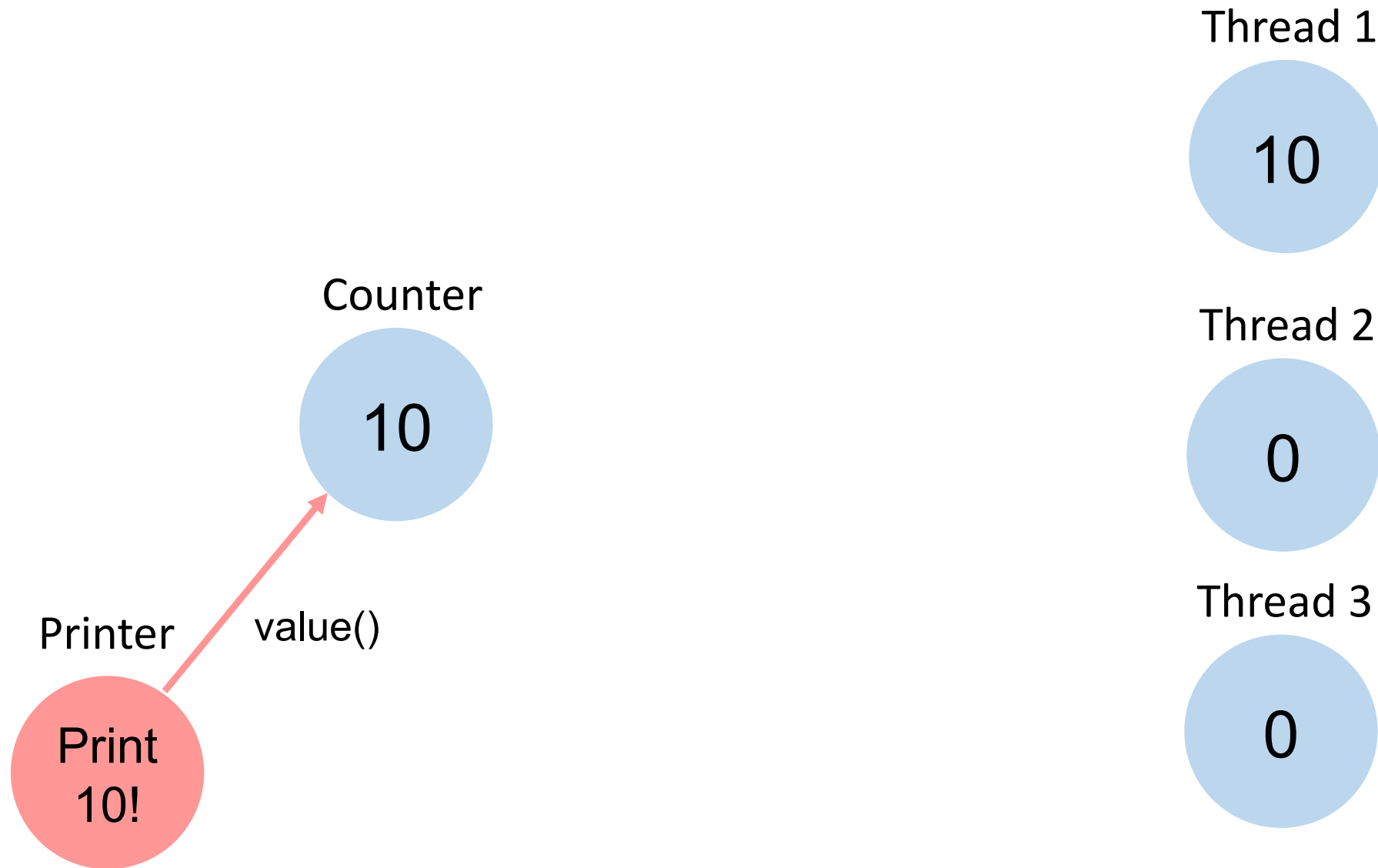
Vergleicht den AtomicCounter mit dem Synchronized Counter durch Messungen der Laufzeit.

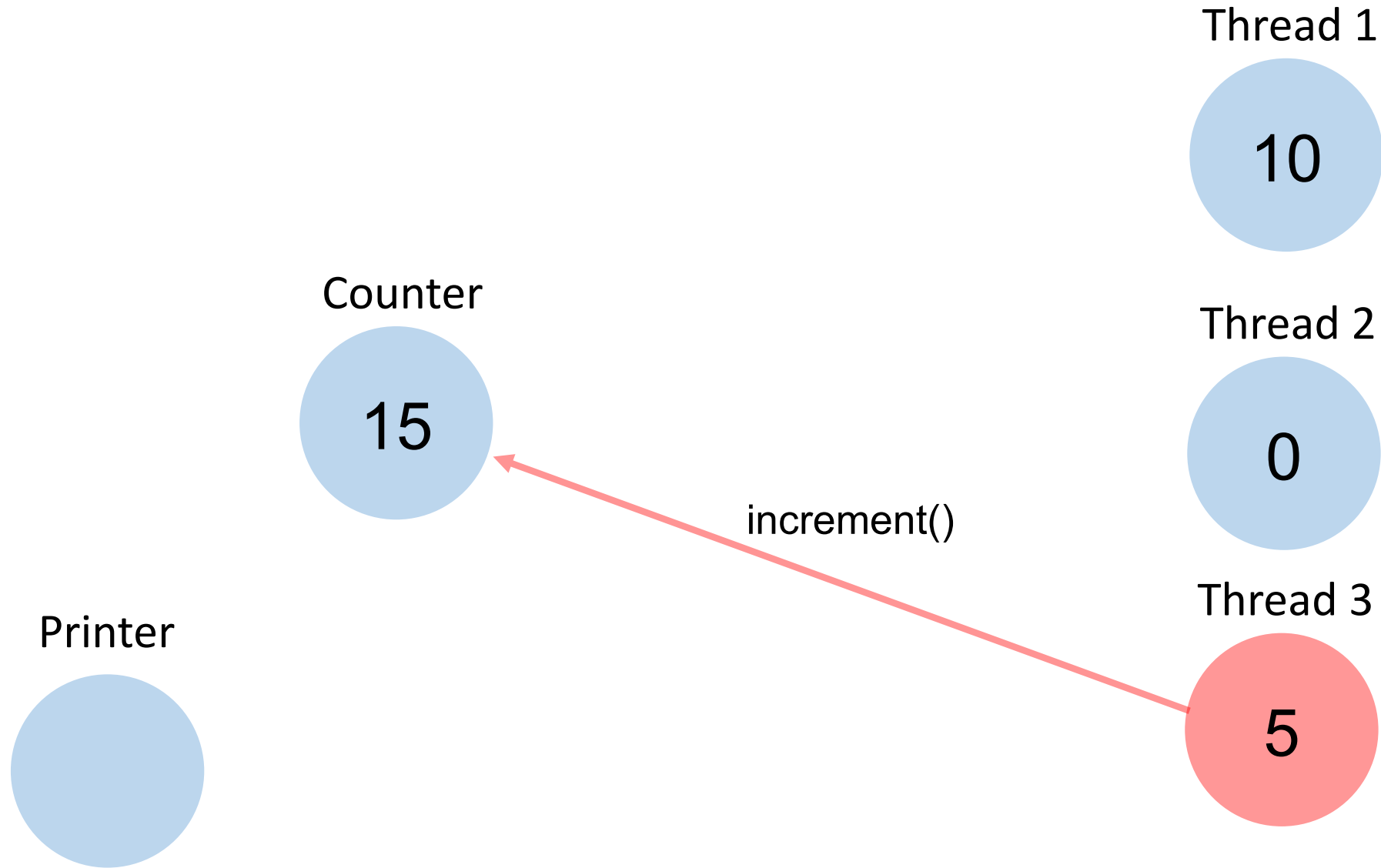
- Variiert die Arbeit pro Thread
- Variiert die Anzahl an Threads

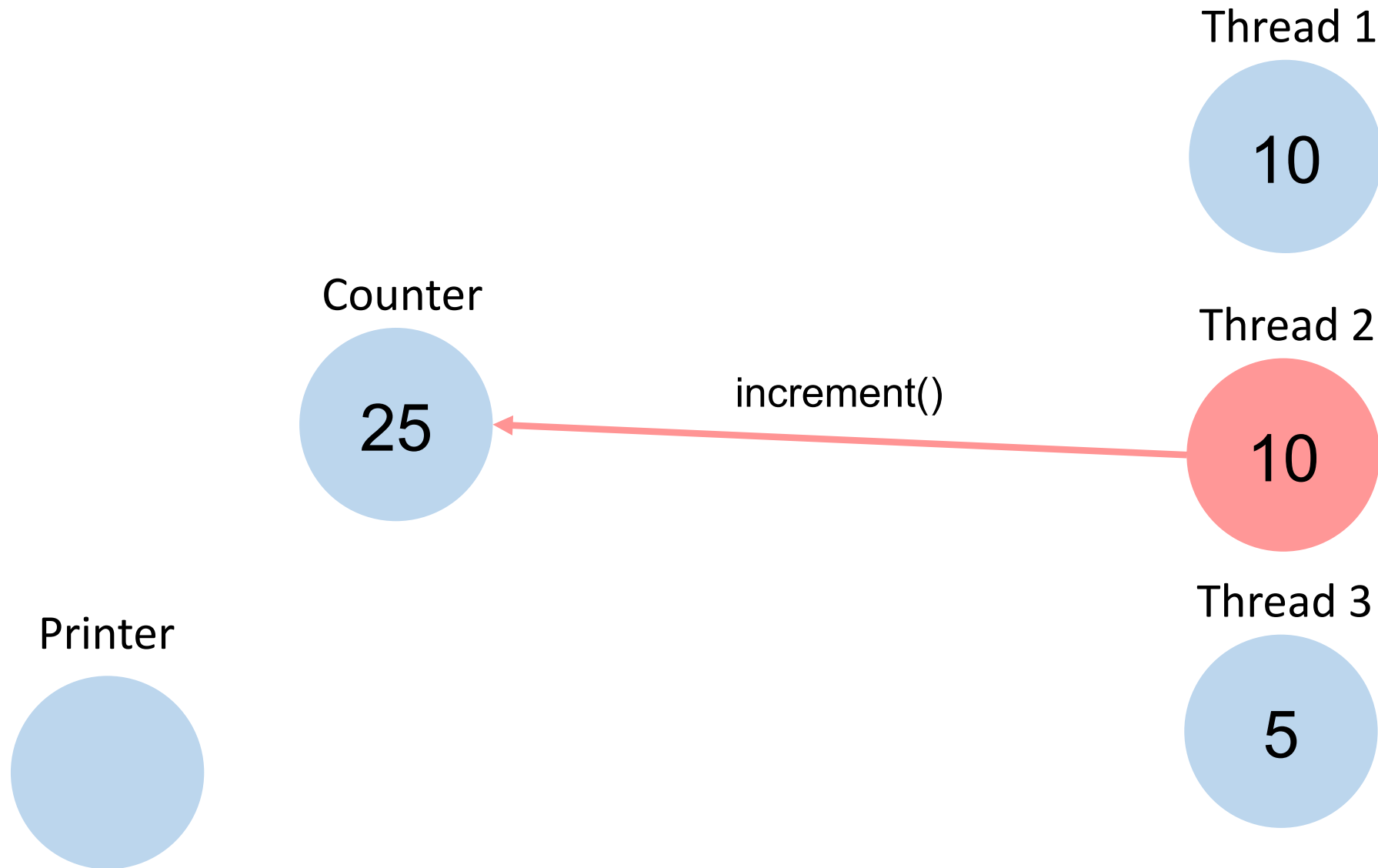
Task G

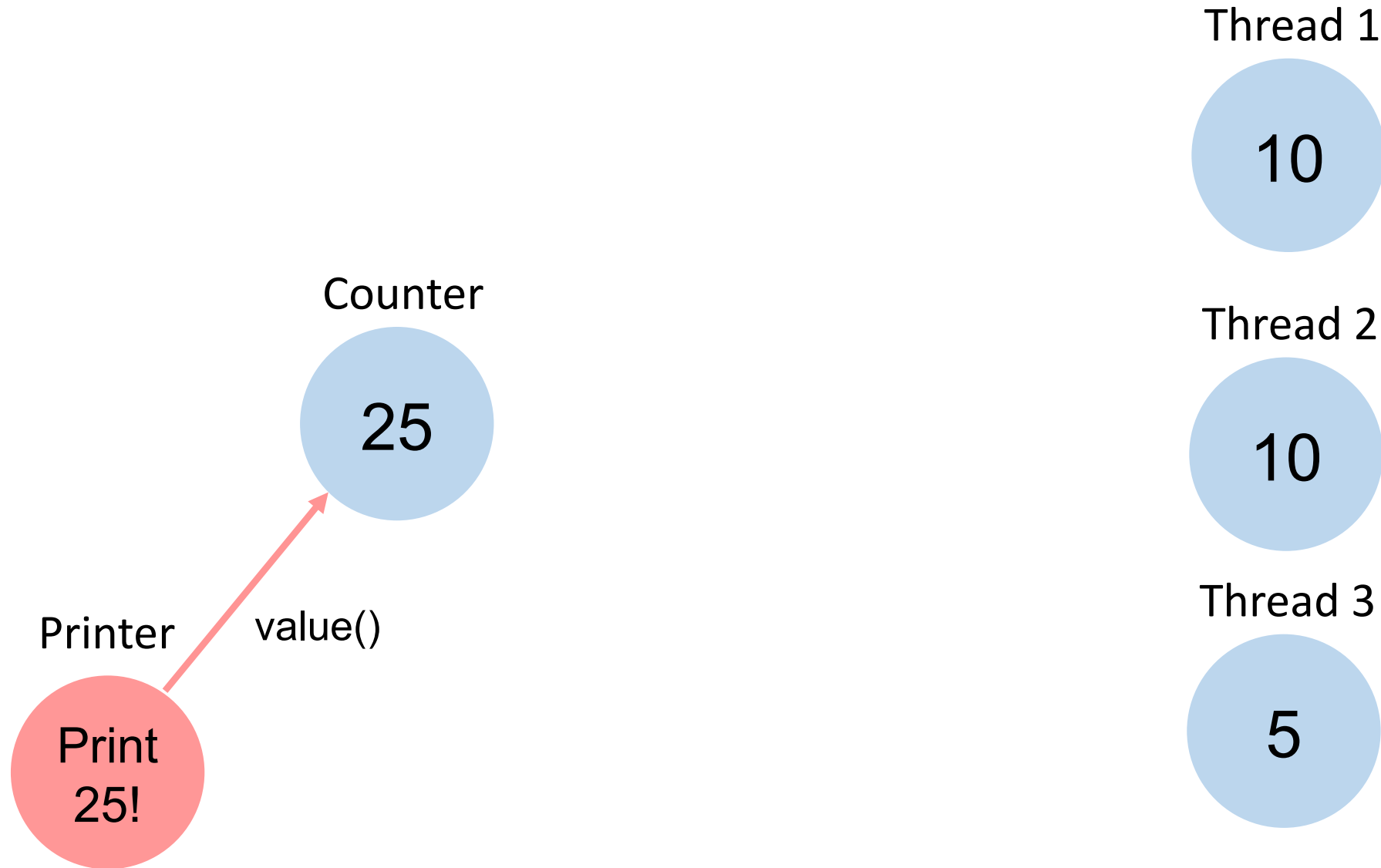
Implementiert einen Thread, der den Fortschritt des Counters misst. Dieser Thread soll den Counter beobachten und den aktuellen Stand in der Konsole darstellen. Der Thread soll am Schluss richtig beendet werden (`thread.interrupt()`).

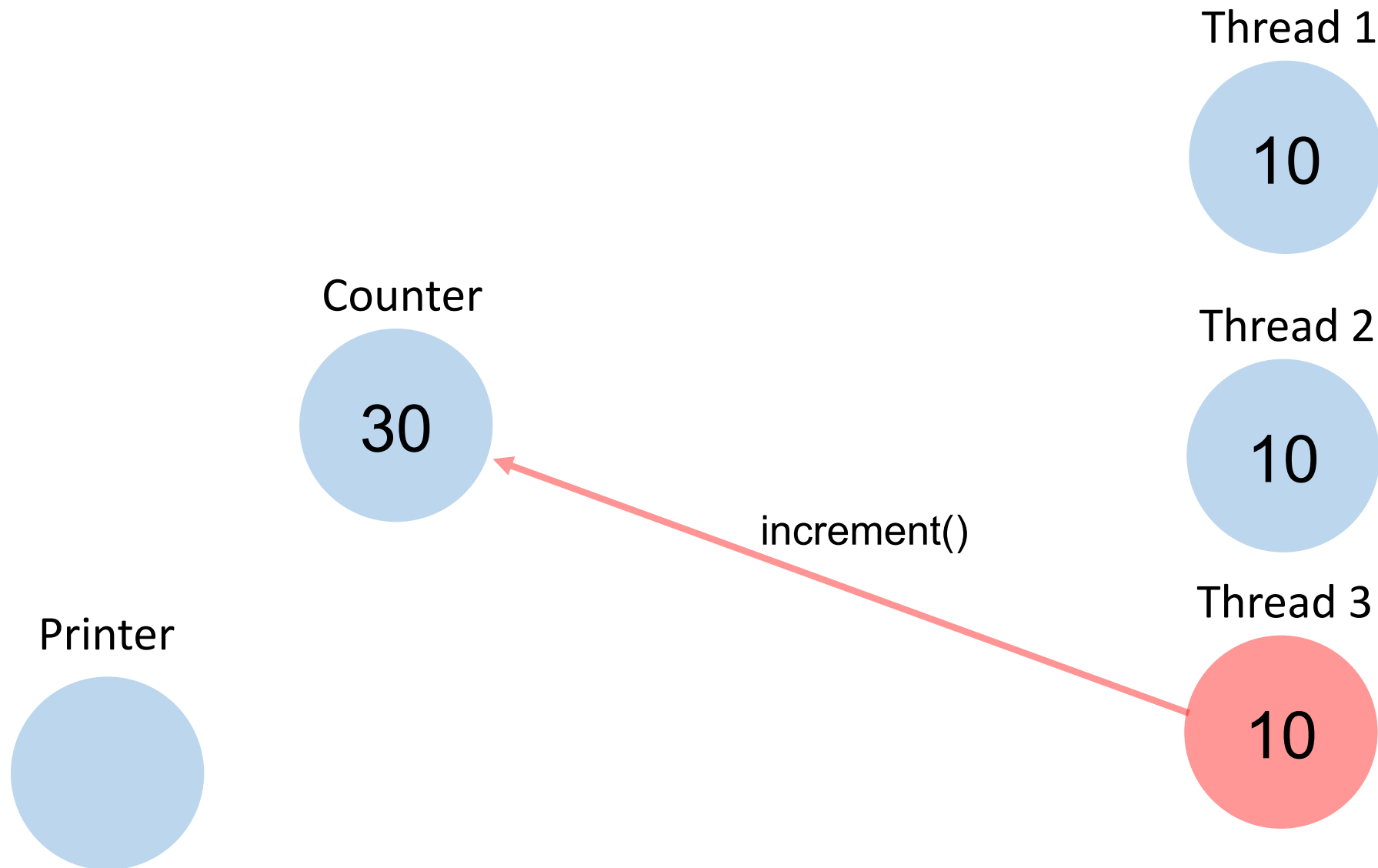


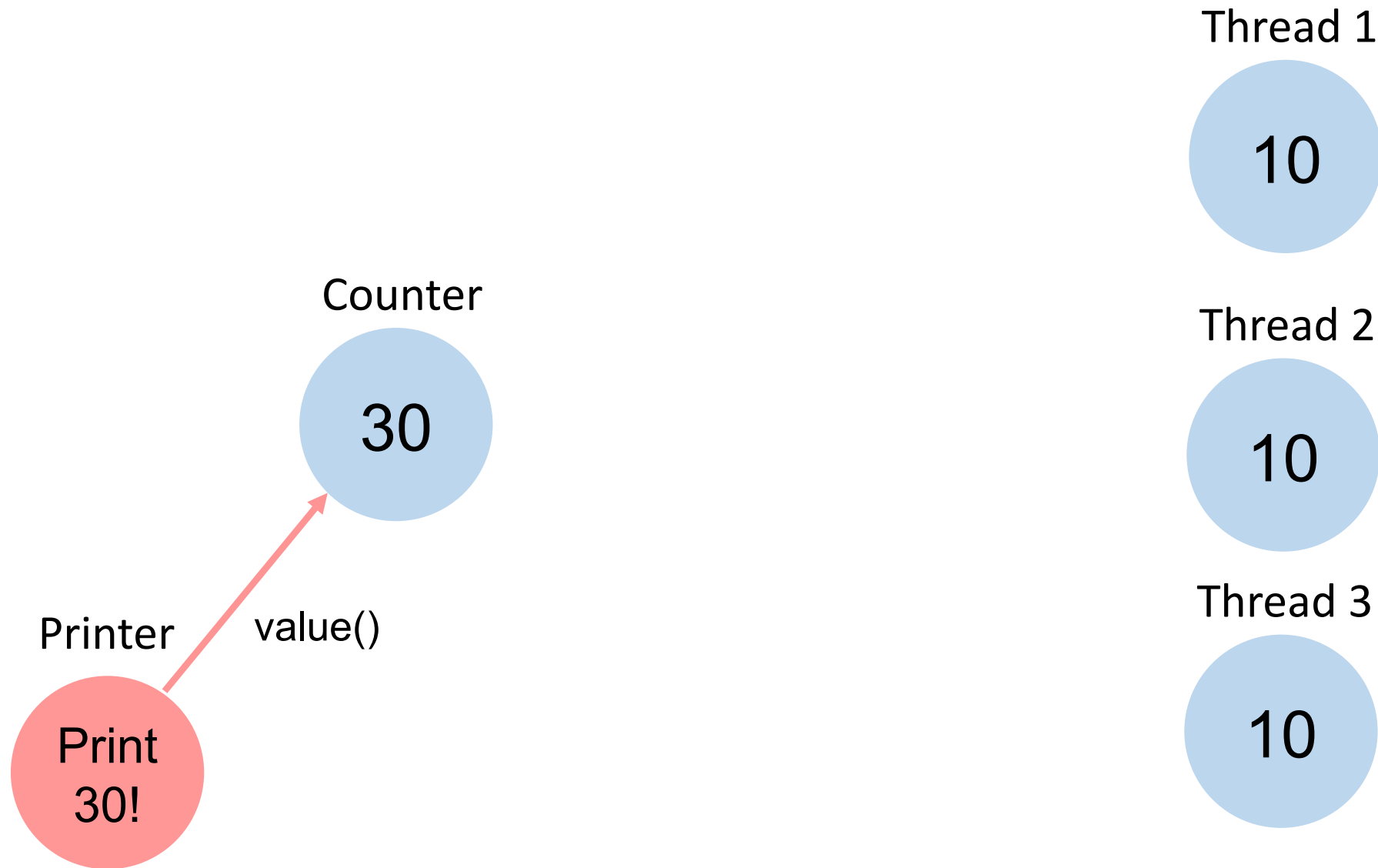












Theorie Recap

Reentrant

Java locks sind “reentrant”. D.h. wenn ein Thread das Lock besitzt kann er in andere synchronized Blöcke des gleichen Objekts gelangen.

Ein Thread kann in Besitz von verschiedenen Locks gleichzeitig sein. Sollte diese dann auch alle wieder freigeben.

Parallelism vs. Concurrency?

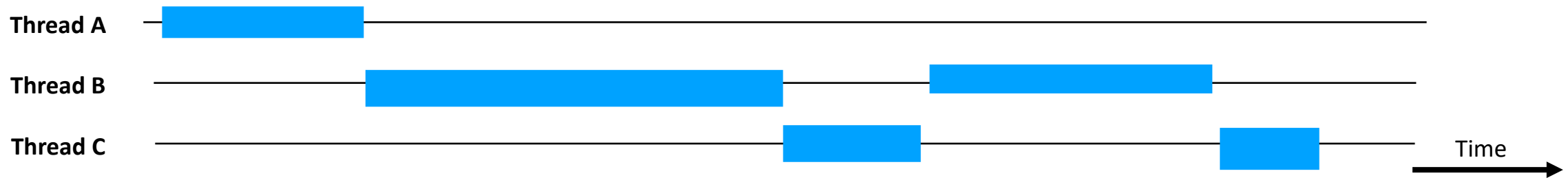
Parallelism: Mehrere Aufgaben zur gleichen Zeit erledigen

Concurrency: Mehrere Aufgaben gleichzeitig bewältigen.

D.h. aber nicht, dass zu einem Zeitpunkt mehrere Aufgaben gleichzeitig bearbeitet werden.

Concurrency vs Parallelism

Concurrent, not parallel



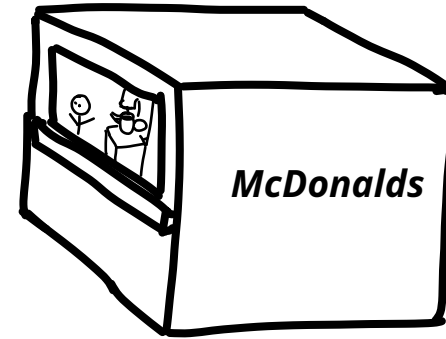
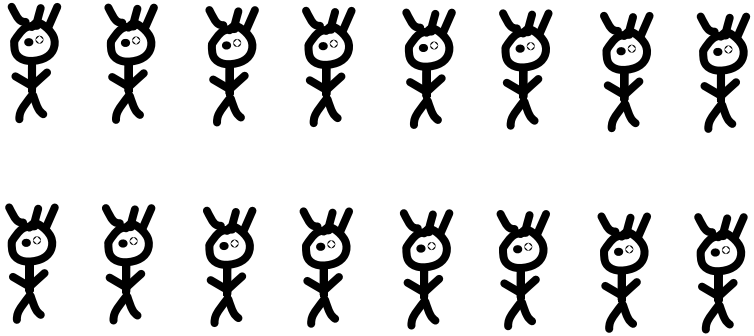
Concurrent, parallel



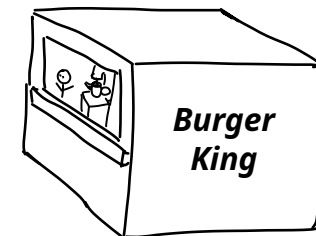
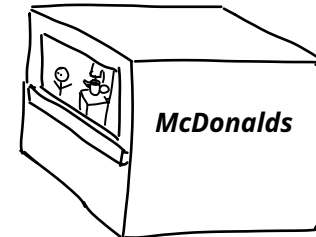
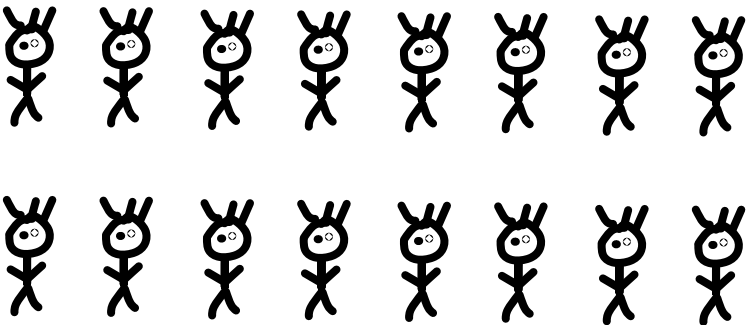
Not concurrent, not parallel



Concurrency vs Parallelism



Concurrency
No parallelism



Concurrency
Parallelism

Demo

Alte Prüfungsaufgaben

Past Exam Task

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

- ☐ Die `start()` Methode in `t = new Thread(); t.start()` ruft automatisch auch die `run()` methode auf.
- ☐ Die `run()` Methode in `t = new Thread(); t.run()` erzeugt einen neuen Thread und führt diesen aus.
- ☐ Ein Codeblock mit mehreren Threads wird immer deterministisch ausgeführt. D.h. der Output ist immer exakt der gleiche.
- ☐ Ein komplett serieller Codeblock kann zur Beschleunigung auf mehreren Prozessoren ausgeführt werden.

Mark all correct statements regarding the execution of Java Threads.

- The `start()` method in `t = new Thread(); t.start()` automatically also calls the `run()` method.*
- The `run()` method in `t = new Thread(); t.run()` creates a new thread and executes the thread.*
- A codeblock with several threads is always executed deterministically. That means the output is always the same.*
- A fully serial block of code can be run on multiple processors to speedup execution.*

Past Exam Task

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

- ☒ Die `start()` Methode in `t = new Thread(); t.start()` ruft automatisch auch die `run()` methode auf.
- ☐ Die `run()` Methode in `t = new Thread(); t.run()` erzeugt einen neuen Thread und führt diesen aus.
- ☐ Ein Codeblock mit mehreren Threads wird immer deterministisch ausgeführt. D.h. der Output ist immer exakt der gleiche.
- ☐ Ein komplett serieller Codeblock kann zur Beschleunigung auf mehreren Prozessoren ausgeführt werden.

Mark all correct statements regarding the execution of Java Threads.

- The `start()` method in `t = new Thread(); t.start()` automatically also calls the `run()` method.*
- The `run()` method in `t = new Thread(); t.run()` creates a new thread and executes the thread.*
- A codeblock with several threads is always executed deterministically. That means the output is always the same.*
- A fully serial block of code can be run on multiple processors to speedup execution.*

Past Exam Task

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

- ☐ Um eine Prioritätenreihenfolge zwischen mehreren Threads zu erzwingen.
- ☐ Um das von dem aktuellen Thread gehaltene Lock freizugeben.
- ☐ Um die Ausführung des aktuellen Threads anzuhalten, bis der Thread, den er `joined`, abgeschlossen ist.
- ☐ Um die Kontrolle an einen anderen Thread zu übergeben, ohne auf dessen Abschluss zu warten.

What is the purpose of the `join()` method in Java Threads?. (2)

To enforce a priority order among multiple threads.

To release the lock held by the current thread.

To pause the current thread's execution until the thread it `joins` completes.

To transfer control to another thread without waiting for its completion.

Past Exam Task

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

- ☐ Um eine Prioritätenreihenfolge zwischen mehreren Threads zu erzwingen.
- ☐ Um das von dem aktuellen Thread gehaltene Lock freizugeben.
- ☒ **Um die Ausführung des aktuellen Threads anzuhalten, bis der Thread, den er joined, abgeschlossen ist.**
- ☐ Um die Kontrolle an einen anderen Thread zu übergeben, ohne auf dessen Abschluss zu warten.

What is the purpose of the `join()` method in Java Threads?. (2)

To enforce a priority order among multiple threads.

To release the lock held by the current thread.

To pause the current thread's execution until the thread it joins completes.

To transfer control to another thread without waiting for its completion.



https://quizizz.com/admin/quiz/6225327b387927001df6b1f5?source=quiz_share

Replace link with link to quiz