Parallele Programmierung FS25

Exercise Session 10

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- Theory
- Intro Assignment 10
- Exam questions
- Kahoot

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too

Organisation

- Mein Name ist Jonas Wetzel
- Meine Website (Materialien und Inhalt der Übungen): n.ethz.ch/~jwetzel
- Meine Email: jwetzel@ethz.ch
- Discord: @jonas.too
- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9

Organisation

- Feedback zur Session: https://forms.gle/qiDnqkfSP2NUQGvc9
- Falls ihr Feedback möchtet kommt bitte zu mir

Organisation

- Wo sind wir jetzt?

Semaphores
Barriers
Monitors
Conditional Locks

Plan für heute

- Organisation
- **Nachbesprechung Assignment 9**
- Theory
- Intro Assignment 10
- Exam questions
- Kahoot

Feedback: Assignment 9

Recap: Critical Section Properties

- **Mutual exclusion**: No more then one process executing in the critical section
- **Progress**: When no process is in the critical section, any process that requests entry must be permitted without delay
- **No starvation (bounded wait)**: If any process tries to enter its critical section then that process must eventually succeed.

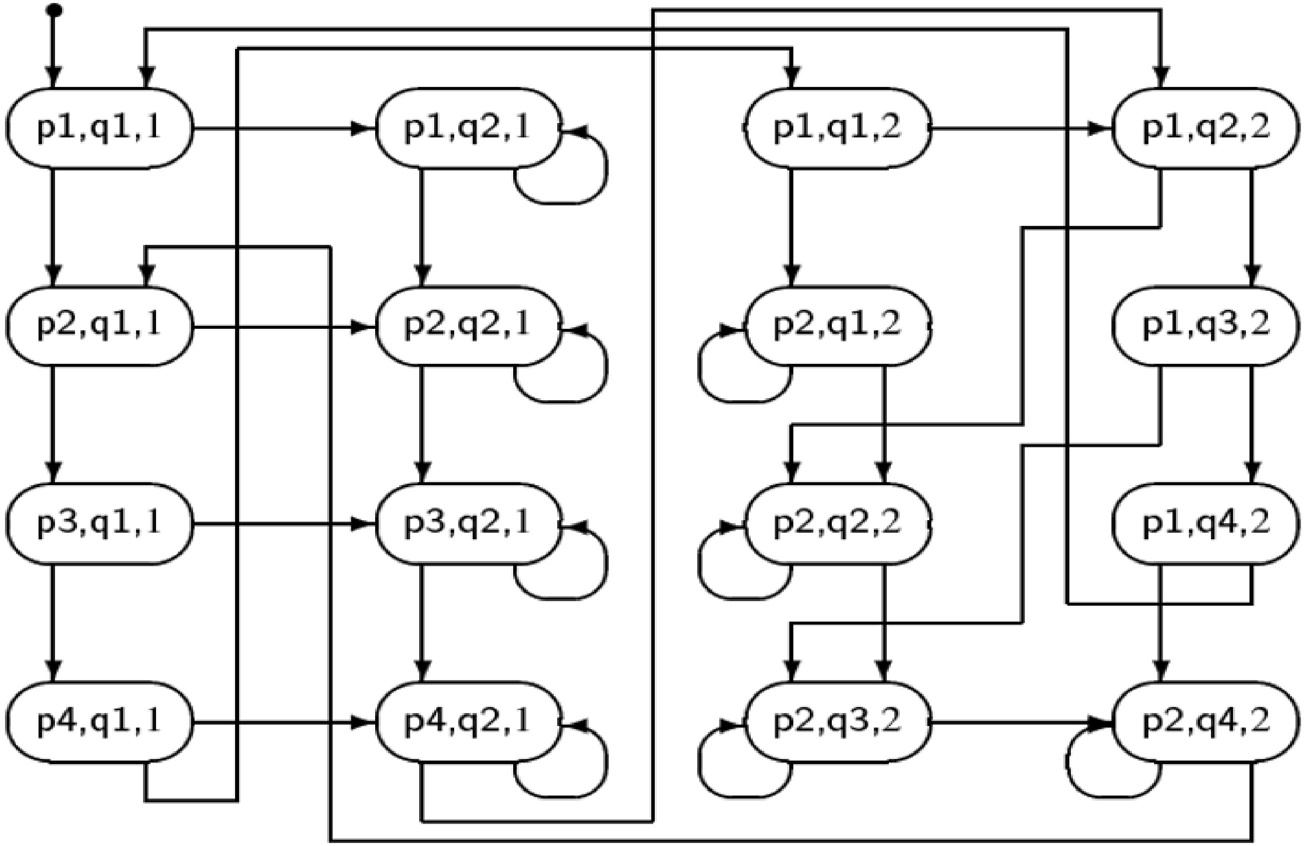
P

```
p1:   Non-critical section P
p2:   while turn != 1
p3:   Critical section
p4:   turn = 2
```

turn = 1

Q

```
q1:   Non-critical section Q
q2:   while turn != 2
q3:   Critical section
q4:   turn = 1
```



- **Mutual exclusion**: E.g. State (p3,q3,_) is not reachable
- **Progress**: E.g. There exists a path for P such that state (P3, _ , _) is reachable from (P2,_,_). Typical couterexamples: deadlocks and livelocks
- **No starvation (bounded wait)**: Possible starvation reveals itself as cycles in the state diagram.

P

p1: Non-critical section P

p2: while turn != 1

p3: Critical section

p4: turn = 2

turn = 1

Q

q1: Non-critical section Q

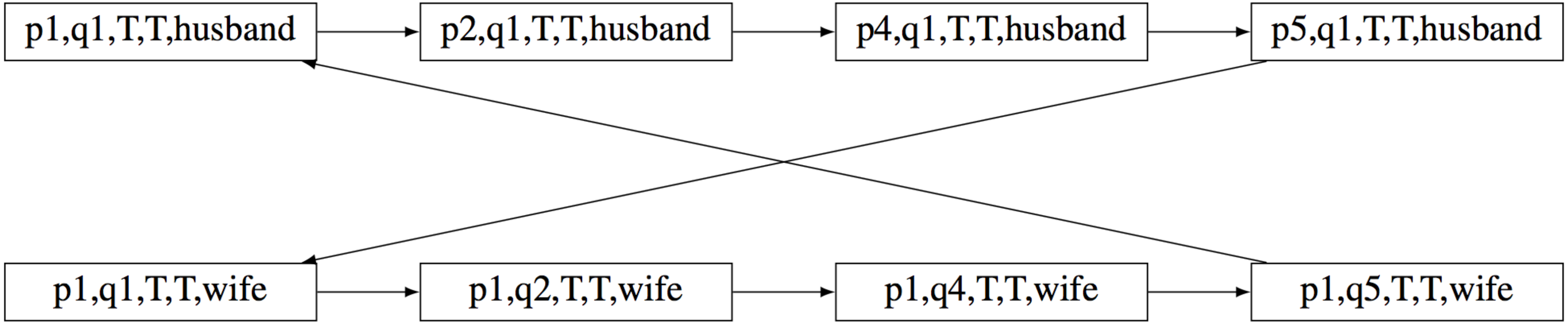
q2: while turn != 2

q3: Critical section

q4: turn = 1

Feedback for Assignment 9

owner			
husband.hungry = true			
wife.hungry = true			
husband		wife	
p1:	while hungry	q1:	while hungry
p2:	owner != me	q2:	owner != me
p3:	sleep	q3:	sleep
p4:	spouse == hungry	q4:	spouse == hungry
p5:	owner = spouse	q5:	owner = spouse
p6:	CR	q6:	CR
p7:	hungry = false	q7:	hungry = false
p8:	owner = spouse	q8:	owner = spouse



Feedback for Assignment 9

- One way to solve the livelock problem is to impose an ordering when acquiring the lock on the shared resource.

- Or one of the spouses can actually take the spoon after certain number of retries

Feedback for Assignment 9

Optimistic vs Pessimistic concurrency control

```
@Override
public int nextInt() {
    // get the current seed value
    long next;
    synchronized (this) {
        long orig = state;
        // using recurrence equation to generate next
        next = (a * orig + c) & (~0L >>> 16);
        // store the updated seed
        state = next;
    }
    return (int) (next >>> 16);
}
```

```
@Override
public int nextInt() {
    while (true) {
        // get the current seed value
        long orig = state.get();
        // using recurrence equation to generate next seed
        long next = (a * orig + c) & (~0L >>> 16);
        // store the updated seed
        if (state.compareAndSet(orig, next)) {
            return (int) (next >>> 16);
        }else{
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {

            }
        }
    }
}
```

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- **Theory**
  - **Recap last week**
- Intro Assignment 10
- Exam questions
- Kahoot

Wieso ist das TAS lock so langsam?

Locks with atomics

- Now we can implement locks for n threads using a single variable:

    - Lock: while (!TAS(l)) {}
    - Unlock: mem[l] = 0

Lets build a spinlock using RMW operations

Test and Set (TAS)

Init (lock)

```
lock = 0;
```

Acquire (lock)

```
while !TAS(lock); // wait
```

Release (lock)

```
lock = 0;
```

In Java...

```java
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    while(state.getAndSet(true)) {
      //do nothing
    }
  }

  public void unlock() {
    state.set(false);
  }

}
```

TAS Spinlock scales horribly, why?

TAS
n = 1, elapsed= 224, normalized= 224
n = 2, elapsed= 719, normalized= 359
n = 3, elapsed= 1914, normalized= 638
n = 4, elapsed= 3373, normalized= 843
n = 5, elapsed= 4330, normalized= 866
n = 6, elapsed= 6075, normalized= 1012
n = 7, elapsed= 8089, normalized= 1155
n = 8, elapsed= 10369, normalized= 1296
n = 16, elapsed= 41051, normalized= 2565
n = 32, elapsed= 156207, normalized= 4881
n = 64, elapsed= 619197, normalized= 9674

Bus Contention

- TAS/CAS are read-modify-write operations:
  - Processor assumes we modify the value even if we fail!
  - Need to invalidate cache
  - Threads serialize to read the value while spinning

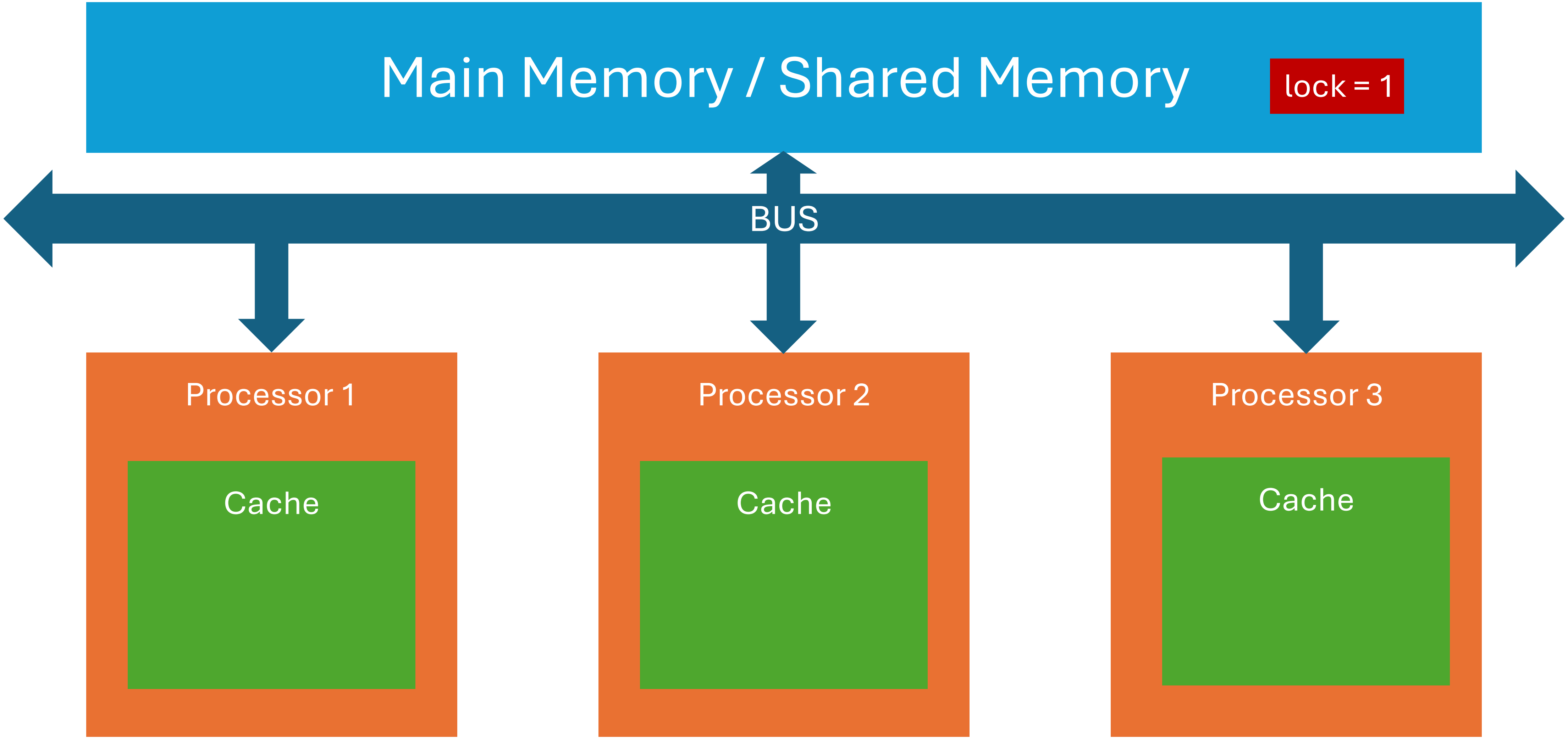
Cache Coherency Protocol ☹

We have a sequential bottleneck!

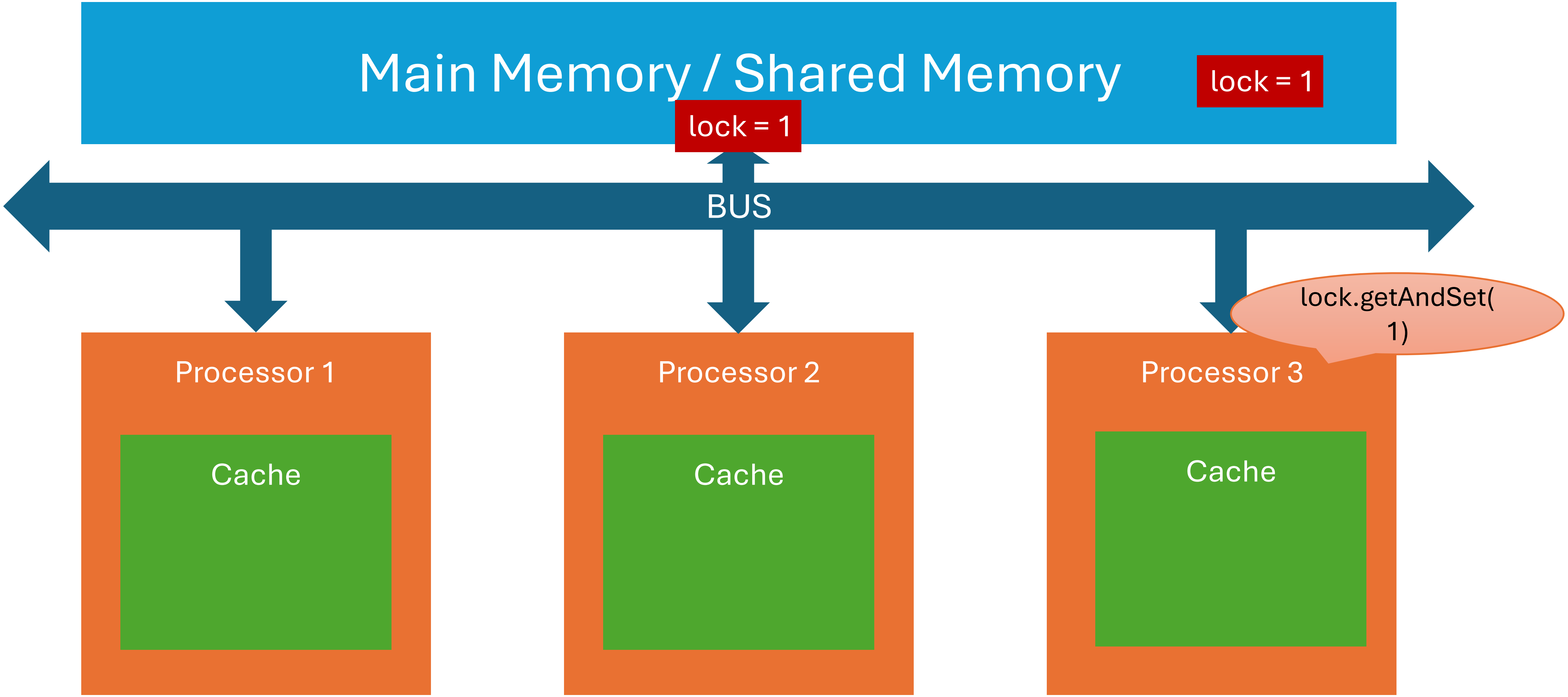
Each call to getAndSet() invalidates cached copies! => Threads need to access memory via Bus => Bus Contention!

“[…] the getAndSet() call forces other processors to discard their own cached copies of the lock, so every spinning thread encounters a cache miss almost every time, and must use the bus to fetch the new, but unchanged value.” - The Art of Multiprocessor Programming

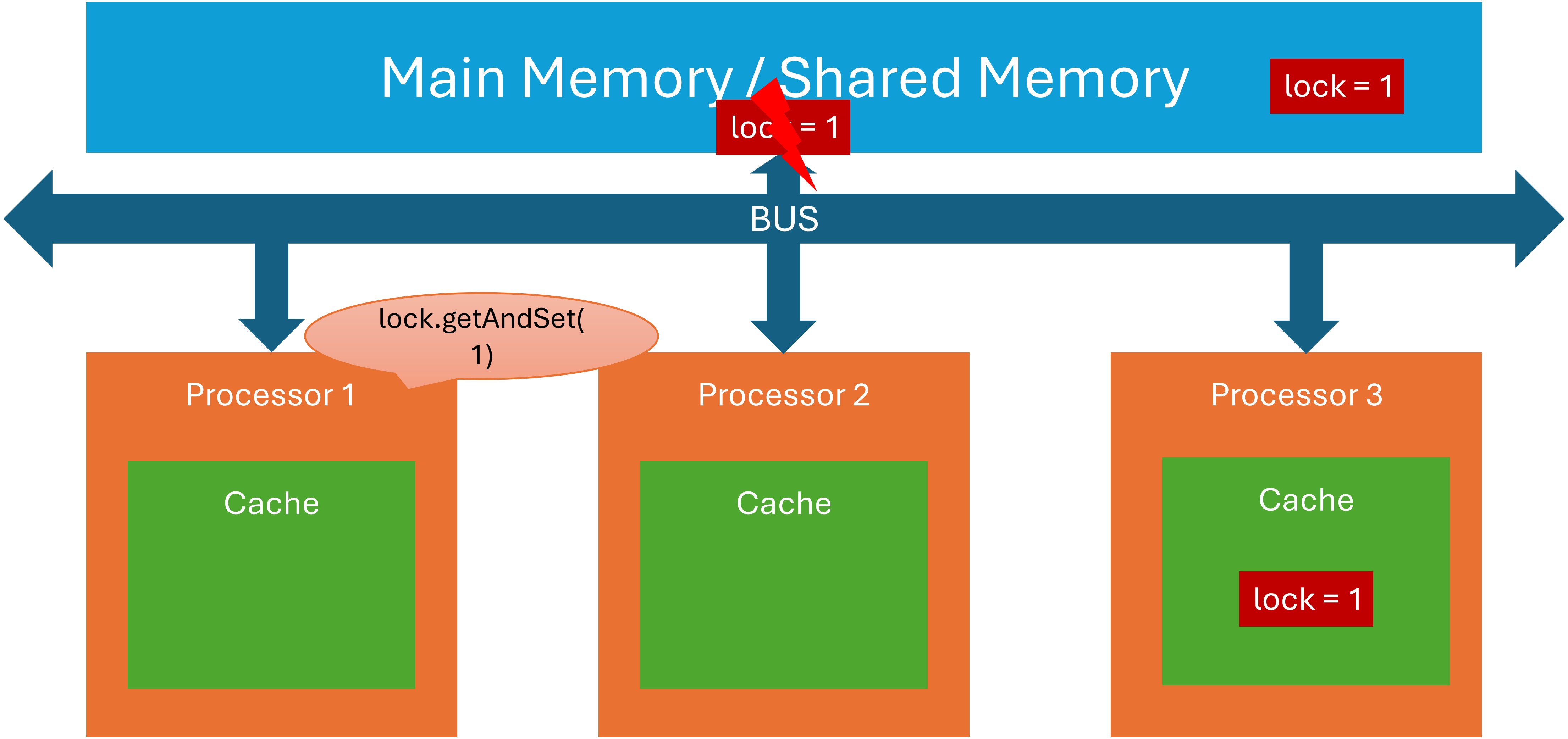
Let’s visualize this



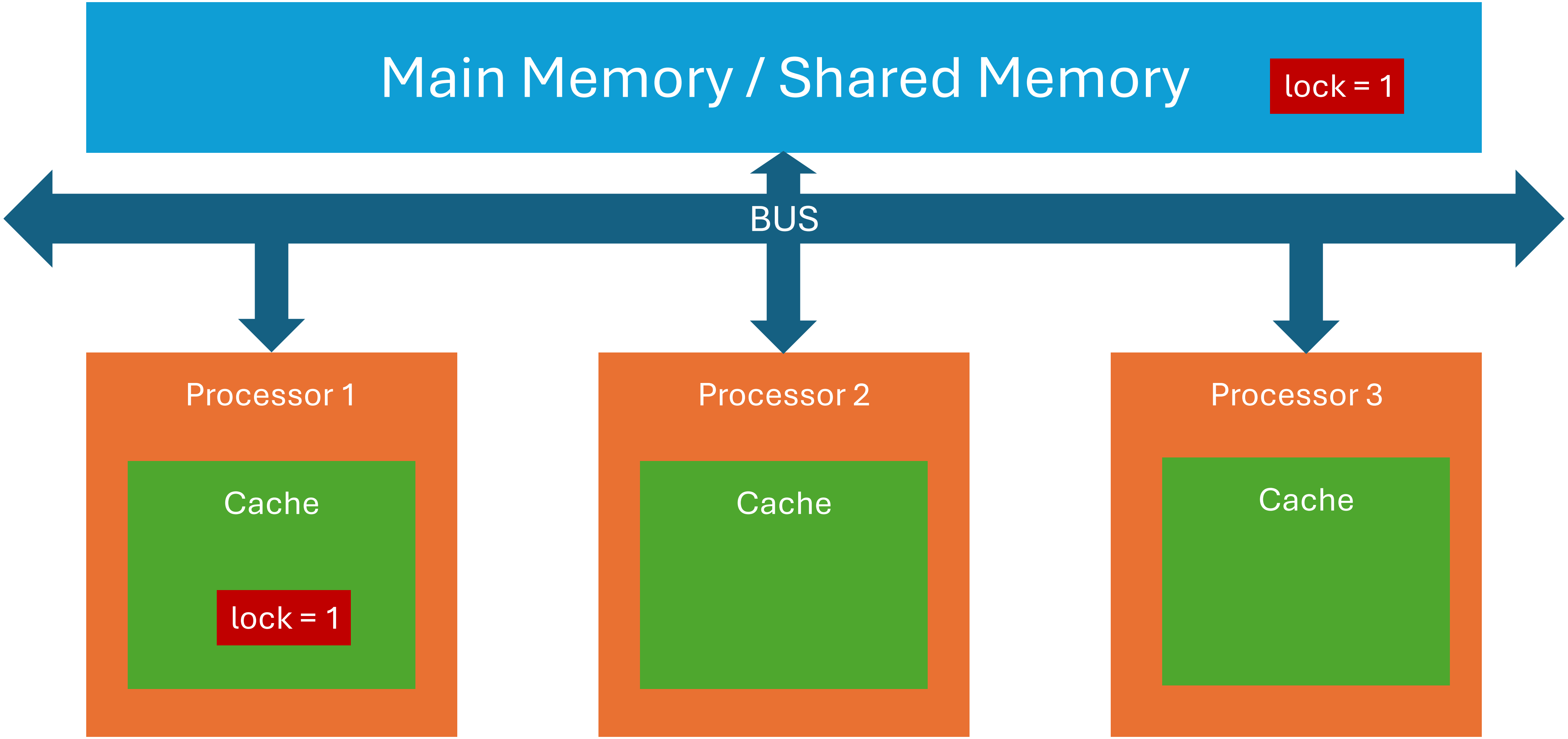
Lets visualize this



Lets visualize this



Lets visualize this



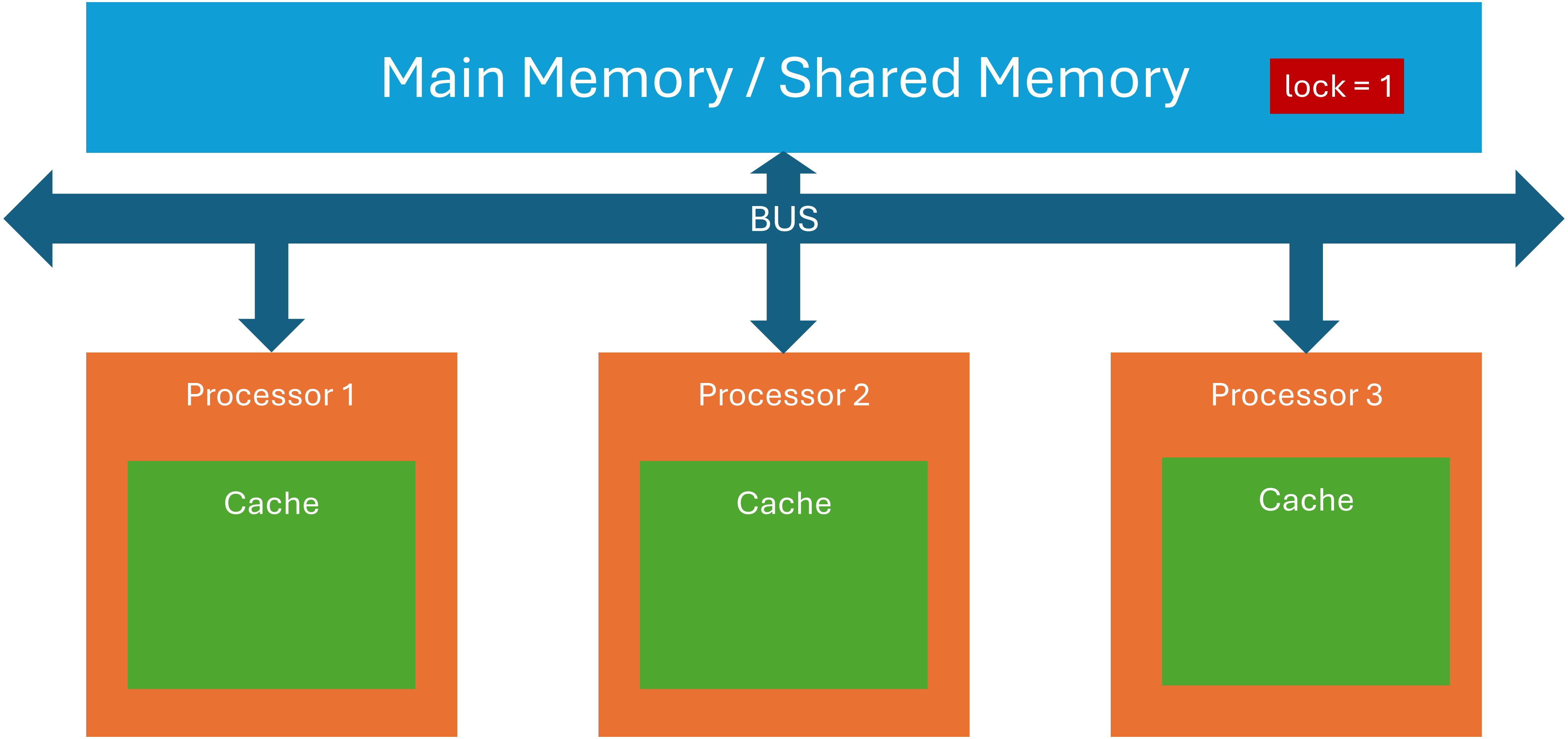
TATAS

- Idea: Use normal operation to read first, try TAS only if first read returns 0

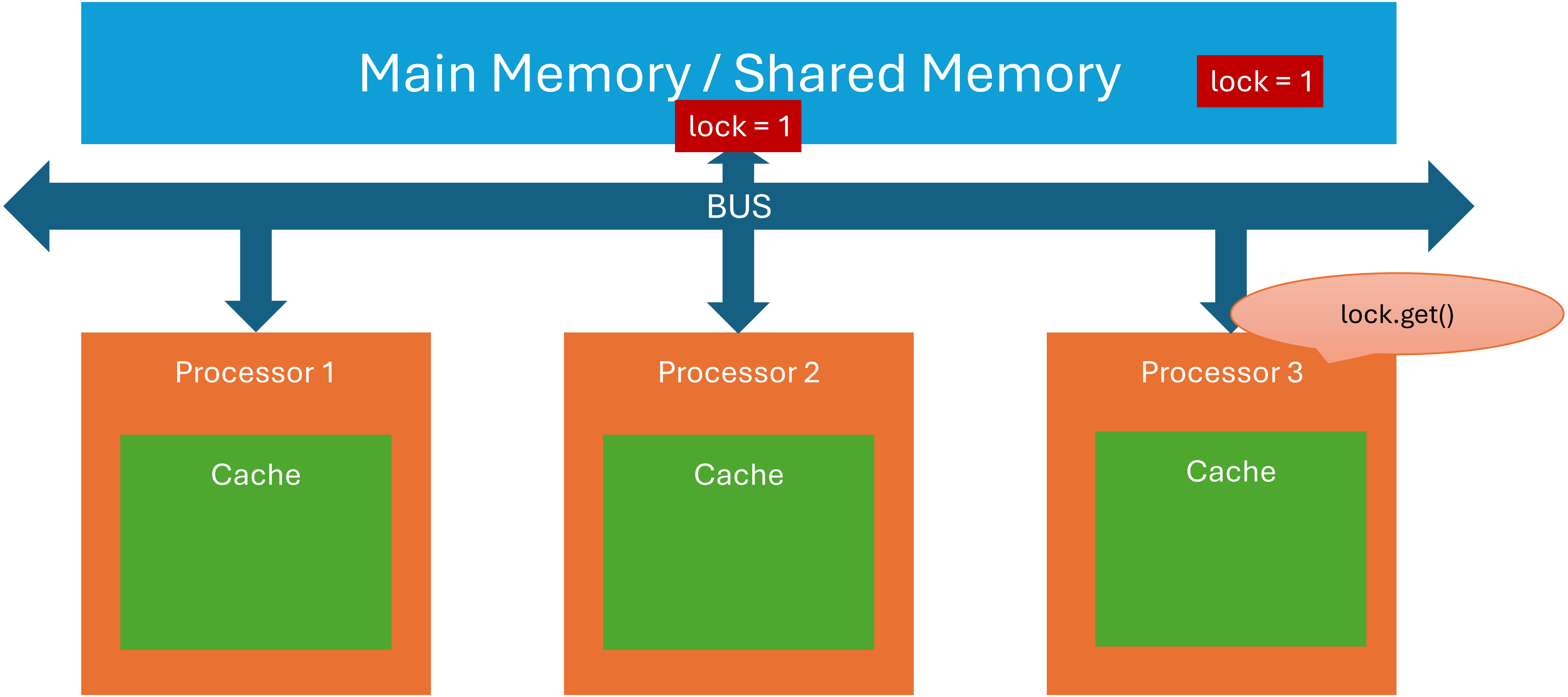
Lets try spinning on local cache

```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    do
      while (state.get() == true) //spins on local cache
    while(!state.compareAndSet(false, true)) {}
  }

  public void unlock() {
    state.set(false);
  }

}
```

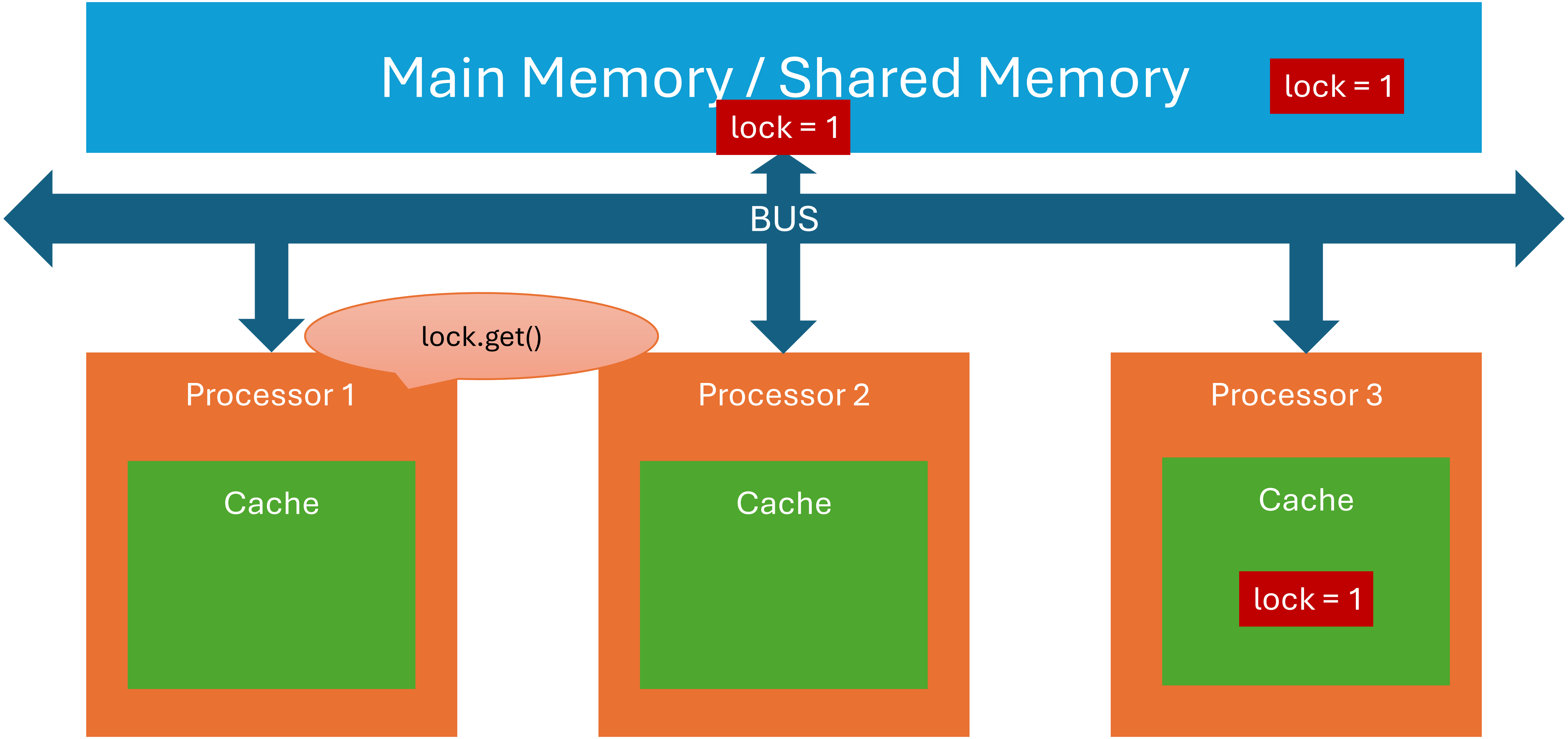
Lets visualize this



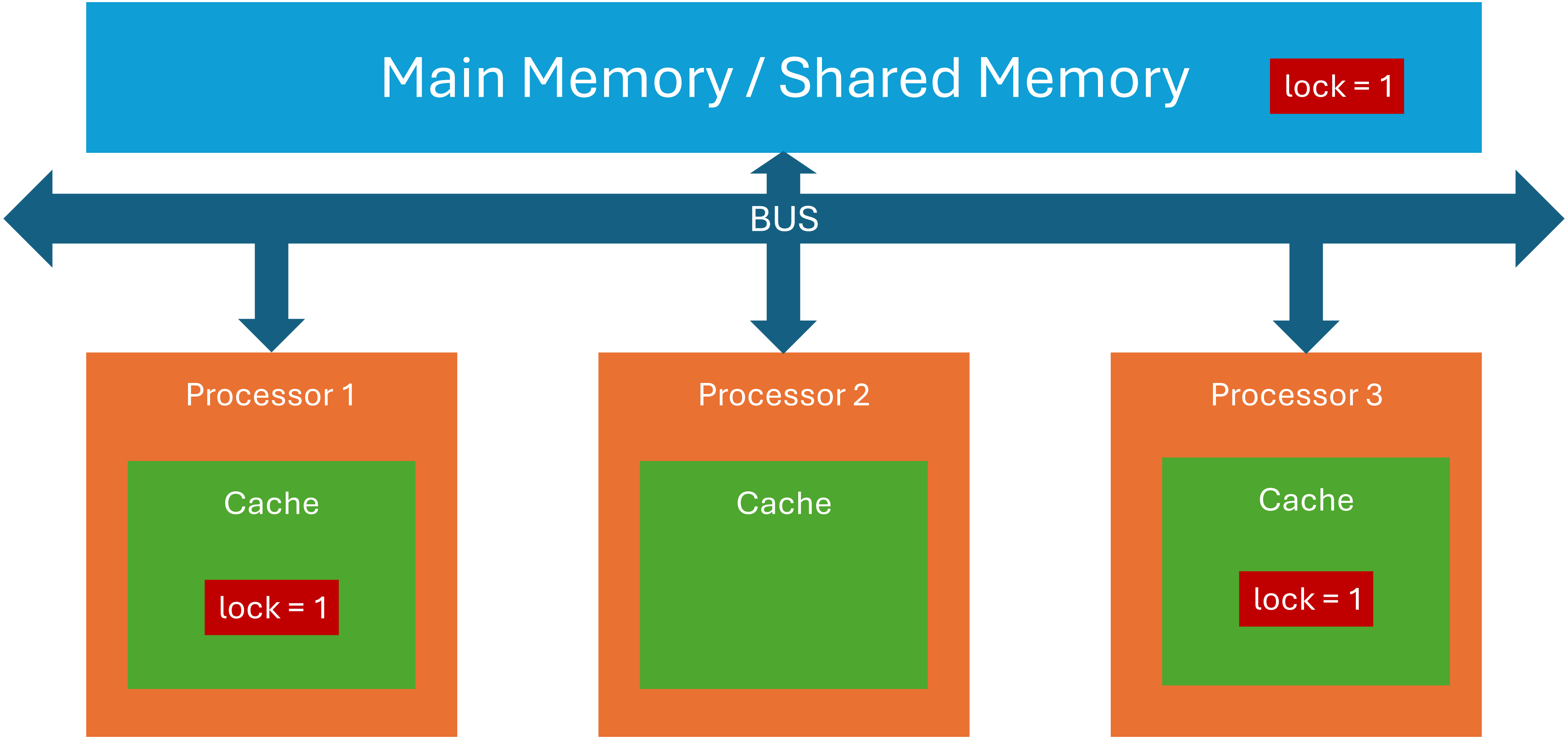
Lets visualize this



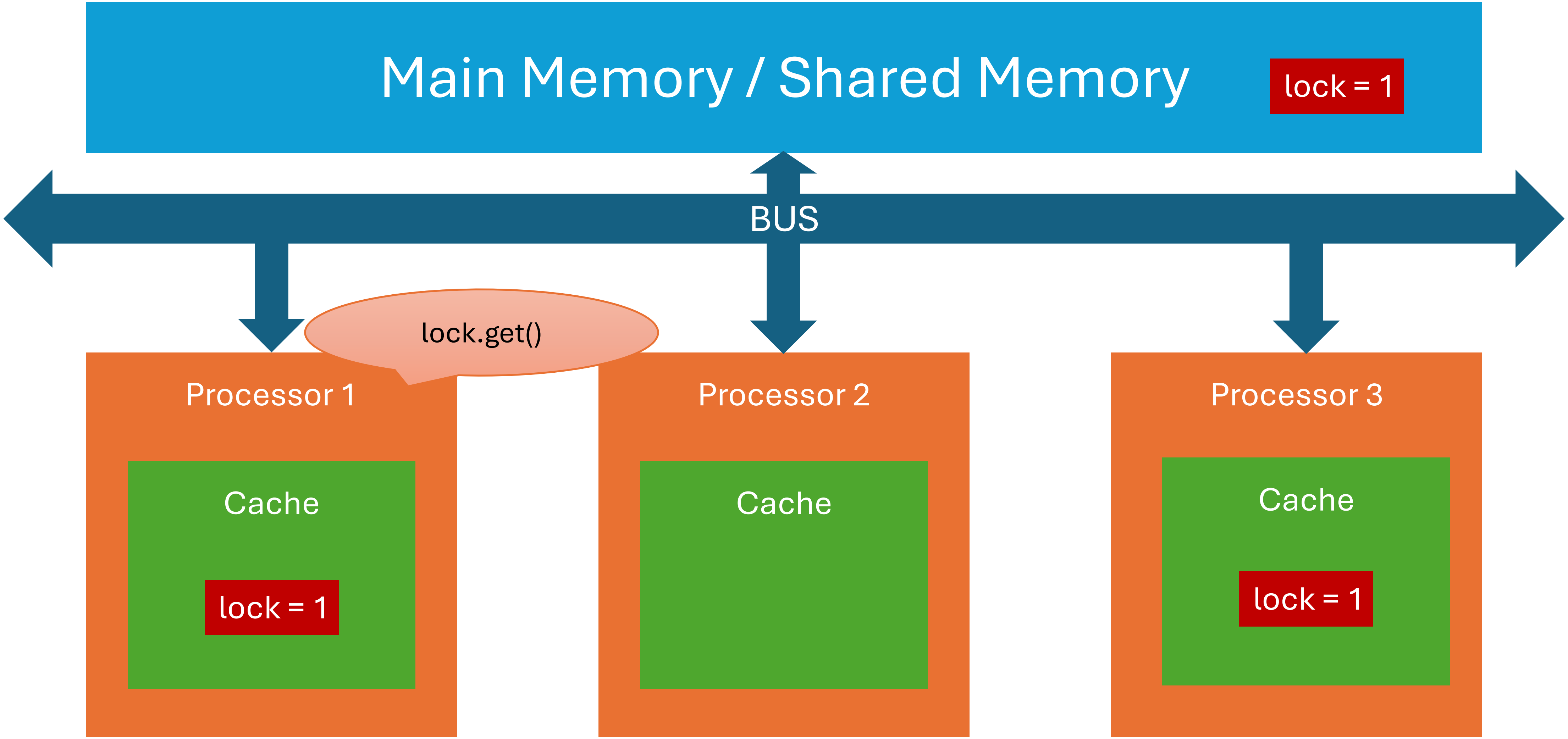
Lets visualize this



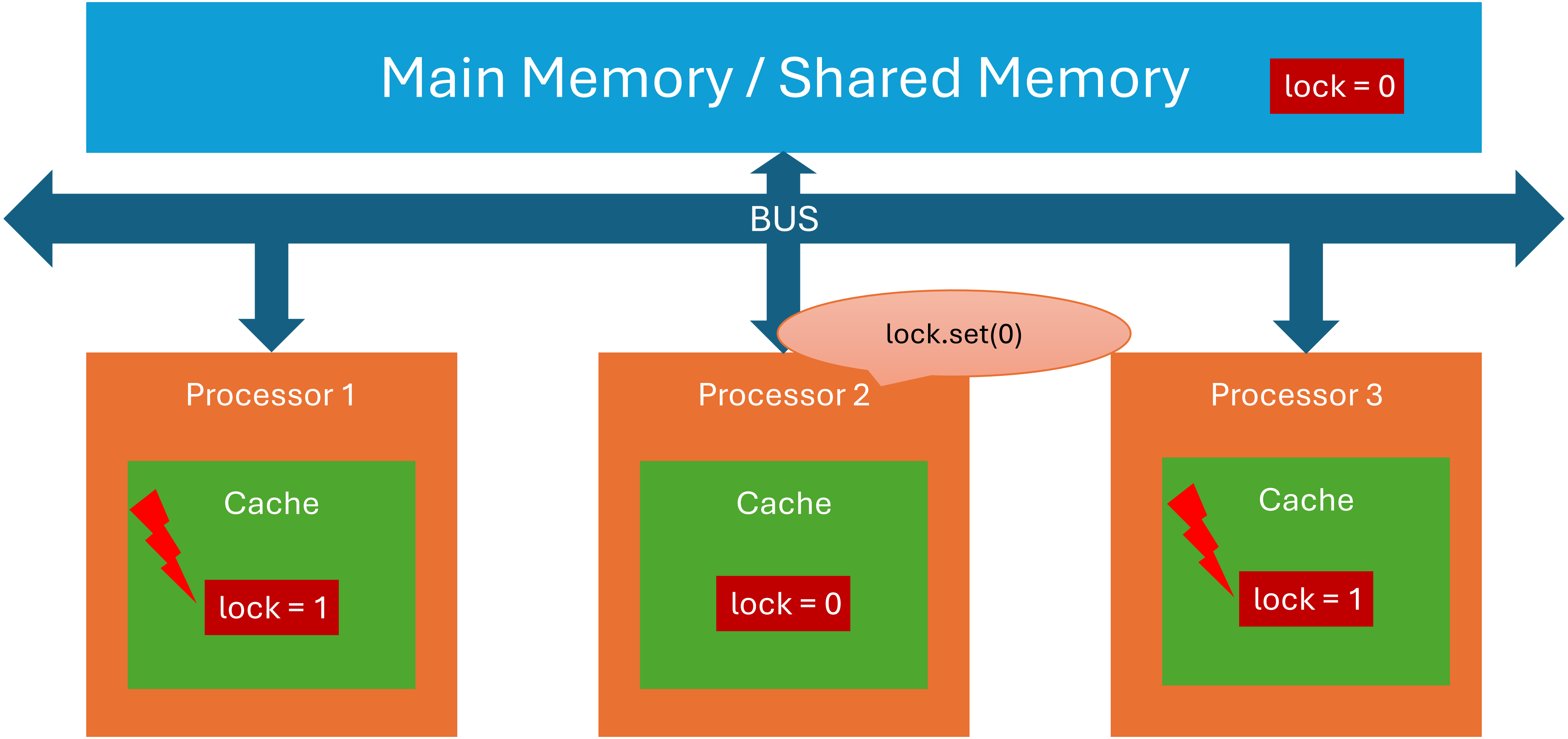
Lets visualize this



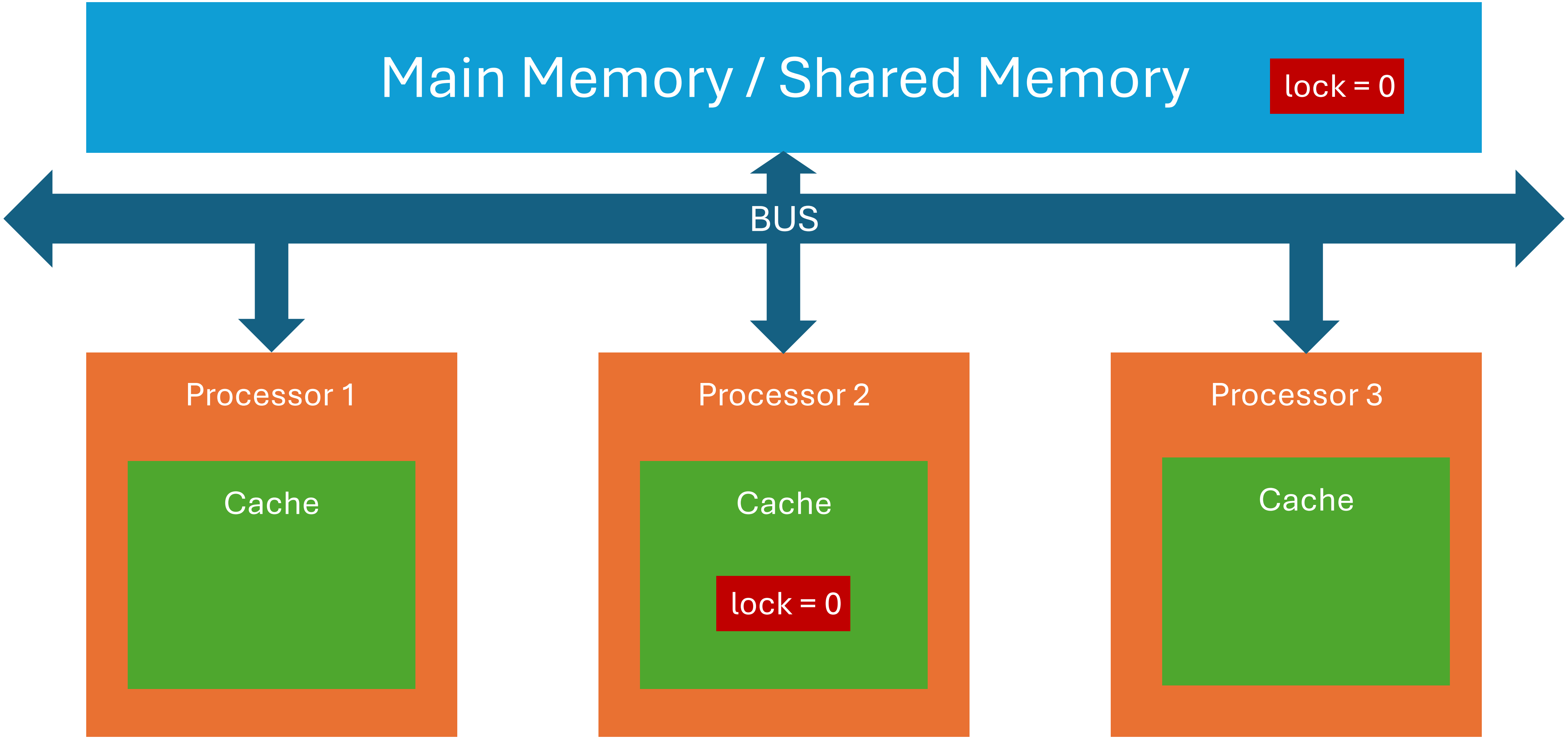
Lets visualize this



Lets visualize this

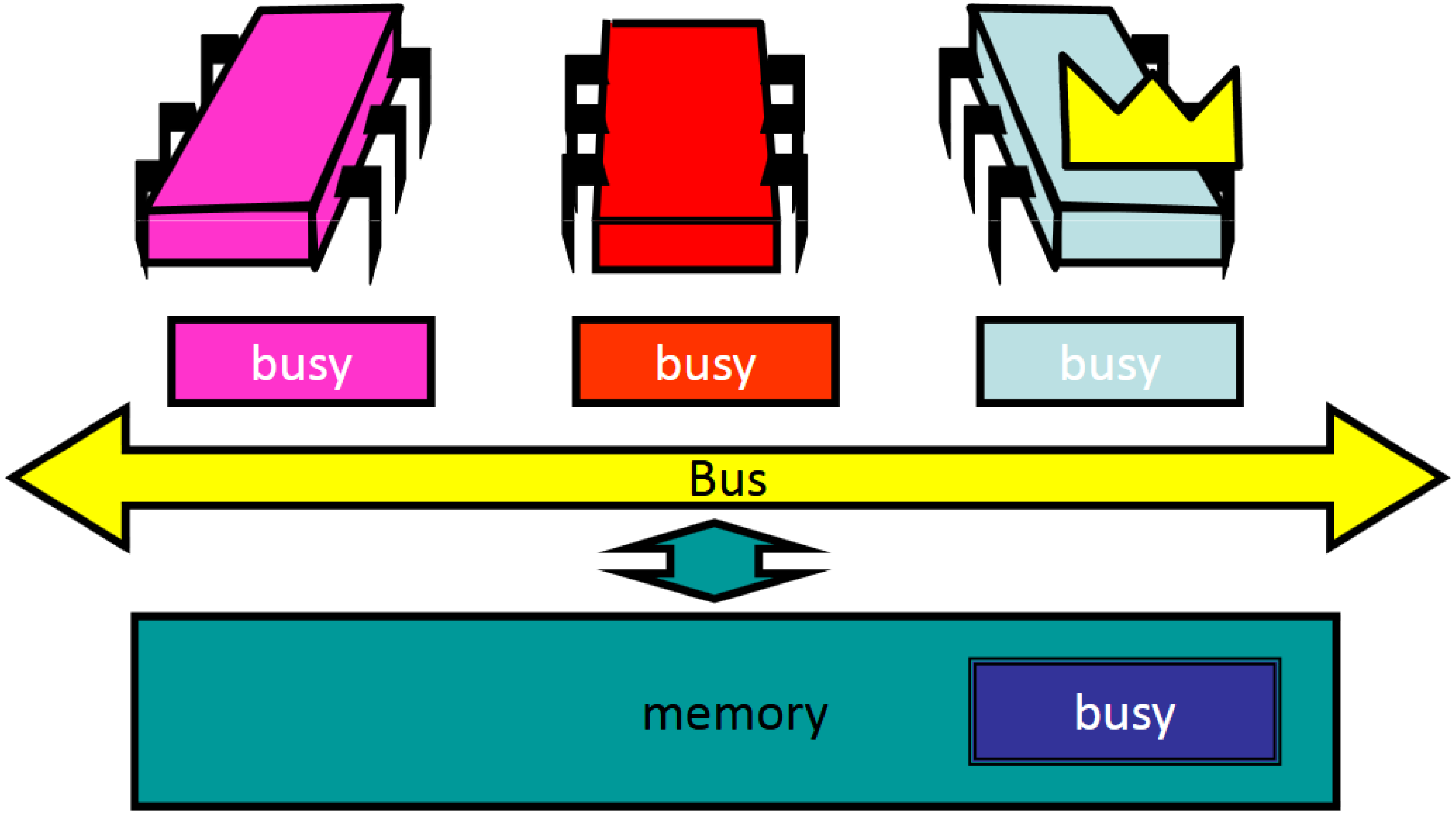


Now the whole problem repeats



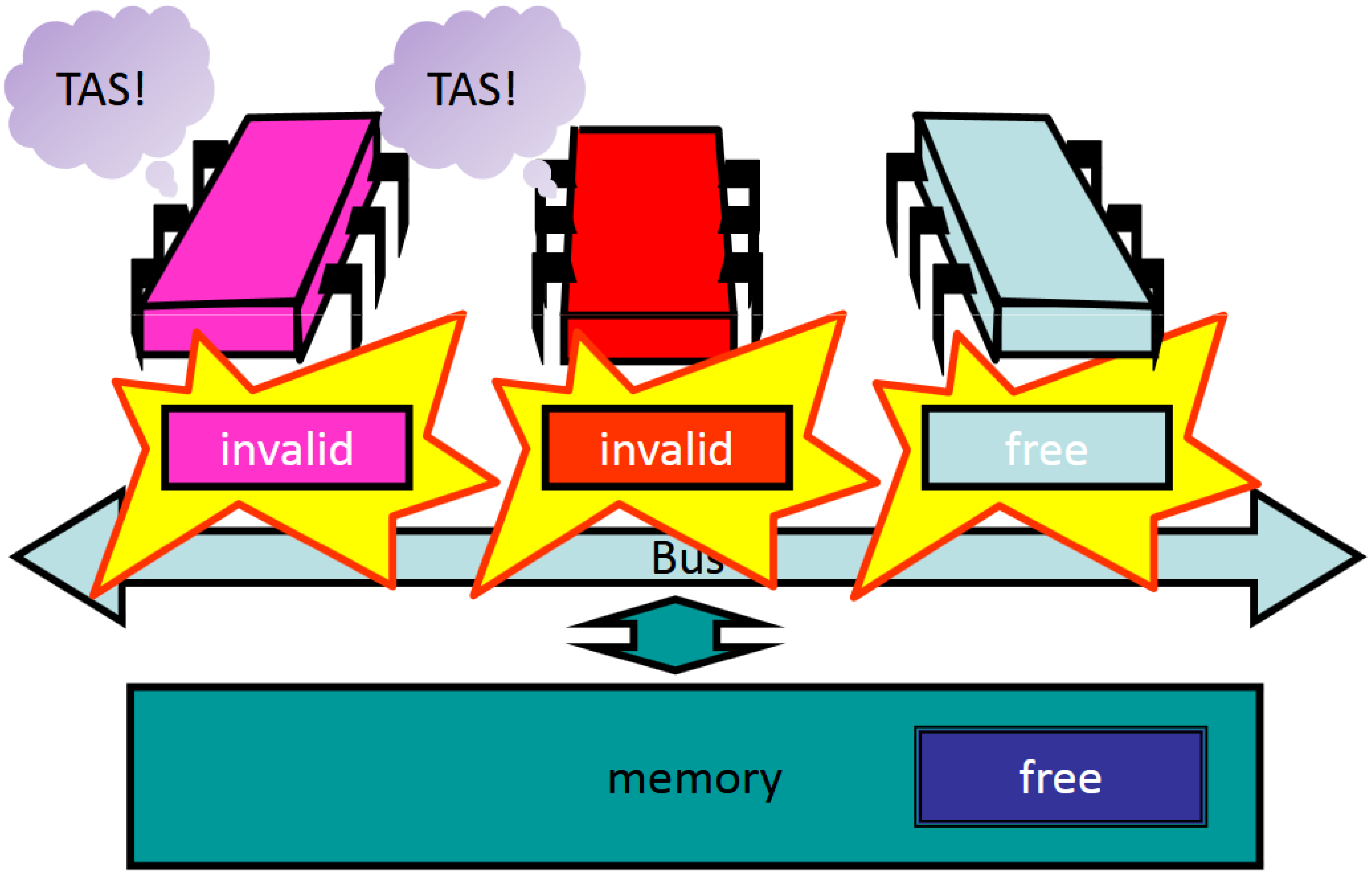
Local Spinning while Lock is Busy

- While the lock is held, all contenders spin in their caches, rereading cached data without causing any bus traffic



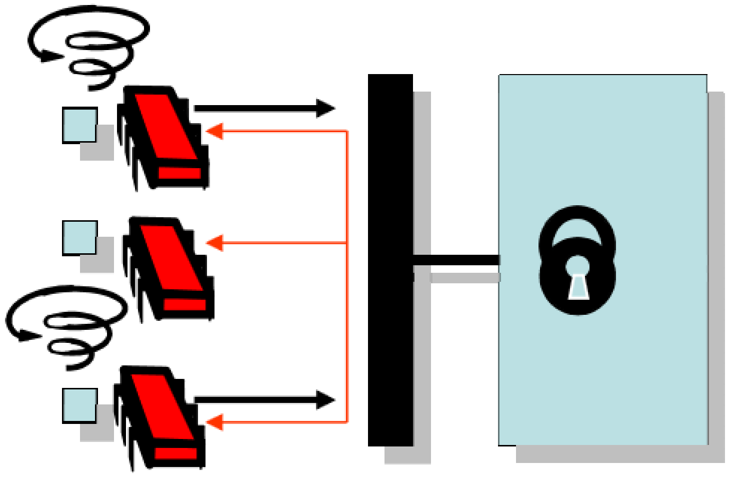
On Release

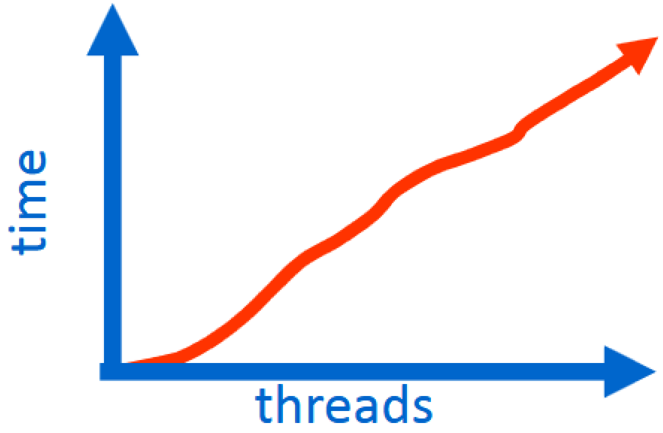
- The lock is released. All spinners take a cache miss and call Test&Set!



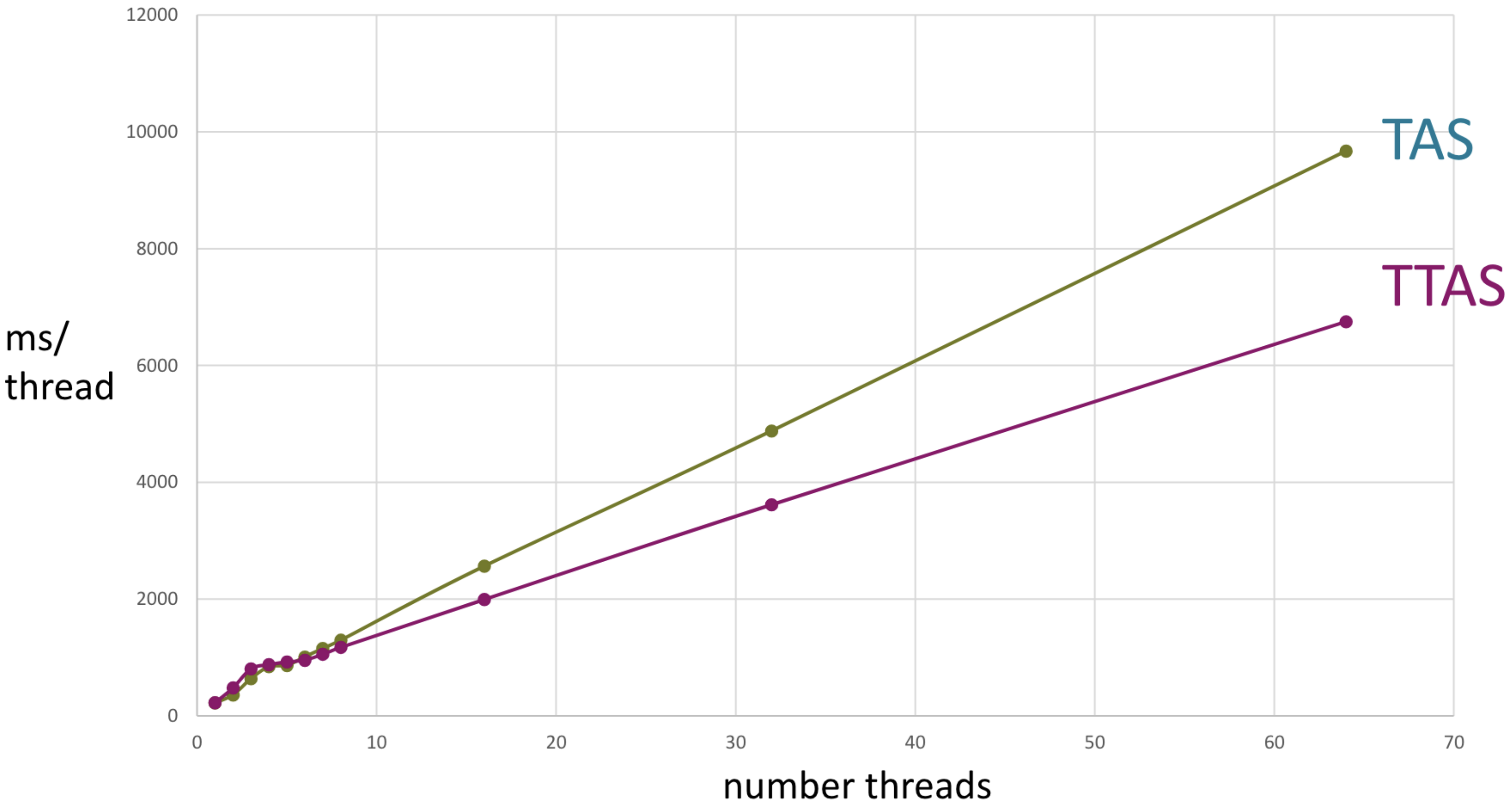
Time to Quiescence

- Every process experiences a cache miss
  - All state.get() satisfied sequentially
- Every process does TAS
  - Caches of other processes are invalidated
- Eventual quiescence ("silence") after acquiring the lock
- The time to quiescence increases linearly with the number of processors for a bus architecture!





It only helped a little bit



TAS vs. TTAS

- TAS invalidates cache lines
- Spinners
  - Always go to bus
- Thread wants to release lock
  - delayed behind spinners!!!

- TTAS waits until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

This is why TAS performs so poorly…

What we learned

- (too) many threads fight for access to the same resource
- slows down progress globally and locally
- **CAS/TAS: Processor assumes we modify the value even if we fail!**

Solution? Exponential Backoff

Idea: Each time TAS fails, wait longer until you re-try

- Backoff must be random!

Exponential Backoff

Essential

- Idea: Each time TAS fails, wait longer until you re-try

Exponential Backoff Lock

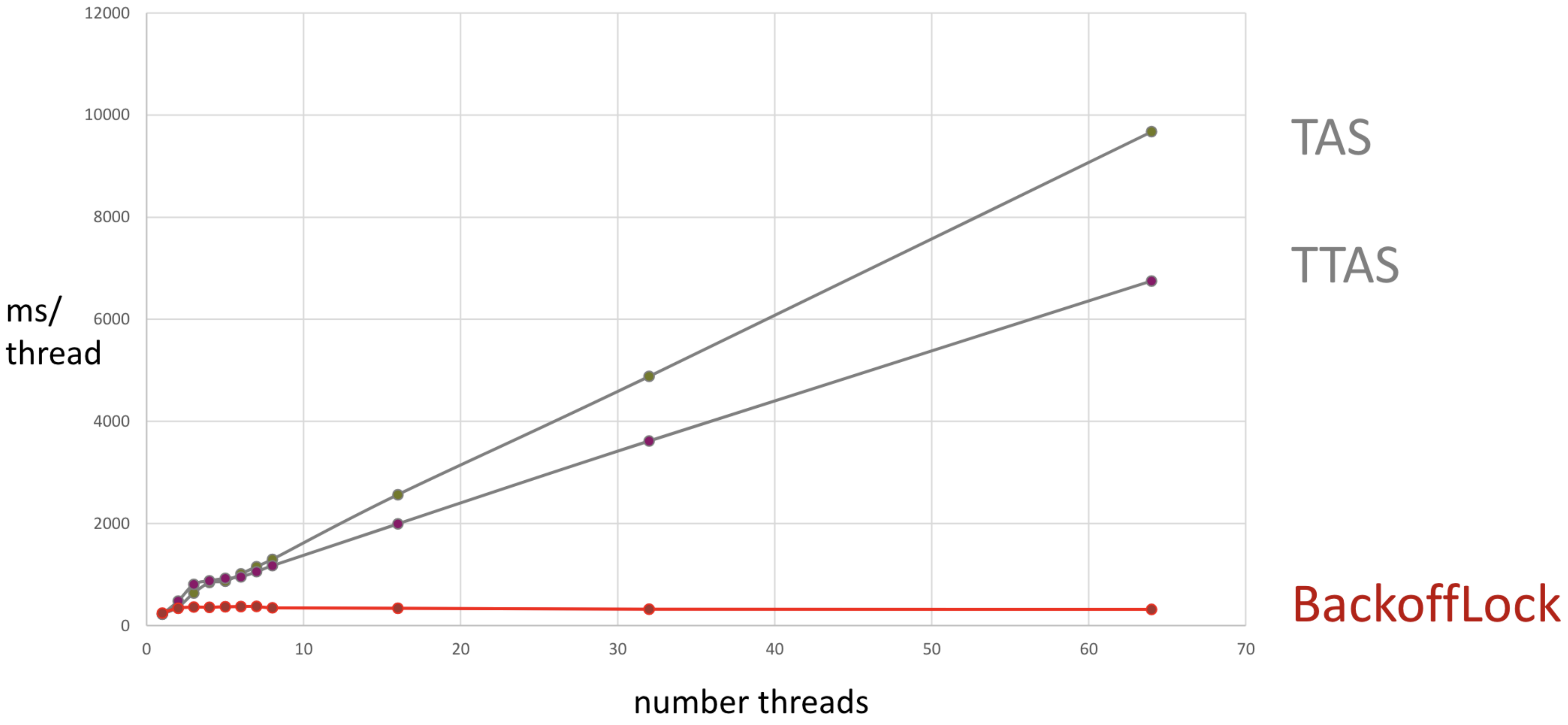
```
public class Backoff implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while(state.get()) {}
      if (!lock.getAndSet())
        return;
      sleep(random() % delay);
      if (delay < MAX_DELAY)
        delay = 2 * delay;
    }
  }

  // unlock() remains the same

}
```

Fix minimum delay

Back off for random duration

Double maximum delay until an upper bound is reached

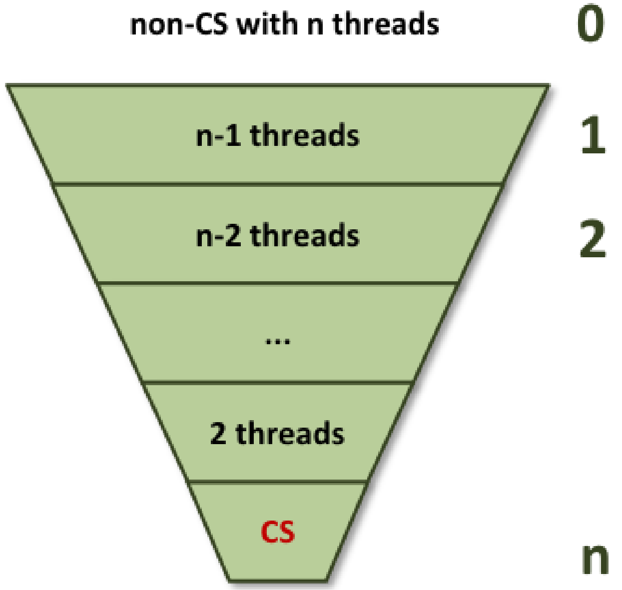
Nice!



12000
10000
8000
6000
4000
2000
0
0
10
20
30
40
50
60
70
ms/
thread
number threads
TAS
TTAS
BackoffLock

Locks für n Threads

Filter Lock




```
int[] level(#threads), int[] victim(#threads)

lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```

Other threads are at same or higher level

And I have to wait

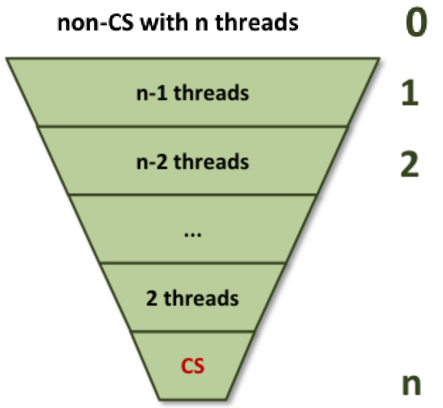
Filter Lock

- Ist das Filter Lock fair (first come first served)?

```
int[] level(#threads), int[] victim(#threads)


lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```

Other threads are at same or higher level

And I have to wait



Bakery Lock

```
integer array[0..n-1] label = [0, ..., 0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) <l (me, label[me])) {};

unlock(me):
   flag[me] = false;
```

$$(k, l_k) <_l (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$$

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- **Theory**
- Intro Assignment 10
- Exam questions
- Kahoot

Semaphores

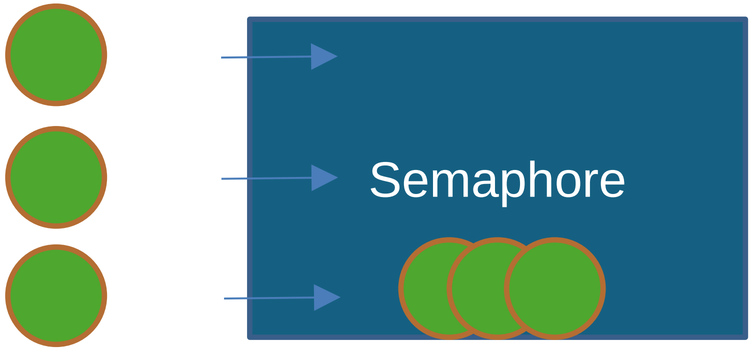
and Barriers

Lecture Recap: Semaphores

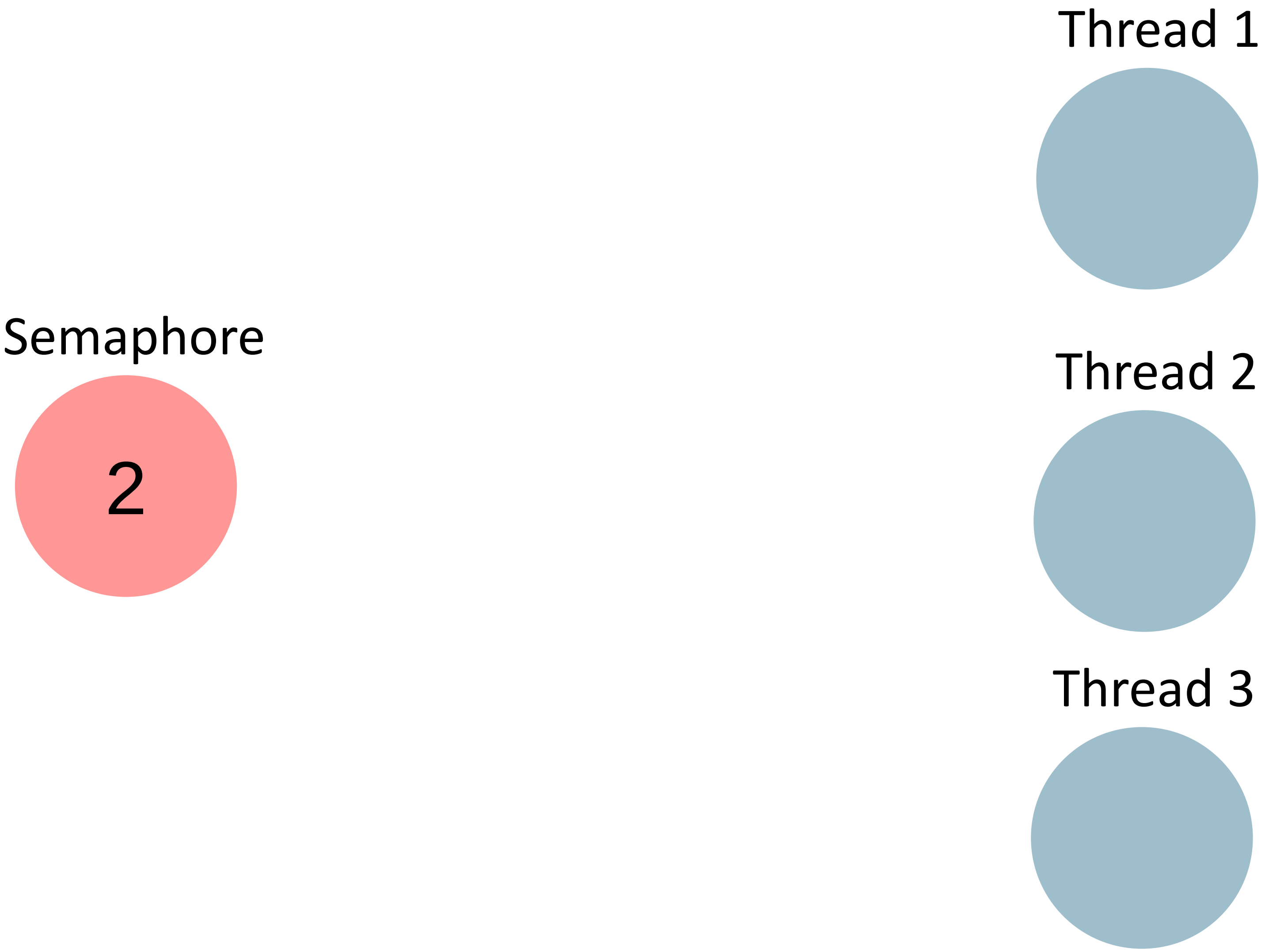
Used to restrict the number of threads that can access a specific resource.

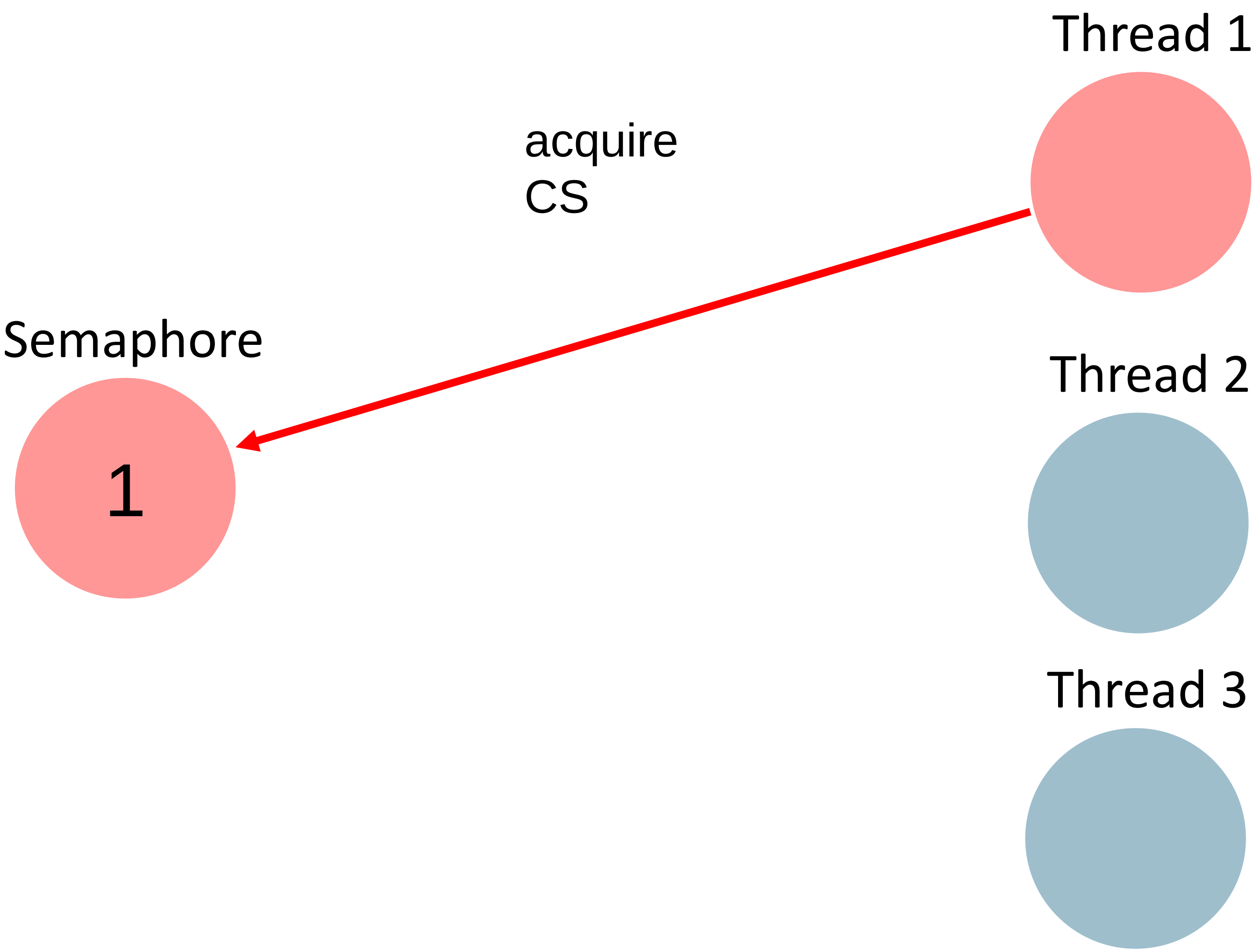
- acquire() gets a permit, if no permit available block
- release() gives up permit, releases a blocking acquirer

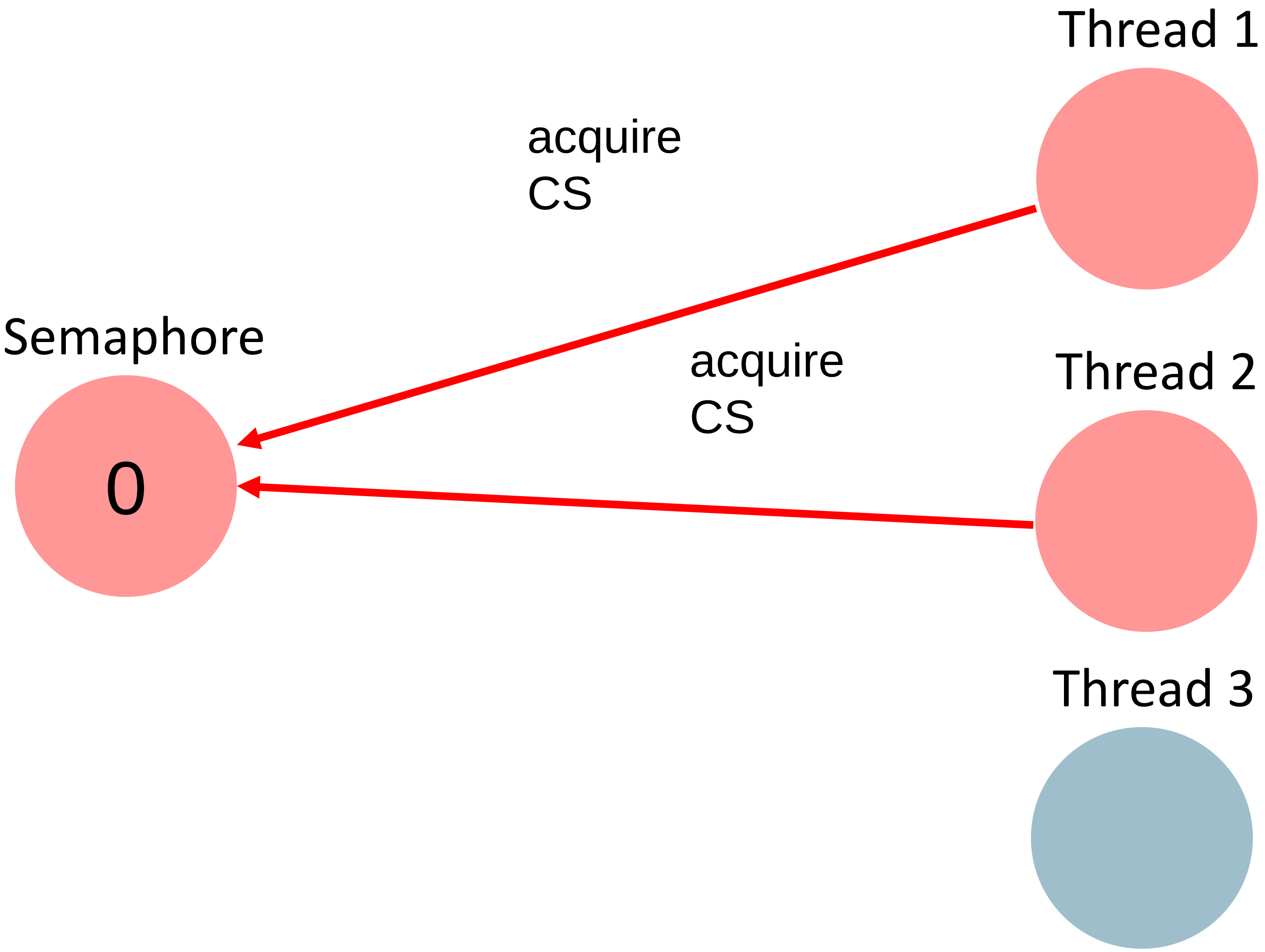
Lecture Recap: Semaphores

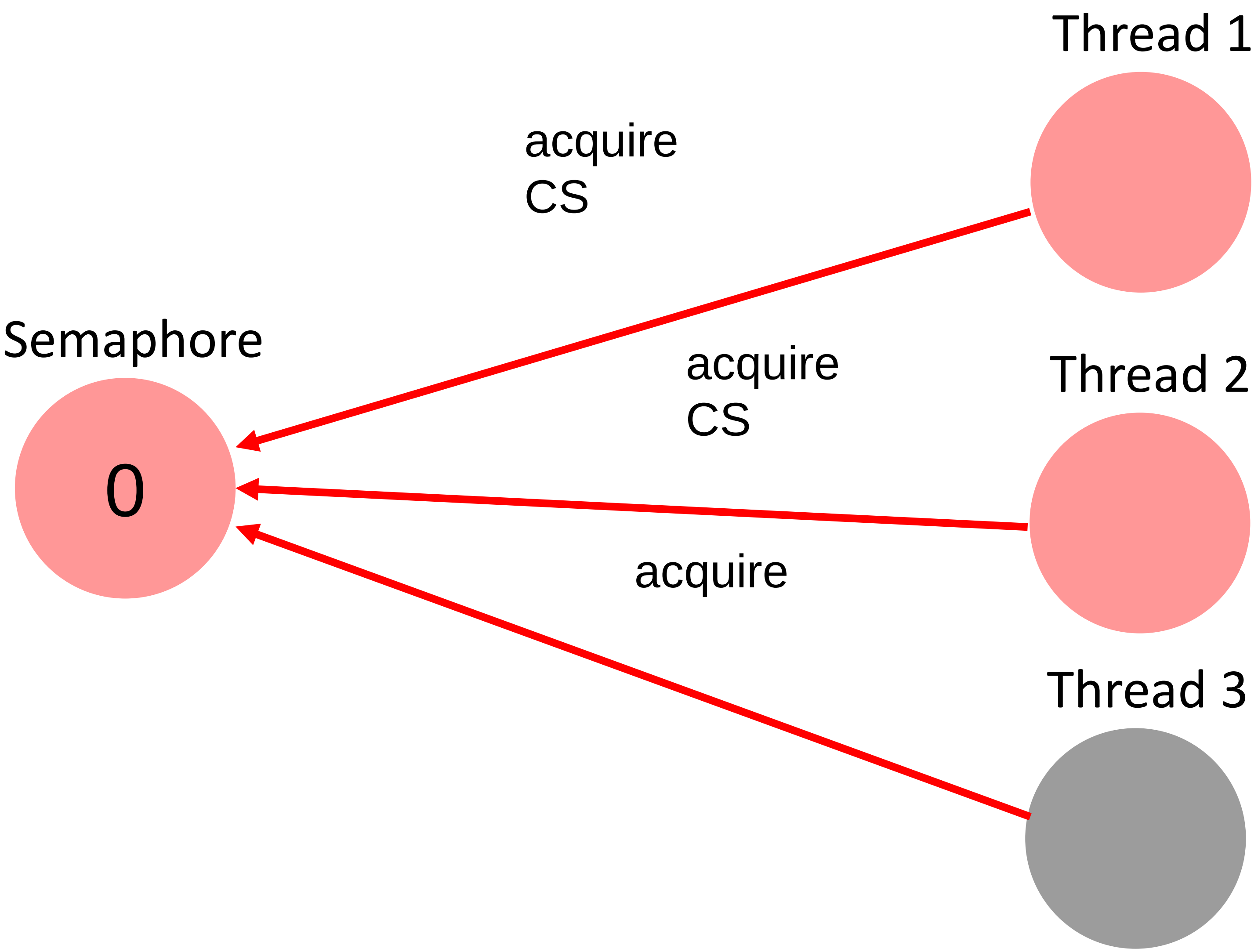


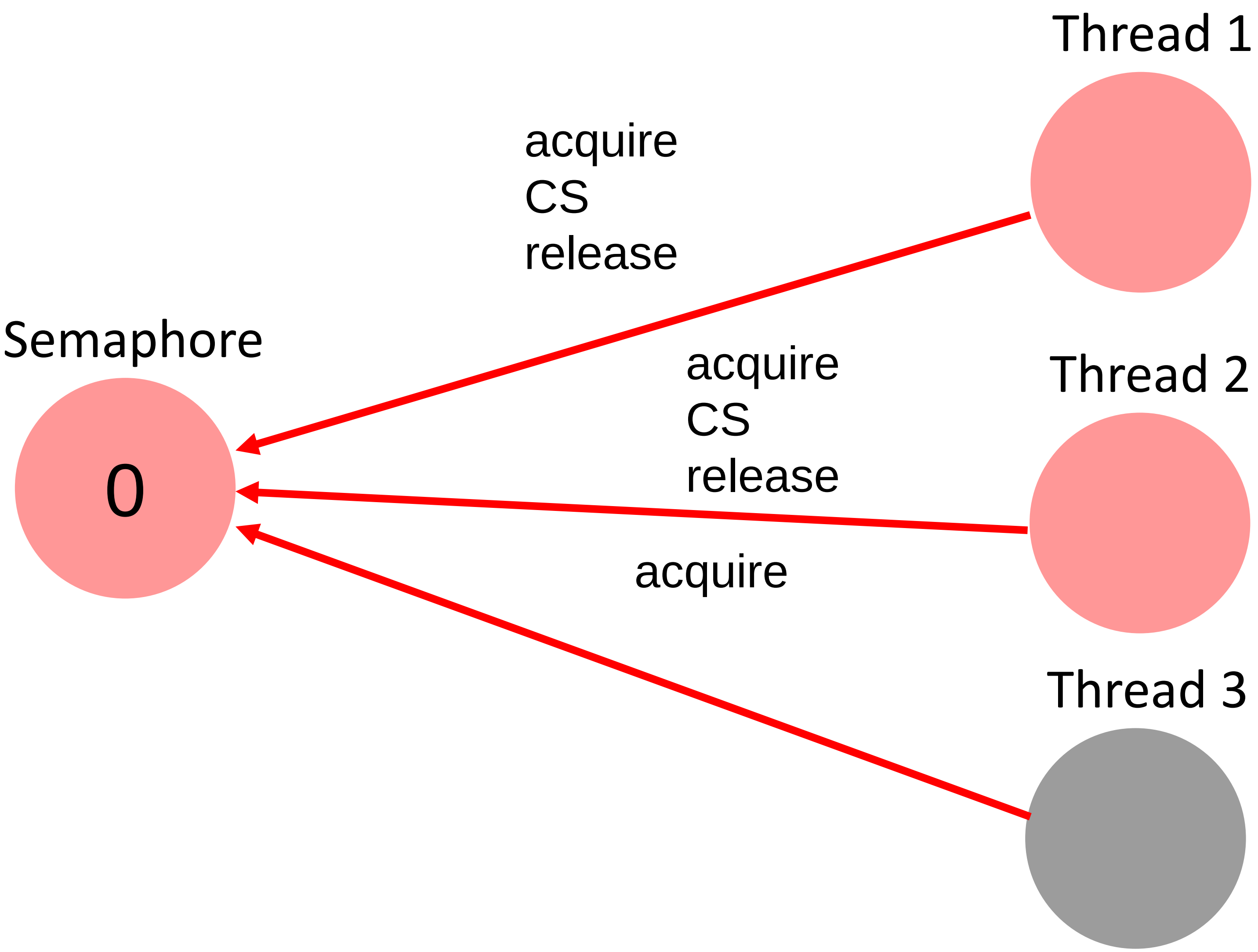
N Threads have permit to a semaphore,
others will wait (blocked) until someone leaves the semaphore

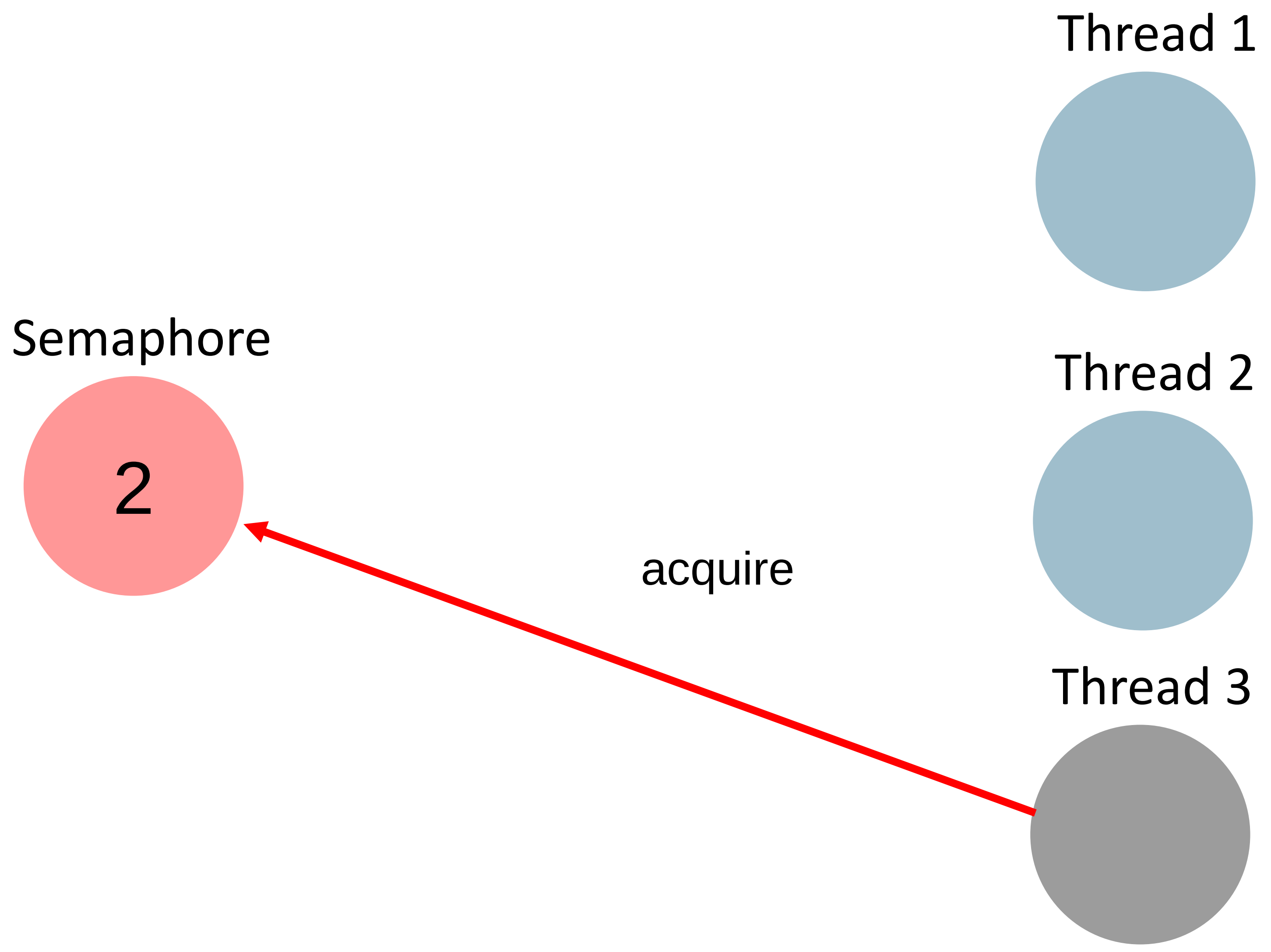

Thread 1
Semaphore
2
Thread 2
Thread 3

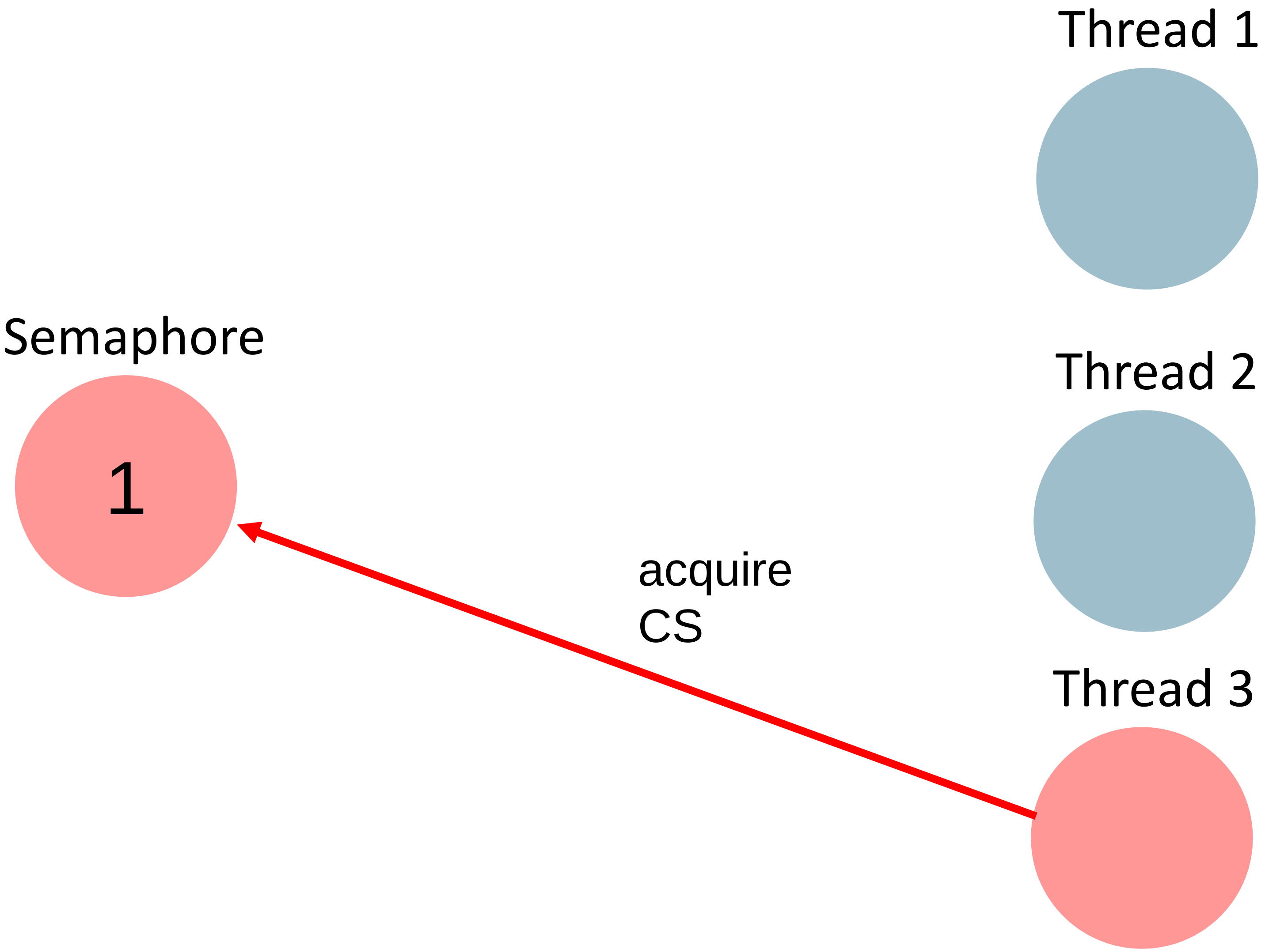

Thread 1
acquire
CS
Semaphore
1
Thread 2
Thread 3


Thread 1
acquire
CS
Semaphore
acquire
CS
Thread 2
0
Thread 3


Thread 1
acquire
CS
Semaphore
0
acquire
CS
Thread 2
acquire
Thread 3


Thread 1
acquire
CS
release
Semaphore
0
acquire
CS
release
Thread 2
acquire
Thread 3


Thread 1
Semaphore
2
Thread 2
acquire
Thread 3



Think of semaphores as bike rentals

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value s≥0 and the following **atomic** operations:

```
acquire(S) {
	wait until S > 0
	dec(S)
}

release(S) {
	inc(S)
}
```

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value s≥0 and the following **atomic** operations:

```
acquire(S) {
       wait until S > 0
       dec(S)
}

release(S) {
       inc(S)
}
```

What is the difference between a Lock and a Semaphore?

Building a lock with Semaphores

mutex = Semaphore(1);

lock mutex := mutex.acquire()

only one thread is allowed into the critical section

unlock mutex := mutex.release()

one other thread will be let in

Semaphore number:

1 → unlocked

0 → locked

x>0 → x threads will be let into "critical section"

Semaphores aren’t Locks!

- We can build Locks with Semaphores
- Some key differences:
  - More than one Thread can be in critical section!
  - How many depends on the number of permits
  - **Threads can release() a Semaphore without accquiring before!**
  - The is no notion of “holding” a Semaphore as we have with ”holding” Locks

Semaphores: Implementation

Semaphore: integer-valued abstract data type S with some initial value s≥0 and the following **atomic** operations:

```
acquire(S) {
       wait until S > 0
       dec(S)
}

release(S) {
       inc(S)
}
```

When would you use a semaphore?

Semaphores

- Locks provide means to enforce atomicity via mutual exclusion
- They lack the means for threads to communicate about changes
- We need something stronger to coordinate threads (e.g. to implement rendezvous)

Semaphores: Usage example

```
class Pool {
        private static final int MAX_AVAILABLE = 100;
        private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

        public Object getItem() throws InterruptedException {
          available.acquire();
          return getNextAvailableItem();
        }

        public void putItem(Object x) {
          if (markAsUnused(x))
            available.release();
        }

        //...
```

Semaphores: Usage example

```
  protected Object[] items = new Object[MAX_AVAILABLE];
  protected boolean[] used = new boolean[MAX_AVAILABLE];

  protected synchronized Object getNextAvailableItem() {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (!used[i]) {
         used[i] = true;
         return items[i];
      }
    }
    return null; // not reached
  }

  protected synchronized boolean markAsUnused(Object item) {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (item == items[i]) {
         if (used[i]) {
           used[i] = false;
           return true;
         } else
           return false;
      }
    }
    return false;
  }
}
```

Semaphores

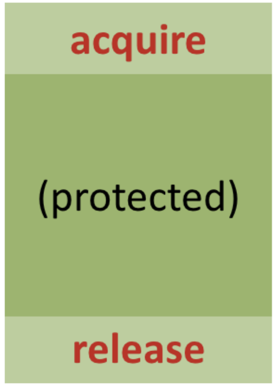
S = new Semaphore(n) - create a new semaphore with n permits

atomic

```
acquire(S)
{
   wait until S > 0
   dec(S)
}
```

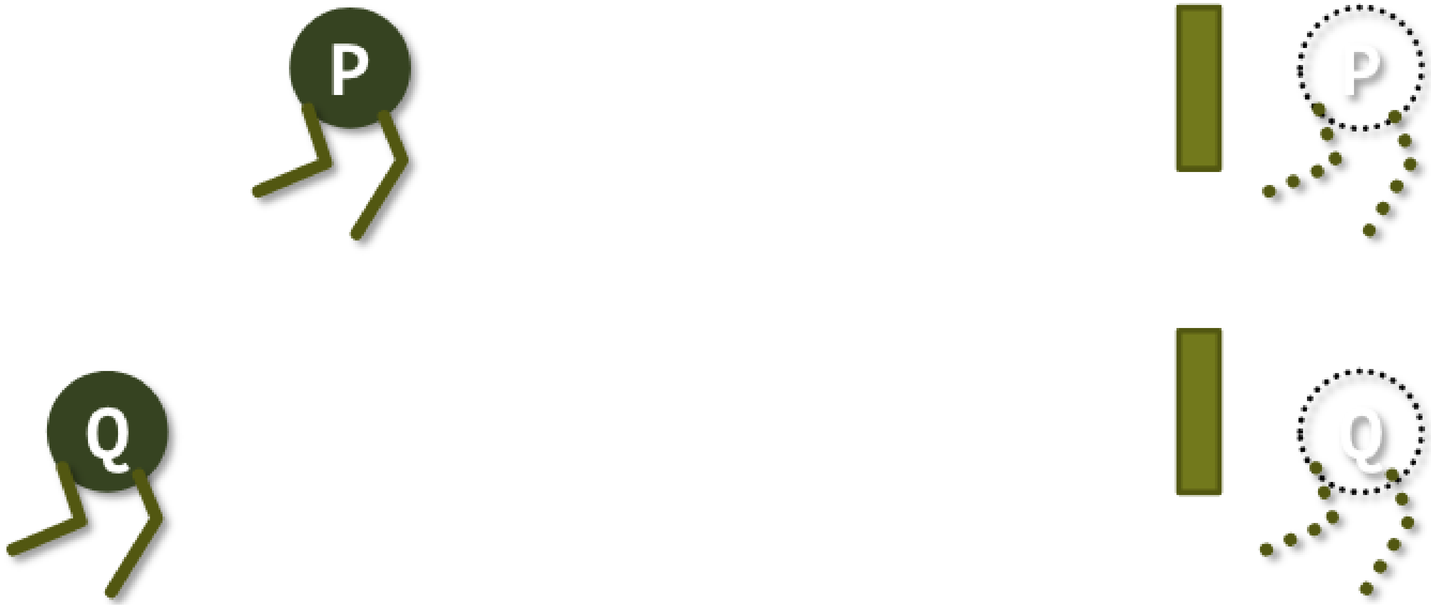
atomic

```
release(S)
{
   inc(S)
}
```



Rendezvous with Semaphores

- Two processes P and Q execute code
- Rendezvous: locations in code, where P and Q wait for the other to arrive. Synchronize P and Q.



First attempt, whats wrong?

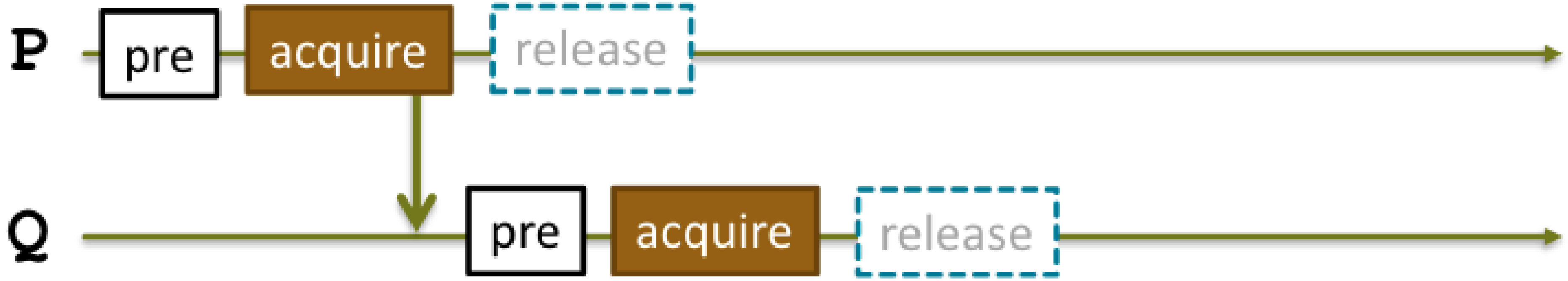
Synchronize Processes P and Q at one location (Rendezvous)

Semaphores `P_Arrived` and `Q_Arrived`

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	acquire(Q_Arrived) release(P_Arrived)	acquire(P_Arrived) release(Q_Arrived)
post	...	...

Deadlock :(

We are never able to release! Both P and Q wait endlessly for each other ☹



Attempt two, better?

Synchronize Processes P and Q at one location (Rendezvous)

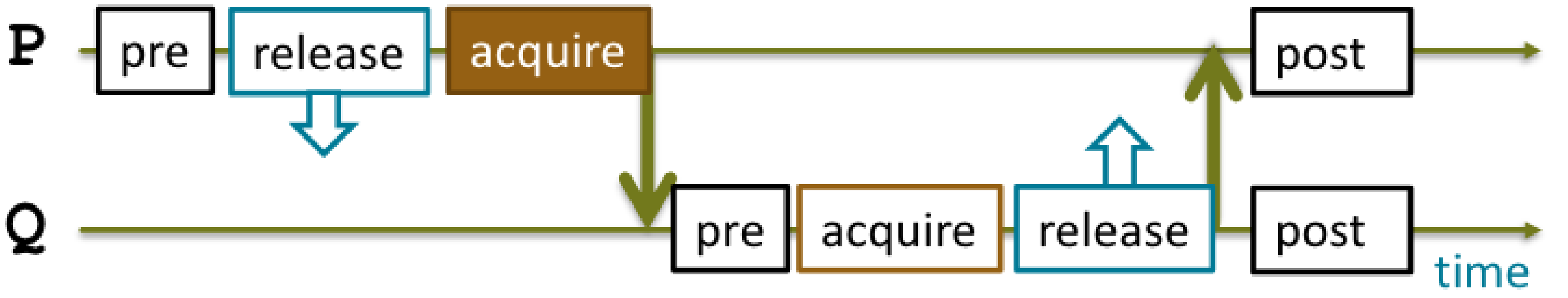
Assume Semaphores P_Arrived and Q_Arrived

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	release(P_Arrived) acquire(Q_Arrived)	acquire(P_Arrived) release(Q_Arrived)
post	...	..

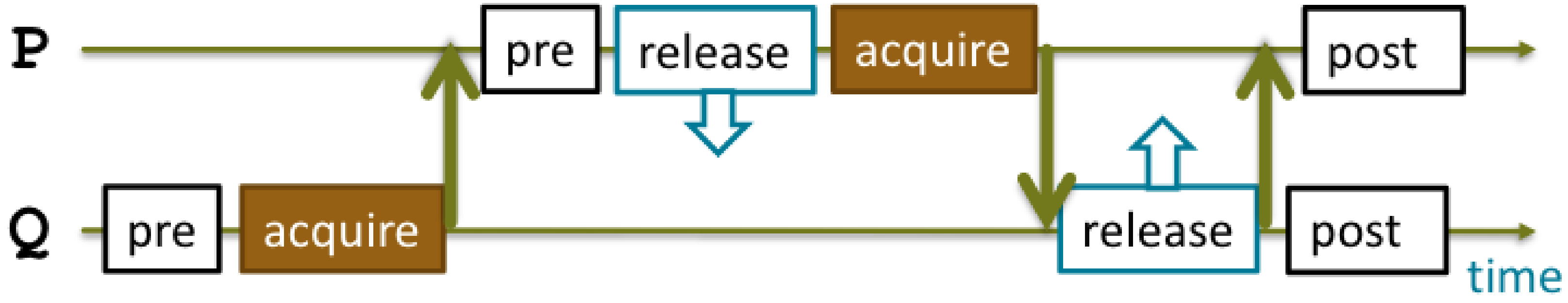
Yes, that works!

	P	Q
init	`P_Arrived=0`	`Q_Arrived=0`
pre	`...`	`...`
rendezvous	`release(P_Arrived)` `acquire(Q_Arrived)`	`acquire(P_Arrived)` `release(Q_Arrived)`
post	`...`	`..`

P first



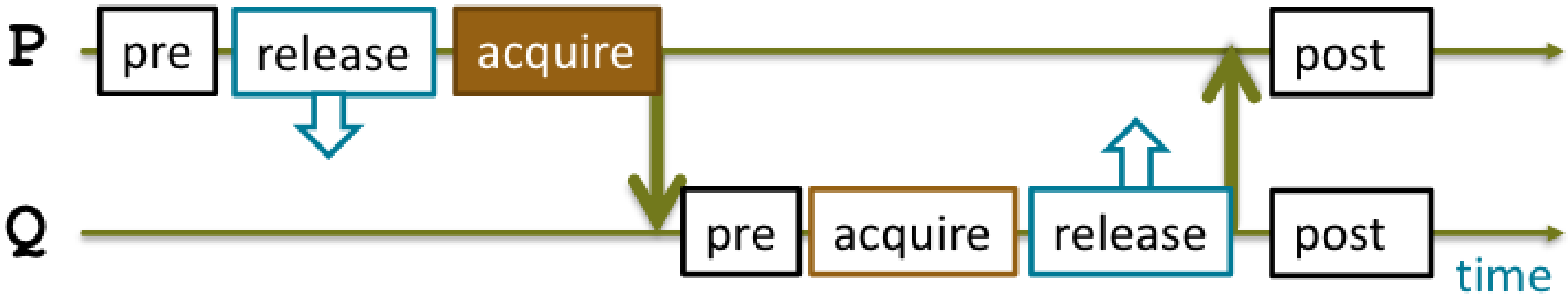
Q first



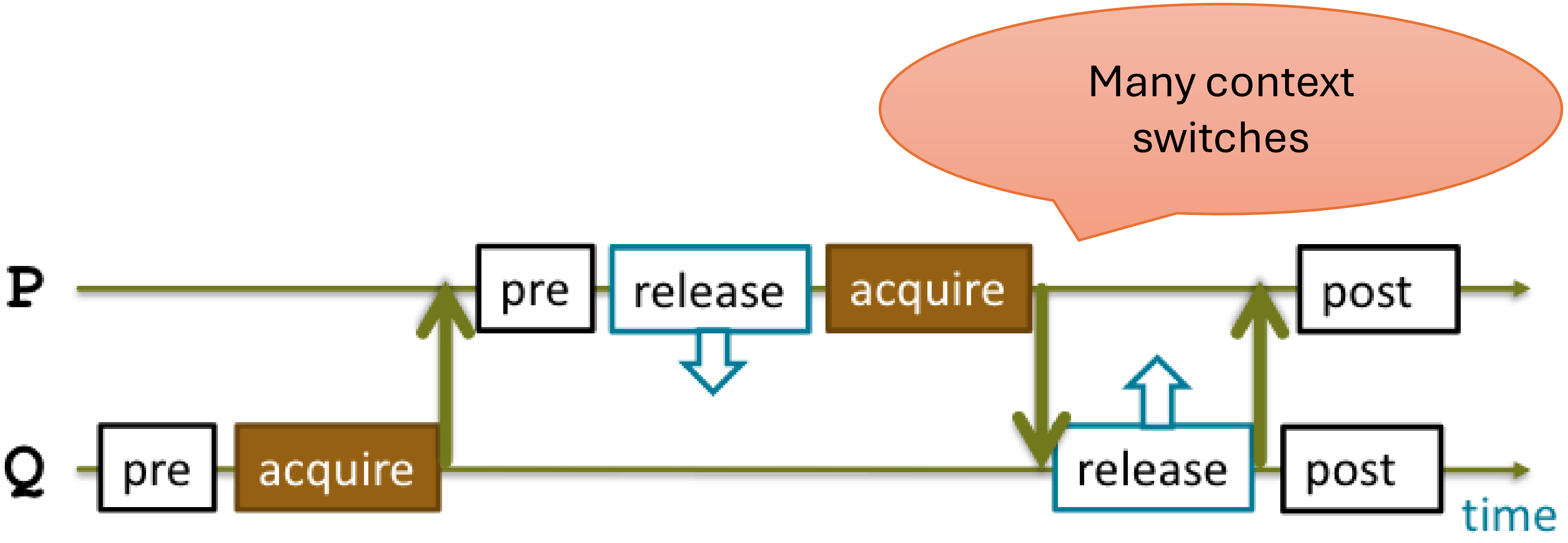
Yes, that works!

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	release(P_Arrived) acquire(Q_Arrived)	acquire(P_Arrived) release(Q_Arrived)
post	...	..

P first



Q first



Lets do better!

Synchronize Processes P and Q at one location (Rendezvous)

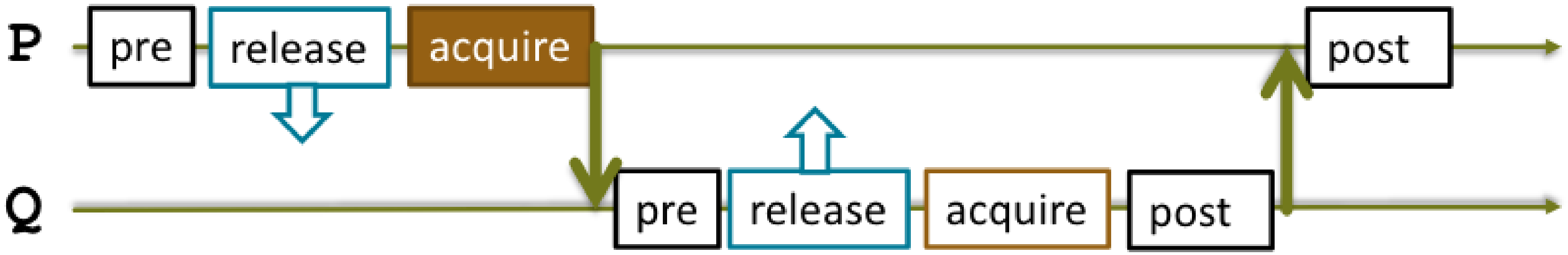
Assume Semaphores P_Arrived and Q_Arrived

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
post	...	..

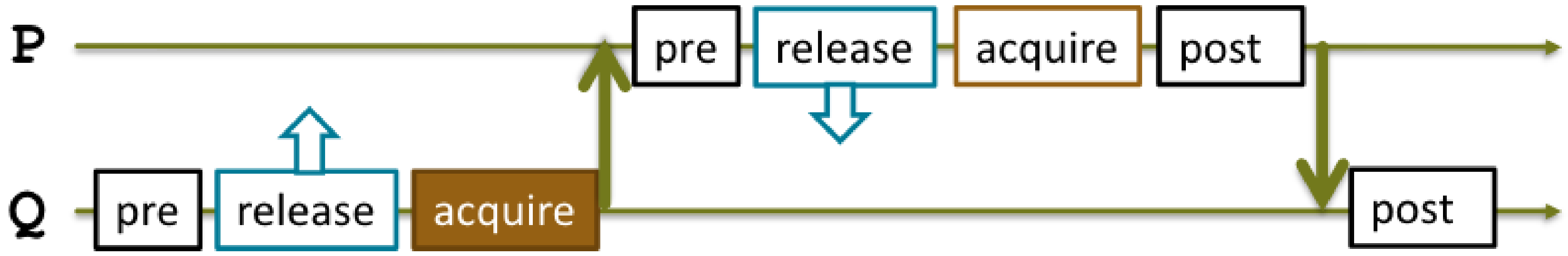
Order does no longer matter

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
post	...	..

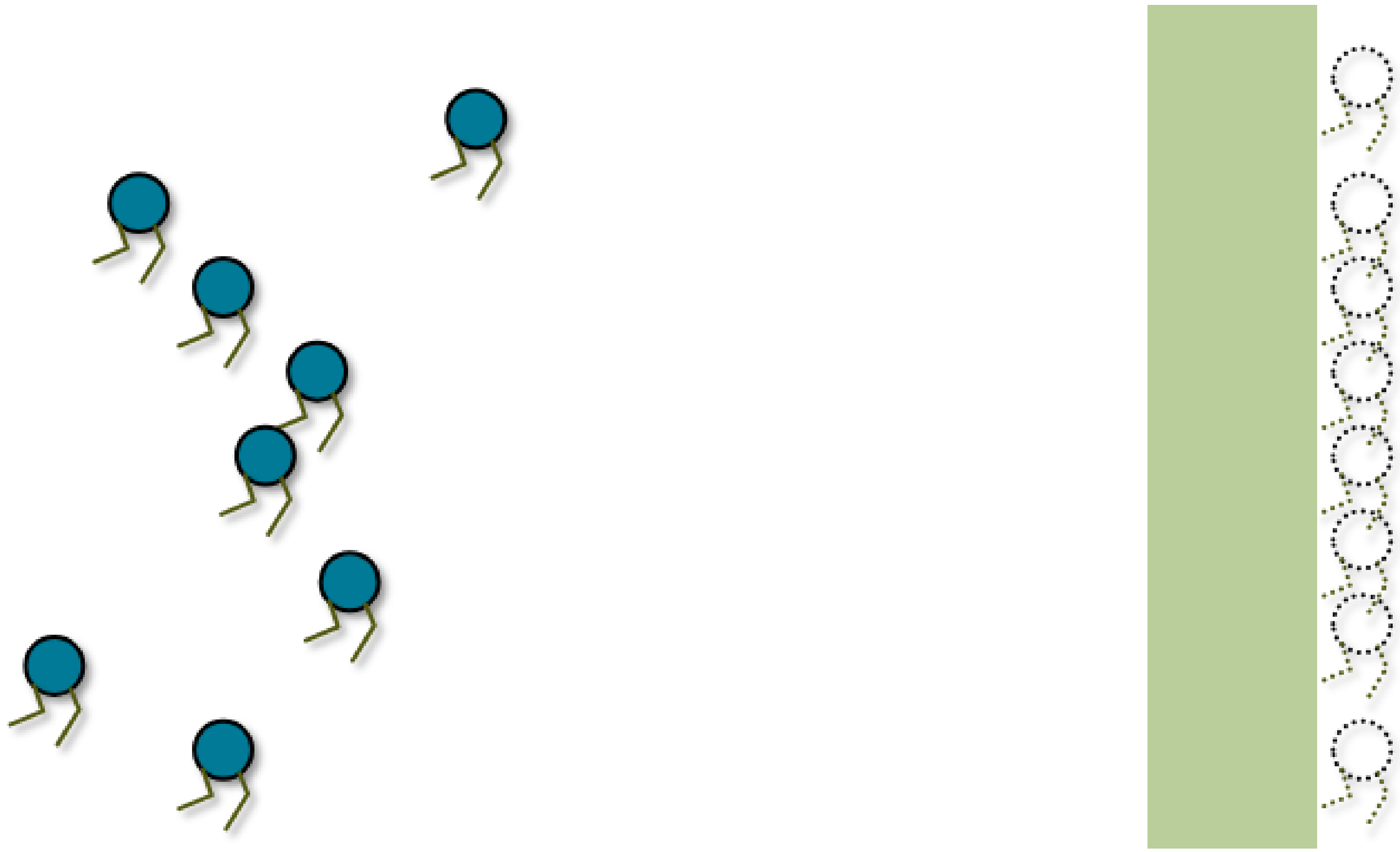
P first



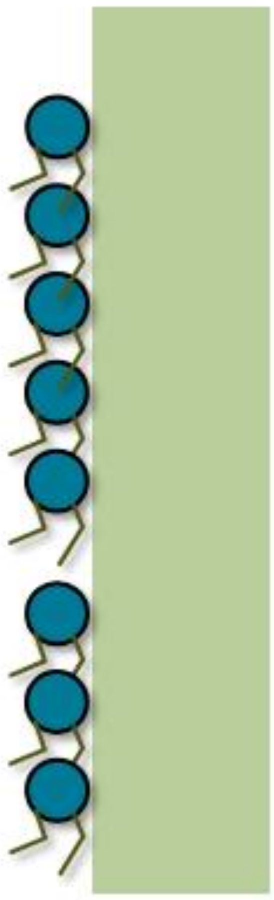
Q first



How about more than two threads? Barriers!



How about more than two threads? Barriers!



First attempt

Synchronize a number (n) of processes.

Semaphore `barrier`. Integer `count`.

	P1	P2	...	Pn
init	barrier = 0; volatile count = 0			
pre	...			
barrier	count++ if (count==n) release(barrier) acquire(barrier)	←	←	←
post	...			

Wrong

Synchronize a number (n) of processes.

Semaphore `barrier`. Integer count.

	P1	P2	...	Pn
init	`barrier = 0; volatile count = 0`			
pre	`...`			

barrier	`count++`
	`if (count==n) release(barrier)`
	`acquire(barrier)`
		←	←	←

post	`...`			

Race Condition !

Some wait forever!

How about this?

Synchronize a number (n) of processes.

Semaphores `barrier`, `mutex`. Integer count.

	P1	P2	. . .	Pn
init	mutex = 1; barrier = 0; count = 0			
pre	...			
barrier	acquire(mutex)			
	count++			
	release(mutex)			
	if (count==n) release(barrier)	←	←	←
	acquire(barrier)			
	release(barrier)			
post	...			

turnstile (acquire(barrier) / release(barrier))

How about this?

Synchronize a number (n) of processes.

Semaphores `barrier`, `mutex`. Integer `count`.

	P1	P2	. . .	Pn
init	mutex = 1; barrier = 0; count = 0			
pre	...			
barrier	acquire(mutex) count++ release(mutex) if (count==n) release(barrier) acquire(barrier) release(barrier) } turnstile	←	←	←
post	...			

Works, but we want it to be reusable!

Reusable Barrier

	P1	. . .	Pn
init	mutex = 1; barrier = 0; count = 0		
pre	…		

barrier	acquire(mutex)
	  count++
	release(mutex)
	if (count==n) release(barrier)
	
	acquire(barrier)
	release(barrier)
	
	acquire(mutex)
	  count--
	release(mutex)
	if (count==0) acquire(barrier)
		←	←

post	...		

Dou you see the problem?

Reusable Barrier

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	...		

barrier	`acquire(mutex)`
	`  count++`
	`release(mutex)`
	`if (count==n) release(barrier)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`  count--`
	`release(mutex)`
	`if (count==0) acquire(barrier)`
		←	←

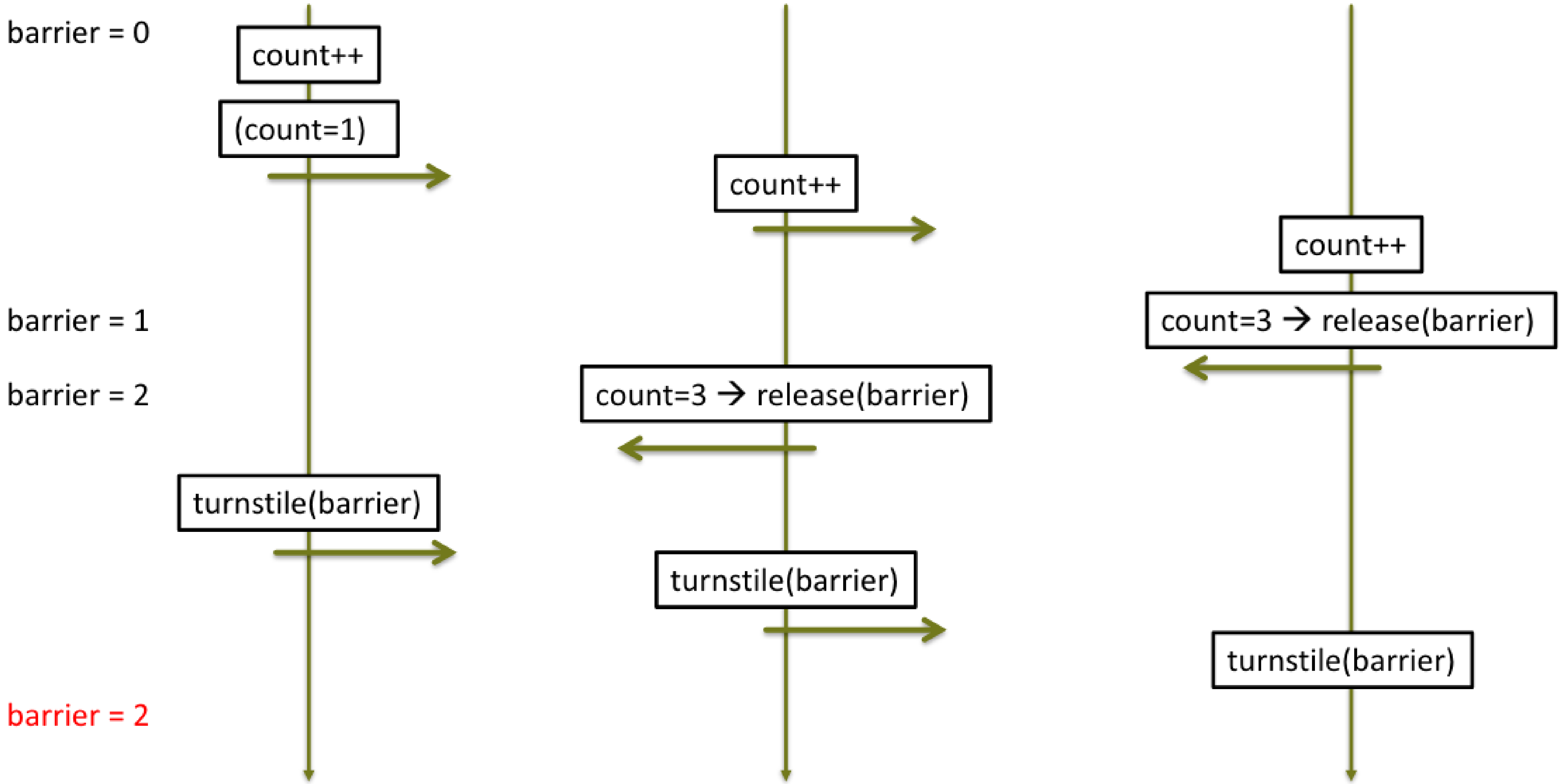
post	...		

Dou you see the problem?

Race Condition !

Race Condition !

Scheduling Scenario



Reusable Barrier 2nd try

	P1	...	Pn
init	mutex = 1; barrier = 0; count = 0		
pre	...		

barrier	acquire(mutex)
	  count++
	  if (count==n) release(barrier)
	release(mutex)		

	acquire(barrier) release(barrier)	←	←

	acquire(mutex)
	  count--
	  if (count==0) acquire(barrier)
	release(mutex)		

post	...		

Dou you see the problem?

Doesn't quite work yet

	P1	. . .	Pn
init	mutex = 1; barrier = 0; count = 0		
pre	...		

barrier	acquire(mutex)
	  count++
	  if (count==n) release(barrier)
	release(mutex)
	
	acquire(barrier)
	release(barrier)
	
	acquire(mutex)
	  count--
	  if (count==0) acquire(barrier)
	release(mutex)
		←	←

post	...		

Dou you see the problem?

Process can pass other processes!

Solution: Two-Phase Barrier

init

```
mutex=1; barrier1=0; barrier2=1; count=0
```

barrier

```
acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
signal(mutex)

acquire(barrier2); release(barrier2)
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

Barriers code examples

- See code

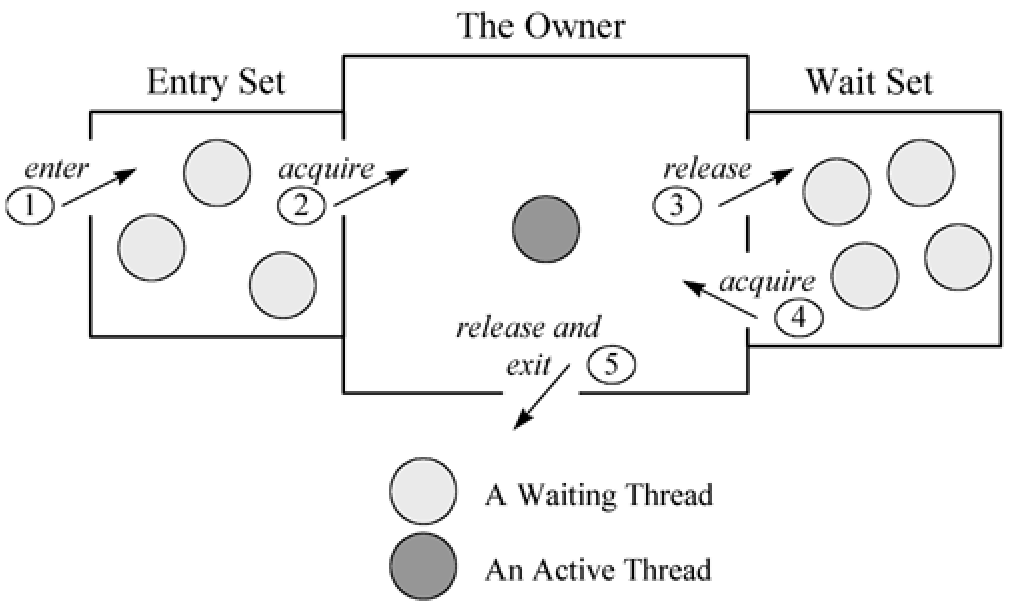
Teaching Awards

- Ich wäre dankbar, wenn ihr für mich abstimmen könntet!

Monitors and Lock Conditions

Lecture Recap: Monitors

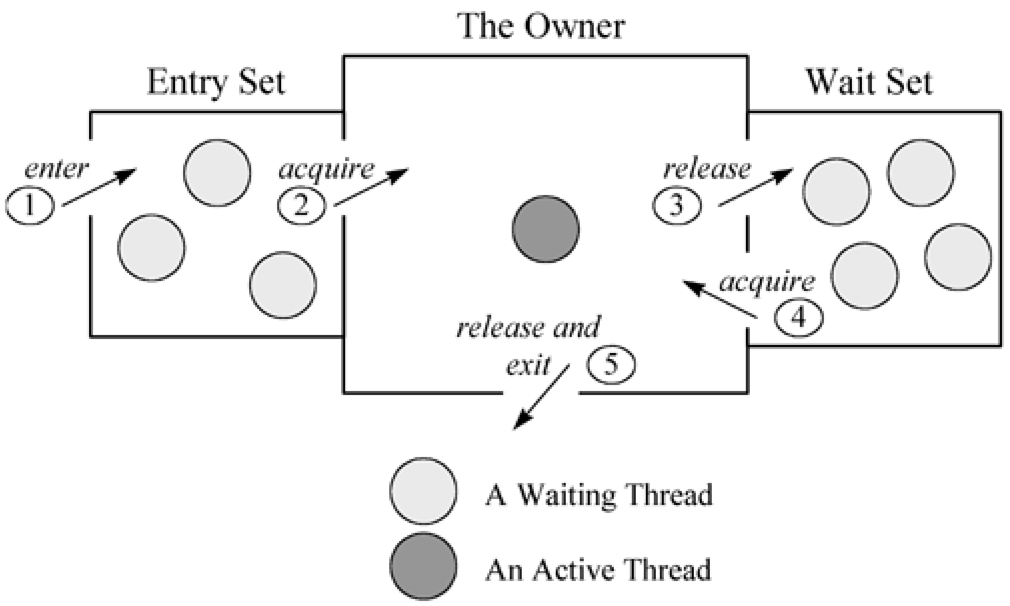
Monitors provide two kinds of thread synchronization: **mutual exclusion** and **cooperation** using a lock



- higher level mechanism than semaphores and more powerful
- instance of a class that can be used safely by several threads
- all methods of a monitor are executed with mutual exclusion

Lecture Recap: Monitors

Monitors provide two kinds of thread synchronization: **mutual exclusion** and **cooperation** using a lock



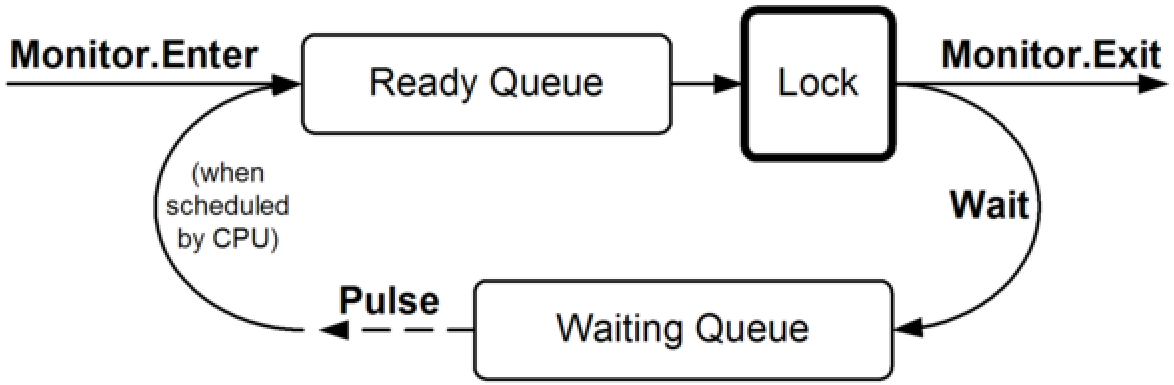
- the possibility to make a thread waiting for a condition
- signal one or more threads that a condition has been met

When thread is sent to wait we release the lock !
Can a monitor induce a deadlock?

Monitors in Java

Uses intrinsic lock (synchronized) of an object

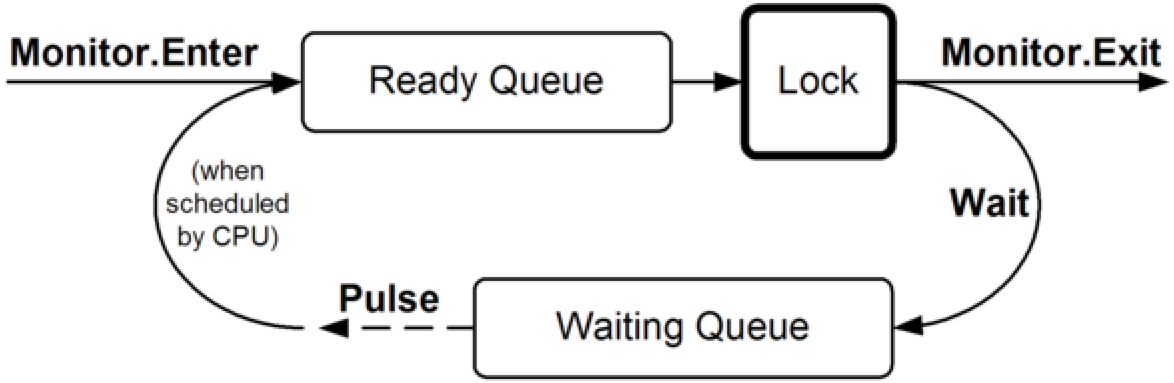
wait() – the current thread waits until it is
notify() – wakes up one waiting thread
notifyAll() – wakes up all waiting threads



Monitors in Java

Uses intrinsic lock (synchronized) of an object

wait() – the current thread waits until it is
notify() – wakes up one waiting thread
notifyAll() – wakes up all waiting threads



When do you use notify, when notifyAll?

Monitors in Java: Signal & Continue

- signalling process continues running
- signalling process moves signalled process to entry queue

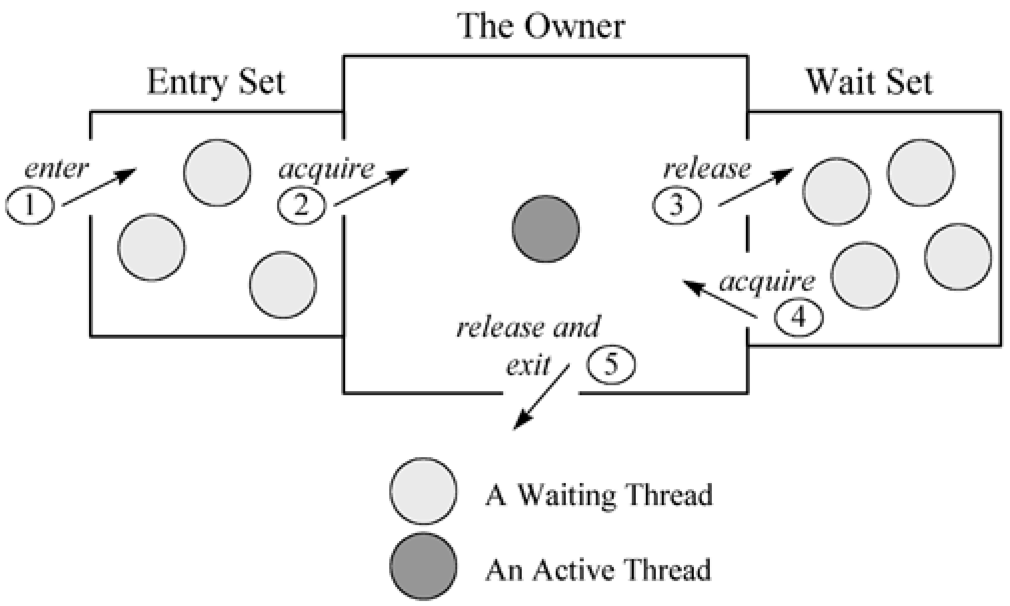


Figure 20-1. A Java monitor.

More theory:

- **Signal & Continue (SC)** : The process who signal keep the mutual exclusion and the signaled will be awaken but need to acquire the mutual exclusion before going. (Java's option)
- **Signal & Wait (SW)** : The signaler is blocked and must wait for mutual exclusion to continue and the signaled thread is directly awaken and can start continue its operations.
- **Signal & Urgent Wait (SU)** : Like SW but the signaler thread has the guarantee than it would go just after the signaled thread
- **Signal & Exit (SX)** : The signaler exits from the method directly after the signal and the signaled thread can start directly.

Monitors in Java: Signal & Continue

- signalling process continues running
- signalling process moves signalled process to entry queue

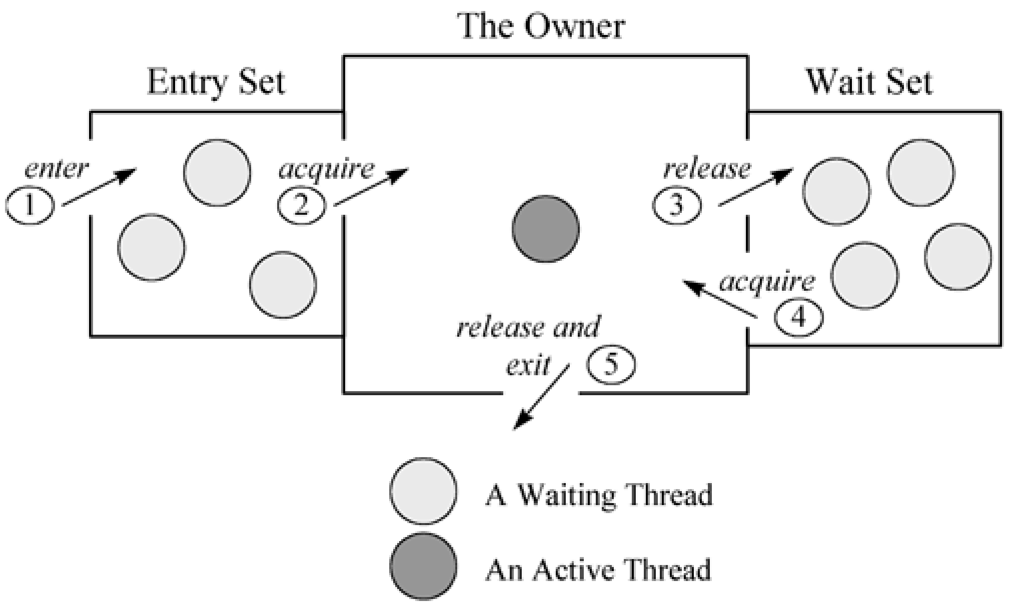
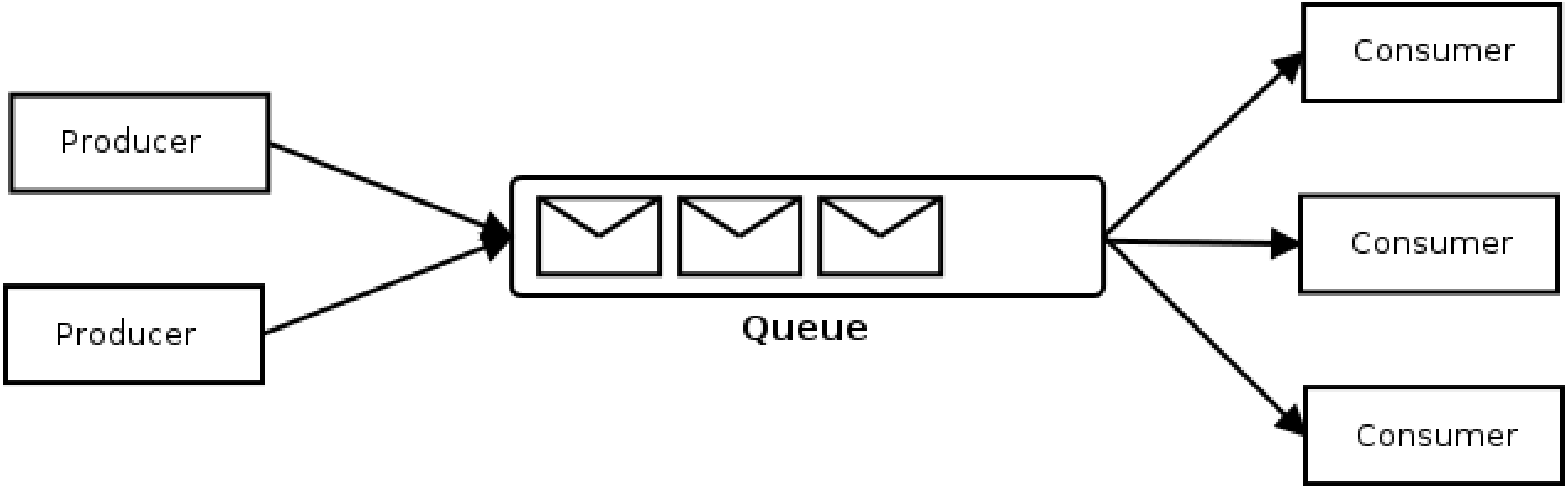


Figure 20-1. A Java monitor.

More abstractly there are 4 options:

- **Signal & Continue (SC)** : The process who signal keep the mutual exclusion and the signaled will be awaken but need to acquire the mutual exclusion before going. (Java's option)
- **Signal & Wait (SW)** : The signaler is blocked and must wait for mutual exclusion to continue and the signaled thread is directly awaken and can start continue its operations.
- **Signal & Urgent Wait (SU)** : Like SW but the signaler thread has the guarantee than it would go just after the signaled thread
- **Signal & Exit (SX)** : The signaler exits from the method directly after the signal and the signaled thread can start directly.

Monitors in Java: Example P/C Queue



A simple implementation, correct?

```java
synchronized void enqueue(long x) {
  if (isFull()){
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  doEnqueue(x);
  notifyAll();
}
```

```java
synchronized long dequeue() {
  long x;
  if (isEmpty()){
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  x = doDequeue();
  notifyAll();
  return x;
}
```

Monitors in Java: Example P/C Queue

```java
synchronized void enqueue(long x) {
  if (isFull()){
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  doEnqueue(x);
  notifyAll();
}
```

1. Queue is full
2. Process Q enters enqueue(), sees `isFull()`, and goes to the waiting list.
3. Process P enters dequeue()
4. In this moment process R wants to enter enqueue() and blocks
5. P signals Q and thus moves it into the ready queue, P then exits dequeue()
6. R enters the monitor before Q and sees `!isFull()`, fills the queue, and exits the monitor
7. Q resumes execution assuming `isFull()` is `false`

=> Inconsistency!

A simple implementation, correct?

```java
synchronized void enqueue(long x) {
  while (isFull()){
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  doEnqueue(x);
  notifyAll();
}
```

```java
synchronized long dequeue() {
  long x;
  while (isEmpty()){
    try {
      wait();
    } catch (InterruptedException e) {}
  }
  x = doDequeue();
  notifyAll();
  return x;
}
```

Whats the problem here?

- Producers and Consumers are in the same “wait” queue
- We must use notifyAll() because we can not target only producer (or consumer)

Lets try Locks

The Lock interface:

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

How do we wait/notify?

Use Conditions!

Can be used to implement monitors!

Java Locks provide conditions that can be instantiated Condition

```
notFull = lock.newCondition();
```

Java conditions offer

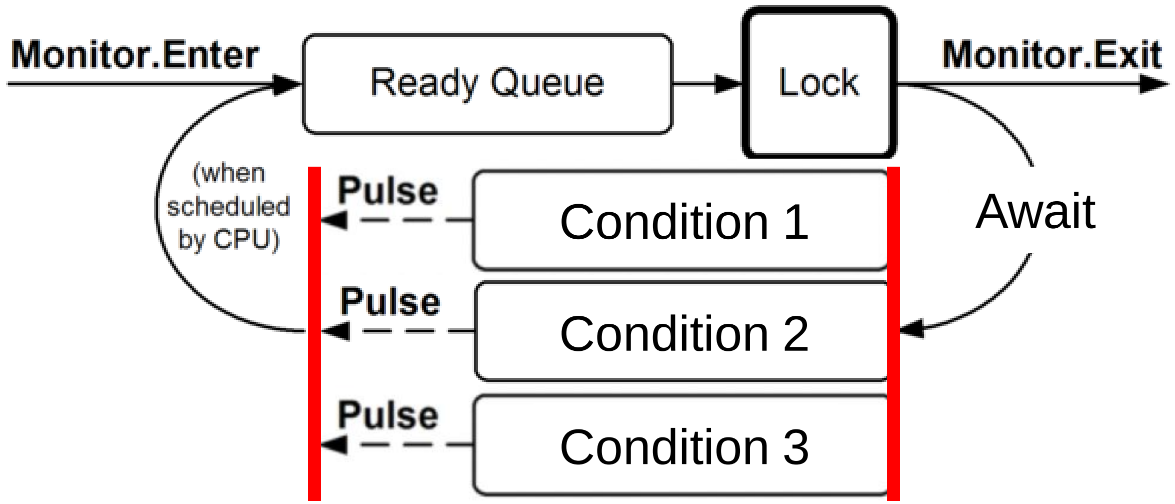
`.await()` – the current thread waits until condition is signaled

`.signal()` – wakes up one thread waiting on this condition

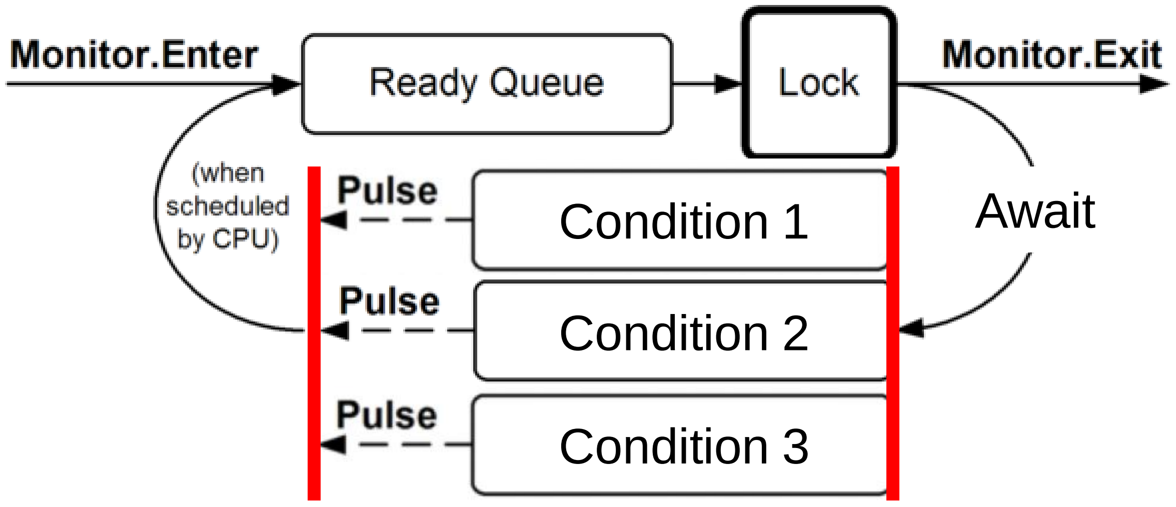
`.signalAll()` – wakes up all threads waiting on this condition

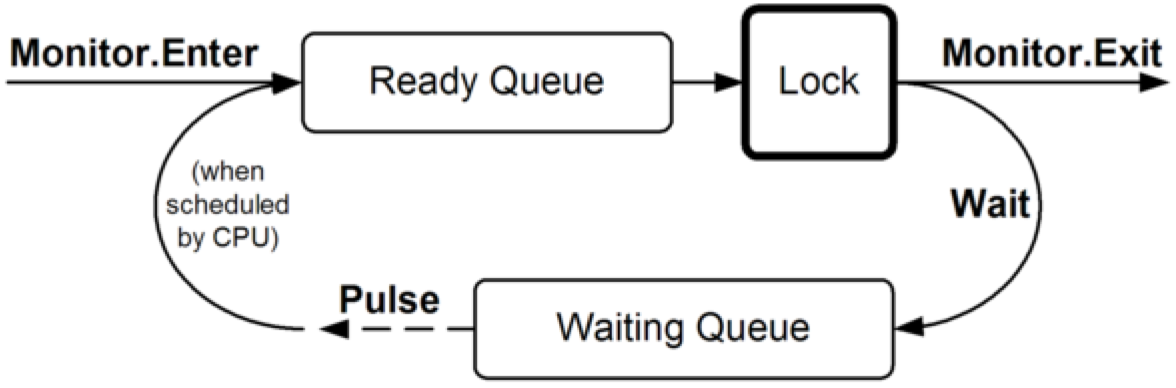
What is the difference to a Monitor?

Lock Conditions



Lock Conditions in Comparison to Monitor





Lock Conditions: Example P/C Queue

```
public class ProducerConsumer {
    private final Queue<Object> items;
    private final int capacity;

    private final Lock lock = new ReentrantLock();

    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    public ProducerConsumer(int capacity) {
        items = new ArrayDeque<Object>(capacity);
        this.capacity = capacity;
    }
```

Lock Conditions: Example P/C Queue

```
public void produce(Object data) throws InterruptedException {
    lock.lock();
    try {
        while (items.size()==capacity) {
            notFull.await();
        }
        items.add(data);
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
public Object consume() throws InterruptedException {
    lock.lock();
    try {
        while (items.isEmpty()) {
            notEmpty.await();
        }
        Object result = items.remove();
        notFull.signal();
        return result;
    } finally {
        lock.unlock();
    }
}
```

Why do we need the lock?

```
public void produce(Object data) throws InterruptedException {
    lock.lock();
    try {
        while (items.size()==capacity) {
            notFull.await();
        }
        items.add(data);
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
public Object consume() throws InterruptedException {
    lock.lock();
    try {
        while (items.isEmpty()) {
            notEmpty.await();
        }
        Object result = items.remove();
        notFull.signal();
        return result;
    } finally {
        lock.unlock();
    }
}
```

What is still not perfect?

notFull and notEmpty signal will be sent in any case, even when no threads are waiting.

- This is expensive!

A simple solution: Sleeping Barber

Sleeping barber requires additional counters for checking if processes are waiting:

$m \leq 0 \Leftrightarrow$ buffer full & -m producers (clients) are waiting

$n \leq 0 \Leftrightarrow$ buffer empty & -n consumers (barbers) are waiting

P/C, Sleeping Barber Variant

```
class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull = lock.newCondition();
    int m; final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s; m = size-1;
        buf = new long[size];
    }
...
}
```

Two variables ☹ sic!
(cf. last lecture)

P/C, Sleeping Barber Variant

```
void enqueue(long x) {

   lock.lock();
   m--; if (m<0)
      while (isFull())
         try { notFull.await(); }
         catch(InterruptedException e){}
   doEnqueue(x);
   n++;
   if (n<=0) notEmpty.signal();
   lock.unlock();

}
```

```
long dequeue() {
   long x;
   lock.lock();
   n--; if (n<0)
      while (isEmpty())
         try { notEmpty.await(); }
         catch(InterruptedException e){}
   x = doDequeue();
   m++;
   if (m<=0) notFull.signal();
   lock.unlock();
   return x;
}
```

Guidelines to using condition waits

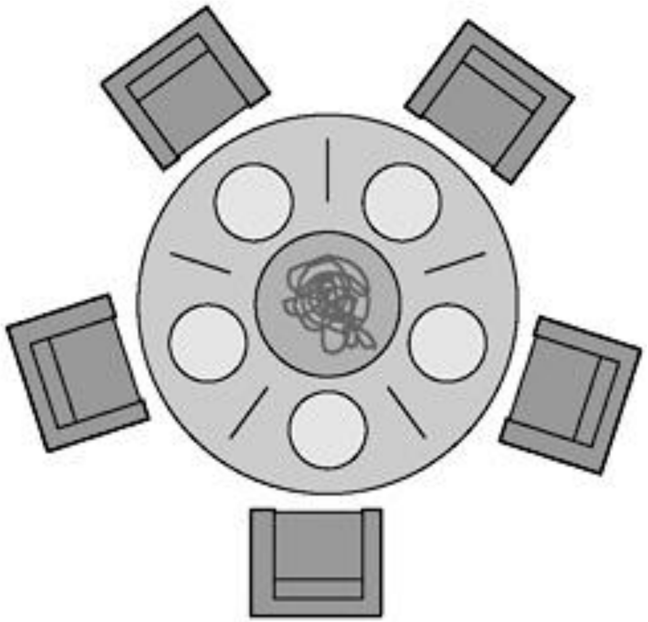
- Always have a condition predicate
- Always test the condition predicate:
  - before calling wait
  - after returning from wait
  - **Always call wait in a loop**
- Ensure state is protected by lock associated with condition
  - What could go wrong if you don’t? (e.g. in sleeping barber variant)

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- Theory
- **Intro Assignment 10**
- Kahoot
- Exam questions

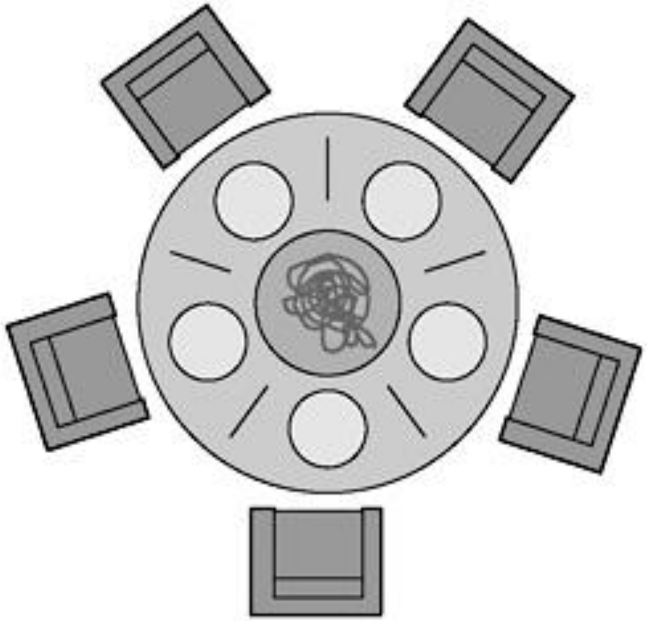
Assignment 10

Task 1 - Dining Philosophers



Originally proposed by E. W. Dijkstra
Imagine five philosophers who spend their lives thinking and eating.
They sit around a circular table with five chairs with a big plate of spaghetti.
However, there are only five chopsticks available.

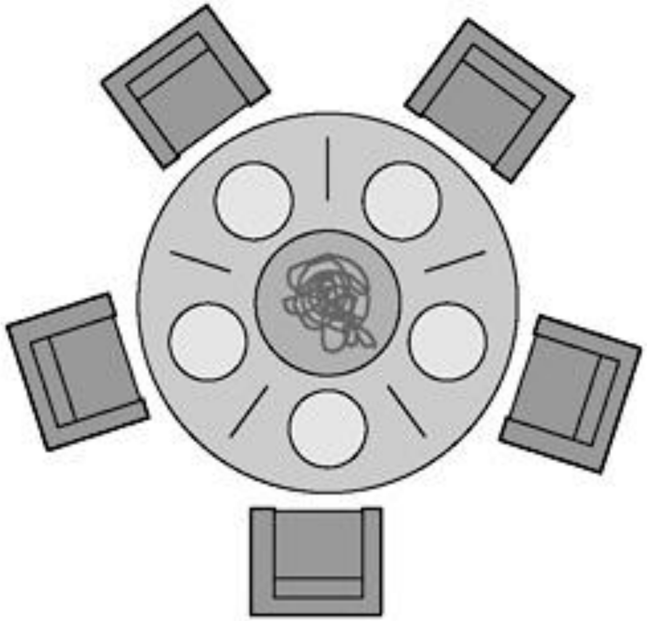
Task 1 - Dining Philosophers



Each philosopher thinks and when he gets hungry picks up the two chopsticks closest to him.

- If a philosopher can pick up BOTH chopsticks, he eats for a while.
- After a philosopher finishes eating, he puts down the chopsticks and starts to think again.

Find a solution that…

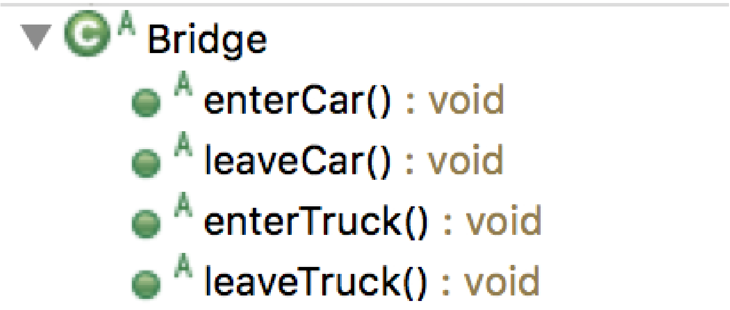


- Makes deadlocks impossible
- Has no starvation
- More than one parallel eating philosopher is possible

Task 2 – Monitors, Conditions and Bridges

Only either 3 cars or one truck may be on the bridge at each moment.

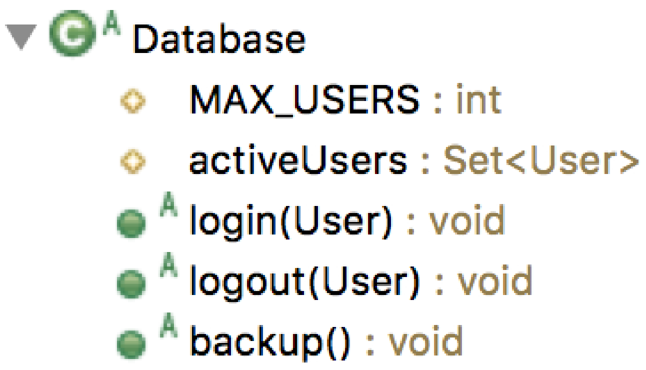
Implement Classes `BridgeMonitor` and `BridgeCondition`



How to Test my Implementation?
Implement method `invariant()` to check if the state is valid: at the end of a method there are never too many cars or trucks on the bridge

Task 3 – Semaphores and Databases

Use semaphores to implement login and logout database functionality that supports up to 10 concurrent users



Use barrier to implement 2-phase backup functionality.

Task 3 – Semaphores and Databases

Implement Classes MySemaphore and MyBarrier

Use monitors for both to avoid busy loop

- Put processes to sleep, when there is no entry into semaphore
- Wake up a waiting process when releasing a semaphore

```
acquire(S) {
       wait until S > 0
       dec(S)
}

release(S) {
       inc(S)
}
```

Try to understand the existing DatabaseJava implementation before implementing your own semaphore and barrier.

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- Theory
- Intro Assignment 10
- **Kahoot**
- Exam questions

Kahoot!

Plan für heute

- Organisation
- Nachbesprechung Assignment 9
- Theory
- Intro Assignment 10
- Kahoot
- **Exam questions**

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```
class Barrier {
      AtomicInteger i = new AtomicInteger(0);
      final int threads = N;
      public void await() throws InterruptedException {
              int cur_threads = i.incrementAndGet();
              if(cur_threads < threads) {
                while (i.get() < threads) {}
              }
      }
}
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```java
class Barrier {
    AtomicInteger i = new AtomicInteger(0);
    final int threads = N;
    public void await() throws InterruptedException {
        int cur_threads = i.incrementAndGet();
        if(cur_threads < threads) {
          while (i.get() < threads) {}
        }
    }
}
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

True, there is no data race since `incrementAndGet` increases `i` atomically.

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```java
class Barrier {
    AtomicInteger i = new AtomicInteger(0);
    final int threads = N;
    public void await() throws InterruptedException {
        int cur_threads = i.incrementAndGet();
        if(cur_threads < threads) {
          while (i.get() < threads) {}
        }
    }
}
```

◯ Der Code beendet sich immer. *Code will always complete.*

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll N threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of N threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```java
class Barrier {
      AtomicInteger i = new AtomicInteger(0);
      final int threads = N;
      public void await() throws InterruptedException {
            int cur_threads = i.incrementAndGet();
            if(cur_threads < threads) {
              while (i.get() < threads) {}
            }
      }
}
```

◯ Der Code beendet sich immer. *Code will always complete.*

True, it is a correct barrier implementation.

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll `N` threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of `N` threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```
class Barrier {
	AtomicInteger i = new AtomicInteger(0);
	final int threads = N;
	public void await() throws InterruptedException {
		int cur_threads = i.incrementAndGet();
		if(cur_threads < threads) {
		  while (i.get() < threads) {}
		}
	}
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

Barriers and Synchronization (9 points)

11. (a) Wir möchten eine einfache Barriere (muss nicht wiederverwendbar sein) implementieren. Die Barriere soll N threads synchronisieren. Markieren Sie welche der folgenden Aussagen auf die jeweiligen implementierungen zutreffen. Sollten Sie den Code für ineffizient halten, nennen sie kurz den Grund.

We want to implement a simple barrier (does not have to be reusable) that allows to synchronize the execution of N threads. Mark whether each of the following statements is true for each implementation. If you consider this code to be inefficient, shortly state why. (4)

i.

```java
class Barrier {
      AtomicInteger i = new AtomicInteger(0);
      final int threads = N;
      public void await() throws InterruptedException {
            int cur_threads = i.incrementAndGet();
            if(cur_threads < threads) {
              while (i.get() < threads) {}
            }
      }
}
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

True, the waiting threads are busy waiting.

ii.

```
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

○ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

ii.

```java
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

◯ Der gezeigte Code hat die gewünschte Semantik. *Code has the desired semantics.*

Yes

ii.

```java
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

◯ Der Code beendet sich immer. *Code will always complete.*

ii.

```java
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

◯ Der Code beendet sich immer. *Code will always complete.*

True

ii.

```
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

◯ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

ii.

```
class Barrier {
        int i = 0;
        final int threads = N;
        public synchronized void await() throws InterruptedException {
                ++i;
                while (i < threads) { wait(); }
                notify();
        }
}
```

○ Der Code verendet die Rechenressourcen unter Umständen ineffizient. Warum?

Code might not use compute resources efficiently. Why?

False, the code makes use of wait/notify and thus does not waste compute resources.

Feedback

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

Teaching Awards

- Ich wäre dankbar, wenn ihr für mich abstimmen könntet!

Danke

- Bis nächste Woche!