Parallele Programmierung FS25

Exercise Session 9

Jonas Wetzel

Plan für heute

- Organisation
- Nachbesprechung Exercise 8
- Theory
- Intro Exercise 9
- 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?

Deckers Lock
Peterson Lock
Atomic Registers
Filter Lock, Bakery Lock

Plan für heute

- Organisation
- **Nachbesprechung Exercise 8**
- Theory
- Intro Exercise 9
- Exam Questions
- Kahoot

Feedback: Exercise 8

Race Conditions

- Race condition occurs if multiple accesses can happen concurrently and at least one access is a write

Thread 1	Thread 2

x=23;
r1 = x;
v = r1;
z = 2;
	y=42;
	r2 = y;
	w = r2;

- Thread 1 accesses x, v, z
- Thread 2 accesses y, w
- No race condition, no bad interleaving possible

Race Conditions

- Race condition occurs if multiple accesses can happen concurrently and at least one access is a write

Thread 1	Thread 2
x = 23;	y = 42;
r1 = x;	r2 = y;
v = r1;	w = r2;
y = 2;	z = 2;

- Thread 1 accesses x, v, y
- Thread 2 accesses y, w, z
- Both write to y! Result depends on interleaving!

Relations

- For a set S we can define a mathematical relation R for members of S

- Example:
  - Set of natural numbers, relation “greater or equal than”
  - Set of all humans, relation “knows”

Relations

- Relations can have different properties

- Example:
  - Transitivity: a R b and b R c implies a R c

  - The “greater than” relation is transitive.
  - The ”knows” relation is not transitive.

Transitive Closure

- For a relation R, the smallest relation which contains R and is transitive, is called the transitive closure of R.

- Example:
  - For the set of airports in the world, we can define the relation “offers direct flight to”
  - This is (probably) not transitive (show why)
  - The transitive closure also has meaning: ”can fly from a to b with stops”

Relations and Code

- When we execute code, ”actions” happen, i.e., a variable gets read or written
- We can define relations for these actions, such as “is executed before”
  - Easy to check because it is a local property
  - Can build the transitive closure if we want to know if actions are ordered!
  - Not all actions are ordered!

Program Order

- Not all actions are ordered!

```
S1:  a=23;
S2:  x=3;
S3:  if (x==3) {
S4:    b += 1;
     } else {
S5:    b *= 2;
S6:    x = 0;
     }
S7:  x=4;
```

Not in program order!

Program Order

- Action in mutually exclusive code paths are not in program order
- Actions in different threads are not in program order!

- But ordering was good for proofs!
- Want to allow the compiler / hardware to reorder sometimes for performance.

- Solution: Let compiler reorder whenever it is not “observable” – need to define a subset of special actions which are visible across threads

Synchronization Actions

- Solution: Let compiler reorder whenever it is not “observable” – need to define a subset of special actions which are visible across threads
  - For this lecture, the most important synchronization actions are
    - Start/End of a thread
    - Read/Write of a volatile or atomic variable
    - Acquire / release of a monitor

Synchronizes With Relation

- The variable x is initially 0.
- Thread A writes x=5
- Thread B reads/prints the value of x

- We could see 0 -> then we expect that B executed before A
- We could see 5 -> then we expect that A executed before B

- The synchronizes with relation says a read of a volatile must return the last value written to it. It synchronizes with that last write across threads!

Synchronizes With Relation

- If we combine (the transitive closure of) program order and “synchronizes with” we get the “happens before” order
- Any output/result we see in a Java Program must be consistent with this happens before order

Java Memory Model Takeaways

- If a variable is not declared volatile you must use the happens-before order to reason about possible values -> requires thought
- If a (primitive) variable is volatile it behaves like an atomic register

- Can sometimes gain performance (and maintain correctness) by not marking everything volatile.
- But you are using Java, is performance really your focus? ☺ - if in doubt declare shared variables as volatile

Plan für heute

- Organisation
- Nachbesprechung Exercise 8
- **Theory**
- Intro Exercise 9
- Exam Questions
- Kahoot

Theory Recap

Memory hierachy (one core)

ALUs

Registers — 0.5ns

L1 Cache — 1 ns

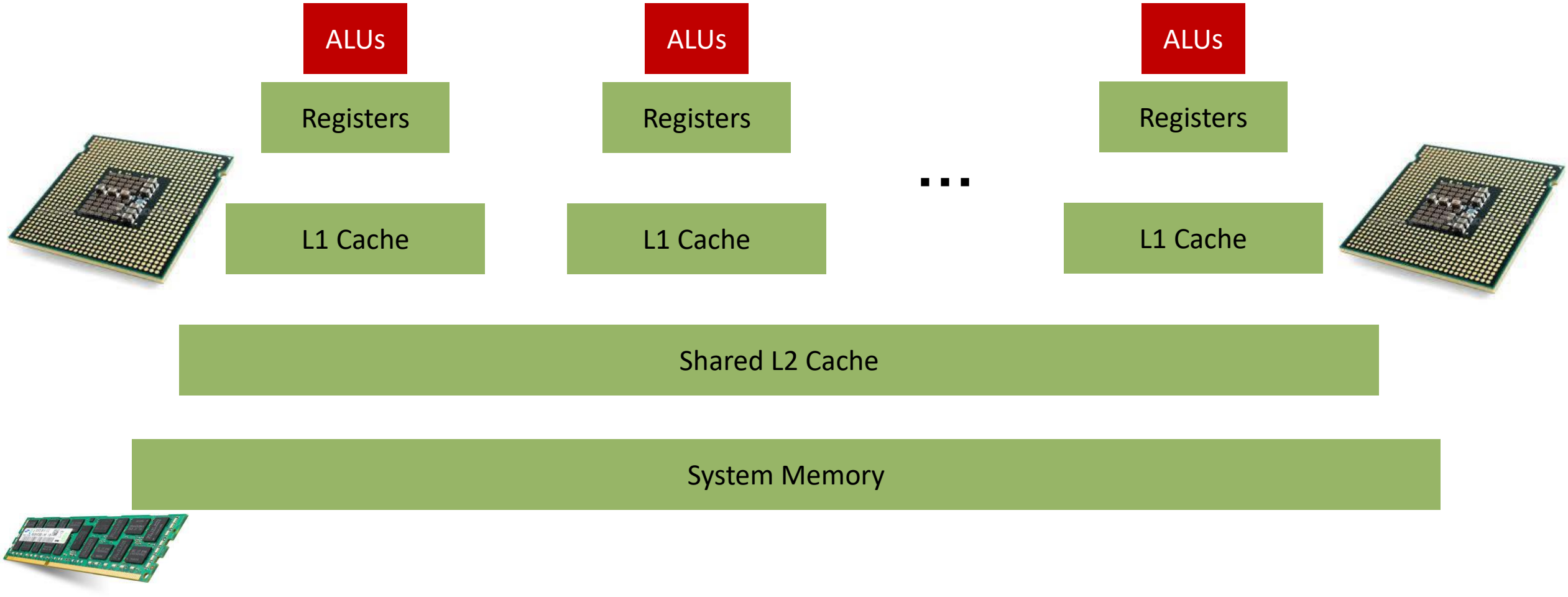
L2 Cache — 7 ns

System Memory — 100 ns

fast, low latency, high cost, low capacity

slow, high latency latency, low cost, high capacity

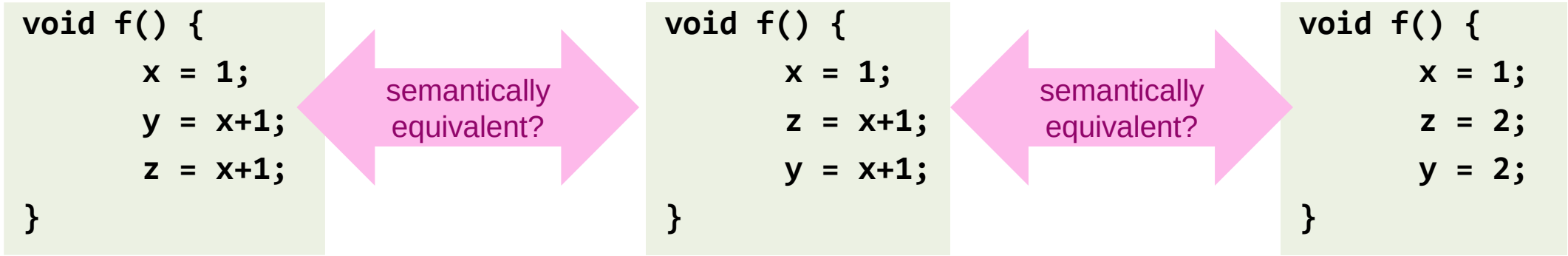
Memory hierachy (many cores)



Lets take a step back

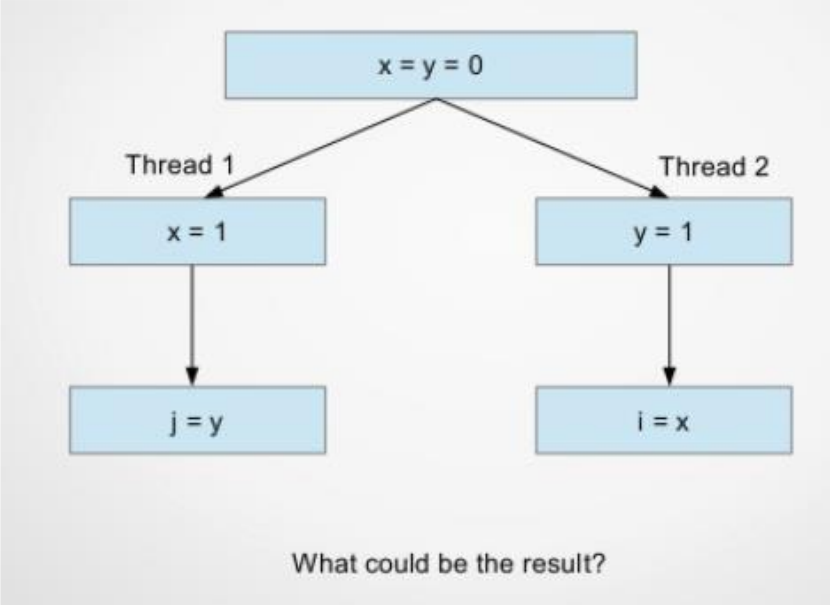
Threads perform actions, those actions are usually read/write to memory locations.

Compiler and Hardware can optimize read/writes by reordering memory accesses.



The big problem

Because of memory reordering we can suddenly get results that we did not expect!



Answer:
i=1, j=1
i=0, j=1
i=1, j=0
i=0, j=0 (but why?)

Visibility not guaranteed

And even if an action has been executed, we do not have guarantees that other threads see them (in the correct order).

In other words, actions that were performed by one thread may not be **visible** to another thread!

We want to make sure that the actions become visible. And we want some guarantees on the ordering.

How? Java Memory Model!

- To reason about this we need some formalism
- This is what the java memory model gives us
- JMM is defined using various orders on our program execution

volatile

- value of a volatile field becomes visible to all readers (other threads in particular) after a write operation completes on it
- Without volatile, readers could see some non-updated value

Program Order

int x,y = 0, volatile int z = 0

Thread 1

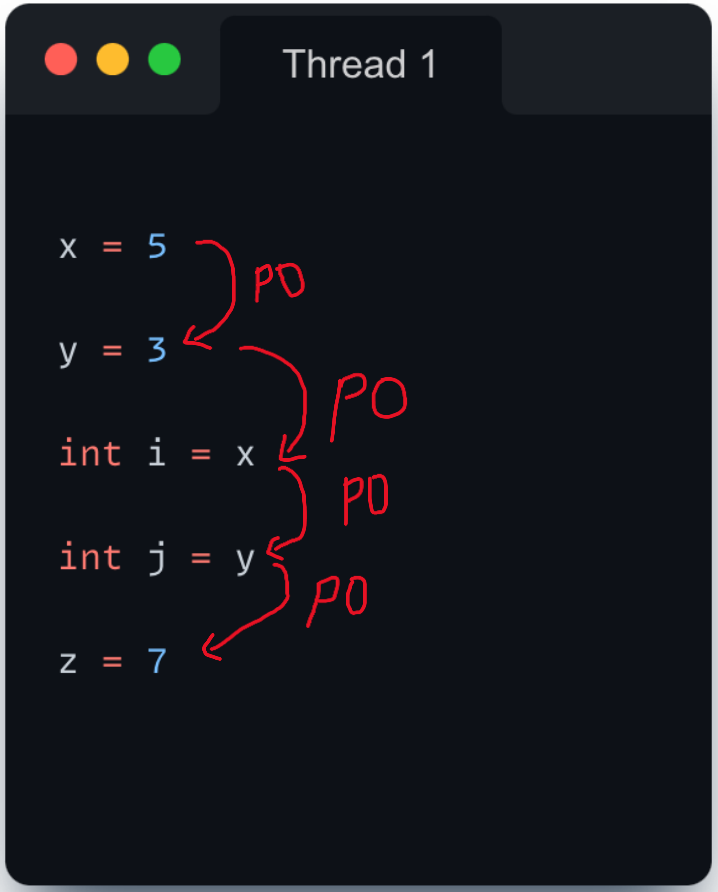
```
x = 5

y = 3

int i = x

int j = y

z = 7
```

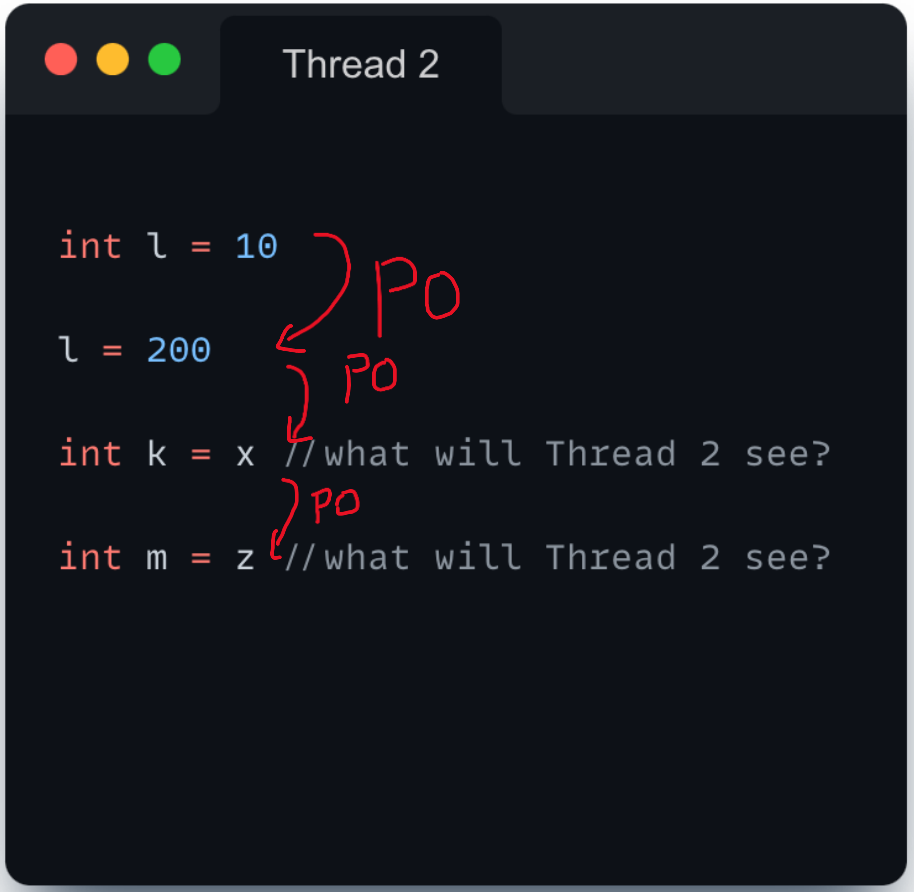
Thread 2

```
int l = 10

l = 200

int k = x //what will Thread 2 see?

int m = z //what will Thread 2 see?
```

Program Order

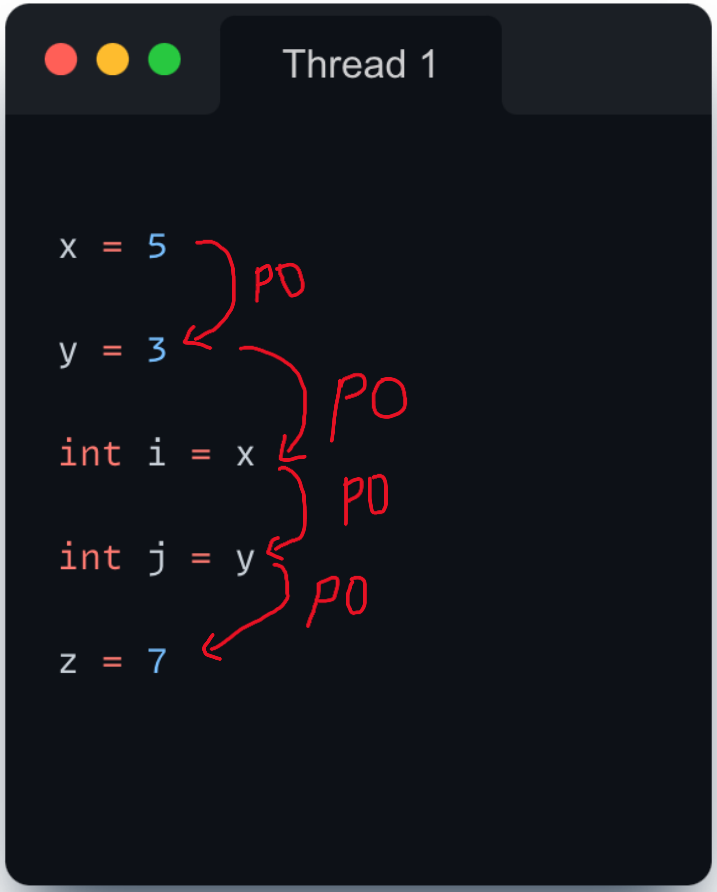
int x,y = 0, volatile int z = 0

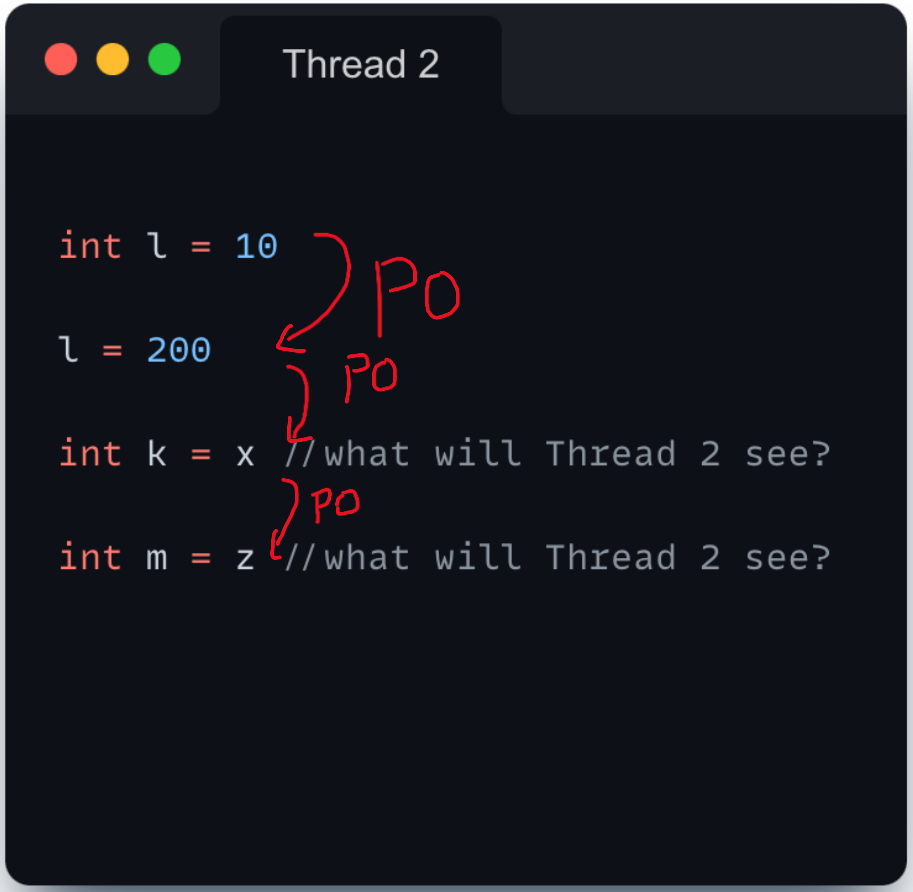




Synchronization Actions

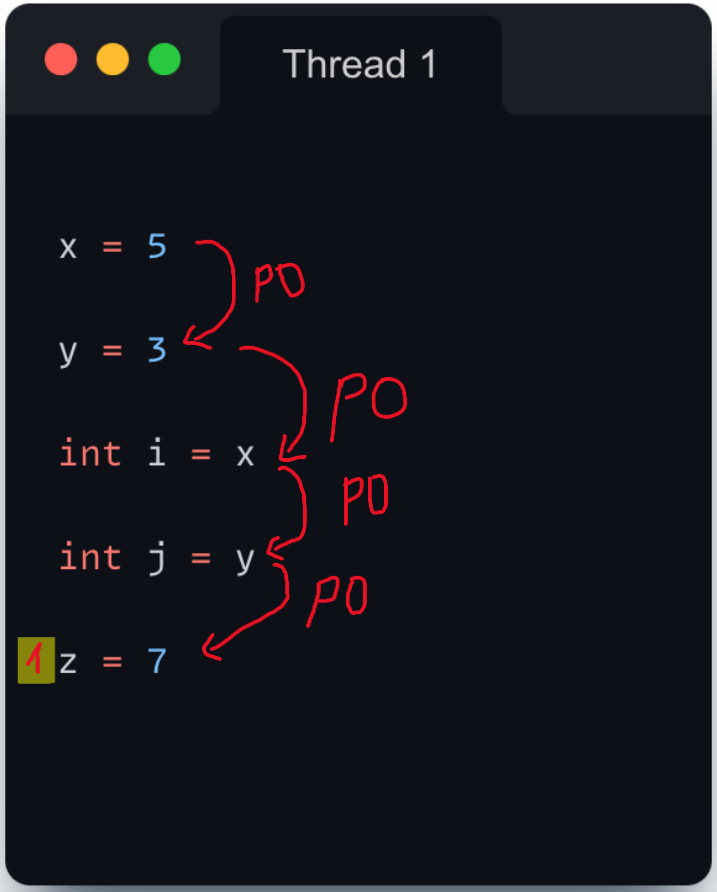
int x,y = 0, volatile int z = 0

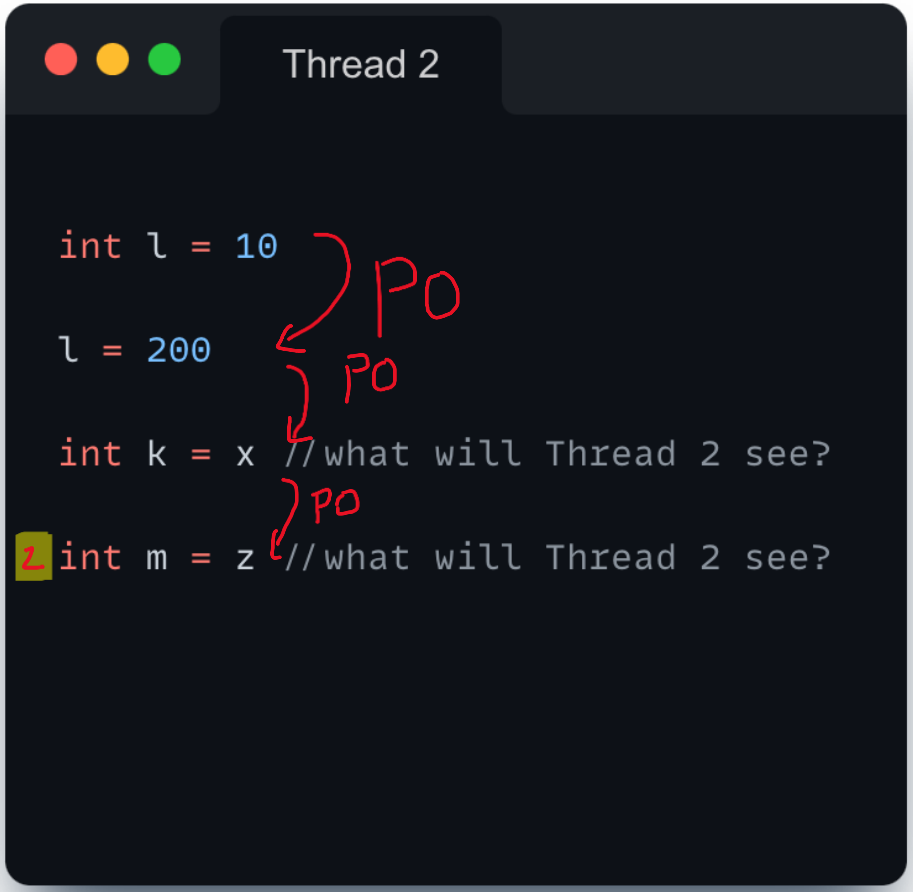




Assume Thread 1 runs first

int x,y = 0, volatile int z = 0 - SO: write(z, 7) --> m = read(z)





JMM: Synchronizes-With (SW) / Happens-Before (HB) orders

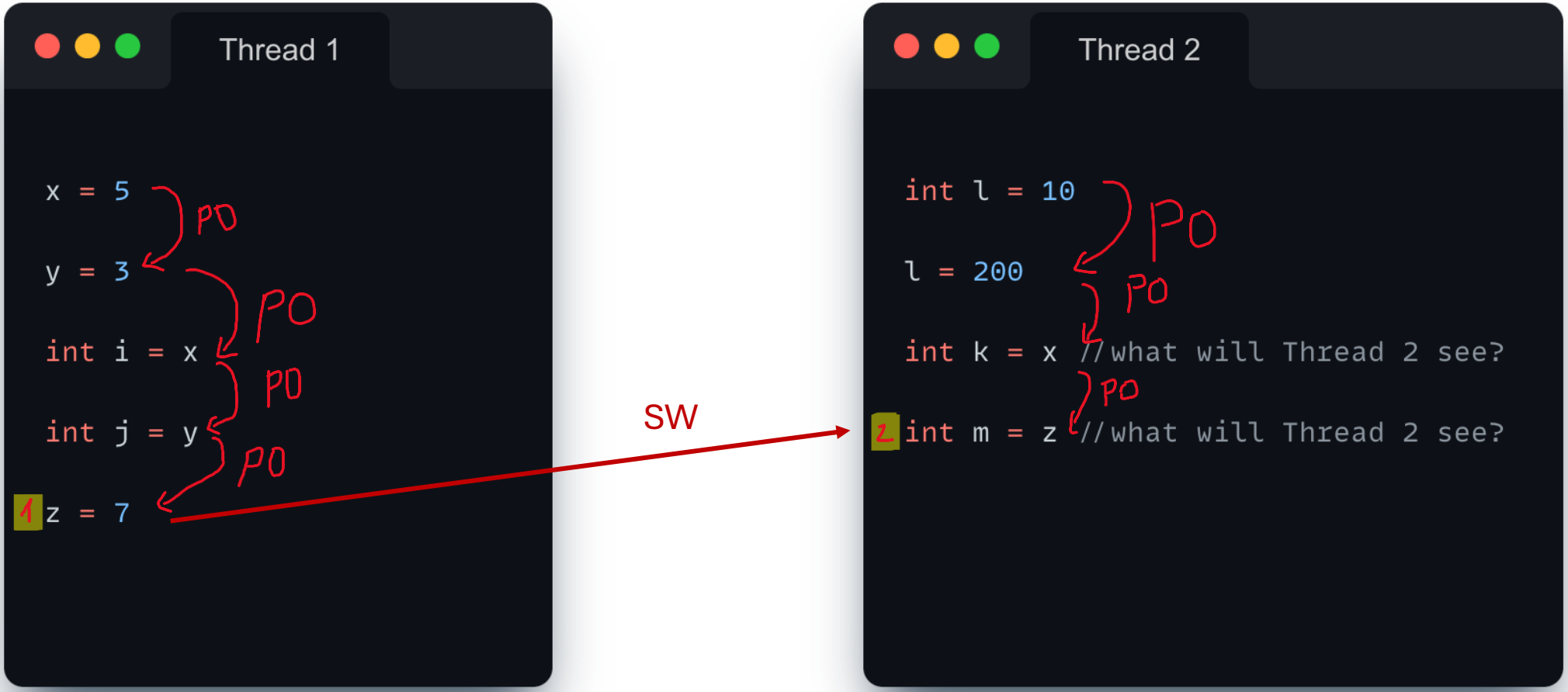
- **SW only pairs the specific actions which "see" each other**
- **A volatile write to x synchronizes with subsequent read of x (subsequent in SO)**
- **The transitive closure of PO and SW forms HB**
- **HB consistency: When reading a variable, we see either the last write (in HB) or any other unordered write.**
  - This means races are allowed!

Synchronization actions induce the *synchronized-with* relation on actions, defined as follows:

- An unlock action on monitor *m* *synchronizes-with* all subsequent lock actions on *m* (where "subsequent" is defined according to the synchronization order).
- A write to a volatile variable *v* (§8.3.1.4) *synchronizes-with* all subsequent reads of *v* by any thread (where "subsequent" is defined according to the synchronization order).
- An action that starts a thread *synchronizes-with* the first action in the thread it starts.

Now we can identify synchronizes-with

int x,y = 0, volatile int z = 0 - SO: write(z, 7) --> m = read(z)



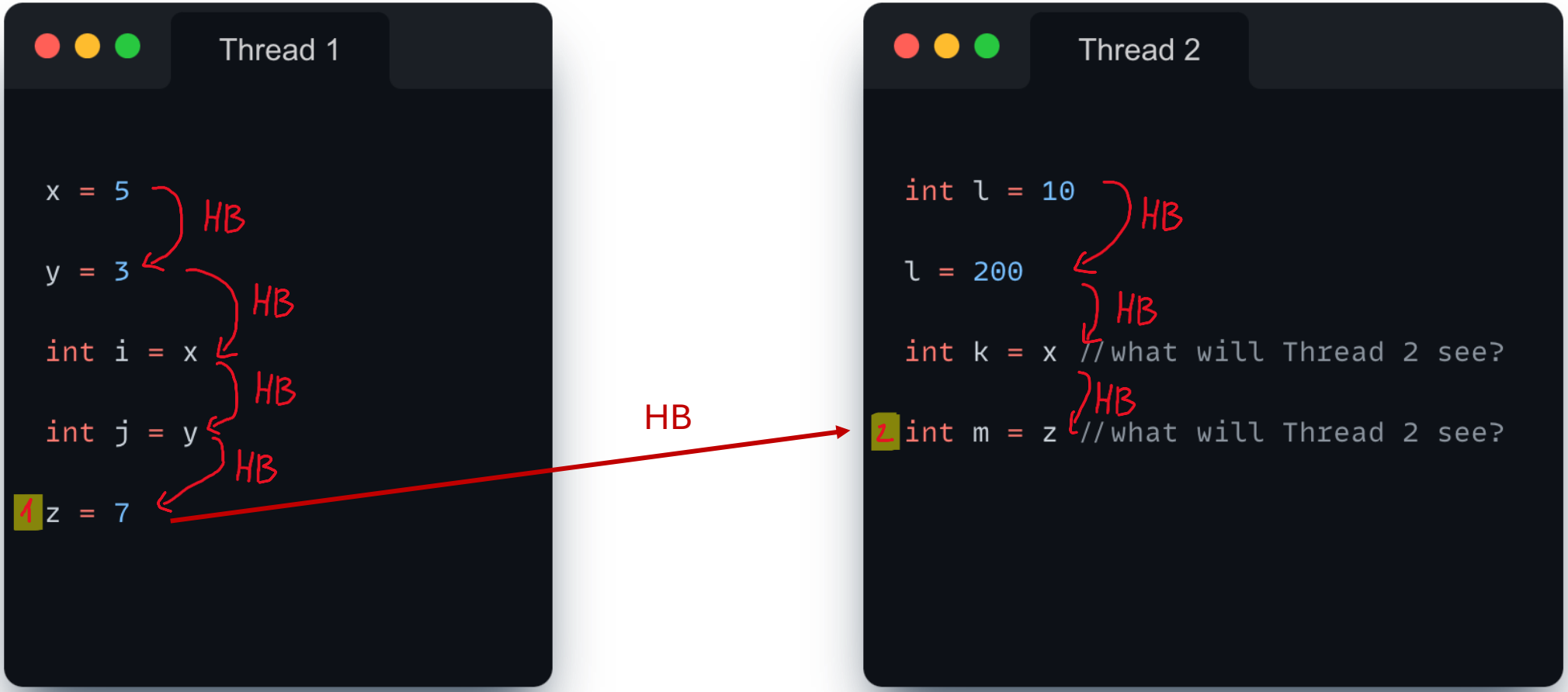
Happens-before relationship

Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second.

- **Transitive closure of PO and SW forms happens before order**
- All values we observe must obey this happens before order!
- If x and y are actions of the same thread and x comes before y in program order, then hb(x, y).
- If an action x synchronizes-with a following action y, then we also have hb(x, y).
- If hb(x, y) and hb(y, z), then hb(x, z).

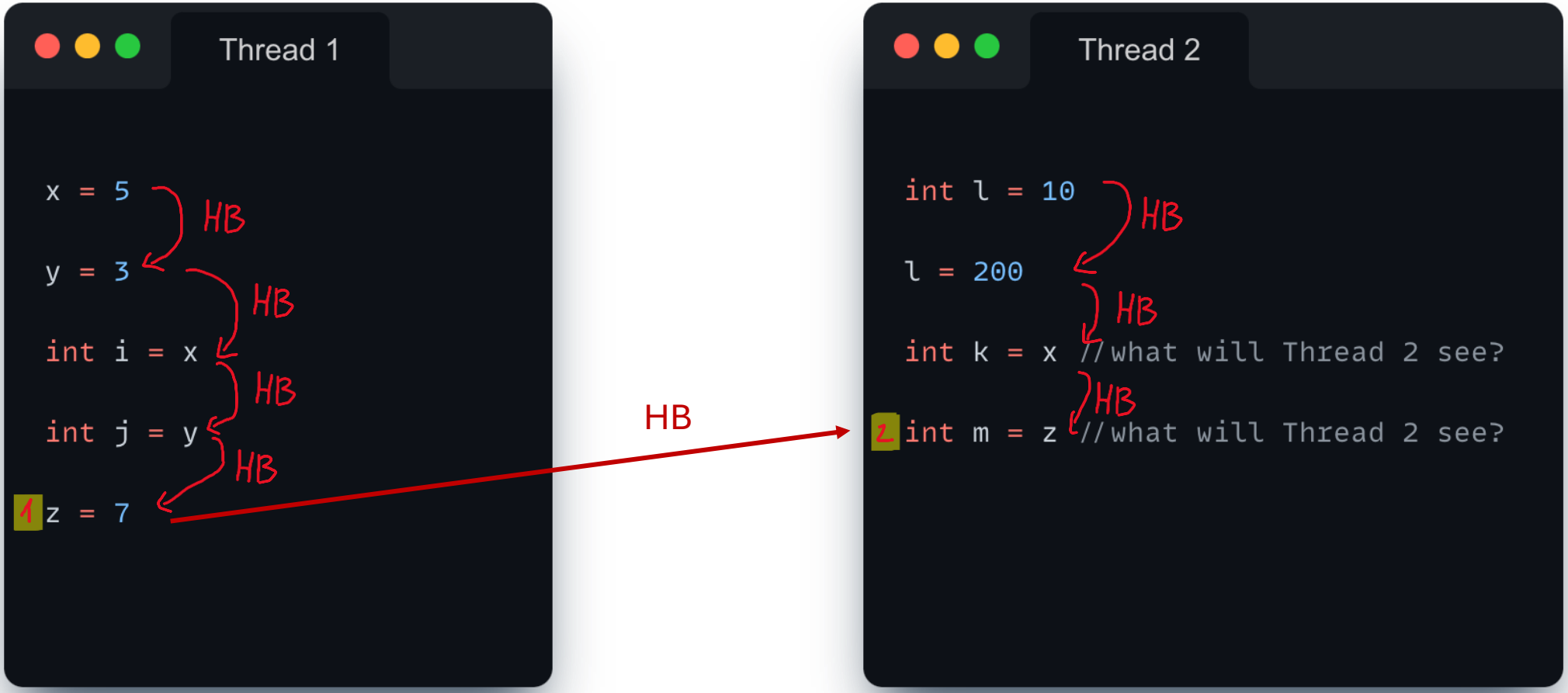
SW + PO gives us Happens-Before relationship

int x,y = 0, volatile int z = 0 - SO: write(z, 7) --> m = read(z)



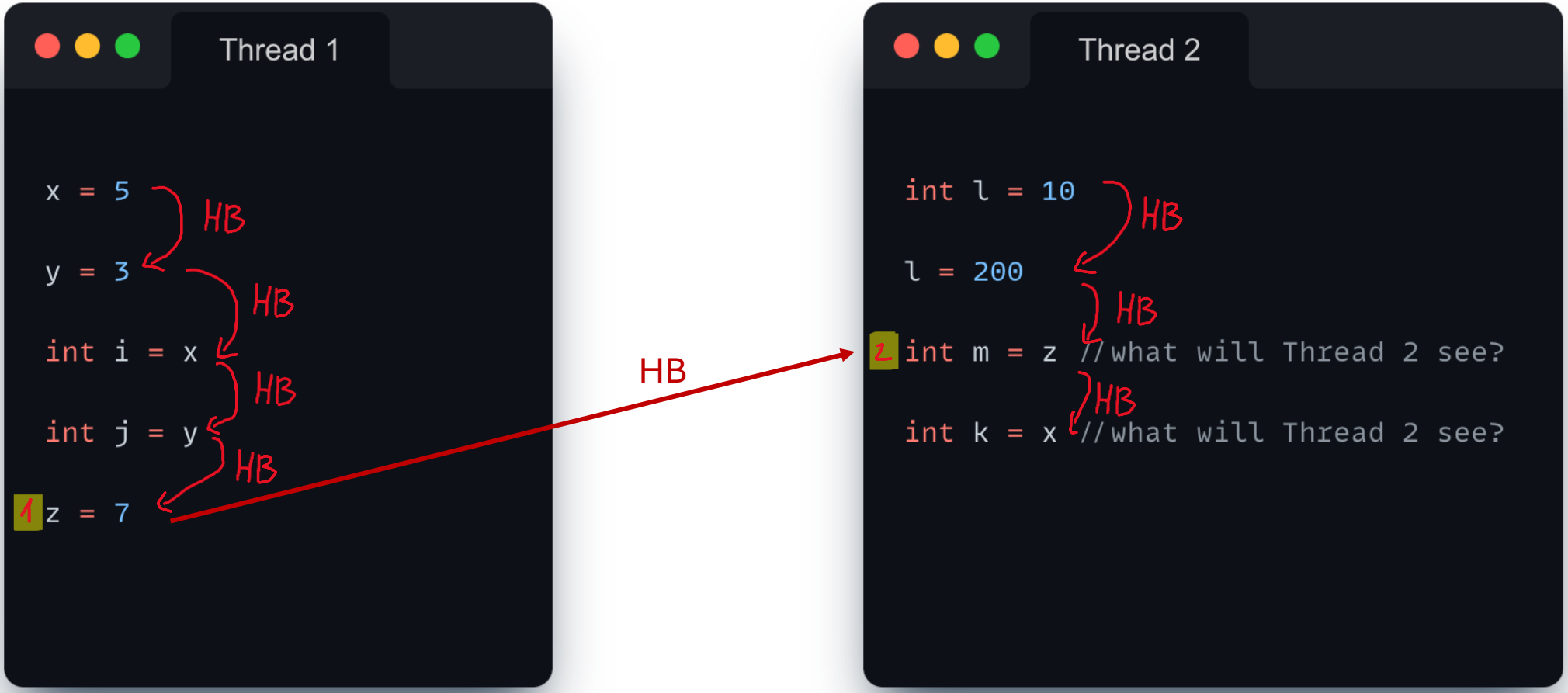
So what does k = read(x) see?

int x,y = 0, volatile int z = 0 - SO: write(z, 7) --> m = read(z)

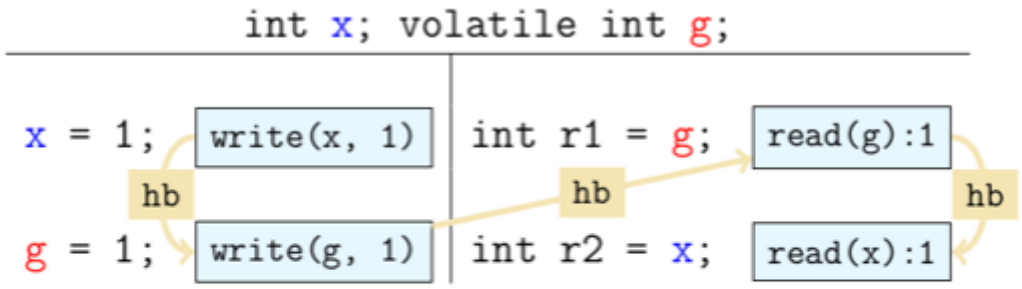


How about now?

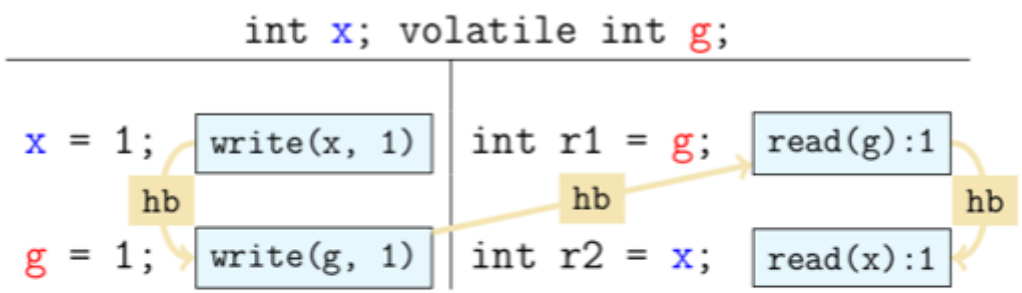
int x,y = 0, volatile int z = 0 - SO: write(z, 7) --> m = read(z)

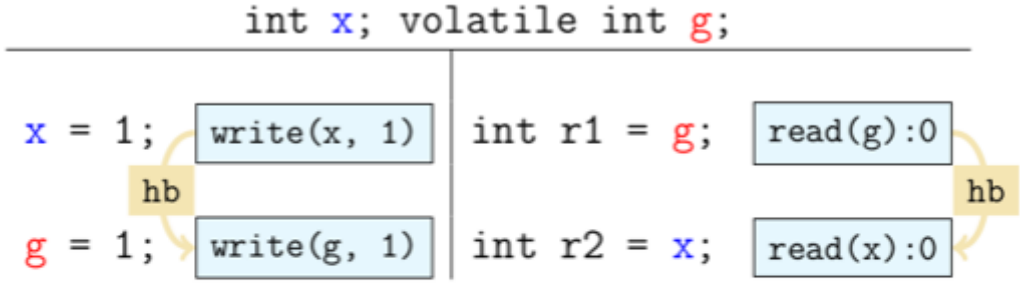


Example


int x; volatile int g;
x = 1;
write(x, 1)
int r1 = g;
read(g):1
hb
hb
hb
g = 1;
write(g, 1)
int r2 = x;
read(x):1

Example

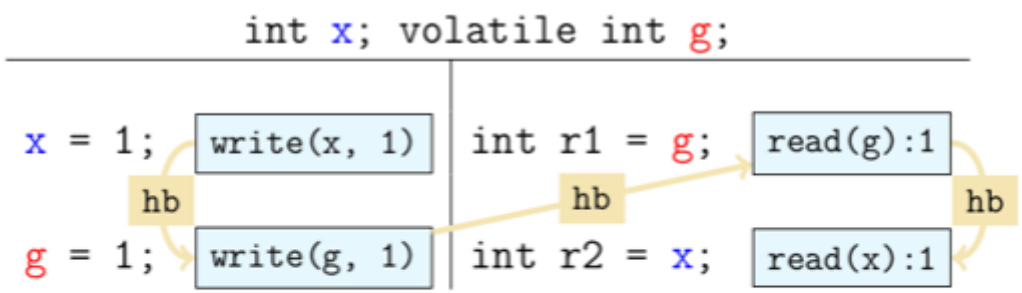




Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$

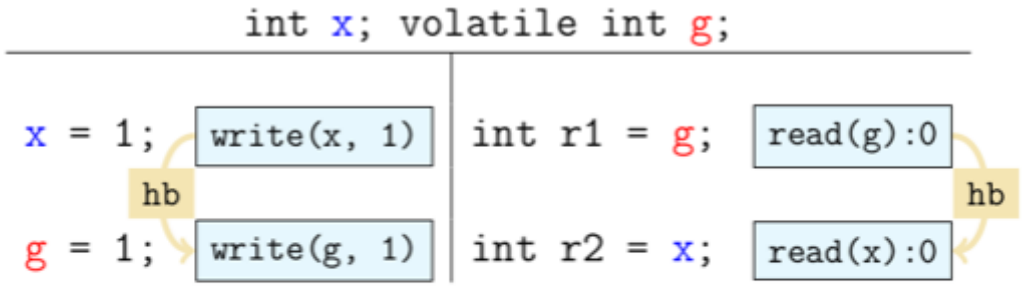
$$(r1, r2) = (1, 1)$$

Example



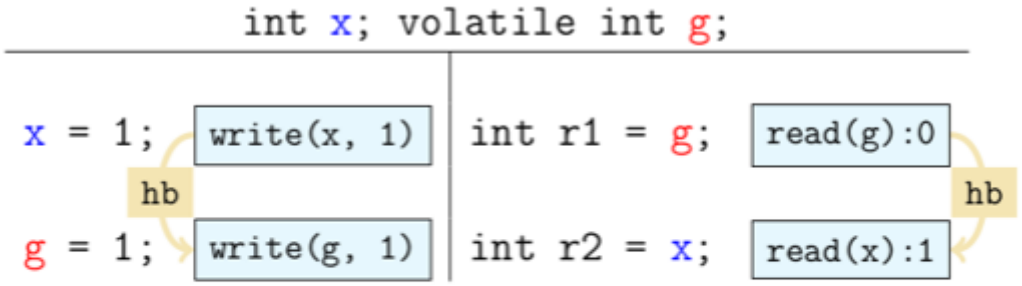
Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$

$(r1, r2) = (1, 1)$

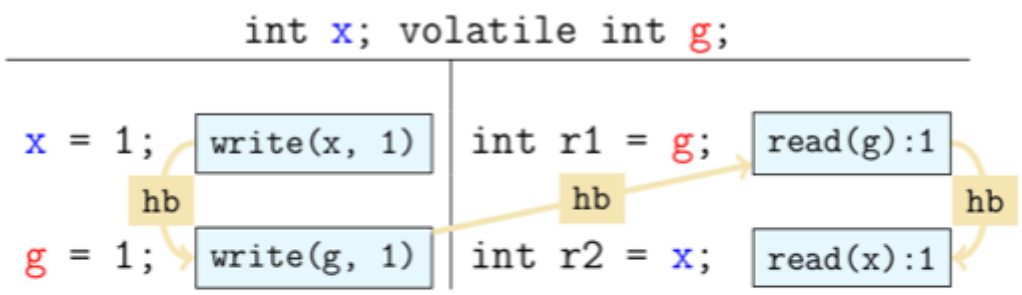


Case 2: HB consistent, observe the default value

$(r1, r2) = (0, 0)$

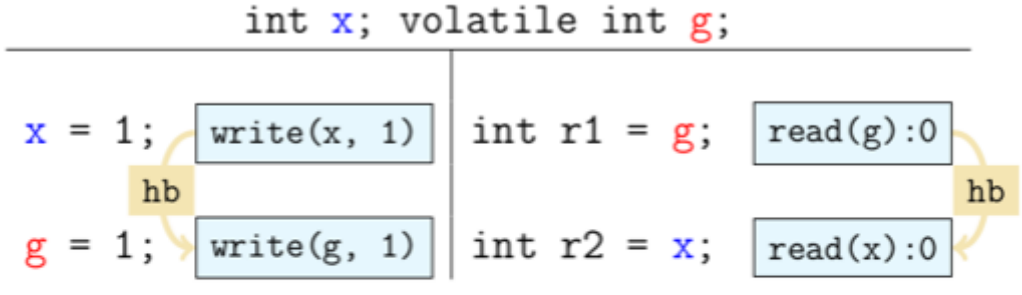


Example



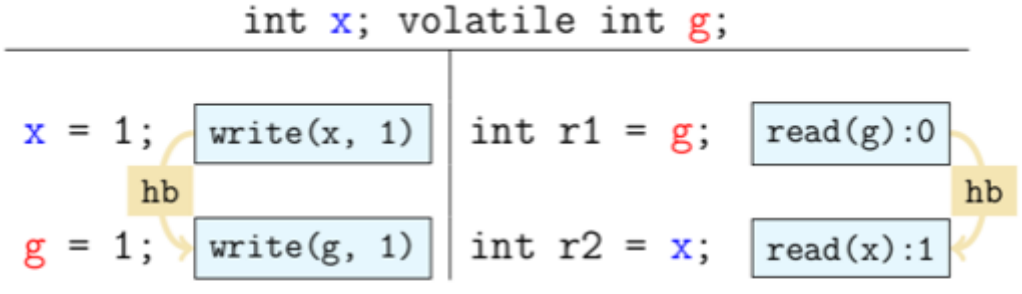
Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$

$(r1, r2) = (1, 1)$



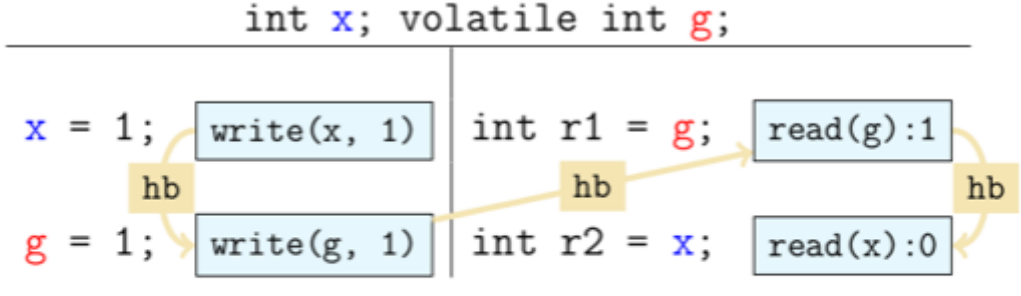
Case 2: HB consistent, observe the default value

$(r1, r2) = (0, 0)$

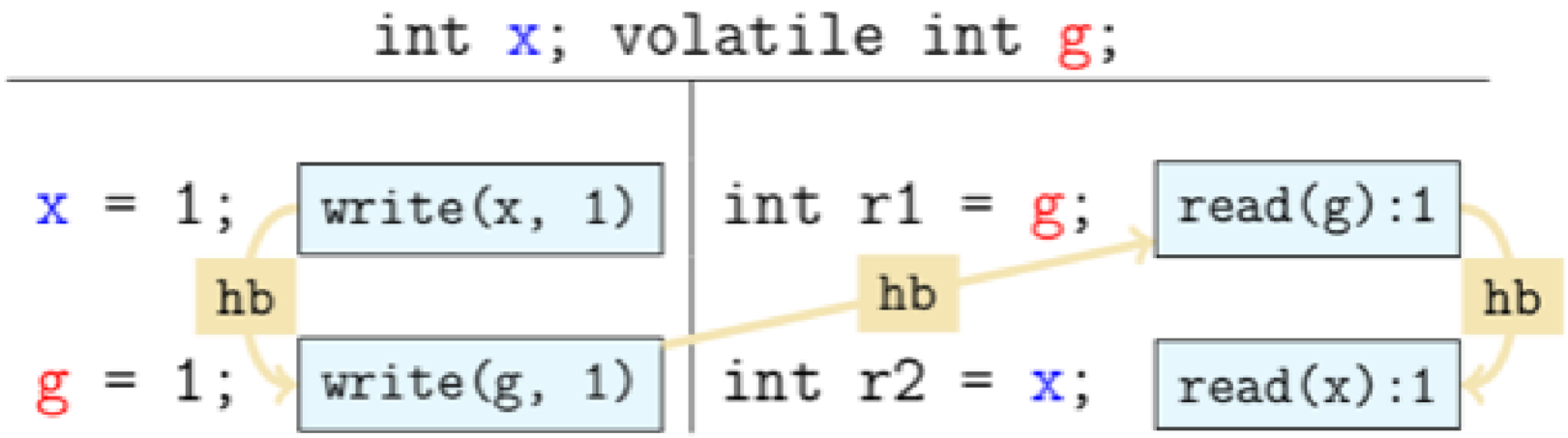


Case 3: HB consistent (!), reading via race!

$(r1, r2) = (0, 1)$



Examples



- Initial value of x,y is 0.
- We can either get r1,r2 = (0,0), (1,1) or (0,1) NOT (1,0) from this code!

Atomic operations

- An atomic action is one that effectively happens at once i.e. this action cannot stop in the middle nor be interleaved
- It either happens completely, or it doesn’t happen at all.
- No side effects of an atomic action are visible until the action is complete
- This essentially means that other threads think that the change happened in an instant

Atomic registers

- Atomic registers => support read and write, nothing else
- Usually we think of reads / writes as atomic, i.e., if we write a line such as x=1 in pseudocode we assume it happens atomically and is globally visible.

- This is not true in Java (unless x is e.g., AtomicInteger)
- Volatile makes it globally visible (but not atomic in all cases)

Atomic registers

- An operation such as x++ (with x being an atomic register) is NOT atomic!
  - Three steps: v = read(x), increment v, write(x, v)

- Problem with atomic registers:
  - Need O(n) space to synchronize n threads (if we only have atomic read write) -> bad
  - Fix: support more than read/write in an atomic operation

Atomic Registers

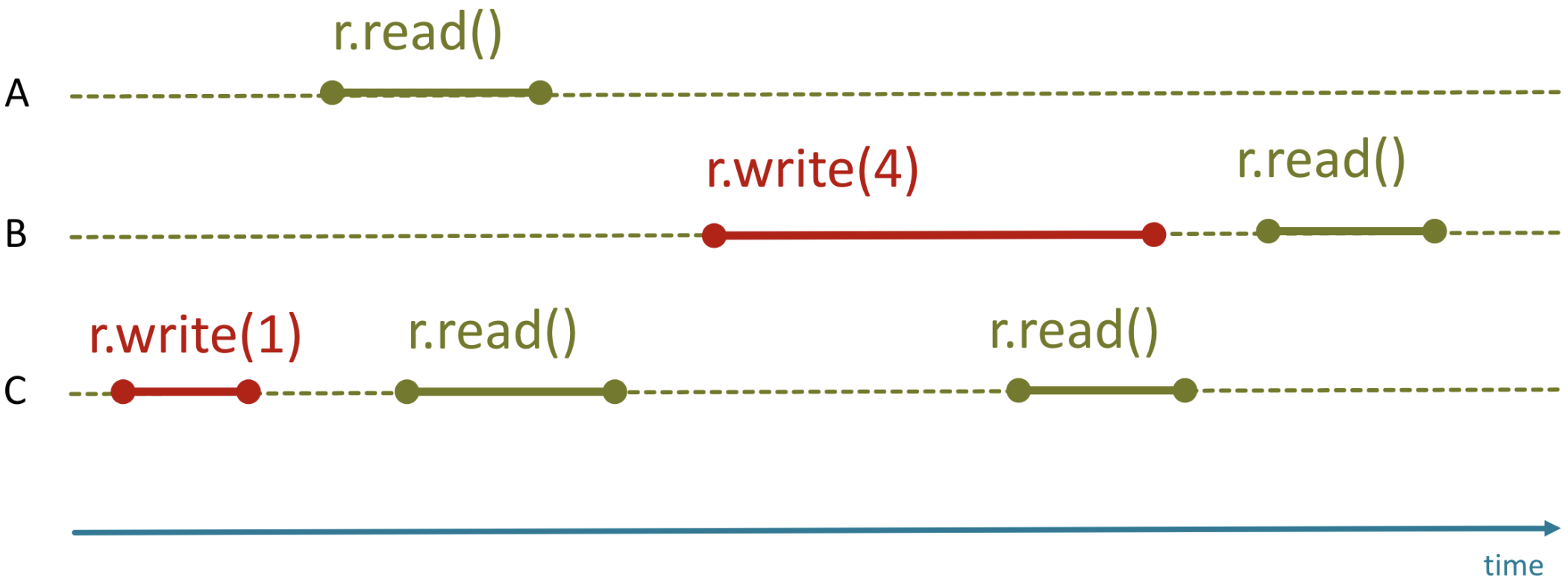
Register: basic memory object, can be shared or not
i.e., in this context register ≠ register of a CPU

Register *r* : operations *r.read()* and *r.write(v)*

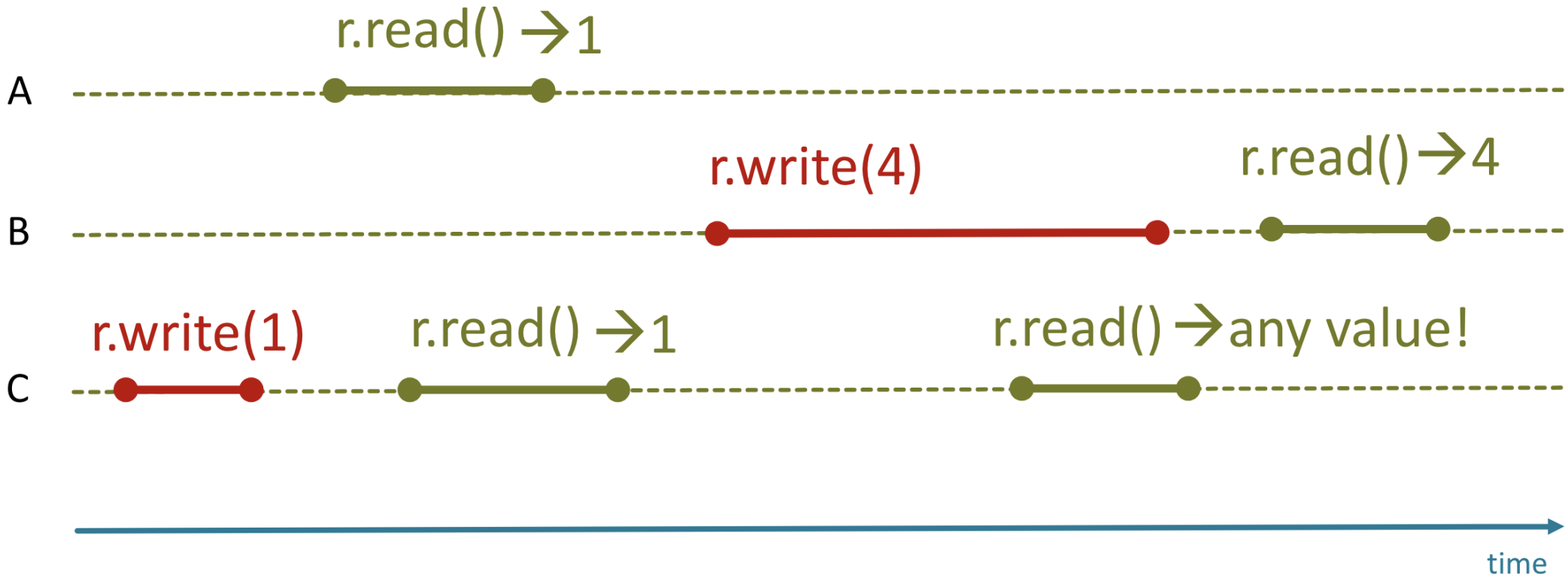
Atomic Register:

- **An invocation *J* of *r.read* or *r.write* takes effect at a single point $\tau(J)$ in time**
- **$\tau(J)$ always lies between start and end of the operation *J***
- **Two operations *J* and *K* on the same register always have a different effect time $\tau(J) \neq \tau(K)$**
- **An invocation *J* of *r.read()* returns the value *v* written by the invocation *K* of *r.write(v)* with closest preceding effect time $\tau(K)$**

Atomic Registers



Atomic Registers



Hardware support for atomic operations

Different atomic operations have been proposed, unclear which is best

- Test-And-Set (TAS)
- Compare-And-Swap (CAS)
- Load Linked / Store Conditional
- http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

Hardware Semantics

boolean TAS(*memref* s)

atomic

```
if (mem[s] == 0) {
        mem[s] = 1;
        return true;
} else

   return false;
```

int CAS (*memref* a, int old, int new)

atomic

```
oldval = mem[a];

if (old == oldval)

        mem[a] = new;

return oldval;
```

java.util.concurrent.atomic.AtomicBoolean

```
boolean set();

boolean get();

boolean compareAndSet(boolean expect, boolean update);

boolean getAndSet(boolean newValue);
```

atomically set to value `update` iff current value is `expect`. Return true on success.

sets `newValue` and returns previous value.

Now we know how getAndIncrement is implemented!

```java
/**
 * Atomically increments by one the current value.
 *
 * @return the previous value
 */
public final int getAndIncrement() {
  for (;;) { //same as while(true)
      int current = get();
      int next = current + 1;
      if (compareAndSet(current, next))
          return current;
  }
}
```

Source: https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/atomic/AtomicInteger.java

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

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;
```

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;
```

Compare and Swap (CAS)

```
Init (lock)
        lock = 0;
```

```
Acquire (lock)
        while (CAS(lock, 0, 1) != 0);
```

```
Release (lock)
        CAS(lock, 1, 0);
```

In 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

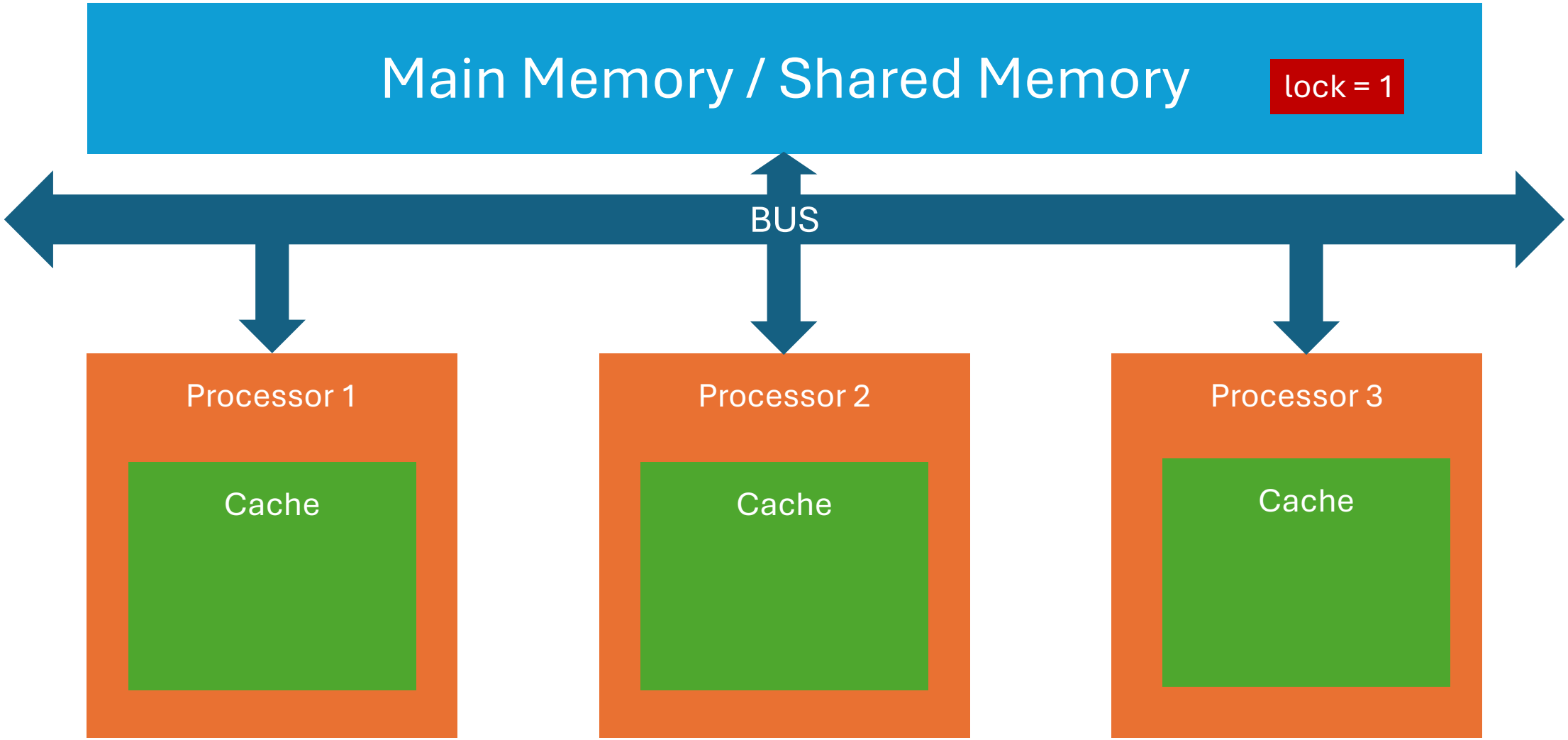
Cache Coherency Protocol ☹

We have a sequential bottleneck!

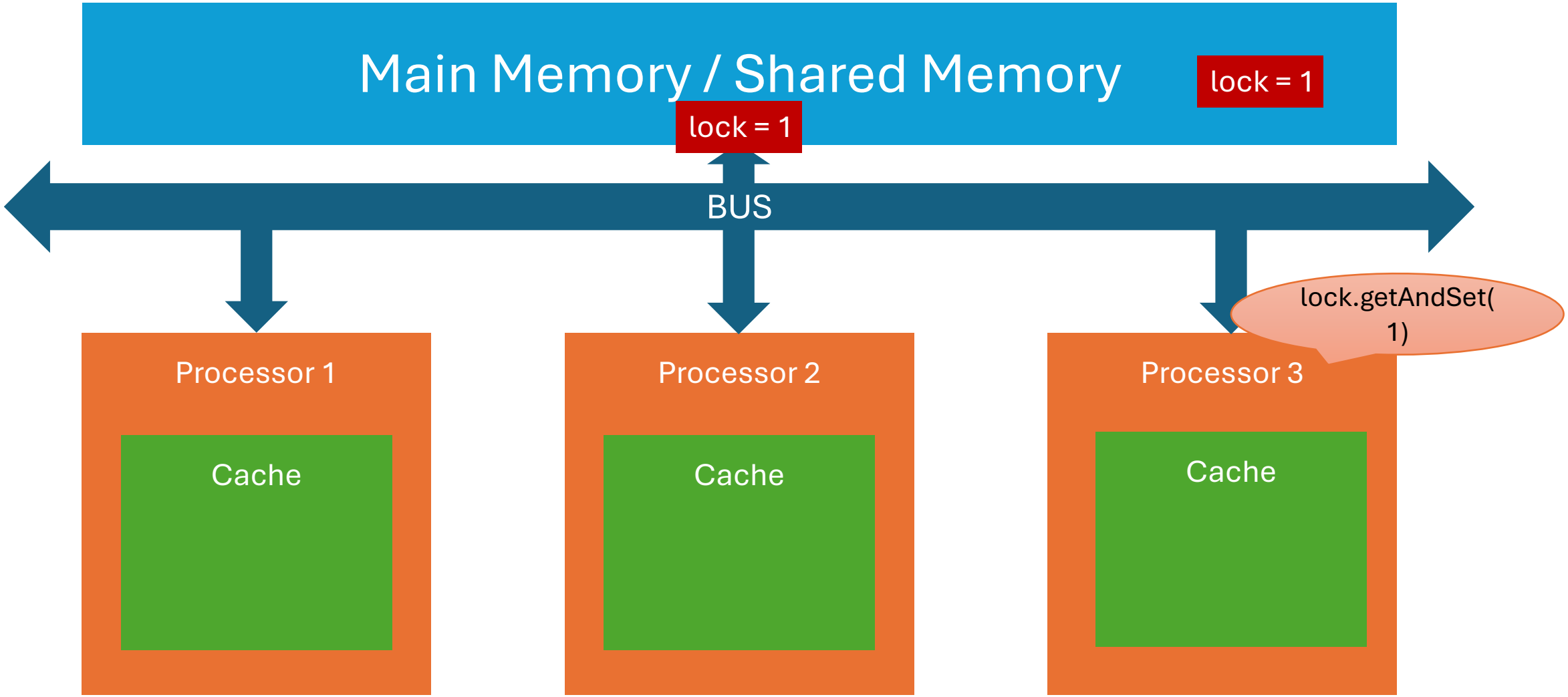
Each call to getAndSet() invalidates cached copies! => Threads need to access memory via Bus => Bus Contention!

“[…] 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

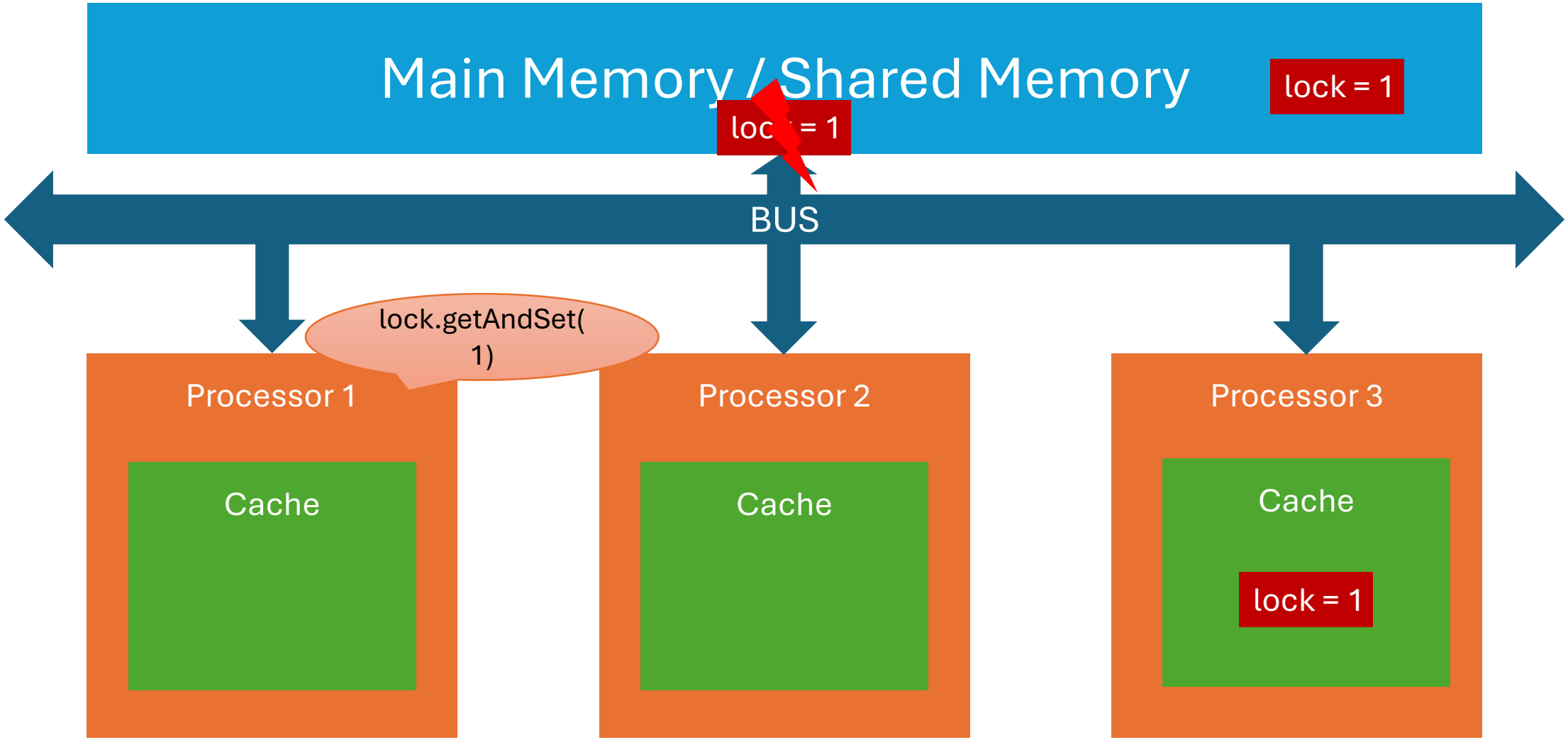
Let’s visualize this



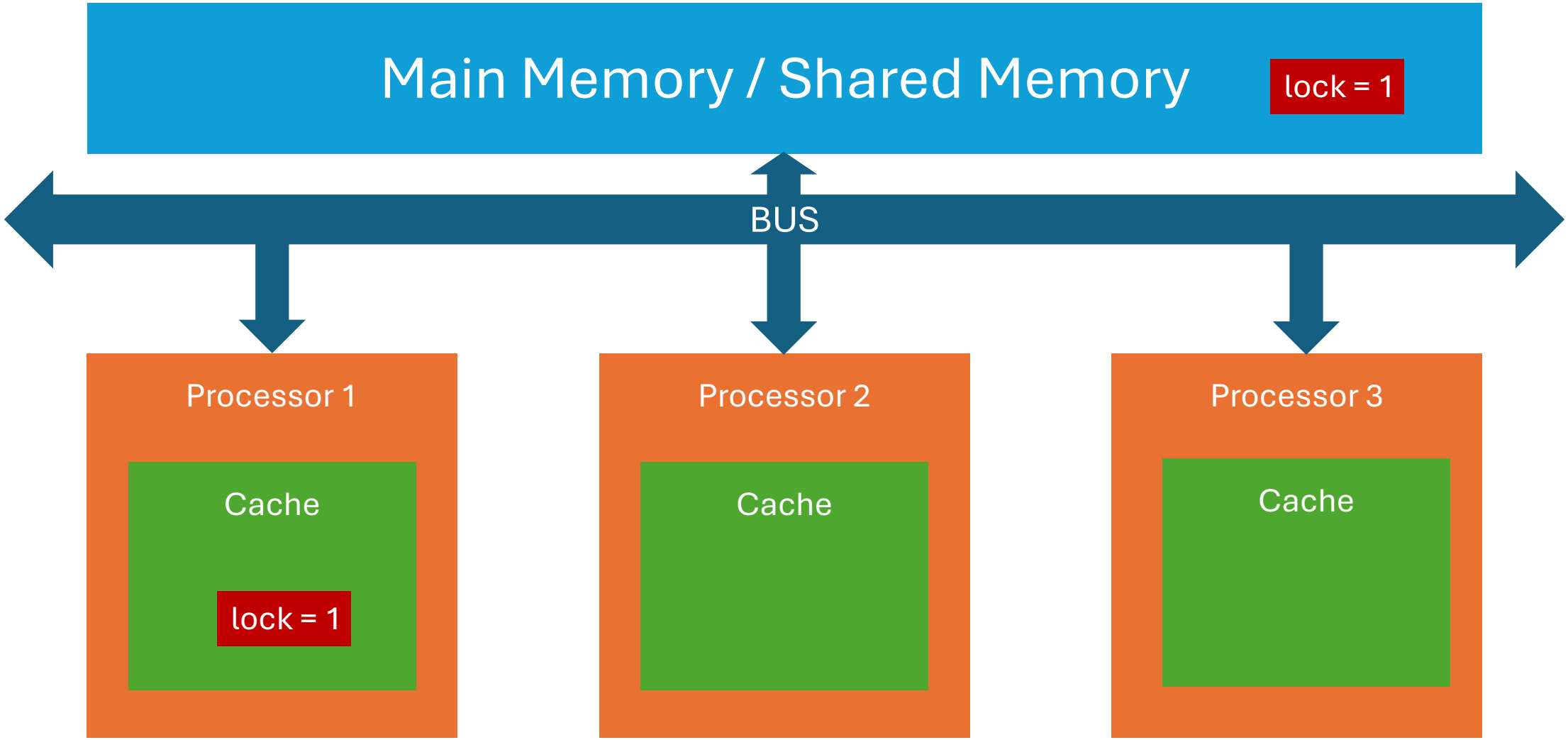
Lets visualize this



Lets visualize this



Lets visualize this



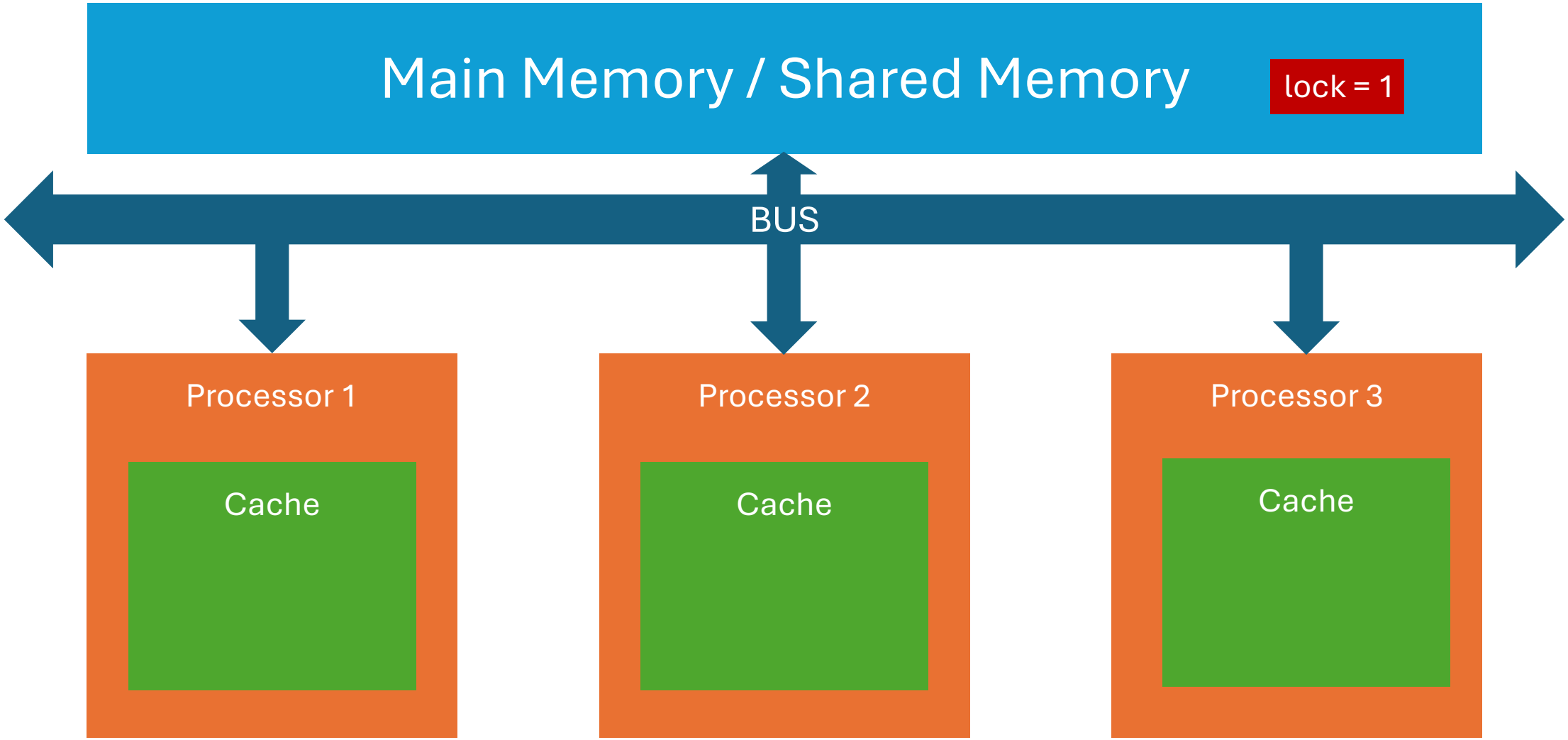
TATAS

- Idea: Use normal operation to read first, try TAS only if first read returns 0
- Helps a bit. But what about the case where we see 0 first, then 1 in TAS? Can this happen?
  - Yes, and the more threads the more likely ☺

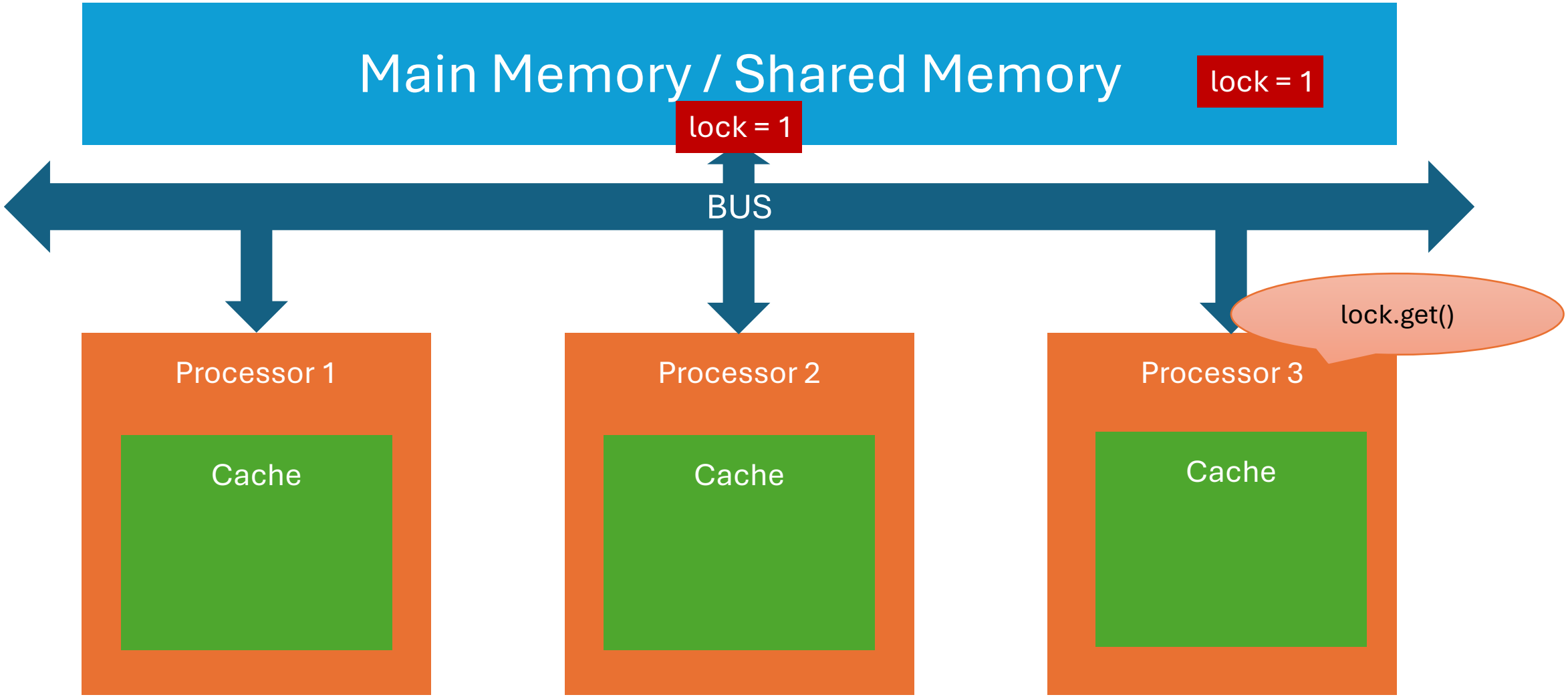
Lets try spinning on local cache

```
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);
  }

}
```

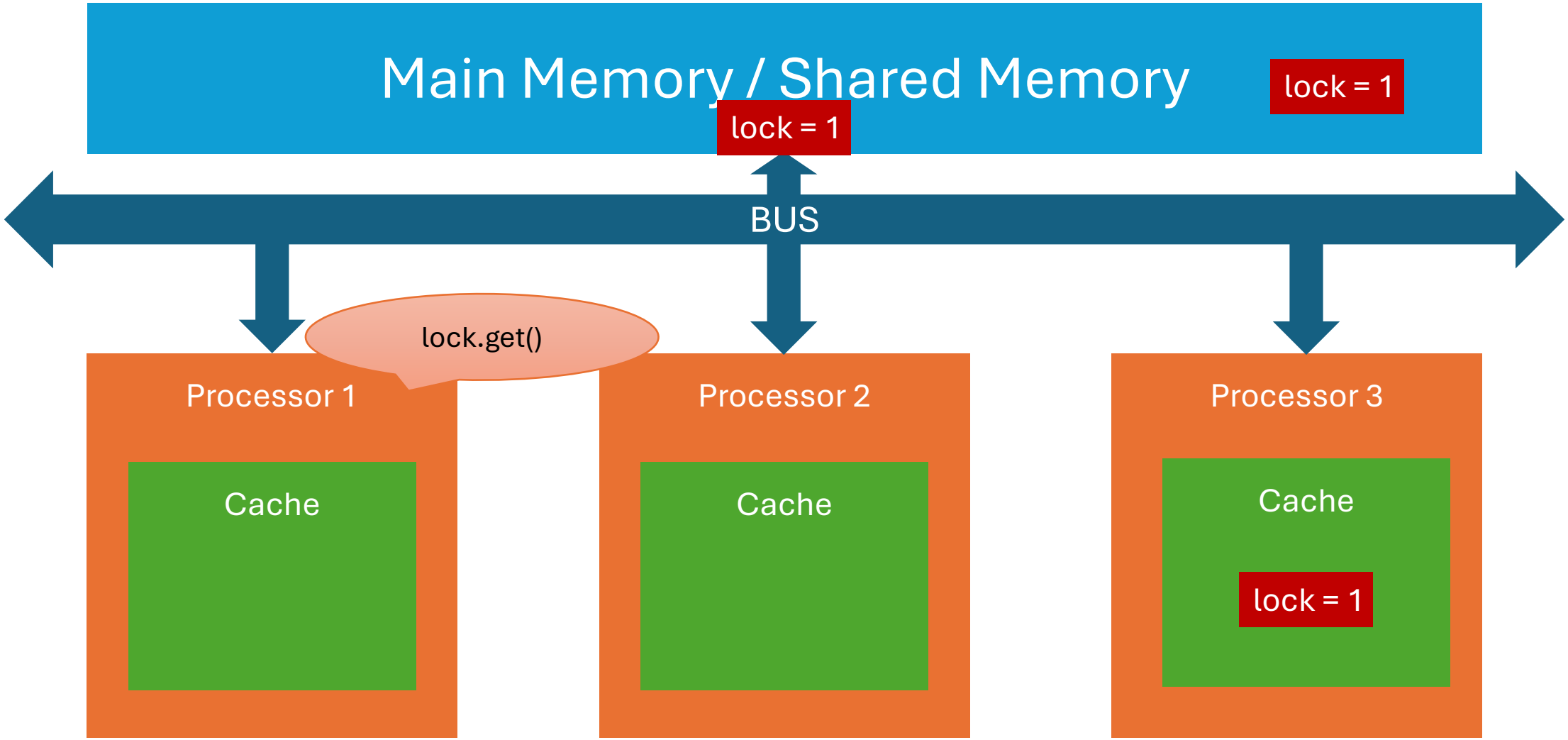
Lets visualize this



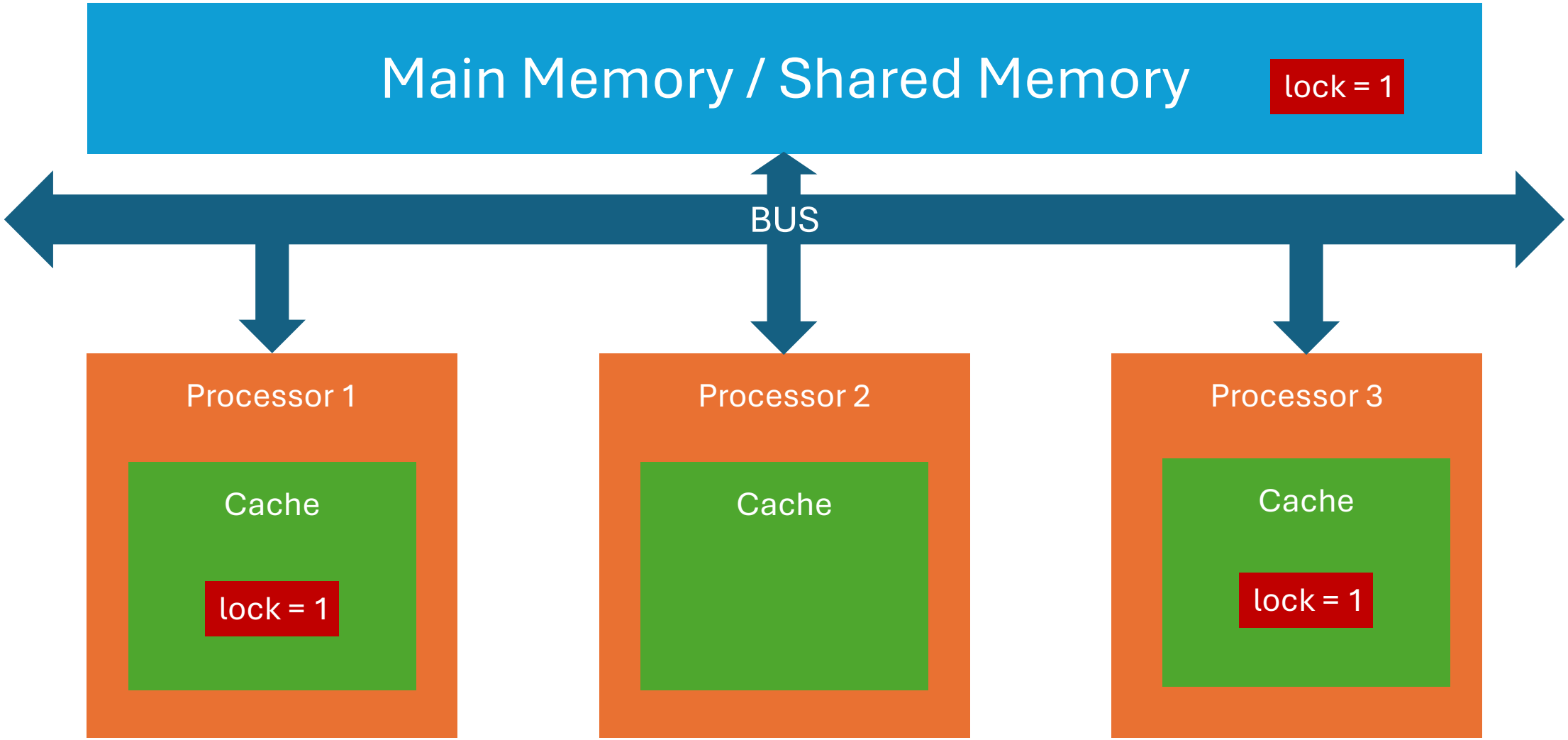
Lets visualize this



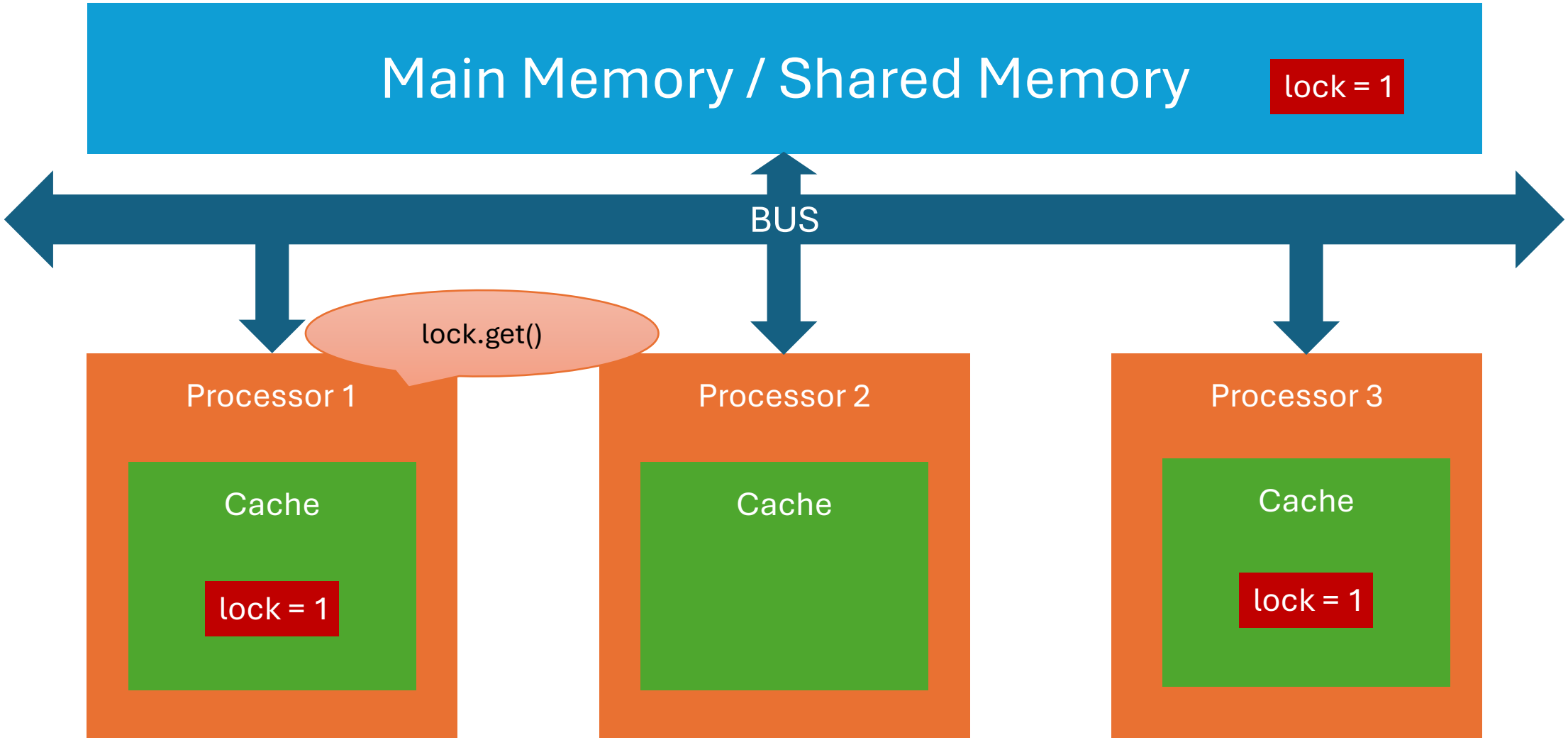
Lets visualize this



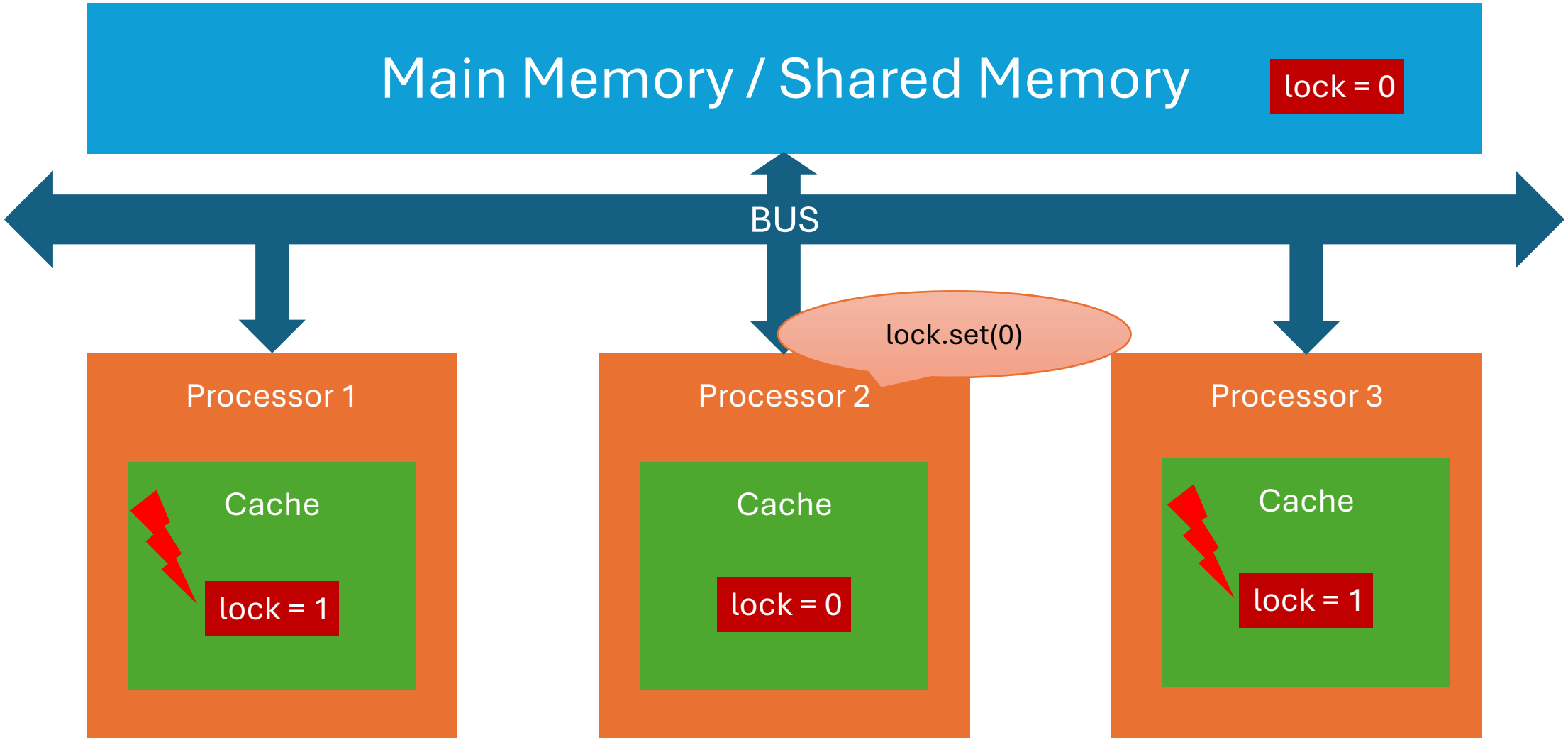
Lets visualize this



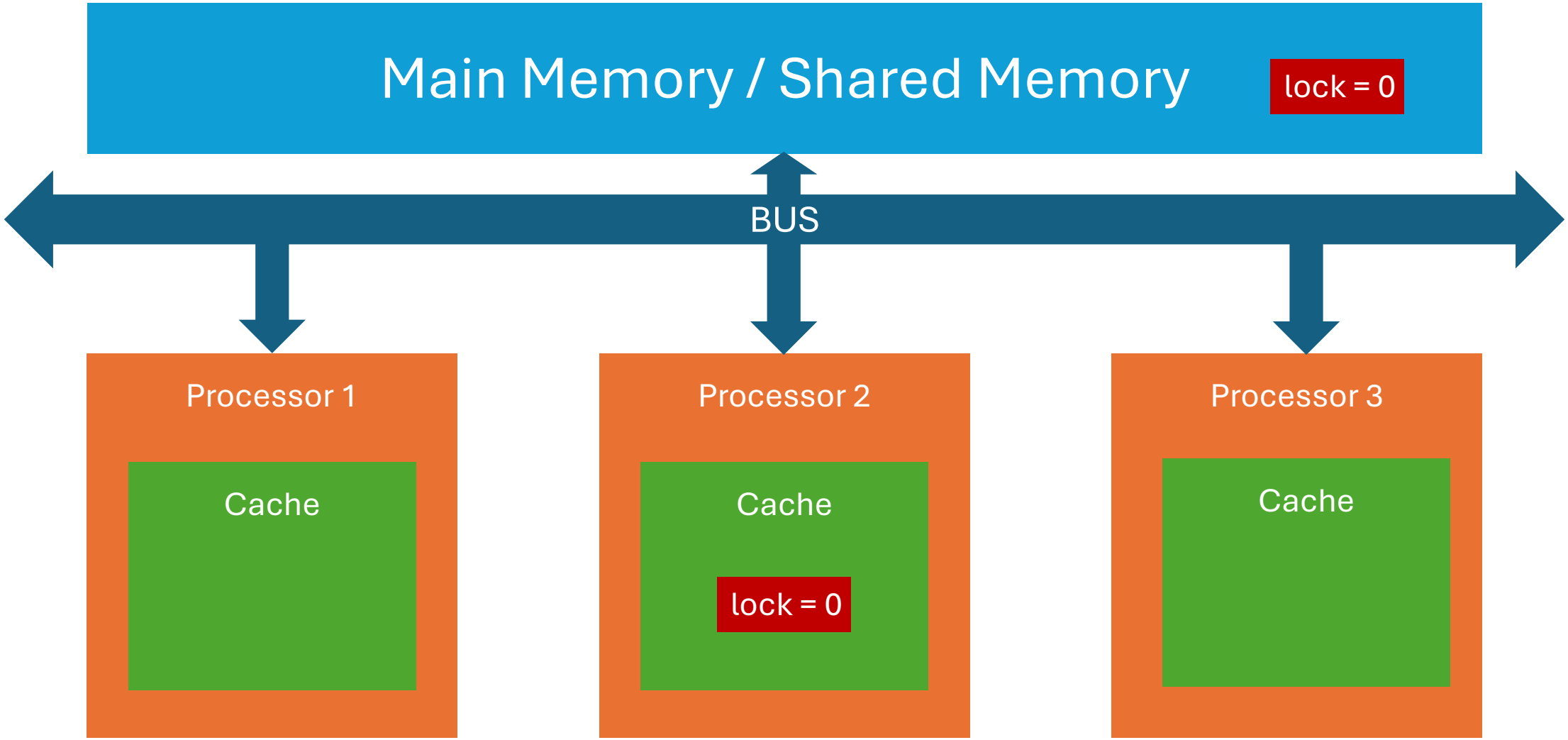
Lets visualize this



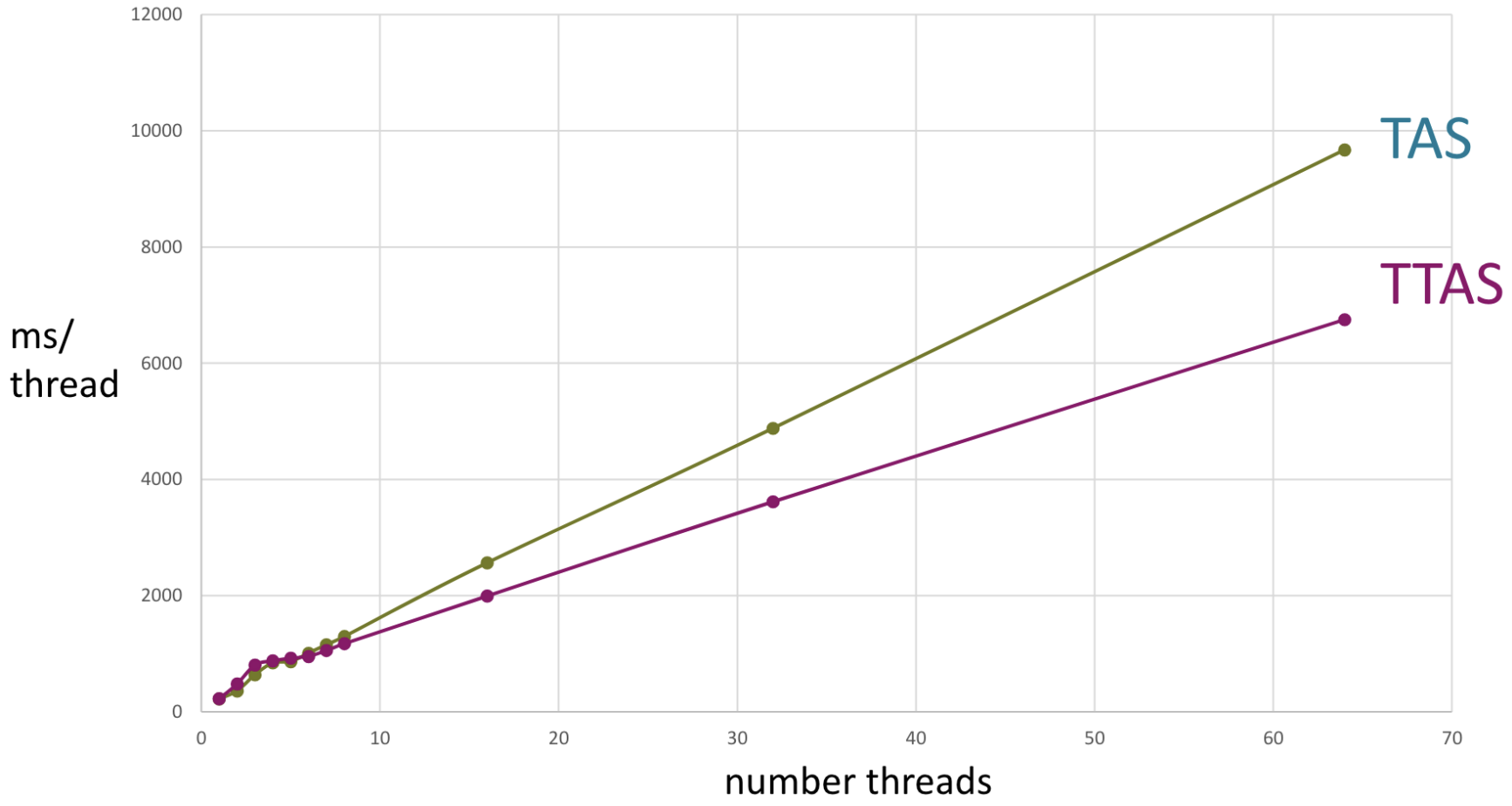
Lets visualize this



Now the whole problem repeats



It only helped a little bit



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

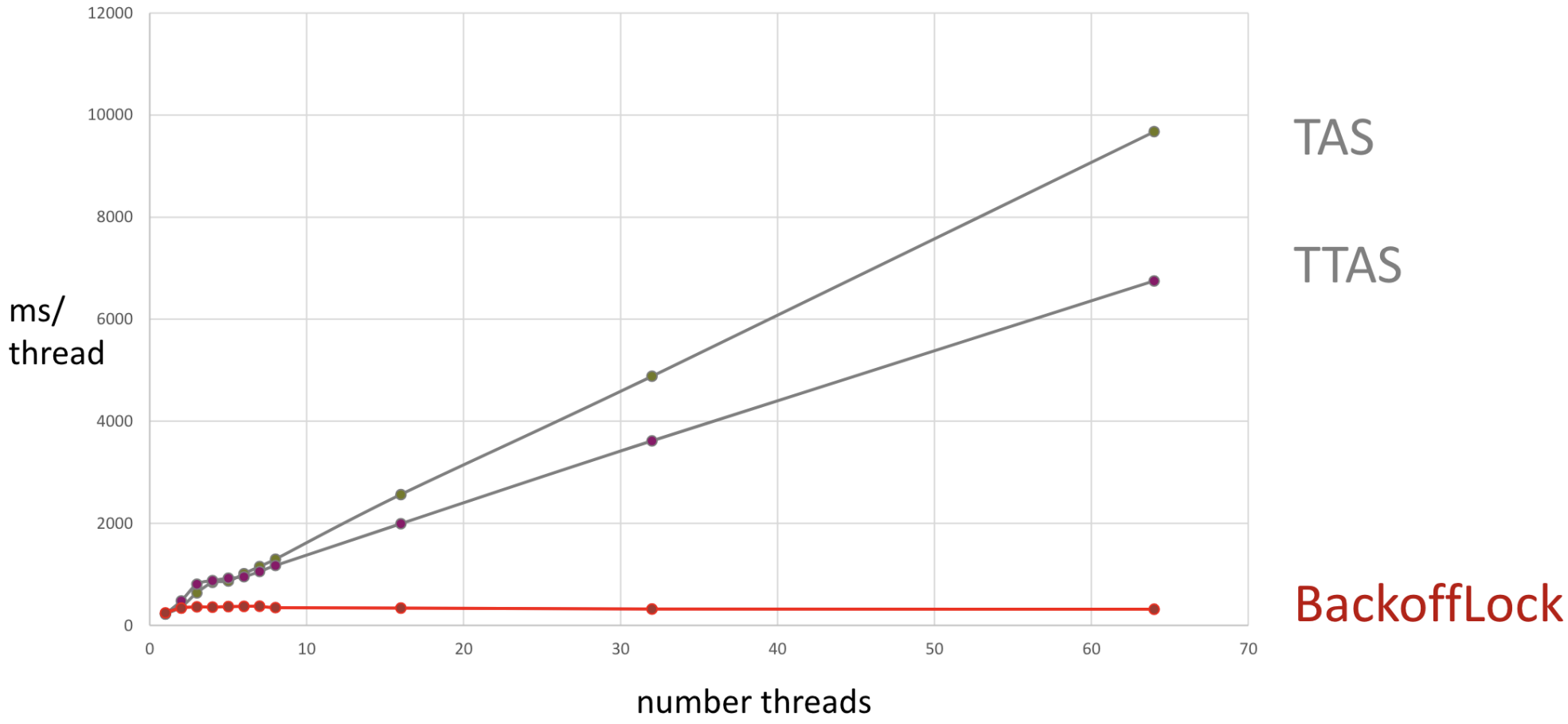
Idea: Each time TAS fails, wait longer until you re-try

- Backoff must be random!

Exponential Backoff

- Idea: Each time TAS fails, wait longer until you re-try
- Works well, must tune parameters (how long to wait initially, when to stop increasing)
- Same concept in networks, people talking in a high-latency zoom call, etc.

Nice!


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

Lock with compare and set

- Now you see why we like atomics so much. It’s much simpler!

PProgFS25 - Jonas Wetzel

```
import java.util.concurrent.atomic.AtomicBoolean;

class SpinLock {
    private AtomicBoolean locked = new
AtomicBoolean(false);

    public void lock() {
        while (!locked.compareAndSet(false, true)) {
            // Keep spinning until we successfully set
locked to true
        }
    }

    public void unlock() {
        locked.set(false);
    }
}
```

Performance of Atomic Lock

- High contention makes performance bad

How do we build locks without atomic

- That also give certain guarantees like fairness
- In our atomic spin lock an unlucky thread might never get to the CS

We need to watch out for: Deadlock

- Circular dependency between resources/lock and threads
- Nobody can make progress

- Avoid by introducing global order in which locks are taken
  - Cannot have circles now since all dependencies go “in one direction”
- Or by not using locks at all! (Lock-free, wait-free, more later)

Deckers Lock

- Each thread sets its flag[id] = true to indicate that it wants access to critical section
- If other thread also wants access, they use turn variable to decide who goes first
- If it's not thread's turn, it backs off, resets its flag, and waits for its turn
- After exiting critical section, thread gives turn to the other thread and resets its flag

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

only when q tries to get lock

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

only when q tries to get lock

and q has preference

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
        non-critical section
        wantp = true
        while (wantq) {
                if (turn == 2) {
                        wantp = false;
                        while(turn != 1);
                        wantp = true; }}
        critical section
        turn = 2
        wantp = false
```

- only when q tries to get lock
- and q has preference
- let q proceed
- and wait
- and try again

```
Process Q
loop
        non-critical section
        wantq = true
        while (wantp) {
                if (turn == 1) {
                        wantq = false
                        while(turn != 2);
                        wantq = true; }}
        critical section
        turn = 1
        wantq = false
```

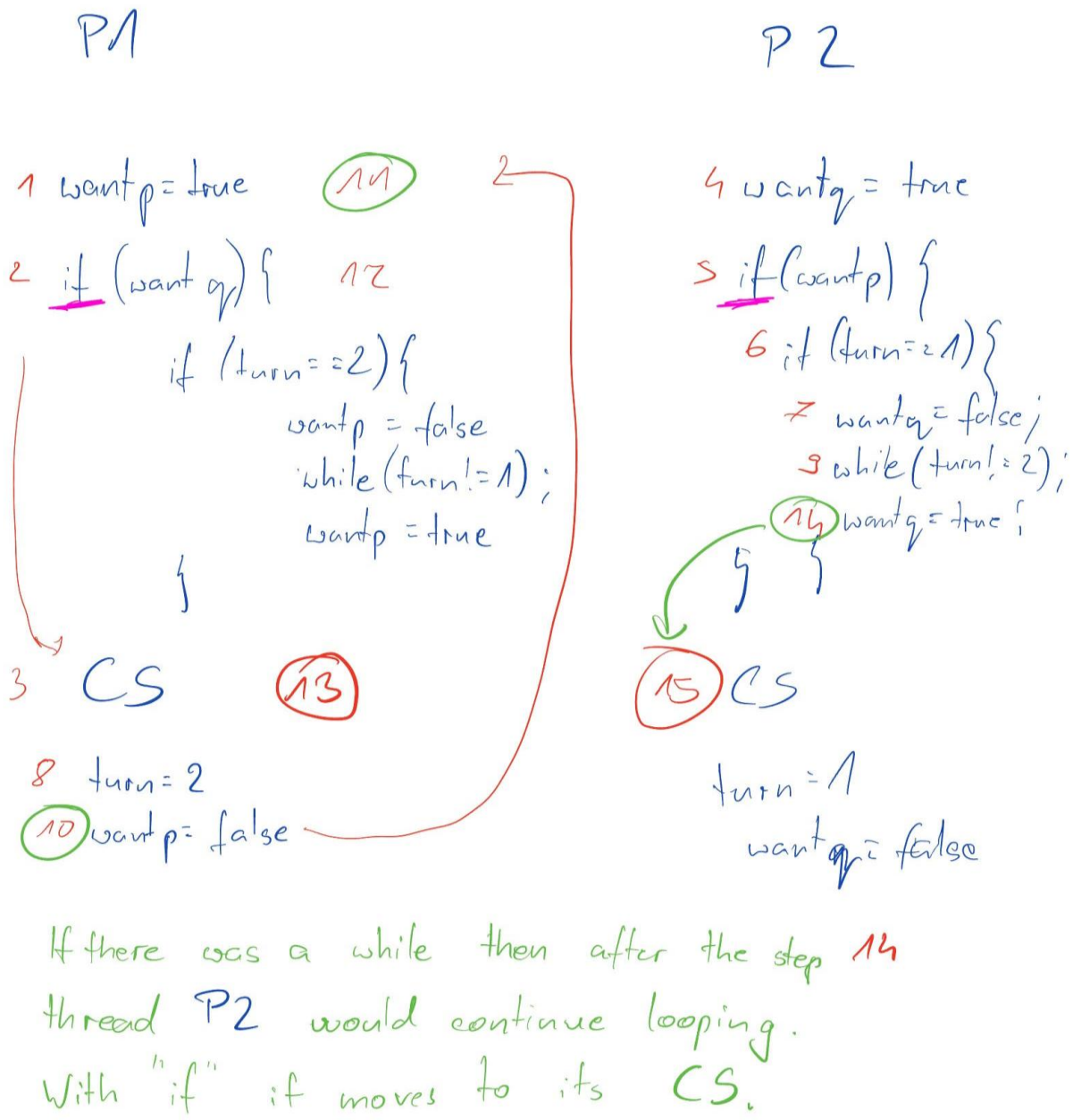
Deckers Lock

- ✅ Ensures mutual exclusion (only one thread enters the critical section at a time)
- ✅ Avoids deadlock by using the turn variable
- ✅ Provides fairness (both threads get their turn)
- ❌ Limited to two threads (doesn’t scale well)

Can we change while
to if in Deckers lock?

Can we change while to if in Deckers lock?

No!



Petersons Lock

- Each thread sets flag[id] = true to indicate it wants access to the critical section.
- The thread gives priority to the other thread by setting turn = other.
- If the other thread also wants access (flag[other] == true) and it’s still its turn, the thread waits.
- Once it gets access, it enters the critical section.
- After exiting, the thread resets flag[id] = false so the other thread can proceed.

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

But isn't the write to victim a data race?

More concise than Decker: Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

Process P (1)

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

Process Q (2)

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

Peterson Lock

- ✅ **Ensures mutual exclusion** (only one thread enters the critical section at a time)
  ✅ **Prevents deadlock** (always allows progress)
  ✅ **Fair (bounded waiting)** (no thread is starved forever)
- ✅ **simpler** than Deckers Lock
  ❌ **Works only for two threads** (not scalable)

Peterson Lock

- ✅ **Ensures mutual exclusion** (only one thread enters the critical section at a time)
  ✅ **Prevents deadlock** (always allows progress)
  ✅ **Fair (bounded waiting)** (no thread is starved forever)
- ✅ **simpler** than Deckers Lock
  ❌ **Works only for two threads** (not scalable)

- Bakery Lock allows us to extend Peterson Lock Idea to n Threads!

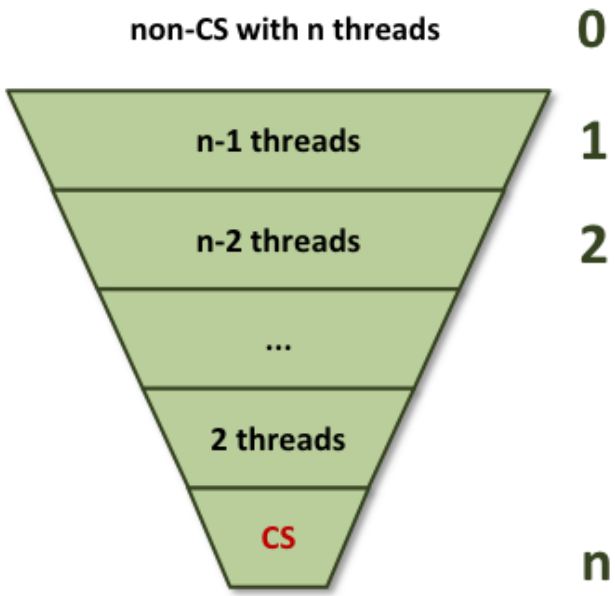
Filter Lock

```
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



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;
```

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

Extra Theory

Semaphores

and Barriers

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

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)
}
```

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

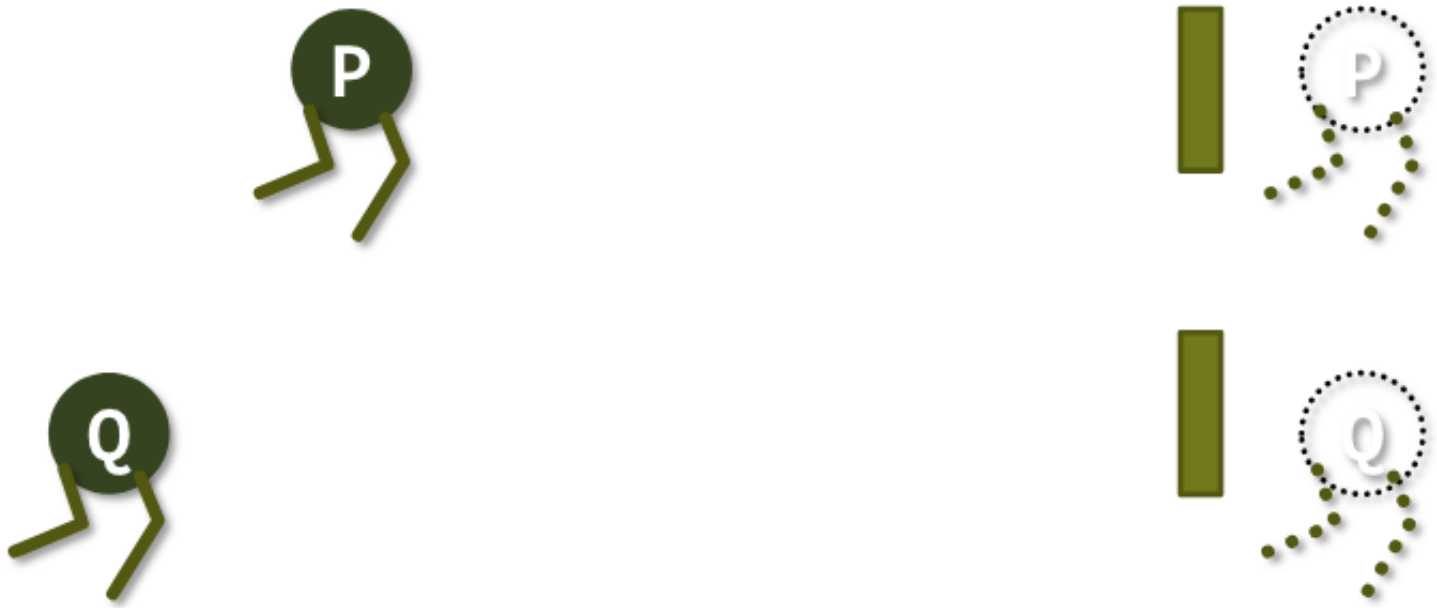
x>0 → x threads will be let into “critical section”

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

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?

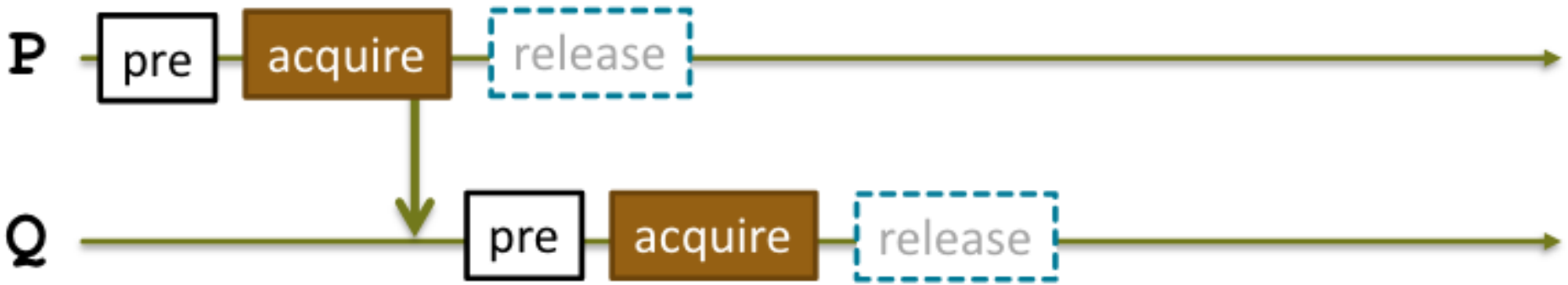
Synchronize Processes P and Q at one location (Rendezvous)

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)

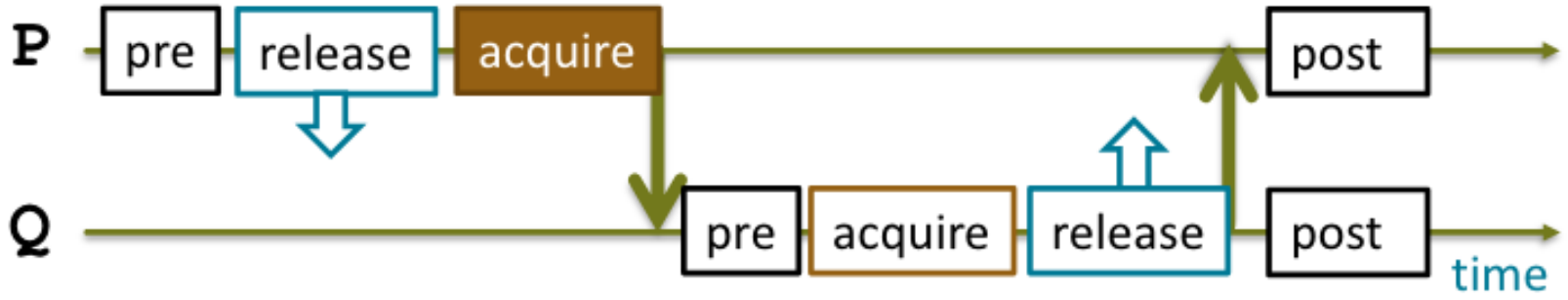
Assume Semaphores `P_Arrived` and `Q_Arrived`

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

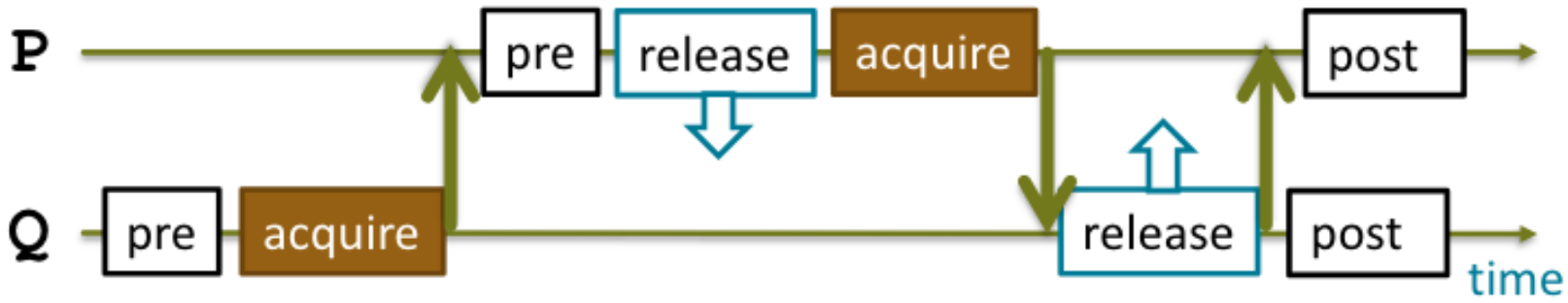
Yes, that works!

	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



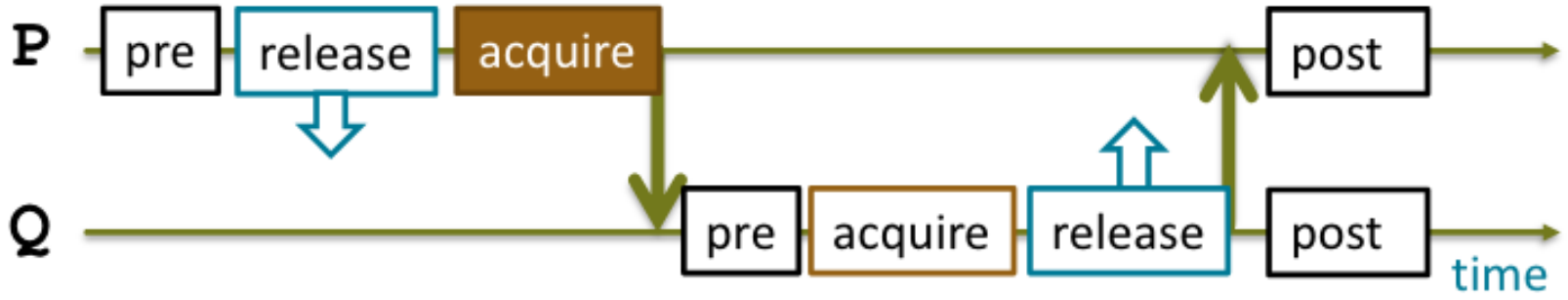
Q first



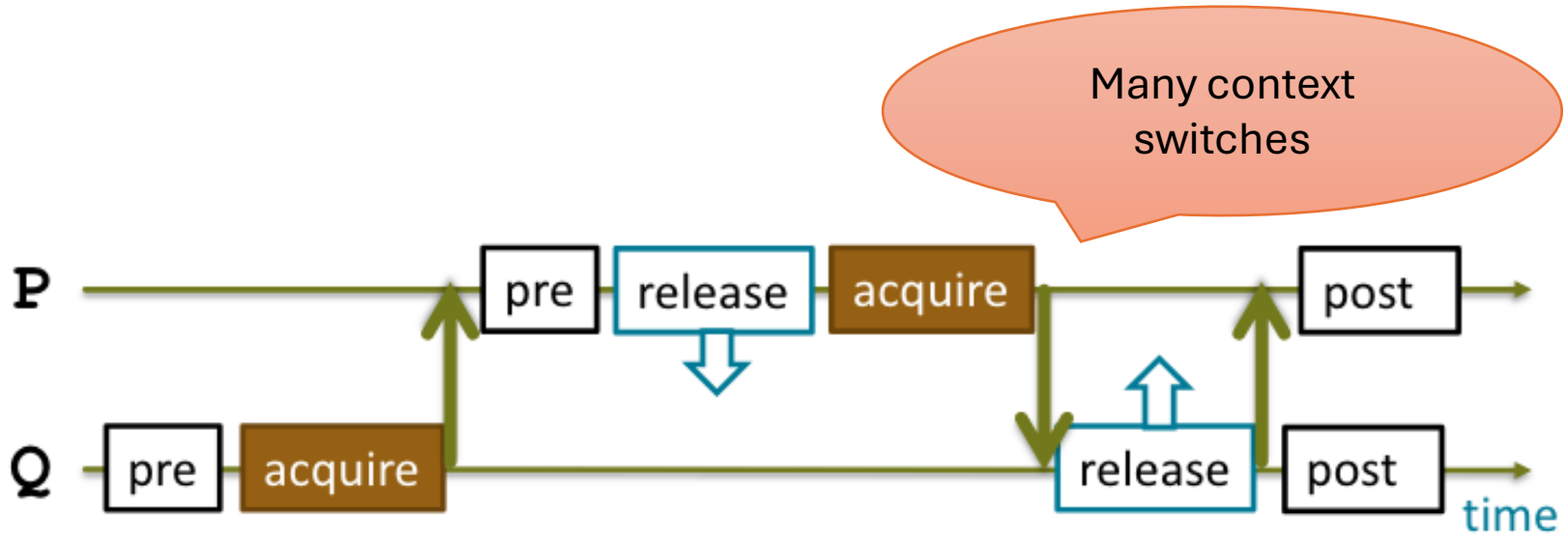
Yes, that works!

	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)

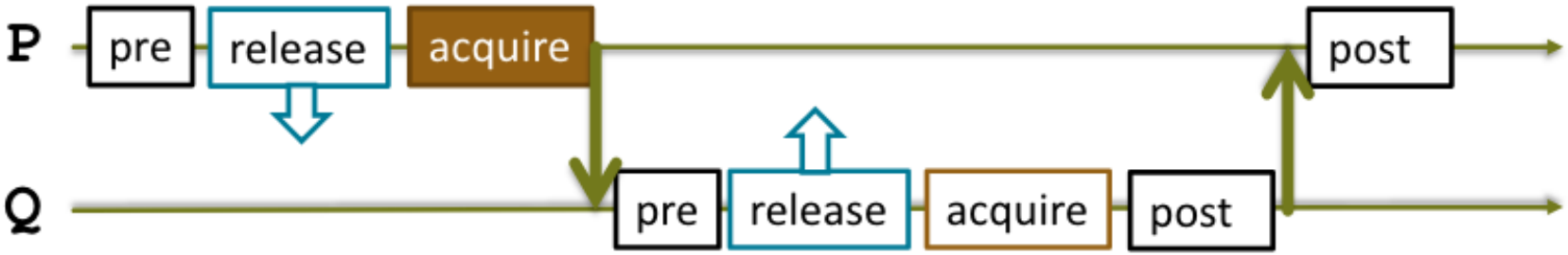
Assume Semaphores P_Arrived and Q_Arrived

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

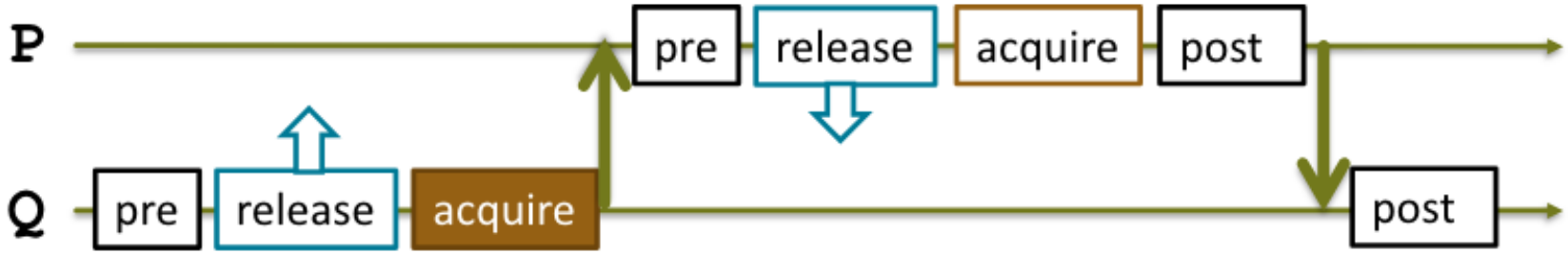
Order does no longer matter

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

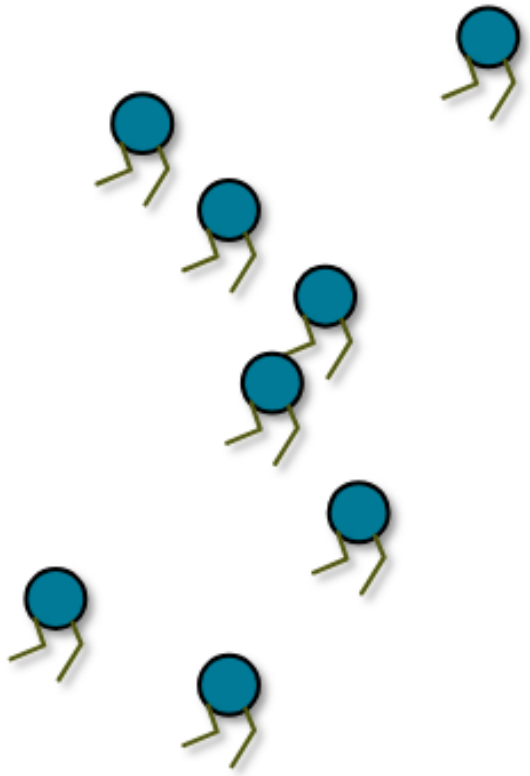
P first

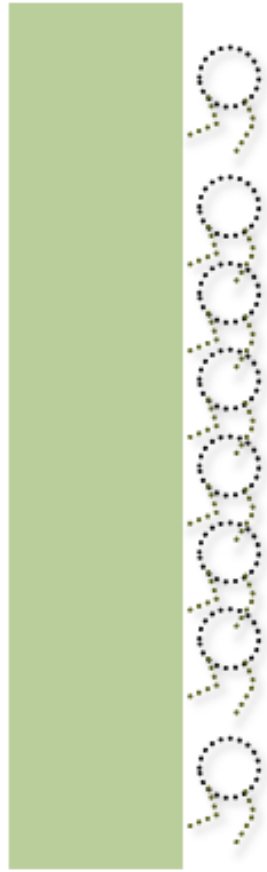


Q first

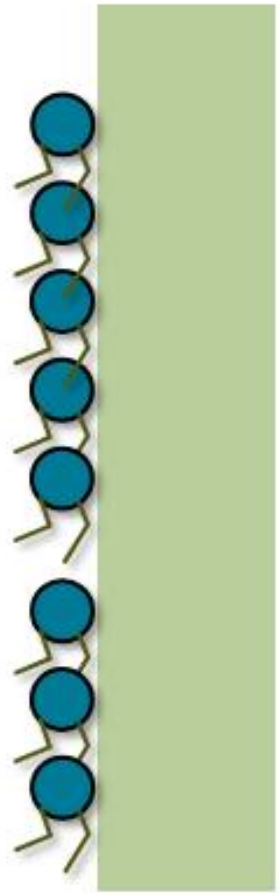


How about more than two threads? Barriers!





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	`...`			

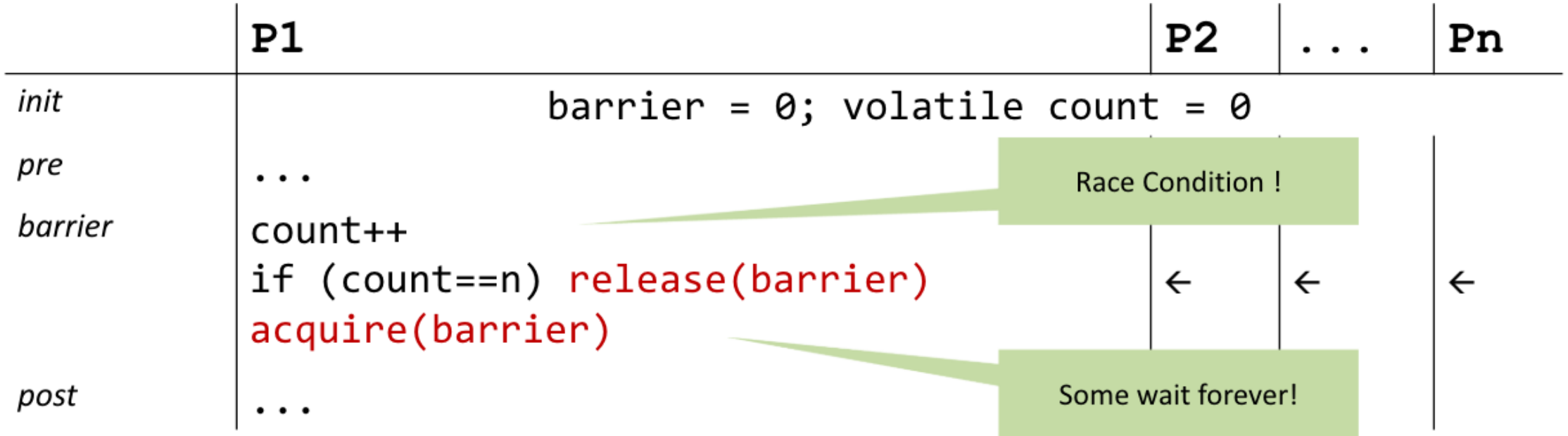
barrier	`count++`
	`if (count==n) release(barrier)`
	`acquire(barrier)`
		←	←	←

post	`...`			

Wrong

Synchronize a number (n) of processes.

Semaphore `barrier`. Integer `count`.



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	`...`			

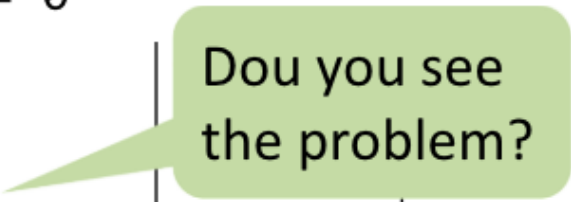
turnstile

Reusable Barrier

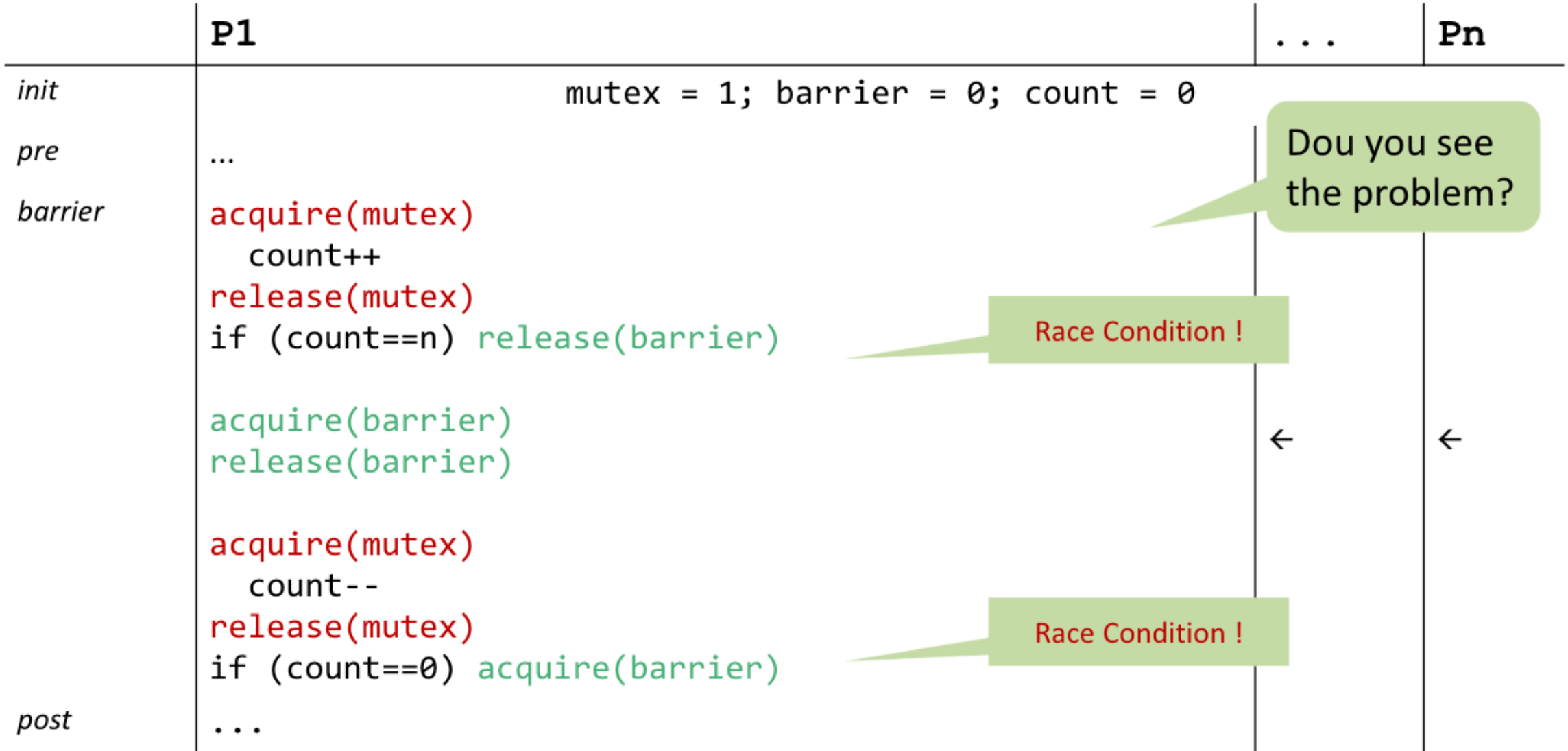
	P1	...	Pn
init	mutex = 1; barrier = 0; count = 0		
pre	…		

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)
		←	←

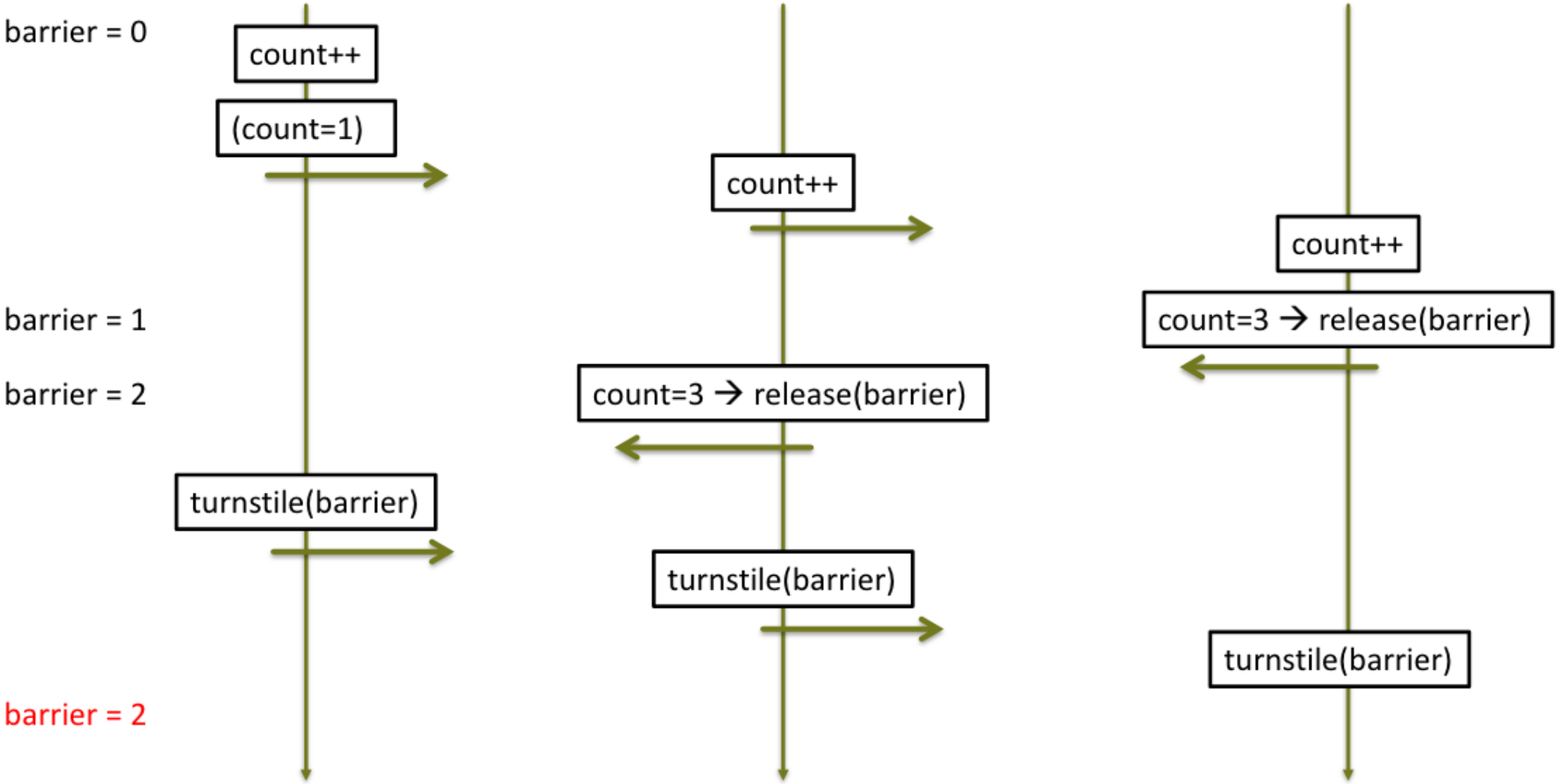
post	...		



Reusable Barrier



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

barrier

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

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
```

Plan für heute

- Organisation
- Nachbesprechung Exercise 8
- Theory
- **Intro Exercise 9**
- Exam Questions
- Kahoot

Assignment 8: Overview

- Analyzing locks

- Atomic operations

Analyzing locks

- The sample code represents the behavior of a couple that are having dinner together, but they only have a single spoon.

- Prove or disprove that the current implementation provides mutual exclusion.
  - HINT: Use State space diagram

Atomic operations

- In this task, we will see and analyze:
  - the usage of atomic operations to perform concurrency control, and
  - the cost of using them when having data contention

- For more details, please refer to the assignment sheet

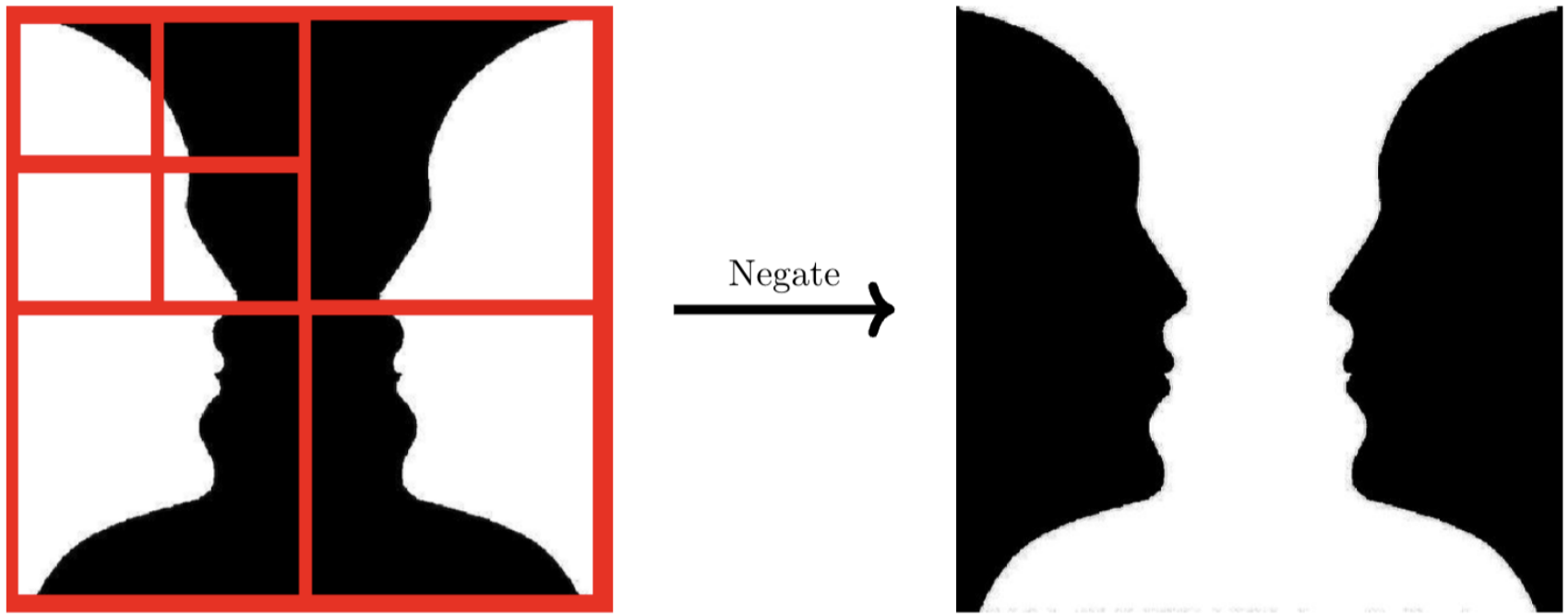
Plan für heute

- Organisation
- Nachbesprechung Exercise 8
- Theory
- Intro Exercise 9
- **Exam Questions**
- Kahoot

Fork/Join Framework (16 points)

3. Der folgende Code zielt darauf ab, ein Bild zu negieren, indem es mithilfe des Fork/Join-Frameworks rekursiv in mehrere Unterfenster (vier pro Rekursionsschritt) unterteilt wird. Die Unterfenster können dann parallel negiert werden. Das folgende Beispiel verdeutlicht die Unterteilung des Bildes und die Negierung der einzelnen Unterfenster.

The following code aims to negate an image by recursively subdividing it into multiple subwindows (four per recursion step) using the Fork/Join framework. The subwindows can then be negated in parallel. The example below illustrates the subdivision of the image and negation of the individual subwindows.



Bitte lesen Sie den Code sorgfältig durch und beantworten Sie dann die Fragen zum Code:

Please read the code carefully and then answer the questions regarding the code:

```java
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```

```
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```
final static int CUTOFF = 32;
```

```
double[][] image, invertedImage;
```

```
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

(a) Welche Annahme trifft der Code bezüglich der Abmessungen des Arrays, das das Eingabebild darstellt?

What assumption does the code make concerning the dimensions of the array representing the input image? (2)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

13

The image should be square $s \times s$ and we should have $s = d2^k$, where $d \leq 32$. This is necessary, because we want `length` to be divisible by 2 in the case `length > 32`. If this would not be the case, we would do floor division and leave pixels unprocessed.

+ Add Comment ··· More

```java
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

(b) Parallelisiert der Code die beabsichtigte Aufgabe korrekt oder gibt es weitere Optimierungsmöglichkeiten? Wenn ja, welche Optimierung würden Sie vorschlagen und warum? (4)

Does the code correctly parallelize the intended task or is there further optimization that could be done? If so, which optimization would you propose and why?

Not good:

```
upperLeft.fork();
upperLeft.join();
upperRight.fork();
upperRight.join();
lowerLeft.fork();
lowerLeft.join();
lowerRight.compute();
```

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 7 months ago

8

No, the parallelization is incorrect, as we have subsequent `fork()` and `join()` calls, which means that we wait for the corresponding subproblem to be finished, before calling `fork()` on the next one. To fix this, we should do the following:

```
upperLeft.fork();
upperRight.fork();
lowerLeft.fork();
lowerRight.compute();
upperLeft.join();
upperRight.join();
lowerLeft.join();
```

```java
public class ImageNegationFJ extends RecursiveAction {
    final static int CUTOFF = 32;
    double[][] image, invertedImage;
    int startx, starty;
    int length;

    public ImageNegationFJ(double[][] image, double[][] invertedImage,
            int startx, int starty, int length) {
        this.image = image;
        this.invertedImage = invertedImage;
        this.startx = startx;
        this.starty = starty;
        this.length = length;
    }

    @Override
    protected void compute() {
```

(c) Vervollständigen Sie das folgende Codegerüst, indem Sie die oben implementierte `ImageNegationFJ` Klasse und die `ForkJoinPool` Klasse verwenden, um die Variable `negatedImage` mit den negierten Werten zu füllen.

Complete the following code skeleton by using the above implemented `ImageNegationFJ` class and the `ForkJoinPool` class to fill the variable `negatedImage` with the negated values.

(4)

```java
double[][] image = {{0, 1}, {1, 0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
.....
.....
.....
.....
.....
.....
```

Tobias Steinbrecher @tsteinbreche · 8 months ago

10

```
double[][] image = {{0,1}, {1,0}};
int imageSize = image.length;
double[][] negatedImage = new double[imageSize][imageSize];
ForkJoinPool fjp = new ForkJoinPool();
ForkJoinTask t = new ImageNegationFJ(image, negatedImage, 0, 0 ,imageSize);
fjp.invoke(t);
```

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoinPool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

Assuming that the `ImageNegationFJ` class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the 2×2 `negatedImage` array from task 3c)? (2)

```
double[][] image = {{0,1}, {1,0}};
```

```
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```
final static int CUTOFF = 32;
```

(d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der ForkJoinPool effektiv, um das 2×2 `negatedImage` Array aus Aufgabe 3c) zu füllen?

Assuming that the `ImageNegationFJ` class is correctly parallelized, how many threads does the ForkJoinPool effectively use to fill the 2×2 `negatedImage` array from task 3c)? (2)

```
double[][] image = {{0,1}, {1,0}};
```

```
@Override
protected void compute() {
    if (this.length <= CUTOFF) {
        for (int offsetX = 0; offsetX < this.length; offsetX++) {
            for (int offsetY = 0; offsetY < this.length; offsetY++) {
                this.invertedImage[this.startx + offsetX][this.starty + offsetY] = 1
                        - this.image[this.startx + offsetX][this.starty + offsetY];
            }
        }
    } else {
        int halfSize = (this.length) / 2;
        ImageNegationFJ upperLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty, halfSize);
        ImageNegationFJ upperRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize, this.starty,
                halfSize);
        ImageNegationFJ lowerLeft = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx, this.starty + halfSize,
                halfSize);
        ImageNegationFJ lowerRight = new ImageNegationFJ(this.image,
                this.invertedImage, this.startx + halfSize,
                this.starty + halfSize, halfSize);
        upperLeft.fork();
        upperLeft.join();
        upperRight.fork();
        upperRight.join();
        lowerLeft.fork();
        lowerLeft.join();
        lowerRight.compute();
    }
}
```

```
final static int CUTOFF = 32;
```

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

Because of the sequential cutoff, only **one** Thread would be used *effectively*.

(e) Gehen Sie von einem konstanten Overhead von $16\,\text{MB} = 2^4\,\text{MB}$ pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe $10\,\text{GB}$ der Speicher ausgeht? Hinweis: $1\,\text{GB} = 2^{10}\,\text{MB}$.

Assume a fixed overhead of $16\,\text{MB} = 2^4\,\text{MB}$ *per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for* `CUTOFF` *if you input an image of size* 4000×4000 *before you run out of memory using a RAM of size* $10\,\text{GB}$*? Hint:* $1\,\text{GB} = 2^{10}\,\text{MB}$.

(4)

(e) Gehen Sie von einem konstanten Overhead von $16\,\mathrm{MB} = 2^4\,\mathrm{MB}$ pro Thread aus und dass pro Split immer vier neue Threads erstellt werden. Dies bedeutet, dass die Anzahl der Threads nicht durch den ForkJoinPool festgelegt wird, sodass kein Thread wiederverwendet wird und es zu keinem Work Stealing zwischen den Threads kommt. Was ist der niedrigste Wert für `CUTOFF`, wenn Sie ein Bild der Größe 4000×4000 eingeben, bevor Ihnen bei einem RAM der Größe $10\,\mathrm{GB}$ der Speicher ausgeht? Hinweis: $1\,\mathrm{GB} = 2^{10}\,\mathrm{MB}$.

Assume a fixed overhead of $16\,\mathrm{MB} = 2^4\,\mathrm{MB}$ *per thread and that there are always four new threads created per split. This means that the number of threads is not fixed by the ForkJoinPool, so no thread is re-used and there is no work stealing among the threads. What is the lowest value for* `CUTOFF` *if you input an image of size* 4000×4000 *before you run out of memory using a RAM of size* $10\,\mathrm{GB}$*? Hint:* $1\,\mathrm{GB} = 2^{10}\,\mathrm{MB}$.

(4)

Tobias Steinbrecher @tsteinbreche · 8 months ago · edited 8 months ago

14

Number of threads, which we can use:

$$N = \frac{10 \cdot 2^{10}}{2^4} = 10 \cdot 2^6 = 10 \cdot 4^3$$

In each recursive call, we will use 4 new threads (under given assumptions). Thereby, we have the constraint ($i :=$ number of divisions)

$$4^i \leq 10 \cdot 4^3 \iff i \leq \log_4(10) + 3 \iff i \leq 4$$

and the smallest possible value is $\texttt{CUTOFF} = 4000/2^4 = \mathbf{250}$ to avoid a fifth division.

+ Add Comment ··· More

Plan für heute

- Organisation
- Nachbesprechung Exercise 5
- Theory Recap
- Intro Exercise 6
- Exam Questions
- **Kahoot**

Kahoot

Feedback

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

Danke

- Bis nächste Woche!