

Parallel Programming

Exercise Session 8

Week 8

Schedule

Post-Discussion Ex. 7

Theory (Recap and Outlook)

Pre-Discussion Ex. 8

Kahoot

Feedback: Exercise 7

Feedback for Assignment 7

- What is wrong with the following code snippet?

```
public synchronized boolean transferMoney(Account from, Account to, int amount) {  
    ...  
    ...  
    return true;  
}
```

Feedback for Assignment 7

- What we should have done for avoiding deadlocks

```
public class Account ... {  
    ...  
    private final Lock lock = new ReentrantLock();  
    ...  
}
```

Feedback for Assignment 7

- What we should have done for avoiding deadlocks

```
public class BankingSystem {  
    ...  
    public boolean transferMoney(Account from, Account to, aint amount) {  
        Account first, second;  
        // Introduce lock ordering:  
        if (to.getId() > from.getId()) {  
            first = from; second = to;  
        } else {  
            first = to; second = from;  
        }  
        ...  
    }  
}
```

Feedback for Assignment 7

- Acquire locks, use finally to always release the locks

```
public class BankingSystem {  
    ...  
    public boolean transferMoney(Account from, Account to, int amount) {  
        ...  
        first.getLock().lock();  
        second.getLock().lock();  
        try {  
            ...  
        } finally {  
            first.getLock().unlock();  
            second.getLock().unlock();  
        }  
    }  
}
```

Feedback for Assignment 7

- Summing up: How to do it safe

Lock each account before reading out its balance, but don't release the lock until all accounts are summed up.

→ Two-phase locking

In the first phase locks will be acquired without releasing,
in the second phase locks will be released.

→ Deadlocks still a problem

→ Ordered locking required

Let us take a look at the master solution

Theory (Recap and Outlook)

What will you see soon in the lectures?

- Memory reordering and optimizations
- Orders:
Program Order, Synchronizes-with, Synchronization Order, Happens-before
- State Space Diagrams
- Dekker's Algorithm, Peterson Lock, Filter Lock (generalization of Peterson Lock), Bakery Lock
- Test-and-set (TAS), compare-and-swap (CAS), Test and Test-and-set (TTAS), Exponential Backoff
- Semaphores and Barriers

We took a look at the following in the last ex. session

- **Memory reordering and optimizations**
- **Orders:**
Program Order, Synchronizes-with, Synchronization Order, Happens-before
- **State Space Diagrams**
- **Dekker's Algorithm, Peterson Lock, Filter Lock (generalization of Peterson Lock), Bakery Lock**
- **Test-and-set (TAS), compare-and-swap (CAS), Test and Test-and-set (TTAS), Exponential Backoff**
- **Semaphores and Barriers**

We will additionally take a look at

- Memory reordering and optimizations
- Orders:
Program Order, Synchronizes-with, Synchronization Order, Happens-before
- State Space Diagrams
- Dekker's Algorithm (why is it the way it is), Peterson Lock, Filter Lock (generalization of Peterson Lock), Bakery Lock
- Test-and-set (TAS), compare-and-swap (CAS), Test and Test-and-set (TTAS), Exponential Backoff
- Semaphores and Barriers

Let's vote: Mentimeter

Motivation

```
class C {  
    private int x = 0;  
    private int y = 0;
```

Thread 1

```
    x = 1;
```

```
    y = 1;
```

```
}
```

Thread 2

```
    int a = y;
```

```
    int b = x;
```

```
    assert(b >= a);
```

```
}
```

```
}
```

Can this fail?

Another proof

```
class C {  
    private int x = 0;  
    private int y = 0;
```

Thread 1

```
    x = 1;
```

```
    y = 1;
```

```
}
```

Thread 2

```
    int a = y;
```

```
    int b = x;
```

```
    assert(b >= a);
```

```
}
```

```
}
```

There is no interleaving of \mathfrak{f} and \mathfrak{g} causing the assertion to fail

Another proof

```
class C {  
    private int x = 0;  
    private int y = 0;
```

Thread 1

```
    x = 1;
```

```
    y = 1;
```

```
}
```

Thread 2

```
    int a = y;
```

```
    int b = x;
```

```
    assert(b >= a);
```

```
}
```

```
}
```

There is no interleaving of f and g causing the assertion to fail

Another proof (by contradiction):

Assume $b < a \Rightarrow a == 1$ and $b == 0$.

But if $a == 1 \Rightarrow y = 1$ *happened before* $a = y$.

And if $b == 0 \Rightarrow b = x$ *happened before* $x = 1$.

Because we assume that programs execute in order:

$a = y$ *happened before* $b = x$

$x = 1$ *happened before* $y = 1$

So by transitivity,

$a = y$ happened before $b = x$ happened before $x = 1$ happened before $y = 1$ happened before $a = y \Rightarrow$ **Contradiction** ⚡

But does this really work?

No

Because of:

Optimizations by Compiler

Optimizations by Hardware

(basically Memory Reordering)

Why it still can fail: Memory reordering

Rule of thumb: Compiler and hardware allowed to make changes that do not affect the *semantics* of a *sequentially* executed program

```
void f() {  
    x = 1;  
    y = x+1;  
    z = x+1;  
}
```

semantically
equivalent?

```
void f() {  
    x = 1;  
    z = x+1;  
    y = x+1;  
}
```

semantically
equivalent?

```
void f() {  
    x = 1;  
    z = 2;  
    y = 2;  
}
```

Are these semantically equivalent?

Example: **Fail** with self-made rendezvous (C / GCC)

```
int x;
```

```
void wait() {  
    x = 1;  
    while(x==1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

Assembly without optimization

```
movl    $0x1, x  
test:  
mov     x, %eax  
cmp     $0x1, %eax  
je      test
```

```
movl    $0x2, x
```

Example: **Fail** with self-made rendezvous (C / GCC)

```
int x;
```

```
void wait() {  
    x = 1;  
    while(x==1);  
}
```

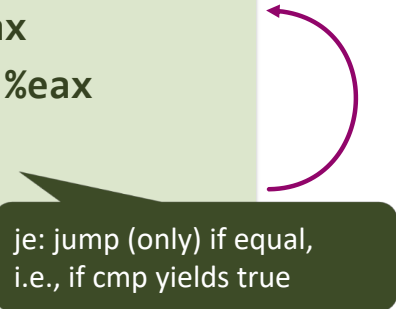
```
void arrive(){  
    x = 2;  
}
```

Assembly without optimization

```
movl    $0x1, x  
test:  
mov     x, %eax  
cmp     $0x1, %eax  
je      test
```

```
movl    $0x2, x
```

je: jump (only) if equal,
i.e., if cmp yields true



Example: **Fail** with self-made rendezvous (C / GCC)

```
int x;
```

```
void wait() {  
    x = 1;  
    while(x==1);  
}
```

```
void arrive(){  
    x = 2;  
}
```

Assembly without optimization

```
movl    $0x1, x  
test:  
mov     x, %eax  
cmp     $0x1, %eax  
je      test
```

je: jump (only) if equal,
i.e., if cmp yields true

```
movl    $0x2, x
```

Assembly with optimization

```
movl    $0x1, x  
test:  
jmp     test
```

jmp: jump always

```
movl    $0x2, x
```

Memory hierarchy (one core)

ALUs

Registers

0.5ns

fast, low latency, high cost, low capacity

L1 Cache

1 ns

L2 Cache

7 ns

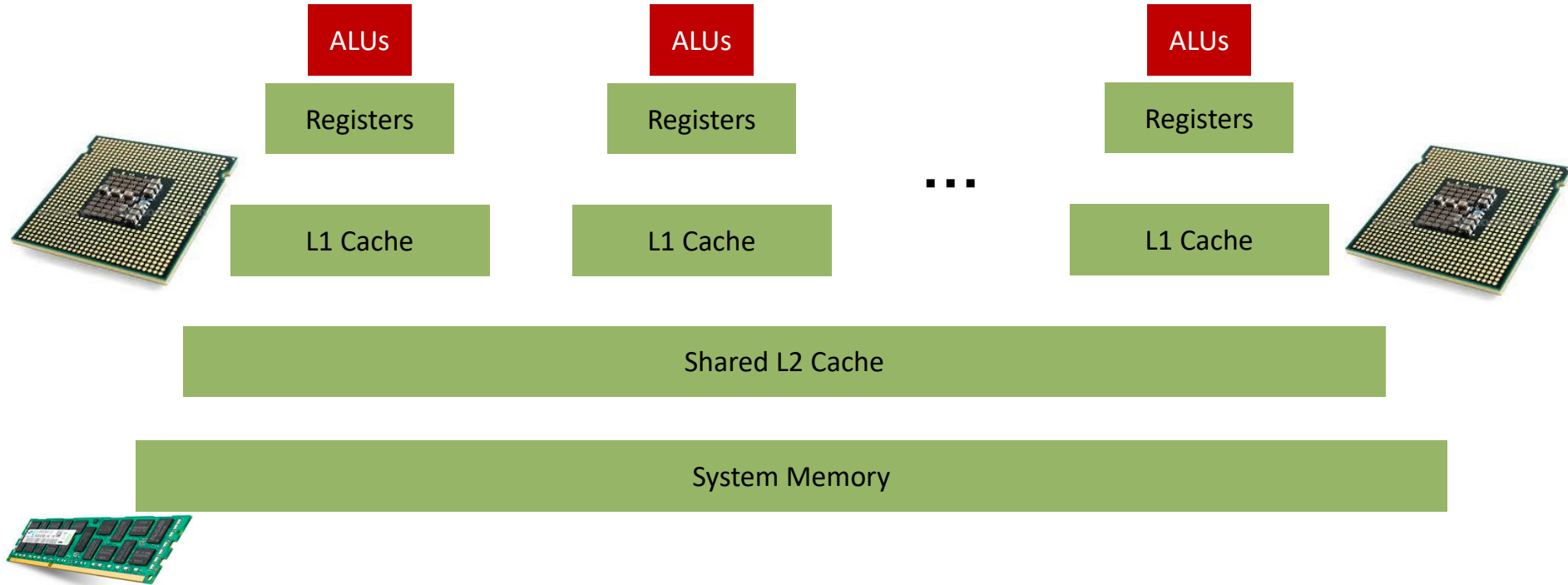
System Memory

100 ns

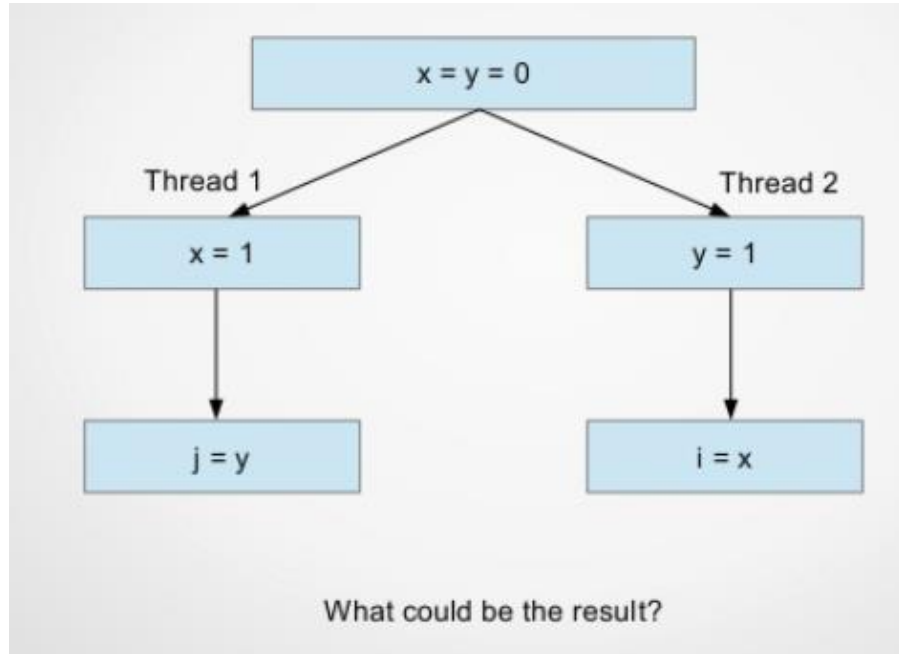
slow, high latency, low cost, high capacity



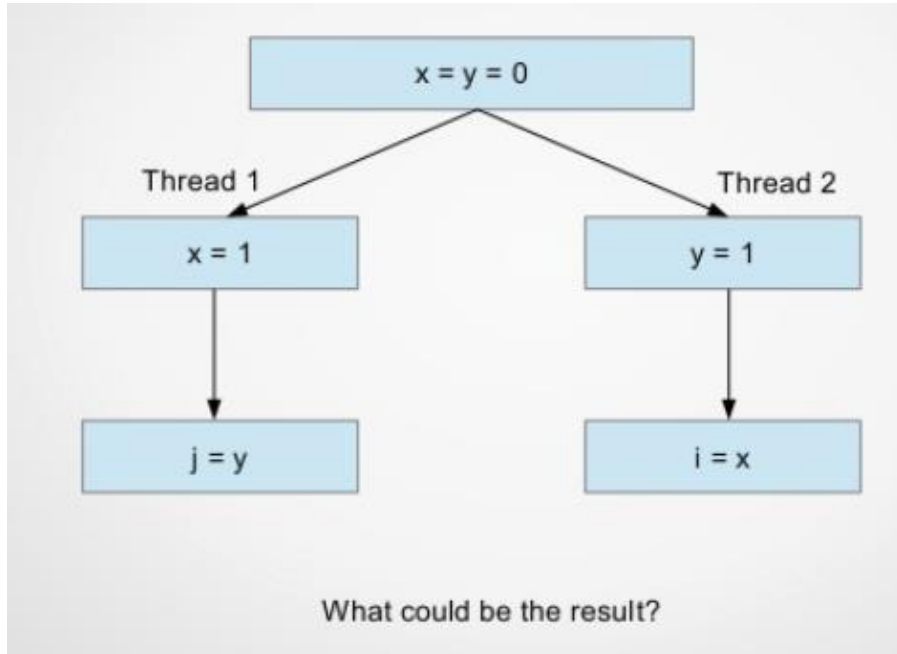
Memory hierarchy (many cores)



Why memory models, x86 example



Why memory models, x86 example



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.

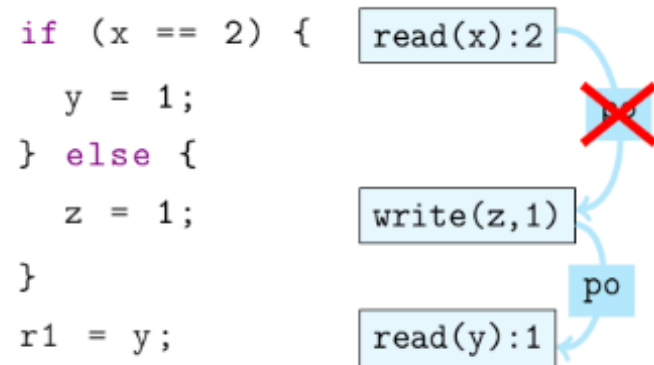
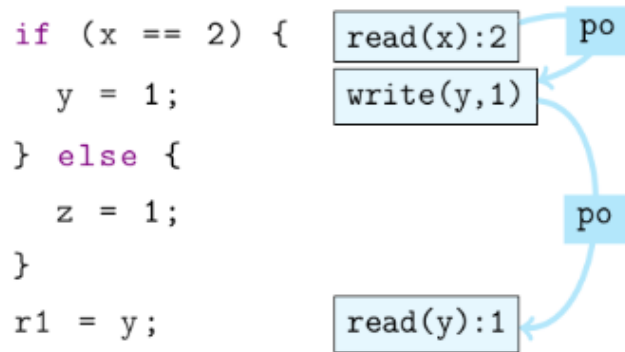
How? Java Memory Model!

Java Memory Model (JMM): Necessary basics

- **JMM restricts allowable outcomes of programs**
 - You saw that if we don't have these operations (volatile, synchronized etc.) – outcome can be “arbitrary” (not quite correct, say unexpected 😊)
- **JMM defines *Actions*: `read(x) : 1` “read variable x, the value read is 1”**
- ***Executions combine actions with ordering:***
 - *Program Order*
 - *Synchronizes-with*
 - *Synchronization Order*
 - *Happens-before*

JMM: Program Order (PO)

- **Program order is a total order of intra-thread actions**
 - Program statements are NOT a total order **across** threads!
- **Program order does not provide an ordering guarantee for memory accesses!**
 - The only reason it exists is to provide the link between possible executions and the original program.
- **Intra-thread consistency: Per thread, the PO order is consistent with the thread's isolated execution**

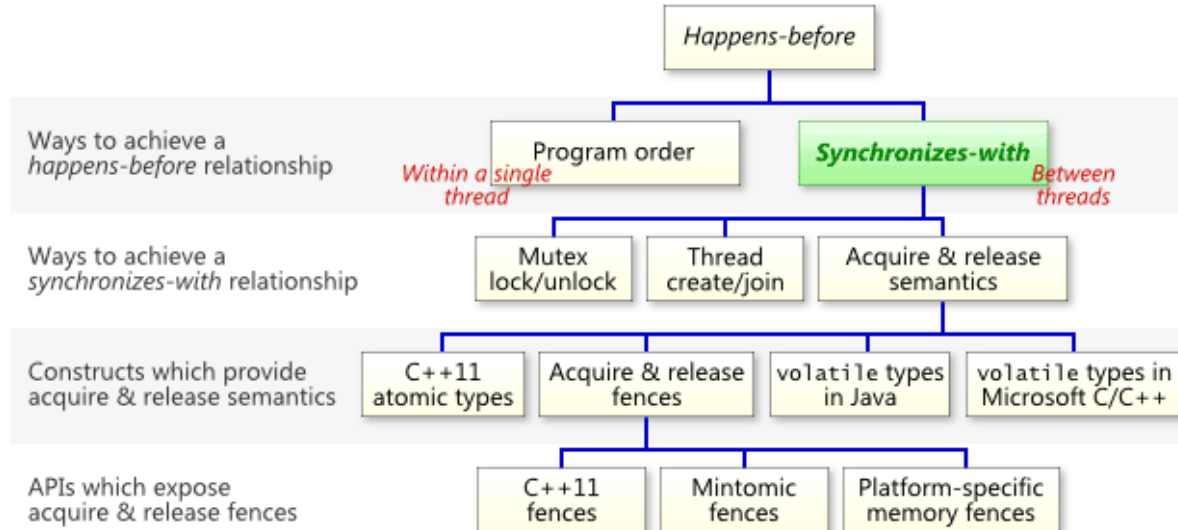


JMM: Synchronization Actions (SA) and Synchronization Order (SO)

- **Synchronization actions are:**
 - Read/write of a volatile variable
 - Lock monitor, unlock monitor
 - First/last action of a thread (synthetic)
 - Actions which start a thread
 - Actions which determine if a thread has terminated
- **Synchronization Actions form the Synchronization Order (SO)**
 - SO is a total order
 - All threads see SA in the same order
 - SA within a thread are in PO
 - SO is consistent: all reads in SO see the last writes in SO

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!



Problem: How do we implement locks?

- For two threads: Dekker's Algorithm, Peterson Lock
- For n threads: Filter Lock, Bakery Lock

Why do we need locks again?

Critical Sections

Critical sections

Pieces of code with the following conditions

1. **Mutual exclusion:** statements from critical sections of two or more processes must not be interleaved
2. **Freedom from deadlock:** if some processes are trying to enter a critical section then one of them must eventually succeed
3. **Freedom from starvation:** if *any* process tries to enter its critical section, then that process must eventually succeed

Critical section problem

global (shared) variables

Process P

local variables

loop

non-critical section

preprotocol

critical section

postprotocol

Process Q

local variables

loop

non-critical section

preprotocol

critical section

postprotocol

Mutual exclusion for 2 processes -- 1st Try

```
volatile boolean wantp=false, wantq=false
```

Process P

local variables

loop

p1 non-critical section

p2 while(wantq);

p3 wantp = true

p4 critical section

p5 wantp = false

Process Q

local variables

loop

q1 non-critical section

q2 while(wantp);

q3 wantq = true

q4 critical section

q5 wantq = false

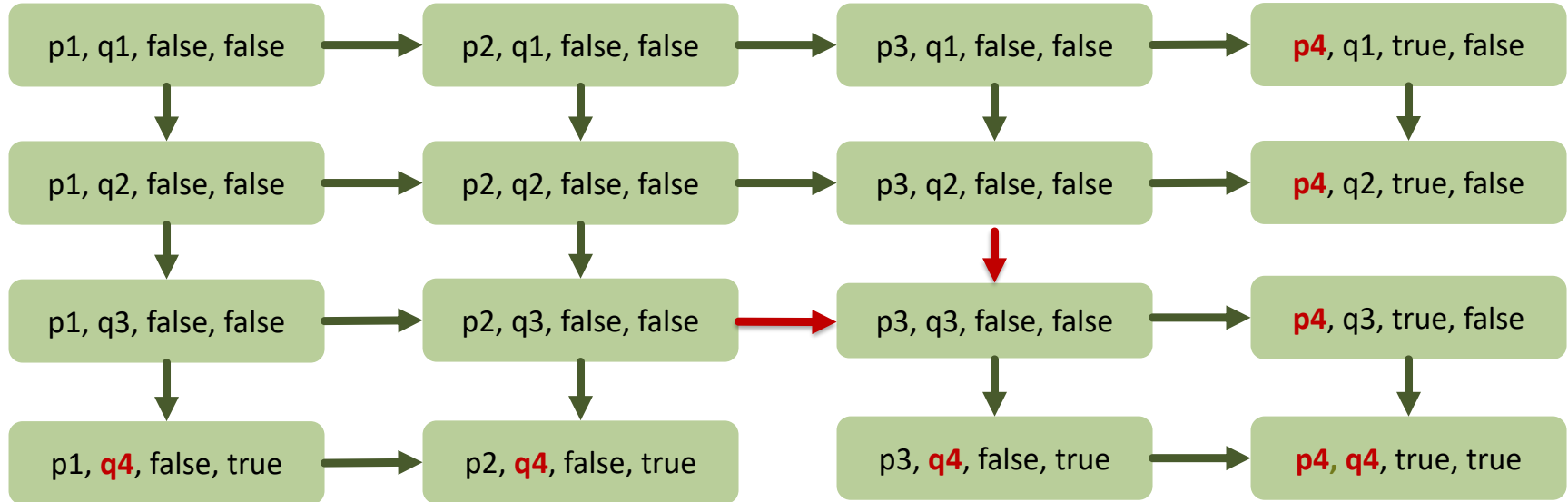
State Space Diagram

- When dealing with mutual exclusion problems, we should focus on:
 - the structure of the underlying state space, and
 - the state transitions that occur
- State diagram captures the entire state space and all possible computations (execution paths a program may take)
- A good solution will have a state space with no bad states

State space diagram [p, q, wantp, wantq]

1 non-critical section 2 while(wantp) while(wantq) 3 wantp = true wantq = true 4 critical section 5 wantp = false wantq = false

p1	non-critical section
p2	while(wantq);
p3	wantp = true
p4	critical section
p5	wantp = false



Mutual exclusion for 2 processes -- 2nd Try

volatile boolean wantp=false, wantq=false

Process P

local variables

loop

p1 non-critical section

p2 wantp = true

p3 while(wantq);

p4 critical section

p5 wantp = false

Process Q

local variables

loop

q1 non-critical section

q2 wantq = true

q3 while(wantp):

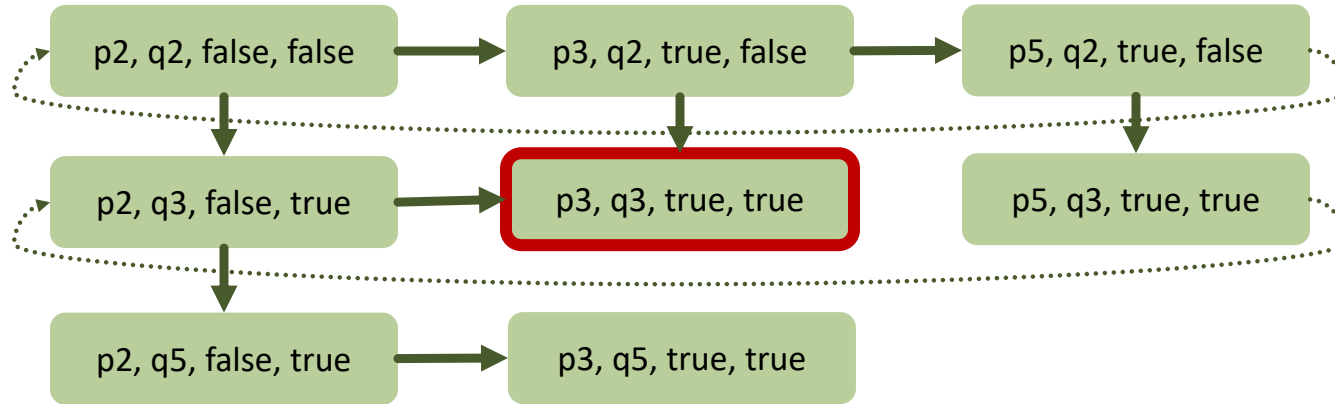
q4 critical section

q5 wantq = false

Do you see the problem?

State space diagram [p, q, wantp, wantq]

- 1 non-critical section 2 wantp = true 3 while(wantp) 4 critical section 5 wantp = false
wantq = true while(wantq) wantq = false



- 1 non-critical section
2 wantq = true
3 while(wantp):
4 critical section
5 wantq = false

deadlock !

Mutual exclusion for 2 processes -- 3rd Try

```
volatile int turn = 1;
```

Process P

local variables

loop

p1 non-critical section

p2 while(turn != 1);

p3 critical section

p4 turn = 2

Process Q

local variables

loop

q1 non-critical section

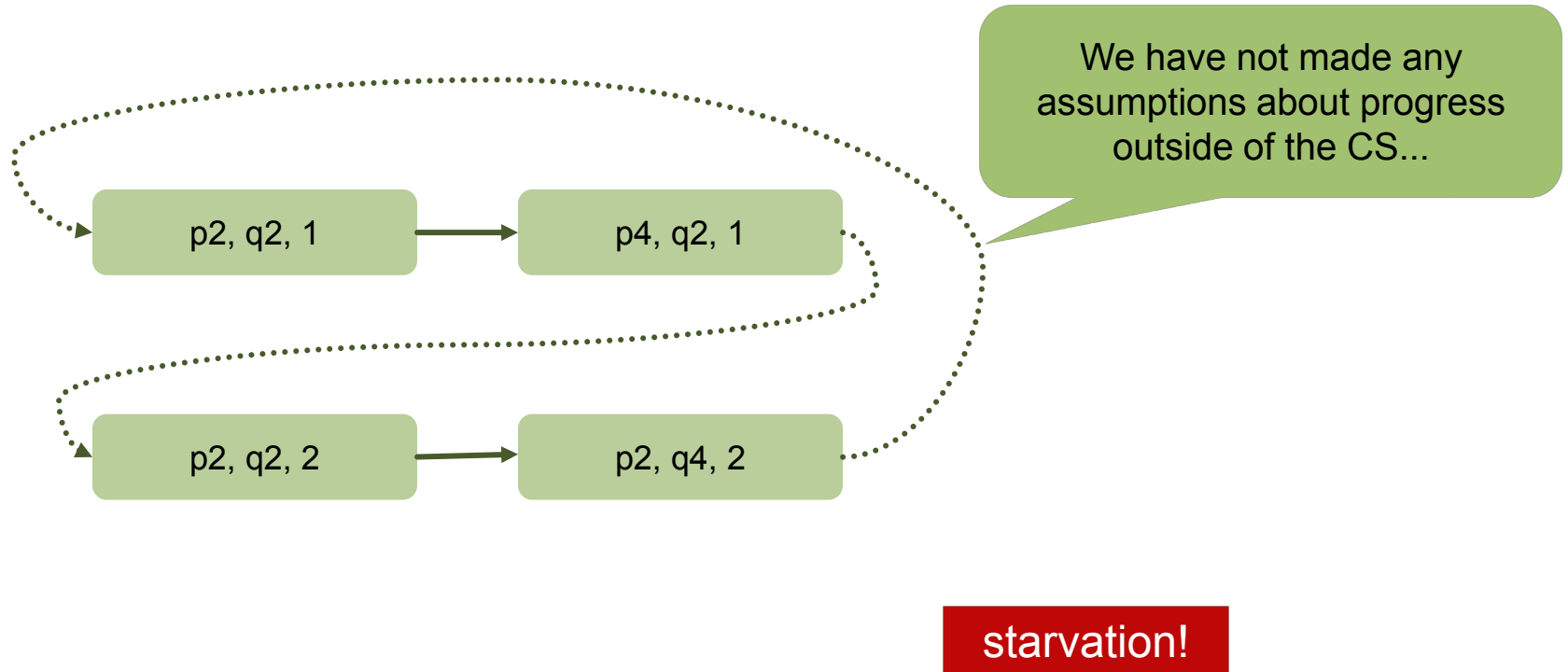
q2 while(turn != 2);

q3 critical section

q4 turn = 1

Do you see the problem?

State space diagram [p, q, turn]



Practice seeing deadlocks, etc.

- <https://deadlockempire.github.io/>

Correctness of Mutual exclusion

- *“Statements from the critical sections of two or more processes must **not** be interleaved.”*
- We can see that there is no state in which the program counters of both P and Q point to statements in their critical sections

Freedom from deadlock

- *“If some processes are trying to enter their critical sections then one of them must eventually succeed.”*
- We don't have a situation when the processes aren't making any progress anymore

Freedom from deadlock

- Since the behaviour of processes P and Q is symmetrical, we only have to check what happens for one of the processes, say P.
- Freedom from deadlock means that from any state where a process wishes to enter its CS (by awaiting its turn), there is *always a path* (sequence of transitions) leading to it entering its CS.

Freedom from deadlock

- Typically, a deadlocked state has no transitions leading from it, i.e. no statement is able to be executed.
- Sometimes a cycle of transitions may exist from a state for each process, from which no useful progress in the parallel program can be made. We call this a **Livelock**. Everyone is 'busy doing nothing'.

Freedom from individual starvation

- *“If any process tries to enter its critical section then that process must eventually succeed.”*
- If a process is wishing to enter its CS (awaiting its turn) and another process refuses to set the turn, the first process is said to be starved.
- Possible starvation reveals itself as cycles in the state diagram.
- Because the definition of the critical section problem allows for a process to not make progress from its Non-critical section, starvation is, in general, possible in this example

Dekker's Algorithm (combination of try 2 and 3)

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

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);

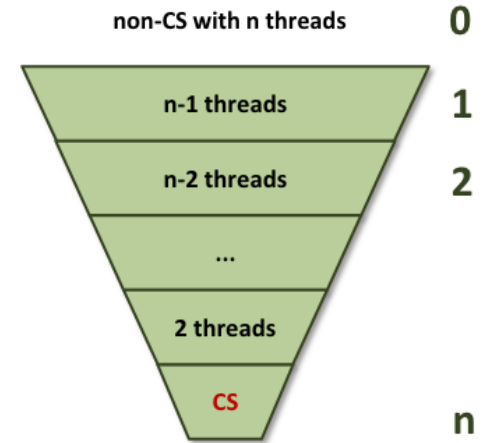
critical section

flag[Q] = false

Filter Lock

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

```
lock(me) {  
    for (int i=1; i<n; ++i) {  
        level[me] = i;  
        victim[i] = me;  
        while ( $\exists k \neq \text{me}: \text{level}[k] \geq i \ \&\& \ \text{victim}[i] == \text{me}$ ) {};  
    }  
}  
  
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]
```

```
lock(me):
```

```
    flag[me] = true;
```

```
    label[me] = max(label[0], ... , label[n-1]) + 1;
```

```
    while ( $\exists k \neq me$ : flag[k] &&  $(k, label[k]) <_l (me, label[me])$ ) {};
```

```
unlock(me):
```

```
    flag[me] = false;
```

Atomic Registers

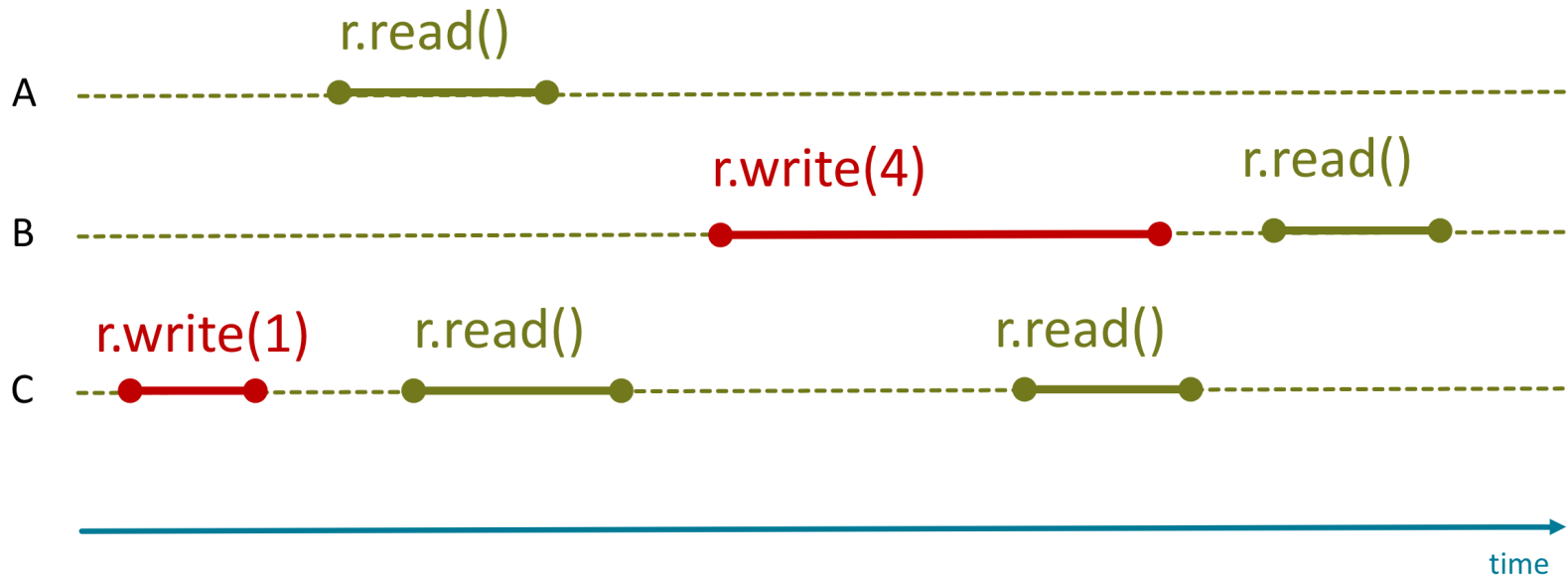
Register: basic memory object, can be shared or not
i.e., in this context register \neq register of a CPU

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

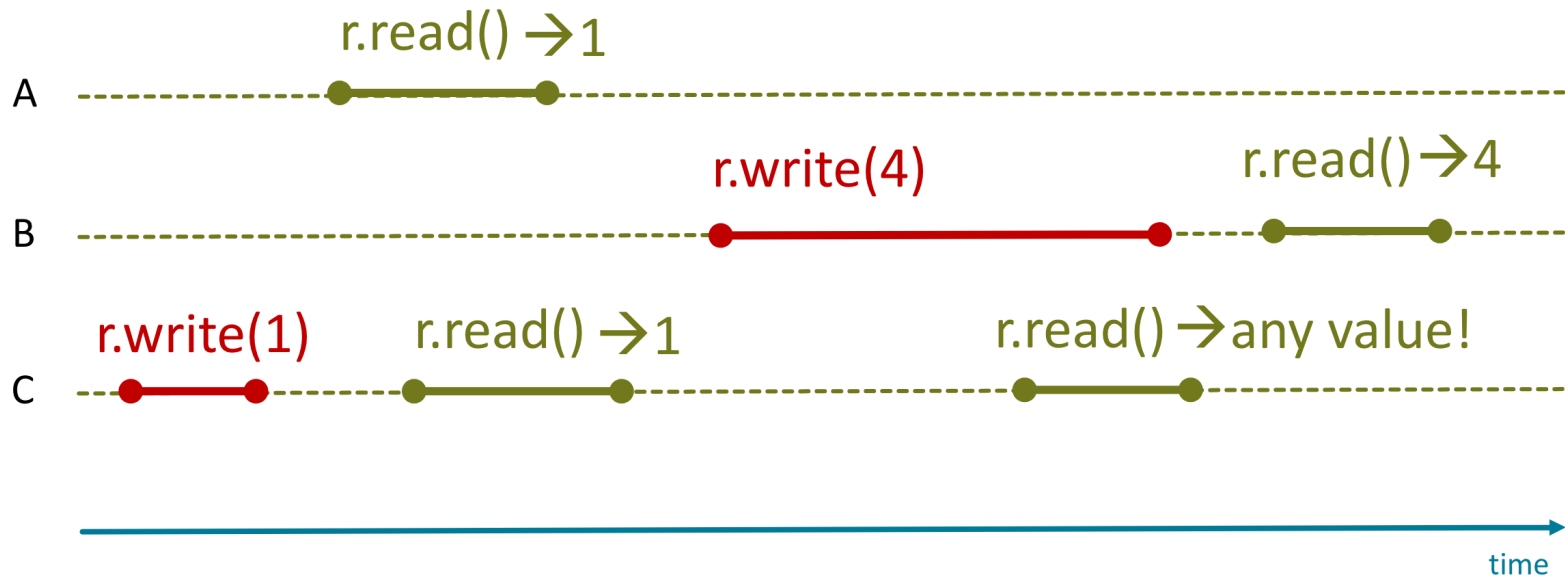
Atomic Register:

- 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



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

Hardware support for atomic operations

- Test-And-Set (TAS)
- Compare-And-Swap (CAS)

TAS and CAS

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;
```

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

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

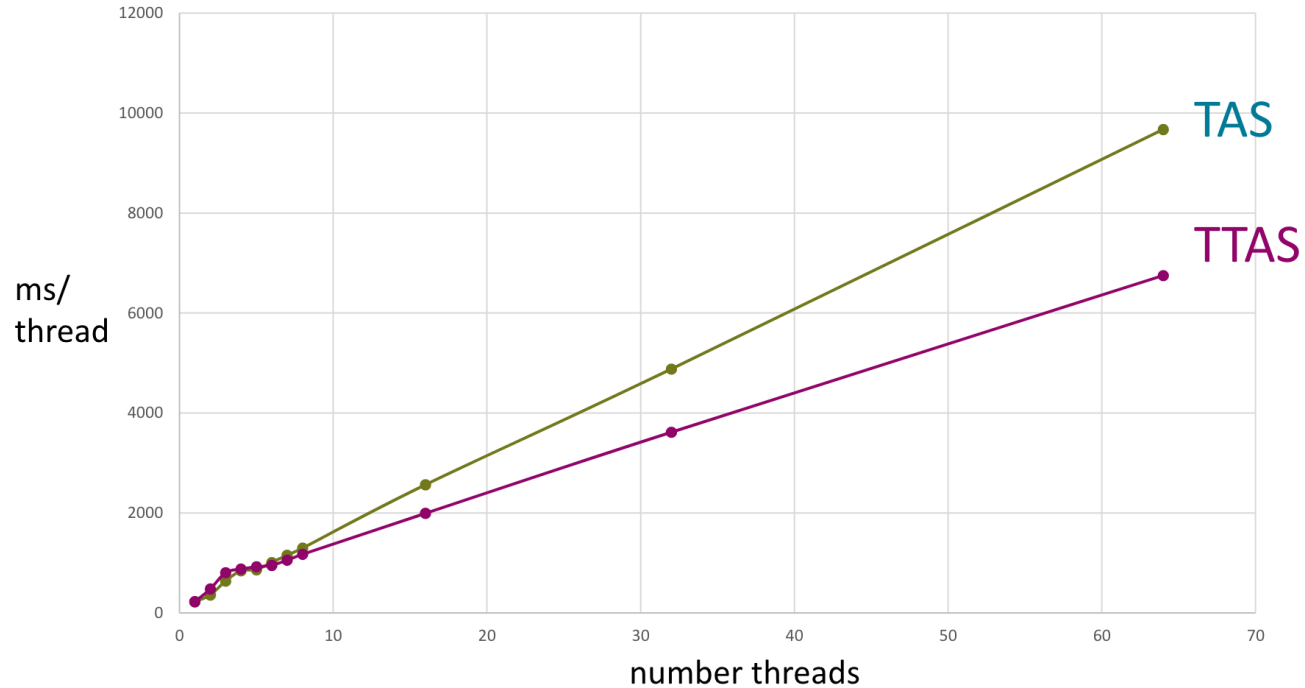
Lets try spinning on local cache

```
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);
    }
}
```

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

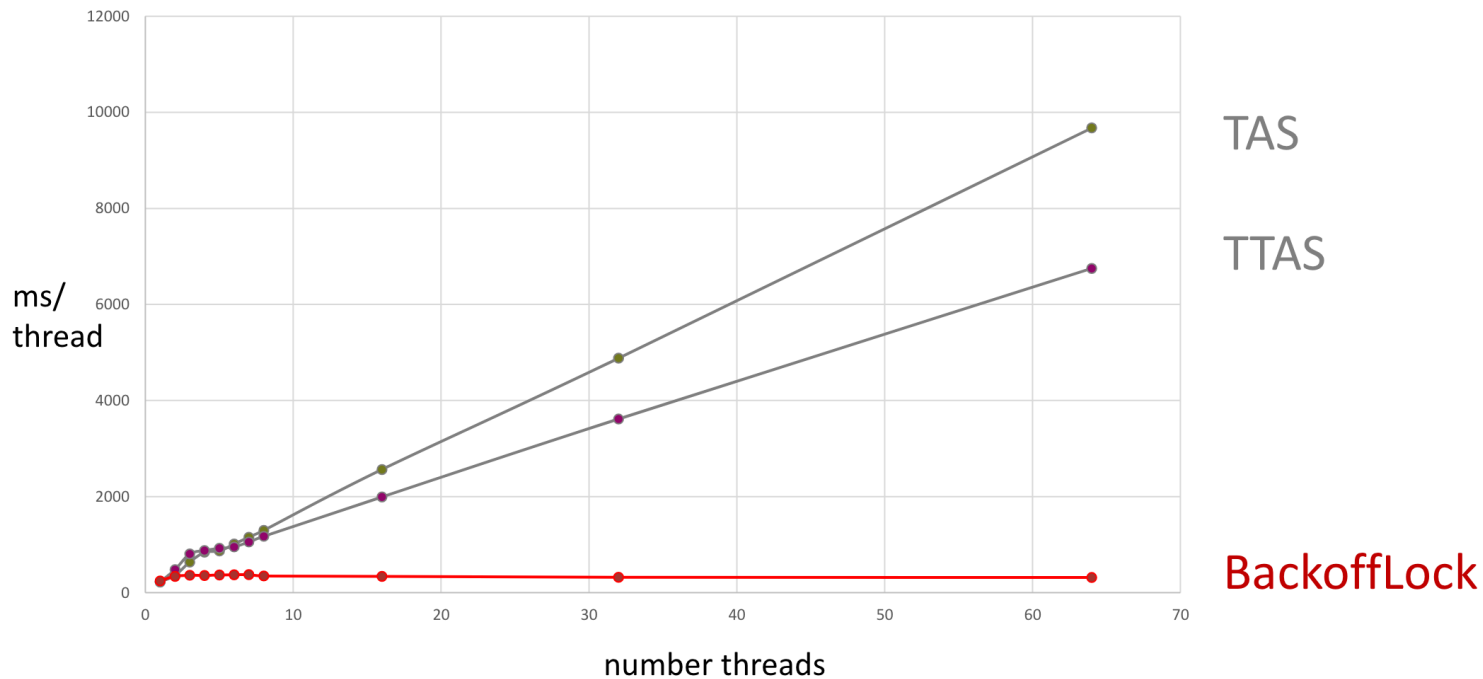
Idea: Each time TAS fails, wait longer until you re-try

- Backoff must be random!

Exponential backoff

```
acquire(lock):  
    while True:  
        while lock == 1: pass  
        if CAS(lock, 0, 1) == 0:  
            return True  
        else:  
            backoff *= 2  
            sleep(backoff)  
  
unlock(lock):  
    lock = 0
```

Nice!



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

acquire(S)

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

release(S)

```
atomic {  
    inc(S)  
}
```

acquire

(protected)

release

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

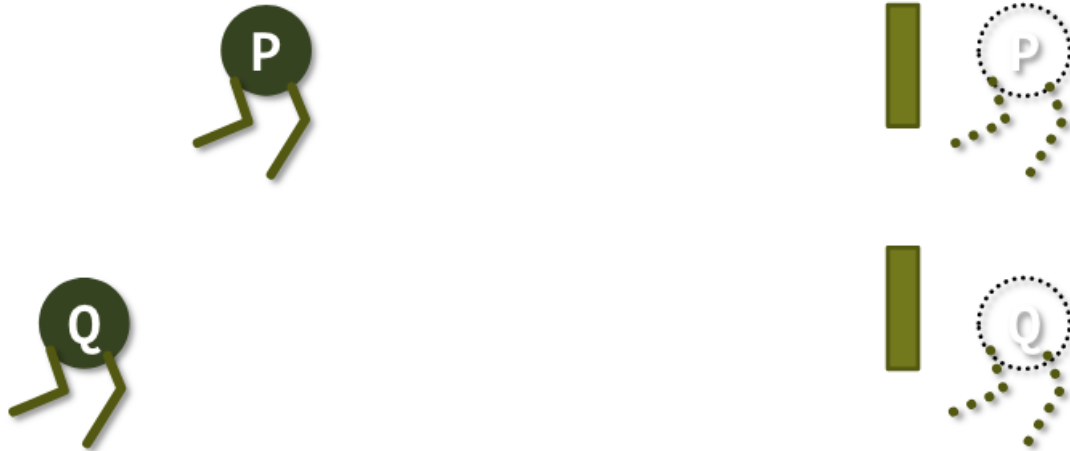
$x > 0$ → x threads will be let into “critical section”

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 acquiring before!**
 - There 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?

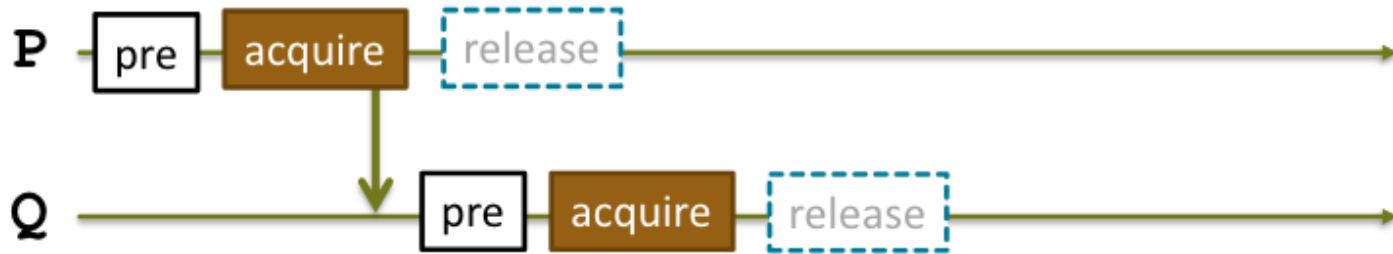
Synchronize Processes P and Q at one location (Rendezvous)

Semaphores **P_Arrived** and **Q_Arrived**

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	acquire(Q_Arrived) release(P_Arrived)	acquire(P_Arrived) release(Q_Arrived)
<i>post</i>

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)

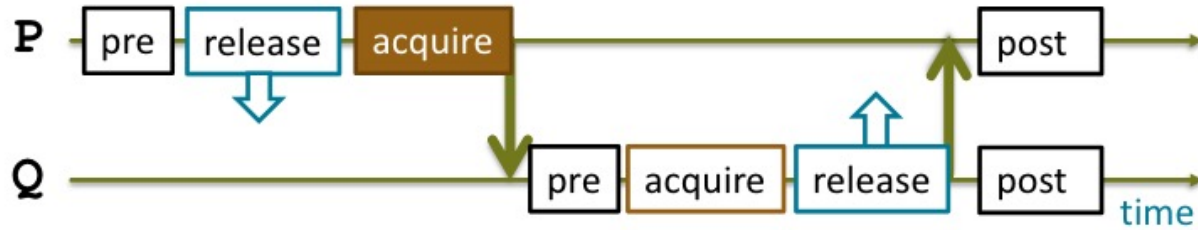
Assume Semaphores **P_Arrived** and **Q_Arrived**

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	acquire(P_Arrived) release(Q_Arrived)
<i>post</i>

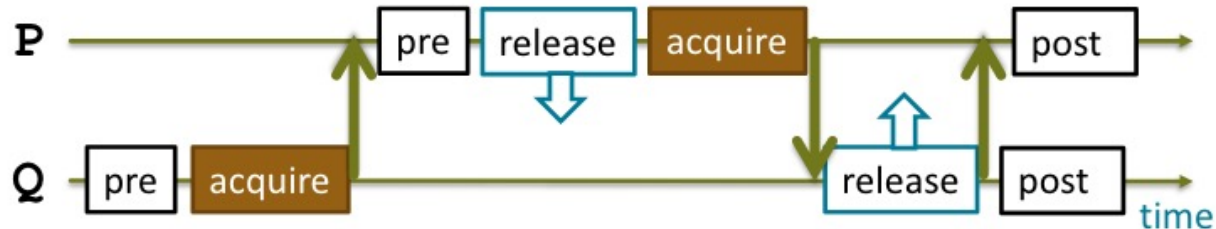
Yes, that works!

P first

	P	Q
<i>init</i>	<code>P_Arrived=0</code>	<code>Q_Arrived=0</code>
<i>pre</i>
<i>rendezvous</i>	<code>release(P_Arrived)</code> <code>acquire(Q_Arrived)</code>	<code>acquire(P_Arrived)</code> <code>release(Q_Arrived)</code>
<i>post</i>



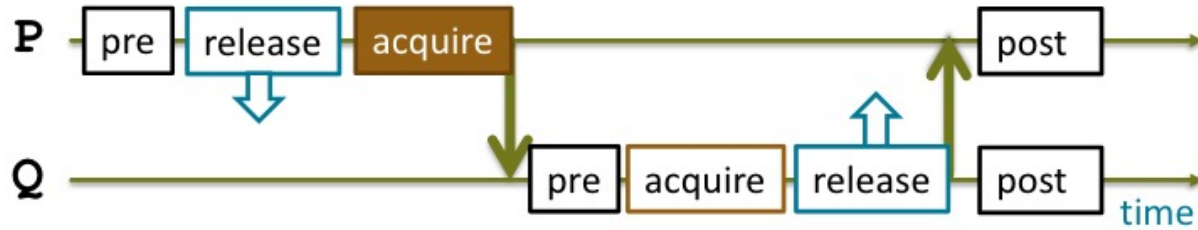
Q first



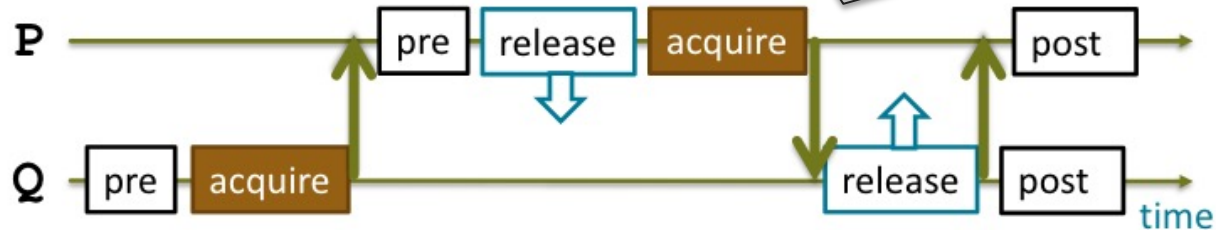
Yes, that works!

P first

	P	Q
<i>init</i>	<code>P_Arrived=0</code>	<code>Q_Arrived=0</code>
<i>pre</i>
<i>rendezvous</i>	<code>release(P_Arrived)</code> <code>acquire(Q_Arrived)</code>	<code>acquire(P_Arrived)</code> <code>release(Q_Arrived)</code>
<i>post</i>



Q first



Lets do better!

Synchronize Processes P and Q at one location (Rendezvous)

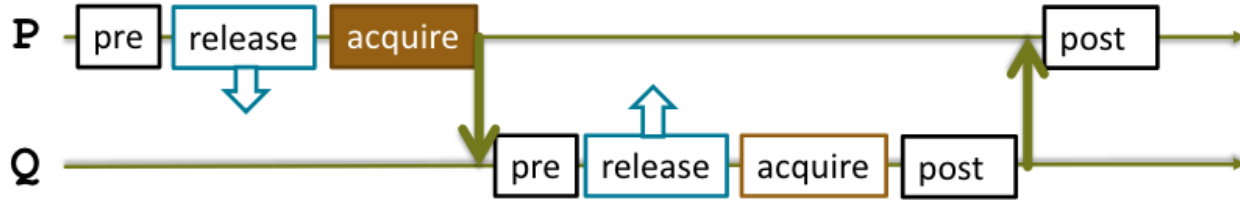
Assume Semaphores **P_Arrived** and **Q_Arrived**

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
<i>post</i>

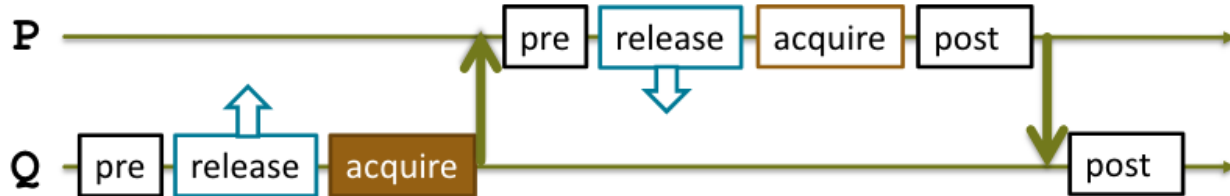
Order does no longer matter

	P	Q
<i>init</i>	P_Arrived=0	Q_Arrived=0
<i>pre</i>
<i>rendezvous</i>	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
<i>post</i>

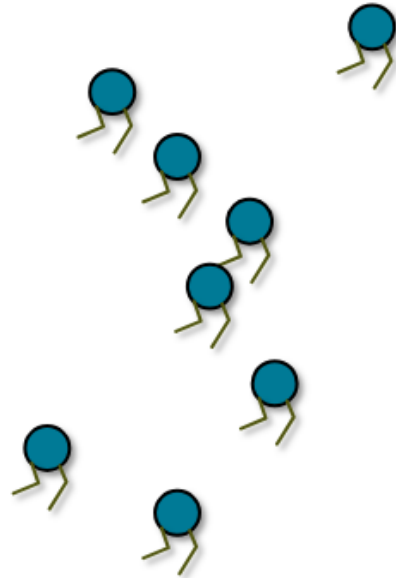
P first



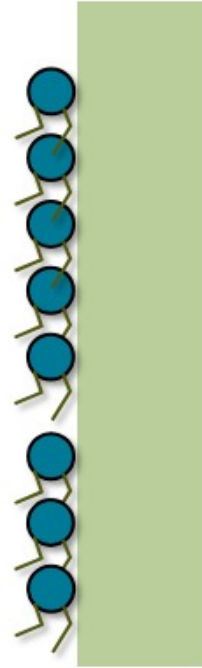
Q first



How about more than two threads? Barriers!



How about more than two threads? Barriers!



First attempt

Synchronize a number (n) of processes.

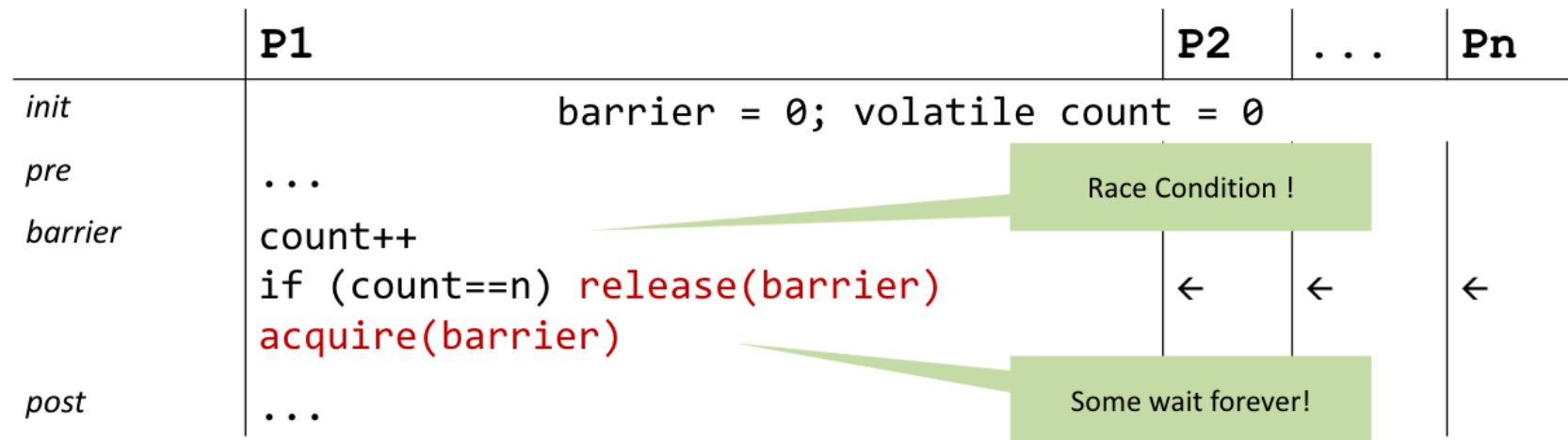
Semaphore **barrier**. Integer count.

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

Wrong

Synchronize a number (n) of processes.

Semaphore **barrier**. Integer count.




How about this?

Synchronize a number (n) of processes.

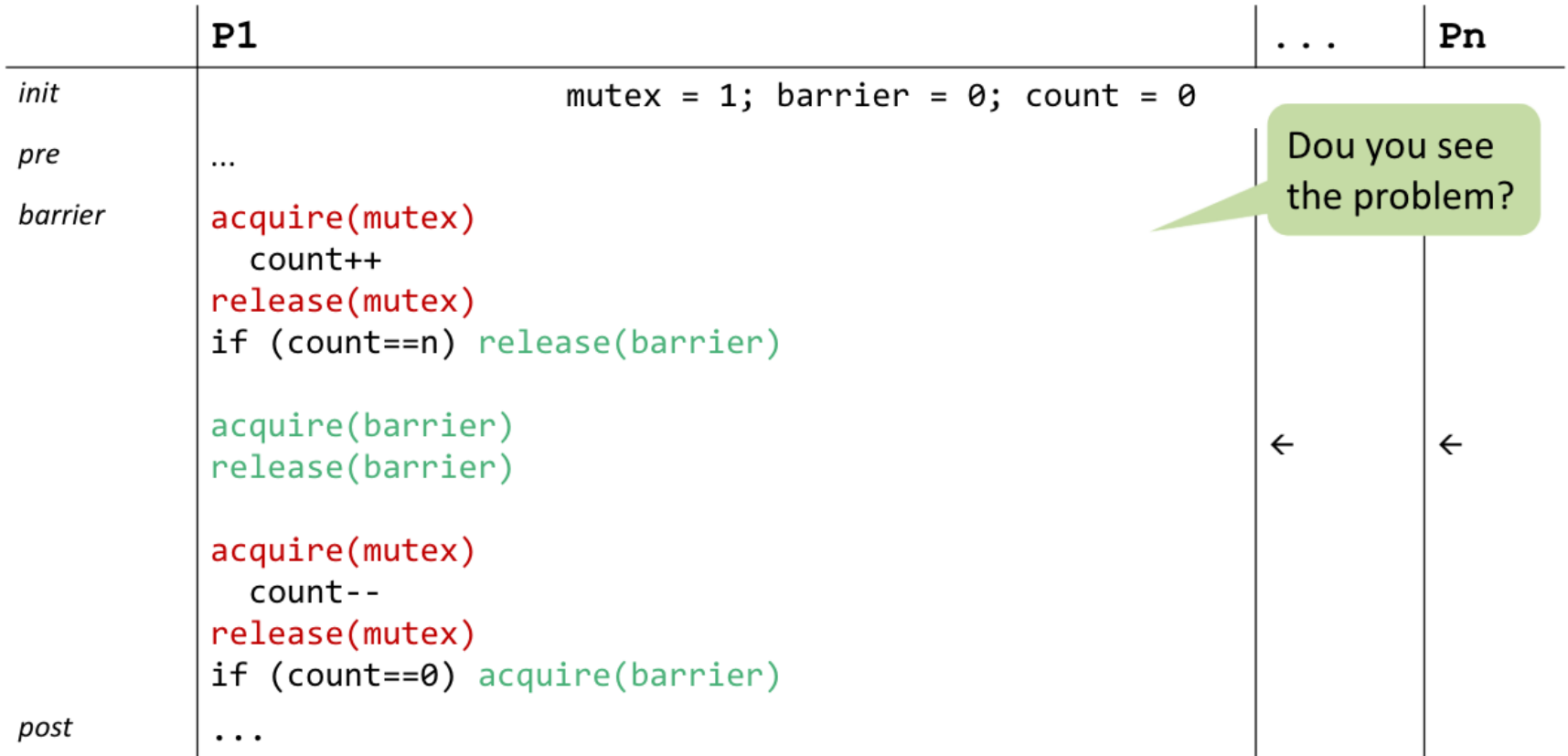
Semaphores **barrier**, **mutex**. Integer count.

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

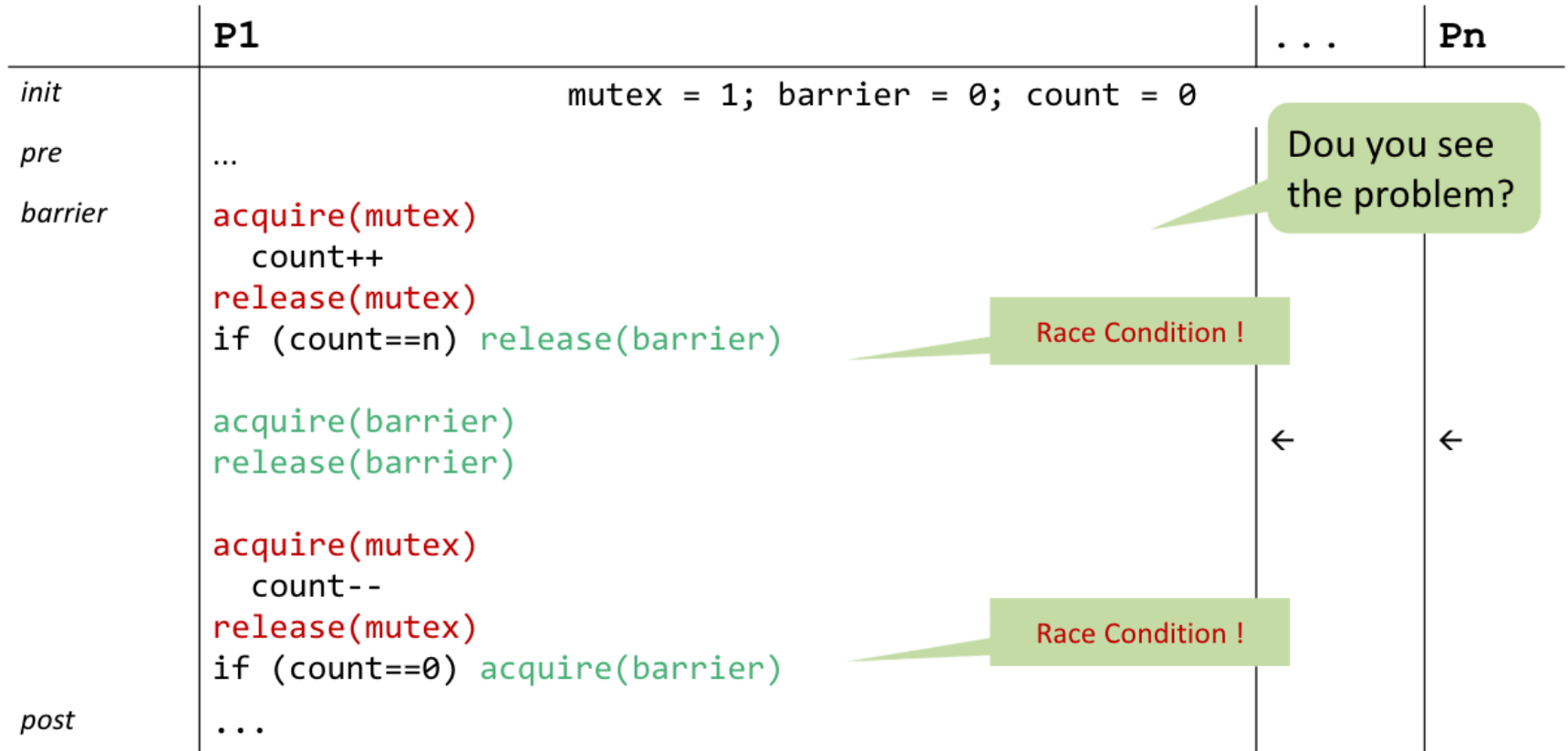


turnstile

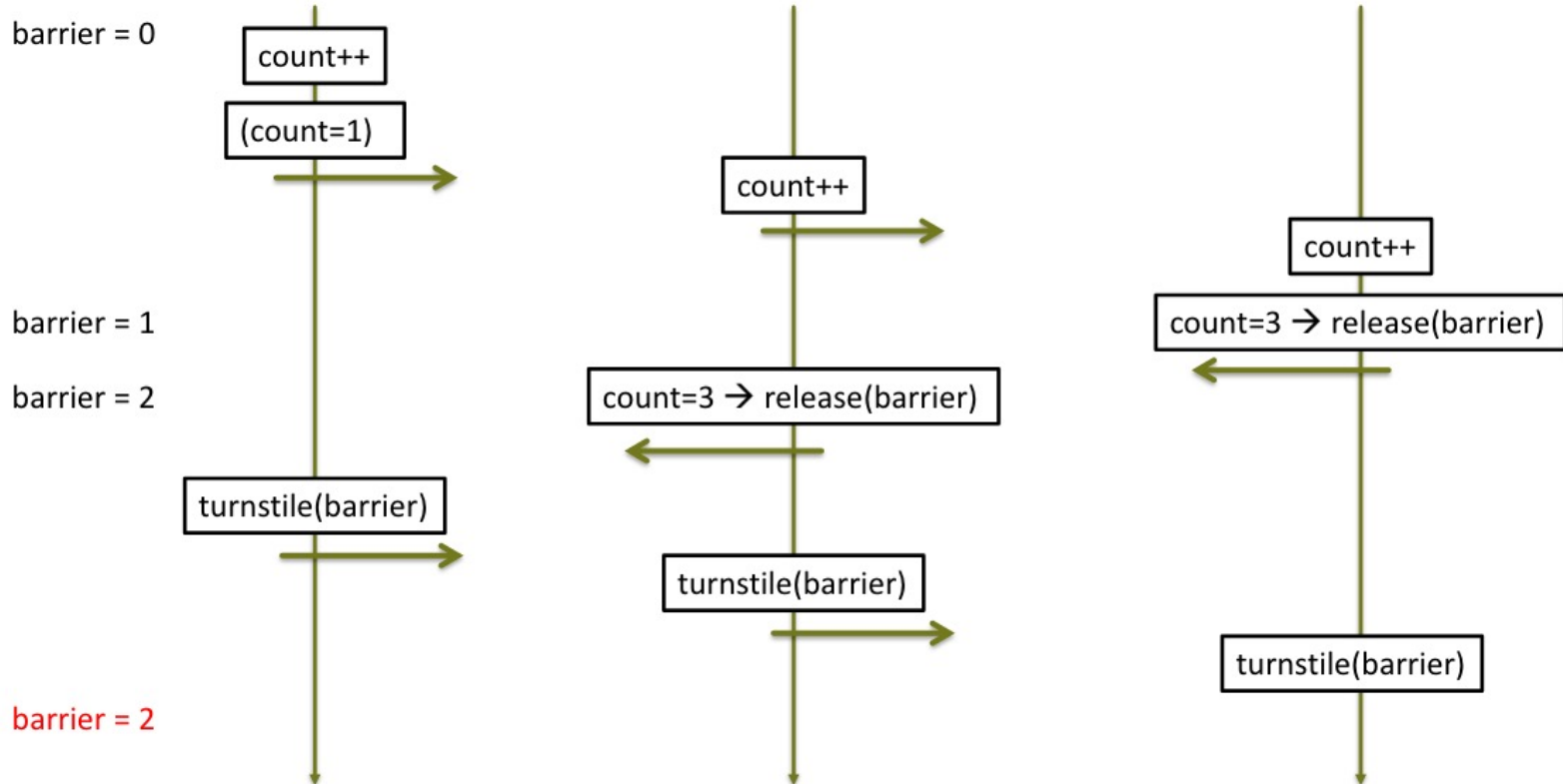
Reusable Barrier



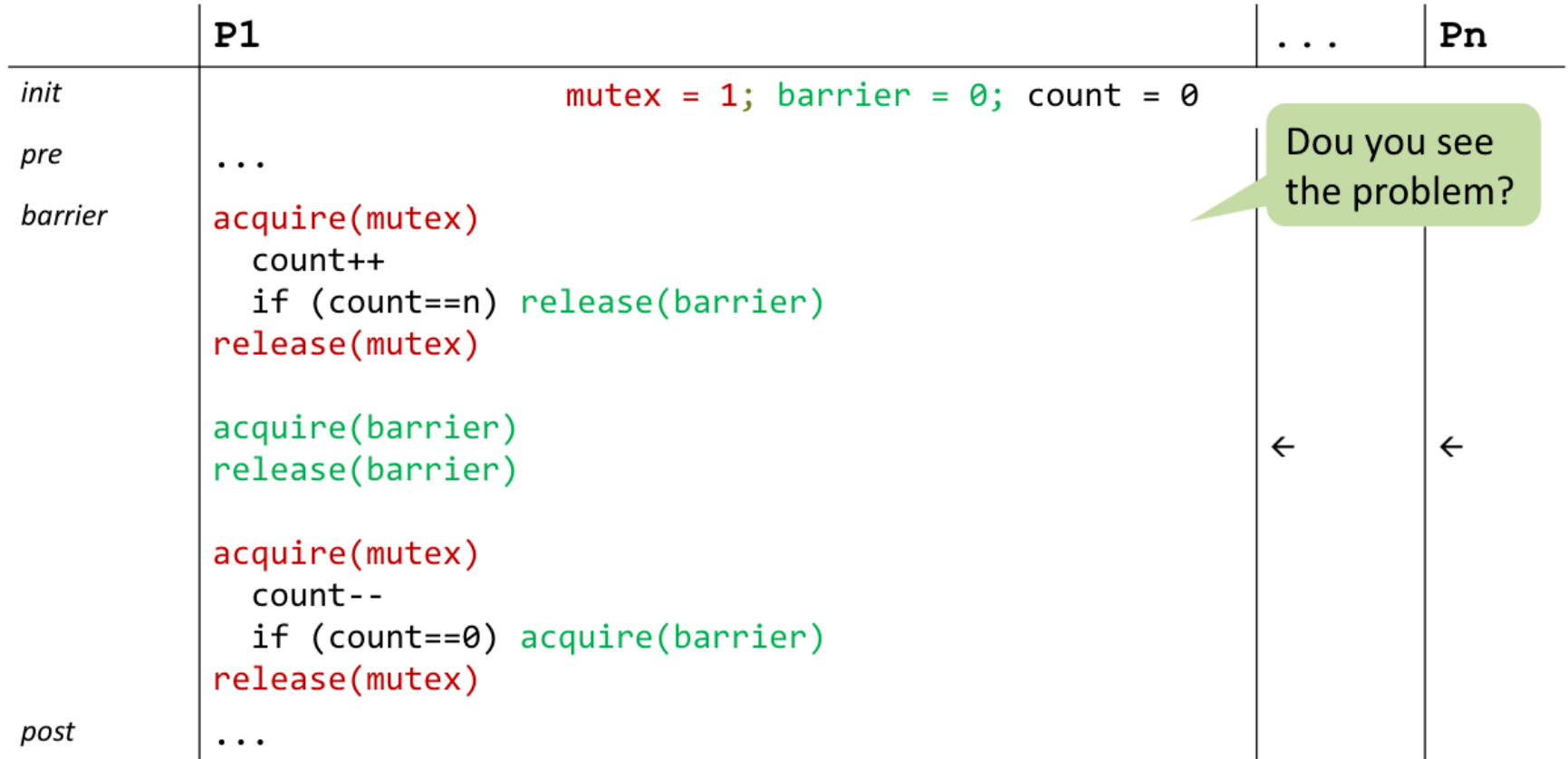
Reusable Barrier



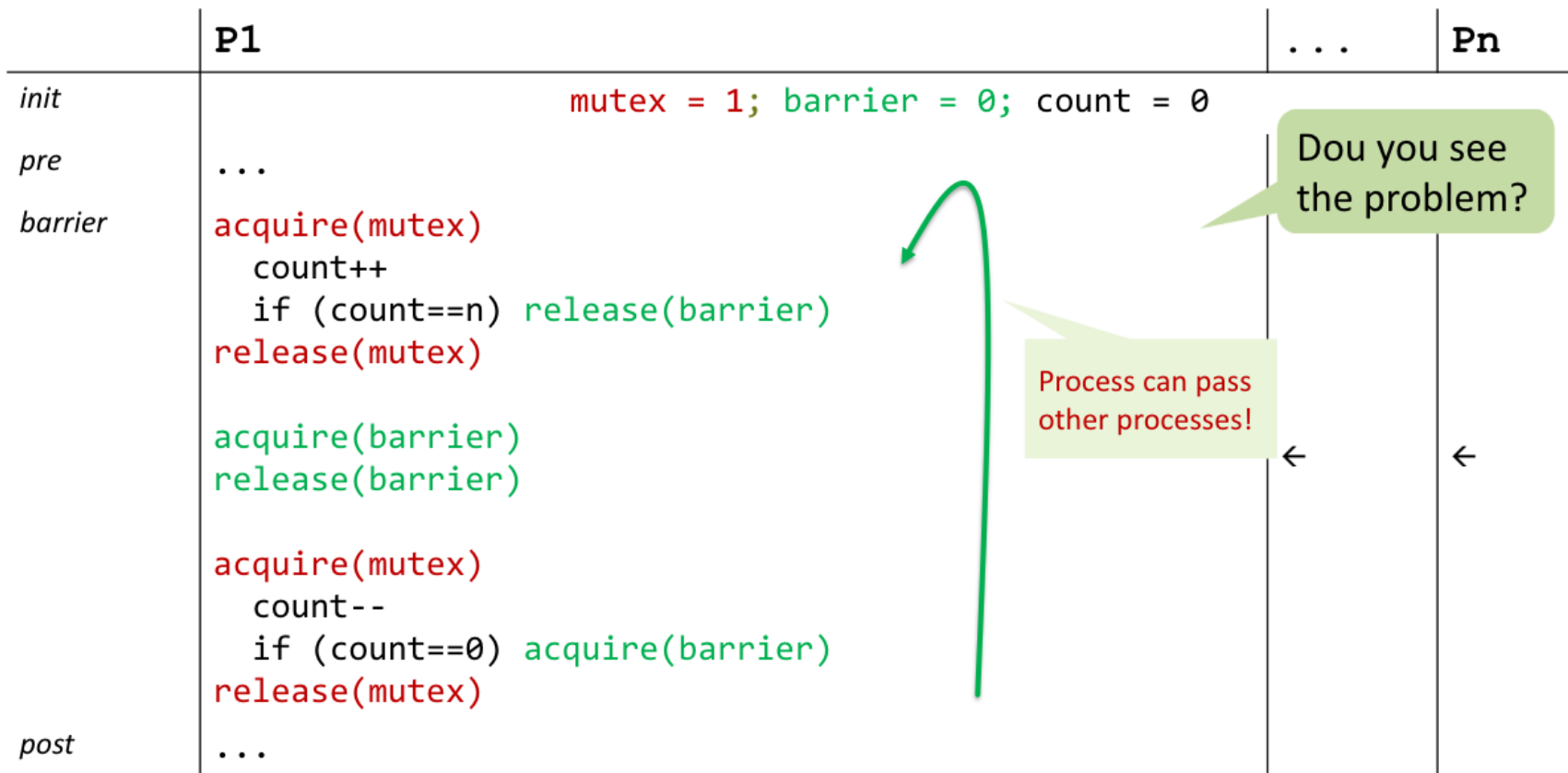
Scheduling Scenario



Reusable Barrier 2nd try



Doesn't quite work yet



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
```

Exercise 8

Assignment 8: Overview

- Why do we need a memory model?
- Why don't we simply tell the compiler "execute everything exactly as I wrote it"?
- How can we use Javas memory model to reason about executions?

Kahoot