

# Parallele Programmierung FS25

Exercise Session 8

Jonas Wetzel

# Plan für heute

- Organisation
- Nachbesprechung Exercise 7
- Theory
- Intro Exercise 8
- 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](mailto: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](mailto: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?

So far:

- Programming with locks and critical sections
- Key guidelines and trade-offs
- Bad interleavings (high level races)

Now:

- The unfortunate reality of parallel programming in practice – memory models
- **Why you must avoid data races** (= low level races / memory reorderings)
- Implementation of a Mutex with Atomic Registers

*Dekker's algorithm*

*Peterson's algorithm*

Interested in Bachelor/Semester project?  
<http://spcl.inf.ethz.ch/SeMa>

# Plan für heute

- Organisation
- **Nachbesprechung Exercise 7**
- Theory
- Intro Exercise 8
- Exam Questions
- Kahoot



# Post-Discussion 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

# How do we get livelocks?

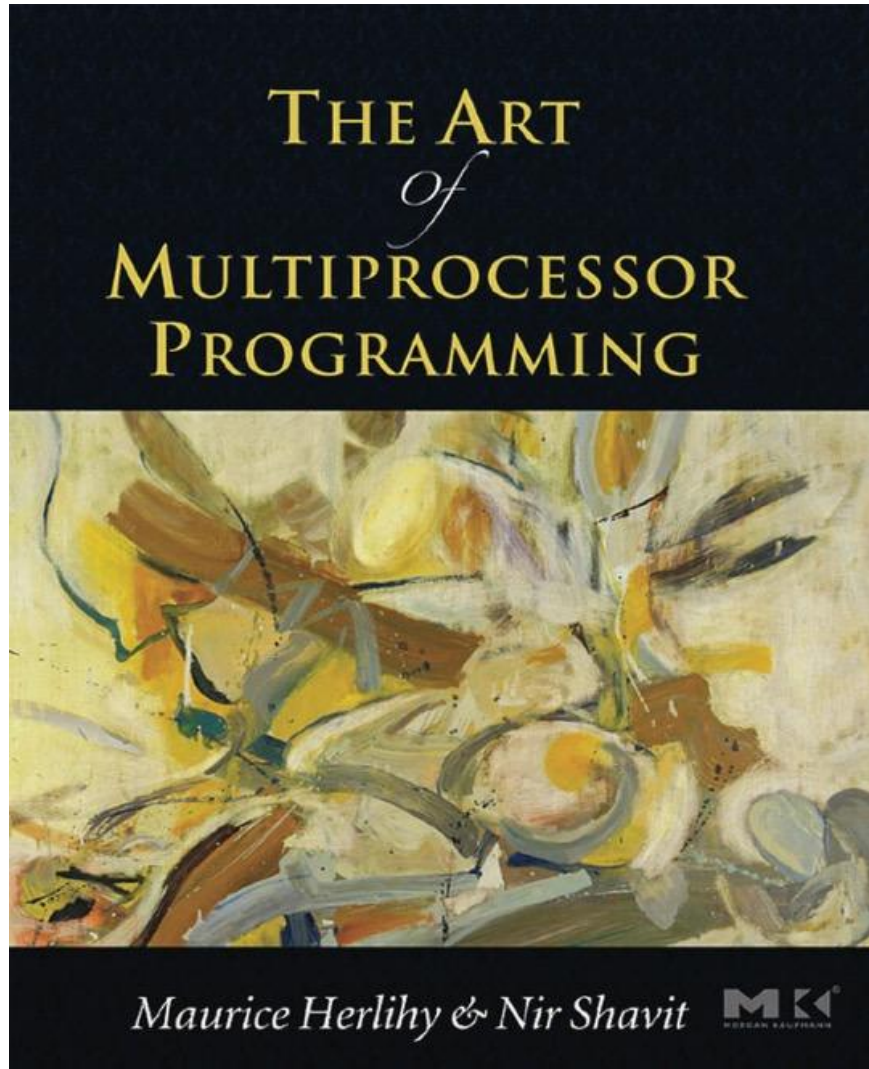
- See livelock example
- Go over solution?

# Plan für heute

- Organisation
- Nachbesprechung Exercise 7
- **Theory**
- Intro Exercise 8
- Exam Questions
- Kahoot



I recommend reading:



**The Art of Multiprocessor Programming:** An excellent book that covers almost everything from the lecture, especially for the second part of the course.

(WARNING: The 2008 version of the book has some small mistakes!)

# 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  $f$  and  $g$  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  $b < a$   $\square$   $a == 1$  and  $b == 0$ .

## 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  $b < a$   $\square$   $a == 1$  and  $b == 0$ .

But if  $a == 1$   $\square$   $y = 1$  *happened before*  $a = y$ .  
And if  $b == 0$   $\square$   $b = x$  *happened before*  $x = 1$ .

Because we assume that programs execute in order:

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

## 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  $b < a$   $\square$   $a == 1$  and  $b == 0$ .

But if  $a == 1$   $\square$   $y = 1$  *happened before*  $a = y$ .  
And if  $b == 0$   $\square$   $b = x$  *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  $a = y$   $\square$  **Contradiction**  $\Leftarrow$

But does this really work?

# No

Optimizations by Compiler  
Optimizations by Hardware

(basically the 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);  
}
```

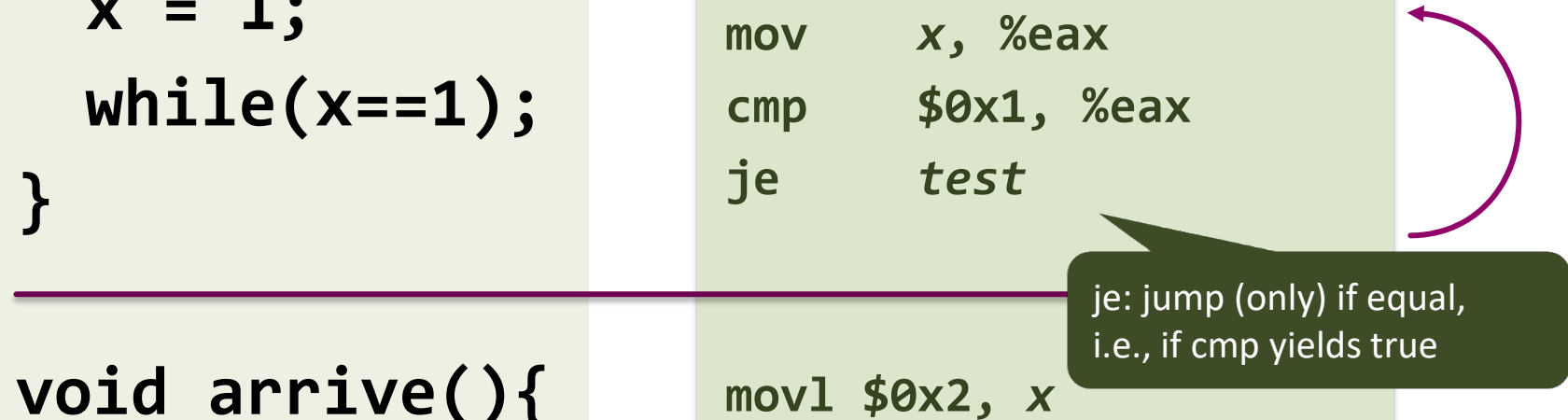
```
void arrive(){  
    x = 2;  
}
```

### 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

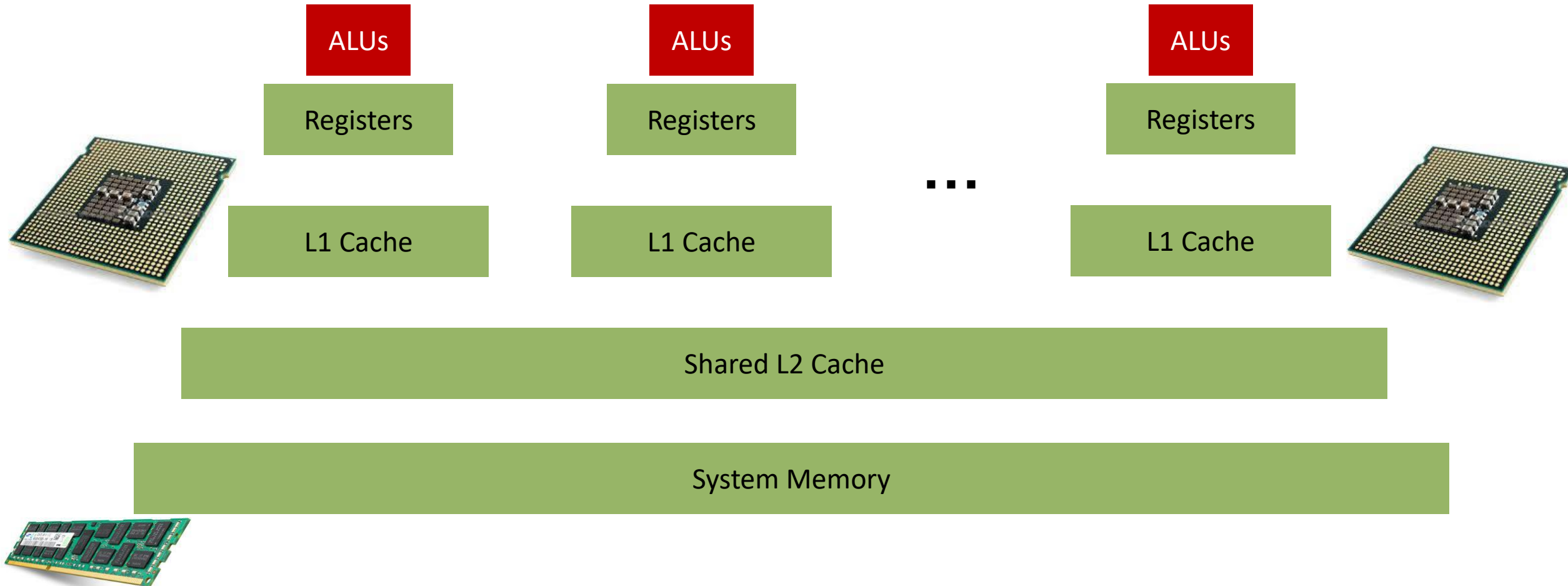
System Memory

100 ns

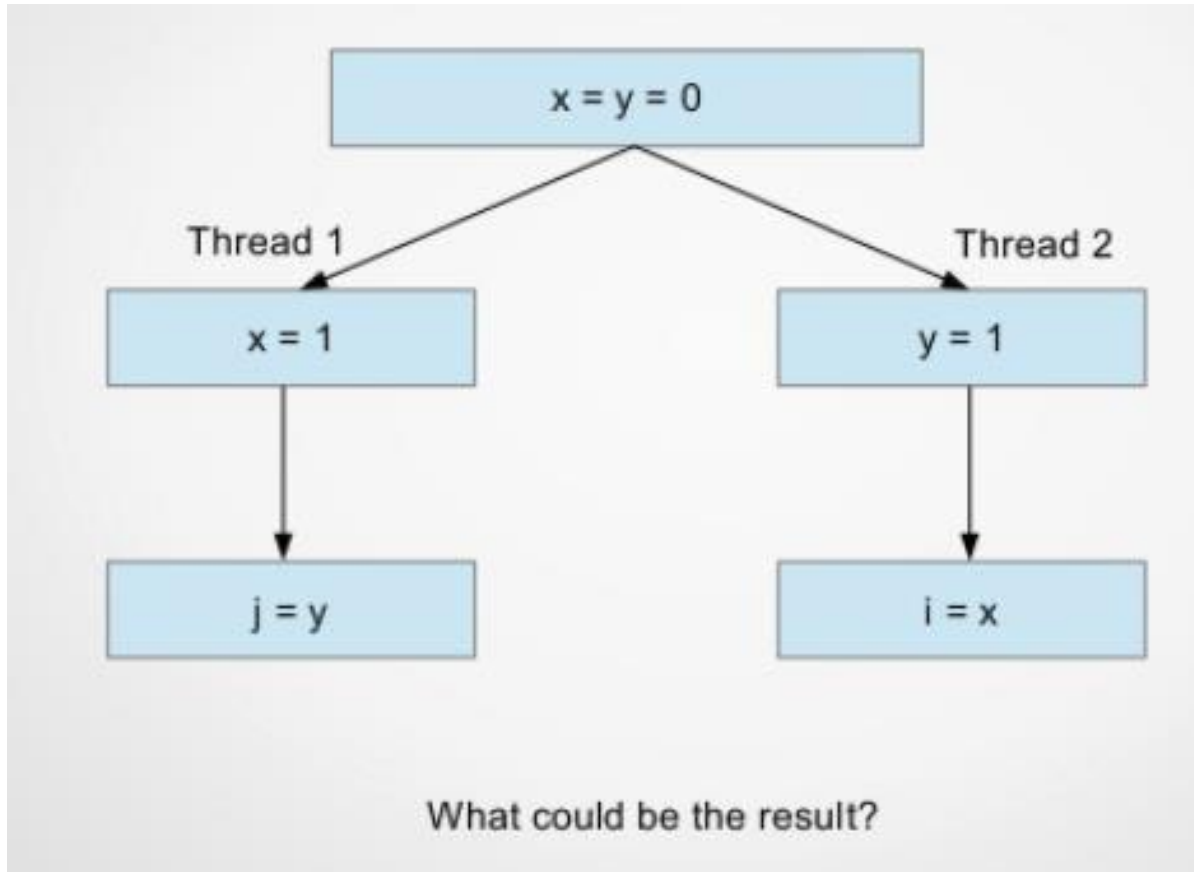
**slow, high latency, low cost, high capacity**



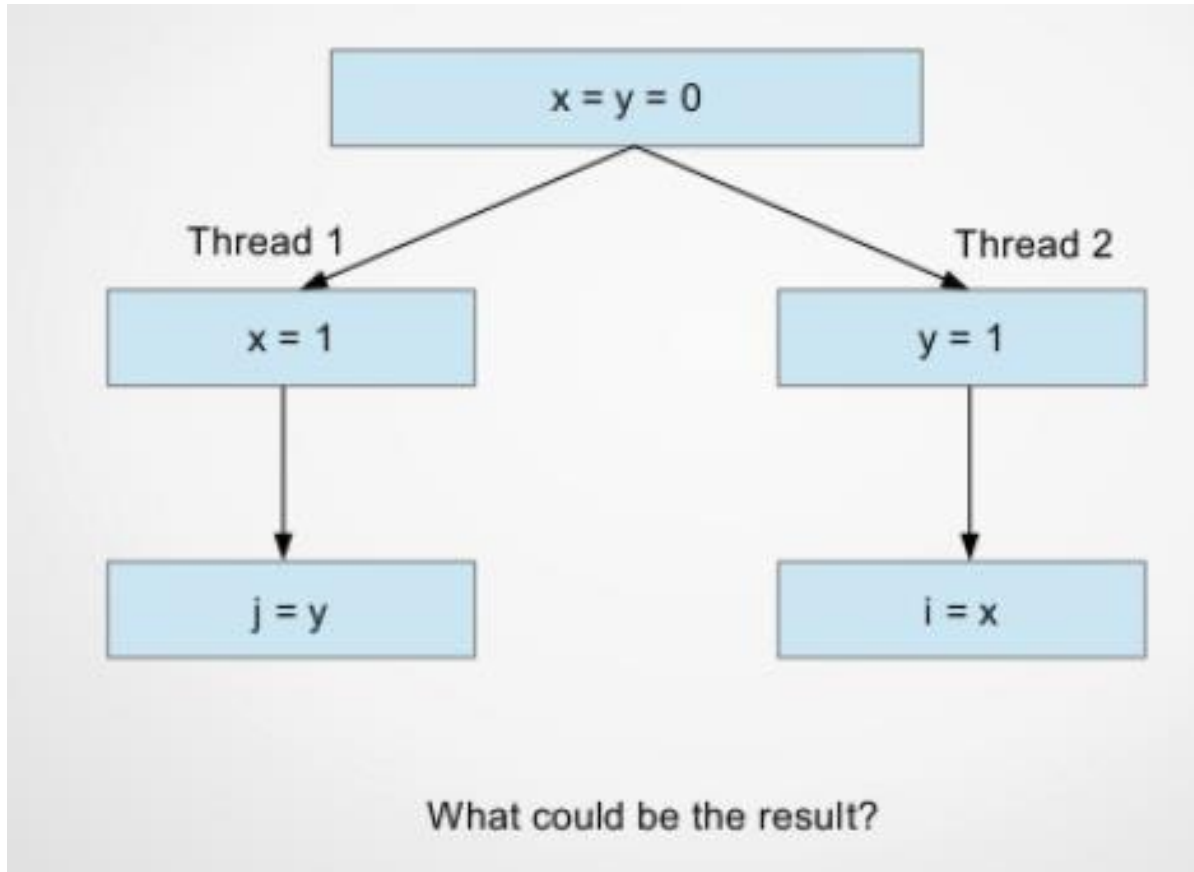
# Memory hierarchy (many cores)



## 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?)**



# 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 □ )

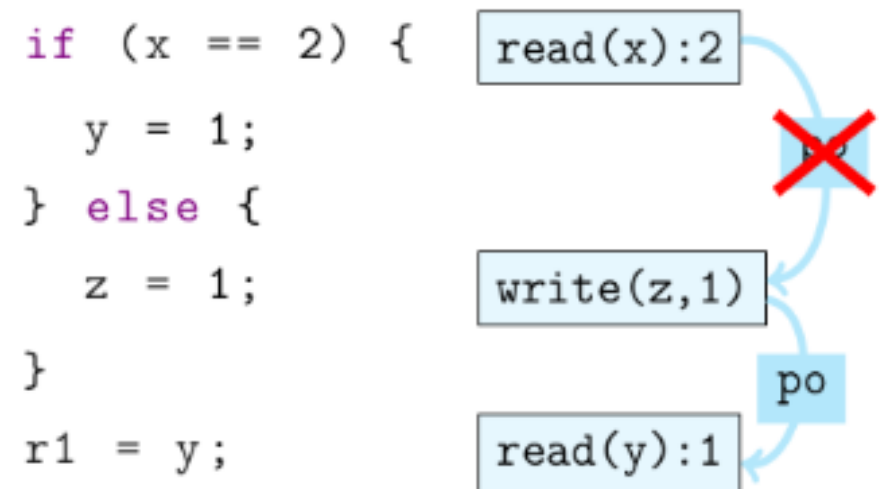
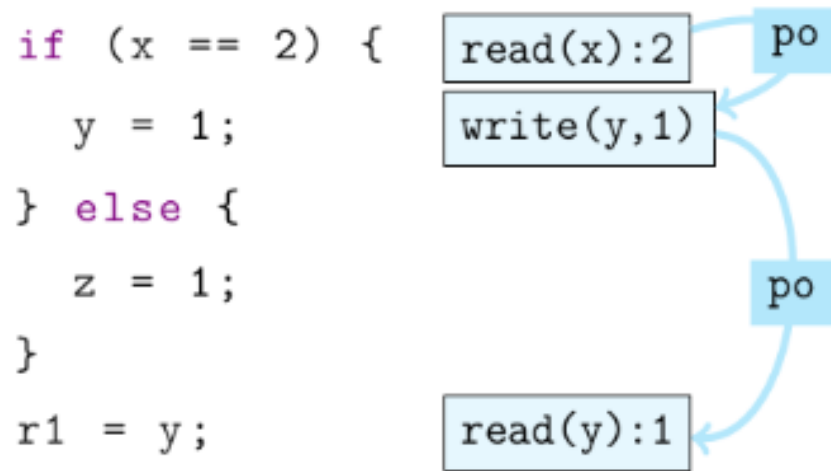
- **JMM defines *Actions*: `read(x) : 1` “read variable x, the value read is 1”**

- ***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

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

## □ Synchronization actions are:

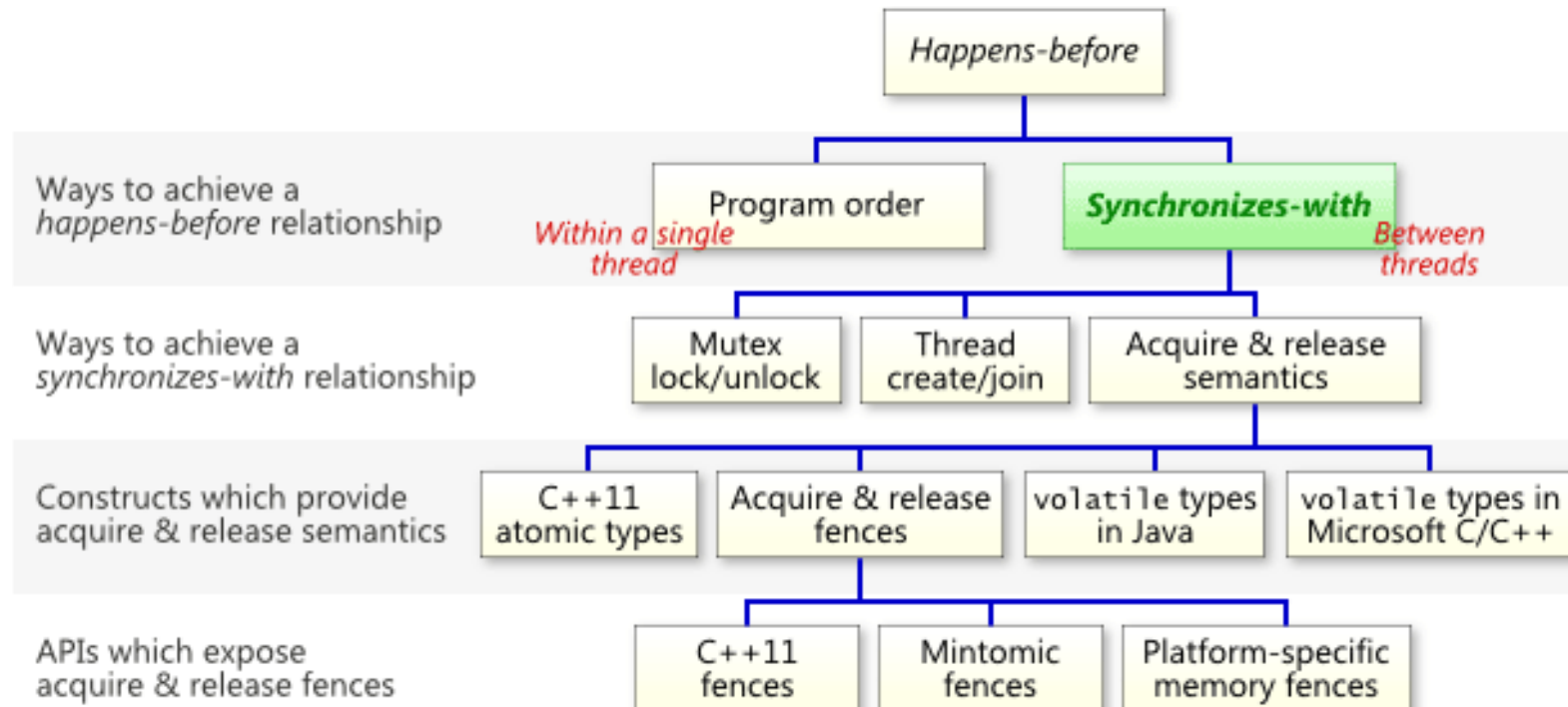
- Read/write of a volatile variable
- Lock monitor, unlock monitor
- First/last action of a thread (synthetic)
- Actions which start a thread
- Actions which determine if a thread has terminated

## □ 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

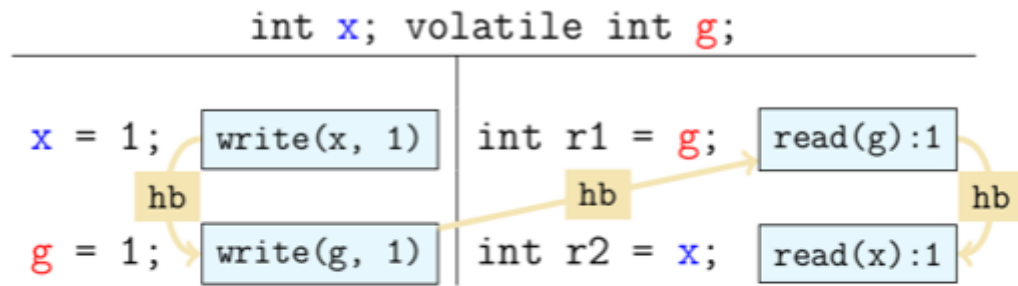
- SW only pairs the specific actions which "see" each other
- A volatile write to x synchronizes with subsequent read of x (subsequent in SO)
- The transitive closure of PO and SW forms HB
- HB consistency: When reading a variable, we see either the last write (in HB) or any other unordered write.
 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



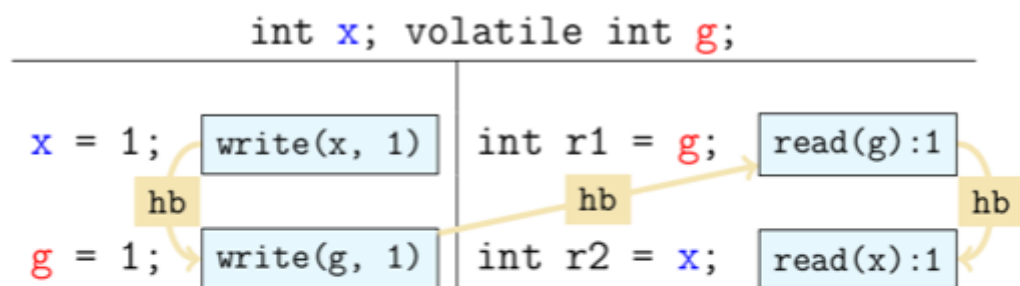
volatile int x, y;	
x = 1;	y = 1;
int r1 = y;	int r2 = x;

Exercise: List all outcomes  
(r1,r2) allowed by the JMM.

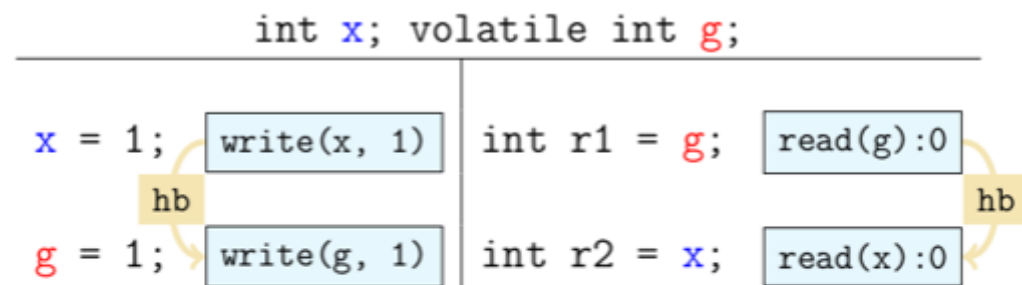
# Example



# Example

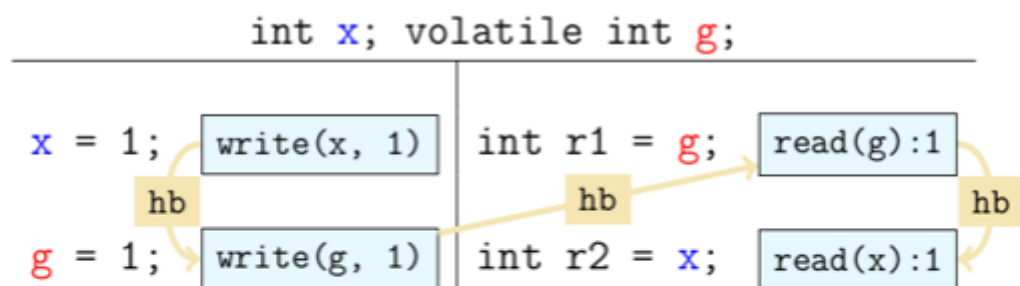


Case 1: HB consistent, observe the latest write in  $\xrightarrow{\text{hb}}$   
 $(r1, r2) = (1, 1)$

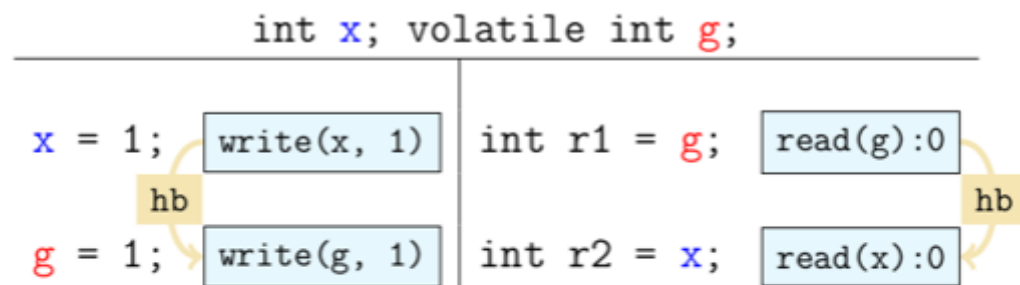




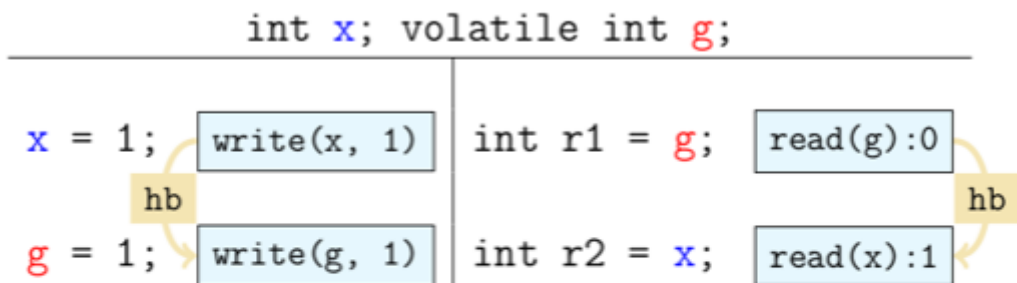
# Example



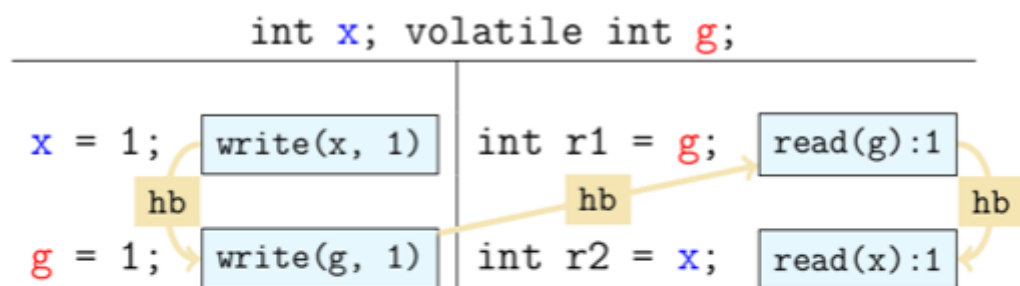
Case 1: HB consistent, observe the latest write in  $\xrightarrow{\text{hb}}$   
 $(r1, r2) = (1, 1)$



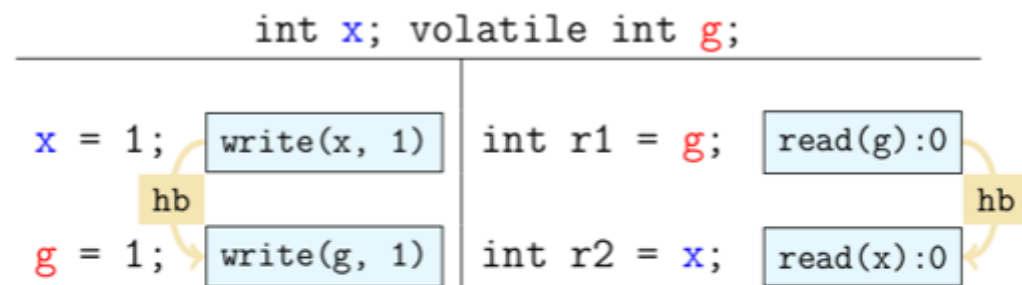
Case 2: HB consistent, observe the default value  
 $(r1, r2) = (0, 0)$



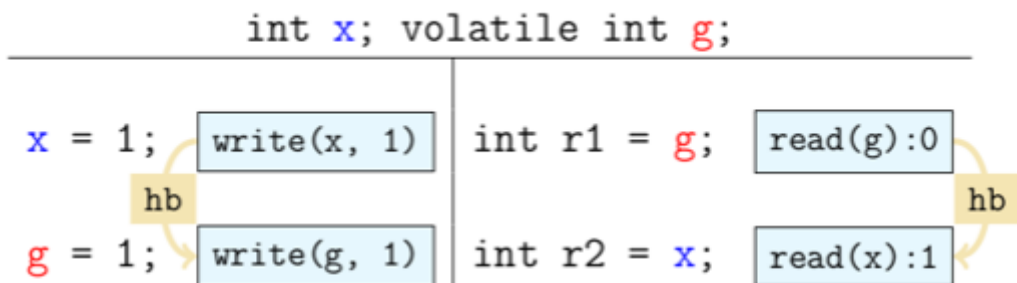
# Example



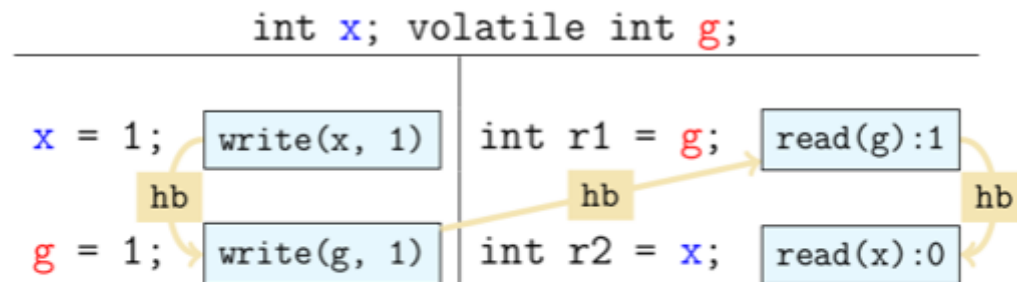
Case 1: HB consistent, observe the latest write in  $\xrightarrow{\text{hb}}$   
 $(r1, r2) = (1, 1)$



Case 2: HB consistent, observe the default value  
 $(r1, r2) = (0, 0)$

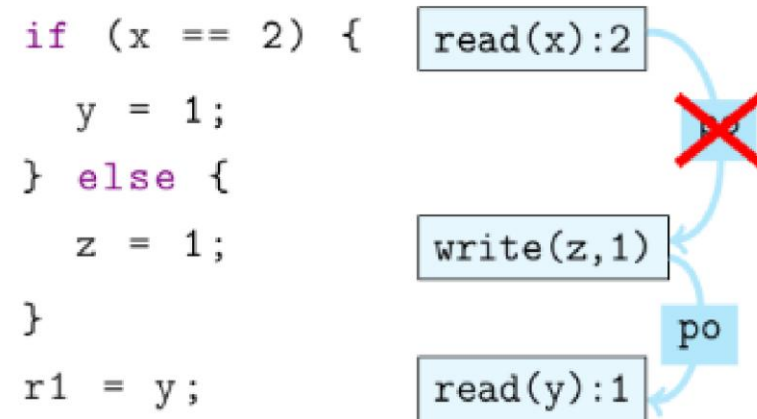
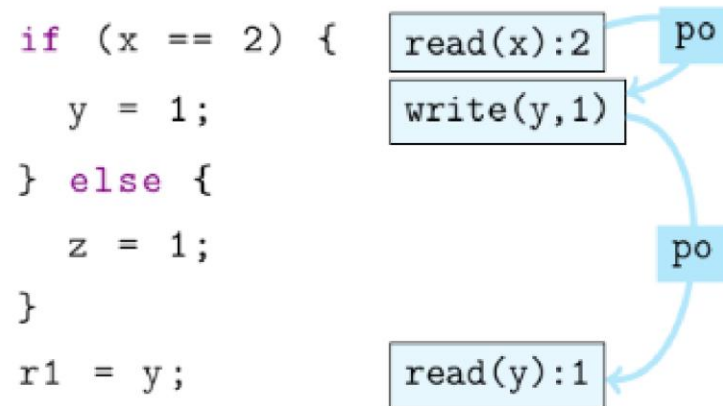


Case 3: HB consistent (!), reading via race!  
 $(r1, r2) = (0, 1)$



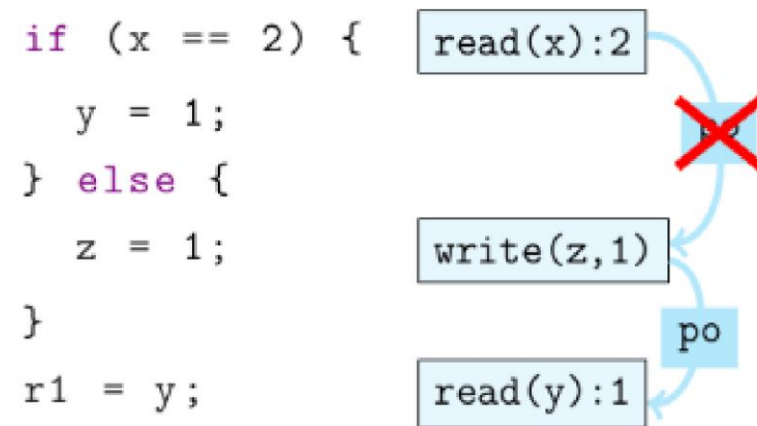
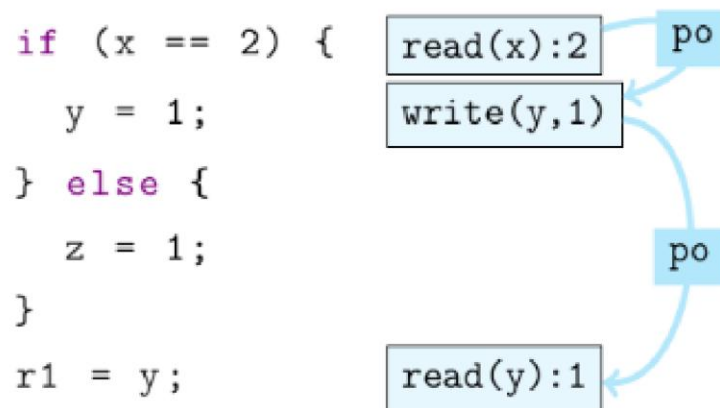
# Lecture recap: Program Order

- The code in a program is executed in a certain order
- Executing a line of code means some “action” is happening → we can look at the code and define a partial order of these actions!



# Lecture recap: Program Order

- PO: Transitive closure of “Action from statement S1 happens before that of S2 (if they both happen)”
- This is a partial order because the relation is not total, i.e. not all statements are part of every execution!



# Lecture recap: Program Order

- We want to allow the compiler / hardware to optimize our code, i.e. remove useless code:
- `int a=0;`
- `for (int i=0; i<10 i++) {a++;}`
- In sequential code we would expect this to be “rewritten” to `a=10` since anyway nobody sees the intermediate values.
- But what if `a` is shared?

# Lecture recap: Synchronization Actions (SAs)

- Java defines synchronization actions (read/write of volatile variable, lock/unlock, etc.)
- SAs within thread obeys PO
- All threads see SAs in the same order (synchronization order SO)
- Reads are consistent in SO (see the last written value)

# Lecture recap: Synchronizes With

- SAs accross threads synchronize with each other, i.e., a volatile read sees the last value written by another thread
- Transitive closure of PO and SW forms happens before order
- All values we observe must obey this happens before order!

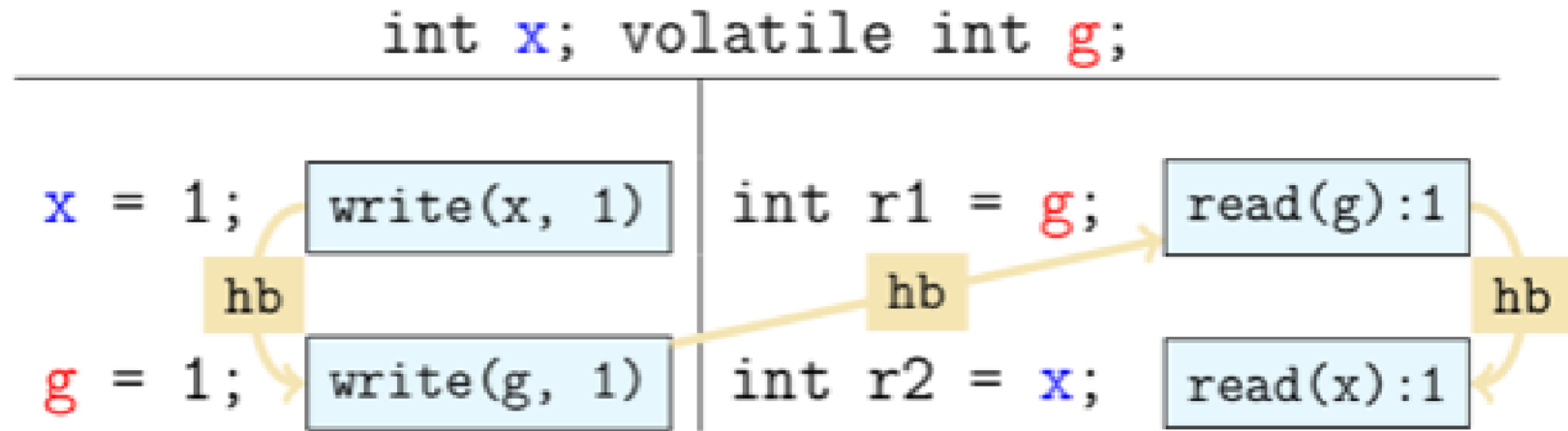
# Lecture recap: Happens-before relationship

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  $hb(x, z)$ .

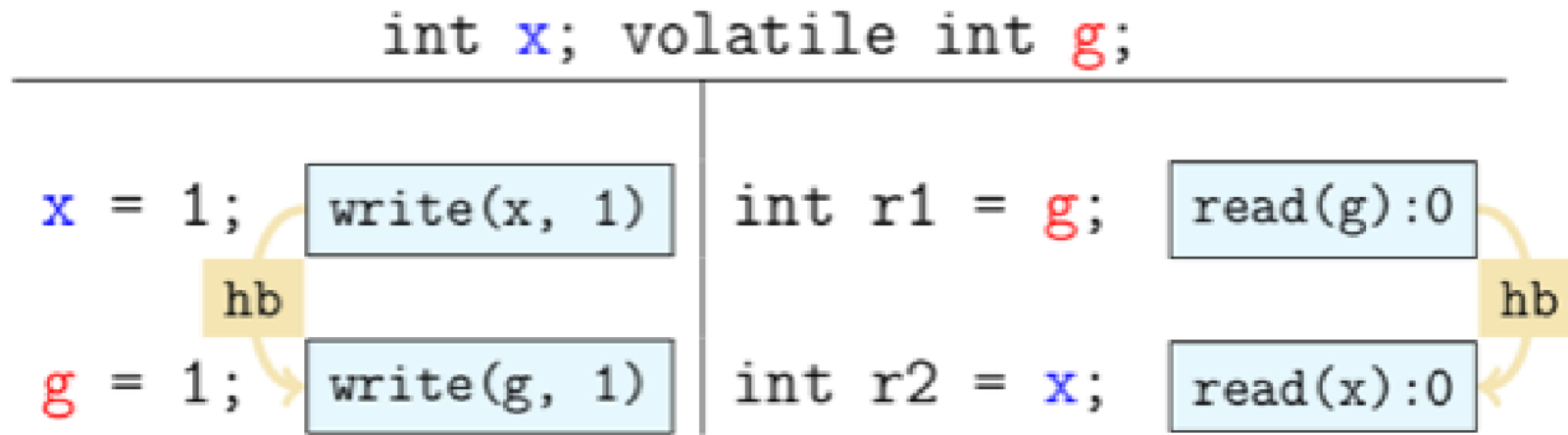


# Examples



- 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!
- Above we show the HB order for (1,1)
- What must happen for (0,0)?

# Examples



- Initial value of x,y is 0.
- What must happen for (0,0)? - right thread runs first!

# Lecture recap: 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
- Remember the 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

# Lecture recap: State Space Diagram

turn = 1;

## ***Process P***

do

p1: Non-critical section P

p2: while turn != 1

p3: Critical section

p4: turn = 2

## ***Process Q***

do

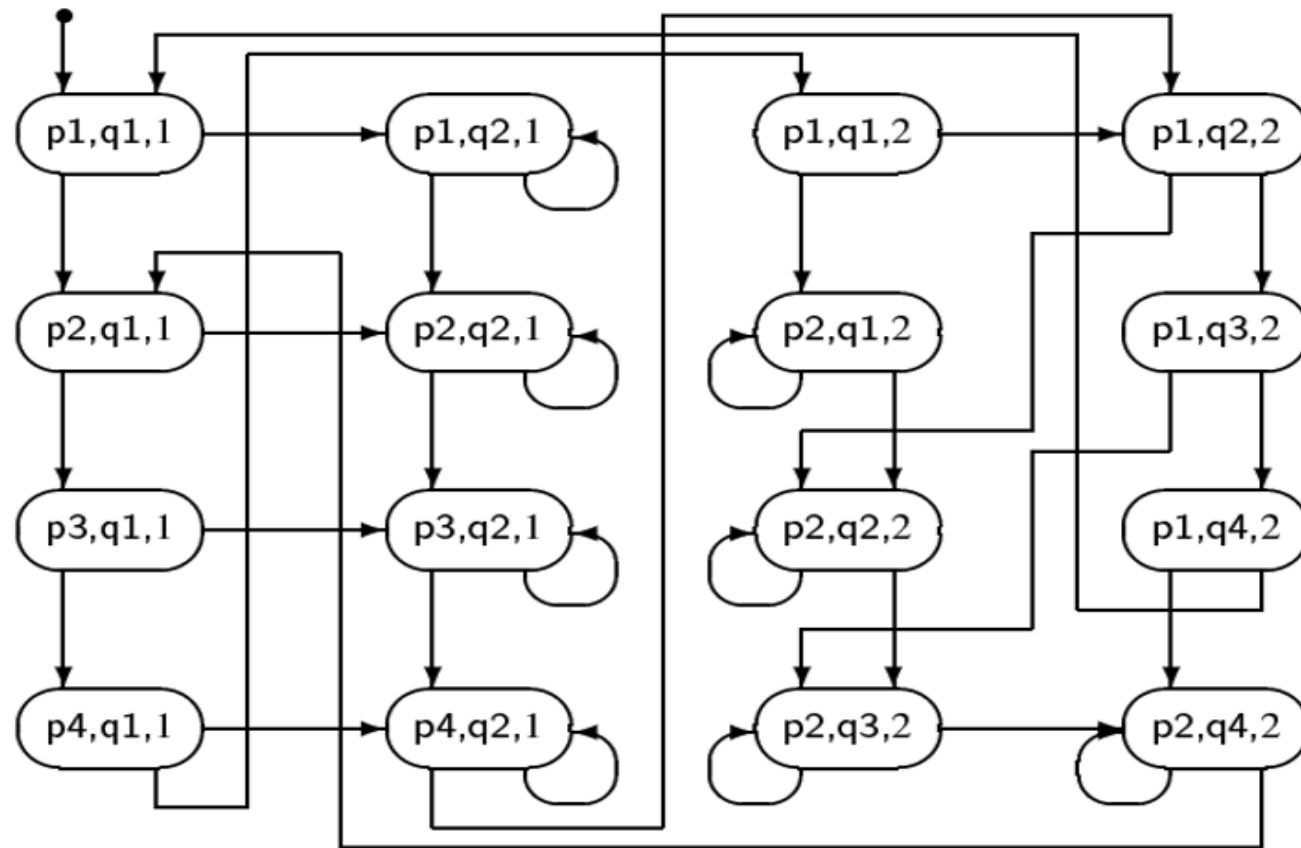
q1: Non-critical section Q

q2: while turn != 2

q3: Critical section

q4: turn = 1

# Essential



**P**

p1: Non-critical section P

p2: while turn != 1

p3: Critical section

p4: turn = 2

**Q**

q1: Non-critical section Q

q2: while turn != 2

q3: Critical section

q4: turn = 1

# 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
- Mutual exclusion holds!

# Freedom from deadlock

- *“If some processes are trying to enter their critical sections then one of them must eventually succeed.”*
- P is trying to enter its CS when the control pointer is at p2  
(awaiting turn to have the value 1. p2: turn==1)
- Q is trying to enter its CS when the control pointer is at q2  
(q2: turn==2)

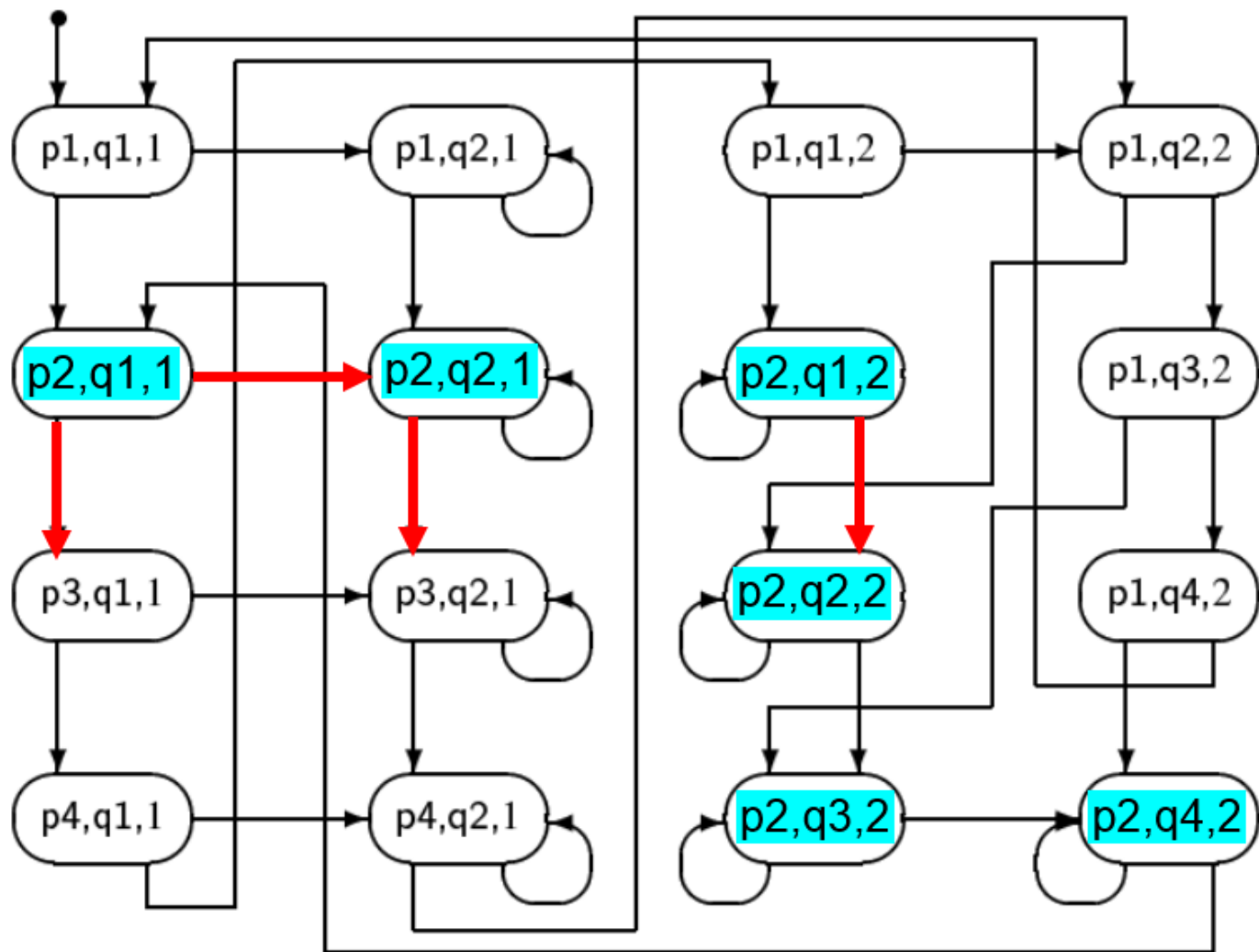
# Freedom from deadlock

- 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.  
i.e. the control pointer can always move to point to p3



# Freedom from deadlock

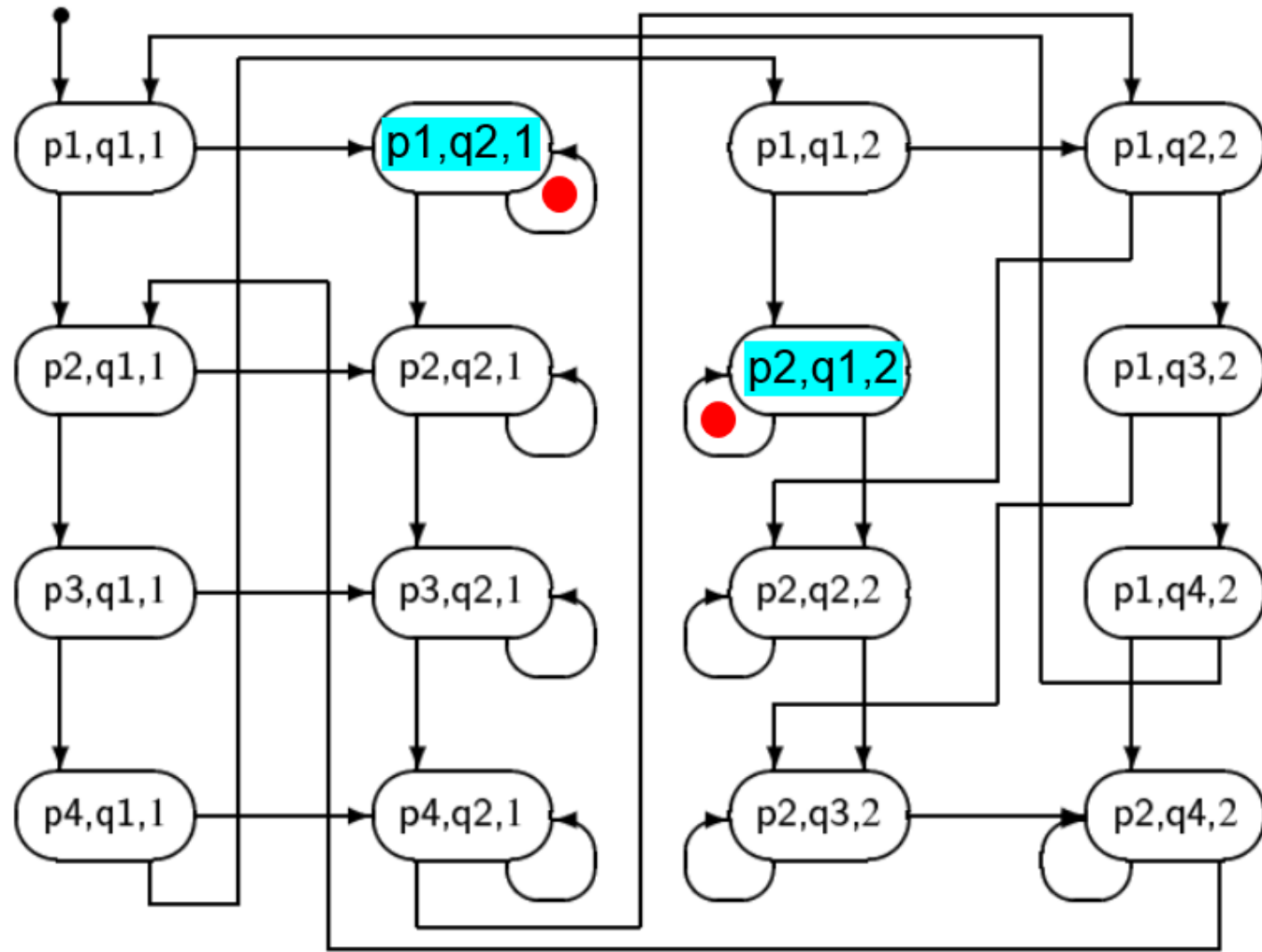
- Typically, a deadlocked state has no transitions leading from it, i.e. no statement is able to be executed.
- Sometimes a cycle of transitions may exist from a state for each process, from which no useful progress in the parallel program can be made. The program is still deadlocked but this situation is sometimes termed '*livelock*'. Every one is 'busy doing nothing'.



There is always a path for P to execute p2 (turn == 1)

# 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



If a process does not make progress  
from its Non-critical section, starvation  
is possible in this example

# 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

# Hardware support for atomic operations

- 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();

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

boolean compareAndSet(boolean expect, boolean update);

boolean getAndSet(boolean newValue);

sets **newValue** and returns previous value.



# But why do these operations work without volatile?

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in [The Java Language Specification \(17.4 Memory Model\)](#):

- get has the memory effects of reading a volatile variable.
- set has the memory effects of writing (assigning) a volatile variable.

Source: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>

# Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?

# Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?
  - Atomics geben uns eine Möglichkeit
  - Was wenn wir keine atomics haben?

# Was ist volatile?

- Wir wollen Locks selber bauen
- Wie machen wir das?
  - Atomics geben uns eine Möglichkeit
  - Was wenn wir keine atomics haben?
- Volatile!

# Volatile Keyword

- When multiple threads access a shared variable, each thread may keep local copy in its CPU cache, updates might not be immediately visible to other threads
- Volatile gives us visibility guarantee!

# Volatile Keyword

- Volatile variable ensures that any read or write operation always happens directly in main memory, so all threads see latest value on next read

```
class Example {  
    private volatile boolean running = true;  
  
    void stop() {  
        running = false; // Changes are immediately visible to all threads  
    }  
  
    void run() {  
        while (running) {  
            // Do work  
        }  
    }  
}
```

# Volatile Keyword

- Without volatile, another thread calling stop() might not be seen by the run() method because the CPU might cache running locally
- With volatile, the change to running is guaranteed to be visible to all threads

```
class Example {  
    private volatile boolean running = true;  
  
    void stop() {  
        running = false; // Changes are immediately visible to all threads  
    }  
  
    void run() {  
        while (running) {  
            // Do work  
        }  
    }  
}
```

# Volatile Keyword

- It also prevents optimizations like out of order execution from happening!
- Java Memory Model allows JVM and CPU to reorder instructions for optimization



# Volatile Keyword

- Without volatile, compiler or CPU might reorder (1) and (2), leading reader() to see flag == true but still read old value of x
- With volatile, (2) happens after (1), ensuring x = 42 is visible before flag = true is read.

```
PProgFS25 - Jonas Wetzel

class Example {
    private int x = 0;
    private volatile boolean flag = false;

    void writer() {
        x = 42;           // (1) Write to x
        flag = true;      // (2) Write to volatile variable
    }

    void reader() {
        if (flag) {       // (3) Read volatile variable
            System.out.println(x); // (4) Guaranteed to
        }
    }
}
```

# Volatile Keyword

- Volatile has limitations
- Does not prevent race conditions: volatile ensures visibility but not atomicity for compound actions like `count++`
- Not a replacement for synchronization: It doesn't provide mutual exclusion (locking)

# Volatile Keyword

- Volatile has limitations
- Does not prevent race conditions: volatile ensures visibility but not atomicity for compound actions like count++
- Not a replacement for synchronization: It doesn't provide mutual exclusion (locking)

```
PProgFS25 - Jonas Wetzel

class Counter {
    private volatile int count = 0;

    void increment() {
        count++; // Not atomic! Two threads might read
                // the same value and overwrite each other.
    }
}
```

```
PProgFS25 - Jonas Wetzel

class Counter {
    private AtomicInteger count = new AtomicInteger(0);

    void increment() {
        count.incrementAndGet(); // Atomic and thread-
        safe
    }
}
```

# Volatile Summary

- Volatile ensures that all reads and writes go directly to main memory, preventing stale values
- It prevents instruction reordering
- It does NOT provide atomicity or mutual exclusion
- Suitable for simple flags and state indicators, but not for counters or complex data structures

# Volatile Code Example

- See code `Visibility.java`

# Volatile

- Suitable for simple flags and state indicators, but not for counters or complex data structures
- Can we build a lock with that?

# Volatile

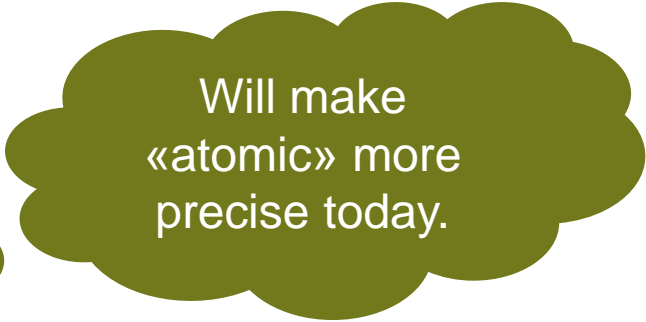
- Suitable for simple flags and state indicators, but not for counters or complex data structures
- Can we build a lock with that?
- Yes, as we'll see in the next few minutes

# Beyond Locks Recap




# Assumptions

In the following we assume



Will make  
«atomic» more  
precise today.



You know  
how to fix this  
with volatile!

- 1) atomic reads and writes of variables of primitive type
- 2) no reordering of read and write sequences (! not true in practice ! here for simplicity !)
- 3) threads entering a critical section will leave it eventually

Otherwise we assume a multithreaded environment where processes can interleave arbitrarily.

We make no assumptions for progress outside of critical sections (i.e., threads may stall outside of a CS)!

# 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

# Lecture recap: 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
- Remember the 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) 3 wantp = true      4 critical section      5 wantp = false
                        while(wantq)  wantq = true

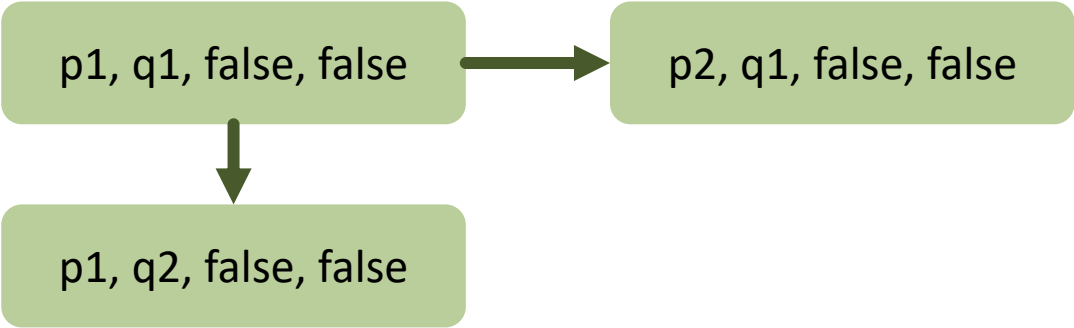
```

p1, q1, false, false

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

# 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

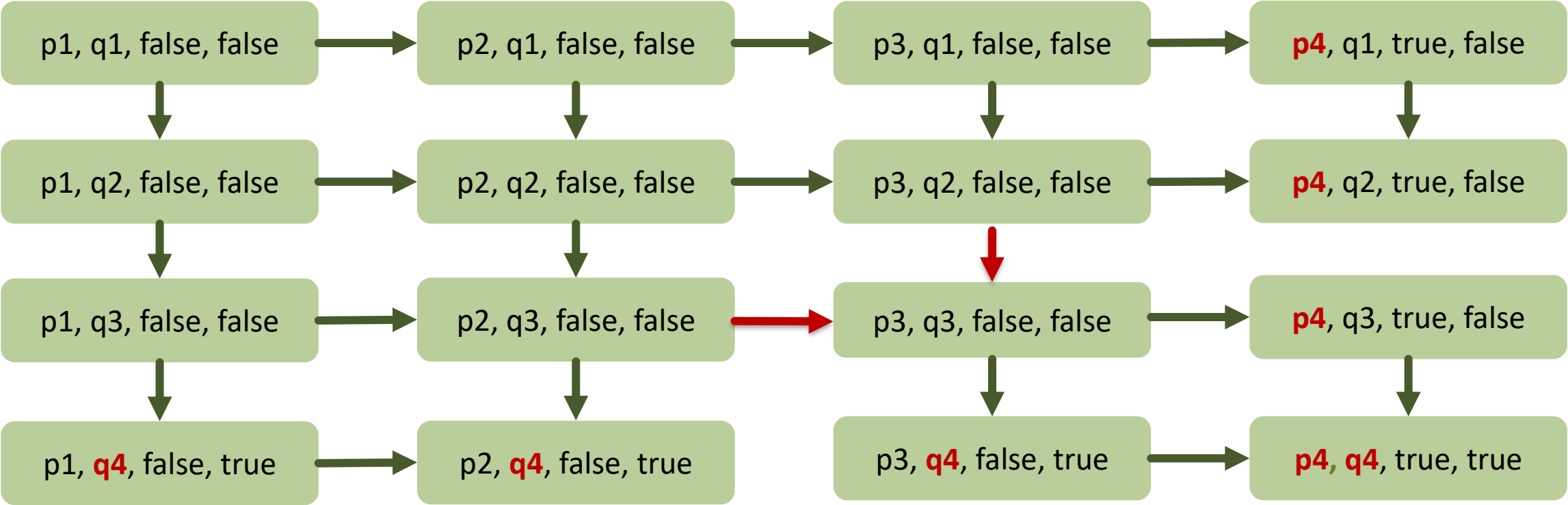


p1	non-critical section
p2	while(wantq);
p3	wantp = true
p4	critical section
p5	wantp = false

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

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

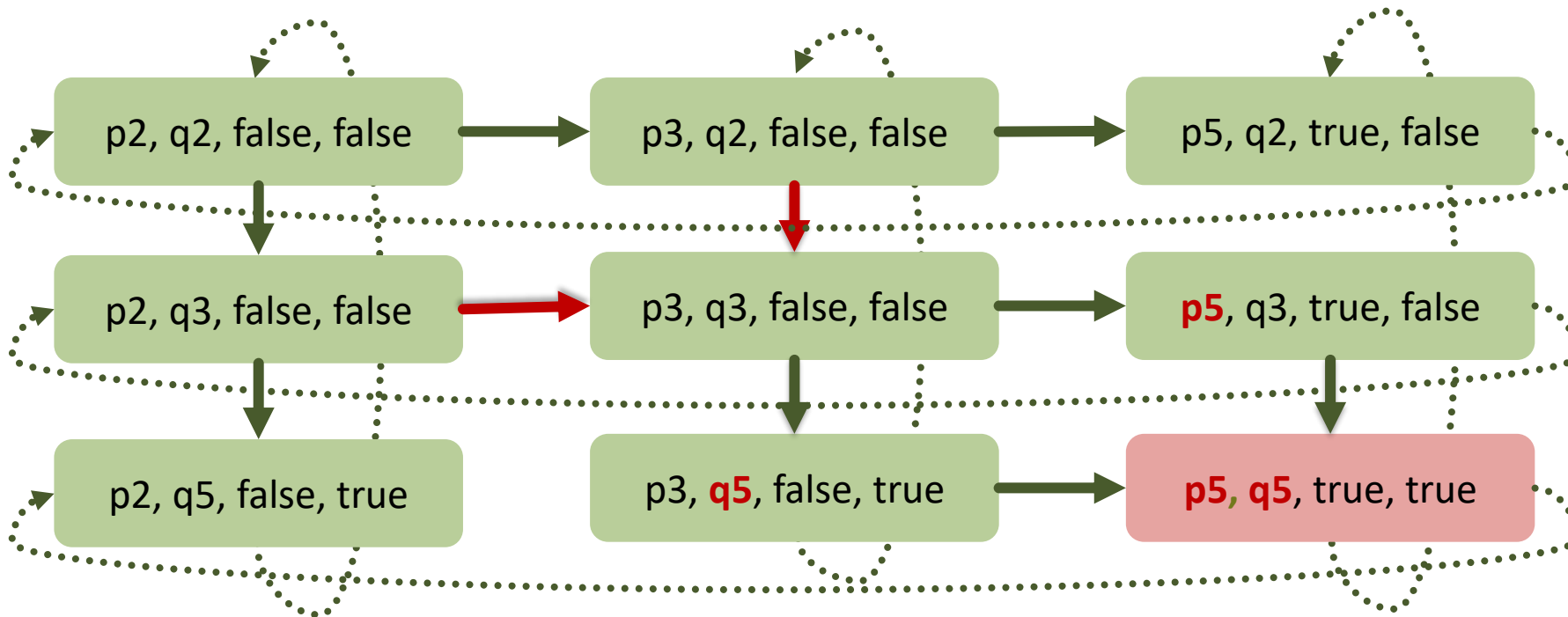




# Reduced state space diagram [p, q, wantp, wantq] – only states 2, 3, and 5

1 non-critical section → 2 await wantq == false  
await wantp == false → 3 wantp = true  
wantq = true → 4 critical section → 5 wantp = false  
wantq = false

All of interest covered:



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

no mutual exclusion !

# 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

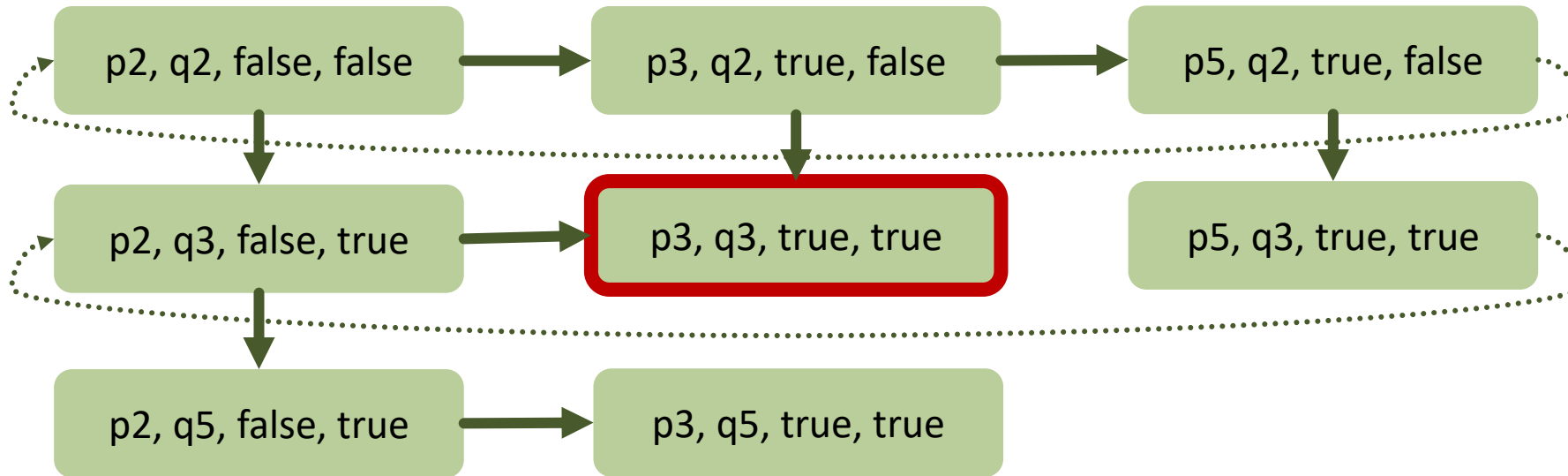
Do you see the problem?

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

```

1 non-critical section  2  wantp = true    3  while(wantp)    4  critical section    5  wantp = false
                        6  wantq = true    7  while(wantq)

```



deadlock !

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

## 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

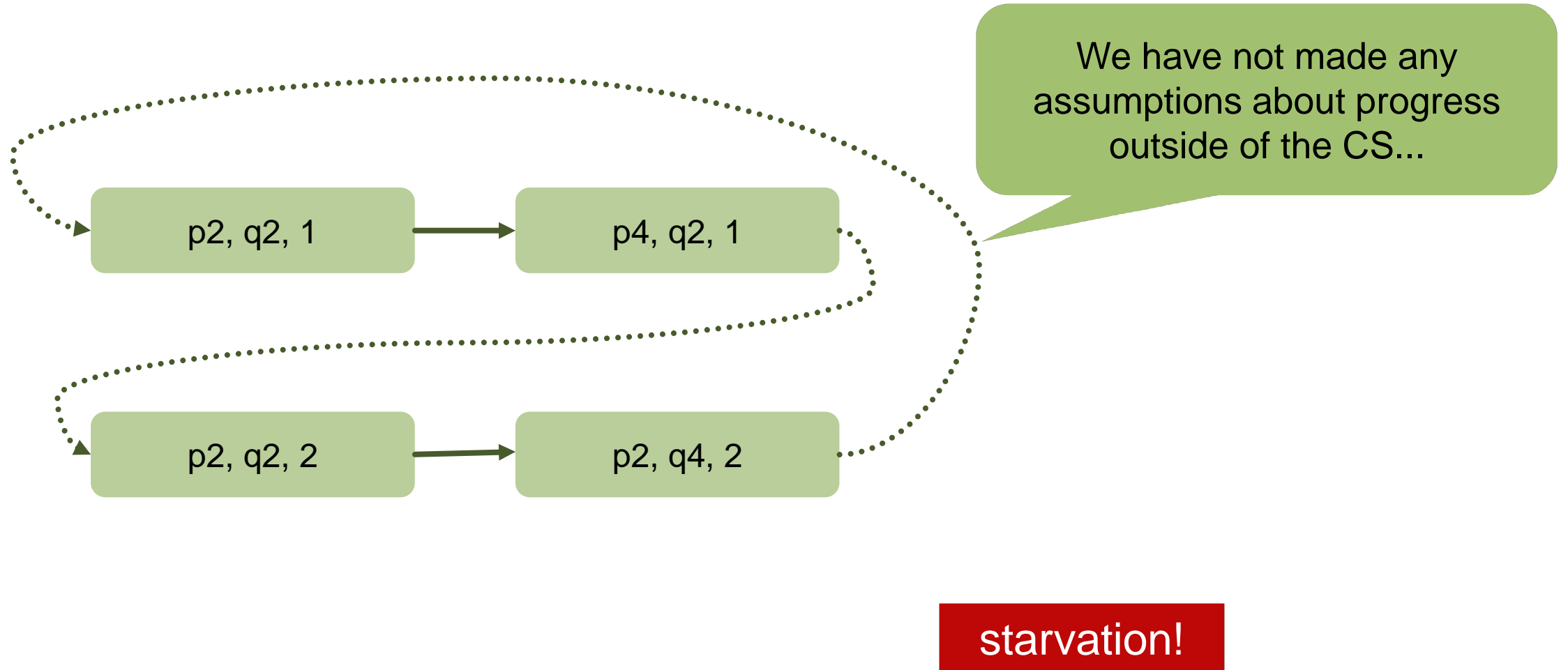
q2 while(turn != 2);

q3 critical section

q4 turn = 1

Do you see the problem?

## State space diagram [p, q, turn]



# 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. The program is still deadlocked but this situation is sometimes termed '*livelock*'. Every one 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

So how do we fix our attempts?

# 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)

# 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**

# 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!
  - We'll see it in like 3 weeks

# Deckers Lock & Peterson Lock

- See code for both locks

# Can we build a lock with atomics?

- How?

# Hardware Semantics

atomic

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

```
    oldval = mem[a];
```

```
    if (old == oldval)
```

```
        mem[a] = new;
```

```
    return oldval;
```

# Can we build a lock with atomics?

- 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



# Plan für heute

- Organisation
- Nachbesprechung Exercise 7
- Theory
- **Intro Exercise 8**
- Exam Questions
- Kahoot

# Pre-Discussion 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?

## 2.1 When are interleavings bad?

Assume there are two Java threads, sharing the variables v, w, x, y and z. The variables r1 and r2 are private. Assume the code these two threads are executing looks as follows:

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

Is the result (the values of the private variables after both threads finish the execution) always the same or could it depend on the order in which the threads are scheduled?

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

How about this version of code? Is the result always the same or does it depend on the scheduling of the

## 3 Building Blocks of the Java Memory Model

### 3.1 Relations

A relation is a mathematical concept defined over elements of a set. For example over the set of natural numbers we know the relation “is less than”. A binary (concerning two elements) relation  $R$  over a set  $S$  can be expressed as ordered pairs over  $S \times S$ . Instead of  $(s_0, s_1) \in R$  we often write  $s_0 R s_1$ .

Relations can have different properties, for example:

**Symmetry:**  $\forall s_0, s_1 \in S : s_0 R s_1 \rightarrow s_1 R s_0$

**Reflexivity:**  $\forall s_0 \in S : s_0 R s_0$

**Transitivity:**  $\forall s_0, s_1, s_2 \in S : s_0 R s_1 \wedge s_1 R s_2 \rightarrow s_0 R s_2$

Show that the relation “beats” in the game of rock-paper-scissors is not transitive.

### 3.2 Transitive Closure

For a relation  $R$  we call the smallest relation which contains  $R$  and is transitive the transitive closure  $R^+$  of  $R$ .

For the set  $S : \{a, b, c, d, e, f\}$  and the relation  $X$  over  $S : (a, b), (c, d), (a, c), (e, f)$  give the transitive closure  $X^+$ .

### 3.4 Is Program Order Enough?

Program order is great to specify the behaviour of a sequential block of code. A very informal memory model for sequential code could be "the program should appear as if all statements have been executed in program order". One way to achieve that would be to simply execute every statement when encountered, without doing any optimizations. However, such a compiler would be wasteful. Imagine a code snippet such as

```
int funca() {
    for (int i=0; i<9999; i++) {
        b=3;
    }
    return b;
}
```

How could a compiler optimize funca() so that it still behaves as intended to an "observer" who is simply calling the function and using the return value?

Note that it is important to be clear which effect of the program are observable. Here we only look at the return value. Probably printed values should also not change. But if the code uses shared variables then suddenly another thread could observe how these change their values! We see that simply relying on program order is not enough, either we disallow all modifications (bad), or we additionally need a set of observable actions and must ensure that these actions "look like" everything was done in program order.

### 3.5 Happens Before

When reasoning about the behaviour of parallel code, it is often useful to define a "happens before" relation as well. For the piece of code below (executed by two threads) look at each given output and draw arrows which indicate in which order statements must have been executed to produce this output. Your "happens before" arrows must contain the program order relation for each threads code. Initially the shared variables x and y are 0.

Thread 1	Thread 2
x = 1;	y = 1;
r1 = y;	r2 = x;

Output 1: r1=0, r2=1.  
Output 2: r1=1, r2=1.

## 4 Javas Memory Model

# Plan für heute

- Organisation
- Nachbesprechung Exercise 7
- Theory
- Intro Exercise 8
- **Exam Questions**
- Kahoot

# Old Exam Task (FS 2023)

5. (a) Erklären Sie den Begriff “Deadlock” im Kontext von gegenseitigem Ausschluss mehrerer Threads. *Explain the term “deadlock” in the context of mutual exclusion in a multi-threaded environment.* (2)
- (b) Was ist der Unterschied zwischen einem “Deadlock” und einem “Livelock”? *What is the difference between a deadlock and a livelock?* (2)



# Old Exam Task (FS 2023)

5. (a) Erklären Sie den Begriff “Deadlock” im Kontext von gegenseitigem Ausschluss mehrerer Threads. *Explain the term “deadlock” in the context of mutual exclusion in a multi-threaded environment.* (2)

**Solution:** A deadlock occurs when no progress can happen in a multi-threaded environment because threads wait for each other's actions.

For mentioning no change in state or the idea thereof with other words (1pt). For mentioning the idea of waiting on each other/circular wait (1pt). If an example is provided but no definition is given (1 pt).

- (b) Was ist der Unterschied zwischen einem “Deadlock” und einem “Livelock”? *What is the difference between a deadlock and a livelock?* (2)

**Solution:** In a deadlock the state of the system does not change. In a livelock, the state of the system changes continuously but without progress being made. (1+1pts)

```

1 public class Main {
2     public static Thread CreateThread(int start) {
3         return new Thread(new Runnable() {
4
5             @Override
6             public void run() {
7                 for (int i = start; i < 7; i+=2) {
8                     System.out.println("Number " + i);
9                 }
10            }
11        });
12    }
13
14    public static void main(String[] args) throws InterruptedException {
15        CreateThread(1).start();
16        CreateThread(2).start();
17    }
18 }

```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

*Mark all the print sequences that can be produced by running the program shown above.*

☐

```

1 Number 1
2 Number 2
3 Number 3
4 Number 4
5 Number 5
6 Number 6

```

☐

```

1 Number 1
2 Number 6
3 Number 3
4 Number 4
5 Number 5
6 Number 2

```

☐

```

1 Number 6
2 Number 5
3 Number 4
4 Number 3
5 Number 2
6 Number 1

```

☐

```

1 Number 2
2 Number 4
3 Number 6
4 Number 1
5 Number 3
6 Number 5

```

```

1 public class Main {
2     public static Thread CreateThread(int start) {
3         return new Thread(new Runnable() {
4
5             @Override
6             public void run() {
7                 for (int i = start; i < 7; i+=2) {
8                     System.out.println("Number " + i);
9                 }
10            }
11        });
12    }
13
14    public static void main(String[] args) throws InterruptedException {
15        CreateThread(1).start();
16        CreateThread(2).start();
17    }
18 }

```

Markieren Sie alle Ausgaben, welche durch den Codeausschnitt ausgegeben werden können.

*Mark all the print sequences that can be produced by running the program shown above.*

☐

```

1 Number 1
2 Number 2
3 Number 3
4 Number 4
5 Number 5
6 Number 6

```

☐

```

1 Number 1
2 Number 6
3 Number 3
4 Number 4
5 Number 5
6 Number 2

```

☐

```

1 Number 6
2 Number 5
3 Number 4
4 Number 3
5 Number 2
6 Number 1

```

☐

```

1 Number 2
2 Number 4
3 Number 6
4 Number 1
5 Number 3
6 Number 5

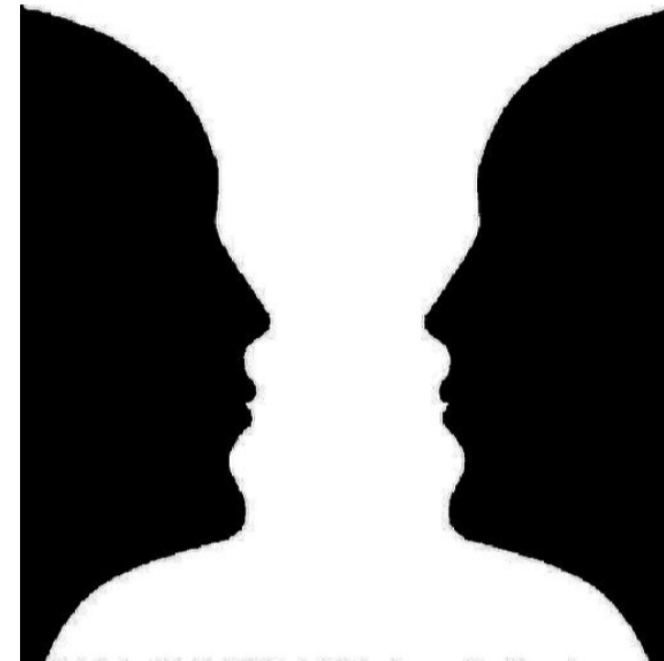
```

## 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.



Negate  
→



*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:*

```
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



```

@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?

*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?* (4)

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();
```

```

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.

```

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

```

*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)

**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);
```

+ Add Comment

... More

- (d) Unter der Annahme, dass die Klasse `ImageNegationFJ` korrekt parallelisiert ist, wie viele Threads verwendet der `ForkJoinPool` effektiv, um das  $2 \times 2$  `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  $2 \times 2$  `negatedImage` array from task 3c)?*

```
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  $2 \times 2$  `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  $2 \times 2$  `negatedImage` array from task 3c)?*

```
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  $4000 \times 4000$  eingeben, bevor Ihnen bei einem RAM der Größe 10 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  $4000 \times 4000$  before you run out of memory using a RAM of size 10 GB? Hint:  $1 \text{ GB} = 2^{10} \text{ MB}$ .* (4)



(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  $4000 \times 4000$  eingeben, bevor Ihnen bei einem RAM der Größe 10 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  $4000 \times 4000$  before you run out of memory using a RAM of size 10 GB? Hint:  $1 \text{ GB} = 2^{10} \text{ MB}$ .* (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  $\text{CUTOFF} = 4000/2^4 = \mathbf{250}$  to avoid a fifth division.

+ Add Comment ... More



# Extra Tasks

## Pipelining

Let us assume that 4 people are at the airport. To prepare for departure, each of them has to first scan their boarding pass (which takes 1 min), and then to do the security check (which takes 10 minutes).

- a) Assume that there is only one machine for scanning the boarding pass and only one security line. Explain why this pipeline is unbalanced. Compute its throughput.
- b) Now assume that there are 2 security lines. Which is the new throughput?
- c) If there were 4 security lines opened, would the pipeline be balanced?

# Pipelining

Let us assume that 4 people are at the airport. To prepare for departure, each of them has to first scan their boarding pass (which takes 1 min), and then to do the security check (which takes 10 minutes).

- a) Assume that there is only one machine for scanning the boarding pass and only one security line. Explain why this pipeline is unbalanced. Compute its throughput.
- b) Now assume that there are 2 security lines. Which is the new throughput?
- c) If there were 4 security lines opened, would the pipeline be balanced?

## Solution

- a) The pipeline is unbalanced, because the latency is not constant. Person 1 has a latency of 11 minutes, whereas person 2 has latency of 20 minutes – i) scan boarding pass (1 min), ii) wait for the first person to finish security check (9 min), iii) pass through security check (10 min). The throughput is 1 person per 10 minutes.
- b) The new throughput is 2 persons per 10 minutes. Note that even though the pipeline is unbalanced, the throughput is constant.
- c) No. For the first 4 people the latency will be constant, but the 5th one would still have to wait. A pipeline is balanced only when the latency is constant for all its inputs.

# Wait and Notify

Consider the following implementation of a FairThreadCounter which implements the Round Robin policy for 2 threads (as described in exercise 3).

rog25-exercises / Repository

```
3    public FairThreadCounter(Counter counter, int id, int numThreads, int numIterations) {
4        super(counter, id, numThreads, numIterations);
5        assert numThreads == 2
6    }
7
8    public void run() {
9        for (int i = 0; i < numIterations; i++) {
10            synchronized (counter) {
11                counter.increment();
12                counter.notify();
13                try {
14                    counter.wait();
15                } catch (InterruptedException e) {
16                    e.printStackTrace();
17                }
18            }
19        }
20    }
21 }
22
23 public static void main(String[] args) {
24     Counter counter = new SequentialCounter();
25     count(counter, 2, ThreadCounterType.FAIR, 10);
26     System.out.println("Counter: " + counter.value());
27 }
```

- a) What will be printed in the console after running the program?
- b) Does the solution behave as expected? If not, explain why and fix the errors.

- a) What will be printed in the console after running the program?
- b) Does the solution behave as expected? If not, explain why and fix the errors.

### **Solution**

- a) Nothing – the program does not terminate, so nothing will be printed. The problem is that in the last iteration, the second thread will be stuck at line 14 waiting to be notified. As the first thread already finished incrementing, the second thread will never be notified.

- a) What will be printed in the console after running the program?
- b) Does the solution behave as expected? If not, explain why and fix the errors.

**Solution**

**b)** There are several mistakes in this solution:

- *Non-deterministic thread order.* Even though both threads increment within a synchronized block, which one starts is not specified. To fix this one should first check whether the thread is supposed to increment (and `wait` if necessary) and increment only after this condition is true.
- *Unhandled spurious wakeups.* After calling `wait` the counter does not check whether it was woken up spuriously or it is indeed its turn to perform the increment.
- *Non-termination.* Even without spurious wake-ups, the program will not terminate as in the last iteration, the second thread will be stuck at line 14 waiting to be notified. A tempting solution is to put an additional `counter.notify()` statement after the for loop. However, this does not solve the problem as there is not guarantee that the notify will be executed after the second thread called `wait`. To fix this issue properly, the code needs to be changed such that it includes an explicit condition that denotes whether the thread should wait or continue. This can be a boolean flag (if there are only two threads) or a thread id as used in exercise 3.

# 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!