Parallel Programming Exercise Session 8

Week 8

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

Lecture Recap

A, B sind shared
r1, r2 sind local

Am Anfang sind alle 0

Was sind möglich
Resultate für r1, r2?

Thread 1	Thread 2
`B = 1;`	`r1 = B;`
`r2 = A;`	`A = 2;`

A, B sind shared
r1, r2 sind local

Am Anfang sind alle 0

Was sind möglich Resultate für (r1, r2)?

Antwort
Alles möglich (0, 0), (1,0), (0,2), (1,2)

Beispiel für data race

Wie ist (1, 2) möglich?

Thread 1	Thread 2
`B = 1;`	`r1 = B;`
`r2 = A;`	`A = 2;`

Data Race

- 2 or more threads in the same process access the same memory location concurrently
- At least one of the accesses is for writing

- Not necessarily a bug

Am Anfang p == q und

p.x = 0

r1, r2, etc. sind local

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

- Compiler kann optimieren

- Jetzt r5 = r2!

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

- Compiler kann optimieren

- Jetzt r5 = r2!

- Es könnte sein, dass
- r6.x = 3; Zwischen
- r2 = r1.x; und r4 = r3.x; passiert.

- Dann hätten wir
- r2 = 0, r4 = 3 und r5 = 0!

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

Java Memory Model

- Gegeben: Ein Programm und eine Ausführung des Programms.
- Java Memory Model (JMM) definiert ob diese Ausführung legal ist.

Java Memory Model

Gibt gewisse Garantien für ein Programm

Visibility: Definiert wann ein Thread eine Aktion eines anderen Threads sieht. Ohne richtige Synchronisierung werden Änderungen evntl. nicht gesehen (-> Caches).

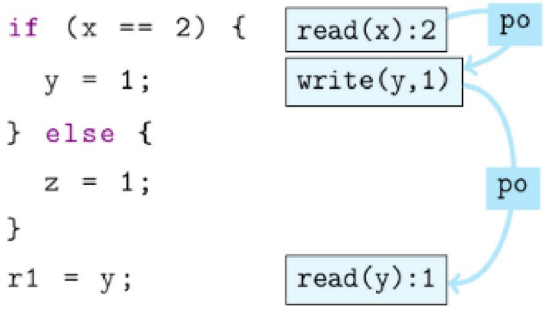
Java Memory Model

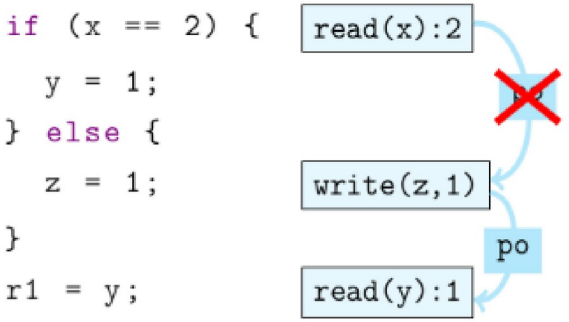
Gibt gewisse Garantien für ein Programm

Atomicity: Garantiert, dass gewisse Vorgänge atomar passieren, z.B. lesen und schreiben von primitive types.

Lecture recap: Program Order

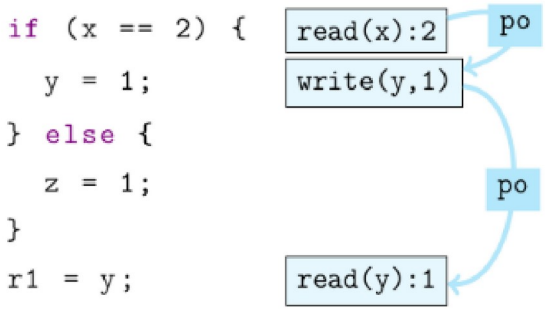
- The code in a program is executed in a certain order
- Executing a line of code means some "action" is happening → we can look at the code and define a partial order of these actions!

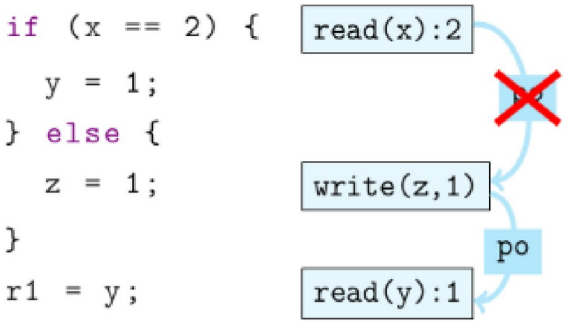




Lecture recap: Program Order

- 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)

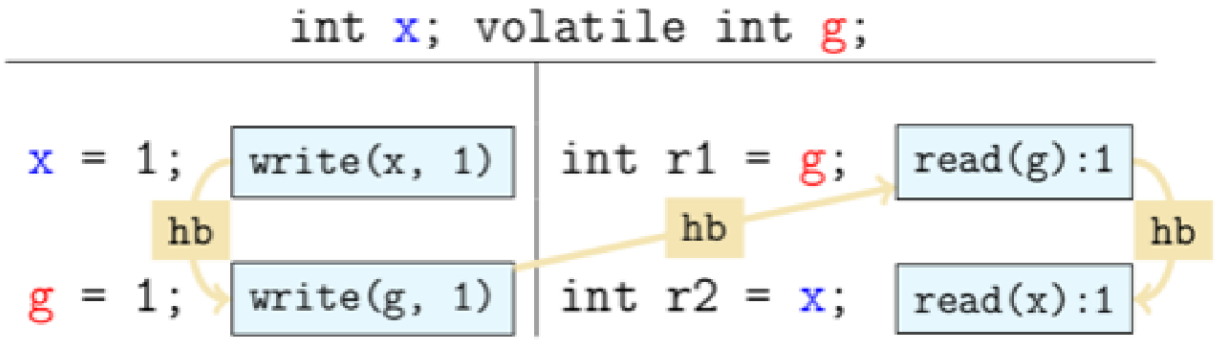
Lecture recap: Synchronizes With

- SAs accross threads synchronize with each other, i.e., a volatile write synchronized with all subsequent reads by any thread

Lecture recap: Happens before

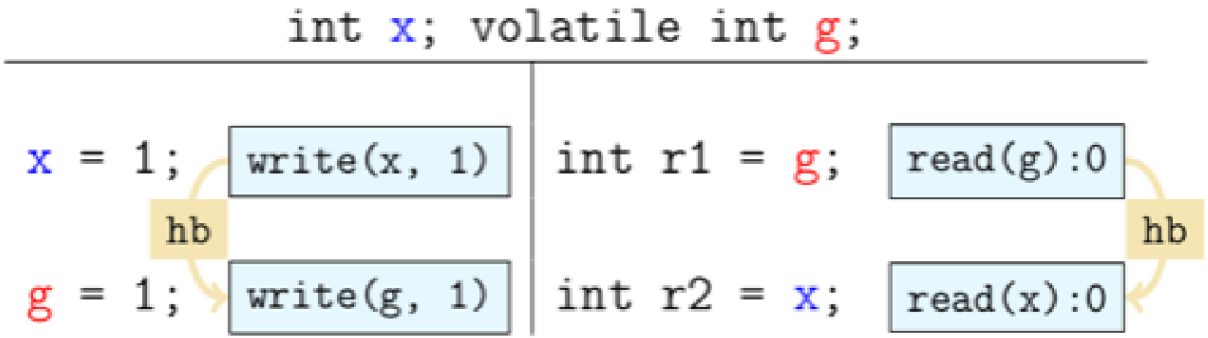
- Transitive closure of PO and SW forms happens before order
- Two actions can be ordered by happens before. If one action happens before another, then the first is visible to and ordered before the second
- All values we observe must obey this happens before order!

Examples



- Initial value of x, g 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!

Volatile

- Read und writes von volatile sind synchronization actions.
- Read und writes von derselben volatile Variable führen synchronizes-with Beziehung ein (die dann in die happens-before Ordnung kommen)

Volatile

Praktische Erklärung:

- Bei volatile write: Alle Variabeln werden von den Caches der CPU zu main-memory geflushed
- Bei volatile read: Alle Variabeln in den Caches werden von main-memory geupdated

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;
```

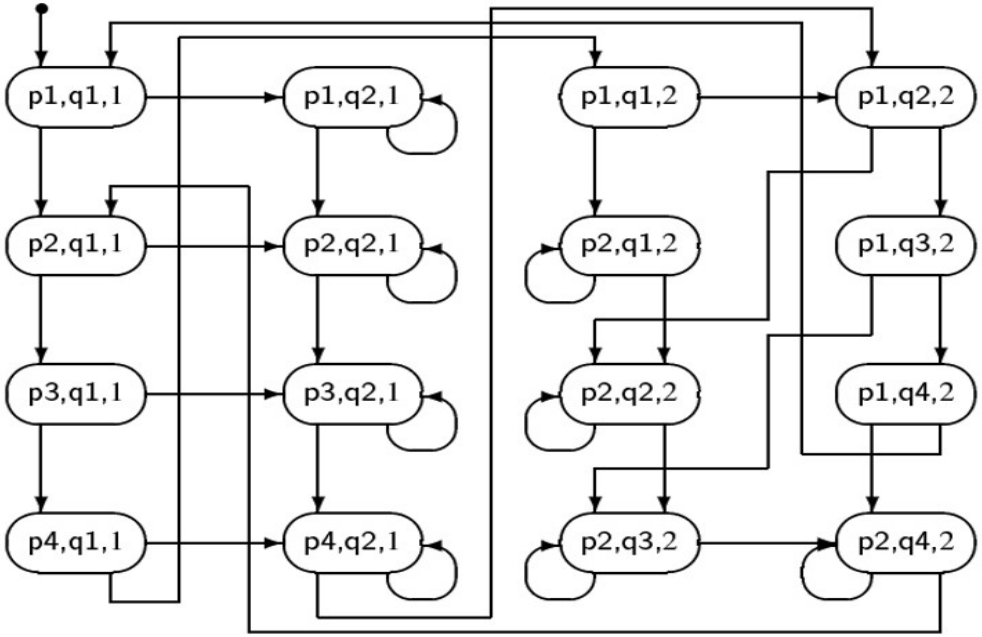
Process P

```
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

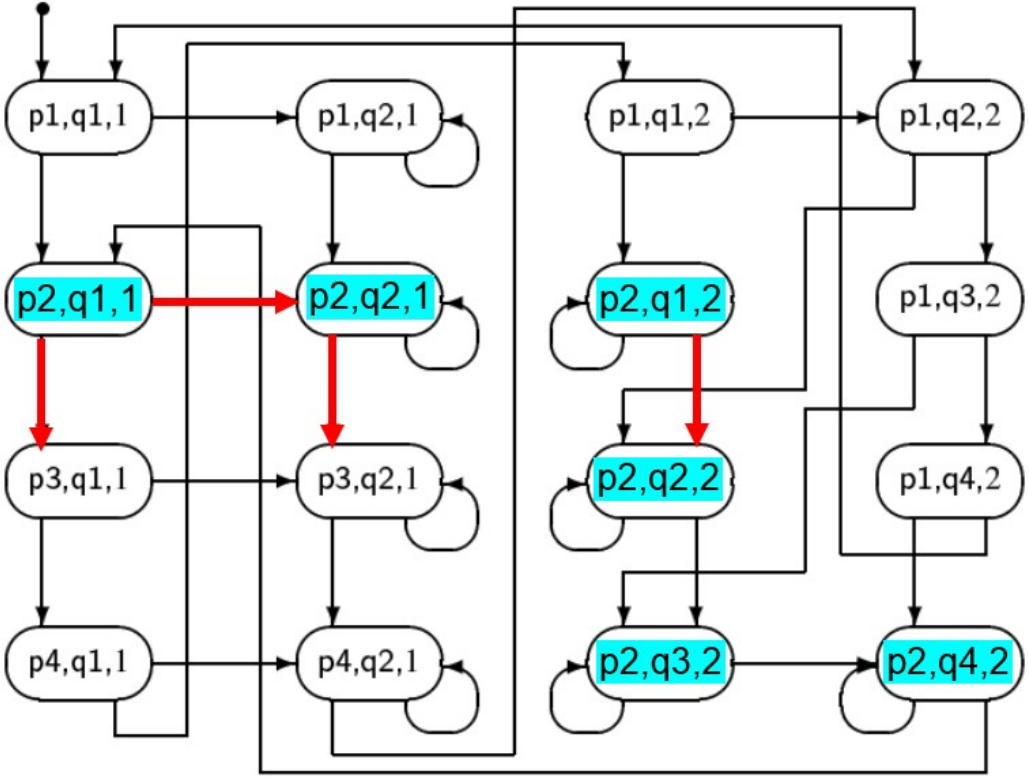
- Since the behaviour of processes P and Q is symmetrical, we only have to check what happens for one of the processes, say P.

- 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

- A deadlocked state has no transitions leading from it

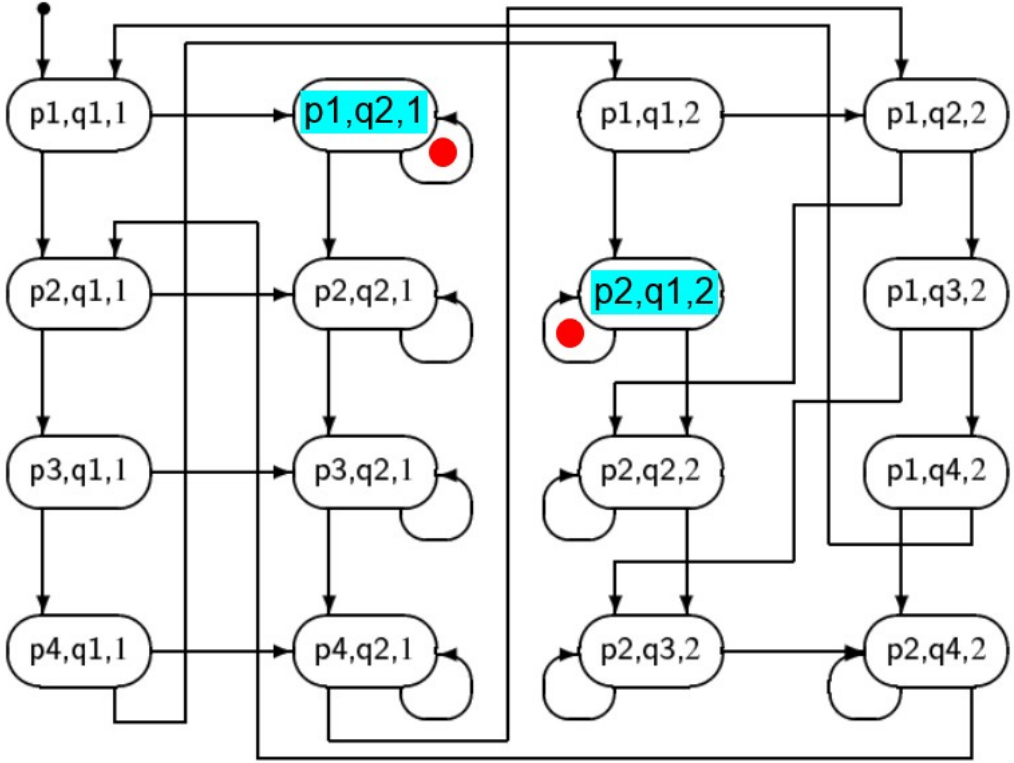
- Sometimes a cycle of transitions may exist from a state for each process, from which no useful progress in the parallel program can be made. The program is still deadlocked but this situation is sometimes termed '*livelock*'. Every one is 'busy doing nothing'.
- A livelock can be detected as cycle in which none of the threads goes through its critical section



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

Stop here if out of time or Atomics not covered yet!
→ Move to slide 32

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();

boolean compareAndSet(boolean expect, boolean update);

boolean getAndSet(boolean newValue);
```

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

sets `newValue` and returns previous value.

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?