Concepts of Concurrent Computation

Spring 2015

Lecture 2: Challenges of Concurrency

Sebastian Nanz

Chris Poskitt

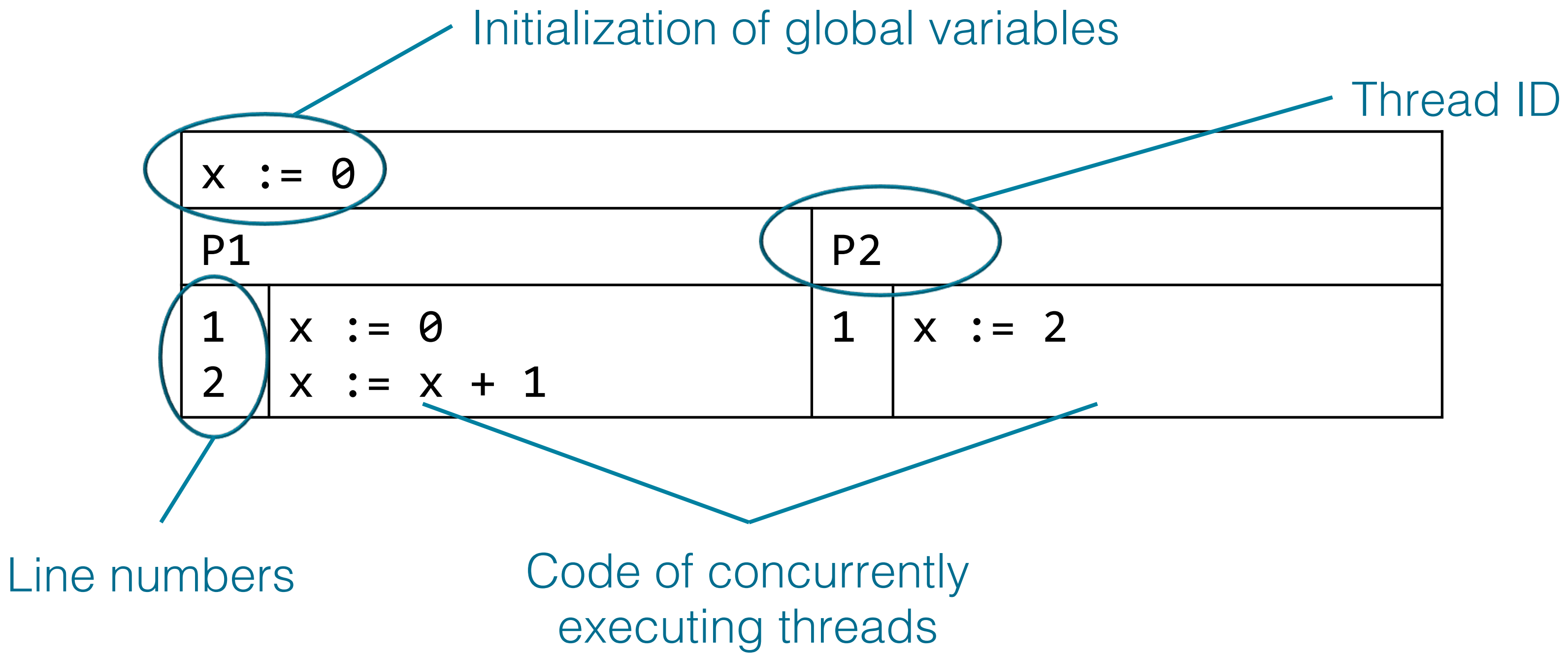
Chair of Software Engineering

ETH zürich

Data races

Abstract notation (Recap)

- A program which at runtime gives rise to a process containing multiple threads is called a concurrent program
- How to specify threads? Every programming language provides a different syntax
- We use an abstract notation for concurrent programs



Initialization of global variables
Thread ID
Line numbers
Code of concurrently executing threads

x := 0			
P1		P2	

1
2
	x := 0
	x := x + 1
		1	x := 2

Execution sequences

x := 0			
P1		P2	

1
2
	x := 0
	x := x + 1
		1	x := 2

- Considering all interleavings, execution can give rise to the following execution sequences:

P2	1	x := 2	x = 2
P1	1	x := 0	x = 0
P1	2	x := x + 1	**x = 1**

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	**x = 3**

P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1
P2	1	x := 2	**x = 2**

- The computation is nondeterministic: depending on the scheduling, `x` may have the final value `1`, `2`, or `3`

Atomicity (1)

- An instruction (or sequence of instructions) is atomic if its execution cannot be interleaved with other instructions before its completion
- Note that the syntax may sometimes be misleading of the actual atomicity of an instruction
- For example, the instruction

  ```
  x := x + 1
  ```

  is on many systems executed as:

  ```
  temp := x          -- LOAD R0, x
  temp := temp + 1   -- ADD R0, #1
  x := temp          -- STORE R0, x
  ```

- Convention: in our notation for concurrent programs, every numbered line can be executed atomically

Atomicity (2)

- To reflect the different assumption on atomicity, the concurrent program is restated:

`x := 0`			
P1		P2	

1
2
3
4
	`x := 0`
	`temp := x`
	`temp := temp + 1`
	`x := temp`
		1	`x := 2`

- One of the possible execution sequences:

P1	1	`x := 0`	`x = 0`
P1	2	`temp := x`	`x = 0, temp = 0`
P2	1	`x := 2`	`x = 2, temp = 0`
P1	3	`temp := temp + 1`	`x = 2, temp = 1`
P1	4	`x := temp`	`x = 1, temp = 1`

Race conditions

- If threads are completely independent, concurrency is easy
- Much more often threads interfere with each other, for example by accessing and modifying the same variables
- The situation that the result of a concurrent execution is dependent on the nondeterministic interleaving is called a race condition
- Race condition errors can stay hidden for a long time and are difficult to find by testing (“Heisenbugs”)

Data race

- A data race occurs when two concurrent threads access a shared memory location and when
    - at least one access is a write and
    - the relative ordering of the two accesses is not enforced
- Not every race condition is a data race
    - The term race condition applies also in situations where there is no memory access (races in file system, network access)
- Not every data race is a race condition
    - There can be data races in memory locations that don’t affect the result of the computation (that you care about)
    - Sometimes also called benign data race

Example: Java Threads (Recap)

- How to associate computations with threads in Java?
    - Inherit from **Thread**, or
    - Implement the **Runnable** interface

```java
class Worker implements Runnable {
  private int input;
  private int result;

  public Worker(int i) {
    input = i;
  }
  public void run() {
    // computation
  }
  public int getResult() {
    return result;
  }
}
```

```java
void compute() {
  Worker w1 = new Worker(23);
  Worker w2 = new Worker(42);

  Thread t1 = new Thread(w1);
  Thread t2 = new Thread(w2);

  t1.start();
  t2.start();
}
```

Example: Java Threads

- Often the final results of thread executions need be combined

```
return worker1.getResult() + worker2.getResult();
```

- We have to wait for both threads to be finished

```
t1.start();
t2.start();
t1.join();
t2.join();
return worker1.getResult() + worker2.getResult();
```

- The `join()` method, when invoked on a thread `t` causes the caller to wait until `t` is finished
- This is a first example of thread synchronization

Temporal logic

Interleaving vs. true-concurrency semantics

- To describe concurrent behavior, we need a model
- Interleaving semantics
  - Assumption that parallel behavior can be represented by the set of all non-deterministic interleavings
  - Imposes total ordering on the events
- True-concurrency semantics
  - Assumption that true parallel behaviors exist
  - Imposes only partial ordering
  - → See lecture on Petri nets

Interleaving semantics

- The interleaving semantics provides a good model for concurrent programs, in particular it can describe:
  - Multitasking: the interleaving is performed by the scheduler
  - Multiprocessing: the interleaving is performed by the hardware
- By considering all possible interleavings, we can ensure that a program runs correctly in all possible scenarios
- Downside: The number of possible interleavings grows exponentially in the number of concurrent processes (state space explosion problem)

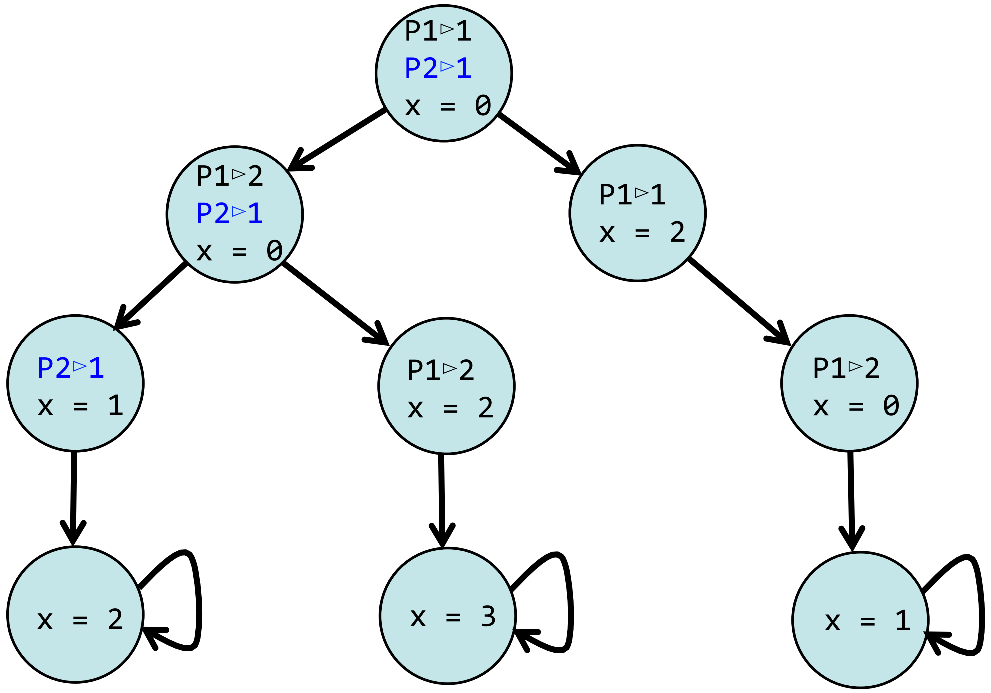
Transition systems

- A formal model that allows us to express concurrent computation are transition systems
- They consist of states and transitions between them
- A state is labeled with atomic propositions, which express concepts such as:
  - $P2 \triangleright 2$ (the program pointer of P2 points to 2)
  - $x = 6$ (the value of variable x is 6)
- There is a transition between two states if one state can be reached from the other by executing an atomic instruction

Example: Transition system

- Transition system for the simple concurrent program

x := 0			
P1		P2	
1	x := 0	1	x := 2
2	x := x + 1		



Transition systems, formally

- Let A be a set of atomic propositions
- A transition system T is a triple (S, $\rightarrow$, L) where
  - S is the set of states
  - $\rightarrow \subseteq S \times S$ is the transition relation
  - $L : S \rightarrow 2^A$ is the labeling function
- The transition relation has the additional property that for every $s \in S$ there is an $s' \in S$ such that $s \rightarrow s'$
  - A transition system with this property is also called Kripke structure
  - Makes it possible to always construct an infinite path
- A path is an infinite sequence of states $\pi = s_1, s_2, s_3, \ldots$ such that for every $i \geq 1$ we have $s_i \rightarrow s_{i+1}$
- We write $\pi[i]$ for the subsequence $s_i, s_{i+1}, s_{i+2}, \ldots$

Temporal logic

- For any concurrent program, its transition system represents all of its behavior
- We are typically interested in specific aspects of this behavior
  - "the value of variable x will never be negative"
  - "the program pointer of P2 will eventually point to 9”
- Temporal logics allow us to express such properties formally
- We will study linear-time temporal logic (LTL)

Syntax of LTL

- The syntax of LTL formulas is given by the following grammar

$$\phi ::= \underbrace{\top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi}_{\text{propositional logic}} \mid G\,\phi \mid F\,\phi \mid \phi\, U\, \phi \mid X\,\phi$$

- Informal semantics of the temporal operators
  - $G\,\phi$ — Globally (in all future states) ϕ holds
  - $F\,\phi$ — in some Future state, ϕ holds
  - $\phi_1\, U\, \phi_2$ — in some future state ϕ_2 holds, and at least Until then, ϕ_1 holds
  - $X\,\phi$ — in the Next state, ϕ holds
- Sometimes we write
  - $\square$ instead of G, and
  - $\diamond$ instead of F

Semantics of LTL

- The meaning of formulas is defined by the satisfaction relation $\models$ for a path $\pi = s_1, s_2, s_3, \ldots$

$\pi \models T$	
$\pi \models p$	iff $p \in L(s_1)$
$\pi \models \neg\phi$	iff $\pi \models \phi$ does not hold
$\pi \models \phi_1 \wedge \phi_2$	iff $\pi \models \phi_1$ and $\pi \models \phi_2$
$\pi \models G\ \phi$	iff for all $i \geq 1$, $\pi[i] \models \phi$
$\pi \models F\ \phi$	iff exists $i \geq 1$, such that $\pi[i] \models \phi$
$\pi \models \phi_1\ U\ \phi_2$	iff exists $i \geq 1$, such that $\pi[i] \models \phi_2$ and for all $1 \leq j < i$, $\pi[j] \models \phi_1$
$\pi \models X\ \phi$	iff $\pi[2] \models \phi$

- If we write $s \models \phi$ we mean that for every path π starting in s we have $\pi \models \phi$

Example: LTL formulas

- “the value of variable x will never be negative”

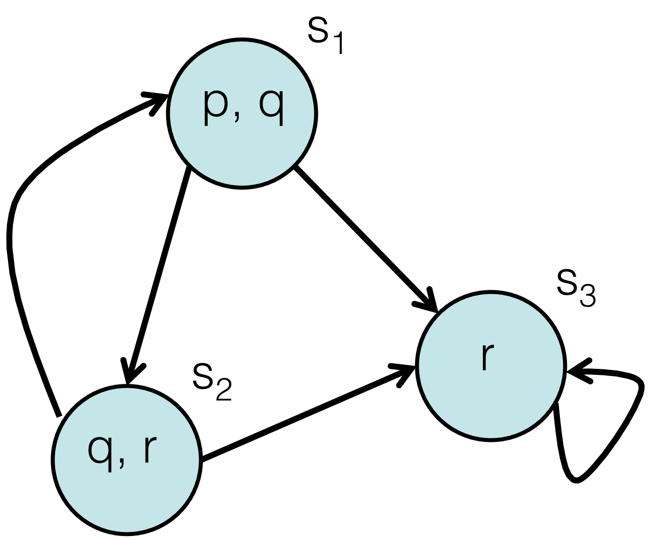
  $G\ \neg(x < 0)$

- “whenever the program pointer of P points to 3, it will eventually point to 9"

  $G\ (P \triangleright 3 \rightarrow F\ P \triangleright 9)$

Example: Semantics of LTL

- Consider the following transition system:



- Which properties hold?
  - $s_1 \models p \wedge q$
  - $s_1 \models \mathbf{G}\ \neg(p \wedge r)$
  - $s_1 \models \mathbf{X}\ q$
  - $s_1 \models \mathbf{F}\ (\neg q \wedge r) \text{ -> } \mathbf{F}\ \mathbf{G}\ r$

Equivalence of formulas

- Two formulas ϕ and ψ are equivalent if for all transition systems and all paths π
  - $\pi \vDash \phi$ iff $\pi \vDash \psi$
- Some equivalent formulas:
  - $F\,\phi \equiv \top\, U\, \phi$
  - $G\,\phi \equiv \neg F \neg \phi$
- Hence both **F** and **G** can be expressed by **U**
- **X** cannot be expressed in terms of the others
- **X** and **U** form an adequate set of temporal connectives, i.e. all temporal operators can be defined in terms of these

Safety and liveness properties (1)

- There are two important types of formal properties in concurrent computation
- Safety properties
  - Informally: “nothing bad ever happens”
  - Properties whose violation always have a finite witness
  - “if something bad happens on an infinite path, then it happens already on some finite prefix of the path”
- Liveness properties
  - Informally: “something good eventually happens”
  - Properties whose violation never have a finite witness
  - “no matter what happens on a finite path, something good could still happen later”

Safety and liveness properties (2)

- For sequential programs
  - Safety: the program will never produce a wrong result (“partial correctness”)
  - Liveness: the program will produce a result (“termination”)
- Liveness properties complement safety properties
  - A safety property can be easily fulfilled by doing nothing
  - Liveness properties allow us to specify some progress

Example: Safety and liveness

- Examples
  - "the value of variable `x` will never be negative" – safety
  - "whenever the program pointer of P points to 3, it will eventually point to 9" – liveness
  - "it cannot happen that the program pointers of both processes points to location `l` simultaneously" – safety
  - "the program is deadlock-free" – safety
  - "the program terminates in 42 steps" – safety
  - "the program eventually terminates" – liveness
  - "P1 can only continue once P2 has arrived at location `l`" – safety
- Many safety properties are invariants, expressible with the LTL formula $\mathbf{G}\ \neg\phi$
- Liveness properties are ofte expressed in LTL by $\mathbf{G}\ (\phi \text{ -> } \mathbf{F}\ \psi)$

Model checking

- Model checking is the problem of automatically checking whether a model of a system meets a given specification
- In our setting
  - the specification is an LTL formula ϕ,
  - the model is a transition system T, and
  - model checking amounts to evaluating $T \models \phi$
- The model is often described with a special-purpose language
- We do not discuss model-checking algorithms in this course
  → See the Software Verification lecture in the Fall
- Exercise class: hands-on experience with the model checker SPIN (properties in LTL + Promela modeling language)

Deadlock and starvation

Process synchronization

- In order to avoid race conditions, processes must synchronize with each other
- Synchronization describes the idea that processes communicate with each other in order to agree on a sequence of actions
- How can processes communicate?

Process communication

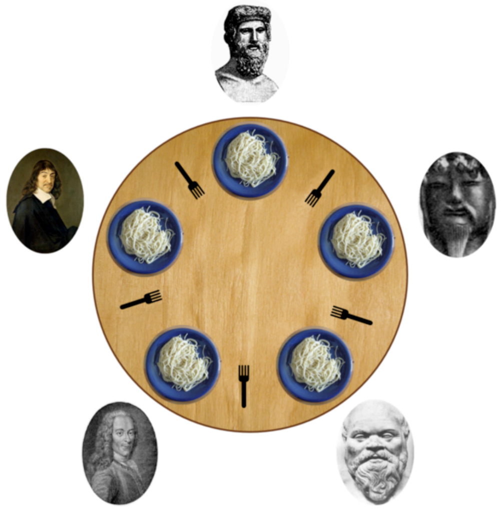
- There are two main means of process communication
- Shared-memory communication
    - Processes communicate by reading and writing to shared sections of memory
    - Used predominantly in shared-memory systems
- Message-passing communication
    - Processes communicate by sending messages to each other
    - Used predominantly in distributed-memory systems

Deadlocks

- The ability to hold resources exclusively is central to providing process synchronization for resource access
- Unfortunately, it brings about other problems
- A deadlock is the situation where a group of processes blocks forever because each of the processes is waiting for resources which are held by another process in the group
- An illustrative example of deadlock is the dining philosophers problem

The dining philosophers problem

- N philosophers are seated around a table and in between each pair of philosophers there is a single fork
- In order to eat, each philosopher needs to pick up both forks which are lying to his sides, and thus philosophers sitting next to each other can never eat at the same tim
- A philosopher only engages in two activities: thinking and eating
- The problem consist of devising an algorithm such that the following properties are ensured
  - Each fork is held by one philosopher at a time
  - Philosophers don't deadlock



Dining philosophers: solution attempt 1

- Each philosopher first picks up the right fork, then the left fork, and then starts eating; after having eaten, the philosopher puts down the left fork, then the right one
- The philosophers can deadlock

The Coffman conditions

- There are a number of necessary conditions for deadlock to occur
    1. Mutual exclusion: processes have exclusive control of the resources they require
    2. Hold and wait: processes already holding resources may request new resources
    3. No preemption: resources cannot be forcibly removed from a process holding it
    4. Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds
- Attempts at avoiding deadlocks typically try to break one of these conditions

Dining philosophers: solution attempt 2

- Each philosopher picks up right fork and the left fork at the same time, and then starts eating; after having eaten, the philosopher puts them both back down
- This solution is indeed deadlock-free: the hold-and-wait condition is broken, as both forks are picked up atomically
- However, a philosopher could starve if always one of the philosophers to his left or right is eating

Starvation

- Even if no deadlock occurs, it may still happen that some processes are not treated fairly
- The situation that processes are perpetually denied access to resources is called starvation

Fairness

- To prove freedom from starvation or other liveness properties, fairness assumptions are sometimes needed
- Fairness is concerned with the resolution of nondeterminism
- Weak fairness*:* if an action is continuously enabled, i.e. never temporarily disabled, then it has to be executed infinitely often
- Strong fairness*:* if an action is infinitely often enabled, but not necessarily always, then it has to be executed infinitely often

Dining philosophers: solution attempt 3

- The forks are numbered 1 to 5. A philosopher always has to pick up his lower-numbered fork first
- This solution is also deadlock-free as the circular-wait condition is broken
- Defining a partial order on resources and respecting that order is a common measure to avoid deadlocks

Dining philosophers: solution attempt 4

- In order to pick up forks, a philosopher must ask a waiter. The waiter only gives permission to one philosopher at a time
- Another deadlock-free solution, which is however undesirable as the amount of concurrency is reduced (it sequentializes the eating)