ETH Zürich
Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 2: Challenges of Concurrency

Today's lecture

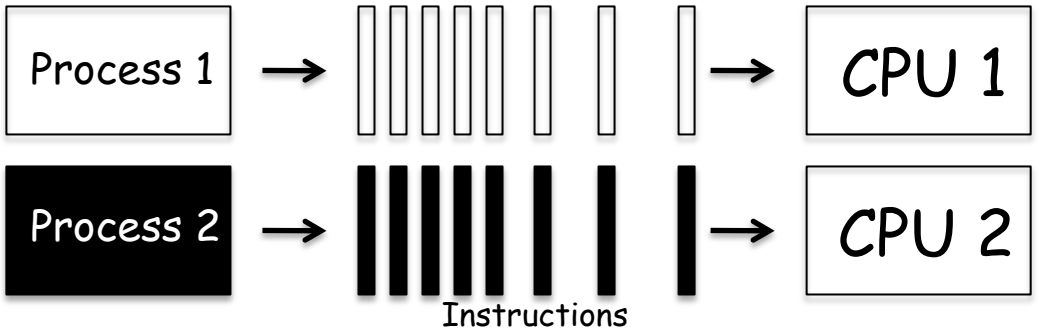
In this lecture you will learn about:

- the basics of concurrent execution of processes in operating systems (multiprocessing, multitasking)
- the interleaving semantics of concurrent computation, and a formalization (transition systems, temporal logic)
- the most important problems related to concurrent programming (race conditions, deadlock, starvation)

Concurrent Processes

Multiprocessing

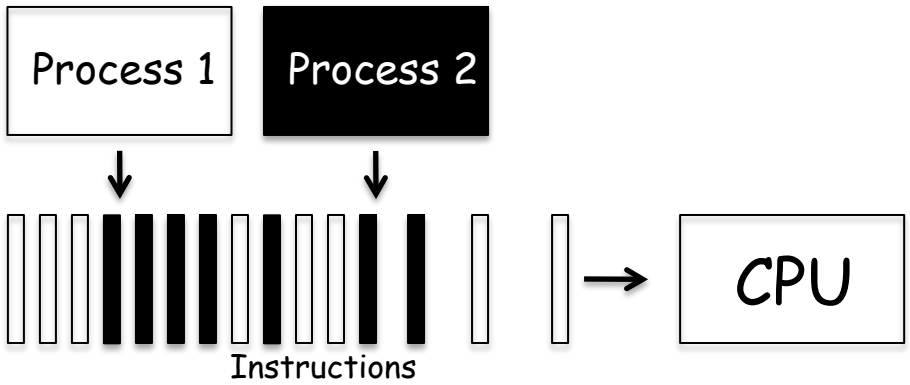
- Until a few years ago: systems with one processing unit standard
- Today: most end-user systems have multiple processing units in the form of multi-core processors



- *Multiprocessing*: the use of more than one processing unit in a system
- Execution of processes is said to be *parallel*, as they are running at the same time

Multitasking

- Also on systems with a single processing unit it appears that programs run "in parallel"
- This is because the operating system implements *multitasking*: the operating system switches between the execution of different tasks



- Execution of processes is said to be *interleaved*, as all are in progress, but only one is running at a time

Processes

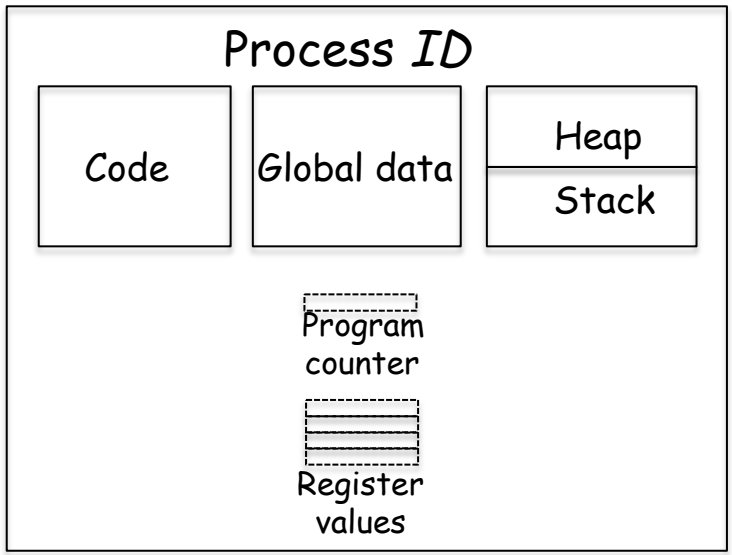
- A (sequential) *program* is a set of instructions
- A *process* is an instance of a program that is being executed

Concurrency

- Both multiprocessing and multitasking are examples of concurrent computation
- The execution of processes is said to be *concurrent* if it is either parallel or interleaved

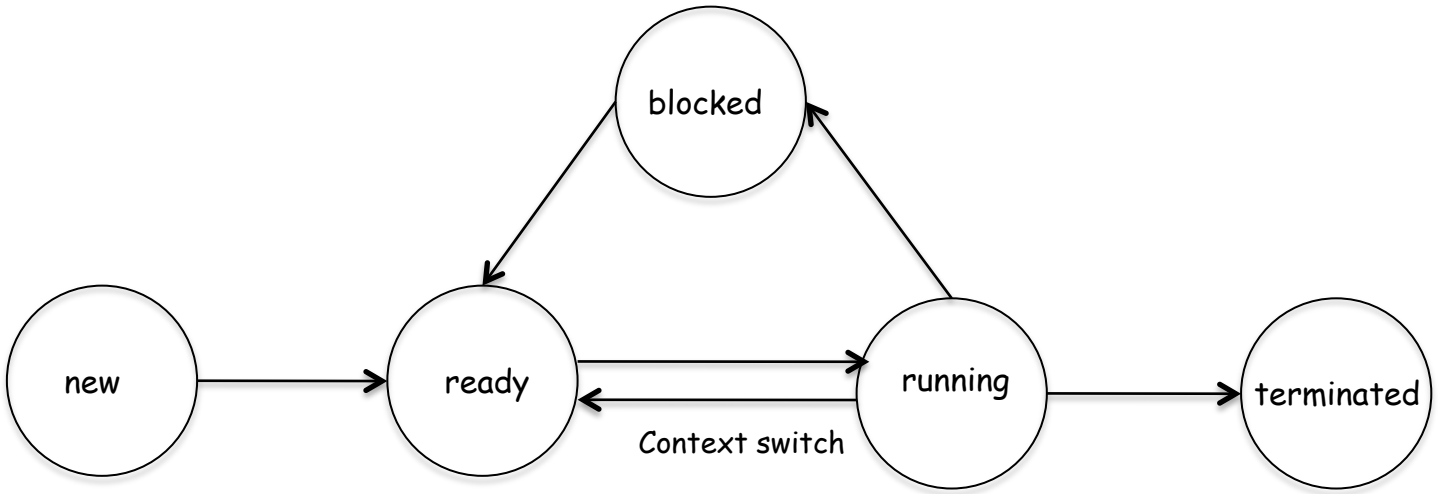
Operating system processes

- How are processes implemented in an operating system?
- Structure of a typical process:
  - *Process identifier:* unique ID of a process.
  - *Process state:* current activity of a process.
  - *Process context*: program counter, register values
  - *Memory*: program text, global data, stack, and heap.



The scheduler

- A system program called the *scheduler* controls which processes are running; it sets the process states:
  - *new:* being created.
  - *running*: instructions are being executed.
  - *blocked:* currently waiting for an event.
  - *ready:* ready to be executed, but not been assigned a processor yet.
  - *terminated*: finished executing.

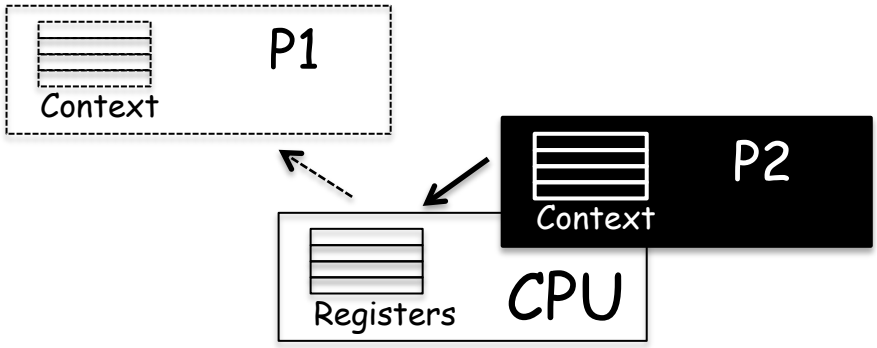


Blocked processes

- A process can get into state *blocked* by executing special program instructions (synchronization primitives, see following lectures)
- When blocked, a process cannot be selected for execution
- A process gets unblocked by external events which set its state to *ready* again

The context switch

- The swapping of processes on a processing unit by the scheduler is called the *context switch*



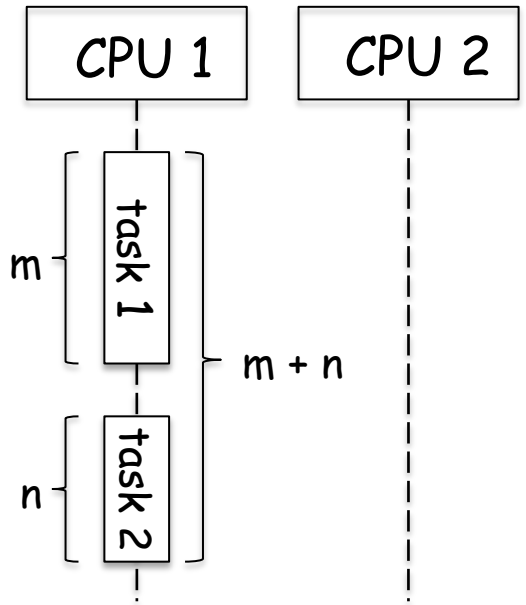
- Scheduler actions when switching processes P1 and P2:
  - P1.state := ready
  - Save register values as P1's context in memory
  - Use context of P2 to set register values
  - P2.state := running

Concurrency within programs

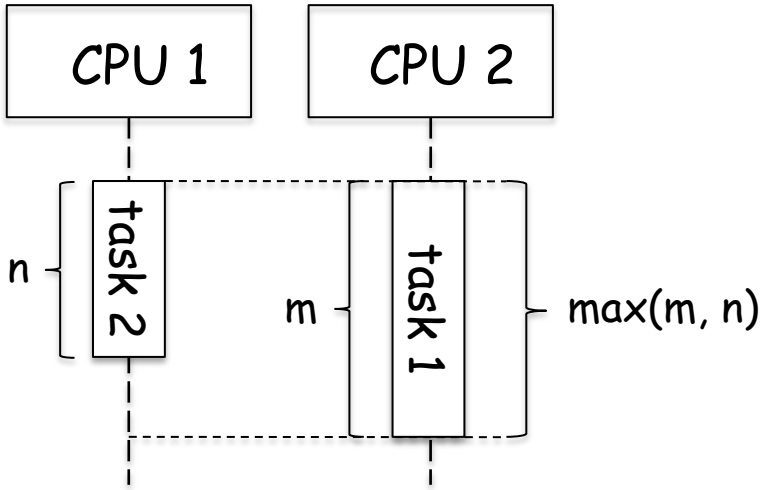
- We also want to use concurrency within programs

```
compute
    do
        t1.do_task1
        t2.do_task2
    end
```

Sequential execution:

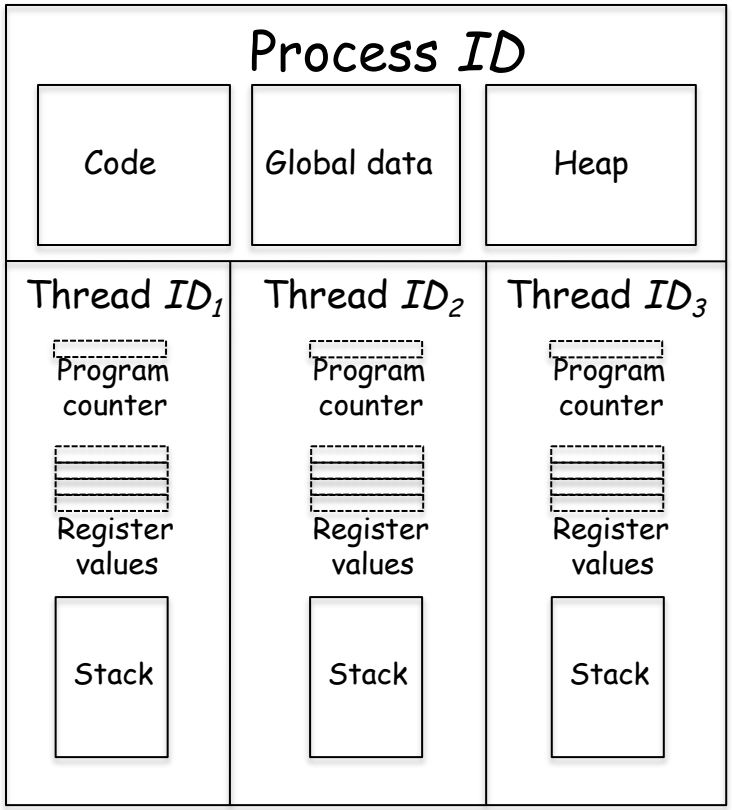


Concurrent execution:



Threads

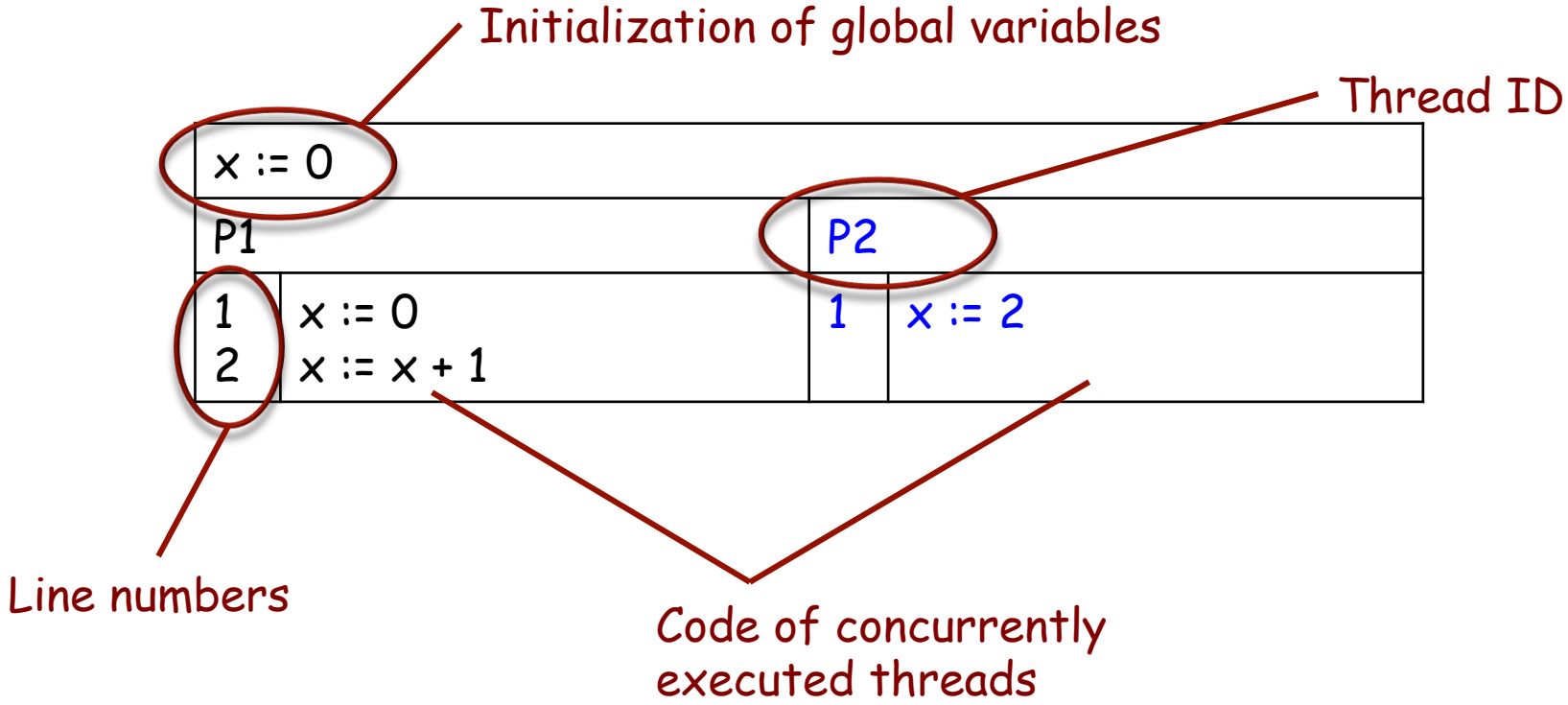
- Make programs concurrent by associating them with threads
- *A thread* is a part of an operating system process
- Components private to each thread
  - Thread identifier
  - Thread state
  - Thread context
  - Memory: only stack
- Components shared with other threads:
  - Program text
  - Global data
  - Heap



The interleaving semantics

Concurrent programs

- 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 different syntax.
- Abstract notation for a concurrent program:



Execution sequences

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

- Execution can give rise to this *execution sequence*:

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

Instruction executed with Thread ID and line number

Variable values after execution of the code on the line

Execution sequences

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

- Execution can also 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
P1	2	x := x + 1	x = 1
P2	1	x := 2	x = 2

- But not to the following (Why?):

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

Atomic instructions

- An instruction is *atomic* if its execution cannot be interleaved with other instructions before its completion
- We can choose different levels of atomicity

The choice is important: 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

Choice of atomicity level

- 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

Side remark: Concurrent programs in Java

- How to associate computations with threads? Example: Java Threads

```
class Thread1 extends Thread {
    public void run() {
        // implement task1 here
    }
}
class Thread2 extends Thread {
    public void run() {
        // implement task2 here
    }
}
```

```
void compute() {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    t1.start();
    t2.start();
}
```

Joining threads in Java

- Often the final results of thread executions need be combined

```
return t1.getResult() + t2.getResult();
```

- We have to wait for both threads to be finished: we have to *join* the threads

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

- Example: the *join()* method, when invoked on a thread *t* causes the caller to wait until *t* is finished

True-concurrency vs. interleaving semantics

- To describe concurrent behavior, we need a *model*
- *True-concurrency semantics:* assumption that true parallel behaviors exist
- *Interleaving semantics:* assumption that all parallel behavior can be represented by the set of all non-deterministic interleavings of atomic instructions

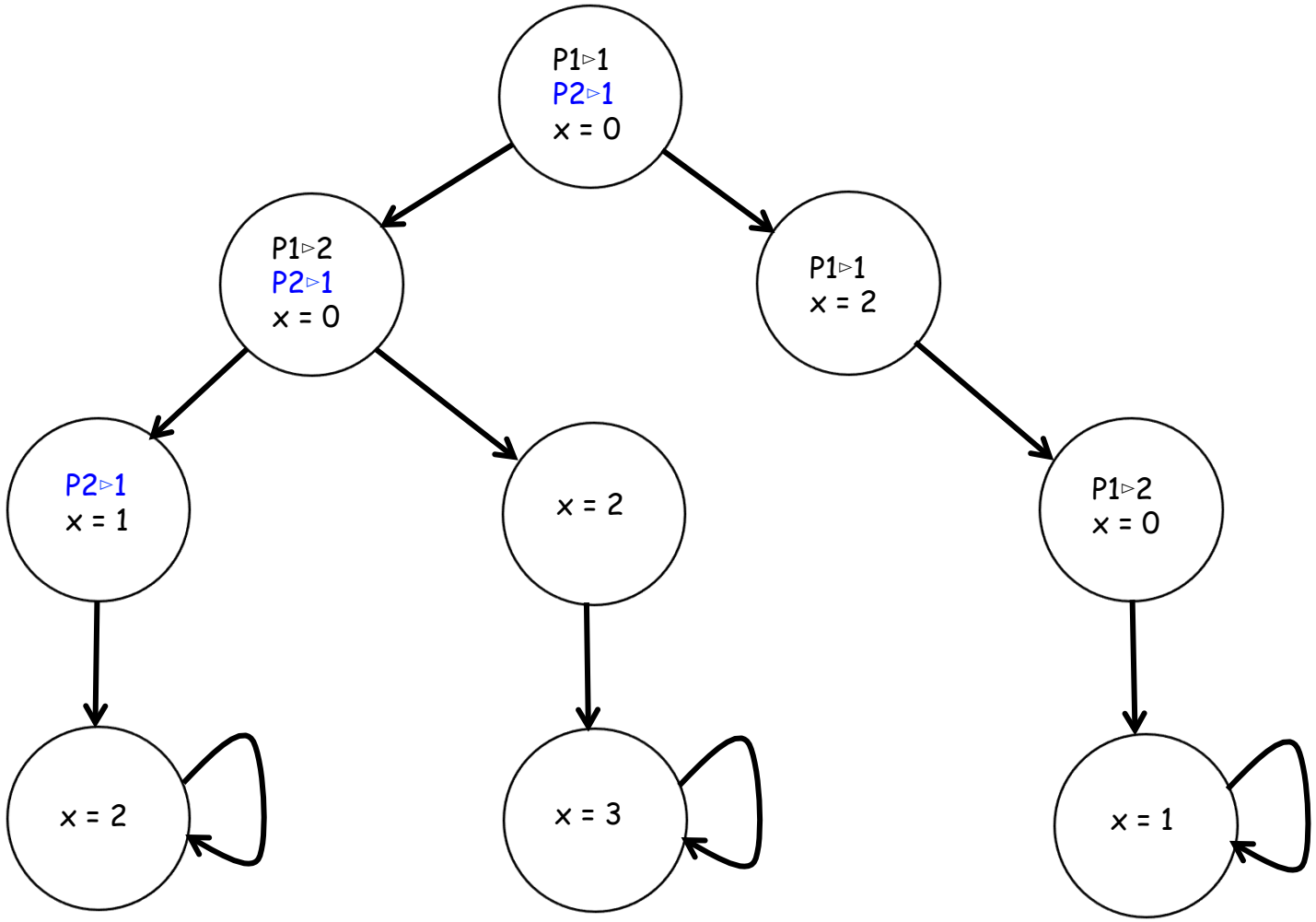
Interleaving semantics

- 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▷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



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

Transition systems, formally

- Let *A* be a set of atomic propositions
- *A transition system* T is a triple (S, ->, L) where
  - S is the *set of states*
  - -> $\subseteq$ S x S is the *transition relation*
  - *L : S -> 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* 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, e.g.
  - "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:

  $\phi ::= T \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{G}\,\phi \mid \mathbf{F}\,\phi \mid \phi\ \mathbf{U}\ \phi \mid \mathbf{X}\,\phi$

- Formulas known from propositional logic:

  $T \mid p \mid \neg\phi \mid \phi \wedge \phi$

- Temporal operators:
  - $\mathbf{G}\,\phi$: *Globally* (in all future states) ϕ holds
  - $\mathbf{F}\,\phi$: in some *Future* state ϕ holds
  - $\phi_1\ \mathbf{U}\ \phi_2$: in some future state ϕ_2 holds, and at least *Until* then, ϕ_1 holds
  - $\mathbf{X}\,\phi$: in the next state, ϕ holds
- Sometimes we write $\Box$ instead of **G**, and $\Diamond$ instead of **F**

Example: LTL formulas

- "the value of variable x will never be negative"

  G ¬(x < 0)

- "whenever the program pointer of P2 points to 3, it will eventually point to 9"

  G (P2▷3 -> **F** P2▷9)

Semantics of LTL

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

$\pi \models \top$

$\pi \models p$ 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 $\pi \models \phi_2$

$\pi \models \mathbf{G}\,\phi$ iff for all $i \geq 1$, $\pi[i] \models \phi$

$\pi \models \mathbf{F}\,\phi$ iff exists $i \geq 1$, such that $\pi[i] \models \phi$

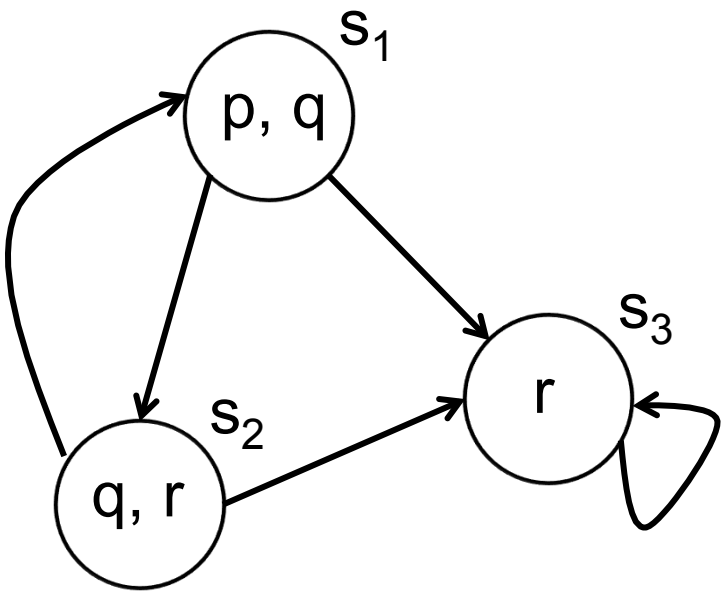
$\pi \models \phi_1\ \mathbf{U}\ \phi_2$ iff exists $i \geq 1$, such that $\pi[i] \models \phi_2$ and for all $1 \leq j < i$, $\pi[j] \models \phi_1$

$\pi \models \mathbf{X}\,\phi$ 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: semantics of LTL

- Consider the following transition system:



- Which properties hold?
  - $s_1 \models p \wedge q$
  - $s_1 \models \mathbf{G}\ \neg(p \wedge r)$
  - $s_1 \models \mathbf{X}\ q$
  - for all s, $s \models \mathbf{F}\ (\neg q \wedge r) \rightarrow \mathbf{F}\ \mathbf{G}\ r$

Equivalence of formulas

- Two formulas φ and ψ are equivalent if for all transition systems and all paths π

  $\pi \models \varphi \quad \text{iff} \quad \pi \models \psi$

- Some equivalent formulas:
  - $\mathbf{F}\,\varphi \equiv \top\,\mathbf{U}\,\varphi$
  - $\mathbf{G}\,\varphi \equiv \neg\,\mathbf{F}\,\neg\varphi$
- 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

- There are two types of formal properties in asynchronous computation:
  - *Safety properties*: properties of the form "nothing bad ever happens"
  - *Liveness properties*: properties of the form "something good eventually happens"
- Example (safety): "the value of variable x will never be negative"
- Example (liveness): "whenever the program pointer of P2 points to 3, it will eventually point to 9"
- Safety properties are often expressible with the LTL formula **G** ¬ϕ, and liveness properties with **G** (ϕ -> **F** ψ)

Concurrency Challenges

Race conditions

- Which value does the function f return?

```
f : INTEGER
    do
        x := 0
        x := x + 1
        result := x
    end
```

Where the trouble starts

- If processes/threads are completely independent, concurrency is easy
- Much more often threads *interfere* with each other, for example by accessing and modifying the same variables or objects
- The function f could execute concurrently with other instructions, e.g. x := 2: we end up with three different results, depending on the particular interleaving taken

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

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

Race conditions

- The situation that the result of a concurrent execution is dependent on the nondeterministic interleaving is called a *race condition* or *data race*
- Such errors can stay hidden for a long time and are difficult to find by testing

Synchronization

- In order to solve the problem of data races, processes have to *synchronize* with each other
- *Synchronization* describes the idea that processes communicate with each other in order to agree on a sequence of actions
- In the above example, it could be agreed that only one process at a time can *hold* the resource (have exclusive use of it); we will see techniques for this in later lectures
- How can processes communicate?

Communication

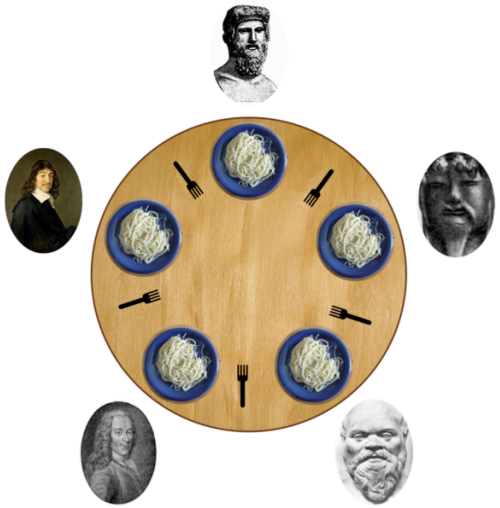
- Two main means of process communication:
  - *Shared memory:* processes communicate by reading and writing to shared sections of memory
  - *Message-passing:* processes communicate by sending messages to each other
- The predominant technique is shared memory communication and we will concentrate on this

The problem of deadlock

- 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 philosopher's 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 time.
- A philosopher only engages in two activities: thinking and eating.
- The problem consist in 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
- Philosophers can deadlock! How?

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
- What is the problem?

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 make solution attempt 2 work, we have to ensure that philosophers are scheduled in a fair way
- Fairness is concerned with a fair 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 activity is *infinitely often enabled*, but not necessarily always, then it has to be executed infinitely often