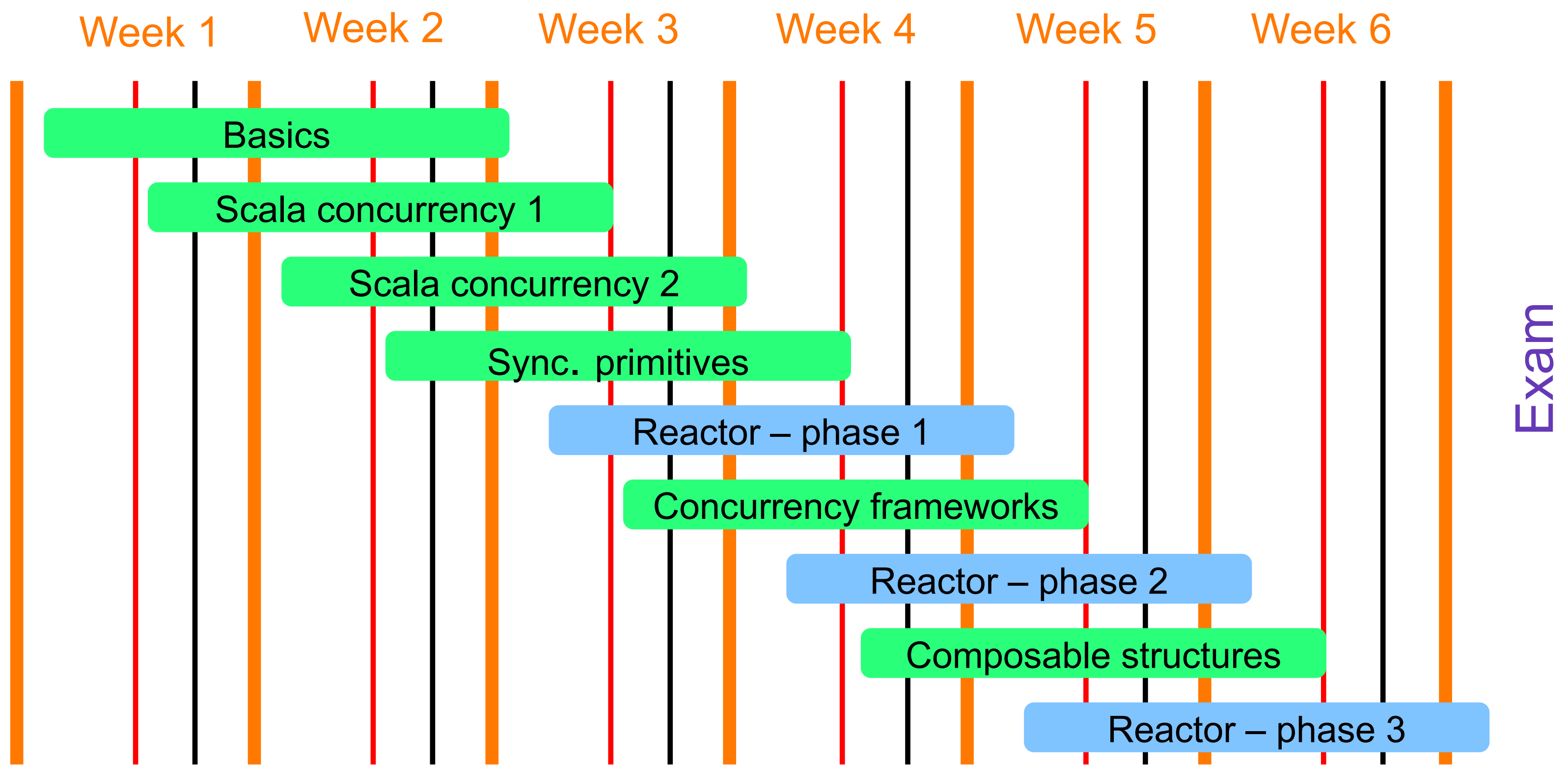
Aalto University
School of Science

An introduction to the course

CS-E4110 Concurrent Programming
Autumn 2020 (II), week 1

Vesa Hirvisalo

The course in a nutshell



General session

Exercise session

🟩 Exercise with automatic grading

🟦 Exercise with manual grading

Week 1 in a nutshell

- Introduction to concurrent programming
  - Where and why it is needed
  - Why is it regarded as “difficult”
- Practical arrangements of the course
  - Do and pass the exercises
    - Learn the theory along the exercises
  - Take and pass the exam
    - Your knowledge on the theory will be tested
- Some basics on concurrent programming
  - Processes and threads
  - Synchronization
  - Formalization of execution

An Introduction to

Concurrent Programming

Concurrency

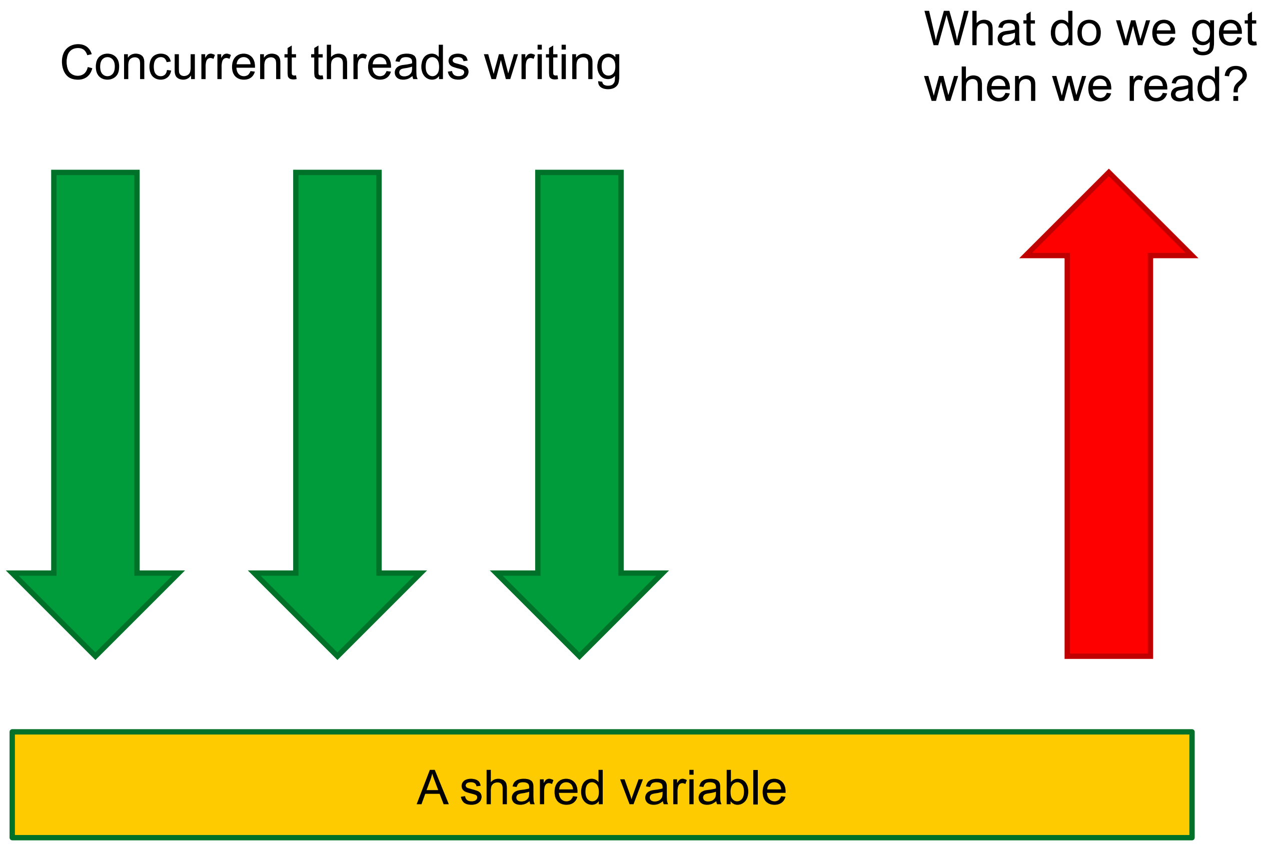
- Note the difference between
  - Parallel and serial execution
  - Concurrent and sequential programs
- Serialization and serial sequential semantics
  - Concurrent programs can also be executed serially
  - Sequential programs can also have parallel execution
- Note difference between
  - Concurrency (correctness)
  - Parallelism (performance)
  - Distribution (clocks)
- Many practical systems have flavors of all three
  - However, usually their problems focus around one

Program execution

- Sequential program execution
  - A sequence of atomic actions producing a *trace* of consecutive states Also called a *scenario* or *computation history*
- Concurrent program execution
  - The *trace* of a concurrent program is *one interleaving* of the traces or its processes/threads into one singular trace
  - Several threads or processes running: physically parallel (=*multiprocessing*)
    - virtually parallel (=*multitasking*)
  - The *sequentially consistent memory model*
  - If n is the number of traces, and m the number of actions per trace, then the number of interleavings is:

$$Interleavings = \frac{(n * m)!}{(m!)^n}$$

The problem


Concurrent threads writing
What do we get when we read?
A shared variable

The problem revisited: Races

- Race condition occurs when multiple processes read or write data items so that the final result depends on the order of execution of the processes
- A very simple program as an example
  - Consider two threads T1, T2 that share a global variable x
  - T1 updates x to the value 1
  - T2 updates x to the value 2
  - Tasks T1 and T2 are in a race considering the write to variable x
  - the “loser” of the race determines the final value the program
- Note that such a race can happen on any resource
  - The accesses or structures can be very complex

    => This makes solving such problems very complex

Mutual exclusions

- Critical section (CS)
  - Critical resource is the resource that is used by competing processes
  - the code that uses the resource is a critical section
- Mutual exclusion
  - no two processes can enter the critical section at the same time
- Synchronization is based on mutual exclusion
  - We force the execution order such the races are resolved
  - The outcome of a concurrent program is deterministic
    - Race-free programs
  - There are a number of mechanisms for this
    - Usually, entry and exit protocols (to/from critical section)

Course arrangements

Overview of arrangements

- Register to the course (Oodi)
  - Register also to course A+ and course Slack
- Follow MyCourses
  - The basic info for all course week will be there
- Practice: There are exercises on A+
  - Nine rounds: 6 automatic and 3 with manual grading
- Theory: Take the course exam
  - Learn the theory along making the exercises
- Advice
  - Asynchronous: Slack (staff checks on daily basis)
  - Synchronous: Zoom Wed. 2-4 pm (theory), Fri. 2-4 pm (practice)
- Course email: cs-e4110 at aalto.fi
  - For formal issues (not advice on exercises, etc.)

Contents and requirements

- Material on the practice
  - Scala and Java documentation (especially Chapter 17 on Lock and Threads, https://docs.oracle.com/javase/specs/jls/se11/html/jls-17.html)
- Material on the theory
  - *Course textbook*: Mordechai Ben-Ari. Principles of Concurrent and Distributed Programming.
- Binding material
  - Weekly slides (on MyCourses) and some tutorials (on A+)
- Requirements
  - Exam
    - Have you understood the concepts?
    - Course registration covers only the main exam (“course examination”), remember to register if you take an other exam
  - Exercises
    - Can you do it with practical examples?

Grading

- Grading is based on the points that you get
  - Knowing basic stuff well => expect 3 as your grade
    - Not knowing well => easily you end up with grade 1 or 2
  - To pass the course (with grade 1) => roughly ½ of points
    - Not recommended for anyone
- Passing both the exercises and the exam is mandatory
  - You must pass 7 out of 9 exercise rounds (max. 700 points)
    - A+ tells the limits per round
    - Rounds 5, 7, 9 are related (and with manual grading)
  - The exam points are scaled so that the max. total is 1400 points
    - i.e., max. 700 points from the exam also
- There are several exam dates available
  - However, the exercises are open only during period II/2020

Course schedule

- The course consists of 6 week that focus on the following topics
  - Week 1: Introduction and Basics (a thread-based view)
  - Week 2: Practical tools (mostly Scala features)
  - Week 3: Synchronization (variety of mechanisms)
  - Week 4: Concurrency frameworks (especially Akka)
  - Week 5: Composable structures (towards parallelism)
  - Week 6: Concurrency models (especially memory model)
- You can see the related exercise schedule from A+

Some basics

The process concept

- *Concurrent* execution of programs cause many problems
  - nondeterministic program operation (=> *races*)
  - *deadlocks* and *livelocks*
  - These of the relate to failed mutual exclusion
- The concept of *process* was developed to thwart these
  - A process is, roughly the same as, a program in execution
- A process consists of
  - an executable program
  - data needed by the program
  - the execution context

On the process abstraction

- A multiprogramming operating system needs to
  - interleave the execution of multiple processes
  - allocate HW resources to processes and prevent other processes accessing resources while not allowed to
  - allow processes to communicate with each other allow processes to synchronize with each other
- The *traditional* solution is to define a process
  - what data structures are needed to represent process in an OS
  - which states characterize the behavior of processes
- Considering the HW *today*, we would not “invent” processes as such…
  - but changing the whole SW world… well, we often do use processes in practical SW

Multitasking

- One physical processor is multiplexed between several process
  - The multiplexing provides for each process a virtual processor by maintaining its state, i.e. its register values in the memory when the process is not running
  - In multitasking the processes are interleaved by giving the processor to each of them in turns.
  - It produces one concrete, fully specified linear interleaving according to our model provided that the *context switch* between processes preserves the atomic actions

(*) Context is the system data associated with a state of a processing (in addition to the data of the program). Note that for a true execution, there are several contexts (e.g., in a runtime system, in an operating system, and in a hardware) and several different ways of switching.

Threads

- As processes are heavy, threads are often used
- Thread do not typically have their on memory
  - Traditionally, they are within a process
  - This makes them lighter
- Threads within a process
  - Share the memory
  - But, typically share also many other the other resources
- Threads have independent control
  - Typically implemented with a control stack
  - Usually, they can be independently suspended and resumed

Mutual exclusion and spinlocks

- Mutual exclusion states that only one process is inside a critical section
- Other processes (threads) must *wait* for the process (the thread) in the critical section to exit it
- We need a method to express this behavior as an LCR expression
  - The simples solution is a so called *busy-wait loop*
  - It is also knowns as a *spinning lock* or *spinlock* for short
- However, busy-wait reserves the computation resources
  - There other ways of *synchronization*
  - The course is much about understanding synchronization

Synchronization

- To implement synchronization
  - You have to have hardware support
  - You have to able able to assume something about hardware operation
- You may be tempted to think hardware issues are easy
  - Often, they are very complicated
  - E.g., pipeline and memory effects
- Machine instructions
  - Often there are specific machine instructions for synchronization
  - Relate usually to coherency and consistency of memory
- Lock-free (or race-free) algorithms
  - Be aware, misnomers are often used
  - They *all make assumptions* on the underlying hardware!

Deadlocks

- Following conditions must be met for a deadlock to be possible
  - mutual exclusion: resource that can be used by one process at a time
  - hold and wait: process can wait other processes while holding resources
  - no preemption: resource cannot be forcibly removed from a process
  - circular wait: resources are waited by a closed chain of processes
- The first three conditions are prerequisite for the fourth condition; if the last one holds, the system is in deadlock
  - whenever first three are met ⇒ possibility of deadlock if the
  - fourth is met ⇒ existence of deadlock

Other problems

- Livelock
  - Processes run, but do not do any useful work
- Starvation
  - A process does not get a resource
- Fairness, immediacy, etc.
  - Actually, the list of “nice” properties is rather long…
- Often connected to scheduling
  - The applications have the restrictions in their code
  - The scheduler makes the selection
    - What path of resource reservations to follow
    - Conflicts with performance (etc.) are common

Programming Tools

Using Scala (Java and JVM)

- If you are not familiar with Scala, it is time to start
  - You will not need all of the language, but the concurrent programming exercises (in A+) are done by using Scala
  - Be careful to understand the notation (some of which may not be so obvious)
  - Note that Scala is based on Java (and uses JVM)
- You can use Aalto course material
- …or, just check the language documentation
  - https://docs.scala-lang.org/

Language support for concurrency

- Common pragmas for controlling compilation
  - A variable V is defined volatile: Whenever V is updated, the change is forced immediately (i.e., cache is flushed)
  - A variable V is defined atomic: The LOAD and STORE instructions for memory item V are performed as atomic actions
- Scala (JVM) has built-in support for concurrency
  - All primitive types except long and double are atomic by default
  - All primitive types become atomic if declared **volatile**
    - Atomicity is not the same as synchronization
    - Often carefully designed *signaling* is needed
    - Note the difference to the **synchronized** keyword
  - An object does not become volatile or atomic if a reference to it is declared so

Waiting for other threads

- Synchronization in practice is waiting for other threads
  - However, do not use “wait” as the wake-up is clock-based
  - Wake-ups initiated by other (running) threads is important
- Mutual exclusion states that only one process is inside a critical section
  - Other processes must wait for that process to exit the CS
  - We need a method to express this behavior as an LCR expression – the simples solution is a *busy-wait loop*
  - It is also knowns as a spinning lock (or *spinlock*)
- Synchronization primitives of the languages
  - Have often hardware support (yields good performance)
  - i.e., the compiler knows the special machine instructions

Practical considerations

- Correctness without losing performance
  - Latency hiding is the key
  - Avoid busy waiting, use the synchronization primitives
- You have to get an understanding of the execution
  - Computing is about manipulating data
  - It is essential to have a working model “what can happen”
  - … and “can” means all possibilities
- Usually, there are three main issues with multithreading:
  1. Race Conditions
  2. Caching / stale memory
  3. Optimizations (compiler, etc.)
- Deoptimizing (volatile, etc.) cannot solve the first problem. Proper synchronization can solve all of them.
  - However, both are tricky (correctness!) and affect performance

Understanding Concurrency

Programs and execution

- Programs are typically just text
  - A typical compiling unit appears as a (long) string
- Semantics
  - Semantics are often semi-formal (i.e., rather loose)
  - Compilers interpret the semantics and emit compiled code
    - Runtime systems go usually with the compilers
    - Often, there is operating systems and virtualization
    - However, there are also linkers, loaders, …, and the HW
- Execution
  - Typically very complex (i.e., “what actually happens”)
- Models of computation
  - Concentration only on the relevant things (wrt. a problem)
  - Agreements enforce the models (e.g., memory consistency)

Consequences of Non-Sequential Control

- The global control flow is non-deterministic
  - When two or more threads run on a machine with a *pre-emptive scheduler**, the operating system will determine when and how much CPU-time they get
  - The threads have no control over this interleaving
  - The scheduling is affected by countless factors: process count, process priorities, CPU usage by other processes, I/O latencies etc...
  - A concurrent program cannot control scheduling, so the programmer must make sure his program is correct, for all the possible interleavings!

(*) Here pre-empt means to give a resource away, for a CPU scheduler it means that the execution of threads can be suspended, and later on, their execution can be resumed. The thread continues as nothing had happened, but the system state may have changed during the suspension.

Our Computational Model

- Computation can be seen as a sequence of steps
  - Each step is a *transition* from one *state* to the next
- The states are more significant than the sequences of steps, as the current state of the computational device determines the next transition
  - The computational model that we use is state based
- Importantly this approach is independent of what drives the state transitions, be it a single thread or multiple threads

States and Transitions

- Describing computations as a sequence of states
  1. The initial state(s)
  2. For every state, all of the possible states reachable from that state by a single atomic action
- Atomic actions
  - A single state transitions
  - The system state cannot be accessed during an atomic action
  - The actions usually correspond to machine instructions on the underlying HW

State of a Concurrent Program

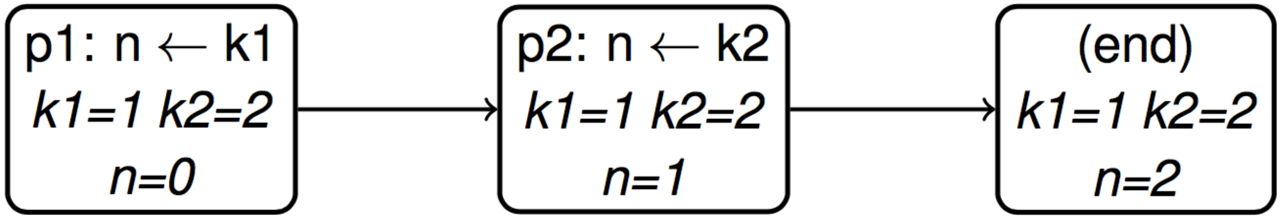
- The Labelled Code Line (LCL) assumption
  - For our pseudo-code notation, each labelled code line is one *atomic action*
  - The labels are marked as "X:" where X is a running number
- The state of a program is an n-tuple
  1. One element for the label of each thread/process. If the process it at termination, the label is substituted by (end)
  2. One element for each global and local variable and the value of that variable

Sequential Example

Trivial sequential program

```
integer n ← 0

    integer k1 ← 1
    integer k2 ← 2
p1:       n ← k1
p2:       n ← k2
```

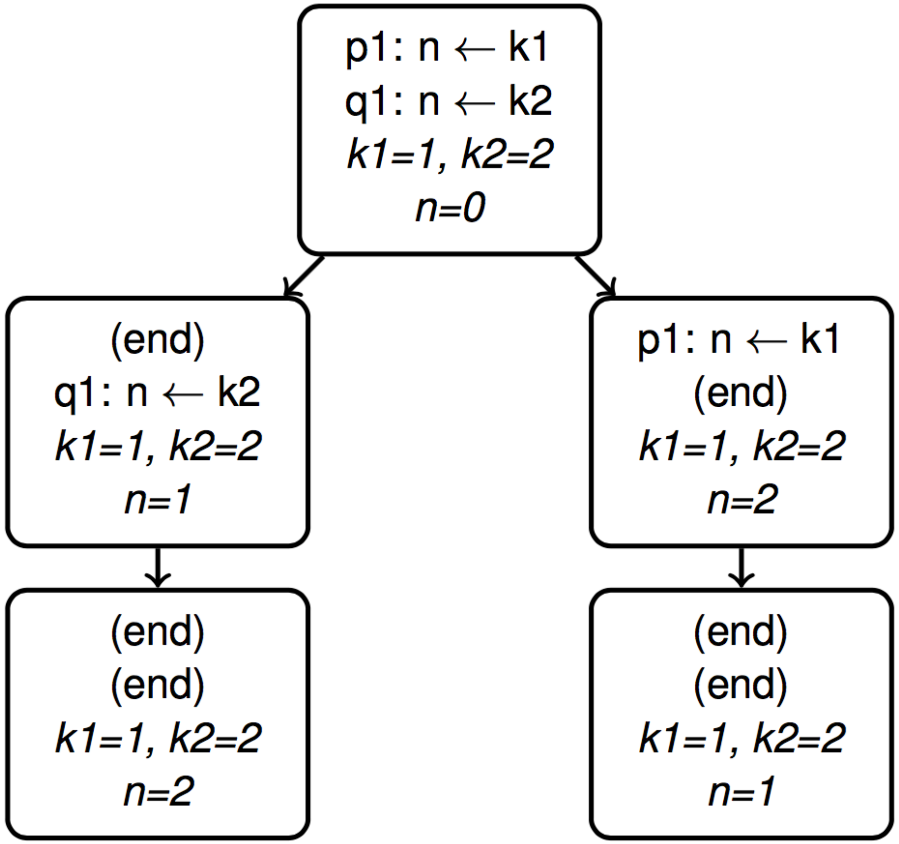


$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{p1: } n \leftarrow k1\\ k1=1\ k2=2\\ n=0\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{p2: } n \leftarrow k2\\ k1=1\ k2=2\\ n=1\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{(end)}\\ k1=1\ k2=2\\ n=2\end{array}}
\end{tikzcd}$$

Concurrent Example

Trivial concurrent program	
integer n ← 0	
P	Q
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

Concurrent Example: States

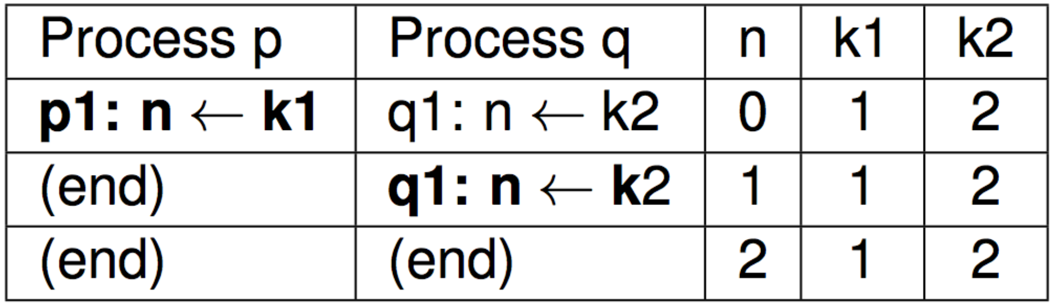


Advice on State Diagrams

- Generally, a state has as many outbound transitions as there are non-terminated processes at that state
- An outbound transition may lead to the state itself
- If a state has no outbound transitions that lead to other states, that state represents a deadlock
- For loops and branching, check carefully if the transitions should lead to an existing state or a new state

Concurrent Example: Trace

- The trace represents one path through the state diagram
  - Each row in the table represents a state in the diagram / concurrent system
  - By convention the bold faced statement in the row indicates the control pointer value and the atomic statement to be executed to get to the next state (= row)

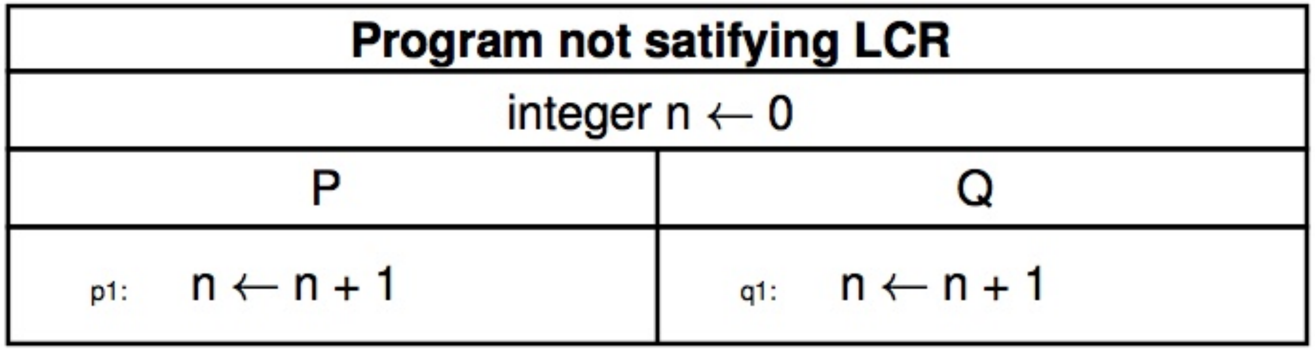


Process p	Process q	n	k1	k2
p1: n $\leftarrow$ k1	q1: n $\leftarrow$ k2	0	1	2
(end)	**q1: n $\leftarrow$ k2**	1	1	2
(end)	(end)	2	1	2

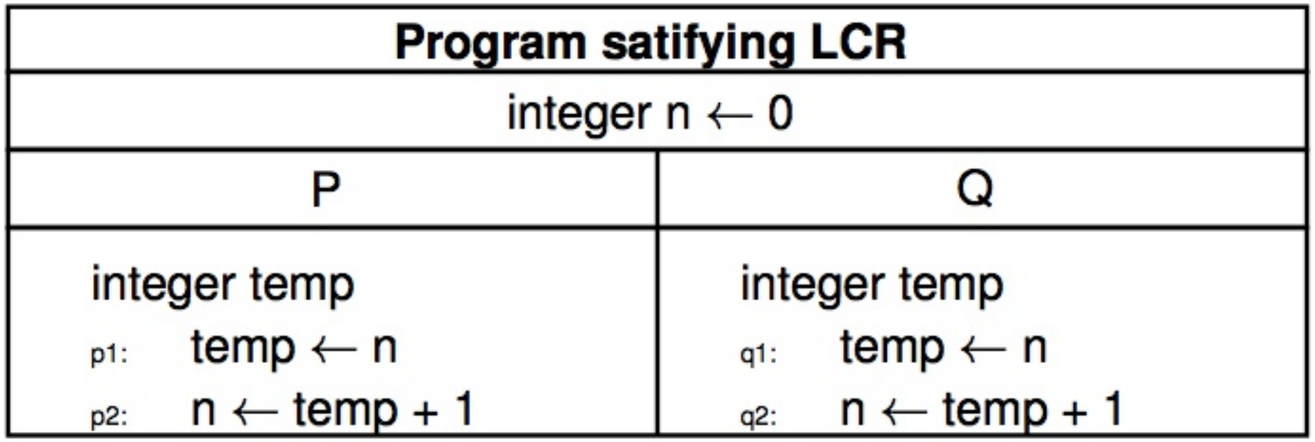
Atomicity and Limited Critical Reference

- For an assignment to be consider atomic, it must satisfy
  1. Any reference to a variable *v* in process *P* is called *critical*, if *v* is written in any other process than *p*. Assignment to a variable *v* in P is critical, if it is referenced in any other process
  2. A statement with at most one critical reference satisfies the *limited critical reference* (LCR) condition
  3. A program consisting only of labelled LCR statements is defined to satisfy LCR
- A LCR-statement has a simple atomic implementation on normal HW
  - Because it refers to the shared state, *i.e.*, shared variables at most once per line
  - It can thus be compiled to a sequence of standard (atomic) LOAD or STORE machine instructions which access at most one shared variable each

Transformation of assignment



Program not satifying LCR	
integer n $\leftarrow$ 0	
P	Q
p1: n $\leftarrow$ n + 1	q1: n $\leftarrow$ n + 1



Program satifying LCR	
integer n $\leftarrow$ 0	
P	Q
integer temp p1: temp $\leftarrow$ n p2: n $\leftarrow$ temp + 1	integer temp q1: temp $\leftarrow$ n q2: n $\leftarrow$ temp + 1

Set your Scala environment up (see A+)

Check the weekly information

=> Section Week 1 on MyCourses