Concurrent Programming

Introduction

Team

- Lecturer: Alejandro Russo
  - Course issues
    - **concurrent-programming-period-3-2012@googlegroups.com**
    - http://groups.google.com/group/concurrent-programming-period-3-2012
    - Other issues: russo@chalmers.se
- Assistants
  - Nick Smallbone: nicsma@chalmers.se
  - Emil Djupfeldt: djupfeld@student.chalmers.se
  - Staffan Björnesjö: staffan.bjornesjo@gmail.com

Introduction

- Why concurrent programming?
  - In general
  - In this course
- Practical course information
- Gentle start
  - Java
  - JR (MPD)

Why?

- Where is John von Neumann?
- Using the processor efficiently in the presence of I/O
  - Operating systems
  - Distributed systems
  - Real-time systems

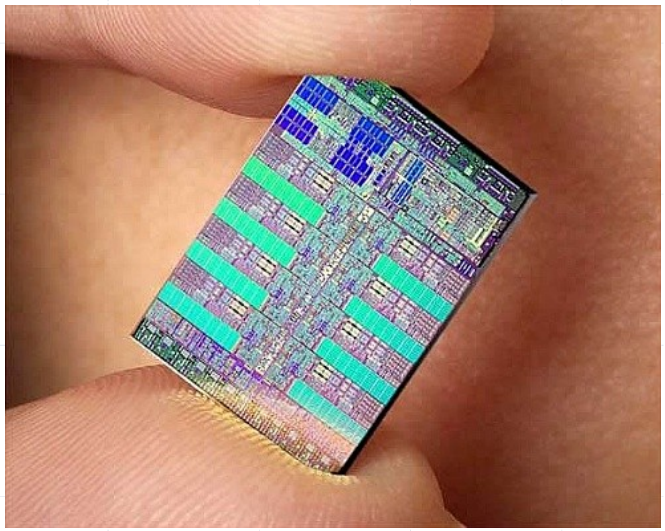
Press any key...

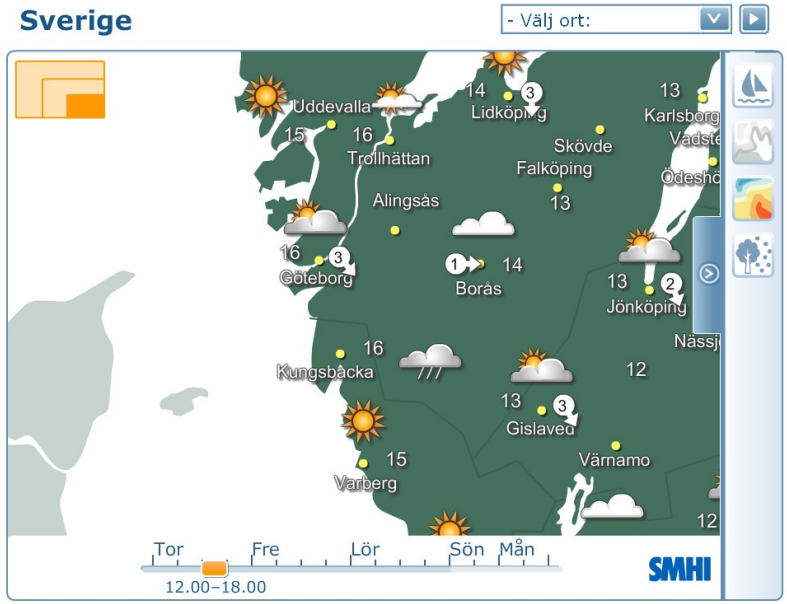
Why?

- Modeling inherently concurrent systems
  - Example: Software controllers which handle responses from several physical sources
- The real world is not sequential!

Why?

- Multi-core/Many-core/Multi-processor
- Performing computationally expensive tasks using multi-X hardware





Concurrency vs Parallelism

Parallell programmering ≠ parallel programming

- *Parallel*
  - physically at the same time
- *Concurrent*
  - logically at the same time, but might be implemented without any real parallelism
- The book covers parallel programming too – but it will not be the focus of this course

Course Goals – General

- Introduction to the problems common to many computing disciplines:
  - Operating systems
  - Distributed systems
  - Real-time systems
- Appreciation of the problems of concurrent programming
  - Classic synchronisation problems

Course Goals – Practical

- Understanding of a range of programming language constructs for concurrent programming
- Ability to apply these in practice to synchronisation problems in concurrent programming
- Practical knowledge of the programming techniques of modern concurrent programming languages

Practical Information

- Two lectures per week
- **Four** programming assignments – “labs”
  - Supervision/helpers available in lab rooms
- Optional weekly exercise classes. Attend at most one, with your lab partner.
- Written Exam
  - 4 hours
  - closed book
- Six supervision/exercise hours

Course Literature

- Mordechai (Moti) Ben-Ari
  - *Principles of Concurrent and Distributed Programming (Second edition)*
  - Main course book (just adopted)
- Gregory R. Andrews
  - *Foundations of Multithreaded, Parallel, and Distributed Programming*
  - Recommended reading
- *Joe Armstrong*
  - *Programming in Erlang*
  - Recommended reading

Course Communication

- Web pages: intended to answer most basic questions
  - http://www.cse.chalmers.se/edu/course/TDA381/
  - Tip: don’t search for JR, use local resources
- E-mail: concurrent-programming-period-3-2012@googlegroups.com

Gentle Start

- Introduction to concurrent programming
- Basic understanding
  - Concurrent programming concepts
    - Threads/Processes
    - State, Execution, Scheduling
  - Synchronisation problems
- Introduction to programming languages
  - Java
  - JR (MPD)

Your Summer Job

- Cremona decide to employ experts to increase sales. Their solution:

Buy @ Cremona !

- The message must be flashed every three seconds

Solution in JR

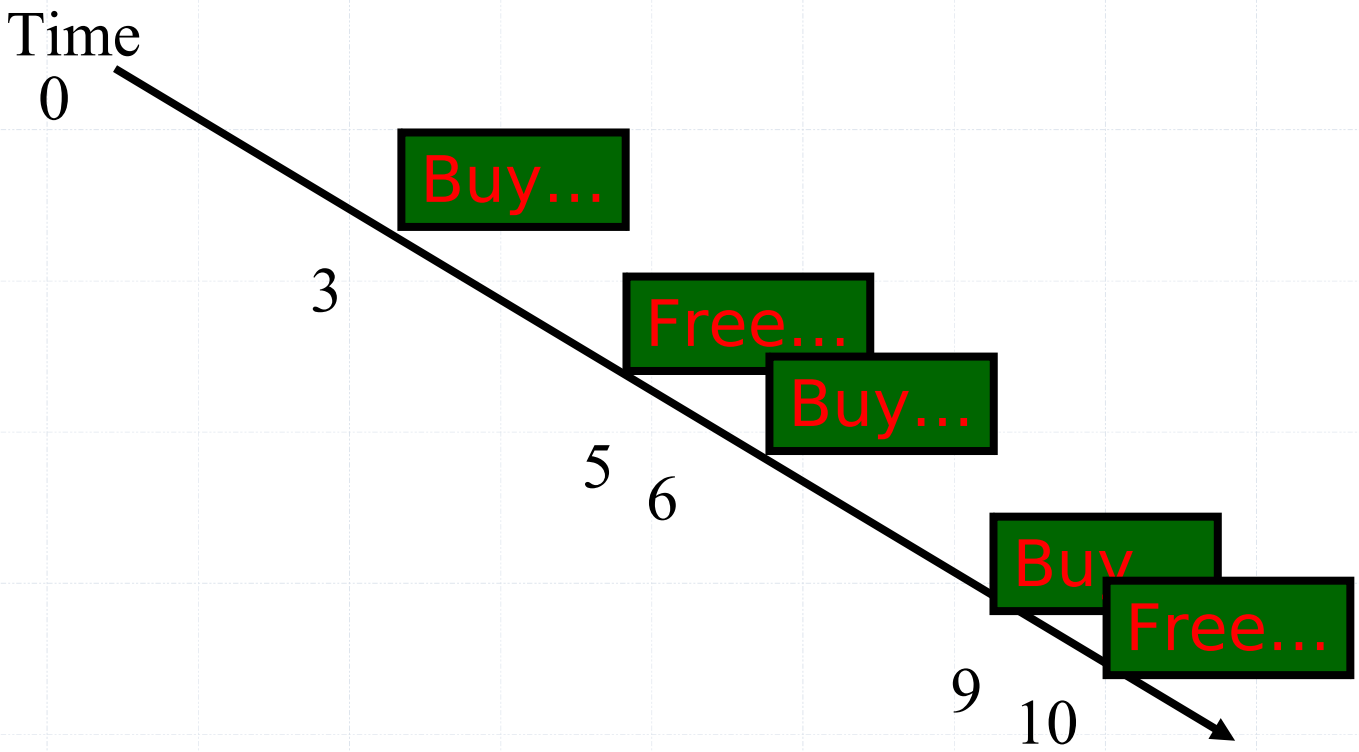
```
import edu.ucdavis.jr.*;
import javax.swing.*;

public class Main {

  public static void main(String[] args) {
    JFlash window = new JFlash("Cremona");
    SwingUtilities.invokeLater(window);
    while (true) {
      window.flash("Buy @ Cremona!");
      JR.nap(3000);
    }
  }
}
```

Next Summer

- The program does not increase sales as predicted. A psychologist is called in to help:
  - An additional message is needed: the sign must flash “Free beer!” every 5 seconds

Timeline

- The program is now more complex…



Revised Code

```
//the same init

    JFlash window = new JFlash("Cremona");
    SwingUtilities.invokeLater(window);

    final int buy_pause = 3000;
    final int beer_pause = 5000;

    int next_buy = buy_pause;
    int next_beer = beer_pause;

//continues on the next slide
```

```
while ( true ) {
  if ( next_buy < next_beer) {
     JR.nap(next_buy);
     window.flash("Buy @ Cremona");
     next_beer = next_beer - next_buy;
     next_buy = buy_pause;
  }
  else if ( next_buy > next_beer ) {
     JR.nap(next_beer);
     window.flash("Free Beer!");
     next_buy = next_buy - next_beer;
     next_beer = beer_pause;
  }
  else {
     JR.nap(next_buy);
     window.flash("Buy @ Cremona - Free Beer!");
     next_buy = buy_pause;
     next_beer = beer_pause;
  }
//the same end
```

Simple Concurrent Processes

- A more natural solution is to run the two simple algorithms *concurrently*:

```
while (true) {
  window.flash("Buy @ Cremona!");
  JR.nap(buy_pause);
}
```

```
while (true) {
  window.flash("Free Beer!");
  JR.nap(beer_pause);
}
```

Simple Concurrent Processes

```
//some init

private process buy {
   while (true) {
     window.flash("Buy @ Cremona!");
     JR.nap(buy_pause);
}}

private process beer {
   while (true) {
     window.flash("Free Beer!");
     JR.nap(beer_pause);
}}

//some end
```

Java Threads

- Java threads are a bit different from JR's simple process declaration
  - But there is more to processes in JR than that
- Java threading framework
  - The `Thread` class provides the API and generic behaviours
  - A concrete thread must provide a `run()` method which is the code that the thread will execute when started

Programming Threads

- Providing thread `run()` method
  - inheritance

```
class Buy extends Thread {
//some init
   public void run() {
       while (true) {
          window.flash("Buy @ Cremona!");
          //add napping here
       }
   }
}
```

Programming Threads

- Providing thread `run()` method
  - implement interface `Runnable`

```
class Buy implements Runnable {
//some init
   public void run() {
       while (true) {
          window.flash("Buy @ Cremona!");
          //add napping here
       }
   }
}
```

Running Java Threads

- Invoking the `run()` method in a new thread
  - Inheritance

```
buyThread = new Buy(…);
buyThread.start();
```

  - Interface

```
buyThread = new Thread(new Buy(…));
buyThread.start();
```

Running Java Threads

- Using anonymous inner classes

```
buyThread = new Thread() {
      public void run() {
         while (true) {
            window.flash("Buy @ Cremona!");
            //add napping here
         }
      }
   };
buyThread.start();
```

Napping in Java

- A sleeping thread can be interrupted, hence the need for the catch/try clause.

```
try {
   Thread.sleep(milliseconds);
}
catch (InterruptedException e) {
   //Panic: do something here!
}
```

- More on this later.

Processes and Threads

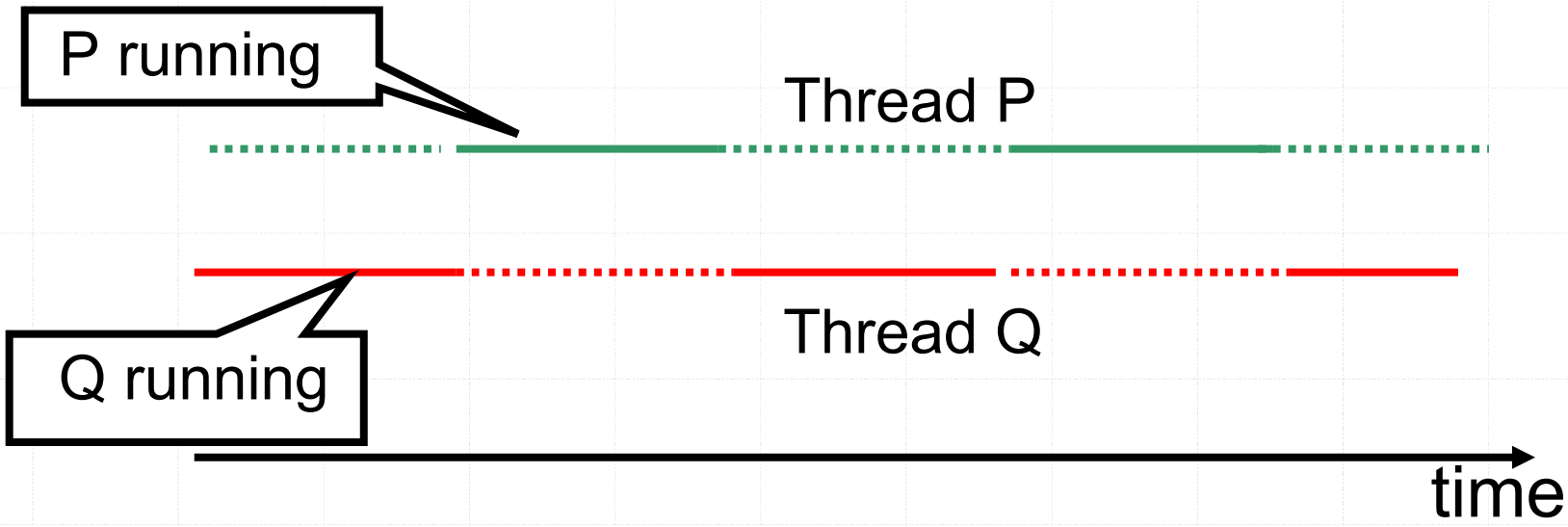
- A JR process is similar to a Java thread
- Terminological confusion: A multi-threaded Java program and a multi-process JR program both run as a single OS process.
- More about this later

Concurrent Programming Languages

- Using concurrent programming languages we will
  - Explore concurrency problems and solutions
  - Understand how modern programming languages support concurrent programming
- Main course programming languages
  - JR
  - Java
  - Erlang

Process Scheduling

- On a uniprocessor system threads appear to run at the same time but in fact their execution must be interleaved



Scheduling

- The job of switching between threads is performed by the scheduler
  - Part of the run-time system, or
  - Performed using the operating system's processes and scheduler

- Many different methods of scheduling exist

Scheduling – Continued

- Two extremes:
  - Cooperative scheduling
    - a thread runs until it is willing to release the processor (e.g. sleep or termination)
  - Preemptive scheduling
    - a thread is interrupted in order to let other threads continue (e.g. time-slicing)
- Erlang have a preemptive scheduler
- Most modern JVM's are also preemptive

Types of Process Behaviour

- Independent processes
  - Relatively rare; Rather uninteresting
- Competing
  - Typical in OS and networks, due to shared resources
- Cooperating
  - Processes combine to solve a common task

A Process

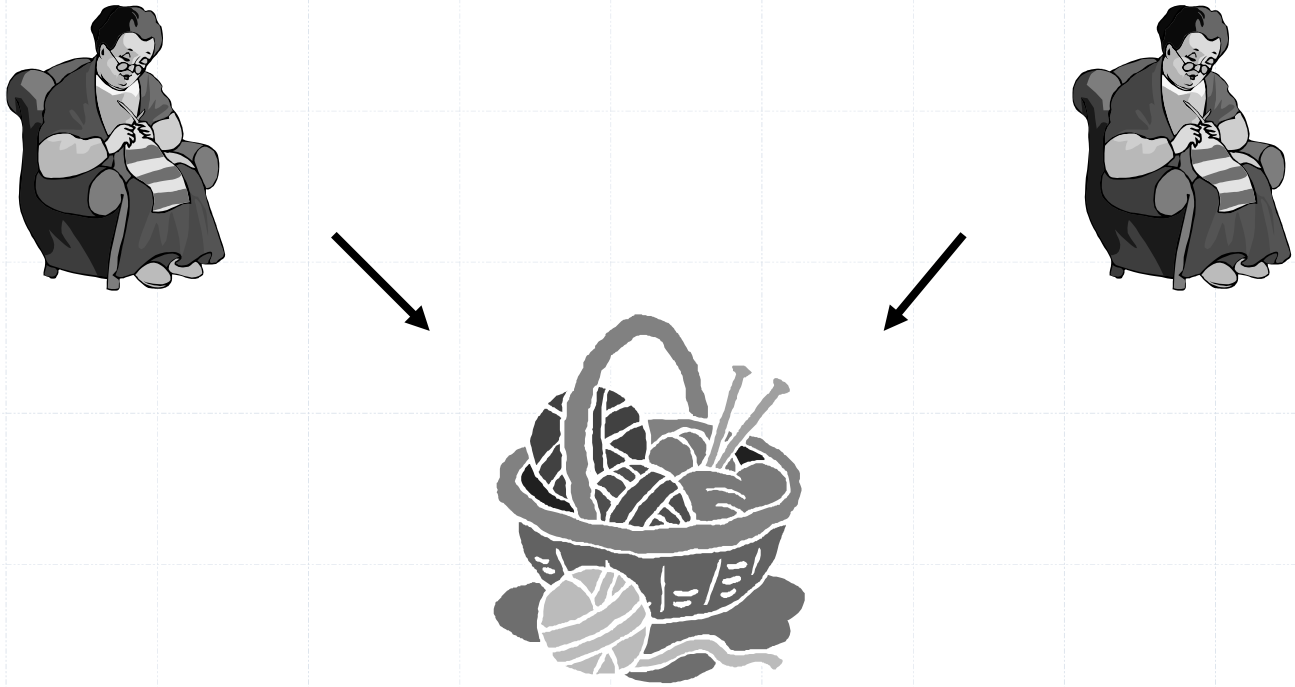
Types of Process Behaviour

- Designing concurrent systems is concerned with synchronisation and communication between processes
- Independent processes
  - Relatively rare; Rather uninteresting



Types of Process Behaviour

- Competing
  - Typical in OS and networks, due to shared resources



Types of Process Behaviour

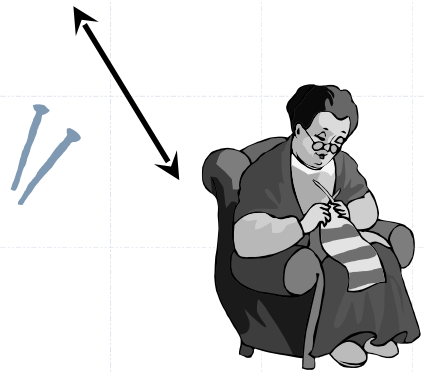
- Competing
  - Typical in OS and networks, due to shared resources

Deadlock

Types of Process Behaviour

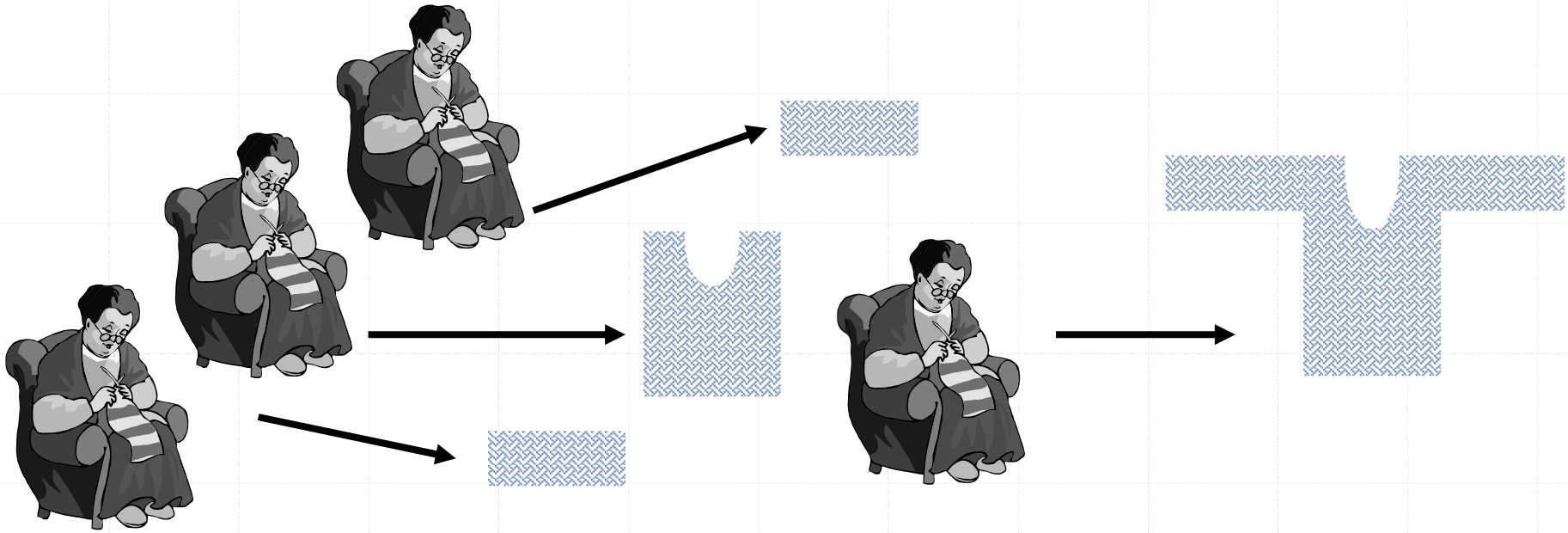
- Competing
  - Typical in OS and networks, due to shared resources

Starvation



Types of Process Behaviour

- Cooperating
  - Processes combine to solve a common task
  - Synchronisation



Atomicity

- An *atomic* action is something that is guaranteed to execute without interruption
- Since the execution of different threads is interleaved, what are the atomic actions?
  - Single instructions?
  - Basic code blocks?
  - *Answer*: might not specified by the language design. We have to assume the worst! Context switch can occur anywhere, also in the middle of a statement.

Atomicity

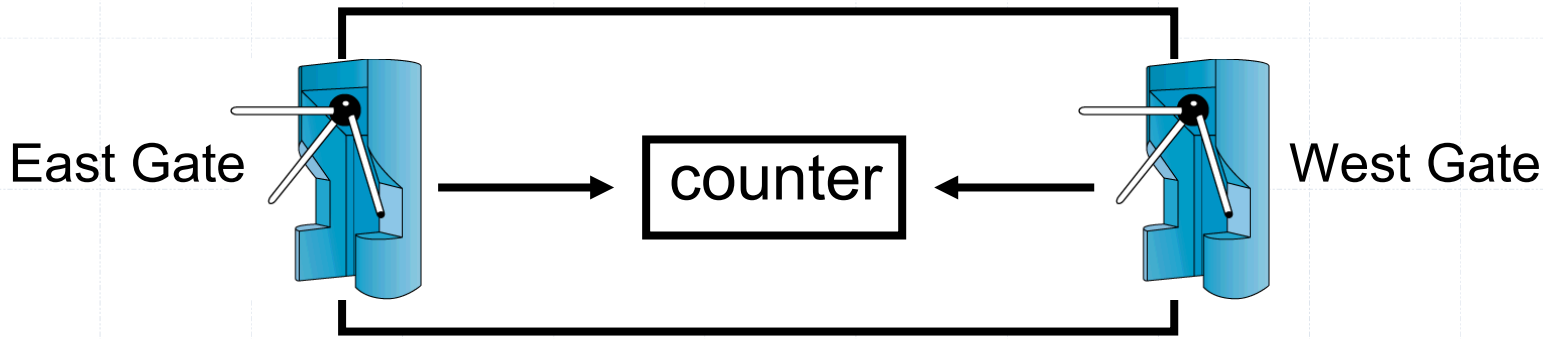
- What if flash is not atomic for the Cremona display?

```
while (true) {
  window.flash("Buy @ Cremona!");
  JR.nap(buy_pause);
}
```

```
while (true) {
  window.flash(“Free Beer!");
  JR.nap(beer_pause);
}
```

Example: The Liseberg Counter

- How many people are in Liseberg at any given time?
  - Each entrance has turnstiles which record when a person enters or leaves:



Simulation

```
private int counter = 0;
private enum Dir {East, West};

public void enter() {   counter++;   }

public process
Turnstile((Dir i : Dir.values())) {
   for(int j = 0; j<100; j++) {
      JR.nap(500+(int)(Math.random()*1000));
      System.out.println(i+" enters "+j);
      enter();
   }
}
```

Simulation – Quiescence

```
public Main() {
   try {
      JR.registerQuiescenceAction(done);
   }
   catch (QuiescenceRegistrationException e) {
     e.printStackTrace();
   }
}

public op void done() {
   System.out.println("Counter: "+counter);
}
```

What is the answer?

- We expect the answer 200.
  - But it depends on the `counter++` operation being *atomic*.
  - What if it is implemented using three atomic actions: `load`, `add`, `store`

```
load R1, counter
add R1, #1
store R1, counter
```

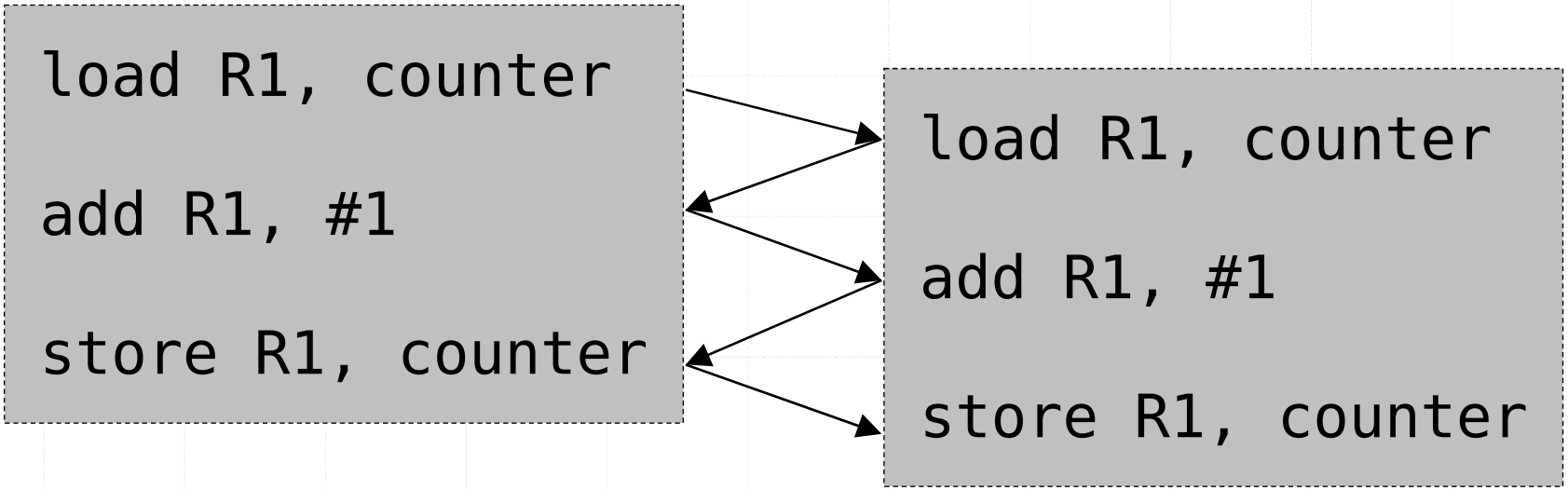
Terminology: States and Traces

- A program executes a sequence of atomic actions
- A *state* is the value of the program variables at any point in time
- A *trace* (or history) is a sequence of states that can be produced by the sequence of atomic actions of a program

A Bad Trace

- Suppose the first atomic actions of the Turnstile processes are interleaved as follows:

```
counter = 0
```




```
counter = 1
```

Program Properties

- A *property* of a program is a logical statement that is true for every possible trace
- Two kinds of property are usual for stating correctness properties of concurrent programs
  - Safety property
    - a trace never enters a “bad” state
  - Liveness property
    - every trace eventually reaches a “good” state

Program Properties

- Example safety properties could be of the form:
  - The program never produces a wrong answer
  - An invariant `(x + y < 2)`
- Example liveness properties:
  - The process terminates
  - The process eventually calls a certain procedure

Synchronisation

- *Synchronisation* is the restriction of the traces of a concurrent program in order to guarantee certain safety properties.

- We will see at least two kinds of synchronisation:
  - Mutual exclusion
  - Conditional Synchronisation

Critical Sections

- The “bad” traces in the Liseberg problem are caused by the code that implements `counter++`
- To fix the problem it must be executed atomically
  - Without any interleaving or parallel activity
- Critical section
  - A part of a program that must be executed atomically

Mutual Exclusion

- Mutual exclusion
  - The property that only one process can execute in a given piece of code
- How can we achieve it?
  - Theory: possible with just shared variables
    - very inefficient at programming language level
    - but sometimes necessary in very low-level (HW)
  - Practice: programming language features (semaphores, monitors, …)

Summary

- Today's lecture
  - Introduction to concurrency
  - Processes/threads in JR/Java
  - The shared update problem: mutex
- Next time
  - Solving the shared update problem
  - Introduction to a first programming language construct for synchronisation:
    - semaphores

Sayings

- The greatest performance improvement of all is when a system goes from not-working to working
- The only thing worse than a problem that happens all the time is a problem that doesn't happen all the time