Concepts of Concurrent Computation
Spring 2015
Lecture 1: Overview

Sebastian Nanz
Chris Poskitt
Practical details

- **Schedule**
  - Tuesday 10-12 RZ F 21 lecture
  - Wednesday 14-15 RZ F 21 exercise class
  - Wednesday 15-17 RZ F 21 seminar

- **Course page**
  - [http://se.inf.ethz.ch/courses/2015a_spring/ccc/](http://se.inf.ethz.ch/courses/2015a_spring/ccc/)

- **Lecturers**
  - Dr. Sebastian Nanz
  - Dr. Chris Poskitt

- **Assistants**
  - Chandrakana Nandi (exercise class)
  - Mischael Schill (project)
Grading

- **Exam: 50%**
  - End-of-semester exam
  - Date: 26 May 2015 (at the usual lecture time)

- **Project: 35%**
  - Build a small concurrent system

- **Seminar talk: 15%**
  - Present a recent research paper
Project

- Organization
  - Teams of 1-3 students
  - Multiple deadlines/milestones
  - Support: Mischael Schill + mailing list

- Project overview
  - “Bomberman” game
  - Less concurrency-relevant code given
  - Implemented using SCOOP

- What is SCOOP?
  - A high-level programming model for concurrency
  - Covered in a future lecture
  - For ease of installation, programming framework and project files provided as virtual machine
Seminar: Overview

- The seminar connects the course topics to recent research results
  - Research papers from 2011-2014

- The seminar consists of student presentations
  - 15 min paper presentation (with slides) + questions

- The seminar lives from discussions about the papers
  - Read papers and prepare questions in advance
Seminar: Grading

- **Content**
  - Technical correctness
  - Coherent development of concepts
  - Selection of content
  - Visualization of content
  - **Own contributions: own examples, own evaluation, tracing of the paper’s impact**

- **Presentation**
  - Slides (style, grammar, spelling)
  - Use of other aids
  - Voice & speech
  - Audience engagement/stage presence
  - Timing/pace
Seminar: Paper selection

- You will get an email today, with a list of papers and instructions for telling us your choice (doodle)
- Respond no later than **this Friday, 20 February, 12:00**
- If you don’t get the email today or miss the deadline, please email the assistants

- **Tomorrow, 18 February:**
  - 14:15 First exercise class
  - Hand-out of the project description
  - No seminar: use the time for paper selection
Purpose of the course

- To introduce you to the main concurrency approaches and give you an idea of their strength and weaknesses
  - Practical approaches to concurrent programming
  - Modelling and reasoning about concurrency
- To enable you to get a concrete grasp of the issues and solutions through a course project
- To connect to recent research through a seminar
Course overview

- Practical approaches to concurrent programming
  - Issues: data races, deadlock, starvation
  - Synchronization algorithms
  - Semaphores
  - Monitors
  - Language examples: SCOOP and others
  - Lock-free programming and Software Transactional Memory

- Modelling and reasoning about concurrency
  - Proofs of concurrent programs
  - Temporal logic
  - Petri nets
  - Process calculi: CCS
Crossing the chasm

- Formal models provide an elegant theoretical basis, but
  - Have little connection with practice
  - Handle concurrency aspects only

- Practice of concurrent programming
  - Little influenced by above
  - Low-level mechanisms still predominant

- In the course, we look at both theoretical and practical approaches to concurrency
Recommended textbooks

- Draft of a textbook for this course
- More literature recommendations: see individual lectures
What is concurrency?
Origins of concurrency in computing

- Concurrency is not a new topic but one most programmers have been able to avoid

- Previously perceived as a very specialized topic
  - *Systems programming*
  - Databases
  - High-performance computing
Multiprocessing

- Many of today’s computations can take advantage of multiple processing units (multi-core processors)

  ![Diagram](image)

  - **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/multithreading

- Even on systems with a single processing unit we may give the illusion of that several programs run at once.

Multitasking/multithreading: the operating system switches between the execution of different tasks/threads.

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

- 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.
- In this terminology, parallelism is a form of concurrency.

In programming, the terms are often used to emphasize the type of problem they solve:

- **Concurrent programming**: nondeterministic composition of independently executing processes.
- **Parallel programming**: efficient execution of a deterministic computation on multiple processing units.
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
- The scheduler 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 assigned to a processor
  - **terminated**: finished executing
Blocked processes

- A process can get into state **blocked** by executing special program instructions (synchronization primitives)
- 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
Threads

- Make programs concurrent by associating them with threads
- A thread is a part of an operating system process

- Private components
  - Thread identifier
  - Thread state
  - Thread context
  - Memory: only stack

- Shared components
  - Program text
  - Global data
  - Heap
Expressing concurrency
Example: Java Threads

- 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;
    }
}

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();
}
```
Abstract notation

- 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

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Code of concurrently executing threads</th>
</tr>
</thead>
</table>
| P1        | 1: x := 0  
            | 2: x := x + 1  |
| P2        | 1: x := 2  |
Execution sequences

Execution can give rise to this execution sequence

<table>
<thead>
<tr>
<th>x := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Variable values after execution of the code on the line

Instruction executed with Thread ID and line number

<table>
<thead>
<tr>
<th>P1</th>
<th>1</th>
<th>x := 0</th>
<th>x = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1</td>
<td>x := 2</td>
<td>x = 2</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
<td>x = 3</td>
</tr>
</tbody>
</table>

Is this the only possible execution sequence?
Benefits and challenges of concurrency
Why concurrency?

- Responsiveness
  - GUI programming
  - Network programming
  - Communicating with multiple hardware devices

- Program structuring
  - Handle nondeterministic events in a modular way
  - Model concurrency in the real world

- Performance
  - Speeding up computations
The end of Moore’s Law as we knew it

Clock speed

Transistor density

Source: Intel
Why do we care?

- The “end of Moore’s law as we knew it” has important implications on the software construction process.
- Computing is taking an irreversible step toward parallel architectures:
  - Hardware construction of ever faster sequential CPUs has hit physical limits.
  - Clock speed no longer increases for every new processor generation.
  - Moore’s Law expresses itself as exponentially increasing number of processing cores per chip.
- If we want programs to run faster on the next processor generation, the software must exploit more concurrency.
Amdahl’s Law

- We go from 1 processor to n processors. What gain may we expect?
- Amdahl’s law severely limits our hopes!
- Define gain as:
  \[
  \text{speedup} = \frac{\text{old\_execution\_time}}{\text{new\_execution\_time}}
  \]
- Not everything can be parallelized!
Amdahl’s law: Example (1)

- Assume 10 processing units. How close are we to a 10-fold speedup?
  - 60% concurrent, 40% sequential:
    
    \[
    \text{speedup} = \frac{1}{1 - 0.6 + \left( \frac{0.6}{10} \right)} = 2.17
    \]

- 80% concurrent, 20% sequential:
  
  \[
  \text{speedup} = \frac{1}{1 - 0.8 + \left( \frac{0.8}{10} \right)} = 3.57
  \]
Amdahl’s law: Example (2)

- 90% concurrent, 10% sequential:

\[
\text{speedup} = \frac{1}{1 - 0.9 + \left( \frac{0.9}{10} \right)} = 5.26
\]

- 99% concurrent, 1% sequential:

\[
\text{speedup} = \frac{1}{1 - 0.99 + \left( \frac{0.99}{10} \right)} = 9.17
\]
Types of concurrent computation
Types of parallel computation

- **Flynn’s taxonomy**: classification of computer architectures
- Considers relationships of instruction streams to data streams

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td></td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

**SISD**
No parallelism (uniprocessor)

**SIMD**
Vector processor
GPU

**MIMD**
Multiprocessing (predominant today)
MIMD variants

- **SPMD** (Single Program Multiple Data)
  - All processors run the same program, but at independent speeds
  - No lockstep as in SIMD

- **MPMD** (Multiple Program Multiple Data)
  - Often manager/worker strategy: manager distributes tasks, workers return result to manager
Shared memory model

- All processors share a common memory
- Shared-memory communication
Distributed memory model

- Each processor has own local memory, inaccessible to others
- **Message-passing** communication
- Common for SPMD architecture