Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz
Chris Poskitt

Lecture 1: Overview
Practical Details

• Schedule
  • Tuesday 10-12, RZ F21: course
  • Wednesday 14-15, RZ F21: exercise
  • Wednesday 15-17, RZ F21: seminar

• Course page
  • http://se.inf.ethz.ch/courses/2014a_spring/ccc/

• Lecturers
  • Prof. Dr. Bertrand Meyer
  • Dr. Sebastian Nanz
  • Dr. Chris Poskitt

• Assistants
  • Dr. Georgiana Caltais
  • Mischael Schill

firstname.lastname@inf.ethz.ch
Grading

Exam: 50%
- End of semester (not in the semester break)
- Date: 27 May 2014 at the usual lecture time

Project: 35% (build a small concurrent system)

Seminar talk: 15%

Credit points: 7
Seminar

- The seminar connects the course topics to the most recent research results
- The seminar consists of student presentations (20 min + questions) on a research paper on concurrency
- The seminar lives from discussions about the papers: prepare questions about the papers to be presented
Seminar grading

• **Content:**
  - technical correctness
  - coherent development of concepts
  - selection of content
  - visualization of content
  - own contributions (such as 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
Paper selection for the seminar

• You will get an email today, with a list of papers and instructions for e-mailing us your choice
• You must respond no later than Friday, 21 February, 12:00
• If you don’t get the email today or miss the deadline, please email the assistants

• In tomorrow’s seminar, 19 February, 15:15 there will be a talk on “How to give a technical presentation”
• No exercise class tomorrow, 19 February (use the time for paper selection)
Purpose of the course

- To give you a practical grasp of the excitement and difficulties of building modern concurrent applications
- To expose you to newer forms of concurrency
- To introduce you to the main concurrency approaches and give you an idea of their strength and weaknesses
- To present some of the concurrency calculi
- To study one particular approach in depth: SCOOP
- 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

Introduction

Concurrent and parallel programming, Multitasking and multiprocessing, Shared-memory and distributed-memory multiprocessing, Notion of process and thread, Performance of concurrent systems

Approaches to concurrent programming

Issues (data races, deadlock, starvation), Synchronization algorithms, Semaphores, Monitors, Java and .NET multithreading

The SCOOP model

Processors, Synchronous and asynchronous feature calls, Separate objects and entities, Synchronization, Examples and applications

Programming approaches to concurrency

Message-passing vs. shared-memory communication, Language examples (Ada, Polyphonic C#, Erlang (Actors), X10, Linda, Cilk and others), Lock-free programming, Software Transactional Memory

Reasoning about concurrent programs

Properties of concurrent programs, Temporal logic, Process calculi (CSP, CCS), Petri nets, Proofs of concurrent programs
Concurrency: benefits and challenges
Why concurrency?

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

Previously perceived as a very specialized topic: high-performance computing, systems programming, databases

Reasons for introducing concurrency into programs:

- **Efficiency**
  - Time (load sharing)
  - Cost (resource sharing)
- **Availability**
  - Multiple access
- **Convenience**
  - Perform several tasks at once
- **Modeling power**
  - Describing systems that are inherently parallel
Modeling a concurrent world

Computer systems are used for modeling objects in the real world

- **Object-oriented programming**

The world often includes parallel operation

**Typical example:**

- Limited number of seats on the same plane
- Several booking agents active at the same time
Multiprocessing, parallelism

Many of today's computations can take advantage of multiple processing units (through *multi-core* processors):

Terminology:

- **Multiprocessing**: the use of more than one processing unit in a system
- **Parallel execution**: processes running at the same time
Multitasking, concurrency

Even on systems with a single processing unit we may give the illusion of that several programs run at once. The OS switches between executing different tasks.

Terminology:
- **Interleaving**: several tasks active, only one running at a time
- **Multitasking**: the OS runs interleaved executions
- **Concurrency**: multiprocessing, multitasking, or any combination
The end of Moore's Law as we knew it

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 $n$. What gain may we expect?

*Amdahl’s law* severely limits our hopes!

Define gain as:  

$$speedup = \frac{old\_execution\_time}{new\_execution\_time}$$

Not everything can be parallelized!

$$speedup = \frac{1}{1 - p + \frac{p}{n}}$$

*3 slides adapted from material by Maurice Herlihy*
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 parallel computation

Flynn’s taxonomy: classification of computer architectures
Considers relationship 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 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
Client-server model

Specific case of the distributed model
Examples: Database-centered systems, World-Wide Web
SCOOP: the trailer
SCOOP mechanism

Simple Concurrent Object-Oriented Programming

Evolved through previous two decades; CACM (1993) and chap. 32 of Object-Oriented Software Construction, 2nd edition, 1997

Prototype-implementation at ETH in 2007

Implementation integrated within EiffelStudio (by Eiffel Software)

Key references: ETH PhD Thesis by Piotr Nienaltowski, 2008; articles by Benjamin Morandi, Sebastian Nanz, Bertrand Meyer, and others (2010-2013)
transfer (source, target: ACCOUNT; amount: INTEGER)
-- If possible, transfer amount from source to target.
do
if source.balance >= amount then
    source.withdraw (amount)
target.deposit     (amount)
end
end

Typical calls:
transfer (acc1, acc2, 100)
transfer (acc1, acc3, 100)
In a concurrent setting, using SCOOP

\[
\text{transfer (source, target: } \text{ACCOUNT}; \text{ amount: INTEGER)}
\]

-- If possible, transfer amount from source to target.

do

if source.balance \geq amount \text{ then}

source.withdraw (amount)

target.deposit (amount)

end

end

Typical calls:

transfer (acc1, acc2, 100)

transfer (acc1, acc3, 100)
A better SCOOP version

transfer (source, target: ACCOUNT; amount: INTEGER)
   -- Transfer amount from source to target.

require
   source.balance >= amount

do
   source.withdraw (amount)
   target.deposit   (amount)

ensure
   source.balance = old source.balance - amount
   target.balance = old target.balance + amount

end
put \((b : \text{BUFFER}[G] ; v : G)\)

-- Store \(v\) into \(b\).

require

not \(b.\text{is\_full}\)

do

...

ensure

not \(b.\text{is\_empty}\)

end

my\_queue : \text{QUEUE}[T]

...

if not my\_queue.is\_full then

put(my\_queue, t)

end
Dining philosophers

class PHILOSOPHER inherit PROCESS
    rename
        setup as getup
    redefine step end

feature {BUTLER}
    step
        do
            think;  eat (left, right)
        end

    eat (l, r: separate FORK)
        -- Eat, having grabbed l and r.
        do ... end
end
The issue

Concurrency everywhere:
- Multithreading
- Multitasking
- Networking, Web services, Internet
- Multicore

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
## Previous advances in programming

<table>
<thead>
<tr>
<th></th>
<th>“Structured programming”</th>
<th>“Object technology”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use higher-level abstractions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Helps avoid bugs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfers tasks to implementation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lets you do stuff you couldn’t before</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Removes restrictions</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Adds restrictions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Has well-understood math basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Doesn’t require understanding that basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Permits less operational reasoning</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Then and now

Sequential programming:
Used to be messy
Still hard but key improvements:
- Structured programming
- Data abstraction & object technology
- Design by Contract
- Genericity, multiple inheritance
- Architectural techniques

Concurrent programming:
Used to be messy
Still messy
Example: threading models in most popular approaches
Development level: sixties/seventies
Only understandable through operational reasoning
The chasm

Theoretical models, process calculi provide an elegant theoretical basis, but

- have little connection with practice (some exceptions, e.g. BPEL)
- handle concurrency aspects only

Practice of concurrent & multithreaded programming

- Little influenced by above
- Low-level, e.g. semaphores
- Poorly connected with rest of programming model