Concepts of Concurrent Computation
Bertrand Meyer
Sebastian Nanz

Lecture 1: Welcome and introduction
Practical Details

• Schedule
  • Course: Tuesday 10-12, RZ F21
  • Exercise: Tuesday 12-13, RZ F21

• Course page
  • Check it at least once a week: http://se.inf.ethz.ch/teaching/2010-S/0268/

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

• Assistant
  • Benjamin Morandi
  • Scott West
Grading

• Exam 50%
  • Will be held at the end of the semester (not in the semester break).
  • Exam date: June 1, 2010 during the usual lecture hours

• Project 50%
  • An application to be built using SCOOP
Course description (from catalog)

• This course explores the connections between the object oriented and concurrent programming paradigms, discussing the problems that arise in the process of attempting to merge them.

• It reviews the main existing approaches to concurrent O-O computation, including both widely used libraries for multi-threading in Java and .NET and more theoretical frameworks, with a particular emphasis on the SCOOP model.

• It also provides some of the formal background for discussing the correctness of concurrent O-O applications.
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 study how the object-oriented paradigm transposes to concurrent settings, and how it can help address concurrency issues
- 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 in depth one particular approach: SCOOP
- To enable you to get a concrete grasp of the issues and solutions through a course project
Two sides of the same coin

“Classic” part
- Survey of classic and modern approaches
- Explains historical evolution
- Illustrates problems and solutions e.g., Java

SCOOP part
- The “object lesson”
- High-level support for concurrency
- Concurrency solution integrated with an OO programming language, i.e., Eiffel
- Starts from object-oriented programming as a given, adds concurrency
Concurrency: benefits and challenges
Material (slightly adapted) from

The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Moore’s Law

- Transistor count still rising
- Clock speed flattening sharply

Source: Intel
Uniprocessor
Shared Memory Multiprocessor (SMP)
Multicore Processor (CMP)

All on the same chip

Sun T2000 Niagara
Why do we care about multicore processors?

• Time no longer cures software bloat
  • The “free ride” is over
• When you double your program’s path length
  • You can’t just wait 6 months
  • Your software must somehow exploit twice as much concurrency
Traditional scaling process

Speedup

1.8x 3.6x 7x

User code

Traditional Uniprocessor

Time: Moore’s law
Multicore scaling process: the hope

Unfortunately, not so simple...
Real scaling process

Speedup
1.8x  2x  2.9x

User code

Multicore

Parallelization and Synchronization require great care...
Sequential computation

memory

object

object

thread
Concurrent computation

memory

object

object
Sudden unpredictable delays

- Cache misses (short)
- Page faults (long)
- Scheduling quantum used up (really long)
Model summary

- Multiple threads
  - Sometimes called processes
- Single shared memory
- Objects live in memory
- Unpredictable asynchronous delays
Concurrency jargon

• Hardware
  • Processors
• Software
  • Threads, processes
• Sometimes OK to confuse them, sometimes not.
Example: parallel primality testing

- **Challenge**
  - Print primes from 1 to $10^{10}$

- **Given**
  - Ten-processor multiprocessor
  - One thread per processor

- **Goal**
  - Get ten-fold speedup (or close)
Load balancing

- Split the work evenly
- Each thread tests range of $10^9$
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10^9+1, j<(i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  • Uneven
  • Hard to predict
• Need dynamic load balancing
Amdahl’s Law

\[ \text{speedup} = \frac{\text{old execution time}}{\text{new execution time}} \]

...of computation given n CPUs instead of 1
Amdahl’s Law

\[
\text{speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

- **Sequential fraction**: \(1 - p\)
- **Parallel fraction**: \(\frac{p}{n}\)
- **Number of processors**: \(n\)
Example

• Ten processors
• 60% concurrent, 40% sequential
• How close to 10-fold speedup?

\[
\text{speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}
\]
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
\text{speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}
\]
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
\text{speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}
\]
Example

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

\[ \text{speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}} \]
The moral

- Making good use of our multiple processors (cores) means finding ways to effectively parallelize our code
  - Minimize sequential parts
  - Reduce idle time in which threads wait without doing something useful.
SCOOP Taster
my_queue : QUEUE [T ]
...
if not my_queue.is_full then
  put (my_queue, t )
end

put (b : QUEUE [G ]; v : G)
  -- Store v into b.
require
  not b.is_full
do
  ...
ensure
  not b.is_empty
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... Elegant theoretical basis, but

- 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
Wrong (in my opinion) assumptions

“Objects are naturally concurrent” (Milner)

- Many attempts, often based on “Active objects” (a self-contradictory notion)
- Lead to artificial issue of “Inheritance anomaly”

“Concurrency is the basic scheme, sequential programming a special case” (many)

- Correct in principle, but in practice we understand sequential best
Simple Concurrent Object-Oriented Programming

Evolved through last decade; CACM (1993) and chap. 30 of Object-Oriented Software Construction, 2nd edition, 1997

Implemented at ETH, integrated into EiffelStudio

Current state is described in Piotr Nienaltowski’s 2007 ETH PhD dissertation
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 /and r.
    do ... end
end