CP — Concurrency: State Models and Design Patterns

1. Introduction

Oscar Nierstrasz
## Concurrent Programming

<table>
<thead>
<tr>
<th>Lecturer</th>
<th>Prof. Oscar Nierstrasz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistants</td>
<td>Haidar Osman, Nevena Milojkovic, Leonel Merino</td>
</tr>
<tr>
<td>Lectures</td>
<td>Wednesday @ 10h15-12h00</td>
</tr>
<tr>
<td>Exercises</td>
<td>Wednesday @ 12h00-13h00</td>
</tr>
<tr>
<td>Labs</td>
<td>ExWi Pool (selected dates …)</td>
</tr>
<tr>
<td>WWW</td>
<td><a href="http://scg.unibe.ch/teaching/cp/">http://scg.unibe.ch/teaching/cp/</a></td>
</tr>
</tbody>
</table>
MSc registration Autumn 2015

JMCS students

• Register on Academia for teaching units by Sept 30, 2015

NB: Hosted JMCS students (e.g. CS bachelor students etc.) must additionally:

• Request for Academia access by Sept 30, 2015
Roadmap

> Course Overview
> Concurrency and Parallelism
> Challenges: Safety and Liveness
> Expressing Concurrency
  — Process Creation
  — Communication and Synchronization
Course Overview
Concurrency and Parallelism
Challenges: Safety and Liveness
Expressing Concurrency
  — Process Creation
  — Communication and Synchronization
Goals of this course

> Introduce *basic concepts of concurrency*
  —safety, liveness, fairness

> Present tools for *reasoning* about concurrency
  —LTS, Petri nets

> Learn the *best practice* programming techniques
  —idioms and patterns

> Get *experience* with the techniques
  —lab sessions
## Schedule

<table>
<thead>
<tr>
<th></th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16-Sep-15</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>23-Sep-15</td>
<td>Java and Concurrency</td>
</tr>
<tr>
<td>3</td>
<td>30-Sep-15</td>
<td>Safety and Synchronization</td>
</tr>
<tr>
<td>4</td>
<td>7-Oct-15</td>
<td>Safety Patterns + Transactional Memory</td>
</tr>
<tr>
<td>5</td>
<td>14-Oct-15</td>
<td>Liveness and Guarded Methods</td>
</tr>
<tr>
<td>6</td>
<td>21-Oct-15</td>
<td>Lab session (ExWi Pool)</td>
</tr>
<tr>
<td>7</td>
<td>28-Oct-15</td>
<td>Liveness and Asynchrony</td>
</tr>
<tr>
<td>8</td>
<td>4-Nov-15</td>
<td>Condition Objects</td>
</tr>
<tr>
<td>9</td>
<td>11-Nov-15</td>
<td>Fairness and Optimism</td>
</tr>
<tr>
<td>10</td>
<td>18-Nov-15</td>
<td>Lab session (ExWi Pool)</td>
</tr>
<tr>
<td>11</td>
<td>25-Nov-15</td>
<td>Parallelism: Hadoop and Scala (invited lectures)</td>
</tr>
<tr>
<td>12</td>
<td>2-Dec-15</td>
<td>Petri Nets</td>
</tr>
<tr>
<td>13</td>
<td>9-Dec-15</td>
<td>Architectural Styles for Concurrency</td>
</tr>
<tr>
<td>14</td>
<td>16-Dec-15</td>
<td>Exam</td>
</tr>
</tbody>
</table>
Texts

Doug Lea: *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996

Further reading

Brian Goetz et al,
Java Concurrency in Practice,
Addison Wesley, 2006.
Roadmap

> Course Overview
> Concurrency and Parallelism
> Challenges: Safety and Liveness
> Expressing Concurrency
  — Process Creation
  — Communication and Synchronization
Concurrenty

> A **sequential program** has a *single thread of control.*
  — Its execution is called a **process.**

> A **concurrent program** has *multiple threads of control.*
  — These may be executed as **parallel processes.**
## Parallelism

A concurrent program can be executed by:

<table>
<thead>
<tr>
<th>Multiprogramming:</th>
<th>processes <em>share one or more processors</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessing:</td>
<td>each process runs on its own processor but with <em>shared memory</em></td>
</tr>
<tr>
<td>Distributed processing:</td>
<td>each process runs on <em>its own processor</em> connected by a network to others</td>
</tr>
</tbody>
</table>

Assume only that all processes make positive finite progress.
Why do we need concurrent programs?

> **Reactive programming**
  — minimize response delay; maximize throughput

> **Real-time programming**
  — process control applications

> **Simulation**
  — modelling real-world concurrency

> **Parallelism**
  — speed up execution by using multiple CPUs

> **Distribution**
  — coordinate distributed services
Roadmap

> Course Overview
> Concurrency and Parallelism
> Challenges: Safety and Liveness
> Expressing Concurrency
  — Process Creation
  — Communication and Synchronization
Difficulties

But concurrent applications introduce complexity:

**Safety**
> concurrent processes may corrupt shared data

**Liveness**
> processes may “starve” if not properly coordinated

**Non-determinism**
> the same program run twice may give different results

**Run-time overhead**
> thread construction, context switching and synchronization take time
Concurrent and atomicity

Programs $P_1$ and $P_2$ execute concurrently:

$$\begin{align*}
\{ x &= 0 \} \\
P_1: & \quad x := x + 1 \\
P_2: & \quad x := x + 2 \\
\{ x &= ? \}
\end{align*}$$

What are possible values of $x$ after $P_1$ and $P_2$ complete?

What is the intended final value of $x$?
Safety

**Safety** = *ensuring consistency*

A safety property says “*nothing bad happens*”

— *Mutual exclusion*: shared resources must be *updated atomically*
— *Condition synchronization*: operations may be *delayed* if shared resources are in the wrong state
  - (e.g., *read from empty buffer*)
Liveness

Liveness = ensuring progress

A liveness property says “something good happens”

— No Deadlock: some process can always access a shared resource
— No Starvation: all processes can eventually access shared resources
Roadmap

- Course Overview
- Concurrency and Parallelism
- Challenges: Safety and Liveness
- Expressing Concurrency
  — Process Creation
  — Communication and Synchronization
Expressing Concurrency

A programming language must provide mechanisms for:

**Process creation**
- how do you specify *concurrent processes*?

**Communication**
- how do processes *exchange information*?

**Synchronization**
- how do processes *maintain consistency*?
Most concurrent languages offer some variant of the following:

> Co-routines
> Fork and Join
> Cobegin/coend
Co-routines are only *pseudo-concurrent* and require *explicit transfers of control*.

Co-routines can be used to implement most higher-level concurrent mechanisms.
Fork and Join

**Fork** can be used to create any number of processes: **Join** waits for another process to terminate.

![Diagram of Fork and Join processes]

Fork and join are unstructured, so require care and discipline!
Cobegin/coend blocks are better structured:

\[
\text{cobegin } S_1 \parallel S_2 \parallel \ldots \parallel S_n \text{ coend}
\]

but they can only create a *fixed number of processes*. The caller continues when all of the coblocks have terminated.
Roadmap

> Course Overview
> Concurrency and Parallelism
> **Challenges: Safety and Liveness**
> **Expressing Concurrency**
  — Process Creation
  — Communication and Synchronization
Communication and Synchronization

In approaches based on shared variables, processes communicate indirectly. Explicit synchronization mechanisms are needed.

In message passing approaches, communication and synchronization are combined. Communication may be either synchronous or asynchronous.
Synchronization Techniques

Different approaches are roughly equivalent in expressive power and can be used to implement each other.

Each approach emphasizes a different style of programming.
Busy-Waiting

Busy-waiting is primitive but effective
Processes atomically set and test shared variables.

*Condition synchronization* is easy to implement:
— to signal a condition, a process *sets* a shared variable
— to wait for a condition, a process *repeatedly tests* the variable

*Mutual exclusion* is more difficult to realize correctly and efficiently ...
Semaphores were introduced by Dijkstra (1968) as a higher-level primitive for process synchronization.

A semaphore is a non-negative, integer-valued variable $s$ with two operations:

$> P(s)$:
- delays until $s > 0$
- then, atomically executes $s := s - 1$

$> V(s)$
- atomically executes $s := s + 1$
Many problems can be solved using **binary semaphores**, which take on values 0 or 1.

```plaintext
process P1
  loop
    P (mutex) { wants to enter }
    Critical Section
    V (mutex) { exits }
    Non-critical Section
  end
end

process P2
  loop
    P (mutex)
    Critical Section
    V (mutex)
    Non-critical Section
  end
end
```

Semaphores can be implemented using busy-waiting, but usually implemented in O/S kernel.
A monitor encapsulates resources and operations that manipulate them:

> operations are invoked like ordinary procedure calls
  — invocations are guaranteed to be mutually exclusive
  — condition synchronization is realized using wait and signal primitives
  — there exist many variations of wait and signal ...
type buffer(T) = monitor
  var
  slots : array [0..N-1] of T;
  head, tail : 0..N-1;
  size : 0..N;
  notfull, notempty:condition;
procedure deposit(p : T);
  begin
    if size = N then
      notfull.wait
    slots[tail] := p;
    size := size + 1;
    tail := (tail+1) mod N;
    notempty.signal
  end
procedure fetch(var it : T);
  begin
    if size = 0 then
      notempty.wait
    it := slots[head];
    size := size - 1;
    head := (head+1) mod N;
    notfull.signal
  end
Problems with monitors

Monitors are more structured than semaphores, but they are still tricky to program:
— Conditions must be manually checked
— Simultaneous signal and return is not supported

A signalling process is temporarily suspended to allow waiting processes to enter!
— Monitor state may change between signal and resumption of signaler
— Unlike with semaphores, multiple signals are not saved
— Nested monitor calls must be specially handled to prevent deadlock
Path Expressions

Path expressions express the *allowable sequence of operations* as a kind of regular expression:

```markdown
buffer : (put; get)*
```

Although they elegantly express solutions to many problems, path expressions are too limited for general concurrent programming.
Message passing combines communication and synchronization:

> The sender specifies the *message and a destination*
  — a process, a port, a set of processes, ...

> The receiver specifies *message variables and a source*
  — source may or may not be explicitly identified

> Message transfer may be:
  — *asynchronous*: send operations never block
  — *buffered*: sender *may block if the buffer is full*
  — *synchronous*: sender and receiver *must both be ready*
Send and Receive

*In CSP and Occam, source and destination are explicitly named:*

OLVE buffer(CHAN OF INT give, take, signal)
...
SEQ
  numitems := 0 ...
WHILE TRUE
ALT
  numitems ≤ size & give?thebuffer[inindex]
    SEQ
      numitems := numitems + 1
      inindex := (inindex + 1) REM size
  numitems > 0 & signal?any
    SEQ
      take!thebuffer[outindex]
      numitems := numitems - 1
      outindex := (outindex + 1) REM size

NB: The consumer must signal!any to inform the buffer that it wishes to take?avalue
In Ada, the caller identity need not be known in advance:

```
task body buffer is ...
begin loop
    select
        when no_of_items < size =>
            accept give(x : in item) do
                the_buffer(in_index) := x;
            end give;
            no_of_items := no_of_items + 1; ...
        or
        when no_of_items > 0 =>
            accept take(x : out item) do
                x := the_buffer(out_index);
            end take;
            no_of_items := no_of_items - 1; ...
    end select;
end loop; ...
```
What you should know!

- Why do we need concurrent programs?
- What problems do concurrent programs introduce?
- What are safety and liveness?
- What is the difference between deadlock and starvation?
- How are concurrent processes created?
- How do processes communicate?
- Why do we need synchronization mechanisms?
- How do monitors differ from semaphores?
- In what way are monitors equivalent to message-passing?
Can you answer these questions?

> What is the difference between concurrency and parallelism?
> When does it make sense to use busy-waiting?
> Are binary semaphores as good as counting semaphores?
> How could you implement a semaphore using monitors?
> How would you implement monitors using semaphores?
> What problems could nested monitors cause?
> Is it better when message passing is synchronous or asynchronous?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/