Concurrency 1

Shared Memory

Jean-Jacques Lévy (INRIA - Rocq)

MPRI concurrency course with :

Pierre-Louis Curien (PPS)
Eric Goubault (CEA)
James Leifer (INRIA - Rocq)
Catuscia Palamidessi (INRIA - Futurs)
Why concurrency?

1. Programs for multi-processors
2. Drivers for slow devices
3. Human users are concurrent
4. Distributed systems with multiple clients
5. Reduce latency
6. Increase efficiency, but Amdahl’s law

\[ S = \frac{N}{b \times N + (1 - b)} \]

\( (S = \text{speedup}, \ b = \text{sequential part}, \ N = \text{processors}) \)
# MPRI concurrency course

<table>
<thead>
<tr>
<th>Date</th>
<th>Speaker</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-30</td>
<td>JJJ</td>
<td>shared memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>atomicity, SOS</td>
</tr>
<tr>
<td>10-07</td>
<td>JJJ</td>
<td>shared memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>readers/writers, 5 philosophers</td>
</tr>
<tr>
<td>10-12</td>
<td>PLC</td>
<td>CCS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>choice, strong bisim.</td>
</tr>
<tr>
<td>10-21</td>
<td>PLC</td>
<td>CCS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>weak bisim., examples</td>
</tr>
<tr>
<td>10-28</td>
<td>PLC</td>
<td>CCS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>obs. equivalence, Hennessy-Milner logic</td>
</tr>
<tr>
<td>11-04</td>
<td>PLC</td>
<td>CCS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>examples of proofs</td>
</tr>
<tr>
<td>11-16</td>
<td>JL</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>syntax, lts, examples, strong bisim.</td>
</tr>
<tr>
<td>11-25</td>
<td>JL</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>red. semantics, weak bisim., congruence</td>
</tr>
<tr>
<td>12-02</td>
<td>JL</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extensions for mobility</td>
</tr>
<tr>
<td>12-09</td>
<td>JL/CP</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encodings : $\lambda$-calculus, arithm., lists</td>
</tr>
<tr>
<td>12-16</td>
<td>CP</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressivity</td>
</tr>
<tr>
<td>01-06</td>
<td>CP</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stochastic models</td>
</tr>
<tr>
<td>01-13</td>
<td>CP</td>
<td>$\pi$-calculus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>security</td>
</tr>
<tr>
<td>01-20</td>
<td>EG</td>
<td>true concurrency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>concurrency and causality</td>
</tr>
<tr>
<td>01-27</td>
<td>EG</td>
<td>true concurrency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Petri nets, events struct., async. trans.</td>
</tr>
<tr>
<td>02-03</td>
<td>EG</td>
<td>true concurrency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other models</td>
</tr>
<tr>
<td>02-10</td>
<td>all</td>
<td>exercices</td>
</tr>
<tr>
<td>02-17</td>
<td></td>
<td>exam</td>
</tr>
</tbody>
</table>

Concurrency $\Rightarrow$ non-determinism

Suppose $x$ is a global variable. At beginning, $x = 0$

Consider

$S = [x := 1;]
\quad T = [x := 2;]

After $S || T$, then $x \in \{1, 2\}$

Conclusion :

Result is not unique.

Concurrent programs are not described by functions.
Implicit Communication

Suppose $x$ is a global variable. At beginning, $x = 0$

Consider

$S = [x := x + 1; x := x + 1 || x := 2 * x]$

$T = [x := x + 1; x := x + 1 || \text{wait } (x = 1); x := 2 * x]$

After $S$, then $x \in \{2, 3, 4\}$
After $T$, then $x \in \{3, 4\}$
$T$ may be blocked

Conclusion

In $S$ and $T$, interaction via $x$
Input-output behaviour

Suppose $x$ is a global variable.

Consider

$S = [x := 1]
T = [x := 0; x := x + 1]

S$ and $T$ same functions on memory state.

But $S || S$ and $T || S$ are different “functions” on memory state.

⇒ Interaction is important.

A process is an “atomic” action, followed by a process. Ie.

$P \approx Null + 2^{\text{action}} \times P$

Part of the concurrency course gives sense to this equation.
Atomicity

Suppose $x$ is a global variable. At beginning, $x = 0$

Consider

$S = [x := x + 1 || x := x + 1]$

After $S$, then $x = 2$.

However if

$[x := x + 1]$ compiled into $[A := x + 1; x := A]$

Then

$S = [A := x + 1; x := A] || [B := x + 1; x := B]$

After $S$, then $x \in \{1, 2\}$.

Conclusion

1. $[x := x + 1]$ was firstly considered atomic
2. Atomicity is important
Critical section – Mutual exclusion

Let $P_0 = [\cdots; C_0; \cdots]$ and $P_1 = [\cdots; C_1; \cdots]$.

$C_0$ and $C_1$ are critical sections (ie should not be executed simultaneously).

Solution 1 At beginning, $\text{turn} = 0$.

$P_0 : \cdots$

\[
\text{while turn} \neq 0 \text{ do} \\
; \\
C_0; \\
\text{turn} := 1; \\
\cdots
\]

$P_1 : \cdots$

\[
\text{while turn} \neq 1 \text{ do} \\
; \\
C_1; \\
\text{turn} := 0; \\
\cdots
\]

$P_0$ privileged, unfair.
Critical section – Mutual exclusion

Solution 2 At beginning, \(a_0 = a_1 = \text{false}\).

\begin{align*}
P_0 & : \quad \cdots \quad \text{P1 : } \quad \cdots \\
& \quad \text{while } a_1 \text{ do} \quad \text{while } a_0 \text{ do} \\
& \quad \quad ; \quad \quad ; \\
& \quad a_0 := \text{true;} \quad a_1 := \text{true;} \\
& \quad C_0; \quad C_1; \\
& \quad a_0 := \text{false;} \quad a_1 := \text{false;} \\
& \quad \cdots \quad \cdots 
\end{align*}

False.

Solution 3 At beginning, \(a_0 = a_1 = \text{false}\).

\begin{align*}
P_0 & : \quad \cdots \\
& \quad a_0 := \text{true;} \\
& \quad \text{while } a_1 \text{ do} \\
& \quad \quad ; \\
& \quad C_0; \\
& \quad a_0 := \text{false;} \\
& \quad \cdots \\

\text{P1 : } \quad \cdots \\
& \quad a_1 := \text{true;} \\
& \quad \text{while } a_0 \text{ do} \\
& \quad \quad ; \\
& \quad C_1; \\
& \quad a_1 := \text{false;} \\
& \quad \cdots 
\end{align*}

Deadlock. Both \(P_0\) and \(P_1\) blocked.
Dekker’s Algorithm (CACM 1965)

At beginning, $a_0 = a_1 = \text{false}$, turn $\in \{0, 1\}$

P0 : ⋯
    a0 := true;
    while a1 do
        if turn != 0 begin
            a0 := false;
            while turn != 0 do
                ;
            a0 := true;
        end;
    C0;
    turn := 1; a0 := false;
    ⋯

P1 : ⋯
    a1 := true;
    while a0 do
        if turn != 1 begin
            a1 := false;
            while turn != 1 do
                ;
            a1 := true;
        end;
    C1;
    turn := 0; a1 := false;
    ⋯

Exercice 1 Trouver Dekker pour $n$ processus [Dijkstra 1968].
Peterson’s Algorithm (IPL June 81) (1/5)

At beginning, \( a_0 = a_1 = \text{false} \), \( \text{turn} \in \{0, 1\} \)

\[
P_0 : \quad \cdots \quad a_0 := \text{true}; \\
\quad \text{turn} := 1; \\
\quad \text{while } a_1 && \text{turn} \neq 0 \text{ do} \\
\quad \quad \cdots \\
\quad C_0; \\
\quad a_0 := \text{false}; \\
\quad \cdots
\]

\[
P_1 : \quad \cdots \quad a_1 := \text{true}; \\
\quad \text{turn} := 0; \\
\quad \text{while } a_0 && \text{turn} \neq 1 \text{ do} \\
\quad \quad \cdots \\
\quad C_1; \\
\quad a_0 := \text{false}; \\
\quad \cdots
\]
Peterson's Algorithm (IPL June 81) (2/5)

c₀, c₁ program counters for P₀ and P₁.
At beginning c₀ = c₁ = 1

... {¬a₀ ∧ c₀ ≠ 2}
1 a₀ := true; c₀ := 2;
   {a₀ ∧ c₀ = 2}
2 turn := 1; c₀ := 1;
   {a₀ ∧ c₀ ≠ 2}
3 while a₀ && turn != 0 do
   . ;
   {a₀ ∧ c₀ ≠ 2 ∧ (¬a₁ ∨ turn = 0 ∨ c₁ = 2)}
   . C₀;
5 a₀ := false;
   {¬a₀ ∧ c₀ ≠ 2}
   ...

... {¬a₁ ∧ c₁ ≠ 2}
a₁ := true; c₁ := 2;
   {a₁ ∧ c₁ = 2}
turn := 0; c₁ := 1;
   {a₁ ∧ c₁ ≠ 2}
while a₀ && turn != 1 do
   . ;
   {a₁ ∧ c₁ ≠ 2 ∧ (¬a₁ ∨ turn = 1 ∨ c₀ = 2)}
   . C₁;
12 a₁ := false;
   {¬a₁ ∧ c₁ ≠ 2}
   ...
Peterson’s Algorithm (IPL June 81) (3/5)

\[(\text{turn} = 0 \lor \text{turn} = 1) \land a_0 \land c_0 \neq 2 \land (\neg a_1 \lor \text{turn} = 0 \lor c_1 = 2) \land a_1 \land c_1 \neq 2 \land (\neg a_0 \lor \text{turn} = 1 \lor c_0 = 2) \equiv (\text{turn} = 0 \lor \text{turn} = 1) \land \text{tour} = 0 \land \text{tour} = 1 \quad \text{Impossible}\]
Peterson's Algorithm (IPL June 81) (4/5)

...  
\{¬a_0 ∧ c_0 ≠ 2\}  
1 a0 := true; c0 := 2;  
\{a_0 ∧ c_0 = 2\}  
2 turn := 1; c0 := 1;  
\{a_0 ∧ c_0 ≠ 2\}  
3 while a1 && turn != 0 do  
  ;  
  \{a0 ∧ c0 ≠ 2 ∧ (¬a1 ∨ turn = 0 ∨ c1 = 2)\}  
  . C_0;  
5 a0 := false;  
\{¬a_0 ∧ c_0 ≠ 2\}  
...

...
\{¬a_1 ∧ c_1 ≠ 2\}  
a1 := true; c1 := 2;  
\{a_1 ∧ c_1 = 2\}  
turn := 0; c1 := 1;  
\{a_1 ∧ c_1 ≠ 2\}  
while a0 && turn != 1 do  
  ;  
  \{a1 ∧ c1 ≠ 2 ∧ (¬a1 ∨ turn = 1 ∨ c0 = 2)\}  
  . C_1;  
a1 := false;  
\{¬a_1 ∧ c_1 ≠ 2\}  
...

\(c_0, c_1\) program counters for \(P_0\) and \(P_1\).  
At beginning \(c_0 = c_1 = 1\)
Peterson’s Algorithm (IPL June 81) (5/5)

\[(\text{turn} = 0 \lor \text{turn} = 1) \land a_0 \land c_0 \neq 2 \land (\neg a_1 \lor \text{turn} = 0 \lor c_1 = 2) \land a_1 \land c_1 \neq 2 \land (\neg a_0 \lor \text{turn} = 1 \lor c_0 = 2) \equiv (\text{turn} = 0 \lor \text{turn} = 1) \land \text{tour} = 0 \land \text{tour} = 1 \quad \text{Impossible}\]
Synchronization

Concurrent/Distributed algorithms

1. Lamport: barber, baker, ...
2. Dekker’s algorithm for $P_0, P_1, P_N$ (Dijsktra 1968)
3. Peterson is simpler and can be generalised to $N$ processes
4. Proofs? By model checking? With assertions? In temporal logic (eg Lamport’s TLA)?
5. Dekker’s algorithm is too complex
6. Dekker’s algorithm uses busy waiting
7. Fairness acheived because of fair scheduling

Need for higher constructs in concurrent programming.

Exercice 2 Try to define fairness.
Semaphores

A generalised semaphore $s$ is integer variable with 2 operations

$acquire(s)$ : If $s > 0$ then $s := s - 1$
Otherwise be suspended on $s$.

$release(s)$ : If some process is suspended on $s$, wake it up
Otherwise $s := s + 1$.

Now mutual exclusion is easy:

At beginning, $s = 1$. Then

$$[\cdots; acquire(s); A; release(s); \cdots] \ || \ [\cdots; acquire(s); B; release(s); \cdots]$$

Exercice 3 Other definition for semaphore:

$acquire(s)$ : If $s > 0$ then $s := s - 1$. Otherwise restart.

$release(s)$ : Do $s := s + 1$.

Are these definitions equivalent?
Operational semantics (seq. part)

Language

\[
P, Q ::= \text{skip} \mid x := e \mid \text{if } b \text{ then } P \text{ else } Q \mid P; Q \mid \text{while } b \text{ do } P \mid \bullet
\]

\[
e ::= \text{expression}
\]

Semantics (SOS)

\[
\langle \text{skip}, \sigma \rangle \rightarrow \langle \bullet, \sigma \rangle
\]

\[
\sigma(e) = \text{true}
\]

\[
\langle \text{if } e \text{ then } P \text{ else } Q, \sigma \rangle \rightarrow \langle P, \sigma \rangle
\]

\[
\langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle \quad (P' \neq \bullet)
\]

\[
\langle P; Q, \sigma \rangle \rightarrow \langle P'; Q, \sigma' \rangle
\]

\[
\sigma(e) = \text{false}
\]

\[
\langle \text{if } e \text{ then } P \text{ else } Q, \sigma \rangle \rightarrow \langle Q, \sigma \rangle
\]

\[
\langle P, \sigma \rangle \rightarrow \langle \bullet, \sigma' \rangle
\]

\[
\langle P; Q, \sigma \rangle \rightarrow \langle Q, \sigma' \rangle
\]

\[
\langle \text{while } e \text{ do } P, \sigma \rangle \rightarrow \langle P; \text{while } e \text{ do } P, \sigma \rangle
\]

\[
\text{\sigma} \in \text{Variables} \mapsto \text{Values}
\]

\[
\sigma[v/x](x) = v
\]

\[
\sigma[v/x](y) = \sigma(y) \text{ if } y \neq x
\]
Operational semantics (parallel part)

Language

\[ P, Q ::= \ldots \mid P \parallel Q \mid \text{wait } b \mid \text{await } b \text{ do } P \]

Semantics (SOS)

\[
\begin{align*}
\langle P, \sigma \rangle &\rightarrow \langle P', \sigma' \rangle \\
\langle P \parallel Q, \sigma \rangle &\rightarrow \langle P', \parallel Q, \sigma' \rangle \\
\langle Q, \sigma \rangle &\rightarrow \langle Q', \sigma' \rangle \\
\langle P \parallel Q, \sigma \rangle &\rightarrow \langle P \parallel Q', \sigma' \rangle \\
\langle \cdot \parallel \cdot, \sigma \rangle &\rightarrow \langle \cdot, \sigma \rangle \\
\sigma(e) = \text{true} &\rightarrow \langle \text{wait } e, \sigma \rangle \rightarrow \langle \cdot, \sigma \rangle \\
\sigma(e) = \text{true} &\rightarrow \langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle \\
\text{await } e \text{ do } P, \sigma &\rightarrow \langle P', \sigma' \rangle
\end{align*}
\]

Exercice 4   Complete SOS for \( e \) and \( v \)

Exercice 5   Find SOS for boolean semaphores.

Exercice 6   Avoid spurious silent steps in if, while and \( \parallel \).
SOS reductions

Notations

\[ \langle P_0, \sigma_0 \rangle \rightarrow \langle P_1, \sigma_1 \rangle \rightarrow \langle P_2, \sigma_2 \rangle \rightarrow \cdots \langle P_n, \sigma_n \rangle \rightarrow \]

We write

\[ \langle P_0, \sigma_0 \rangle \rightarrow^* \langle P_n, \sigma_n \rangle \text{ when } n \geq 0, \]
\[ \langle P_0, \sigma_0 \rangle \rightarrow^+ \langle P_n, \sigma_n \rangle \text{ when } n > 0. \]

Remark that in our system, we have no rule such as

\[ \sigma(e) = \text{false} \]
\[ \langle \text{wait } e, \sigma \rangle \rightarrow \langle \text{wait } b, \sigma \rangle \]

Ie no busy waiting. Reductions may block. (Same remark for \text{await } e \text{ do } P).
Atomic statements (Exercises)

Exercice 7 If we make following extension

\[ P, Q ::= \ldots \mid \{ P \} \]

what is the meaning of following rule?

\[
\begin{align*}
\langle P, \sigma \rangle & \rightarrow^+ \langle \bullet, \sigma' \rangle \\
\langle \{ P \}, \sigma \rangle & \rightarrow \langle \bullet, \sigma' \rangle
\end{align*}
\]

Exercice 8 Show \textit{await } e \textit{ do } P \equiv \{ \text{wait } e; P \}

Exercice 9 Code generalized semaphores in our language.

Exercice 10 Meaning of \{\textit{while true do skip} \}? Find simpler equivalent statement.

Exercice 11 Try to add procedure calls to our SOS semantics.
Producer - Consumer
A typical thread package. Modula-3

INTERFACE Thread;

TYPE
  T <: ROOT;
  Mutex = MUTEX;
  Condition <: ROOT;

A Thread.T is a handle on a thread. A Mutex is locked by some thread, or unlocked. A Condition is a set of waiting threads. A newly-allocated Mutex is unlocked; a newly-allocated Condition is empty. It is a checked runtime error to pass the NIL Mutex, Condition, or T to any procedure in this interface.
PROCEDURE Acquire(m: Mutex);

Wait until m is unlocked and then lock it.

PROCEDURE Release(m: Mutex);

The calling thread must have m locked. Unlocks m.

PROCEDURE Wait(m: Mutex; c: Condition);

The calling thread must have m locked. Atomically unlocks m and waits on c. Then relocks m and returns.

PROCEDURE Signal(c: Condition);

One or more threads waiting on c become eligible to run.

PROCEDURE Broadcast(c: Condition);

All threads waiting on c become eligible to run.
A LOCK statement has the form:

```
LOCK mu DO S END
```

where S is a statement and mu is an expression. It is equivalent to:

```
WITH m = mu DO
    Thread.Acquire(m);
    TRY S FINALLY Thread.Release(m) END
END
```

where m stands for a variable that does not occur in S.
A statement of the form:

\[
\text{TRY } S_1 \text{ FINALLY } S_2 \text{ END}
\]

executes statement \( S_1 \) and then statement \( S_2 \). If the outcome of \( S_1 \) is normal, the TRY statement is equivalent to \( S_1 ; S_2 \). If the outcome of \( S_1 \) is an exception and the outcome of \( S_2 \) is normal, the exception from \( S_1 \) is re-raised after \( S_2 \) is executed. If both outcomes are exceptions, the outcome of the TRY is the exception from \( S_2 \).
Concurrent stack

Popping in a stack :

VAR nonEmpty := NEW(Thread.Condition);

LOCK m DO
    WHILE p = NIL DO Thread.Wait(m, nonEmpty) END;
    topElement := p.head;
    p := p.next;
END;
return topElement;

Pushing into a stack :

LOCK m DO
    p = newElement(v, p);
    Thread.Signal (nonEmpty);
END;

Caution: WHILE is safer than IF in Pop.
Concurrent table

VAR table := ARRAY [0..999] of REFANY {NIL, ...};
VAR i:[0..1000] := 0;

PROCEDURE Insert (r: REFANY) =
    BEGIN
        IF r <> NIL THEN
            table[i] := r;
            i := i+1;
        END;
    END Insert;

Exercice 12  Complete previous program to avoid lost values.
Deadlocks

Thread $A$ locks mutex $m_1$
Thread $B$ locks mutex $m_2$
Thread $A$ trying to lock $m_2$
Thread $B$ trying to lock $m_1$

Simple strategy for semaphore controls

Respect a partial order between semaphores. For example, $A$ and $B$ uses $m_1$ and $m_2$ in same order.
Conditions and semaphores

Semaphores are stateful; conditions are stateless.

\[
\text{Wait (m, c) :} \\
\quad \text{release}(m); \\
\quad \text{acquire}(c\text{-sem}); \\
\quad \text{acquire}(m); \\
\]

\[
\text{Signal (c) :} \\
\quad \text{release}(c\text{-sem});
\]

**Exercice 13** Is this translation correct?

**Exercice 14** What happens in Wait and Signal if it does not atomically unlock \( m \) and wait on \( c \).
Exercices

Exercice 15  Readers and writers. A buffer may be read by several processes at same time. But only one process may write in it. Write procedures StartRead, EndRead, StartWrite, EndWrite.

Exercice 16  Give SOS for operations on conditions.