Concurrency 1
Shared Memory

Catuscia Palamidessi
INRIA Futurs and LIX - Ecole Polytechnique

The other lecturers for this course:

Jean-Jacques Lévy (INRIA Rocquencourt)
James Leifer (INRIA Rocquencourt)
Eric Goubault (CEA)

http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2005/
Outline

1. Motivation
2. Overview of the course
3. Concurrency in Shared Memory: Effects and Issues
4. Critical Sections and Mutual Exclusion
   - Some attempts to implement a critical section
   - Some famous algorithms
   - Semaphores
   - The dining philosophers
   - Exercises
Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl’s law

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

\( (S = \text{speedup}, \ b = \text{sequential part}, \ N \ \text{processors}) \)
### Overview of the course

<table>
<thead>
<tr>
<th>Date</th>
<th>Speaker</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-28</td>
<td>CP</td>
<td>Shared memory: atomicity</td>
</tr>
<tr>
<td>10-05</td>
<td>CP/JJL</td>
<td>Shared memory: verification, report on Ariane 501</td>
</tr>
<tr>
<td>10-12</td>
<td>CP</td>
<td>CCS: syntax and transitions, coinduction</td>
</tr>
<tr>
<td>10-19</td>
<td>CP</td>
<td>CCS: weak and strong bisimulations, axiomatization</td>
</tr>
<tr>
<td>10-26</td>
<td>CP</td>
<td>CCS: examples, Hennessy-Milner logic</td>
</tr>
<tr>
<td>11-02</td>
<td>JL</td>
<td>$\pi$-calculus: syntax; reduction, transitions, strong bisimulation</td>
</tr>
<tr>
<td>11-09</td>
<td>JL</td>
<td>$\pi$-calculus: sum, abstractions, data structures, bisimulation proofs</td>
</tr>
<tr>
<td>11-16</td>
<td>JL</td>
<td>$\pi$-calculus: bisimulation “up to”, congruence, barbed bisimulation</td>
</tr>
<tr>
<td>11-23</td>
<td>Review</td>
<td></td>
</tr>
<tr>
<td>11-30</td>
<td>MT exam</td>
<td></td>
</tr>
<tr>
<td>12-07</td>
<td>JL</td>
<td>$\pi$-calculus: comparison between equivalences</td>
</tr>
<tr>
<td>12-14</td>
<td>JJL</td>
<td>Expressivity of the pi-calculus and its variants</td>
</tr>
<tr>
<td>12-21</td>
<td>vacation</td>
<td></td>
</tr>
<tr>
<td>12-28</td>
<td>vacation</td>
<td></td>
</tr>
<tr>
<td>01-04</td>
<td>JJJ</td>
<td>Distributed pi-calculus</td>
</tr>
<tr>
<td>01-11</td>
<td>JJJ</td>
<td>Problems with distributed implementation</td>
</tr>
<tr>
<td>01-18</td>
<td>EG</td>
<td>True concurrency versus interleaving semantics</td>
</tr>
<tr>
<td>01-25</td>
<td>EG</td>
<td>Event structures and Petri nets</td>
</tr>
<tr>
<td>02-01</td>
<td>EG</td>
<td>Application to the semantics of CCS</td>
</tr>
<tr>
<td>02-08</td>
<td>EG</td>
<td>Comparison of the expressiveness of different models</td>
</tr>
<tr>
<td>02-15</td>
<td>Review</td>
<td></td>
</tr>
<tr>
<td>02-22</td>
<td>Final exam</td>
<td></td>
</tr>
</tbody>
</table>
Note: we assume that the update of a variable is *atomic*

Let $x$ be a global variable. Assume that at the beginning $x = 0$

Consider two simple processes

$S = [x := 1;]$ and $T = [x := 2;]$

After the execution of $S || T$, we have $x \in \{1, 2\}$

Conclusion:

- Result is not unique.
- Concurrent programs are not described by functions.
Implicit Communication

- Let \( x \) be a global variable. Assume that at the beginning \( x = 0 \)
- Consider the two processes

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

- After the execution of \( S \), we have \( x \in \{2, 3, 4\} \)
- After the execution of \( T \), we have \( x \in \{3, 4\} \)
- \( T \) may be blocked
- Conclusion: The parallel subcomponents of a program may interact via their shared variables
Input-output behavior

- Let $x$ be a global variable.
- Consider the two processes
  
  \[ S = [x := 1] \quad \text{and} \quad T = [x := 0; x := x + 1] \]

- $S$ and $T$ are the same function on memory state.
- However, $S \parallel S$ and $T \parallel S$ are different “functions” on memory state.

- A process is an \textit{atomic action}, followed by a process:
  
  \[ \mathcal{P} \simeq \text{Null} + 2^{\text{action}} \times \mathcal{P} \]

- Part of the concurrency course aims at giving sense to this equation.
Atomicity

- Let $x$ be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \ || \ x := x + 1]$
- After the execution of $S$ we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, $S$ may behave as $[A := x + 1; x := A] \ || \ [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*
Some attempts to implement a critical section

The problem

- Let \( P_0 = [\cdots; C_0; \cdots] \) and \( P_1 = [\cdots; C_1; \cdots] \)

- We intent \( C_0 \) and \( C_1 \) to be critical sections, i.e. they should not be executed simultaneously.
Some attempts to implement a critical section

**Attempt n.1**

- Use a variable $\textit{turn}$. At beginning, $\textit{turn} = 0$.

- However the method is unfair, because $P_0$ is privileged. Worse yet, until $P_0$ executes its critical section, $P_1$ is blocked.

```c
P0
...;
  while turn != 0 do ;
  C0;
  turn := 1;
  ...

P1
...;
  while turn != 1 do ;
  C1;
  turn := 0;
  ...
```
Some attempts to implement a critical section

**Attempt n.2**

- Use two boolean variables $a_0$, $a_1$.
  - At beginning, $a_0 = a_1 = \text{false}$.

```
P0
...;
while a1 do ;
a0 := true ;
C0;
a0 := false ;
...
```

```
P1
...;
while a0 do ;
a1 := true ;
C1;
a1 := false ;
...
```

- Incorrect. It does not ensure mutual exclusion.
Some attempts to implement a critical section

**Attempt n.3**

- Use two boolean variables $a_0, a_1$.
  - At beginning, $a_0 = a_1 = \text{false}$.

```
P0
...;
  a0 := true;
  while a1 do;
  a0 := true;
  C0;
  a0 := false;
  ...

P1
...;
  a1 := true;
  while a0 do;
  a1 := true;
  C1;
  a1 := false;
  ...
```

- We may get a deadlock. Both $P_0$ and $P_1$ may block.
Some famous algorithms

Dekker’s Algorithm (early Sixties)

- The first correct mutual exclusion algorithm
- Use both the variable $\text{turn}$ and the boolean variables $a_0$ and $a_1$. At beginning, $a_0 = a_1 = \text{false}$, $\text{turn} \in \{0, 1\}$

A variant of Dekker’s algorithm for the case of $n$ processes was presented by Dijkstra (CACM 1965).
Some famous algorithms

Peterson's Algorithm (IPL 1981)

- The simplest and most compact mutual exclusion algorithm in literature
- Use both the variable \textit{turn} and the boolean variables \(a_0\) and \(a_1\). At beginning, \(a_0 = a_1 = \text{false}\), \(\text{turn} \in \{0, 1\}\)

\begin{align*}
\text{P0} & \quad \ldots; \\
& \quad a0 := \text{true} \; ; \\
& \quad \text{turn} := 1; \\
& \quad \text{while} \; a1 \; \text{and} \; \text{turn} \neq 0 \; \text{do} \; ; \\
& \quad \text{C0;} \\
& \quad a0 := \text{false} \; ; \\
& \quad \ldots \\
\text{P1} & \quad \ldots; \\
& \quad a1 := \text{true} \; ; \\
& \quad \text{turn} := 0; \\
& \quad \text{while} \; a0 \; \text{and} \; \text{turn} \neq 1 \; \text{do} \; ; \\
& \quad \text{C1;} \\
& \quad a1 := \text{false} \; ; \\
& \quad \ldots
\end{align*}
To show the correctness it is convenient to add two variables, $pc_0$, $pc_1$, which represent a sort of program counters for $P_0$ and $P_1$. At beginning $pc_0 = pc_1 = 1$

P0

...;
{\neg a_0 \land pc_0 \neq 2}
\begin{align*}
a_0 & := \text{true} \ ; pc0 := 2; \\
\{ a_0 \land pc_0 = 2 \}
\end{align*}
turn := 1; pc0 := 1;
{ a_0 \land pc_0 \neq 2}
\begin{align*}
\text{while } a1 \text{ and turn }\neq 0 \text{ do } ; \\
\{ a0 \land pc_0 \neq 2 \land (\neg a_1 \lor turn = 0 \lor pc_1 = 2) \}
\end{align*}
C0;
\begin{align*}
a0 & := \text{false} \ ; \\
\{ \neg a_0 \land pc_0 \neq 2 \} \\
\end{align*}
...

P1

...;
{\neg a_1 \land pc_1 \neq 2}
\begin{align*}
a1 & := \text{true} \ ; pc1 := 2; \\
\{ a_1 \land pc_1 = 2 \}
\end{align*}
turn := 0; pc1 := 1;
{ a_1 \land pc_1 \neq 2}
\begin{align*}
\text{while } a0 \text{ and turn }\neq 1 \text{ do } ; \\
\{ a1 \land pc_1 \neq 2 \land (\neg a_0 \lor turn = 1 \lor pc_0 = 2) \}
\end{align*}
C1;
\begin{align*}
a1 & := \text{false} \ ; \\
\{ \neg a_1 \land pc_1 \neq 2 \} \\
\end{align*}
...
Correctness of Peterson’s Algorithm (2/2)

We can prove the correctness by contradiction. If both programs were in their critical section, then the formulas

\[
\{ a_0 \land pc_0 \neq 2 \land (\neg a_1 \lor turn = 0 \lor pc_1 = 2) \} \quad \text{and} \quad \{ a_1 \land pc_1 \neq 2 \land (\neg a_0 \lor turn = 1 \lor pc_0 = 2) \}
\]

should be true at the same time, but:

\[
\begin{align*}
    a_0 & \land pc_0 \neq 2 \land (\neg a_1 \lor turn = 0 \lor pc_1 = 2) \\
    \land \quad a_1 & \land pc_1 \neq 2 \land (\neg a_0 \lor turn = 1 \lor pc_0 = 2) \\
\equiv \quad turn & = 0 \land turn = 1
\end{align*}
\]

Contradiction!
Some famous algorithms

Synchronization in Concurrent/Distributed algorithms

- Dekker’s algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to $N$ processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs? By model checking? With assertions? In temporal logic (eg Lamport’s TLA)?

Need for higher constructs in concurrent programming.
Semaphores

A generalized semaphore $s$ is an integer variable with 2 operations

- **Acquire** ($\text{acquire}(s)$): If $s > 0$ then $s := s - 1$, otherwise suspend on $s$. (atomically)

- **Release** ($\text{release}(s)$): If some process is suspended on $s$, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$[\cdots; \text{acquire}(s); C_0; \text{release}(s); \cdots] \parallel [\cdots; \text{acquire}(s); C_1; \text{release}(s); \cdots]$

**Question** Consider another definition for semaphore:

- **Acquire** ($\text{acquire}(s)$): If $s > 0$ then $s := s - 1$. Otherwise restart.
- **Release** ($\text{release}(s)$): Do $s := s + 1$.

Are these definitions equivalent?
The dining philosophers

Problem proposed by Dijkstra for testing concurrency primitives

5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.

Desiderata

- if one philosopher gets hungry, some philosopher will eventually eat (progress)
- if one philosopher gets hungry, he will eventually eat (starvation-freedom)
Exercises

- (Difficult) Generalize Dekker’s algorithm to the case of $n$ processes
- Generalize Petersons’s algorithm to the case of $n$ processes
- Implement the Semaphore in Java
- Write a program for the dining philosophers which ensure progress
- Discuss how to modify the solution so to ensure starvation-freedom
- Problem: A certain file is shared by some Reader and some Writer processes: we want that only one writer can write on the file at a time, while the readers are allowed to do it concurrently. Write the code for the Reader and the Writer.