Concurrent 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/

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 = speedup, b = sequential part, N processors)

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

Overview of the course

09:28 CP Shared memory: atomism
09:55 CP,JL Shared memory: verification, report on Ariane 501
10:12 CP ESS, syntax and semantics, construction
10:19 CP CCS, weak and strong bisimulations, axiomatization
10:26 CP CCS examples, Hennessy-Milner logic
11:02 JI \epsilon-calculus: syntax, reduction, transduction, strong bisimulation
11:09 JL \epsilon-calculus: sum, abstractions, data structures, bisimulation proofs
11:16 JL \epsilon-calculus: bisimulation up to, congruence, barred bisimulation
11:23 Review
11:31 JL Examen
12:07 JL \epsilon-calculus: comparison between equivalences
12:14 JL Expressivity of the \epsilon-calculus and its variants
12:21 
12:38 
01:04 JL Distributed \epsilon-calculus
01:11 JL Problems with distributed implementation
01:18 EG True concurrency versus interleaving semantics
01:25 EG Event structures and Petri nets
02:01 EG Application to the semantics of CCS
02:06 EG Comparison of the expressiveness of different models
02:15 Review
02:22 Final exam
Non-determinism

• 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 $S \parallel T$, we have $x \in \{1, 2\}$
• Conclusion:
  • Result is not unique.
  • Concurrent programs are not described by functions.

Input-output behavior

• Let $x$ be a global variable.
• Consider the two processes
  $S = [x ::= 1]$ and $T = [x ::= 0; x ::= x + 1]$
• $S$ and $T$ are the same function on memory state.
• However, $S \parallel S$ and $T \parallel T$ are different “functions” on memory state.
• A process is an atomic action, followed by a process:
  $P \Rightarrow \text{null} + 2^{\text{action}} \times P$
• Part of the concurrency course aims at giving sense to this equation.

Implicit Communication

• Let $x$ be a global variable. Assume that at the beginning $x = 0$
• Consider the two processes
  $S = [x ::= x + 1; x ::= x + 1 \parallel x ::= 2 + x]$
  $T = [x ::= x + 1; x ::= x + 1 \parallel \text{wait}(x - 1); x ::= 2 + x]$
• 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.

Atomicity

• Let $x$ be a global variable. Assume that at beginning $x = 0$
• Consider the process $S = [x ::= x + 1 \parallel x ::= x + 1]$
• 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 $[A ::= x + 1; x ::= A] \parallel [B ::= x + 1; x ::= B]$
• 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
The problem

- Let $P_0 = [\cdots; C_0; \cdots]$ and $P_1 = [\cdots; C_1; \cdots]$.
- We intend $C_0$ and $C_1$ to be critical sections, i.e., they should not be executed simultaneously.

Attempt n.1

- Use a variable `turn`. At beginning, `turn = 0`.

```
P0
...;
while turn != 0 do:
    C0;
    turn = 1;
...;
```
```
P1
...;
while turn != 1 do:
    C1;
    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.

Attempt n.2

- Use two boolean variables $a_0, a_1$.
  At beginning, $a_0 = a_1 = false$.

```
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.

Attempt n.3

- Use two boolean variables $a_0, a_1$.
  At beginning, $a_0 = a_1 = false$.

```
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.
Dekker’s Algorithm (early Sixties)

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

```c
... a0 := true; while a0 do
  if turn = 0 begin
    a0 := false, while turn = 0 do ;
    a0 := true; end;
  C0;
  turn := 1; a0 := false;
...
```

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

Peterson’s Algorithm (IPL 1981)

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

```c
... a1 := true; while a0 do
  if turn = 1 begin
    a1 := false, while turn = 1 do ;
    a1 := true; end;
  C1;
  turn := 0; a1 := false;
...
```

Correctness of Peterson’s Algorithm

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 \( pc_0 = pc_1 = 1 \)

```c
P0
... (!a0 \& !pc0) \& (!a1 \& !pc1) \& turn = 1; pc0 := 1; ... a0 := true; pc0 := 2; a0 \& pc0 \& !pc1 := 2; while a1 \& turn = 0 do ;
  (a0 \& pc0 \& !pc1 := 2 \& (a1 \& turn = 0 \& !pc0 := 2));
  C0; a0 := false; (!a0 \& !pc0 := 2)
...
```

```c
P1
... (!a1 \& !pc1) \& (!a0 \& !pc0) \& turn = 1; pc1 := 1; ... a1 := true; pc1 := 2; a1 \& pc1 := 2; while a0 \& turn = 1 do ;
  (a1 \& pc1 := 2 \& (a0 \& turn = 1 \& !pc0 := 2));
  C1; a1 := false; (!a1 \& !pc1 := 2)
...
```

We can prove the correctness by contradiction. If both programs were in their critical section, then the formulas
\((a0 \land pc_0 \neq 2 \land (\neg a_1 \lor turn = 0 \lor pc_1 = 2))\) and
\((a1 \land pc_1 \neq 2 \land (\neg a_0 \lor turn = 1 \lor pc_0 = 2))\) should be true at the same time, but:

\[
a_0 \land pc_0 \neq 2 \land (\neg a_1 \lor turn = 0 \lor pc_1 = 2) \\
\land a_1 \land pc_1 \neq 2 \land (\neg a_0 \lor turn = 1 \lor pc_0 = 2) \\
\iff turn = 0 \land turn = 1
\]

Contradiction!
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($s$): If $s > 0$ then $s := s - 1$, otherwise suspend on $s$.
  (atomically)
- release($s$): If some process is suspended on $s$, wake it up, otherwise $s := s + 1$.
  (atomically)

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

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

Question: Consider another definition for semaphore.
- acquire($s$): If $s > 0$ then $s := s - 1$. Otherwise restart.
- release($s$): Do $s := s + 1$.

Are these definitions equivalent?

Exercises

- (Difficult) Generalize Dekker’s algorithm to the case of $n$ processes.
- Generalize Peterson’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.

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)