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 lattency
6. Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

($S =$ speedup, $b =$ sequential part, N processors)

MPRI concurrency course

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

http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2004/

Concurrency $\Rightarrow$ non-determinism

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

Consider

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

After $S \mid\mid T$, 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

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

$T = [x := x + 1; x := x + 1 \mid\mid$ 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 $S \mid\mid S$ and $T \mid\mid S$ are different "functions" on memory state.

$\Rightarrow$ Interaction is important.

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

$$\mathcal{P} \simeq \mathit{Null} + 2^{action \times \mathcal{P}}$$

Part of the concurrency course gives sense to this equation.

Atomicity

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

Consider

$S = [x := x + 1 \mid\mid x := x + 1]$

After S, then $x = 2$.

However if

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

Then
$S = [A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$

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, *turn* $= 0$.

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

```
P1 : ···
  while turn != 1 do
    ;
  C1;
  turn := 0;
  ···
```

P_0 privileged, unfair.

Critical section – Mutual exclusion

Solution 2 At beginning, $a_0 = a_1 =$ false .

```
P0 : ···                        P1 : ···
  while a1 do                     while a0 do
    ;                               ;
  a0 := true;                     a1 := true;
  C0;                             C1;
  a0 := false;                    a1 := false;
  ···                             ···
```

False.

Solution 3 At beginning, $a_0 = a_1 =$ false .

```
P0 : ···                        P1 : ···
  a0 := true;                     a1 := true;
  while a1 do                     while a0 do
    ;                               ;
  C0;                             C1;
  a0 := false;                    a1 := false;
  ···                             ···
```

Deadlock. Both P_0 and P_1 blocked.

Dekker's Algorithm (CACM 1965)

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

```
P0 : ···                              P1 : ···
  a0 := true;                           a1 := true;
  while a1 do                           while a0 do
    if turn != 0 begin                    if turn != 1 begin
      a0 := false;                          a1 := false;
      while turn != 0 do                    while turn != 1 do
        ;                                     ;
      a0 := true;                           a1 := true;
    end;                                  end;
  C0;                                   C1;
  turn := 1; a0 := false;               turn := 0; a1 := false;
  ···                                   ···
```

Exercice 1 Trouver Dekker pour n processus [Dijkstra 1968].

Peterson's Algorithm (IPL June 81) (1/5)

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

```
P0 : ···                                P1 : ···
  a0 := true;                             a1 := true;
  turn := 1;                              turn := 0;
  while a1 && turn != 0 do                while a0 && turn != 1 do
    ;                                       ;
  C0;                                     C1;
  a0 := false;                            a0 := false;
  ···                                     ···
```


Peterson's Algorithm (IPL June 81) (2/5)

c_0, c_1 program counters for P_0 and P_1.
At beginning $c_0 = c_1 = 1$

```
     ···                                      ···
     {¬a0 ∧ c0 ≠ 2}                           {¬a1 ∧ c1 ≠ 2}
1 a0 := true; c0 := 2;                     a1 := true; c1 := 2;
     {a0 ∧ c0 = 2}                            {a1 ∧ c1 = 2}
2 turn := 1; c0 := 1;                      turn := 0; c1 := 1;
     {a0 ∧ c0 ≠ 2}                            {a1 ∧ c1 ≠ 2}
3 while a1 && turn != 0 do                 while a0 && turn != 1 do
. ;                                          ;
     {a0 ∧ c0 ≠ 2 ∧ (¬a1 ∨ turn = 0 ∨ c1 = 2)} {a1 ∧ c1 ≠ 2 ∧ (¬a1 ∨ turn = 1 ∨ c0 = 2)}
. C0;                                      C1;
5 a0 := false;                             a1 := false;
     {¬a0 ∧ c0 ≠ 2}                           {¬a1 ∧ c1 ≠ 2}
     ···                                      ···
```

Peterson's Algorithm (IPL June 81) (3/5)

$$
\begin{array}{ll}
 & (turn = 0 \vee turn = 1) \\
\wedge & a_0 \ \wedge \ c_0 \neq 2 \ \wedge \ (\neg a_1 \vee turn = 0 \vee c_1 = 2) \wedge a_1 \ \wedge \ c_1 \neq 2 \ \wedge \ (\neg a_0 \vee turn = 1 \vee c_0 = 2) \\
\equiv & (turn = 0 \vee turn = 1) \wedge tour = 0 \wedge tour = 1 \qquad \text{Impossible}
\end{array}
$$

Peterson's Algorithm (IPL June 81) (4/5)

c_0, c_1 program counters for P_0 and P_1.
At beginning $c_0 = c_1 = 1$

P_0	P_1
...	...
$\{\neg a_0 \wedge c_0 \neq 2\}$	$\{\neg a_1 \wedge c_1 \neq 2\}$
`1 a0 := true; c0 := 2;`	`a1 := true; c1 := 2;`
$\{a_0 \wedge c_0 = 2\}$	$\{a_1 \wedge c_1 = 2\}$
`2 turn := 1; c0 := 1;`	`turn := 0; c1 := 1;`
$\{a_0 \wedge c_0 \neq 2\}$	$\{a_1 \wedge c_1 \neq 2\}$
`3 while a1 && turn != 0 do`	`while a0 && turn != 1 do`
`. ;`	`;`
$\{a0 \wedge c_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee c_1 = 2)\}$	$\{a1 \wedge c_1 \neq 2 \wedge (\neg a_1 \vee turn = 1 \vee c_0 = 2)\}$
. C_0;	C_1;
`5 a0 := false;`	`a1 := false;`
$\{\neg a_0 \wedge c_0 \neq 2\}$	$\{\neg a_1 \wedge c_1 \neq 2\}$
...	...

Peterson's Algorithm (IPL June 81) (5/5)

$$
\begin{array}{ll}
 & (turn = 0 \vee turn = 1) \\
\wedge & a_0 \ \wedge \ c_0 \neq 2 \ \wedge \ (\neg a_1 \vee turn = 0 \vee c_1 = 2) \wedge a_1 \ \wedge \ c_1 \neq 2 \ \wedge \ (\neg a_0 \vee turn = 1 \vee c_0 = 2) \\
\equiv & (turn = 0 \vee turn = 1) \wedge tour = 0 \wedge tour = 1 \qquad \text{Impossible}
\end{array}
$$

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

$$\begin{array}{rcl} P, Q & ::= & \text{skip} \mid x := e \mid \text{if } b \text{ then } P \text{ else } Q \mid P; Q \mid \text{while } b \text{ do } P \mid \bullet \\ e & ::= & \text{expression} \end{array}$$

Semantics (SOS)

$$\langle \text{skip}\ ,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma \rangle \qquad \langle x := e,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma[\sigma(e)/x] \rangle$$

$$\frac{\sigma(e) = \text{true}}{\langle \text{ if } e \text{ then } P \text{ else } Q,\ \sigma \rangle \rightarrow \langle P,\ \sigma \rangle} \qquad \frac{\sigma(e) = \text{false}}{\langle \text{ if } e \text{ then } P \text{ else } Q,\ \sigma \rangle \rightarrow \langle Q,\ \sigma \rangle}$$

$$\frac{\langle P,\ \sigma \rangle \rightarrow \langle P',\ \sigma' \rangle}{\langle P; Q,\ \sigma \rangle \rightarrow \langle P'; Q,\ \sigma' \rangle}\ (P' \neq \bullet) \qquad \frac{\langle P,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma' \rangle}{\langle P; Q,\ \sigma \rangle \rightarrow \langle Q,\ \sigma' \rangle}$$

$$\frac{\sigma(e) = \text{true}}{\langle \text{while } e \text{ do } P,\ \sigma \rangle \rightarrow \langle P; \text{while } e \text{ do } P,\ \sigma \rangle} \qquad \frac{\sigma(e) = \text{false}}{\langle \text{while } e \text{ do } P,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma \rangle}$$

$\sigma \in$ Variables $\mapsto$ Values $\qquad \sigma[v/x](x) = v \qquad \sigma[v/x](y) = \sigma(y)$ if $y \neq x$

Operational semantics (parallel part)

Language

$$P, Q ::= \ldots \mid P \mid\mid Q \mid \text{ wait } b \mid \text{await } b \text{ do } P$$

Semantics (SOS)

$$\frac{\langle P,\ \sigma \rangle \rightarrow \langle P',\ \sigma' \rangle}{\langle P \mid\mid Q,\ \sigma \rangle \rightarrow \langle P' \mid\mid Q,\ \sigma' \rangle} \qquad \frac{\langle Q,\ \sigma \rangle \rightarrow \langle Q',\ \sigma' \rangle}{\langle P \mid\mid Q,\ \sigma \rangle \rightarrow \langle P \mid\mid Q',\ \sigma' \rangle}$$

$$\langle \bullet \mid\mid \bullet,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma \rangle$$

$$\frac{\sigma(e) = \text{true}}{\langle \text{ wait } e,\ \sigma \rangle \rightarrow \langle \bullet,\ \sigma \rangle} \qquad \frac{\sigma(e) = \text{true} \quad \langle P,\ \sigma \rangle \rightarrow \langle P',\ \sigma' \rangle}{\langle \text{await } e \text{ do } P,\ \sigma \rangle \rightarrow \langle P',\ \sigma' \rangle}$$

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 $||$.

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

$\langle P_0,\ \sigma_0 \rangle \rightarrow^* \langle P_n,\ \sigma_n \rangle$ when $n \geq 0$,

$\langle P_0,\ \sigma_0 \rangle \rightarrow^+ \langle P_n,\ \sigma_n \rangle$ when $n > 0$.

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

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

Ie no busy waiting. Reductions may block. (Same remark for await e do P).

Atomic statements (Exercices)

Exercice 7 If we make following extension

$$P, Q ::= \ldots \mid \{P\}$$

what is the meaning of following rule ?

$$\frac{\langle P,\ \sigma\rangle \to^{+} \langle \bullet,\ \sigma'\rangle}{\langle \{P\},\ \sigma\rangle \to \langle \bullet,\ \sigma'\rangle}$$

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

Exercice 9 Code generalized semaphores in our language.

Exercice 10 Meaning of $\{$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.

Locks

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.

Try Finally

A statement of the form :

```
TRY S_1 FINALLY S_2 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 stragegy 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.

```
Wait (m, c) :                Signal (c) :
  release(m);                  release(c-sem);
  acquire(c-sem);
  acquire(m);
```

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.