Concurrency 2

From shared memory to synchronization by communication on channels

Jean-Jacques Lévy (INRIA - Rocq)

MPRI concurrency course with :

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

Plan

- exercises (followup)
- readers and writers
- the five philosophers
- synchronous communication channels
- CML
- coding semaphores

Readers and Writers (1/6)

A shared resource is concurrently read or modified.

- Several processes may concurrently read the shared resource.
- A single process (the writer) may modify the resource.
- When readers are running, no writer can be executed.
- When a writer is running, no other writer, nor a reader can run concurrently.

```
PROCEDURE Read() =
BEGIN
  AcquireShared();
  (* read shared data *)
  ReleaseShared();
END Read;
```

```
PROCEDURE Write() =
BEGIN
  AcquireExclusive();
  (* write shared data *)
  ReleaseExclusive();
END Write;
```

Les lecteurs et les écrivains (2/6)

En lecture, on a $nReaders$ simultanés ($nReaders > 0$).

En écriture, on a $nReaders = -1$.

```
PROCEDURE AcquireShared() =
BEGIN
  LOCK m DO
    WHILE nReaders = -1 DO
      Thread.Wait(m, c);
    END;
    ++ nReaders;
  END;
END AcquireShared;

PROCEDURE ReleaseShared() =
BEGIN
  LOCK m DO
    -- nReaders;
    IF nReaders = 0 THEN
      Thread.Signal(c);
  END:
END ReleaseShared;
```

```
PROCEDURE AcquireExclusive() =
BEGIN
  LOCK m DO
    WHILE nReaders != 0 DO
      Thread.Wait(m, c);
    END;
    nReaders := -1;
  END;
END AcquireExclusive;

PROCEDURE ReleaseExclusive() =
BEGIN
  LOCK m DO
   nReaders := 0;
   Thread.Broadcast(c) ;
  END;
END ReleaseExclusive;
```

Les lecteurs et les écrivains (3/6)

Broadcast réveille trop de processus se retrouvant immédiatement bloqués. Avec deux conditions cR et cW, on a un contrôle plus fin :

```
PROCEDURE AcquireShared() =
BEGIN
  LOCK m DO
    ++ nWaitingReaders;
    WHILE nReaders = −1 DO
      Thread.Wait(m, cR);
    END;
    −− nWaitingReaders;
    ++ nReaders;
  END;
END AcquireShared;

PROCEDURE ReleaseShared() =
BEGIN
  LOCK m DO
    −− nReaders;
    IF nReaders = 0 THEN
      Thread.Signal(cW);
  END:
END ReleaseShared;
```

```
PROCEDURE AcquireExclusive() =
BEGIN
  LOCK m DO
    WHILE nReaders != 0 DO
      Thread.Wait(m, cW);
    END;
    nReaders := −1;
  END;
END AcquireExclusive;

PROCEDURE ReleaseExclusive() =
BEGIN
  LOCK m DO
   nReaders := 0;
   IF nWaitingReaders > 0 THEN
     Thread.Broadcast(cR) ;
   ELSE
     Thread.Signal(cW) ;
  END;
END ReleaseExclusive;
```

Les lecteurs et les écrivains (4/6)

Exécuter *signal* à l'intérieur d'une section critique n'est pas très efficace. Avec un seul processeur, ce n'est pas un problème car les réveillés passent dans l'état prêt attendant la disponibilité du processeur.

Avec plusieurs processeurs, le processus réveillé peut retomber rapidement dans l'état bloqué, tant que le verrou n'est pas relaché.

Il vaut mieux faire *signal* à l'extérieur de la section critique.
(Ce qu'on ne fait jamais ! !)

```
PROCEDURE ReleaseShared() =
BEGIN
  VAR doSignal: BOOLEAN;
  LOCK m DO
    -- nReaders;
    doSignal := nReaders = 0;
  END:
  IF doSignal THEN
    Thread.Signal(cW) ;
END ReleaseShared;
```

Les lecteurs et les écrivains (5/6)

Des blocages inutiles sont possibles (avec plusieurs processeurs) sur le *Broadcast* de fin d'écriture.

Comme avant, on peut le sortir de la section critique.

Si plusieurs lecteurs sont réveillés, un seul prend le verrou. Mieux vaut faire *signal* en fin d'écriture, puis refaire *signal* en fin d'accès partagé pour relancer les autres lecteurs.

```
PROCEDURE AcquireShared() =
BEGIN
  LOCK m DO
    ++ nWaitingReaders;
    WHILE nReaders = −1 DO
      Thread.Wait(m, cR);
    END;
    −− nWaitingReaders;
    ++ nReaders;
  END;
  Thread.Signal(cR);
END AcquireShared;
```

```
PROCEDURE ReleaseExclusive() =
BEGIN
  LOCK m DO
   nReaders := 0;
   IF nWaitingReaders > 0 THEN
     Thread.Signal(cR) ;
   ELSE
     Thread.Signal(cW) ;
  END;
END ReleaseExclusive;
```

Les lecteurs et les écrivains (6/6)

Famine possible d'un écrivain en attente de fin de lecture. La politique d'ordonnancement des processus peut aider. On peut aussi logiquement imposer le passage d'un écrivain.

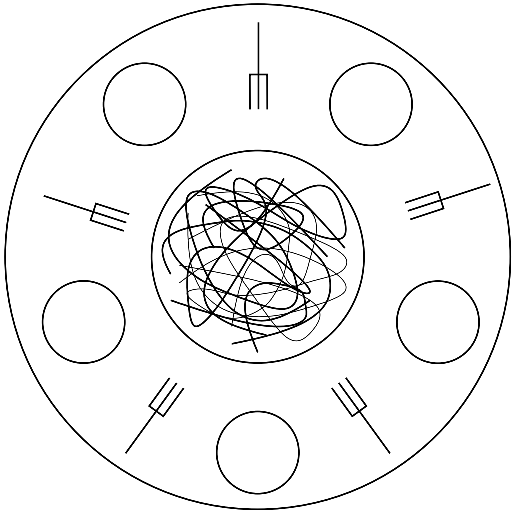
```
PROCEDURE AcquireShared() =
BEGIN
  LOCK m DO
    ++ nWaitingReaders;
    IF nWaitingWriters > 0 THEN
      Thread.Wait(m, cR);
    WHILE nReaders = −1 DO
      Thread.Wait(m, cR);
    END;
    −− nWaitingReaders;
    ++ nReaders;
  END;
  Thread.Signal(cR);
END AcquireShared;
```

```
PROCEDURE AcquireExclusive() =
BEGIN
  LOCK m DO
    ++nWaitingWriters;
    WHILE nReaders != 0 DO
      Thread.Wait(m, cW);
    END;
    −−nWaitingWriters;
    nReaders := −1;
  END;
END AcquireExclusive;
```

Contrôler finement la synchronisation peut être complexe.

Les 5 philosophes (1/7)

- Problème de [Dijkstra] pour tester les primitives concurrentes : verrous, conditions, sémaphores, sémaphores généralisés, etc.
- 5 moines philosophes Φ_i pensent et mangent. Pour manger, ils vont dans la salle commune, où ils dégustent un plat de spaghettis.
- il faut deux fourchettes pour manger les spaghettis. Mais, le monastère ne dispose que de 5 fourchettes.



- Comment arriver à ce qu'aucun moine ne meure de faim ?

Les 5 philosophes (2/7)

```
(* ——————————— Première solution ———————— *)

VAR s: ARRAY [0..4] OF MUTEX;

PROCEDURE Philosophe (i: CARDINAL) =
BEGIN
   (* penser *)
   Thread.Acquire(s[i]);
   Thread.Acquire(s[(i+1) MOD 5]);
   (* manger *)
   Thread.Release(s[i]);
   Thread.Release(s[(i+1) MOD 5]);
END Philosophe;
```

Sureté, mais interblocage.

Les 5 philosophes (3/7)

```
(* ————————— Deuxième solution ———————— *)

VAR f: ARRAY 0..4 OF CARDINAL = {2, 2, 2, 2, 2};
    manger: ARRAY 0..4 OF Thread.Condition;

PROCEDURE Philosophe (i: CARDINAL) =
BEGIN
  WHILE true DO
    (* penser *)
    PrendreFourchettes(i);
    (* manger *)
    RelacherFourchettes(i);
  END;
END Philosophe;
```

- $f[i]$ est le nombre de fourchettes disponibles pour Φ_i

Les 5 philosophes (4/7)

```
PROCEDURE PrendreFourchettes (i: CARDINAL) =
BEGIN
  LOCK m DO
    WHILE f[i] != 2 DO
      Thread.Wait (m, manger[i]);
    END;
    -- f[(i-1) MOD 5]; -- f[(i+1) MOD 5];
  END;
END PrendreFourchettes;

PROCEDURE RelacherFourchettes (i: CARDINAL) =
BEGIN
  VAR g := (i-1) MOD 5, d := (i+1) MOD 5;
  LOCK m DO
    ++ f[g]; ++ f[d];
    IF f[d] = 2 THEN
      Thread.Signal (manger[d]);
    IF f[g] = 2 THEN
      Thread.Signal (manger[g]);
  END;
END RelacherFourchettes;
```

Les 5 philosophes (5/7)

- L'invariant suivant est vérifié

$$\sum_{i=0}^{4} f[i] = 10 - 2 \times mangeurs$$

- interblocage $\Rightarrow$ $mangeurs = 0$
  $\Rightarrow$ $f[i] = 2$ pour tout i $(0 \leq i < 5)$
  $\Rightarrow$ pas d'interblocage pour le dernier à demander à manger.
- famine, si, par exemple, les philosophes 1 et 3 complottent contre le philosophe 2, qui mourra de faim.

Les 5 philosophes (6/7)

- On reprend la première solution + sémaphore généralisé $salle$
  Au début $salle = 4$
  - Pour manger, les philosophes rentrent dans la salle ;
  - il y a au plus 4 philosophes dans la salle ;
  - ils sortent de la salle après le repas ;
  - et retournent penser dans leur cellule.

```
(* ————————————— Troisième solution ————————— *)
PROCEDURE Philosophe (i: CARDINAL) =
BEGIN
  (* penser *)
  SemaphoreGen.P(salle) ; (* —————— début zone critique ———— *)
  Thread.Acquire(s[i]);
  Thread.Acquire(s[(i+1) MOD 5]);
  (* manger *)
  Thread.Release(s[i]);
  Thread.Release(s[(i+1) MOD 5]);
  SemaphoreGen.V(salle) ; (* —————— fin zone critique ————— *)
END Philosophe;
```

Les 5 philosophes (7/7)

- 4 philosophes au plus dans la salle $\Rightarrow$ pas d'interblocage.
- l'invariant suivant est vérifié

$$salle + \text{nombre de processus dans la zone critique} = 4$$

- Si Φ_i exécute P(s[i]), alors il finira cette instruction.
- Si Φ_i attend indéfiniment sur P(s[(i+1)%5]), alors Φ_{i+1} attend indéfiniment sur P(s[(i+2)%5]).
- Si Φ_i exécute P(s[(i+1)%5]), alors il finira cette instruction.
- $\Rightarrow$ Pas de famine.

Exercice 1 Programmer cette solution des 5 philosophes avec les seuls Thread.Wait, Thread.Signal et Thread.Broadcast.

Communication channels (1/2)

- shared memory is not structured (see Ariane 502 on-board software) $\Rightarrow$ restricted communication between processes
- network of processes
- channels as FIFOs `[Kahn, Macqueen]`
  eg Eratosthenes sieve

Communication channels (2/2)

- channels just contains scalars
  communication by rendez-vous (Occam, Ada, etc)

- in

$$[P; \mathit{send}(c, x); P'] \;||\; [Q; \mathit{receive}(c); Q']$$

  P and Q get synchronized by the communication on c.
  Then Q and Q' may start concurrently.

- basic model $\Rightarrow$ CSP [Hoare, 78], CCS [Milner, 80], π-calcul [Milner, Parrow, Walker, 90],

- CCS and π-calculus are easily implementable on top of shared memory. Much more difficult for distributed environments, since one has to fight with the global consensus problem.
  See at end of MPRI concurrency course.
  $\Rightarrow$ join-calculus [Fournet, Gonthier, 96], π1-calculus [Amadio, 97], nomadic-pic [Sewell, Wojciechowski, 00], ... polyphonic C# [MSR, 03].

Concurrent ML

The Event library in Ocaml implements [Reppy] SML library.

```
sig
  type 'a channel
  val new_channel : unit -> 'a Event.channel
  type 'a event
  val send : 'a Event.channel -> 'a -> unit Event.event
  val receive : 'a Event.channel -> 'a Event.event
  val choose : 'a Event.event list -> 'a Event.event
  val wrap : 'a Event.event -> ('a -> 'b) -> 'b Event.event
  val guard : (unit -> 'a Event.event) -> 'a Event.event
  val sync : 'a Event.event -> 'a
  val select : 'a Event.event list -> 'a
  ...
end
```

Updatable storage cell (1/4)

```
open Event;;

type 'a request = GET | PUT of 'a;;

type 'a cell = {
  reqCh: 'a request channel;
  replyCh: 'a channel;
  } ;;

let send1 c msg = sync (send c msg) ;;
let receive1 c = sync (receive c) ;;

let get c = send1 c.reqCh GET; receive1 c.replyCh ;;
let put c x = send1 c.reqCh (PUT x) ;;

let cell x =
  let reqCh = new_channel() in
  let replyCh = new_channel() in
  let rec loop x = match (receive1 reqCh) with
    GET -> send1 replyCh x ; loop x
  | PUT x' -> loop x'
  in
  Thread.create loop x ;
  {reqCh = reqCh; replyCh = replyCh} ;;
```

Updatable storage cell (2/4)

```
open Event;;

type 'a cell = {
  getCh: 'a channel;
  putCh: 'a channel;
  } ;;

let get c =
  sync (receive c.getCh);;

let put c x =
  sync (send c.putCh x);;

let cell x =
  let getCh = new_channel() in
  let putCh = new_channel() in
  let rec loop x = select [
    wrap (send getCh x) (function () -> loop x);
    wrap (receive putCh) loop
  ]
  in
  Thread.create loop x ;
  {getCh = getCh; putCh = putCh} ;;
```

Updatable storage cell (3/4)

```
open Event;;

type 'a sem = {
  getCh: 'a channel;
  putCh: 'a channel;
  } ;;

let get c = sync (receive c.getCh);;
let put c x = sync (send c.putCh x);;

let sem () =
  let getCh = new_channel() in
  let putCh = new_channel() in
  let rec loop x =
    let putEvt = wrap (receive putCh) (function y -> loop (Some y)) in
    match x with
      None -> sync putEvt
    | Some y ->
      let getEvt = wrap (send getCh y) (function () -> loop x) in
      select [ getEvt; putEvt ]
  in
  Thread.create loop None ;
  {getCh = getCh; putCh = putCh} ;;
```

Updatable storage cell (4/4)

$$\begin{array}{rcl} loop(None) & = & Put(y) \cdot loop(Some\, y) \\ loop(Some\, y) & = & \overline{Get}\langle y\rangle \cdot loop(None) \\ & + & Put(z) \cdot loop(Some\, z) \end{array}$$

$$Sem(x) \stackrel{\text{def}}{=} loop(Some\, x)$$

$$\Downarrow$$

$$\begin{array}{rcl} Sem(x) & = & \overline{Get} \cdot Put(y) \cdot Sem(y) \\ & + & Put(y) \cdot Sem(y) \end{array}$$

Exercice 2 Give a program and equations for generalized semaphores.

Conclusion

- What are the equations of interactions ?
- Simplify by considering just interaction.
- Find a logic for interaction.
- Is there an interesting denotational semantics ?
- Find new/correct paradigms for programming.
- What's about distribution ?
- Mobility ?
- Security ?

- $\Rightarrow$ next lectures in MPRI Concurrency course.