Méthodologie de Programmation avec Objective CAML

Cours 9

Vincent Poirriez

courriel :`Vincent.Poirriez@univ-valenciennes.fr`

MCours.com

Programmation de processus concourents

- Concurrence : exécuter plusieurs instructions *en même temps*.
- Processus *lourd* :
  - créé par un `fork` (propre à Unix) ;
  - Zones mémoires distinctes, codes identiques (un seul programme) ;
  - Communiquent par signaux ou prises ou canaux ;
  - Se reporter au cours système.

- Processus *léger* (thread in english) :
  - Partagent leur zone mémoire, codes identiques ;
  - Communiquent par lecture/écriture de la mémoire ou par canaux.
  - Un processus lourd peut contenir plusieurs processus légers.

Un (ou une) thread

plus légère à gérer (qu'un processus) et sa gestion peut être personalisée.

La commutation de contexte est plus simple entre threads.

États uniques pour chaque thread :
– identificateur de thread ;
– état des registres ;
– pile ;
– masques de signaux (décrivent à quels signaux la thread répond) ;
– priorité ;
– données privées à la thread.

Perte du déterminisme

programme principal	
`let x = ref 1`	
processus A	processus B
`x :=!x + 1`	`x :=!x * 2`

Quelle est la valeur de `x` à la fin du programme ?

Que ce soit pour les processus lourds et les threads ce n'est pas vous qui gérez l'ordonnancement (schedule) des processus.

Synchronisation et problèmes

Supposons que `send` et `receive` existent et prennent en argument un processus et éventuellement une valeur.

processus A	processus B
`let x = ref 1`	`let y = ref 2`
`send(B, !x)`	`y := !y + 2`
`x := !x * 3`	`y := !y + receive(A)`
`send(B, !x)`	`send(A, !y)`
`x := !x + receive(B)`	`y := !y + receive(A)`

Que se passe-t-il suivant que `receive` et/ou `send` est bloquant ou non ?

On appelle Deadlock (inter-blocage) une situation ou deux (des) processus s'attendent mutuellement.

Différentes situations

Les processus concurrents sont :

- Sans relation
- avec relation, sans synchronisation
- avec relation d’exclusion mutuelle
- avec relation d’exclusion mutuelle et communication
- avec relation, sans exclusion mutuelle et avec communication synchrone.

- saisie et correction ortographique *simultanée*
- plusieurs applets dans une page interactive
- communiquer des résultats intermédiaires et continuer à calculer
- …

Les processus légers ne font pas gagner du temps.
Ils permettent d'exprimer *simplement* des actions concurrentes.

Threads et OCaml

Les modules de la bibliothèque

Il y a quatre modules :

– `Thread` : création et exécution des processus légers.

– `Mutex` : les verrous pour l'exclusion mutuelle

– `Condition` pour gérer les attentes, et les réveils

– `Event` canaux de communication et fonctions de communication. Permet de la programmation fonctionnelle concurrente.

MCours.com

Deux saveurs de Threads

- threads et bytecode,
  - Gérés par la machine virtuelle, donc portables.
  - Ne nécessitent pas un système multithreadé

- threads et code natif
  - Il faut avoir compilé ocaml avec l'option `-with-pthread`
  - Nécessite un système qui dispose des threads posix

La bibliothèque de threads ne fait pas partie de la bibliothèque standard.

Il y a donc des options de compilation à écrire :

– Threads et bytecode :

```
$ ocamlc -thread unix.cma threads.cma fichiers.ml
```

– Threads et code natif :

```
$ ocamlopt -thread unix.cmxa threads.cmxa fichiers.ml
```

Sur solaris :

```
$ ocamlopt -thread unix.cmxa threads.cmxa\
         fichiers.ml -cclib -lposix4
```

Créer un toplevel qui contient les threads :

```
$ ocamlmktop -thread -o ocamlthr unix.cma threads.cma
```

Le Module `Thread`

```
Thread.create: ('a -> 'b) -> 'a -> Thread.t
```

- Les deux arguments sont la fonction à exécuter et son argument
- Attention, remarquez que le résultat est perdu ! ! !

```ocaml
let ma_fonct (deb, fin) =
  for i = deb to fin do  Printf.printf " (%d)\n " i done;
  print_newline ()

let tid = Thread.create ma_fonct (0, 10)
```

Patienter, Tuer et Suivre

– Se mettre au repos :

`Thread.delay : float -> unit`
L'argument est le nombre de secondes

– Tuer un thread dont je connais l'identité

Il faut éviter, le thread peut avoir verrouillé des ressources partagées.

$\Rightarrow$ Que chaque thread surveille si il a encore le droit de vivre, sinon, qu'il se suicide en libérant les ressources qu'il possède.

– Suivre un thread dont je connais l'identité :

`Thread.join : Thread.t -> unit`

Exclusion Mutuelle

Création et manipulation de verrous :

```
Mutex.create : unit -> Mutex.t
Mutex.lock : Mutex.t -> unit
Mutex.try_lock : Mutex.t -> bool
Mutex.unlock : Mutex.t -> unit
```

`Mutex.try_lock` essaye de verrouiller, renvoie true s'il réussit false sinon.
Alors que `Mutex.lock` se bloque en attendant que le verrou soit libre.

Un tueur asynchrone

Les ressources partagées :

```
let vivre = ref true
let verrou_de_vivre = Mutex.create ()
```

Le tueur :

```
let tueur_patient attente =
  Thread.delay attente;
  Mutex.lock verrou_de_vivre;
  vivre := false;
  Mutex.unlock verrou_de_vivre
```

⇒ Chaque accès à une ressource partagée est protégée.

La fonction qui travaillera si elle a le droit

On utilise `Thread.yield` pour indiquer à l'ordonnanceur que c'est un bon endroit pour donner la main à un autre thread.

```
let ma_fonct (deb, fin, pas) =
  for i = deb to fin do
    if i mod pas = 0 then begin
      Thread.yield();
      Mutex.lock verrou_de_vivre;
      if not !vivre then begin
        Mutex.unlock verrou_de_vivre;
        Thread.exit () end
      else Mutex.unlock verrou_de_vivre
    end;
    Printf.printf " (%d)\n " i
  done
```

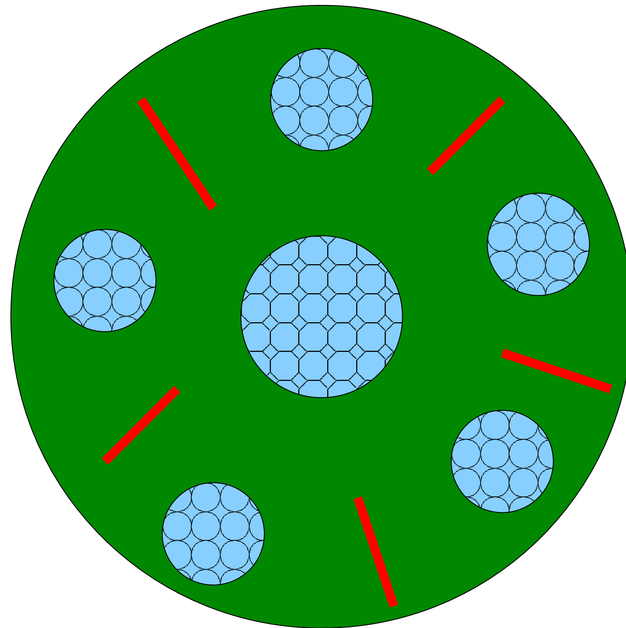
Le point d'entrée

```
let essai =
  let tid_tueur = Thread.create tueur_patient 0.0001 in
  let tid = Thread.create ma_fonct (1,123456,10) in
  Thread.join tid_tueur;
  print_endline "\n C'est fini"
```

Compilé par

```
ocamlc -thread -o testthr unix.cma threads.cma exthreads.ml
```

Exemple classique : les philosophes



Due à **Dijkstra** :

"Cinq philosophes partagent leur temps entre l'étude et la venue au réfectoire pour manger un bol de riz. À chaque philosophe est affecté un bol. La table commune est ronde et il y a cinq baguettes, chacune est positionnée entre deux bols. Pour manger un philosophe a besoin de deux baguettes. Il doit donc éventuellement attendre que ses voisins aient libéré les baguettes qui entourent son bol."

Modélisation des baguettes

- Une baguette est une ressource partagée.
- Deux philosophes ne peuvent la tenir en même temps.
- Nous les modélisons par des verrous.
- Les philosophes font semblant de penser (et de manger)

```
let baguettes = Array.init 5 (fun i -> Mutex.create ())

let méditer = Thread.delay and manger = Thread.delay
```

Modélisation d'un philosophe

Prendre une baguette, c'est prendre un verrou.

```
let philosophe i =  let droite_i = (i+1) mod 5 in
  while true do
    méditer (0.2 +. Random.float 5.);

    Mutex.lock baguettes.(i);
    Printf.printf "Le␣philosophe␣%d␣prend␣la␣baguette␣à␣sa␣gauche␣\n␣" i;
    Printf.printf "␣et␣patiente␣un␣peu\n␣";flush stdout;
    méditer 0.2;

    Mutex.lock baguettes.(droite_i);
    Printf.printf
      "Le␣philosophe␣%d␣prend␣la␣baguette␣à␣sa␣droite,␣il␣mange␣␣\n␣" i;
    flush stdout; manger (0.5 +. Random.float 2.);

    Mutex.unlock baguettes.(i);
    Printf.printf "␣Le␣philosophe␣%d␣repose␣la␣baguette␣à␣sa␣gauche␣\n␣" i;
    Printf.printf "␣et␣patiente␣un␣peu\n";flush stdout;
    méditer 0.1;

    Mutex.unlock baguettes.(droite_i);
    Printf.printf "␣Le␣philosophe␣%d␣repose␣la␣baguette␣à␣sa␣droite," i;
    Printf.printf "␣il␣retourne␣penser.␣\n"; flush stdout;
  done
```

Le programme principal

```
let entrée =
 for i = 0 to 4 do ignore (Thread.create philosophe i) done;
 (* boucle indéfiniment *)
 while true do Thread.delay 3. done
```

Compilé par

```
ocamlc -thread -o phil unix.cma threads.cma philosophes.ml
```

MCours.com

Problèmes de cette solution

– Rien n'interdit que tous les philosophes aient en main la baguette à leur droite.
  $\Rightarrow$ Il y a alors Deadlock.
– Rien n'interdit que deux philosophes s'entendent pour empêcher leur voisin commun de manger :
  $\Rightarrow$ Il y alors famine.

Plusieurs solutions ont été proposées. Vous pouvez réfléchir. Ce n'est pas trivial.

Les sémaphores

Motivation

- Un processus peut avoir besoin de la réalisation d’un calcul par un autre processus pour continuer.
- Exemple : un consommateur peut devoir attendre qu’un producteur ait produit.
- Il faut *faire signe* pour dire que la *condition* attendue est réalisée.

Le modèle théorique des sémaphores

Un sémaphore est une variable entière `s` positive ou nulle.

Quand `s` est initialisée, on dispose de seulement deux opérations pour la manipuler : `wait` et `signal` notée `P(s)` et `V(s)`.

`s` correspond au nombre de ressources disponibles.

– `wait(s)` si $s > 0$ alors s := s - 1, sinon le processus est suspendu
– `signal(s)` si un processus est suspendu en attente que `s` devienne positive, alors le réveiller, sinon s := s + 1

Condition et Signal

Le module Condition

C'est l'adaptation Posix des sémaphores.

Réveil et suspension de processus sur un *signal*.

```
create: unit -> Condition.t
signal : Condition.t -> unit
broadcast : Condition.t -> unit
wait : Condition.t -> Mutex.T -> unit
```

Schéma classique pour les conditions

Soit `D` une structure de donnée partagée, `m` son verrou (mutex), et `c` une variable de condition.

```
Mutex.lock m;
while (* Un prédicat P sur  D n'est pas vérifiée *) do
  Condition.wait c m
done;
   (* un code qui modifie D *)
if (* Le prédicat P sur D est vérifié maintenant *) then
    Condition.signal c;
Mutex.unlock m;
```

L’exemple producteur/consommateur

Tout est dans un module `Prodcons`

Le type de donnée `'a Prodcons.t`

```
type 'a t =
 {buffer: 'a array ;      (* le tampon partagé *)
  lock: Mutex.t ;         (* son verrou *)
  mutable readpos: int ; (* la position de lecture *)
  mutable writepos: int ;(* la position d'écriture *)
  notempty: Condition.t ;(* pour signaler qu'il y a quelque chose à consommer*)
  notfull: Condition.t    (* Pour signaler qu'il y a de la place pour écrire.*)
 }
```

Créer un tampon

```
(** [create_buf size init] renvoie un tampon de taille [size]
   initialisé avec les valeurs [init]. *)
```

```
let create_buf size init =
  { buffer = Array.create size init;
    lock = Mutex.create();
    readpos = 0;
    writepos = 0;
    notempty = Condition.create();
    notfull = Condition.create()
  }
```

Produire

Respecte le schéma classique :

```
let put p data =
  Mutex.lock p.lock;                      (* Verrouiller le tampon *)
  while (p.writepos + 1) mod Array.length p.buffer = p.readpos do
    Condition.wait p.notfull p.lock (* Attendre de pouvoir écrire*)
  done;
  p.buffer.(p.writepos) <- data;      (* écrire *)
  p.writepos<- (p.writepos + 1) mod Array.length p.buffer;
  Condition.signal p.notempty;        (* Prévenir qu'il y a quelque ch
  Mutex.unlock p.lock                 (* Déverrouiller *)
```

Consommer

Respecte le même schéma :

```
let get p =
  Mutex.lock p.lock;                          (* Verrouiller *)
  while p.writepos = p.readpos do
    Condition.wait p.notempty p.lock(* Attendre qu'il y ait qqc *)
  done;
  let data = p.buffer.(p.readpos) in(* consommer *)
  p.readpos <- (p.readpos + 1) mod Array.length p.buffer;
  Condition.signal p.notfull;                 (* Prévenir qu'il y a de la plac
  Mutex.unlock p.lock;                        (* Déverrouiller *)
  data                                        (* Renvoyer la valeur *)
```

Pour créer des producteurs/consommateurs

Les acteurs partagent tous le même lieu de stockage des données. Nous définissons un type acteur avec deux champs qui permettent de créer des producteurs ou des consommateurs nommés.
Nous choisissons de produire des données de type `int`.

```
type acteurs_integers =
    { prod: string -> int -> unit;      (*[prod nom data]*)
      cons: string -> unit-> unit       (*[cons nom ()] *)
    }
```

Protéger l'impression

Nous voulons visualiser sur la sortie standard l'alternance des acteurs.

La sortie standard est une ressource partagée :

⇒ Il faut la protéger.

Plus généralement, protégeons l'application d'une fonction :

```
let protège f arg verrou =
  Mutex.lock verrou;
  f arg;
  Mutex.unlock verrou
```

Les fonctions d'impressions d'un producteur et d'un consommateur :

```
let imprime_prod (nom,n) =
  Printf.printf "%s   --> %2d \n" nom n; flush stdout
let imprime_cons (nom,n) =
  Printf.printf "%2d   --> %s \n" n nom; flush stdout
```

Création d'acteurs

C'est à dire une fonction de production et une de consommation, ainsi qu'une taille de tampon.

```
let create_acteurs size_buf=
  let buff = create_buf size_buf 0 in
  let verrou_impr = Mutex.create () in
  let rec produce nom n =
    protège imprime_prod (nom,n) verrou_impr;
    put buff n;
    Thread.delay (Random.float 0.15);
    produce nom (n+1)
  and consume nom ()=
    let n = get buff in
    protège imprime_cons (nom,n) verrou_impr;
    Thread.delay (Random.float 0.2);
    consume nom ()
  in {prod=produce; cons=consume}
```

Première mise en oeuvre

Un producteur, un consommateur, un tampon de taille 10

Le fichier `prodcons1.ml`

```
(*prodcons1.ml*)

open Prodcons
let acteurs = create_acteurs 10

let c'est_parti =
  ignore (Thread.create (acteurs.prod "a") 0);
  Thread.delay 0.5;
  acteurs.cons "b" ()
```

Compiler par :

```
ocamlc -thread -o prcons1 unix.cma threads.cma  \
      prodcons.cmo prodcons1.ml
```

3 producteurs et deux consommateurs

Le fichier `prodcons2.ml`

```
(*prodcons2.ml*)

open Prodcons;;
let acteurs = create_acteurs 10;;
let j'y_vais =
  let p1 = Thread.create (acteurs.prod "p1") 1
  and p2 = Thread.create (acteurs.prod "p2") 1
  and p3 = Thread.create (acteurs.prod "p3") 1
  and c1 = Thread.create (acteurs.cons "c1") ()
  and c2 = Thread.create (acteurs.cons "c2") ()
  in
  Thread.delay 0.5;
  if not !Sys.interactive then exit(0)
```

Compiler par :

```
ocamlc -thread -o prcons2 unix.cma threads.cma \
    prodcons.cmo prodcons2.ml
```

tester

– Compilez ce programme et testez le.

– Remarquez que l'ordre des acteurs varie entre deux évaluations.

MCours.com