Révision

Différence entre un tuyau et une prise ?

Étapes pour se connecter à un service ?

Étapes pour offrir un service ?

Que fait exactement `listen` ?

Les coprocessus

- Les coprocessus
- Les verrous
- Les conditions
- Communication synchrone

Insuffisance des processus

Lancer une tache en parallèle

Le processus fils est une copie du père. Il devient indépendant.

Avantages $\Longleftrightarrow$ inconvénients

- Synchronization facile (peu de compétion), car chaque processus a une copie des resources, sauf pour les ressources système, *e.g.* descripteur de fichier.
  Communication par tuyau presque exclusivement.
- La communication intime n'est pas possible.
  - Mise en place de protocoles de communications parfois lourds : tuyaux + marshalling pour échanger simplement un enregistrement de valeurs.
  - Source d'inefficacité pour la communication peu fréquente de grosses données ou de données structurées : il faut recopier.

Les coprocessus

Processus à plusieurs têtes

- À l'inverse des processus, qui sont clonés, les coprocessus partagent tous la même mémoire.
- Seul leur identité (pid) et quelques informations système (*e.g.* le masque des signaux) sont duppliquées :

La vision du système

- Le système ne voit pas le partage de mémoire. Il gère chaque coprocessus (presque) comme un autre processus.
- Sous linux : la création d'un processus ou d'un coprocessus sont le même appel système `clone` appelé avec des options différentes (avec ou sans partage de la mémoire).

Un peu de zoologie

coprocessus = threads dans la litérature.

Mais attention !

Processus léger = *light threads* $\neq$ coprocessus

light threads fait référence à des threads qui tournent dans le même processsus Unix. Possible sur certains OS, mais les processus légers sont souvents simulés, *i.e.* par une librairie.

coprocessus $\neq$ coroutines

Les coroutines ne peuvent pas s'évaluer en parallèle. Il y a plusieurs fils d'exécutions entrelacés, mais toujours évalués séquentiellement : le changement de coroutine se fait à des points convenus, en général à la demande explicite du programme : **pas de pré-emption**.

Coroutines

Avantages

Comme les coprocessus, la mémoire est partagée.

La synchronisation est plus simple, car l'éxécution est séquentialisée : pas besoin de protéger l'accès aux resources.

Inconvénients

- Si une coroutine bloque, tout le programme bloque.
- Petite illusion de parallèlisme mais seulement si les coroutines donnent la main suffisement fréquemment...
- donc seulement s'il n'y a pas d'appels sytèmes bloquants... remplacés par de l'attente active ou une gestion centralisée de toute la communication avec le monde extérieur.

Les coroutines sont à mi-chemin.

Les coprocessus en OCaml

L'implémentation native passe par le système.

C'est la version utilisée dans le cours et installée sur les machines.

L'implémentation simulée

- Un seul processus Unix.
- Une biliothèque implémente l'ordonnancement et redéfinit certains appels systèmes pour ne pas bloquer le programme...

Compatibilité

- OCaml donne une seule interface indépendante de la version.
- Autant que possible la sémantique est identique, mais...
- Un appel à C bloquant bloque un seul thread en natif, mais bloque tout le programme dans la version simulée.

La version simulée est à mi-chemin : ordonnancement (séquentiel) de l'exécution mais avec une «petite granularité».

Exemples d'utilisation des coprocessus

Pour simplifier ou réduire la communication

Serveurs

- Gestion d'une piscine de serveurs chauds, près à démarrer.
- Partage de données (cache en mémoire).

Partage + Parallèlisme

- Tri de tableaux de grosses structures et comparaison pouvant bloquer (*e.g.* nécessittant un accès réseau).
  Le tri « rapide » ou « fusion » se parallèlisent bien.
- Recherche en table (même contexte).

Interface graphique et programme principal en $\|$

- Un coprocessus gère les événements et l'affichage.
- Communication de haut niveau avec le programme principal.
- Simplification du programme principal.

Fausses vérités

Les coprocessus sont plus rapides au démarrage

- En fait, cela dépend de ce que l'on fait après,
- La mémoire est copiée paresseusement (copy-on-write).
- La création d'un processus et d'un coprocessus sont chers.
  La différence n'est pas forcément visible.
- Dépend beaucoup du système et de la librairie de threads.

Occupation mémoire

Exemple du GC, après création, au premier GC :

- Un processus vide sa mémoire des choses inutiles.
- Un coprocessus garde les racines de tous les coprocessus.
  Sa mémoire cumule les besoins de tous.
  Il n'y a qu'un GC (synchrone, en général, ou alors très couteux) pour tous les coprocessus.

Les deux comportements sont difficilement comparables.

Mise en œuvre

Compilation

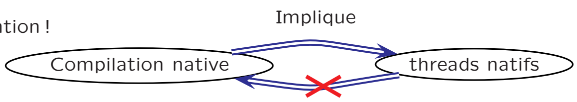
Il faut ajouter l'option `-thread` et la librairie `thread.cma`

```
ocamlc -thread unix.cma threads.cma -o prog a.ml b.ml
```

Compilation en code natif, seulement avec threads natifs :

```
ocamlopt -thread unix.cmxa threads.cmxa -o prog a.ml b.ml
```

Attention !



Les threads ne sont pas utilisables avec le toplevel

Créer un coprocessus


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

- `Thread.create` f x calcule f x dans un nouveau coprocessus.
- Retourne un identifiant du coprocessus.

Terminaison précoce

- Du coprocessus courant
  ```
  Thread.exit : unit -> unit
  ```
- Du programme
  ```
  Pervasives.exit : int -> 'a
  ```

Surtout, ne pas les confondre !

Mise en attente

Attendre la terminaison d'un autre coprocessus

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

Le coprocessus appelant est bloqué jusqu'à ce que le coprocessus en argument ait terminé.

Utile pour que le coprocessus principal attende la terminaison de tous les autres avant de faire `exit` (terminaison du programme).

Mise en sommeil

```
Thread.delay : float -> unit
```

Pour compatibilité avec les threads simulés.

Avec les threads natifs, défini par...

```
let delay time = ignore(Unix.select [] [] [] time)
```

Exemples (sans synchronisation)

Sections critiques

Tout accès à des ressources partagées.

Tris, recherche

Les coprocessus tournent sur des parties disjointes des ressources.

La primitive `join` suffit comme synchronisation.

- Une variable partagée `found`.
- Une seule valeur écrite possible avant le join.
- Lecture après le join de tous les fils.

Recherche en parallèle


```
let simple_search = ...

let rec psearch k cond v =
  let n = Array.length v in
  let slice i = Array.sub v (i * k) (min k (n - i * k)) in
  let slices = Array.init (n/k) slice in
  let found = ref false in
  let pcond v = if !found then Thread.exit(); cond v in
  let search v =
    if simple_search pcond v then found := true in
  let proc_list =
    Array.map (Thread.create search) slices in
  Array.iter Thread.join proc_list;
  !found;;
```

Variante : copie récursive de répertoires.

Exemple : coexec

Les coprocessus ne retournent pas

Mais on peut retourner une valeur dans une référence :

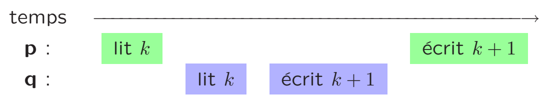
```
exception Killed
type 'a result = Running | Returned of 'a | Exception of exn
let eval f x = try Returned (f x) with z -> Exception z
let coexec (f : 'a -> 'b) (x : 'a) : unit -> 'b =
  let result = ref Running in
  let p = Thread.create (fun x -> result := eval f x) x in
  function() ->
    match join p; !result with
     Running -> raise Killed
    | Returned v -> v
    | Exception z -> raise z;;
let v1 = coexec succ 4 and v2 = coexec succ 5 in v1()+v2();;
```

Compteur : synchronisation nécessaire

Cas général

Empêcher certains entrelacements à l'exécution

Trace erronée :



Solution : les verrous

Les verrous

Les `Mutex` (pour *exclusion mutuelle*)

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

Sémantique

- Un seul coprocessus peut avoir le verrou.
- `lock` prend le verrou s'il est disponible. Sinon le coprocessus bloque jusqu'à ce que le verrou soit disponible.
- `try_lock` prend le verrou s'il est disponible et retoure `true`, Sinon `try_lock` ne bloque pas, ne prend pas le verrou et retourne `false`.
- `unlock` libère le verrou (réveille les coprocessus en attente).

Implémentation

L'instruction «Test and set» ou «read and clear»

Les processeurs modernes ont une instruction qui lit une adresse et la met à zéro (L'atomicité est garantie par le fait qu'il s'agisse d'une seule instruction.)

Prise de verrou

- Avec attente active

```
type lock = { free : bool; }
let unlock l = l.free <- true;;
let lock l = while not (test_and_set l.free) do done;;
```

Implémentation

L'instruction «Test and set» ou «read and clear»

Les processeurs modernes ont une instruction qui lit une adresse et la met à zéro (L'atomicité est garantie par le fait qu'il s'agisse d'une seule instruction.)

Prise de verrou

- Avec attente active
- Plus raisonablement

```
let unlock l =
  l.free <- false; wakeup_lock l
let lock l =
  while not (test_and_set l.free; sleep_on_lock l)
  do done;;
```

où `sleep_on_lock` endort le coprocessus en attente jusqu'à ce que quelqu'un libère le verrou. Il faut retester, car un autre coprocessus plus rapide et peut avoir déjà repris le verrou.

Compteurs


```
type counter = { lock : Mutex.t; mutable counter : int }
let newcounter() = { lock = Mutex.create(); counter = 0 }
```

Accès : pas de problème (la lecture est atomique)

```
let getcounter c = c.counter
```

Ajout : bloquer le compteur entre la lecture et l'écriture.

```
let addtocounter c k =
  Mutex.lock c.lock;
  c.counter <- c.counter + k;
  Mutex.unlock c.lock;;
```

Si la lecture n'est pas atomique, on peut autoriser soit plusieurs lecteurs, soit un écrivain mais pas les deux à la fois.

Exemples d'utilisation de verrous

L'exemple du compteur est typique/fréquent

- Quasiment toutes les structures de données mutables : tampons, graphes, etc.
- La communication entre coprocessus passe par des structures mutables (un coprocessus ne retourne pas !).
  Le cas sans utilisation de verrous sont rares (*e.g.* `psearch`).

Exemples d'utilisation de verrous

L'exemple du compteur est typique/fréquent

Les fonctions non réentrantes

- Ce sont des fonctions qui utilisent des variables globales (ou des structures système partagées (*e.g.* descripteurs) et tel qu'un nouvel appel de la fonction ne peut se faire que lorsque le précédent est terminé.
- Il faut mettre des verrous.
- Exemple : les librairie `Str`, `Lexing`, *etc.* Les impressions.

Exemples d'utilisation de verrous

L'exemple du compteur est typique/fréquent

Les fonctions non réentrantes

- Ce sont des fonctions qui utilisent des variables globales (ou des structures système partagées (*e.g.* descripteurs) et tel qu'un nouvel appel de la fonction ne peut se faire que lorsque le précédent est terminé.
- Il faut mettre des verrous.
- Exemple : les librairie `Str`, `Lexing`, *etc.* Les impressions.

```
let strlock = Mutex.create()
let regexp_match r string =
  Mutex.lock strlock;
  try_finalize (unsafe_regexp_match r) string
    Mutex.unlock strlock;
```

Attention ! verrou laissé = deadlock

Utilisation de plusieurs verrous

Risque de deadlock

- p prend A puis attend B
- q prend B puis attend A

Si possible

Hiérarchiser les verrous : A est toujours pris avant B

Serveur avec coprocessus

Modèle

- Serveur séquentiel mais traitement par un coprocessus.

```
let tcp_coserver treat_connection sockaddr =
  tcp_server
    (fun server client ->
      Thead.create (treat_connection server) client);;
```

- Attention à se protéger contre les appels non ré-entrants.

Exemple : serveur http

Attention ! Avec le prototole HTTP 1/1 les entêtes sont complexes et analysées avec `Str` (ou `Lexing`) qui ne sont (malheureusement) pas des librairies réentrantes...

Problème de l'allocation mémoire

Le code OCaml est séquentialisé.

- Au plus un seul coprocessus peut être en train d'exécuter du code OCaml.
- Parallèlisme possible pendant l'éxecution de certains appels système et du code C.

Conséquences

- Pas besoin de synchronisation avec le runtime OCaml (l'allocation).
- Pas plus de sureté pour l'utilisateur, car le verrou peut-être rendu à presque tout moment : l'entrelacement des coprocessus est volontairement auto-provoqué par un timer, donc (quasi) asynchrone.
- Iconvénients : le seul vrai parallèlisme (sur une machine multi-processus) vient des appels système ou du code C.

Insuffisance des verrous

Comment implémenter la file d'attente ?

- Certains processus écrivent dans la file et d'autres lisent.
- Une opération lecture ou écriture change l'état interne et doit avoir le verrou (pour exclure une autre opération en parallèle).
- On ne peut pas lire une file vide. Il faut donc tester avant.
- Il faut prendre le verrou avant de tester, sinon un autre coprocessus peut vider la file entre temps. Si la file est
  - non vide, on retire l'élément, puis on relache le verrou.
  - vide, il faut rendre le verrou pour éviter un blocage (personne ne pourrait remplir la file).
    puis réessayer un peu plus tard...
- **C'est mauvais car on fait de l'attente active.**

Insuffisance des verrous

Comment implémenter la file d'attente ?

- Il faut prendre le verrou avant de tester, sinon un autre coprocessus peut vider la file entre temps. Si la file est
  - non vide, on retire l'élément, puis on relache le verrou.
  - vide, il faut rendre le verrou pour éviter un blocage (personne ne pourrait remplir la file).
    puis réessayer un peu plus tard...
- **C'est mauvais car on fait de l'attente active.**

Ce qu'il manque

- S'endormir quand on ne peut rien faire
  (on un changement)
- Être réveillé quand la situation a changé.

Solution : les conditions

Le module `Condition` pour exclusion mutuelle

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

Sémantique des conditions

- Ce sont des objets sur lesquels un coprocessus peut se mettre en attente. (une condition correspond en général à une propriété devant être satisfaite).
- Un coprocesssus peut signaler un changement de situation sur une condition : tous les processus en attente sur cette condition seront réveillés.

En fait

- Une condition correspond à un problème d'exclusion mutuelle, donc elle est associée à un verrou.
- Le `verrou` associé à la condition doit être pris avant de tester la propriété associée à la condition (sinon, cette propriété pourrait avoir changé entre le test et la mise en attente)
- Le verrou est relaché (automatiquement) pendant la mise en sommeil et repris (automatiquement) au réveil.

Répartition des tâches

Responsabilité du programmeur

- Endormir un processsus sur une condition. (En général, après avoir observé qu'une propriété n'est pas remplie)
- Réveiller un ou les processus endormis sur une condition.
  En général, après avoir effectué un changement qui rend (peut-être) une propriété valide.
- Prendre le `verrou` avant le test et le relâcher lorsque le calcul est réalisé.

Responsabilité du système

- Endormir le processus et relacher le verrou pendant l'attente.
- Assurer la reprise du verrou au réveil.
- Transmettre l'ordre de réveil à un ou à tous les processus endormis sur la condition.

Détails importants

Réveil

- `signal` réveille un seul processus en attente sur la condition et ne fait rien si aucun processus n'est en attente.
- `broadcast` réveille tous les processus en attente sur une condition.

Le choix entre ces deux options est important

- `signal` est moins coûteux, mais peut ne pas suffire.
- `broadcast` est toujours sûr (au pire réveil pour rien) mais peut être inutilement coûteux.

Détails importants

Une condition c est associée à un verrou v

wait

- Lorsqu'on fait `wait` sur c, on doit impérativement posséder le verrou v (c'est une erreur sinon).
- Le verrou est relaché (par le système) pendant le `wait` mais repris (par le système) juste avant le retour du `wait`.

signal (ou broadcast)

- On n'est pas obligé de posséder le verrou v associé à la condition sur laquelle on signal (ce n'est pas une erreur système).
- Mais cela ne pose pas de problème : à la libération du verrou, les processus bloqués sur le verrou seront réveillés (et ne peuvent pas l'être àvant puisque le verrou n'est pas disponible).

Les files d'attentes

Queue + verrou (lock) + condition (non_empty)

```
type 'a t =
    { queue : 'a Queue.t;
      lock : Mutex.t;
      non_empty : Condition.t }
let create () =
  { queue = Queue.create();
    lock = Mutex.create();
    non_empty = Condition.create() }
```

Les files d'attentes

Queue + verrou (lock) + condition (non_empty)

Retrait

```
let take q =
  Mutex.lock q.lock;
  while Queue.length q.queue = 0
  do Condition.wait q.non_empty q.lock done;
  Queue.take q.queue;
  Mutex.unlock q.lock;;
```

- Attention ! il faut retester au réveil, car un autre processus, qui n'était pas forcément en attente sur le verrou au moment où la condition a été signalée, peut avoir demandé et pris le verrou, fait `take` et relaché le verrou, avant que le processsus qui était en attente ne soit réveillé.

Les files d'attentes

Queue + verrou (lock) + condition (non_empty)

Ajout

```
let add e q =
  Mutex.lock q.lock;
  let was_empty = Queue.length q.queue = 0 in
  Queue.add e q.queue;
  if was_empty then Condition.signal q.non_empty;
  Mutex.unlock q.lock;;
```

Files d'attentes bornées

Contrôle par deux conditions sur le même verrou

- On peut maintenant bloquer en lecture ou en écriture.
- Chaque opération combine le contrôle de l'ajout et du retrait dans la version non bornée.

```
type 'a t =
    { queue : 'a Queue.t; size : int; lock : Mutex.t;
      non_empty : Condition.t; non_full : Condition.t; }
```

```
let create k =
  if  k > 0 then
    { queue = Queue.create(); size = k;
      lock = Mutex.create();
      non_empty = Condition.create();
      non_full = Condition.create() }
  else failwith "Tqueue.create: empty size";;
```

Files d'attentes bornées

Contrôle par deux conditions sur le même verrou

Ajout et retrait sont symétriques

```
let add x q =
  Mutex.lock q.lock;
  while Queue.length q.queue = q.size
  do Condition.wait q.non_full q.lock done;
  if Queue.length q.queue = 0 then
    Condition.broadcast q.non_empty;
  Queue.add q x;
  Mutex.unlock q.lock;;
```

Files d'attentes bornées

Contrôle par deux conditions sur le même verrou

Ajout et retrait sont symétriques

```
let take q =
  Mutex.lock q.lock;
  while Queue.length q.queue = 0
  do Condition.wait q.non_empty q.lock done;
  if Queue.length q.queue = q.size then
    Condition.broadcast q.non_full;
  Queue.take q.queue;
  Mutex.unlock q.lock;;
```

Files d'attentes bornées

Contrôle par deux conditions sur le même verrou

Ajout et retrait sont symétriques

Peut-on partager la condition ?

- C'est possible, mais moins efficace : on réveille trop de monde.
- Il est préférable de mettre des conditions différentes pour des propriétés différentes.

Communication synchrone par événements

Problème

Pour envoyer une valeur d'un coprocessus à un autre, il faut écrire et lire dans une variable mutable, ce qui implique une synchronisation compliquée par verrou et condition.

Les cannaux de communication

- Un canal est un « co-tuyaux » sur lequel deux coprocessus peuvent s'envoyer des messages (valeurs structurées homogènes).
- Les coprocessus partagent le même canal mais la synchronization de l'échange est transparente.
- La communication est synchrone : émetteur et receveur se sychronise au moment de l'échange.

Le module Event


```
new_channel : unit -> 'a channel
send : 'a channel -> 'a -> unit event
receive : 'a channel -> 'a event
choose : 'a event list -> 'a event
sync : 'a event -> 'a
poll : 'a event -> 'a option
```

Les événements (valeurs de type `event`) sont des données passives. Ils peuvent être combinés avec `choose`.

C'est seulement lorsque `sync` est appelé que le processus se met en attente jusqu'à la réalisation de cet événement.

Communication par messages

Schéma simple

1. Fabriquer un canal visible par les coprocessus.
2. Lancer les coprocessus (avant ou après l'étape précédente).
3. Construire et synchroniser les événements
   - Envoyer d'un coté : `sync (send c v1)`
   - Revevoir de l'autre coté : `sync (receive c)`

Example

```
let channel = new_channel () in
let son() = sync (send c 3); sync (send c 4) in
let _ = Thread.create son () in
let v1 = sync (receive c) in
let v2 = sync (reveive c) in
Print.printf "%v1=%d v2=%d" v1 v2;;
```

Événement complexes

Construction et synchronization

- `receive` et `send` construisent des événements (structures de données) mais ne les synchronisent pas (pas de communication).
- La communication n'est entreprise qu'au moment où `sync` est appelé.

Alternative

```
choose : 'a event list -> 'a event
```

Construit un événement qui réalisera exclusivement un seul des événements parmi ceux de la liste.

- Pour recevoir sur `c1` ou sur `c2`
  ```
  sync (choose [ receive c1; receive e2])
  ```

Exemple : « co-tuyaux »

Simuler la transmission de données par tuyaux

```
let copipe () = new_channel();;

let input chan =
  match sync (receive chan) with
    Some v -> v
  | None -> raise End_of_file
let output chan x = match Event.sync (send chan (Some x))
let close_out chan = sync (send chan None);;
```

Améliorations

- Polariser la communication, fermer définitivement les canaux.
- Tamporisation des communications (très important)

Cas d'étude : crible d'Ératosthène

Création de coprocessus (plutôt que de processus).

Communication

- par tuyau (interne)
  - pas de gain.
- Communication directe
  - synchronisation nécessaire.
  - plus lent : plusieurs appels système à chaque élément + changement de contexte (en lecteur et écrivant).
- Communication directe temporisée
  - Synchronisation compliquée (librairie ?)
  - Ni gain, ni perte.

Coût du changement de contexte prépondérant

Les Futex : *Fast user-space Mutex*

Par défaut `Mutex.lock` coûte un appel system, même si le verrou est libre.

En pratique un coprocessus peut prendre et rendre de nombreuses fois le même verrou avant un entrelacement, et cela devrait pouvoir se faire efficacement.

Les Futex combinent un test local, et en cas de compétition potentielle (le lock pourrait être pris) effectue effectivement un appel système qui gère la compétition.

L'astuce est d'utiliser un mot en mémoire partagée entre tous les coprocessus et des instructions atomiques pour incrémenter-décrémenter un compteur (généralisation de test-and-set).

La prise d'un verrou libre devient alors un calcul très peu coûteux.

Retenir

Partage de la mémoire

La synchronisation est difficile

- Utiliser des primitives de haut niveau.
- Bien réfléchir à la communication dès le départ.
- Ne pas utiliser les coprocessus si ce n'est pas nécessaire...

La synchronisation coûte

- La communication coûte (besoin de synchronisation).
- Les coprocessus sont idéals pour échanger de grosses données par passage d'adresse (là où plusieurs processus devraient faire un communication compliquées ou coûteuse).
- Il faut limiter le besoin de synchronisation en tamporisant les échanges si possible (comme pour les lectures/écriture dans des fichiers).