Cours 3 : Threads (modèle préemptif)

1. Sémaphores et moniteurs
2. Threads en O'Caml, Mutex, Condition
3. Threads en Java

Sémaphores (1)

(Dijkstra 1965)
Un sémaphore est une variable entière s ne pouvant prendre que des valeurs positives (ou nulles). Une fois s initialisé, les seules opérations admises sont : $P(s)$ et $S(s)$

- $P(s)$: si $s > 0$ alors $s := s - 1$ ($await\ s\ do\ s := s - 1$), sinon l'exécution du processus ayant appelé $P(s)$ est suspendue.
- $S(s)$: si un processus a été suspendu lors d'une exécution antérieure d'un $S(s)$ alors le réveiller, sinon $s := s + 1$.

s correspond au nombre de ressources d'un type donné.

Sémaphores (2)

- Un sémaphore ne prenant que les valeurs 0 ou 1 est appelé *sémaphore binaire*.
- Les primitives $P(s)$ et $S(s)$ s'excluent mutuellement si elles portent sur le même sémaphore (l'ordre n'est donc pas connu).
- La définition de S ne précise pas quel processus est réveillé s'il y en a plusieurs.

Exclusion mutuelle

On peut utiliser les sémaphores pour l'exclusion mutuelle. Les deux processus `p 1` et `p 2` sont exécutés en parallèle grâce à la bibliothèque de `threads` d'OCaml.

```
let s = ref 1;;

let p i =
  while true do
  begin
    wait(s);
    crit();
    signal(s);
    suite()
  end
;;

Thread.create p 1;;
Thread.create p 2;;
```

Explications

Dans cet exemple, si un processus veut entrer en section critique, il entrera en section critique si :

- il n'y a que 2 processus (si P_1 est suspendu alors P_2 est en section critique);
- si aucun processus ne s'arrête en section critique (si P_2 est dans `crit`) alors il exécutera $S(s)$.

Cette vérification ne fonctionne plus à partir de 3 processus. Il peut y avoir privation si le choix du processus se fait toujours en faveur de certains processus. Par exemple, si le choix s'effectue toujours en faveur du processus d'indice le plus bas, P_1 et P_2 pourraient se liguer pour se réveiller mutuellement, P_3 étant alors indéfiniment suspendu.

Moniteurs

Hoare (1973)

- mécanisme de haut niveau associant données et fonctions (points d'entrée) les manipulant
- un seul processus/thread actif dans un moniteur à un instant donné (les autres appelants sont bloqués) : exclusion mutuelle
- possibilité de relacher le moniteur par le processus actif (wait) ou de réveiller un processus l'ayant relaché (signal)

Exemple

- une variable entière `v` (+ une constante `vmax`)
- 2 points d'entrées : les fonctions `incr()` et `decr()`
- contrainte $v \in [0, vmax]$

```
let v = ref 10 and vmax = 100;;
let incr() =
  while !v == vmax do done;
  v := !v + 1
```

```
let decr() =
  while !v == 0 do done;
  v := !v - 1
```

ou mieux

```
 let c1 = Condition.create() and   c2 = Condition.create() in

let incr() =
  while !v == vmax do wait(c1) done;
  v := !v + 1;
  signal(c2)
```

```
let decr() =
  while !v == 0 do wait(c2) done;
  v := !v - 1;
  signal(c1)
```

Moniteurs

- Mutex + Condition (variables conditionnelles) permet de construire des moniteurs en O'Caml
- En Java (méthodes ou blocs synchronisés)

Processus légers (Thread) en O'Caml

Modèle de parallélisme à mémoire partagée

- contexte d'exécution sur une même machine virtuelle
- même zone mémoire ($\neq$ `fork`)
- ne va pas plus vite
- sert à exprimer des algos concurrents

Bibliothèque O'Caml

La bibliothèque sur les threads d'Objective CAML est découpée en cinq modules dont les quatre premiers définissent chacun des types abstraits :

- module Thread : création et exécution de processus légers (type Thread.t);
- module Mutex : création, pose et libération de verrous (type Mutex.t);
- module Condition : création de conditions (signaux), attente et réveil sur condition (type Condition.t);
- module Event : création de canaux de communication (type 'a Event.channel), des valeurs y transitant (type 'a Event.event), et des fonctions de communication.
- module ThreadUnix : fonctions d'entrées-sorties du module Unix non bloquantes. (déprécié)

Compilation avec threads

- basés sur les threads système (posix 1003) : byte-code et natif
- internes à la MV O'Caml : byte-code
- ocamlc ou toplevel

```
$ ocamlc -thread -custom threads.cma  fichiers.ml -cclib -lthreads
$ ocamlmktop -thread -custom -o threadtop unix.cma threads.cma -cclib -lth
$ ./threadtop -I +threads
```

- ocamlopt (+ threads posix)

```
$ ocamlc -thread -custom threads.cma fichiers.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamltop -thread -custom threads.cma fichiers.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamlcopt -thread threads.cmxa fichiers.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
```

Relations entre threads

1. sans relation
2. avec relation mais sans synchronisation
3. relation d’exclusion mutuelle
4. relation d’exclusion mutuelle avec communication

Module Thread

- `create f a` crée le processus de l'application de `f` sur `a`;
- `self ()` retourne le processus courant et `id t` son identificateur.
- `exit ()` termine le processus courant et `kill t` le processus indiqué.

```
# Thread.create;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
# Thread.self;;
- : unit -> Thread.t = <fun>
# Thread.exit;;
- : unit -> unit = <fun>
# Thread.join;;
- : Thread.t -> unit = <fun>
# Thread.delay;;
- : float -> unit = <fun>
```

Module Mutex

module Mutex

- `create ()` crée un verrou d'exclusion mutuelle (mutex);
- `lock m` capture un "mutex",
- `try_lock m` capture si possible un verrou, retourne `true` si cela est fait et `false` sinon,
- et `unlock` libère un verrou.

Module Condition

module Condition

- `create ()` : crée une variable condition;
- `wait c m` : libère `m` et suspend le processus appelant sur la variable condition `c`, `signal c` réveille un des processus suspendus sur la variable condition `c`, et `broadcast c` réveille tous les processus suspendus sur `c`.

```
# Condition.wait;;
- : Condition.t -> Mutex.t -> unit = <fun>
# Condition.signal;;
- : Condition.t -> unit = <fun>
# Condition.broadcast;;
- : Condition.t -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (1)

```
# let f = Queue.create ()  and  m = Mutex.create () ;;
val f : '_a Queue.t = <abstr>
val m : Mutex.t = <abstr>

# let produire i p d =
   incr p ;
   Thread.delay d ;
   Printf.printf "Le producteur (%d) a produit %d\n" i !p ;
   flush stdout ;;
val produire : int -> int ref -> float -> unit = <fun>

# let stocker i p =
   Mutex.lock m ;
   Queue.add (i,!p) f ;
   Printf.printf "Le producteur (%d) a ajout son %d-ime produit\n" i !p ;
   flush stdout ;
   Mutex.unlock m ;;
val stocker : int -> int ref -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (2)

```
# let producteur i =
   let p = ref 0 and d = Random.float 2.
   in while true do
        produire i p d ;
        stocker i p ;
        Thread.delay (Random.float 2.5)
      done ;;
val producteur : int -> unit = <fun>
```

Exemple 1: Producteur/Consommateur (3)

```
# let consommateur i =
   while true do
     Mutex.lock m ;
     ( try
         let ip, p = Queue.take f
         in Printf.printf "Le consommateur(%d) " i ;
            Printf.printf "a retir le produit (%d,%d)\n" ip p ;
            flush stdout ;
       with
          Queue.Empty ->
            Printf.printf "Le consommateur(%d) " i ;
            print_string "est reparti les mains  vides\n" ) ;
     Mutex.unlock m ; Thread.delay (Random.float 2.5)
   done ;;
val consommateur : int -> unit = <fun>
```

Exemple 1: Producteur/Consommateur (4)

Le programme de test suivant crée quatre producteurs et quatre consommateurs.

```
for i = 0 to 3 do
  ignore (Thread.create producteur i);
  ignore (Thread.create consommateur i)
done ;
while true do Thread.delay 5. done ;;
```

Exemple 2 : Producteur/Consommateur (1)

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
```

On modifie la fonction de stockage du producteur en lui ajoutant l'émission d'un signal.

```
# let stocker2 i p =
   Mutex.lock m ;
   Queue.add (i,!p) f ;
   Printf.printf "Le producteur (%d) a ajoute son %d-ieme produit\n" i !p ;
   flush stdout ;
   Condition.signal c ;
   Mutex.unlock m ;;
val stocker2 : int -> int ref -> unit = <fun>
# let producteur2 i =
   let p = ref 0 in
   let d = Random.float 2.
   in while true do
        produire i p d;   stocker2 i p;
        Thread.delay (Random.float 2.5)
      done ;;
val producteur2 : int -> unit = <fun>
```

Exemple 2 : Producteur/Consommateur (2)

L'activité du consommateur se fait alors en deux temps:

- attendre qu'un produit soit disponible
- puis emporter le produit.

Le verrou est pris quand l'attente prend fin et il est rendu dès que le consommateur a emporté son produit. L'attente se fait sur la variable c.

```
# let attendre2 i =
   Mutex.lock m ;
   while Queue.length f = 0 do
     Printf.printf "Le consommateur (%d) attend\n" i ;
     Condition.wait c m
   done ;;
val attendre2 : int -> unit = <fun>
```

Exemple 2 : Producteur/Consommateur (3)

```
# let emporter2 i =
   let ip, p = Queue.take f in
     Printf.printf "Le consommateur (%d) " i ;
     Printf.printf "emporte le produit (%d, %d)\n" ip p ;
     flush stdout ;
     Mutex.unlock m ;;
val emporter2 : int -> unit = <fun>
# let consommateur2 i =
   while true do
     attendre2 i;
     emporter2 i;
     Thread.delay (Random.float 2.5)
  done ;;
val consommateur2 : int -> unit = <fun>
```

Notons qu'il n'est plus besoin de vérifier l'existence effective d'un produit puisqu'un consommateur doit attendre l'existence d'un produit dans la file d'attente pour que sa suspension prenne fin. Vu que la fin de son attente correspond à la prise du verrou, il ne court pas le risque de se faire dérober le nouveau produit avant de l'avoir emporté.

Processus légers (Thread) en Java

Modèle de parallélisme à mémoire partagée

- contexte d'exécution sur une même machine virtuelle
- même zone mémoire ($\neq$ `fork`)
- ne dépend pas du système (Windows 3.1)
- possède un nom et une priorité
- préemption ou coopération à priorité égale
  $\Rightarrow$ selon l'implantation
  dépend de l'implantation de la JVM
- ne va pas plus vite
- sert à exprimer des algos concurrents

Durée de vie

Un programme exécutant plusieurs threads (*user*) attendra que tous soient finis avant de se terminer.
Un thread s'exécute jusqu'au moment où

- une thread de plus grande priorité devient "exécutable";
- une méthode `yield` ou `sleep` est lancée;
- ou que son quota de temps a expiré pour les systèmes implantant un temps partagé.

préemptif et coopératif

Il y aura une différence de comportement selon l'implantation de la JVM sur les systèmes gérant le temps partagé et les autres. Dans le premier cas le multi-threading sera préemptif alors que dans le second cas il sera coopératif. Pour écrire un code portable sur différents systèmes, il est préférable de ne pas faire d'hypothèse sur l'implantation des threads.

Moniteurs

Les threads JAVA implantent un mécanisme de “moniteurs” protégeant l’accès aux objets “synchronisés”. :

- L’exclusion mutuelle s’effectue sur l’appel des méthodes déclarées `synchronized` dans la définition de la classe de l’objet receveur.
- Une seule exécution d’une méthode `synchronized` peut être effectuée en même temps, les autres appels sont alors bloqués.
- Il existe deux méthodes primitives principales `wait` et `notify` qui permettent une communication entre threads ayant accès au même moniteur.

Création et Exécution

2 possibilités :

- sous-classer la classe **Thread**
  et redéfinir la méthode `public void run()`
  ```
  MCThread x = new MCThread (); x.start();
  ```
- Implanter l'interface **Runnable**
  et implanter `public void run`
  ```
  MCTheadI x = new MCThreadI();
  MCThreadI y = new Thread(x,"Nom"); y.start()
  ```

Quand **start()** retourne, la tâche termine.

Exemple (1)

```
class MyThread extends Thread {
  int num;
  int st = 1;
  MyThread(int n) { num = n;}

  public void run() {
    int i=0;
    while (st < 5000001) {
      if ((st % 1000000) == 0) {
        System.out.println("MT" + num+ " trace = "+st);
      }
      st++;
      yield();
    }
  }
}
```

Exemple (2)

```
class TestThreads {

  public static void main (String []args) {

    MyThread mt1 = new MyThread(1);
    MyThread mt2 = new MyThread(2);

    mt1.setPriority(10);
    mt2.setPriority(10);

    mt1.start();
    mt2.start();
  }
}
```

Méthodes sur les threads

- **start()** (**stop()** dépréciée)
- (dépréciées **suspend()** et **resume()**)
- **wait()** et **notify** (synchronisation)
- **sleep(i)** en millisecondes
- **yield()**
- **join()**
- **setPriority(i)**

Etats

Un thread une fois créé peut être dans les états suivants :

- exécutable
- en cours d'exécution
- bloqué
- mort

Méthodes (1)

- La méthode `start()` rend un thread “exécutable”, et lance la méthode `run()`. Le thread se termine quand cette méthode a fini son exécution.
- La méthode `stop()` arrête un thread (dépréciée).
- Les méthodes `suspend()` (dépréciée), `sleep(i)` et `wait` met un thread dans l’état bloqué.
- Les méthodes `resume()` (dépréciée) et `notify()` fait passer un thread de l’état “bloqué” à l’état “exécutable”.

Méthodes (2)

- La méthode `yield()` fait passer de l'état "en cours d'exécution" à l'état exécutable.
- La méthode `join()` attend la fin d'un thread.
- La méthode `setPriority(i)` modifie la priorité d'un thread (valeur entre `MIN_PRIORITY` et `MAX_PRIORITY`.

Relations entre threads

1. sans relation
2. avec relation mais sans synchronisation
3. relation d'exclusion mutuelle
4. relation d'exclusion mutuelle avec communication

Exemple de deux threads sans relatio

```
public class SR {
  public static void main ( String []s) {
    Thread t1 = new Aff("Bonjour");
    t1.start();
    new Aff("Au revoir").start();
  }
}

class Aff extends Thread {
  String txt;
  Aff (String s){txt=s;}

  public void run () {
    while(true) {
      ES.writeln(txt);
      yield();
    }
} }
```

Relation sans synchronisation

sur une même structure de données à des endroits différents :
Exemple :

- calcul de la nouvelle génération du jeu de la vie par différentes threads.
- ou ouverture d'une connexion sur une *socket*

Synchronisation

Verrou (accès exclusif) sur un objet :

```
  synchronized (o) { ... }
  }

// ou sur une methode

synchronized type nom_methode( ..) { ...}
```

Un seul *thread* peut prendre le verrou (les autres attendent).
Le verrou est rendu à la sortie du bloc de synchronisation.

Relation d'exclusion mutuelle

moniteurs: :

Dès que le verrou est pris :

1. appel d'une méthode `synchronized`
   il garantit qu'une seule méthode "synchronisée" peut être appelée sur cet objet
2. bloc `synchronized` sur un objet
   il verouille l'objet.

Moniteurs (1)

- Un moniteur est associé à une donnée et aux fonctions qui verrouillent l'accès à cette donnée. En Java, cette donnée sera un objet et les fonctions les méthodes déclarées `synchronized` dans la définition de la classe de cet objet.
- Il y a création d'un moniteur pour chaque objet qui possède au moins une méthode `synchronized`. Quand un thread a pris le moniteur sur un objet (c'est-à-dire est entré dans

Moniteurs (2)

- l'exécution d'une méthode "synchronisée"), les autres threads voulant exécuter une méthode "synchronisée" sur cet objet sont bloqués.
- Quand le premier thread a terminé l'exécution du code de la méthode "synchronisée" elle libère le moniteur qui peut alors être pris par un autre thread.
- Les moniteurs permettent aussi la communication entre threads par le mécanisme d'attente/notification (`wait` et `notify`).

Communication

à l'intérieur d'un bloc de synchronisation

- **o.wait()** : relache le verrou et attend une *notification*
- **o.notify()** : relance une tâche en attente (une au hasard). Elle doit en premier réacquérir le verrou.
- **o.notifyAll()** : relance toutes les tâches.

Exclusion mutuelle avec communicat

Producteur/Consommateur :

```
// thread du producteur
entree code sync
  while(buffer plein)
    wait()
  produit_1()
  notify()
```

```
// thread consommateur
  entree code sync
  while (buffer vide)
    wait()
  consomme_1()
  notify()
```

producteur/consommateur (1)

```
public class Sync2 {
  public static void main (String [] args) {
     Producteur p = new Producteur();
     p. start();
     Consommateur c1 = new Consommateur(p,"1"); c1.start();
  }
}

class Consommateur extends Thread {
  Producteur mien;
    String nom = "";
  Consommateur(Producteur un, String n){mien=un;nom=n;}

  public void run() {
    while(true) {
       String r = mien.consomme();
       System.out.println("Consommation "+"("+nom+") : "+r);
   }
 }
}
```

producteur/consommateur (2)

```
class Producteur extends Thread {
  private String [] buffer = new String [8];
  private int ip = 0;
  private  int ic = 0;

  public void run() {
    while(true) {
      produce();
    }
  }

}
```

producteur/consommateur (3)

```
synchronized void produce() {
    while (ip-ic+1 > buffer.length) {System.out.println("PLEIN");
     try{wait();} catch(Exception e) {} }
  buffer[ip % 8] = "Machine "+ip;
  System.out.println("produit : ["+(ip % 8)+"]  "+buffer[ip % 8]);
  ip++;
  notify();
}

synchronized String consomme() {
    while (ip == ic) {System.out.println("VIDE");
      try{wait();} catch(Exception e) {}}
   notify();
   return buffer[ic++%8];
}
}
```