TP9

November 22, 2021

1 Programmation fonctionnelle

1.1 TP 9 : Programmation multicoeur

1.1.1 Etienne Lozes - 2021

Pour ce TP vous pouvez travailler avec la librairie `Thread` du compilateur Caml "officiel". On rappele que pour compiler `toto.ml` il faut taper

`ocamlc -thread unix.cma threads.cma toto.ml`.

1.2 Exercice 1 : Alphabet mélangé

Écrivez un programme qui lance 26 threads. Chacun affiche une lettre de l'alphabet et termine. On affiche enfin un point derrière ces 26 lettres.

Exemple d'affichage:

`vxdjhnfqlukbcgoazyeiprtsmw.`

1.3 Exercice 2 : Verouillage local

Deux threads se partagent un tableau `Data`. De manière répétée, ils choisissent au hasard deux valeurs dans ce tableau et échangent leur place. On veut s'assurer que chaque permutation est "atomique" et qu'on retrouve bien à la fin toutes valeurs qui étaient présentes au début. On pourrait mettre un verrou global sur le tableau, mais pour améliorer le parallélisme on adopte une approche par verouillage local: chaque case du tableau dispose de son propre verrou.

Complétez le code ci-dessous pour implémenter cette approche. Ne modifiez que la fonction swap, le reste sert à tester votre solution.

[ ]:
```
(** un module qui simule un tableau Data tel que tout accès au tableau suspend␣
 ↪le thread   **)
(** nécessaire pour pouvoir tester votre solution, à cause du fait que␣
 ↪l'ordonnanceur      **)
(** de thread Caml n'est pas préemptif.                                    ␣
 ↪           **)
(** Avec domain on aurait pu utiliser directement un 'a array et les opération␣
 ↪a.(i) et    **)
```

```
(** a.(i) <- v standard.
 ↪             **)

module type DATA = sig
  type value
  val size : int
  val get : int -> value
  val set : int -> value -> unit
  val check_integrity : unit -> bool
end

module Data : DATA = struct
  type value = int
  let size = 10
  let a  = Array.init size (fun i -> i)
  let get k = Thread.delay (Random.float 0.001); a.(k)
  let set k v = Thread.delay (Random.float 0.001); a.(k) <- v
  let check_integrity () =
    let b = Array.make size false in
    Array.iter (fun i -> b.(i) <- true) a;
    Array.fold_left (&&) true b
end



(** LE TABLEAU DES VERROUS : chaque verrou protège une case de Data **)

let mutex = Array.init Data.size (fun _ -> Mutex.create ())



(** LA FONCTION A MODIFIER **)

let swap i j =
  (* AJOUTER UNE SECTION CRITIQUE A CETTE FONCTION! *)
  let vi, vj = Data.get i, Data.get j in
  Data.set i vj;
  Data.set j vi



(** TESTEUR DE VOTRE SOLUTION : NE PAS MODIFIER **)

let nb_iterations = 100

let swapper () =
  for _ = 1 to nb_iterations do
```

```
    let i, j = Random.int Data.size, Random.int Data.size in
    swap i j
  done

let test () =
  let t1 = Thread.create swapper () in
  let t2 = Thread.create swapper () in
  Thread.join t1;
  Thread.join t2;
  if Data.check_integrity()
  then print_string "ok\n"
  else print_string "races!\n"

let () = test ()
```

1.4 Exercice 3 : Le chat et le canari

1. Il y a 10 arbres numérotés de 0 à 9. Le canari peut voler d'un arbre à n'importe quel autre arbre, le chat peut seulement sauter d'un arbre à un voisin. L'utilisateur contrôle le canari (on tape l'arbre sur lequel on veut aller au clavier) et l'ordinateur le chat. Le jeu s'arrête lorsque le chat attrape le canari. On pourra par exemple supposer que le chat met une seconde pour changer d'arbre.
2. Pimentez le jeu! toutes les 5 secondes arrive un nouveau chat.

Fonctions utiles: `Thread.delay`, `read_int`, `print_string`, `print_newline`. Évitez les fonctions entrées-sorties bufferisées (`Printf.printf`, notamment) et pensez à forcer l'affichage avec `print_newline`. Exceptionnellement, pour cet exercice, les data races sont autorisées (mais vous avez le droit de vous entrainer avec `Atomic`).

1.5 Exercice 4: Verrou "lecture/écriture"

Complétez le module `RWMutex` dans le code ci-dessous. Plusieurs threads peuvent acquérir simultanément le verrou en lecture avec `lock_r`, mais un seul thread peut l'acquérir en écriture avec `lock_w`.

Indication: vous devrez utiliser des variables de condition.

[ ]:
```
module type RWMUTEX = sig
  type t
  val create : unit -> t
  val lock_r : t -> unit
  val lock_w : t -> unit
  val unlock : t -> unit
end


module RWMutex : RWMUTEX = struct
  (* A COMPLETER! DEFINITIONS PAR DEFAUT A IGNORER *)
  type t = unit
```

```
  let create () = ()
  let lock_r _rwmut = ()
  let lock_w _rwmut = ()
  let unlock _rwmut = ()
end

module type DATA = sig
  val read : unit -> unit
  val write : unit -> unit
  val achieved_several_readers : unit -> bool
end

module Data : DATA = struct
  let nb_readers = ref 0
  let writing = ref false
  let several_readers = ref false
  let read () =
    if !writing then failwith "reader meets writer";
    nb_readers := !nb_readers + 1;
    if !nb_readers >= 2 then several_readers := true;
    Thread.delay (Random.float 0.001);
    nb_readers := !nb_readers - 1
  let write () =
    if !writing then failwith "writer meets writer";
    if !nb_readers <> 0 then failwith "writer meets reader";
    writing := true;
    Thread.delay (Random.float 0.001);
    writing := false
  let achieved_several_readers () = !several_readers
end

module RWMutexTester (RWMutex:RWMUTEX) = struct
  let nb_iterations = 100
  let nb_readers = 2
  let nb_writers = 3
  let run () =
    let rwmut = RWMutex.create () in
    let reader () =
      for _ = 1 to nb_iterations do
        Thread.delay (Random.float 0.001);
        RWMutex.lock_r rwmut;
        Data.read();
        RWMutex.unlock rwmut
      done in
    let writer () =
      for _ = 1 to nb_iterations do
        Thread.delay (Random.float 0.001);
```

```
        RWMutex.lock_w rwmut;
        Data.write();
        RWMutex.unlock rwmut
      done in
    let readers =
      Array.init nb_readers (fun _ -> Thread.create reader ()) in
    let writers =
        Array.init nb_writers (fun _ -> Thread.create writer ()) in
    Array.iter Thread.join readers;
    Array.iter Thread.join writers;
    assert (Data.achieved_several_readers ())
end

let () =
  let open RWMutexTester(RWMutex) in run ()
```