Système – ENS L3 Info.
Année 2012-2013

Examen
30 mai 2013

L'énoncé est composé de 5 pages. Cette épreuve est prévue pour une durée de 2h. Les notes de cours et de TDs sont autorisées.

Vol de tâche

Le but de cet exercice est de réaliser une bibliothèque OCaml qui permet d'exécuter des tâches indépendantes en parallèle en utilisant un ensemble de threads borné. Les tâches seront des fonctions OCaml avec l'argument sur lequel elles doivent être appliquées. Cette bibliothèque devra respecter l'interface suivante :

```
module type Pool = sig
  val nb_threads : int
  val start : ('a -> 'b) -> 'a -> unit
  val run : ('a -> 'b) -> 'a -> unit
end
```

La valeur `nb_threads` contient le nombre de threads qui vont exécuter les tâches en parallèle. La fonction `start f x` crée les `nb_threads` threads et demande le calcule de `f x`. Enfin, `run f x` ajoute la nouvelle tâche `f x`.

Question 1. Écrire une première implantation de l'interface `Pool` qui va contenir un ensemble de tâche à exécuter et les `nb_threads` threads qui iront y chercher du travail a exécuter. Pour représenter l'ensemble de tâche, vous pouvez par exemple utiliser le module `Queue` d'OCaml. Ainsi, cette implantation pourra commencer de la façon suivante :

```
module Pool (Config: sig val n: int end) = struct
  let nb_threads = Config.n
  let tasks = Queue.create ()
  ...
```

L'ordre d'exécution des tâches n'a pas d'importance. On ne cherche pas à déterminer l'arrêt du programme lorsqu'il y a plus lorsqu'il n'y a plus de tâches à exécuter. On supposera que la mémoire est séquentiellement consistante.

Dans l'implantation précédente, il peut y avoir beaucoup de conflits lors de l'accès à la queue partagée. Pour réduire les conflits, chaque thread peut avoir un ensemble de tâches à exécuter et lorsque cet ensemble est vide, il va chercher à voler une tâche à un autre ensemble.

Question 2. Indiquer précisément les modifications à faire à l'implantation précédente pour réaliser du vol de tâche.

Considérons que les ensembles de tâches sont représentés par des queues à double entrées dont l'interface est la suivante :

```
module type DEQueue = sig
  type 'a t
  val create: unit -> 'a t
  val push_bottom: 'a -> 'a t -> unit
  val pop_bottom: 'a t -> 'a option
  val pop_top: 'a t -> 'a option
end
```

Chaque thread ajoute et retire par les tâches par le bas de la queue et vole des tâches dans les autres ensembles par le haut des autres queues.

Une implantation correcte pour ces queues à double entrées est donnée ci-dessous. Elle commence par l'implantation de tableaux circulaires :

```
module CircularArray = struct
  type 'a t =
      { content : 'a array;
        capacity : int; }

  let create n init =
    { content = Array.create n init;
      capacity = n; }

  let get t i = t.content.(i mod t.capacity)

  let put t i v = t.content.(i mod t.capacity) <- v

  let resize t bottom top =
    let new_t = create (t.capacity lsl 1) (get t bottom) in
    for i = top to bottom do
      put new_t i (get t i)
    done;
    new_t
end
```

Les tableaux sont implantés par un enregistrement qui contient le tableaux (`content`), la taille du tableau (`capacity`). Les fonctions `get` et `put` permettent de lire et écrire dans le tableau modulairement. La fonction `resize` crée un tableau deux fois plus grand que le tabeau passé en argument et qui contient les valeurs comprisent entre `top` et `bottom`.

Les queues sont implantés avec des tableaux circulaires et les indices `top` et `bottom` qui représentent les deux bouts de la queue. Plus précisement, `bottom` contient l'indice du premier emplacement vide dans le tableau et la taille de la queue est donnée par la différence `bottom - top` (la valeur de `bottom` est supperrieur à la valeur de `top` lorsque la queue est non vide). Comme nous utilisont des tableaux circulaires, la valeur de `top` ne sera jamais décrémentée.

```
module DEQueue_seq : DEQueue = struct
```

```
type 'a t =
    { mutable queue: 'a CircularArray.t;
      top : int ref;
      bottom : int ref;
      mutex: Mutex.t; }

let create () =
  { queue = CircularArray.create (1 lsl 4) (Obj.magic None);
    top = ref 0;
    bottom = ref 0;
    mutex = Mutex.create (); }

let atomic f q =
  Mutex.lock q.mutex;
  let v = f q in
  Mutex.unlock q.mutex;
  v

let push_bottom' v q =
  let old_bottom = !(q.bottom) in
  let old_top = !(q.top) in
  let size = old_bottom - old_top in
  if q.queue.CircularArray.capacity - 1 <= size then
    q.queue <- CircularArray.resize q.queue old_bottom old_top;
  CircularArray.put q.queue old_bottom v;
  q.bottom := old_bottom + 1

let push_bottom v q = atomic (push_bottom' v) q

let pop_bottom' q =
  let old_bottom = !(q.bottom) in
  let new_bottom = old_bottom - 1 in
  q.bottom := new_bottom;
  let old_top = !(q.top) in
  let new_top = old_top + 1 in
  let size = old_bottom - old_top in
  if size - 1 < 0 then begin
    q.bottom := old_top;
    None
  end else begin
    let v = CircularArray.get q.queue new_bottom in
    if size - 1 > 0 then begin
      Some v
    end else begin
      q.top := new_top;
      q.bottom := new_top;
      Some v
```

```
      end
    end

  let pop_bottom q = atomic pop_bottom' q

  let pop_top' q =
    let old_top = !(q.top) in
    let new_top = old_top + 1 in
    let old_bottom = !(q.bottom) in
    let size = old_bottom - old_top in
    if size <= 0 then None
    else
      let r = CircularArray.get q.queue old_top in
      q.top := new_top;
      Some r

  let pop_top q = atomic pop_top' q
end
```

Cette implantation est correcte car chaque fonction de `DEQueue_seq` qui modifie une même queue est exécutée en exclusion mutuelle grâce au mutex associé à chaque queue.

En utilisant l'hypothèse que les vols dans la queue sont fait par le haut et que les opérations faites par le thread associé à la queue sont fait par le bas, le nombre de synchronisations peut être réduit.

Question 3. Expliquer le principe d'une implantation efficace.

Question 4. Réaliser cette implantation. Vous pouvez supposer qu'il existe une fonction `atomic_set: 'a ref -> 'a -> unit` qui réalise une affectation de façon atomique et une fonction `compare_and_set: a' ref -> 'a -> 'a -> bool` telle que `compare_and_set x old_v new_v` réalise l'affection `x := new_v` si la valeur de la référence `x` n'a pas été modifiée depuis la dernière fois où elle valait `old_v`.

Question 5 (optionnel)**.** Détecter la fin du traitement de toutes les tâches.

Modélisation d'un verrou et d'écritures concurrentes

Le but de cet exercice est de modéliser le mécanisme de verrou ou *Mutex* d'Unix. Vous êtes libre d'effectuer cette modélisation en Lustre ou sous forme de systèmes de transition composés en parallèle ou en utilisant le langage de Cubicle. Pour simplifier la modélisation, on considère que le seul type de données nécessaire est le type `bool`.

Le rôle d'un verrou est d'assurer qu'une portion de code s'exécute en section critique, c'est-à-dire qu'au plus un processus exécute cette section à un instant donné.

```
(* creation/initialisation du verrou *)
let m = Mutex.create() in
...
```

```
(* un process P1 *)
Mutex.lock(m);
...
(* partie P1.critique *) en exclusion mutuelle *)
...
Mutex.unlock(m)
```

Un modèle simple de verrou est un composant à deux entrées : `lock`, `unlock` correspondant chacun à l'exécution de `Mutex.lock(m)` et `Mutex.unlock(m)`. Ce système produit une sortie `ok` avec la convention qu'il est présent (ou vrai) lorsque `Mutex.lock(m)` a réussi, absent (faux) sinon.

Question 6. Proposer un modèle de verrou. Dans le cas de Lustre, la signature sera :

```
node mutex(lock, unlock: bool) returns (ok: bool)
```

On considère maintenant deux processus P_1 et P_2 pouvant être exécutés en parallèle. Chacun a accès au verrou partagé m. Du point de vue de la modélisation, P_1 et P_2 produisent respectivement des sorties (c'est-à-dire des requêtes) `lock1`, `unlock1` et ont en entrée `ok1` pour P_1 ; des sorties `lock2`, `unlock2` et l'entrée `ok2` pour P_2.

```
node P1(ok1:bool) returns (lock1, unlock1: bool)
node P2(ok2:bool) returns (lock2, unlock2: bool)
```

Ces deux processus écrivent dans une mémoire partagée. Une mémoire peut être modélisée par un processus non déterministe qui choisit, de manière interne, de servir une écriture plutôt qu'une autre. Le système principal peut être modélisé par la mise en parallèle de quatre processus :

```
node main(oracle:bool) returns (ok1, ok2: bool)
  var lock1, unlock1, lock2, unlock2, lock, unlock: bool;
  let
    (lock1, unlock1) = P1(ok1);
    (lock2, unlock2) = P2(ok2);
    (ok1, ok2) = mutex(lock, unlock);
    (lock, unlock) = memoire(oracle, lock1, lock2, unlock1, unlock2);
```

L'oracle est un signal booléen non déterministe pour modéliser le choix interne pour l'écriture mémoire. Dans un premier temps, on fait l'hypothèse que P_1 et P_2 sont exécutés sur une machine monoprocesseur, c'est-à-dire où une seule instruction peut être effectué à un instant donné.

Question 7. Comment peut-on exprimer cette hypothèse sur les entrées/sorties `lock1`, `unlock1`, `ok1`, `lock2`, `unlock2`, `ok2` (sous forme d'une propriété booléenne invariante) ?

Question 8. Proposer une modélisation de la mémoire, par exemple, sous la forme d'une définition du noeud `memoire`, sous cette hypothèse d'exécution.

Question 9. Proposer une modélisation de la mémoire dans le cas général.

Question 10. Quelle est la propriété de sûreté (c'est-à-dire vraie à chaque instant) que doivent respecter les sorties `ok1` et `ok2` pour se convaincre que l'implémentation du verrou est correcte ?