Système – ENS L3 Info.
Année 2014-2015

Examen

28, mai 2015

L'énoncé est composé de 4 pages. Cette épreuve est prévue pour une durée de 2h30. Les notes de cours et de TDs imprimées ou sur votre ordinateur sont autorisées.

Le code pourra être écrit en OCaml ou en C. La lisibilité du code et la qualité de la rédaction seront prises en compte.

1 Opérateur Map/Fold parallèle

L'objectif de cet exercice est d'implémenter un opérateur parallèle qui applique à tous les éléments d'un tableau une fonction f puis agrège les résultats en utilisant une fonction associative et commutative g. Si t est un tableau de n éléments, la fonction $mapfold(f)(g)(i)(t)$ rend une valeur v telle que :

$$v = g\left(...(g\left(g\left(i\right)\left(f\left(t(0)\right)\right)\right)\left(f\left(t(1)\right)\right)...\right)\left(f\left(t(n)\right)\right)$$

Vous pourrez supposer que l'opération g est associative et commutative. La signature en OCaml sera :

```
val map_fold : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a array -> 'b
```

Question 1. Proposer une implémentation parallèle de la fonction `map_fold` utilisant les processus légers (threads).

Question 2. Proposer une implémentation d'une fonction de calcul du nombre de zéros d'un tableau et de la somme des éléments d'un tableau.

On étend maintenant la fonction précédente en une fonction de signature :

```
val map_fold_max_threads :
  int -> ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a array -> 'b
```

telle que `map_fold_max_threads n f g i t` retourne le même résultat que, `map_fold` mais où `n` est le nombre total de threads créés durant l'exécution.

Question 3. Quelle solution proposez-vous afin qu'aucun thread ne soit inoccupé alors que tous les calculs ne sont pas terminés ?

2 Le coiffeur endormi

Ce problème a été posé par Dijkstra. Il a pour cadre un salon de coiffure, où `nb_chaises` sont présentes, ainsi qu'un fauteuil et un coiffeur. En l'absence de clients, le coiffeur s'endort sur son fauteuil. Lorsqu'un client arrive, il réveille le coiffeur, et celui-ci le coiffe. Les clients suivants, s'ils le peuvent, s'assoient sur les chaises, en attendant leur tour ; s'il n'y a plus de chaises disponibles, ils quittent le salon.

Le problème consiste à écrire un programme qui pilote le coiffeur et les clients, en évitant les interblocages.

En termes informatiques, les clients invoquent une fonction `obtenir_coupe`, et le coiffeur une fonction `coiffer`. Lorsque le coiffeur invoque `coiffer`, il doit y avoir exactement en même temps un processus léger (thread) qui invoque `obtenir_coupe`. On suppose les fonctions `obtenir_coupe` et `coiffer` données. Elles ont les signatures suivantes :

```
val coiffer : unit -> unit;
val obtenir_coupe : unit -> unit;
```

Les clients seront implantés par des threads exécutant la fonction `thread_client` et le coiffeur par un thread exécutant la fonction `thread_coiffeur`. La structure du code (sans les synchronisations) de la fonction `thread_client` est la suivante :

```
let thread_client () =
  if !nb_clients = nb_chaises then ()
  else
    nb_clients := !nb_clients + 1;
    obtenir_coupe ();
    nb_clients := !nb_clients - 1
```

Question 4. Quel est le problème avec la version de la fonction `thread_client` donnée ci-dessus. Donner le principe et le code d'une implantation des fonctions `thread_client` (en complétant le code précédent) et `thread_coiffeur`.

Vous pouvez supposer l'existence d'un module `Sem` des sémaphores POSIX avec l'interface suivant.

```
module type SEM =
sig
  type t

  val init : int -> t
  val wait : t -> unit    (* P *)
  val post : t -> unit    (* V *)
  val getvalue : t -> int
end
```

Question 5. Donner les déclarations des variables globales. Définir un programme principal qui simule un salon de coiffure où un nouveau client arrive toutes les secondes.

Question 6. (Bonus) Implémenter le module `Sem` en utilisant le module OCaml `Condition`.

3 `rptr` : répétons dans des tubes

Il s'agit de développer une commande `rptr` qui va répéter son entrée standard sur les entrées standard d'un ensemble de n commandes.

Cette commande `rptr` :

— crée n tubes ;

— lance n commandes qui vont chacune lire depuis un de ces tubes ;

— répète dans chacun de ces tubes ce qu'elle lit depuis son entrée standard.

Les n commandes lancées par `rptr` peuvent accepter des paramètres comme illustré par l'exemple suivant :

```
--> rptr "cat -n" "grep -v foo" wc
```

On utilisera une fonction auxiliaire `decoupe` pour décomposer les arguments de la ligne de commande.

```
val decoupe : string -> string array
(* let decoupe str = Array.of_list (Str.split (Str.regexp "[ \t]+") str) *)
```

Question 7. Expliquez la hiérarchie de processus qui doit être créée pour réaliser la commande `rptr`. Explicitez ce que doit faire chacun des processus. En particulier, notez bien les créations de tubes et les fermetures de descripteurs inutiles.

Nous allons utiliser un tableau `procs` pour mémoriser les informations des n commandes lancées par `rptr`, à savoir :

— le numéro du processus en charge de la commande ;

— le descripteur d'écriture dans le tube sur lequel lit la commande.

— un booléen pour indiquer si le processus est toujours actif.

```
type cmd_proc = {
  cmd_pid : int;
  cmd_pipew : Unix.file_descr;
  mutable cmd_active : bool
}
```

Question 8. Compléter la fonction `run_proc` du squelette suivant pour créer les tubes nécessaires, créer le processus en charge de l'exécution d'une commande, assurer que l'entrée standard de ce processus se fasse depuis son tube, réaliser la mutation de ce processus pour l'exécution de la commande.

```
(* lancement de la commande *)
let run_proc cmd_str = (* ... *)

let rptr () =
  let n = Array.length Sys.argv in
  if n < 2 then exit 1;
  let procs = Array.init (n - 1) (fun i -> run_proc Sys.argv.(i + 1)) in

  (* ... *)
```

La commande `rptr` se doit ensuite de répéter ce qu'elle lit sur son entrée standard dans chacun des tubes la connectant aux commandes. À la rencontre de la fin de fichier sur son entrée standard, la commande `rptr` se termine.

Question 9. Compléter la fonction `rptr` pour réaliser cette recopie de l'entrée standard dans les tubes connectés aux commandes.

Une commande lancée peut se terminer avant la fin de `rptr`, par exemple sur une erreur ou parce que la commande a trouvé ce qu'elle cherchait sur l'entrée standard. Si toutes les commandes lancées se terminent avant la fin de `rptr`, il n'y a plus lieu de continuer.

Question 10. Il s'agit donc d'identifier la terminaison de ces commandes lancées. Deux choix sont possibles :

- utilisation de `wait` ou `waitpid` pour attendre la fin d'un processus fils ;
- définition d'une fonction traitement de signal pour le signal de terminaison d'un processus fils.

Expliquer et motiver, dans le cas présent de la commande `rptr`, le choix le plus judicieux pour détecter la terminaison des processus fils.

Question 11. Fournir le complément à l'implantation actuelle de `rptr` pour mettre fin à l'exécution de `rptr` à la terminaison de toutes les commandes lancées.

Vous pourrez utiliser le champs `cmd_active` de struct `cmd_proc` pour indiquer qu'un processus est terminé.