Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2012-2013**

Examen réparti du 07 novembre 2012

Exercice 1 : Simulation d'un bar (Fair threads)

BC possède un bar très en vue. Dans ce bar on vend des cocktails. Et BC souhaiterait optimiser le nombre de barmen et le nombre de bouteilles présentes dans son bar à chaque instant pour que ses clients soient satisfaits. Il est assez simple de recueillir le nombre de commandes et la nature des cocktails commandés en fonction du temps mais l'optimisation des deux nombres est un problème épineux. BC après quelques temps se dit donc qu'il va simuler son bar à l'aide d'un programme et il fera varier les paramètres pour voir ce que cela donne.

Le cycle des commandes est le suivant :

1. Les commandes se font au comptoir et chaque barman a sa propre file d'attente et à chaque fois qu'un client arrive il prend un ticket (il récupère un numéro et celui ci est incrémenté).
2. Quand le barman arrive il hurle le numéro de client, tous les client en attente vérifient leur numéro et celui qui est servi hurle a son tour son numéro de cocktail.
3. Une fois la commande prise le barman incrémente le nombre de cocktails vendus.
4. après les barmen partent à la recherche des bouteilles pour le cocktails là, les barmen étant des gentlemen ils respectent scrupuleusement l'ordre d'arrivée.
5. Une fois le cocktail réalisé le barman revient et donne le cocktail au client.

On rappelle ici que la fonction `ft_thread_generate` génère un événement pour la totalité de l'instant présent, c'est à dire que **tous** les threads en attente de cet événement le reçoivent !

Le nombre de Barmen sera donné par une MACRO `#define NBBARMEN ...` connue à la compilation du programme de simulation.
Les recettes des cocktails seront aussi globales et définies ainsi :

```
typedef struct {
  int nb_bouteille;
  int * numero bouteille;
  } recette;
recette *recettes;
```

Les listings proposés sont écrits en C, mais vous pouvez répondre aux différentes questions en C, Java ou Ocaml.

Question 1 Expliquer comment vous modéliseriez la file d'attente des barmen en utilisant l'API des fair threads ? On prendra soin d'indiquer quelles variables et quels événements seront gérés.

Question 2 Écrire la fonction `void client(void *numbarman);` qui modélisera les clients.

Question 3 En supposant que l'accès aux différentes bouteilles est régi par des schedulers, et qu'il faut une seconde à un barman pour se servir, écrire une première version de la fonction `void barman(void*)` qui simulera le barman. Numéroter les lignes.

Question 4 Il se trouve que les bouteilles ne sont pas infinies (sic :-( ) et il est possible qu'elles se vident. lorsqu'une bouteille est vide le barman indique ceci à un commis, disons `mB`, qui cours en chercher une nouvelle et indique que la nouvelle est en place à tous ceux qui l'attendait qu'ils peuvent reprendre leur composition. `mB` met 30 secondes à répondre à la demande. `mB` est seul ! Et il répond aux demandes des barmen dans l'ordre dans lequel elle arrivent.

Indiquer quelles structures de données supplémentaires il faut pour modéliser ce phénomène. Indiquer quelles modifications vous apporteriez à la fonction `barman` pour le prendre en compte.

Question 5 On souhaite maintenant chronométrer le temps moyen en secondes passé par commande. Pour faire ceci on va rajouter un thread supplémentaire dans le scheduler associé à chaque barman. Ce thread sera charge de chronométrer le temps qui s'écoule entre une commande et l'arrivée de celle-ci. Pour compter le temps en secondes, on utilisera une fonction `horloge` qui renverra le temps en secondes depuis le premier janvier 1970. Écrire cette fonction nommée `void W(void *)`.

Question 6 Afin de modéliser l'arrivée de clients on va utiliser un thread **POSIX**, `eC`. Le rôle de ce thread POSIX sera de répéter à l'infini :

- Tirer au hasard (fonction `rand()`) un numéro de barman et une recette.
- Créer un client pour ce barman et cette recette.

Question 7 Donnez les lignes de la fonction `main` pour :

- Créer et mettre en activité une file d'attente pour un barman.
- Créer le thread du barman dans ce scheduler.
- Créer le thread `eC`.

Exercice 2 : Retour vers les "Futures" (Java ou Ocaml)

On cherche à construire la structure de contrôle concurrente appelée "future". Ce mécanisme permet de lancer un calcul asynchrone, et de se synchroniser lors de la première utilisation de la valeur censée être retournée par ce calcul. Pour être intéressant, ce calcul peut être exécuté sur un thread particulier. On va implanter ce mécanisme en lançant un thread par "future". Si le résultat du calcul est nécessaire avant que le calcul ne soit terminé, alors l'accès à cette valeur est alors bloquant.

Voici une interface simplifiée de "future" en Java et un équivalent OCaml :

Java	OCaml

`public interface Future<V>`
`  extends Runnable {`
`  public void run()`
`  public V call()`
`  public V get();`
`  public boolean isDone();`
`}`
	`type 'a future`
	`val spawn : ('a -> 'b) -> 'a -> 'b future`
	`val get : 'a future -> 'a`
	`val isDone : 'a future -> bool`

- `get` retourne si elle est calculée la valeur de la "future" et sinon attend
- `isDone` retourne un booléen à `true` si le calcul a effectivement eu lieu, et `false` sinon
- `call` ou `spwan`

– `run` : lance le thread de calcul d'une valeur de type `V` en appelant la méthode `call`, la valeur retournée qui pourra être accédée ensuite par `get`
– `spawn` : lance le thread de calcul de la fonction de type `'a -> 'b` sur le paramètre de type `'a` , et construire une `'b future`.

Les questions peuvent être traitées en Java ou en OCaml.

Question 8 Indiquer le langage que vous prenez, et donner les mécanismes de base que vous allez utiliser pour implanter les "futures". Dans un premier temps on considère que le calcul demandé n'est jamais interrompu.

Question 9 Ecrire :
– soit en Java une classe abstraite `MyFuture` qui implante l'interface `Future<V>` en implantant le mécanisme de base des `future`, c'est-à-dire toutes les méthodes sauf `call`,
– soit en OCaml la définition du type `future` et les quatre méthodes du tableau précédent.

Question 10 Détailler le déroulement d'un des deux programmes suivants, où la classe `f`(ou la fonction `f`) calcule le ième nombre de Fibonacci [1], en fonction de votre implantation des "futures".

Java	OCaml

`/* class f extends MyFuture<Integer> { ...} */`
`{ MyFuture<Integer> x1 = new f(5);`
`MyFuture<Integer> x2 = new f(4);`
`MyFuture<Integer> x3 = new f(6);`
`int result =`
`(x1.get()+x2.get()+x3.get())/3;`
`}`
	`(* val f : int -> int *)`
	`let x1 = spawn f 5`
	`and x2 = spawn f 4`
	`and x3 = spawn f 6 ;;`
	`let result =`
	`((get x1)+(get x2)+(get x3))/3;;`

Question 11 On cherche maintenant à traiter les cas où le calcul est interrompu par une exception pendant le calcul.

Java	OCaml

`public interface FutureExc<V>`
`extends Future<V> {`
`public boolean isCancelled();`
`}`
	`val isCancelled : 'a future -> bool`

– `isCancelled` retourne un booléen à `true` si le calcul a été interrompu avant d'avoir retourné un résultat.

Attention cette prise en compte modifie le comportement de `get` qui lors d'un accès redéclenche l'exception (au sens `Runtime Exception` de Java) qui a provoqué l'arrêt du calcul. Si vous modifier la création de "futures" vous pouvez indiquer uniquement les différences par rapport à la question précédente. Implanter ce nouveau comportement.

Question 12 On cherche à définir un itérateur à base de "futures" qui applique un même traitement sur l'ensemble des éléments d'une structure linéaire classique (à la `ArrayList<T>` ou à la `'a array` ou `'a list` et qui retourne la structure linéaire des (futurs) résultats (à la `map`). Chaque calcul intermédiaire est lui-même un "future". Proposer une architecture permettant de lancer de tels calculs puis l'implanter.

Question 13 On cherche à ajouter un future de synchronisation sur l'ensemble des calculs de la structure indiquant que tous les calculs intermédiaires sont finis. Cela permet d'avoir d'une part accès au ième calcul sans avoir à attendre la fin de tous les autres et d'autre part avoir accès à l'ensemble de la structure alors complètement calculée. Indiquer comment implanter ce mécanisme à partir de la question précédente.

1. où $F_0 = 0, F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n > 1$

Exercice 3. PPNU avec Event (OCaml ou C)

Les étudiants de l'UE MI019 PC2R jouent au jeu PPNU (plus petit nombre unique) qui consiste à deviner le plus petit nombre qui ne sera choisi par aucun autre joueur. En pratique, chaque étudiant écrit une fonction (de type `unit -> int` en OCaml) qui rend un nombre (les fonctions peuvent faire des opérations d'entrées/sorties avec le monde extérieur, mais on suppose qu'elles finissent toutes par terminer en temps raisonnable).
Soit `jVec` le vecteur qui contient toutes les fonctions des joueurs. En OCaml, `jVec` est de type `(unit -> int) array`.
Attention à bien gérer la synchronisation de manière à ne pas obtenir un « simple programme séquentiel » !

Question 14 Créer un canal d'événements nommé `cRes` qui vous servira à envoyer les résultats des fonctions des joueurs (donc des paires d'entiers : numéro du joueur $\times$ nombre choisi) à un *thread* jouant le rôle d'arbitre.

Question 15 Donner une définition du *thread* arbitre qui se charge de garder en mémoire le plus petit nombre unique qu'il a reçu ainsi que le numéro du joueur bien sûr. À la fin du jeu, le *thread* arbitre affiche le numéro du gagnant ainsi que le nombre qu'il a choisit, et meurt.

Question 16 Créer un canal d'événements nommé `cFun` qui vous servira à envoyer les fonctions des joueurs aux *thread*s chargés de les appliquer. Chaque fonction sera envoyée avec son numéro.

Question 17 Créer un *thread* qui envoie toutes les fonctions du vecteur `jVec` dans le canal d'événement `cFun`.

Question 18 Créer un lot de `nTh` *threads*, chacun se chargeant, **tant qu'il est nécessaire de le faire**, de choisir une fonction dans le canal d'événements `cFun`, de l'exécuter, et d'envoyer le résultat de la fonction sur le canal `cRes`.

Question 19 Écrire le code qui manque pour lancer le programme et afficher le résultat (une seule fois, et si on veut rejouer, on relance le programme). Si aucun nombre choisi n'est unique, il n'y a aucun gagnant, et on pourra déclarer vainqueur le joueur `-1`.

Question 20 Expliquer (en le moins de mots possible) pourquoi vous n'avez pas eu besoin d'utiliser des *mutex*. (Toutefois, si vous avez vraiment dû en utiliser, expliquer pourquoi ils vous ont été indispensables.)

Question 21 Votre programme est-il robuste aux limitations des ressources du système ? Expliquer pourquoi (en le moins de mots possible). (Selon vos choix d'implantation, la réponse peut être oui comme elle peut être non, ce qui compte ici est donc l'explication.)

Question 22 Bonus : expliquer comment gérer une durée de vie limitée (*timeout*) pour chaque fonction des joueurs, ou bien implantez-le.

Voici les signatures des fonctions que vous pouvez utiliser si vous choisissez C en langage d'implantation.

```
typedef struct channel* channel_t;
typedef struct event* event_t;
channel_t new_channel ();
event_t send (channel_t, void*);
event_t receive (channel_t);
event_t always (void*);
event_t choose2 (event_t, event_t);
event_t choose3
  (event_t, event_t, event_t);
event_t chooseN (event_t*);
void* sync (event_t);
void* select2 (event_t, event_t);
void* select3 (event_t, event_t, event_t);
void* selectN (event_t*);
int poll (event_t, void**);
/* renvoie TRUE si une valeur est disponible,
auquel cas elle est stockée dans le second paramètre.
 */
```