Master d'informatique — *Documents autorisés*

module PC2R (4I507) — *Durée 2 heures*

Université Pierre et Marie Curie — **Année 2015-2016**

Examen réparti du 03 mars 2016

Exercice 1 : Chaîne de montage en Fair Threads

Nous construisons une chaîne de travail avec les acteurs suivants :

- `NB_MAX = 3` robots numérotés de 0 à 2,
- un transporteur `in`,
- un transporteur `out`.

Elle fonctionne de la manière suivante :

- Au départ, ces acteurs rentrent dans la chaîne dans un ordre quelconque (voir la procédure `main()` ci-dessous) et travaillent en continu, sans fin.
- La variable `numero_courant` indique quel acteur peut travailler et les autres de la chaîne se mettent en attente :
  - -1 : valeur du transporteur `in`. C'est le tour du transporteur `in` qui peut travailler. Il met la variable `numero_courant` à 0, quitte la chaîne pendant une durée aléatoire et y revient.
  - i : compris entre 0 et `NB_MAX`, valeur des robots. C'est le tour du robot numéroté i. Il incrémente la variable `numero_courant` de 1, reste dans la chaîne et attend l'instant suivant.
  - `NB_MAX` : valeur du transporteur `out`. C'est le tour du transporteur `out`. Il remet la variable `numero_courant` à -1, quitte la chaîne pendant une durée aléatoire et y revient.

On souhaite que tous les acteurs ne fassent aucune attente active ou inutile. Tous les travaux qui peuvent être faits dans l'instant doivent être faits (voir l'exemple d'exécution ci-dessous).

canevas du programme

```c
#include "fthread.h"
#include "stdio.h"
#include "unistd.h"
#include "traceinstantsf.h"
#include "stdlib.h"
#include "pthread.h"

#define NB_ROBOTS   3

ft_thread_t    ft_trace;
ft_thread_t    robots[NB_ROBOTS], transporteur_in, transporteur_out;
ft_scheduler_t sched;

/********** ?????????? **********/

int numero_courant = -1;

int alea (int max) {
  return (int)(1.0 * rand() / RAND_MAX * max);
}

void robot (void *arg) {
  long n;

  /********** ?????????? **********/
}


void in (void *arg) {

  /********** ?????????? **********/
}
```

```c
void out (void *arg) {

  /********** ?????????? **********/
}

int main (void)
{
  int i;

  sched = ft_scheduler_create ();

  evt_in  = ft_event_create(sched);
  evt_out = ft_event_create(sched);

  /********** ?????????? **********/

  ft_trace     = ft_thread_create(sched, traceinstants, NULL, (void *)50);

  robots[0] = ft_thread_create(sched, robot, NULL, (void *)0);

  transporteur_out = ft_thread_create(sched, out, NULL, NULL);
  transporteur_in  = ft_thread_create(sched, in, NULL, NULL);

  robots[1] = ft_thread_create(sched, robot, NULL, (void *)1);
  robots[2] = ft_thread_create(sched, robot, NULL, (void *)2);

  ft_scheduler_start(sched);

  ft_exit();

  return 0;
}
```

Un extrait d'un exemple d'exécution

```
>>>>>>>>>>> instant 0 :
Robot 0 attend.
Le transporteur out est present.
Le transporteur out attend.
Le transporteur in est present.
Le transporteur in a mis en place une piece.
Le transporteur in s'en va pour 8 micro-secondes.
Robot 1 attend.
Robot 2 attend.
Robot 0 travaille sur la piece.
Robot 0 a fini.
Robot 1 travaille sur la piece.
Robot 1 a fini.
Robot 2 travaille sur la piece.
Robot 2 a fini.
Le transporteur out est present.
Le transporteur out enleve la piece.
Le transporteur out s'en va pour 3 micro-secondes.
>>>>>>>>>>> instant 1 :
Robot 0 attend.
Robot 1 attend.
Robot 2 attend.
>>>>>>>>>>> instant 2 :
Le transporteur out revient.
Le transporteur out est present.
Le transporteur out attend.
Le transporteur in revient.
Le transporteur in est present.
Le transporteur in a mis en place une piece.
Le transporteur in s'en va pour 7 micro-secondes.
Robot 0 travaille sur la piece.
```

```
Robot 0 a fini.
Robot 1 travaille sur la piece.
Robot 1 a fini.
Robot 2 travaille sur la piece.
Robot 2 a fini.
Le transporteur out est present.
Le transporteur out enleve la piece.
Le transporteur out s'en va pour 7 micro-secondes.
>>>>>>>>>>> instant 3 :
Robot 0 attend.
Robot 1 attend.
Robot 2 attend.
>>>>>>>>>>> instant 4 :
Le transporteur in revient.
Le transporteur in est present.
Le transporteur in a mis en place une piece.
Le transporteur in s'en va pour 9 micro-secondes.
Le transporteur out revient.
Le transporteur out est present.
Le transporteur out attend.
Robot 0 travaille sur la piece.
Robot 0 a fini.
Robot 1 travaille sur la piece.
Robot 1 a fini.
Robot 2 travaille sur la piece.
Robot 2 a fini.
Le transporteur out est present.
Le transporteur out enleve la piece.
Le transporteur out s'en va pour 1 micro-secondes.
>>>>>>>>>>> instant 5 :
...
```

1. Compléter la procédure `in()` pour répondre au comportement du transporteur `in`.
2. Compléter la procédure `out()` pour répondre au comportement du transporteur `out`.
3. Compléter la procédure `robot()` pour répondre au comportement des robots.

Exercice 2 : Table d'association partagée (`OCaml` [1])

On cherche à implanter une table d'association clefs (chaînes de caractères) / valeurs (entiers) partagée à l'aide des canaux synchrones d'`OCaml`. Les "clients" et "serveurs" de cet exercice sont des noms de threads et n'ont rien à voir avec les clients-serveurs internet de l'exercice suivant.
On dispose de l'interface suivante décrivant une implantation séquentielle de cette table :

```
type 'a option = Some of 'a | None
type assoc
cr_assoc : unit ->  assoc
setv: assoc * string * int -> assoc
getv: assoc * string -> int option
```

`cr_assoc` permet de créer une table d'association vide, `setv` d'ajouter une association clef / valeur dans la table et `getv` de récupérer une valeur de type option qui contient la valeur `Some(v)` associée à une clef `k` si `k` existe dans la table, et `None` sinon. L'implémentation de cette table n'est pas demandée dans l'exercice.

On veut créer un thread "serveur" `table_p` qui maintient une table de type `assoc` et qui est joignable sur deux canaux publics `s` et `g`, utilisés respectivement pour la mise à jour de la table et la récupération d'une valeur.
Les clients qui partagent la table ont tous un canal personnel (différent), tous connaissent les canaux `s` et `g` du "serveur" et tous les utilisent de la manière suivante :

- pour mettre à jour, ils envoient sur `s` une paire `(k, v)` où `k` et `v` sont une association clef/valeur à inclure dans la table ;
- pour récupérer une valeur, ils envoient sur `g` une paire `(cp, k)` où `cp` est leur canal personnel et `k` une clef de la table. Ils attendent ensuite sur `cp` qu'on leur envoie la **valeur entière** de l'association avec `k`. Si la clef n'existe pas dans la table, ils attendent indéfiniment.

1. Réaliser le schéma d'un système avec plusieurs clients. Puis donner le type de tous les canaux du système.
2. Ecrire un client qui met à jour la clef `"brouette"` à la valeur `28`, puis qui met à jour cette même clef à la valeur `13`.

1. Une syntaxe fonctionnelle approximative est acceptée.

Puis écrire un client qui récupère la valeur de `"brouette"` et met à jour la clef `"saucisse"` avec le double de la valeur de `"brouette"`.

Expliciter les différents comportements possibles lors de l'exécution en parallèle des deux clients.

3. Ecrire la fonction `table_p` exécutée par le thread "serveur". **Attention** : le "serveur" doit se relancer indéfiniment, et il doit traiter la première requête disponible, qu'elle arrive sur `s` ou sur `g`.
4. On souhaite mettre au point un système pour empêcher un client qui interroge la table avec une clef inconnue de rester bloqué. Maintenant chaque client dispose de deux canaux personnels différents `cp` et `cerr` : quand un client interroge la table sur le canal `g`, il envoie un triplet `(cp, cerr, k)`, puis il attend une réception sur un deux canaux `cp` ou `cerr`. Si la clef existe dans la base, il récupère la valeur associée sur `cp`, sinon il reçoit un message de type `unit` sur `cerr`.
   (a) Donner les types des canaux pour cette version de la table.
   (b) Modifier le code du serveur pour qu'il prenne en compte ce système.
   (c) Ecrire un client qui cherche à récupérer, séquentiellement, la valeur de trois clefs différentes. A chaque essai, si l'une d'elle n'est pas dans la table, il termine immédiatement.
5. On reprend le système initial et on le modifie pour pouvoir lui faire calculer des futures. Le serveur maintient une table de type `assoc` associant des **résultats** à des identifiants. Les clients envoient des calculs (c'est-à-dire des fonctions de type `unit -> int`) sur le canal `s`. A la réception d'un calcul, le serveur crée un nouveau thread pour exécuter ce calcul et écrire le résultat dans la table quand le calcul est terminé. Quand un client demande un résultat, s'il n'est pas dans la table, il patiente jusqu'à ce qu'il y soit.
   (a) Décrire les modifications à apporter au système.
   (b) Ecrire un client qui demande un le calcul de 10 puissance 10, qui attend 5s, puis qui récupère le résultat et le multiplie par deux.

Exercice 3 : Serveur de tâches paramétrables

On cherche à implanter les mécanismes de base d'un service de contrôle de tâches distantes partageant un état commun sous la forme d'un serveur qui répond à des requêtes textuelles de programes clients. Les tâches lancées sur le serveur peuvent se dérouler soit en séquentiel (l'une après l'autre), soit en parallèle. Chaque tâche peut être interrompue dans l'un ou l'autre mode. Les effets de la tâche sur l'état commun peuvent être effectifs au fur et à mesure de son avancée, ou uniquement à la fin de la tâche si cette dernière n'a pas été interrompue. Ces différents modes d'exécution sont indiqués lors des requêtes d'un client. Bien sûr ce paramétrage concerne un seul client mais dans les cas de calcul en parallèle, les différentes requêtes et tâches d'un même client s'influencent.
Le serveur est en attente de connexion. Quand une connexion survient, il construit une prise de communication avec le client ayant effectué la connexion tant que celui-ci ne rompera pas la communication soit via une commande du protocole soit en fermant la socket. Dans cet exercice on ne s'intéressera qu'à la partie serveur. Toutes les communications avec le client s'effectuera à travers la socket créée à la connexion du client.

On définit le protocole suivant :

— pour la fermeture de la socket

```
DISCONNECT/user/
(C->S) demande la fermeture de la socket de communication du client "user"
```

— pour le lancement de tâches et la récupération de résultats :

```
SEND/task/
(C->S) demande le lancement de la tâche "task" par le client
STOP/task/
(C->S) demande d'interruption de la tâche "task" par le client
RECEIVE/task/state/result/
(S->C) retourne l'état et le résultat de la tâche
INTERRUPT/task/
(S->C) indique que la tâche "task" a été interrompue
```

pour le paramétrage du lancement des tâches :

```
TASK/mode/
(C->S) indique au serveur le mode d'évaluation des tâches d'un même client :
       "mode" peut valoir SEQ (séquentiel) ou PARA (parallèle)
STATE/mode/
(C->S) indique au serveur le mode d'évaluation de l'état commun pour un client :
       "mode" peut valoir IMM (immédiat) ou LATE (retardé)
```

Dans tout l'exercice vous pouvez reprendre sans les ré-écrire des parties de code du polycopié en les citant. Vous pourrez répondre dans votre langage préféré pour le serveur (OCaml, Java, C). Vous pouvez supposer connues des fonctions de construction et d'extraction des arguments des requêtes et réponses du protocole. Indiquez tout de même leur signature et ce qu'elles font. On supposera connues les fonctions PUTSTATE et GETSTATE qui respectivement modifie et consulte l'état global.

1. **préambule** : Donner l'organisation générale de votre serveur ainsi que les types, fonctions et variables globales dont vous vous servirez par la suite. Cela englobe les "tâches", leur nommage, leur lancement et l'état global. Implanter le serveur qui gèrera uniquement la connexion et la déconnexion de clients.
2. **tâche séquentielle non interruptible** : Dans ce premier codage on cherche juste à traiter séquentiellement une tâche non interruptible dont les modifications d'état sont immédiates. Une fois la tâche terminée, le résultat est envoyé au client.
3. **interruption** : On ajoute maintenant le traitement d'une interruption demandée par le client dans le cadre d'une tâche séquentielle. Pour le moment l'état est changé immédiatement dès qu'une tâche doit le modifier. Si la tâche termine normalement le résultat est envoyé au client mais si la tâche est interrompue avant la fin du calcul le serveur le signale au client en suivant le protocole indiqué..
4. **tâches en parallèle** : On introduit maintenant le calcul des tâches en parallèle. Comme les tâches peuvent à certains moments de leurs calculs consulter et écrire l'état, il est demandé de synchroniser ces accès/modifications pour que seulement une tâche puisse effectuer une telle action à un instant donné. Indiquer le modèle de synchronisation utilisé puis implanter le traitement parallèle des tâches dans votre serveur. Les tâches sont toujours interruptibles et les modifications d'état immédiats. On tiendra compte de la lecture du mode `TASK` comme indiqué dans le protocole.
5. **modification d'état retardée** : On cherche à modifier la manière dont l'état global est modifié. Pour cela seules les tâches terminant correctement leur exécution peuvent le modifier. Une tâche interrompue ne doit pas changer l'état. Il y a plusieurs solutions possibles : enregistrement des modifications pour pouvoir les défaire, modification d'une copie de l'état en cours de calcul, ... Indiquer quelle solution comptez-vous mettre en œuvre puis implanter la. On tiendra compte de la lecture du mode `STATE` comme indiqué dans le protocole.
6. **variation sur la modification d'état** : Dans cette question, on n'effectuera pas la mise à jour retardée de l'état si celui-ci a changé entre le début et la fin du calcul de la tâche. Dans ce cas là, on relance le calcul de la tâche jusqu'au moment où l'état est stable pendant le calcul de la tâche, et cela au maximum 4 fois. Au delà de cette limite on repassera en calcul séquentiel avec l'état verouillé le temps de ce calcul.