Master d'informatique | **module PC2R (4I507)** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2016-2017**

Examen réparti du 16 mars 2017

Exercice 1 : Chaînes de montage en FairThread

Nous proposons un système utilisant les fairthreads composé de
- 10 robots (`ft_thread_t robot[10]`),
- 3 chaînes de montage (`ft_scheduler_t montage[3]`),
- 1 chaîne de repos des robots (`ft_scheduler_t repos`),
- 1 chef (`ft_thread_t chef`),
- et 1 secrétaire (`ft_thread_t secretaire`).

avec les contraintes suivantes :
- dans chaque chaîne de montage, il y a au maximum 2 robots qui travaillent en coopération.
- le nombre de robots dans ces chaînes de montage est géré par un tableau `int nb_robots[3]` initialisé à 0 (`nb_robots[i]` pour `montage[i]`).
- un tableau `int affectation[10]` initialisé à `-1` (libre), indique à chaque instant l'affectation de chaque robot à une chaîne de montage. Par exemple : si `affectation[7]` vaut 2, cela indique que le `robot[7]` est affecté à la chaîne de `montage[2]`.
- le nombre de robots est ici choisi pour éviter toute gestion de pénurie.

Le système fonctionne de la manière suivante :
- Les 10 robots, le chef et le(a) secrétaire sont liés au départ au scheduler `ft_scheduler_t repos`.
- Le `chef` passe vérifier dans chaque chaîne de `montage[i]` si le nombre de robots y est au maximum (c'est-à-dire 2). Si ce n'est pas le cas, il informe le(a) secrétaire pour faire venir dans cette chaîne un robot. Il doit informer une deuxième fois si un deuxième robot est nécessaire.
- Informé par le chef, le(a) secrétaire choisit dans le tableau `affectation[]` un robot i non encore affecté , affecte `affectation[i]` avec le numéro de la chaîne de montage indiqué par le chef et fait le nécessaire pour permettre au robot i de s'installer dans cette chaîne pour travailler.
- Chaque robot i dans la chaîne `repos` vérifie dans `affectation[i]` s'il a une affectation (c'est-à-dire $\neq -1$). Dans ce cas, il doit intégrer la chaîne indiquée pour y travailler.
- Le travail d'un robot i dans une chaîne de `montage[j]` consiste à :
  - incrémenter `nb_robots[j]`,
  - boucler 5 fois en coopérant et en travaillant pendant n secondes (simuler avec un `sleep(n)` où `n = (rand() * 10 / RAND_MAX) + 1`),
  - décrémenter `nb_robots[j]`,
  - retourner vers la chaîne `repos` et remettre `affectation[i]` à -1 pour indiquer qu'il est libre.

A l'aide du canevas donné ci-dessous et en évitant toute boucle d'attente active et/ou inutile :

Question 1. Ajouter les éléments nécessaires pour répondre aux comportements décrits ci-dessus.

Question 2. Compléter la procédure `_robot()`.

Question 3. Compléter la procédure `_chef()`.

Question 4. Compléter la procédure `_secretaire()`.

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

ft_thread_t    robot[10], chef, secretaire;
ft_scheduler_t montage[3], repos;
int            nb_robots[3], affectation[10];

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

void _robot (void *arg) {
  /********** ?????????? **********/
}

void _chef (void *arg) {
  /********** ?????????? **********/
}

void _secretaire (void *arg) {
  /********** ?????????? **********/
}


int main (void)
{
  int i;

  for (i = 0; i < 3; ++i) {
    nb_robots[i] = 0;
    montage[i] = ft_scheduler_create();
    ft_thread_create(montage[i], traceinstants, NULL, (void *)50);
  }

  repos = ft_scheduler_create();
  ft_thread_create(repos, traceinstants, NULL, (void *)50);
  for (i = 0; i < 10; ++i) {
    robot[i] = ft_thread_create(repos, _robot, NULL, (void *)i);
    affectation[i] = -1;
  }
  chef       = ft_thread_create(repos, _chef, NULL, (void *)0);
  secretaire = ft_thread_create(repos, _secretaire, NULL, (void *)0);

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

  for (i = 0; i < 3; ++i) {
    ft_scheduler_start(montage[i]);
  }

  ft_scheduler_start(repos);

  ft_exit();

  return 0;
}
```

Exercice 2 : Implantation de co-routines (C, Java ou OCaml)

Les co-routines sont des structures de contrôle générales dans lesquelles le contrôle du flux d'exécution passe de manière coopérative d'une routine à une autre sans attendre le retour (return) de la première. Bien sûr il sera possible de revenir à cette première routine ultérieurement juste après l'instruction de coopération.

On cherche dans cet exercice à fournir une implantation d'un modèle spécifique de co-routines. Pour cela on se donne les primitives suivantes sous forme de pseudo-instructions :

- `call` qui prend une fonction et un argument et lance son exécution en tant que co-routine ;
- `status` qui prend une co-routine et retourne son état ;
- `yield` qui repasse la main à l'appelant de la co-routine (juste après le `call` ou le `resume`) ;
- `resume` qui donne la main à une co-routine spécifique déjà lancée par `call` juste après son dernier point de coopération (`yield`).

et voici un exemple en pseudo OCaml d'un tel programme :

```
(* fonction des co-routines *)



let p nom   =
  let r = ref 0 in
  for i = 1 to 6 do
    Printf.printf("%s = %d \n",
                          nom, !r);
    incr r ;
    yield ()
  done ;;




(* ... *)
```

```
(* programme principal *)

let main () =
  let d = ref 3 in
  let a = call (p, "A")  in
  let b = call (p, "B")  in
  let c = call (p, "C") in
  for i = 1 to 6 do
    Printf.printf("d = %d\n",!d);
    (match (!d mod 3) with
        0 -> resume(a)
     |  1  -> resume(b)
     |  2  -> resume(c) );
    incr d ;
  done ;;

main();;
```

```
(* trace   *)

A = 0
B = 0
C = 0
d = 3
A = 1
d = 4
B = 1
d = 5
C = 1
d = 6
A = 2
d = 7
B = 2
d = 8
C = 2
```

Question 1. Donner un type et une structure de données pour de telles co-routines, en particulier vous préciserez comment vous passez d'une co-routine à une autre, tout particulièrement comment vous conservez l'ensemble des piles, une par co-routine. Ecrire la fonction `status` sachant que les différents états d'une co-routine sont : `RUN` en cours d'exécution, `SUSPEND` si elle a passé la main à une autre co-routine, `END` lorsqu'elle a fini normalement (fin de l'exécution ou déclenchement d'une exception).

Question 2. Ecrire les fonctions `call`, `yield` et `resume` en détaillant bien les changements d'état des co-routines et comment une co-routine retrouve sa pile d'exécution lors d'un `resume`. L'appel d'un `call` retourne une co-routine qui pourra être rappelée ensuite par un `resume`. L'appel de `resume` sur une co-routine qui n'est pas dans le mode `SUSPEND` ou `END` déclenche une exception. L'appel de `yield` d'une co-routine qui n'est pas en mode `RUN` déclenche une exception.

Question 3. Ecrire un producteur-consommateur simple où les producteurs et consommateurs sont rattachés à une même file d'attente (queue). Le producteur produit quelques items et les stocke dans le magasin puis redonne la main, chaque producteur a une capacité maximale de production indiquée à son lancement (quand celle-ci est atteinte, il termine) et identifie ses produits par un compteur interne ; les consommateurs enlèvent quelques items, les utilisent, et redonnent la main. Ils ont aussi une capacité de consommation indiquée à leur lancement. Le programme crée l'ensemble des producteurs-consommateurs en précisant leurs capacités de production et de consommation, puis attend la fin de tous.

Question 4. Que faut-il modifier dans votre proposition pour qu'un `yield` puisse retourner une valeur (soit comme pour un `return`, soit via `status`, soit par des primitives de communications supplémentaites), et pour que le `resume` puisse lui-aussi communiquer une valeur à la reprise de la co-routine (soit en tant que nouvelle valeur de l'argument, soit via `status`, soit là-aussi avec une primitive de communication) ?

Exercice 3 : Election distribuée en multicast (C, Java ou OCaml)

Dans les cadres des réseaux répartis, il arrive régulièrement que de nombreuses machines veuillent se connecter les unes aux autres (par exemple dans les applications *peer-to-peer*). Dans de tels cas, il n'y a alors pas réellement de "serveur" centralisé, mais plutôt une adresse de *multicast* à laquelle les machines font des requêtes pour pouvoir se connecter ou connaître l'état actuel du réseau décentralisé. Cependant, il arrive régulièrement que de tels réseaux nécessitent l'existence d'un *coordinateur* permettant de gérer de telles requêtes.

Nous allons commencer par implémenter les mécanismes de bases d'un tel réseau, à savoir que lorsqu'une nouvelle machine veut se connecter à un réseau multicast, elle doit

1. Envoyer une requête sur une adresse multicast (UDP).
2. Attendre de recevoir un message du coordinateur qui contiendra l'adresse et port de celui-ci, ainsi qu'un *identifiant réseau* (simple entier aléatoire)
3. Etablir une connexion (TCP) avec le coordinateur du réseau
4. Demander au coordinateur la liste des utilisateurs du réseau (adresses, ports et identifiants)
5. Etablir une connexion (TCP) avec tous les utilisateurs

On voit ainsi que de nombreuses connexions seront faites et que le client qui est alors part du réseau pourra directement discuter avec n'importe quel membre du réseau.

Question 1. Expliciter (graphiquement ou non) les différentes classes et mécanismes nécessaires pour ces interactions au niveau d'un simple client

Question 2. Décrire les classes et fonctions du client d'un tel réseau en spécifiant les différentes connexions nécessaires, éventuels *threads* et variables à maintenir.

Question 3. Implémenter les fonctions correspondantes.

Question 4. Décrire et implémenter les fonctionnalités supplémentaires nécessaires au fonctionnement du coordinateur.

Comme on peut s'en douter par le cheminement précédent, le *coordinateur* n'est rien d'autre qu'un client comme les autres qui a une charge supplémentaire. Ainsi, lorsque le premier client se connecte au réseau (alors vide), un mécanisme de *timeout* lui permet de s'assurer que personne ne répond, et il devient alors automatiquement *coordinateur du réseau.*

Question 5. Décrire et implémenter les modifications nécessaires à la procédure de connexion pour effectuer ce mécanisme.

Il arrive qu'un coordinateur se déconnecte ou plante, auquel cas il faut pouvoir continuer à garder le réseau vivant. Ainsi on comprend qu'il nous faut deux mécanismes supplémentaires.

1. Détecter la disparition du coordinateur actuel
2. Procéder à l'élection d'un nouveau coordinateur

On utilise ainsi l'*algorithme d'élection de Bully* pour décider d'un nouveau coordinateur. Lorsqu'un des ordinateurs présents sur le réseau s'aperçoit que le coordinateur ne répond plus, il initie une élection. Le principe en est le suivant

— Un processus P détecte qu'une élection doit être initiée.
— P envoie un message ELECTION à tous les ordinateurs ayant des *identifiants* plus élevés que lui.
— Si aucun ne répond, P gagne l'élection et devient coordinateur.
— Si un processus avec un numéro plus élevé répond, il prend la main et P a terminé.

Question 6. Ecrire une fonction qui permet de savoir si une machine est active, et expliquer comment le mécanisme de détection de disparition du coordinateur pourra être mis en oeuvre par les différents clients (ainsi que les éventuelles modifications nécessaires)

Question 7. Décrire et implémenter le mécanisme de l'élection de Bully.