Maitrise d'Informatique
Module PC2R

Programmation Concurrente, Réactive et Répartie

Université Pierre et Marie Curie
Année 2002–2003

Examen du 11 septembre 2003

Documents autorisés - Durée 3 heures

Exercice 1 : Fair Thread en Java

Le but de cet exercice est d'implanter le contrôle entre un *scheduler* et des *fair threads* en Java. On ne s'intéresse pas aux événements. Pour cela on se donne les interfaces suivantes :

```
public interface FairSchedulerInterface {
  void start(FairThread thread);
  void stop(FairThread thread);
  void suspend(FairThread thread);
  void resume(FairThread thread);
}
```

```
public interface FairThreadInterface {
  void run(FairScheduler scheduler);
  void start(FairScheduler scheduler);
  void stop(FairScheduler scheduler);
  void suspend(FairScheduler scheduler);
  void resume(FairScheduler scheduler);
  void cooperate();
}
```

On implantera l'ordonnancement du *scheduler* et des *fair thread* en utilisant un jeton unique et des communications `wait` et `notify`. Voici un cadre pour les classes `FairScheduler` et `FairScheduler` :

```
public class FairScheduler extends Thread
        implements FairSchedulerInterface {
 Vector toStart = new Vector(),
   toStop =new Vector(), toSuspend =new Vector(),
   toResume = new Vector();
 Vector actual = new Vector();
 boolean token = true;

 public FairScheduler(){setDaemon(true);start();}

 synchronized void getToken(){ ... }
 synchronized void
   transferTokenTo(FairThread thread){...}
 }
...
}
```

```
public class FairThread extends Thread
  implements FairThreadInterface {
 public FairScheduler scheduler;
 boolean token = false;
 public FairThread(){ super(); }

 synchronized void getToken(){ ... }
 synchronized void waitForToken(){ ... }
 synchronized void transferTokenToScheduler(){ ... }

...
}
```

Une instance de la classe `FairScheduler` possède 5 champs de type `Vector` pour conserver les *fair threads* rattachés à lui selon leurs états : `actual` : ceux qui s'exécutent dans l'instant; `toStart` (`toStop`) : ceux qui démarrent (s'arrêtent) à l'instant suivant. On ne s'intéresse pas encore aux autres champs.

1. Complétez les méthodes suivantes :

   - `getToken` qui prend le jeton et le notifie;
   - `transferTokenTo` qui transfère le jeton du *scheduler* au *fair thread*, le *scheduler* se met en attente du jeton;
   - `waitForToken` qui met le *fair thread* en attente du jeton;
   - `transferToSceduler` qui transfère le jeton du *fair thread* à son *Scheduler*.

   puis écrire la méthode `cooperate` de la classe `FairThread`.

2. Ecrire les méthodes `start` et `stop` de la classe `FairScheduler` qui mettent à jour les champs `toStart` et `toStop`, puis écrire les méthodes `start` et `stop` de la classe `FairThread`.

3. Ecrire la méthode `one_instant` de la classe `FairScheduler` met à jour `actual` en fonction des threads à démarrer (dans `toStart`) et à stopper (dans `toStop`) puis passe le jeton une fois à chaque thread dans `actual`.

4. Ecrire la méthode `run` de la classe `FairScheduler` qui fait passer d'un instant à l'autre dans une boucle sans fin.

5. Ecrire les méthodes `resume` et `suspend` des deux classes : quand le `scheduler` est suspendu alors tous les *fair threads* attachés le sont. En cas de *resume* sur le *scheduler* alors tous les *fair threads* attachés qui peuvent se réveiller le sont.

Exercice 2 : Moniteurs en O'Caml

On s'intéresse ici à implanter en O'CAML un mécanisme de moniteurs à la Java. C'est-à-dire à l'implantation de méthodes *synchronized* dans des déclarations de classes.

1. Définir une classe `monitor` ayant comme champs de données un `mutex` m et une `condition` c; et définir les méthodes `lock` qui verrouille et `unlock` qui libère le `mutex` m, la méthode `wait` qui libère le `mutex` sur la condition ainsi que les méthodes `notify` et `notifyAll` qui indiquent un changement de condition aux autres threads. Le constructeur de cette classe créera les valeurs du mutex et de la condition.

2. On rappelle le code de la classe `shop` :

```
let m = Mutex.create();;
let c = Condition.create();;

class shop n =
object(self)
  val  mutable size =n;
  val  mutable buffer = ([||] : product array)
  val  mutable ip = 0
  val  mutable ic = 0

  initializer buffer<-Array.create 12 (new product "empty")

   method display1 () = ...
   method display2 () = ...

  method put  = function p ->
    Mutex.lock m;
    while (ip-ic+1 > Array.length(buffer)) do Condition.wait c m done;
    buffer.(ip mod size) <- p;
    self#display1();
    ip <- ip+1;
    Condition.signal c;
    Mutex.unlock m

  method get  = function () ->
    Mutex.lock m;
    while(ip == ic) do Condition.wait c m done;
    self#display2();
    let r = buffer.(ic mod size) in
      ic<- ic+1;
      Condition.signal c;
      Mutex.unlock m;
      r
end;;
```

   Reformulez la classe `shop` comme une sous-classe de `monitor` en utilisant uniquement les méthodes héritées pour la synchronisation de `put` et `get`.

3. On cherche à simuler le fonctionnement du mot clé *synchronized* de Java en O'CAML. Indiquez les modifications à apporter en début et en fin de code des méthodes `get` et `put` pour simuler ce comportement. Ecrivez une macro `synchronized`, au sens de `cpp`, qui englobe l'expression de la méthode en ajoutant le code nécessaire.

Excercie 3 : Objets distants migrants en RMI

Les objets distants proposés en RMI sont enregistrés dans des serveurs RMI.

On s'intéresse ici à la migration d'objet d'un serveur à un autre serveur tout en garantissant les invocations vers le serveur d'origine ainsi que vers les serveurs où cet objet est passé.

Exemple :

```
--> S1 enregistre un PointDM p1

--> C1 obtient une référence distante de p1 sur S1

--> C1 invoque la méthode distante rmoveto(2,2)  sur p1

--> l'objet p1 de S1 migre vers S2 suite à une décision de S1,
     S2 reçoit l'objet p1 et l'expose sous le même nom

--> C1 invoque la méthode rmoveto(1,3) sur p1 à travers sa référence distante,
    cette invocation est traitée par S1, qui sachant que l'objet a migré vers S2 invoque
la méthode distante m sur p1 de S2, le résultat obtenu est envoyé à S1 qui le transmet à C1
```

On utilisera dans les questions suivantes l'interface PointRMI suivante :

```
import java.rmi.*;
public interface PointRMI
  extends Remote {

  void moveto (int a, int b)
    throws RemoteException;

  void rmoveto (int dx, int dy)
    throws RemoteException;

  void affiche()
    throws RemoteException;

  double distance()
    throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class PointD extends UnicastRemoteObject
    implements PointRMI {
  int x,y;
  PointD(int a, int b) throws RemoteException {x=a;y=b;}
  PointD() throws RemoteException {x=0;y=0;}

  public void moveto (int a, int b)  throws RemoteException {
    x=a; y=b;}

  public void rmoveto (int dx, int dy)throws RemoteException {
    x = x + dx; y = y + dy;}

  public void affiche() throws RemoteException {
    System.out.println("(" + x + "," + y + ")");}

  public double distance()    throws RemoteException {
     return Math.sqrt(x*x+y*y);}
}
```

1. Pour pouvoir accepter un transfert d'un objet, écrire une interface `Migrate` qui étend `Remote` en définissant la signature d'une méthode `send` qui envoie une instance de `PointD` à une URL (sous forme d'une `String`).
2. Ecrire une classe `Migration` qui implante l'interface `Migrate`; la méthode `send` expose alors l'objet passé à l'URL indiquée.
3. On cherche à écrire une classe `PointDM` sous classe de `PointD`. Une instance de cette classe réagit comme une instance de la classe `PointD` tant qu'elle ne reçoit pas l'appel à sa méthode `migrate`. Une fois qu'un objet à migrer tous les appels de méthode sont renvoyés vers le nouveau serveur. On ne migre qu'une fois d'un serveur. Détailler les fonctionnalités de votre classe et l'implanter.
4. Ecrire les programmes correspondant à l'exemple du début de l'énoncé (2 serveurs S1 et S2 et un client C1). On suppose que le serveur S2 expose un objet `r` instance de `Migration`.
5. Indiquer comment détecter un cycle dans la migration d'un objet et proposer une solution pour éviter cette situation.
6. Indiquer comment intégrer un mécanisme de callback pour les objets migrants. Comment peut-on simplifier l'acheminement de la réponse vers l'appelant dans ce cas? Faire un schéma pour illustrer vos explications.