Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2006-2007**

Examen du 22 janvier 2007

Exercice 1 : Production équitable avec les Fair Threads (C, Java ou O'Caml)

Le but de cet exercice est de simuler une chaîne de production composée de 3 ateliers numérotés 0, 1 et 2 et des robots transporteurs. Chaque atelier `x` comprend

- un panier `A[x]` de pièces à usiner et un panier `B[x]` de pièces déjà usinées (on suppose que chaque panier peut contenir au maximum `MAX_PANIER` pièces),
- des robots qui, en permanence et chacun leur tour, vont chercher une pièce à usiner dans le panier `A[x]`, usine la pièce pendant un temps plus ou moins long et dépose la pièce usinée dans le panier `B[x]`.

Si le panier `A[x]` (resp. `B[x]`) est vide (resp. pleine), le robot qui veut prendre (resp. déposer) une pièce attendra sans bloquer les autres robots.

Chaque robot transporteur peut déplacer au maximum `MAX_TRANSPORT` pièces (avec `MAX_TRANSPORT < MAX_PANIER`). En respectant sa charge maximum et en prenant un temps plus ou moins long pour chaque déplacement, il

- déplace `MAX_TRANSPORT` pièces de l'entrepôt vers le panier `A[0]`, puis
- déplace les pièces du panier `B[0]` vers le panier `A[1]`, puis
- déplace les pièces du panier `B[1]` vers le panier `A[2]`, puis
- déplace les pièces du panier `B[2]` vers l'entrepôt et
- recommence la première étape.

Si un robot transporteur ne peut pas décharger dans un panier les pièces qu'il transporte (on rappelle que chaque panier a la capacité de `MAX_PANIER` pièces), il attendra pour pouvoir les décharger.

On suppose qu'il existe une fonction `aleatoire(n, m)` qui rend une valeur entière aléatoire comprise les entiers n et m. Elle permettra de simuler un temps plus ou moins long.

Le langage d'implantation est libre : C, Java ou O'Caml ; par contre la bibliothèque concurrente est celle des fair threads du langage de votre choix.

1. Ecrire la procédure `robot_atelier()` pour simuler les actions d'un robot travaillant dans un atelier. Elle affichera à la sortie chaque action exécutée avec le numéro du robot, même les attentes.
2. Ecrire la procédure `robot_transporteur()` pour simuler les actions d'un robot transporteur. Elle affichera à la sortie cheque action exécutée avec le numéro du robot, même les attentes.
3. Mettre en place la chaîne de production avec 2 robots dans chaque atelier et 2 robots transporteurs.

Exercice 2 : Gestion d'un réservoir en Esterel

Le but de cet exercice est de simuler un réservoir. En permanence, le remplissage (respectivement l'évacuation de son contenu) est assuré par un certain nombre de robinets et chaque robinet a son propre débit.

Le système fonctionne de la manière suivante :

- On suppose au départ que le réservoir est vide et tous les robinets (remplissage et évacuation) fermés.
- Le réservoir gère son niveau. A l'instant où ce dernier est supérieur (resp. inférieur ou égal) à la valeur `niveau_haut`, le réservoir émet le signal `FERMER_REMPLISSAGE` (resp. `OUVRIR_REMPLISSAGE`).
- Et symétriquement, à l'instant où son niveau est inférieur ou égal (resp. supérieur) à la valeur `niveau_bas`, le réservoir émet le signal `FERMER_EVACUATION` (resp. `OUVRIR_EVACUATION`).
- Le signal `OUVRIR_REMPLISSAGE` (resp. `FERMER_REMPLISSAGE`) indique à tous les robinets de remplissage d'ouvrir (resp. fermer) instantannément.
- Et symétriquement, le signal `OUVRIR_EVACUATION` (resp. `FERMER_EVACUATION`) indique à tous les robinets d'évacuation d'ouvrir (resp. fermer) instantannément.
- Lors du remplissage, chaque robinet émet à chaque instant un signal valué `DEBIT` dont la valeur positive associée correspond à son débit.
- Lors de l'évacuation, chaque robinet émet à chaque instant un signal valué `DEBIT` dont la valeur négative associée correspond à la valeur de son débit.
- Le robinet dont le débit est positif (resp. négatif) est utilisé pour le remplissage (resp. évacuation).
- Les signaux `OUVRIR_REMPLISSAGE` et `FERMER_REMPLISSAGE` sont exclusifs. Idem pour les signaux `FERMER_EVACUATION` et `OUVRIR_EVACUATION`.

1. Ecrire le module `ROBINET` qui
   - contient les signaux `OUVRIR`, `FERMER` et `DEBIT`,
   - contient la variable (ou constante) `debit`, et d'autres variables ou constantes si nécessaire,
   - permet de simuler le comportement d'un robinet (remplissage ou évacuation) indiqué ci-dessus.

   La valeur du débit sera déterminée à la mise en place du système.
2. Ecrire le module `RESERVOIR` qui
   - émet à chaque instant le niveau de son contenu `NIVEAU(...)`,
   - contient les signaux `FERMER_REMPLISSAGE`, `OUVRIR_REMPLISSAGE`, `FERMER_EVACUATION` et `OUVRIR_EVACUATION`,
   - contient les variables (ou constantes) `niveau_haut` et `niveau_bas`, et d'autres variables ou constantes si nécessaire,
   - permet de simuler le comportement d'un réservoir indiqué ci-dessus.

   La valeur des certaines variables sera déterminée à la mise en place du système.
3. Ecrire le module `RESERVOIR_ROBINETS` permettant de mettre en place le système ci-dessus avec
   - un réservoir de `niveau_haut` = 100 et `niveau_bas` = 10,
   - 3 robinets indépendants de remplissage de débit respectif 1, 2 et 3,
   - 1 robinet d'évacuation de débit -8.

   On souhaite ne voir afficher à la sortie que le signal `NIVEAU` avec sa valeur associée et pas du tout le signal valué `DEBIT`.

Exercice 3 : Client-serveur pour la synchronisation de date (O'Caml et Java)

Le but de cet exercice est d'implanter un service de mise à jour des dates, à la manière de `rdate` (RFC 868). Le serveur sera écrit en O'CAML et les clients en Java. Voici le schéma du service de date, `S` est le serveur et `C` un client :

```
S: écoute sur le  port 37.

C: connexion sur le port 37.

S: envoi du temps comme un nombre entier en format texte.

C: réception du temps.

C: fermeture de la connexion.

S: fermeture de la connexion.
```

Le temps de ce protocole est compté en secondes écoulées depuis le premier janvier 1900 (GMT). En O'CAML la fonction `Unix.time` retourne le temps écoulé depuis le 1er janvier 1970 en secondes. En Java on utilisera la classe `java.util.Date` dont le constructeur sans argument initialise l'instance avec la date courante. La méthode `long getTime()` récupère une date et la méthode `void setTime(long)` la modifie. Les entiers java (de type *long*) argument ou résultat de ces méthodes correspondent au nombre de millisecondes écoulées depuis le 1er janvier 1970. On appellera `decal` la constante du nombre de secondes entre 1/1/1900 et 1/1/1970. On utilisera la méthode statique (de la classe `Long`) `public static long valueOf(String s) throws NumberFormatException` pour convertir une chaîne en *long* ;

Dans les différentes questions, vous pouvez supposer connus différents fragments de code issus des polycopiées de cours en indiquant bien lesquels vous utilisez.

1. Ecrire le serveur O'CAML répondant à ce protocole. Le serveur doit pouvoir répondre à plusieurs requêtes simultanées.
2. Ecrire un client Java simple, demandant une date et l'affichant. Il n'est pas demandé d'écrire la fonction de conversion de secondes en date.

On cherche à améliorer ce protocole client-serveur pour contre balancer les effets dus à l'encombrement du réseau. Pour cela le client lance 3 requêtes distinctes simultanément en conservant la date de l'émission. Quand le serveur répond à une requête, le client construit 1 triplet : (date émission, date envoyée du serveur, date de réception). Quand les 3 triplets sont construits, le client détermine alors la différence de date entre lui et le serveur et se met à jour.

3. Proposer une solution pour la synchronisation des réponses du serveur à ces triples requêtes et indiquer comment effectuer la mise à jour de l'heure en tenant compte des 3 triplets.
4. Implanter votre solution et modifier votre client en conséquence.

Exercice 4 : Dictionnaire en RMI (Java)

On cherche à implanter un service de dictionnaire en utilisant le mécanisme d'appel distant RMI de la plate-forme d'exécution Java.

Pour implanter un dictionnaire, on utilisera l'interface `Map<K,V>` et son implantation `TreeMap<K,V>` où le paramètre de type `K` correspond au type de la clé et le paramètre de type `V` au type de la valeur associée à la clé. On utilisera principalement les deux méthodes suivantes de la classe `TreeMap<K,V>` :

- `V get(Object c)` : retourne la valeur associée à la clé `c` ou `null` s'il n'y en a pas ;
- `V put(K c, V v)` : associe la valeur `v` à la clé `c` ; s'il y a déjà une liaison la nouvelle valeur remplace l'ancienne.

On définit alors l'interface distante d'un dictionnaire de la manière suivante :

```
import java.rmi.*;

public interface RDictInt extends Remote   {
    public String get (String o) throws RemoteException;
    public String put (String c, String v) throws RemoteException;
}
```

1. Ecrire une la classe `RDict` implantant cette interface.
2. Ecrire un serveur qui crée deux objets de la classe `RDict` et les expose sous les noms `"dict1"` et `"dict2"`. Ecrire les commandes pour lancer le serveur sur la machine `exam.jussieu.fr` (sur laquelle vous êtes connecté) sur le port 2007.
3. Ecrire un client simple qui recherche le nom `"université"` d'abord dans `"dict1"`, puis si celui-ci n'existe pas recherche dans `"dict2"`. En cas d'échec dans la deuxième recherche le client déclenche l'exception `Not_found`, préalablement définie, sur le client.
4. Les méthodes `get` et `put` de la classe `TreeMap` ne sont pas synchronisées (au sens du mot clé `synchronized`). Donner un exemple où la recherche d'un mot dans un dictionnaire distant ne retourne pas la valeur attendue par rapport à l'état du dictionnaire au début de la recherche. Indiquer ensuite comment modifier votre programme pour éviter ce comportement.
5. On veut maintenant rechercher dans les 2 (potentiellement n) dictionnaires sans blocage en implantant un mécanisme de rappel RMI. Dans l'exemple, l'idée est de lancer la recherche sur les 2 dictionnaires et de pouvoir utiliser le résultat dès qu'une requête a retourné un résultat différent de `null` en faisant attention que la deuxième requête n'efface pas ce résultat.
   (a) Indiquer la synchronisation souhaitée du côté du client et du côté serveur si nécessaire.
   (b) Modifier l'interface, la classe, le serveur puis le client en fonction de la synchronisation indiquée.