**Master d'informatique** | **module PC2R** | **Université Pierre et Marie Curie**
*Documents autorisés* | *Durée 2 heures* | **Année 2013-2014**

# Examen de rattrapage du 20 mai 2014

## Exercice 1 : Serrure numérique en Esterel

On propose dans cette exercice de simuler une serrure numérique utilisant deux clefs de type entier `CLEF1` et `CLEF2` (voir `Exo_esterel.strl.canevas`). Le principe de son fonctionnement est le suivant :

- Le signal DEBUT est émis pour informer que le programme est prêt à recevoir le premier signal `ENTREE`.
- S'il n'y a aucun signal `ENTREE`, rien ne se passe, aucun signal est émis (voir l'exemple test4 ci-dessous).
- On donne un premier signal valué `ENTREE`. Ensuite, on doit donner dans un délai de `DUREE_ESSAI` ticks, un deuxième signal valué `ENTREE` (voir `test1`). A chaque `tick` où ce dernier est absent, le signal valué `TEMPS` est émis indiquant le nombre de ticks qui reste (voir `test2`).
- Si on dépasse ce délai, le signal `TEMPS_DEPASSE` est émis et on recommence à partir du début.
- Une fois ces deux valeurs reçues, elles sont comparées respectivement aux `CLEF1` et `CLEF2`.
- Si elles sont égales, on termine le programme en émettant le signal `OUVERT`.
- Sinon, le signal `ERREUR` est émis et on recommence à partir du début.
- Chaque erreur est comptée et au bout de `MAX_ESSAI` erreur, le signal `BLOQUE` est émis en continu (voir `test3`).

Soit le canevas `Exo_esterel.strl.canevas` suivant :

Exo_esterel.strl.canevas

```
module serrure :

constant MAX_ESSAI = 3 : integer;

constant CLEF1 = 1, CLEF2 = 22,
         DUREE_ESSAI = 4: integer;

input ENTREE : integer;
output DEBUT, BLOQUE, OUVERT,
       TEMPS_DEPASSE, ERREUR;
output TEMPS : integer;

  ...

end module
```

### Question 1

Compléter `Exo_esterel.strl.canevas` pour ces comportements décrits dans les différents tests suivants.

test1

```
$ Exo_esterel
serrure> ;; ENTREE(1); ENTREE(22);;;;
--- Output: DEBUT
--- Output:
--- Output:
--- Output: OUVERT
--- Output:
--- Output:
--- Output:
serrure>
```

test2

```
$ Exo_esterel
serrure> ;;;ENTREE(12345);;;;; ENTREE(1)
             ;;;ENTREE(22);;;
--- Output: DEBUT
--- Output:
--- Output:
--- Output:
--- Output: TEMPS(3)
--- Output: TEMPS(2)
--- Output: TEMPS(1)
--- Output: DEBUT TEMPS_DEPASSE TEMPS(0)
--- Output:
--- Output: TEMPS(3)
--- Output: TEMPS(2)
--- Output: OUVERT
--- Output:
--- Output:
serrure>
```

| test3 | test4 |
|---|---|
| `$ Exo_esterel`<br>`serrure>  ;; ENTREE(22); ENTREE(1);`<br>`ENTREE(12345); ENTREE(123);`<br>`ENTREE(4); ENTREE(5);;`<br>`--- Output: DEBUT`<br>`--- Output:`<br>`--- Output:`<br>`--- Output: DEBUT ERREUR`<br>`--- Output:`<br>`--- Output: DEBUT ERREUR`<br>`--- Output:`<br>`--- Output: BLOQUE`<br>`--- Output: BLOQUE`<br>`serrure>` | `$ Exo_esterel`<br>`serrure> ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;`<br>`;;;;;;;;;;;;;;;;`<br>`--- Output: DEBUT`<br>`--- Output:`<br>`--- Output:`<br>`--- Output:`<br>`...`<br>`--- Output:`<br>`--- Output:`<br>`--- Output:`<br>`--- Output:`<br>`--- Output:`<br>`serrure>` |

## Exercice 2 : Map er reduce threadés

Dans cet exercice on cherche à effectuer un calcul sur une structure linéaire en multi-threads. L'idée est d'améliorer l'efficacité du programme sans perdre en sûreté d'exécution. Le choix du langage est libre mais le côté fonctionnel facilite les choses. En C on pourra utiliser les pointeurs de fonction, en OCaml les valeurs fonctionnelles et en Java les valeurs fonctionnelles introduites à la version 1.8, ou tout autre mécanisme qui convient.

On cherche à paralléliser une version simplifiée d'un `MapReduce`. La première partie correspond à un `Map` prenant en entrée une fonction $f$ de calcul et une structure linéaire $l$, et retourne une nouvelle structure linéaire dont les éléments correspondent à l'appel de $f$ sur chaque élément de $l$. Une fois ces calculs effectués, la fonction `Reduce` peut effecuter un calcul sur l'ensemble du résultat de l'appel de `Map`.

### Question 2
Définir une stucture linéaire accessible par un index (vecteur ou liste indexée) que l'on appelera `vecteur` par la suite.

### Question 3
Ecrire une fonction map_aux qui prend une fonction $f$ de type $\alpha \rightarrow \beta$, un vecteur $src$ de type $\alpha$, un vecteur $dst$ de type $\beta$, une position $pos$ entière et un $pas$ entier qui applique la fonction $f$ sur tous les éléments de $src$ d'indice $pos + k * pas$ et en stockant les résultats dans $dst$ au même indice.

### Question 4
Ecrire une fonction $map$ qui prend une fonction $f$ de type $\alpha \rightarrow \beta$, un vecteur $src$ de type $\alpha$, et $nbt$ un nombre entier de threads et qui retourne un vecteur de type $\beta$ contenant les résultats de l'appel de $f$ sur tous les éléments de $src$ . La fonction $map$ lance les $nbt$ threads qui travaillent chacun indépendamment des autres. La fonction $map$ termine en retournant un vecteur résultat quand tous les threads ont fini. Pour cela il est nécessaire de se synchroniser sur la fin de tous les threads.

### Question 5
On supose définie une fonction `reduce` qui prend une fonction $g$, une racine $a$ et un vecteur $v$ et qui calcule : $g\ \ldots\ (g\ (g\ (g\ a\ v_0)\ v_1)\ v_2)\ \ldots v_{n-1}$. Donner son type dans le langage que vous avez choisi. Montrer ensuite le fragment de programme pour calculer la somme (reduce) des longueurs des éléments d'un vecteur de chaînes de caractères (on supposera connue une fonction $length$ qui retourne la longueur d'une chaîne de caractères).

### Question 6
On cherche maintenant à écrire un `reduce` qui calcule les appels de $g$ au fur et à mesure de l'avancée des calculs de $f$ du `map` threadé. Ecrire une fonction `mapreduce` qui prend une fonction $f$, un vecteur $v$, une fonction $g$, une racine $a$ et retourne un résultat équivalent à $reduce\ g\ a\ (map\ f\ v)$ sans séparer les phases de $map$ et de $reduce$ . On utilisera un thread supplémentaire pour le calcul incrémental du `reduce`. Avant de coder la réponse, préciser la synchronisation employée entre ces différents calculs. Indiquer une exécution possible sur l'exemple de calcul de la question précédente.

## Exercice 3 : Ventes en ligne en client-serveur Java

On cherche à modéliser un système de vente en ligne : des clients se connectent et envoient des requêtes pour des produits, le serveur cherche les produits dans un stock et les expédie aux clients.

Les produits sont des objets Java de classe `Produit`, qui comporte un champ `Modele` (une chaine de caractères indiquant la nature du produit) et un champ `Id` (un entier identifiant le produit).

Le client se connecte au serveur sur le port 2014 et lui envoie des requêtes de `Modele` (par exemple "`ORDER---THEIERE`"). Il récupère ensuite, à travers la socket, les objets commandés.
Le serveur est composé de :

- une structure de données `Stock` qui contient les `Produit` mis en vente (on peut supposer qu'elle contient un champ `getModele(Modele mod)` qui retire de `Stock` un objet dont le champ `Modele` vaut `mod` et le retourne),
- une classe `Commande` utilisée pour représenter, en interne, les requêtes des clients. Elle doit stocker le `Modele` demandé et comporter un moyen d'identifier l'origine de la commande (quel client l'a émise),
- une structure de données (de votre choix, FIFO recommandé) `Carnet` qui stocke les commandes de tous les clients,
- une classe `Expedition` utilisée pour représenter les produits qui vont être envoyés. Elle doit contenir un objet `Produit` et comporter un moyen d'identifier sa destination (vers quel client il doit être envoyé),
- une structure de données (de votre choix) `Envoi` qui stocke les `Expedition` en attente,

et des threads suivants :

- un thread principal `Vente` qui ouvre une socket sur le port 2014, accepte des connexions *simultanées* et crée à chaque nouvelle connexion un thread `Commercial` traitant les requêtes du client,
- des threads `Commercial` qui recoivent les requêtes des clients, les transforment en `Commande`, et les ajoutent au `Carnet` de commandes,
- des threads `Ouvrier` qui lisent `Carnet`, traitent les commandes en cherchant les `Produit` correspondants dans le `Stock` et créent une nouvelle `Expedition` dans `Envoi`,
- un thread `Expediteur` qui lit les `Expedition` dans `Envoi`, et envoie le `Produit` au client.

### Préliminaires

### Question 7

a. Réaliser un croquis annoté succinct du système client-serveur en entier.
b. Si le serveur ne contient qu'un seul `Ouvrier`, quelles sont les structures sujettes à la concurrence ? Comment les protéger ?
c. Même question si le serveur contient plusieurs `Ouvrier`.
d. Quel mécanisme utiliser pour envoyer un `Produit` à travers la socket ?
e. Que doit contenir la classe `Commande` pour que l'objet soit expédié au bon client ?

### Modèle du système - Client

### Question 8

Ecrire le code d'un client qui se connecte au serveur, envoie des requêtes sur trois modèles différents et réceptionne les produits.

### Modèle du système - Serveur

### Question 9

a. Donner la classe `Commande`.
b. Donner l'interface de la classe `Carnet`.
c. Ecrire les threads `Vente` et `Commercial`. Precisez la syntaxe des messages envoyés du client au serveur.

### Question 10

a. Donner la classe `Expedition` et l'interface de `Envoi`.
b. Ecrire le code des threads `Ouvrier`. On supposera que le produit commandé est toujours disponible dans `Stock`.

### Question 11

Ecrire le code du thread `Expediteur`.

### Améliorations

## Question 12
Expliquer quelles parties du code modifier pour gérer l'absence éventuelle d'un produit dans le stock (en le signalant au client).

## Question 13
Ecrire le code d'un thread `DRH` qui gère les threads `Ouvrier`, il regarde périodiquement la taille $t$ du carnet de commandes et s'assure que le nombre de threads `Ouvrier` n'est pas plus grand que $(t/5) + 1$ et pas plus petite que $(t/5)$ en créant ou fermant des threads. Le système doit garantir qu'un `Ouvrier` ne peut être détruit pendant qu'il traite d'une commande.

## Question 14
Expliquer quelles parties du code modifier pour autoriser les clients à vendre des objets (qui sont ajoutés au `Stock`).