Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2011-2012**

Examen réparti du 08 novembre 2011

Exercice 1 (Fair threads) : l'art du sushi sans souci

Afin de prévoir son investissement en vaisselle, maître Robert, patron d'un bar à sushis, cherche à simuler la gestion des assiettes au sein de son commerce.

Dans le bar à sushi, le cycle typique d'une assiette est le suivant :

1. Maître Robert prépare quelques sushis, puis prend une assiette dans la pile d'assiettes propres pour les y disposer. S'il n'y en a plus, il râle.
2. Il pose alors l'assiette sur le petit train circulaire, si toutefois un de ses wagonnets est libre. Sinon, il s'endort, et sera réveillé aux cris d'un client affamé trouvant le train vide.
3. On suppose que le restaurant est très fréquenté, et ce par des personnes aux goûts variés, et donc notre assiette finira rapidement par intéresser quelqu'un. Un client prend donc cette assiette, et libère ainsi une place sur le petit train.
4. Une fois son assiette terminée, le client doit participer un peu en allant déposer son assiette sur la pile des assiettes sales. Il râle si la pile est pleine.
5. Marcel, jeune apprenti de maître Robert, est alors chargé de laver et essuyer notre assiette, puis de la stocker sur la pile d'assiettes propres.

Primitives fournies Les primitives suivantes vous sont données (en langage C, mais vous pouvez les transposer dans le langage de votre choix).

Sur les assiettes :

```
typedef enum { PROPRE, PLEINE, SALE } etat;
assiette nouvelle_assiette ();
etat verifier_etat (assiette);         /* récupère l'état d'une assiette */
void définir_etat (assiette, etat);    /* modifie l'état d'une assiette */
```

Sur les piles :

```
pile nouvelle_pile ();
void empiler (pile, assiette); /* a n'appeler que sur une pile non pleine */
assiette depiler (pile);       /* a n'appeler que sur une pile non vide */
int pile_vide (pile);          /* TRUE si la pile est vide, FALSE sinon */
int pile_pleine (pile);        /* TRUE si la pile est pleine, FALSE sinon */
```

La simulation est paramétrée par les constantes suivantes :

```
TAILLE_PILES     /* taille maximale des piles d'assiettes */
TEMPS_LAVAGE     /* temps de lavage d'une assiette par l'apprenti */
TEMPS_CONFECTION /* temps de confection d'une assiette de sushis */
TEMPS_MANGEAGE   /* temps que met un client à manger une assiette */
APPETIT          /* nombre moyen d'assiettes par client */
TAILLE_TRAIN     /* nombre de places sur le petit train */
FREQUENTATION    /* temps moyen entre deux clients */
```

Question 1 On utilise trois structures globales : `pile_sale` et `pile_propre` pour les piles d'assiettes, et `train` pour le petit train. Choisissez une structure de données pour le petit train et donnez le code de la fonction `main` initialisant de ces trois structures.

Question 2 Sachant qu'on modélisera les clients, le maître et l'apprenti par des threads, donnez alors le nombre de schedulers que comprendra votre modélisation, et le rôle de chacun. Donnez le(s) nom(s) de ce(s) scheduler(s) et la partie de la fonction `main` l(es) initialisant.

Question 3 En utilisant les constantes définies plus haut, donnez le code de la fonction simulant l'apprenti, et la partie de la fonction `main` initialisant le thread associé.

Question 4 On cherche maintenant à simuler le maître. Afin de gérer ses périodes de repos et le réveils aux cris des clients, quel(s) mécanisme(s) pensez-vous employer ?

Question 5 Donnez le code de la fonction simulant le maître, et la partie de la fonction `main` initialisant le thread associé. Lorsque le maître râle, vous incrémenterez un compteur global `ralage`.

Question 6 Donnez le code de la fonction simulant un client. Les râles du client incrémentent le même compteur global.

Question 7 (Bonus) La simulation fonctionne alors comme ceci : on simule l'arrivée de clients pendant 4 heures, puis on donne le nombre moyen de râlages par client afin de se rendre compte de l'efficacité du dimensionnement. Donnez le code de la fonction `main` effectuant une telle simulation. Vous pourrez utiliser un thread intermédiaire pour simuler l'arrivée des clients.

Exercice 2 (Threads POSIX) : challenges

Un défi oppose les intervenants en PC2R à propos de quel langage de programmation est le plus fort. Maitre Shayu et Maitre Thoong décident d'organiser une compétition entre tous les jeunes Jedi pour décider lequel a raison. Pour ce faire ils décident de mettre en place un serveur avec des défis qui seront disponibles au fil de l'eau pour tous les participants. C'est ce serveur qui va nous intéresser.

On commencera par supposer que nous disposerons des types suivants :

- `file` avec les accesseurs classiques `put` `get` `is_empty` et `is_full`
- `challenge` avec comme accesseurs : `get_text` et `get_number` et possédant des membres/champs `mutex` et `tab_thread` un tableau d'identifiants de threads.

On supposera en outre données les fonctions suivantes :

- `get_challenge` qui prendra une file et rendra (une référence/pointeur etc...) sur un challenge
- `send_challenge` qui prendra une socket et enverra le challenge au compétiteur qui écoute sur cette socket.
- `wait_answer` qui prendra une socket et attendra une réponse dessus (appel bloquant).

Question 8 La file de challenge est de longueur bornée (définie par l'implantation), et l'imagination des deux maitres est féconde, tout autant que le talent des jeunes Jedi. Ainsi, il peut arriver que la file de challenges soit ou bien vide ou bien pleine. Décrire les structures de données additionnelles que vous utiliserez le cas échéant et donnez une implantation des fonctions `put_thread` et `get_thread` versions thread-safe de `put` et `get` permettant de bloquer le thread appelant le temps que l'action puisse être réalisée.

Question 9 Pour gérer les différents challengers l'architecte du serveur compte utiliser des threads : un thread par challenger. Ces threads devront :

- Récupérer le challenge
- Envoyer le challenge au compétiteur.
- Attendre une réponse.
- Essayer de prendre un verrou sur le mutex du challenge réussi. (fonction `pthread_mutex_trylock` qui se comporte comme `pthread_mutex_lock` à ceci près qu'elle est non bloquante et renvoie un entier non nul si le mutex est déjà verrouillé)
- Retirer le challenge de la file en cas de réussite du verrou précédent et incrémenter un compteur local.
- Recommencer les opérations précédentes jusqu'à ce que le texte du challenge soit la chaîne `FIN`
- Dans ce cas là, le thread devra appeler la fonction/méthode `fin_challenger` (qui prendra un thread id et le compteur de challenges résolus, voir en dessous).

Donnez une implantation de la fonction/classe implantant la gestion d'un challenger.

Question 10 Pour déterminer qui a gagné, les threads challengers auront à disposition deux variables partagées/globales : `maxchal` pour le nombre de challenges résolus et un tableau `tabthread` pour stocker les threads id des personnes qui ont résolu ce nombre là (`maxchal`) de challenges. La fonction `fin_challenger` devra accéder prudemment à ces variables, et rajouter le thread id qui lui est passé en paramètre si le nombre de challenges résolus est égal au max actuel ou sinon devra mettre sa propre valeur et mettre un tableau à une case contenant son thread id à la place de `tabthread`.

Question 11 Ecrire une fonction `main` déclarant toutes les variables et lançant deux maîtres (fonction/classe/procédure `master` qu'on supposera bien définies ailleurs) et lançant 5 challengers.

Exercice 3 (Event OCaml) : M-Variables tanponnées

Cet exercice est fortement inspiré de l'article *Concurrent Haskell*.

On cherche à construire un tampon pour un producteur-consommateur en utilisant les m-variables (`M-Var`) comme définies ci-dessous.

```
type 'a mvar = MV of ('a Event.channel * 'a Event.channel * bool Event.channel);;

let mVar () =
  let takeCh = Event.new_channel ()  and putCh = Event.new_channel ()
  and ackCh = Event.new_channel () in
  let rec empty () =
    let x = Event.sync (Event.receive putCh) in
      Event.sync (Event.send ackCh true);
      full x
  and full x =
    Event.select
      [Event.wrap (Event.send takeCh x) empty ;
       Event.wrap (Event.receive putCh)
                  (fun _ -> (Event.sync (Event.send  ackCh false); full x))]
  in
    ignore (Thread.create empty ());  MV (takeCh, putCh, ackCh) ;;

let mTakeEvt  ( mv : 'a mvar) = match mv with
   MV (takechannel, _, _ ) -> Event.receive takechannel ;;

let mTake mv = Event.sync (mTakeEvt mv);;

exception Put;;
```

```
let mPut mv x = match mv with
  MV (takechannel, putchannel, ackchannel) ->
     Event.sync (Event.send putchannel x);
     if (Event.sync( Event.receive ackchannel)) then ()
     else raise Put ;;
```

Question 12 Ajouter la fonction `swapMvar` de type `'a mVar -> 'a -> 'a` qui stocke la valeur passée dans la `M-Var` argument et retourne l'ancienne valeur contenue dans celle-ci.

Question 13 On cherche à définir un tampon (buffer) pour des producteurs/consommateurs à base de `M-Var`. Pour cela on construit des `C-Var` (channel variable) contenant deux `M-Var` : l'une pour la communication du producteur vers le consommateur qui contiendra la valeur produite (`data`), et l'autre du consommateur au producteur qui indiquera la prise d'une valeur (`ack`).

```
type 'a state = {ack  : unit mvar; data : 'a mvar};;
type 'a cvar  = Buf of 'a state;;
```

Ecire les fonctions suivantes :

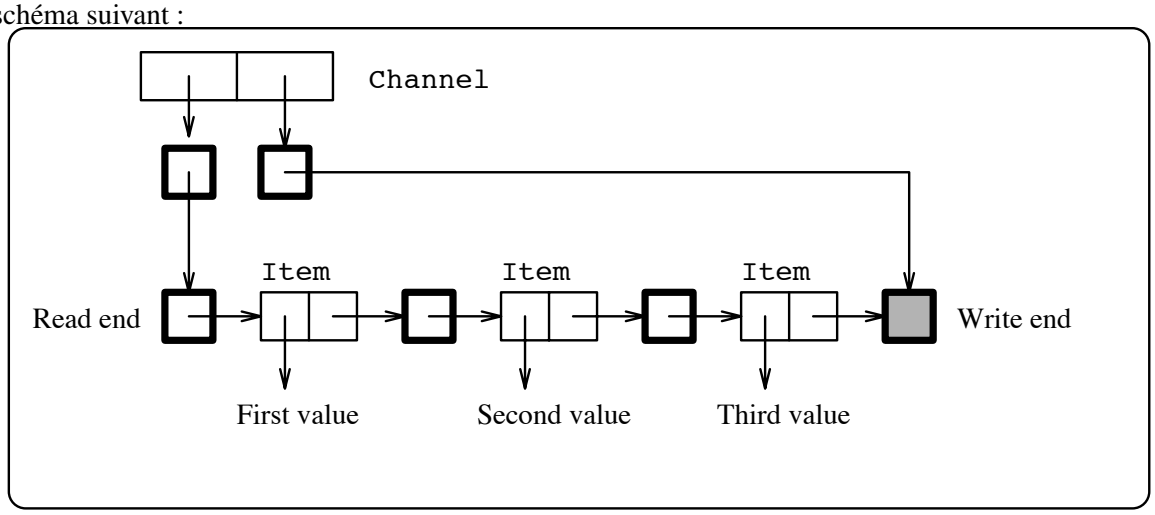
```
val putCvar : 'a cvar -> 'a -> unit
val getCvar : 'a cvar -> 'a
```

Question 14 La difficulté avec les `C-Var` est de n'avoir qu'une seule valeur stockée. On cherche donc à créer des canaux avec tampons. Pour cela on va définir le type `'a bvar` suivant :

```
type 'a bvar  = {read : 'a stream mvar; write : 'a stream mvar}
and 'a stream = 'a item mvar
and 'a item = Item of 'a * 'a stream ;;
```

Une `B-Var` est une paire de `M-Var` (l'une pour les opérations d'écriture et l'autre de lecture). Les données sont contenues dans la structure de type `stream`, elle même une `M-Var` de type `'a item` (structure linéaire).

Le schéma suivant :



représente une telle valeur. Les cases dont le tour est en gras correspondent à des `M-var`.

Ecrire les fonctions suivantes :

```
val bPut : 'a bvar -> 'a -> unit
val bGet : 'a bvar -> 'a
```

qui respectivement ajoute et retire une valeur stockée dans une telle `B-var`.