Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2010-2011**

-

Examen réparti du 09 novembre 2010

Exercice 1 : Fair threads

Cet exercice utilisera les fair threads, libre à vous de choisir le langage qui vous semblera le plus indiqué, on recommande quand même le C pour cet exercice.

Un système d'annuaire (DNS, LDAP, NIS, ...) est un système permettant à des clients d'accéder à des informations. En règle générale, les requêtes effectuées à un annuaire sont traitées dans l'ordre dans lequel elle arrivent.

On souhaite modéliser dans cet exercice un système d'annuaire à deux étages : un étage maître et un étage esclave. On supposera que le maître a la totalité des informations de la base et que l'esclave en possède une partie.

Quand une requête arrive à un esclave, celui-ci la traite. Les requêtes sont traitées dans l'ordre dans lequel elles arrivent. Toutefois, l'esclave peut ne pas avoir l'information demandée. Dans ce cas, il va devoir contacter le serveur maître pour récupérer l'information manquante. On souhaite que la requête soit mise en sommeil en attendant le retour de la demande au maître. Il n'y a aucun besoin que les réponses du maître soient traitées de manière synchrone avec le reste des réponses.

Le traitement d'une requête se fait en appelant la fonction `int treat_request(struct request *)` depuis le fair thread *portant* la requête. Si son code de retour est non nul cela signifie que l'esclave n'a pas l'information désirée. Le thread *portant* la requête doit alors sortir de son scheduler et faire appel à la fonction `void contact_master(struct request *)` elle doit ensuite rejoindre son scheduler, faire appel à la fonction `update_base(const struct request *)`, puis elle devra rendre la main au scheduler pour passer un instant, appeler la fonction `check_request(void *)` et verifier qu'elle rend bien un code de retour nul enfin avant de se terminer le thread devra appeler `void return_answer(const struct request*)`. Si `check_request` rend un code de retour non nul on devra quitter le processus avec un code d'erreur (de retour) égal à 2.

1. En tenant compte des contraintes énoncées ci dessus identifier en termes de fair thread et de scheduler les différents éléments qui interviennent dans le traitement d'une requête à un annuaire.
2. Ecrire la fonction de traitement de requête executée dans un thread portant une requête.
3. En supposant donnée la fonction *bloquante* `new_request_from_network( struct request *)` écrire une fonction gérant l'arrivée des requêtes par le réseau et créant les fair threads portant les requêtes.
4. Ecrire la fonction `main` créant le scheduler et le lançant.

Exercice 2 : threads POSIX

Cet exercice sera à traiter dans le langage qui vous plait le plus parmi C, Java ou O'Caml.

Le quicksort, ou tri rapide, est un benchmark standard de la programation multithread.

On rappelle l'algorithme :

- On sélectionne le premier élément e.
- Si la liste des éléments à trier est réduite à au plus un élément, on sait faire !
- Sinon
  - On crée deux listes, une première liste, $l1$, composée des éléments plus grands que e et une deuxième, $l2$, composée des éléments plus petits que e.

– On trie en faisant des appels récursifs sur chacune des deux listes.
– on rend $l1_{trie} :: e :: l2_{trie}$

1. Isoler des tâches que l'on peut effectuer dans des threads dans cet algorithme. Sachant que l'on gagne peu à effectuer en parallèle des tâches agissant sur des données partagées jugez de l'intérêt des tâches que vous avez trouvées.
2. Ecrire une version du quicksort lançant un thread posix pour trier la liste $l1$ et un autre pour trier la liste $l2$.
3. Rajouter à votre implantation un compteur partagé permettant de compter les threads lancés. On prendra un soin particulier à maintenir la cohérence de ce compteur global.
4. Modifier votre programme pour que dès lors que le nombre de threads lancés dépasse `MAX_NUM_THREAD` un thread qui voudrait lancer deux threads se met en attente sur une condition (que vous créerez). (On oubliera pas de faire signaler les threads sur cette condition quand ils finissent leur exécution.)

Exercice 3 : Ca c'est fort de fruits

Dans cet exercice, nous allons modéliser une chaîne de production et d'approvisionnement de compotes multifruits. La modélisation se fera à l'aide du module `Event` d'OCaml. On pourra utiliser la fonction `Unix.sleep` pour simuler l'écoulement du temps. Si vous préférez le langage C, vous pouvez utiliser l'équivalent du module `Event` en C donné en annexe.

Les compotes produites peuvent être pomme-fraise ou pomme-banane. La chaîne de production comprend donc trois cannaux d'approvisionnement `pommes`, `fraises` et `bananes`. Sur ces canaux, on fait transiter des caisses, avec le type suivant :

```
type fruit = Banane | Pomme | Fraise
type caisse = {
  contenu : fruit ;
  provenance : string ;
}
```

1. Donnez la fonction `lance_producteur` prenant en paramètre un fruit et un lieu, et lançant un thread de production de fruits sur un canal donné. À chaque cycle, le producteur cueille 10 caisses de fruits pendant 1 seconde, puis les envoie sur le canal synchrone (produisant une vitesse approximative de 10 caisses par secondes si l'usine les récupère suffisamment rapidement).
   `lance_producteur : fruit -> string -> caisse channel -> unit`
2. Définissez les trois canaux d'approvisionnement, et lancez 4 sites de production comme dans le tableau ci-dessous.

fruit	site
pomme	France (Corrèze)
pomme	France (Loire)
fraise	Espagne
banane	France (Martinique)

3. Les fruits arrivent alors à l'usine, qui fabrique une caisse de compote à partir de deux caisses de fruits. On appellera `compotes` le canal de production sur lesquelles l'usine dépose les caisses de compotes, modélisées par le type `caisse_compote`.

```
type caisse_compote = {
  fruit_1 : caisse ;
  fruit_2 : caisse
}
```

Définissez le canal de `compotes`, et écrivez le thread modélisant l'usine, qui produit les compotes à partir des fruits arrivant des trois canaux d'approvisionnement. La production doit satisfaire aux exigences d'un système concurrent, justifiez votre réponse en conséquence.

4. En sortie d'usine, les magasins viennent directement acheter des caisses de compote selon la production disponible.

   On suppose que l'usine a 10 clients réguliers. Écrivez la fonction `lance_magasin` prenant un nom de magasin en paramètre, et consommant selon arrivage le rythme déduit à la question précédente, puis lancez les 10 clients. On simulera l'utilisation des caisses de compote par leur affichage.

5. On suppose maintenant qu'il y a deux chaînes d'approvisionnement `pommes_1` et `pommes_2`, une pour chaque producteur. L'usine devra alors s'approvisionner de pommes sur l'un ou l'autre des sites. Donnez la ou les primitive(s) adéquate(s) du module `Event` permettant d'implanter de comportement et réécrivez le code de l'usine en conséquence.

Event.h

```
/* cf. la documentation OCaml des fonctions */
typedef struct channel* channel_t ;
typedef struct event* event_t ;
channel_t new_channel () ;
event_t send (channel_t, void*) ;
event_t receive (channel_t) ;
event_t always (void*) ;
event_t choose2 (event_t, event_t) ;
event_t choose3 (event_t, event_t, event_t) ;
event_t chooseN (event_t*) ;
void* sync (event_t) ;
void* select2 (event_t, event_t) ;
void* select3 (event_t, event_t, event_t) ;
void* selectN (event_t*) ;
int poll (event_t, void**) ;
/* renvoie TRUE si une valeur est disponible,
   auquel cas elle est stockée dans le second paramètre. */
```

Exercice 4 : Un conducteur de train en Esterel

Dans cet exercice, nous voulons simuler le fonctionnement simplifié d'un train dont vous êtes le conducteur.
On a trois niveaux de freinage ou d'accélération : 1 (léger), 2 (moyen) ou 3 (fort).
Disons que

- ACCERELER(1) augmente ou réduit de 1 unité la vitesse à chaque tick pour la maintenir à la vitesse 1.
- ACCERELER(2) augmente ou réduit de 2 unité la vitesse à chaque tick pour la maintenir à la vitesse 5.
- ACCERELER(3) augmente de 2 unité la vitesse à chaque tick pour la maintenir à la vitesse 10 (vitesse maximum du train).
- FREINER(n) diminue la vitesse de n à chaque tick (avec n compris entre 1 et 3).

Il y a un dispositif de sécurité qui permet d'arrêter complètement le train si au bout d'un certain temps, le conducteur ne manifeste pas sa présence en actionnant le dispositif de présence : par exemple, il s'est endormi ou a une malaise, etc, ...

1. Pour ce dispositif, on écrit un module `securite` qui termine son exécution en émettant le signal `FREINER(3)` si dans les 10 tick, il ne reçoit pas le signal `PRESENC`E.

```
module securite :
inputoutput FREINER : integer;
inputoutput PRESENCE;
   ...
end module
```

2. Écrire un module `presence` qui contrôle à chaque tick le changement de la valeur de l'accélération ou du freinage pour émettre un signal `PRESENCE`.

```
module presence :
inputoutput FREINER : integer;
input     ACCELERER : integer;
inputoutput PRESENCE;
   ...
end module
```

3. On suppose que vous êtes le conducteur. Vous pouvez donc accélérer ou freiner le train en envoyant les signaux valués correspondants. Vous pouvez (devez) signaler (dans les 10 tick) votre présence en changeant la valeur de l'accélération ou du freinage ou simplement en envoyant le signal PRESENCE.

   Écrire un module `controle` qui attend vos ordres pour faire bouger le train et en émettant à chaque tick sa vitesse.

```
module controle :
input     ACCELERER : integer;
inputoutput FREINER : integer;
input     PRESENCE;
output     VITESSE : integer;
   ...
end module
```

4. Écrire un module `train` permettant de simuler le fonctionnement d'un train en utilsant les différents modules ci-dessus.