Systèmes Concurrents

1h45, documents autorisés

24 janvier 2012

Les exercices sont indépendants.

1 Questions de cours (3 pt)

1. Qu'est-ce que l'attente active ? Pourquoi est-ce néfaste ? Existe-t-il des cas où l'attente active est préférable à un blocage ?
2. Pourquoi en Java, a-t-on besoin de mettre une boucle `while` autour d'un appel à `wait` ou `await`, alors qu'un `if` est suffisant avec un moniteur de Hoare classique ?
3. Quelle différence y a-t-il entre `synchronized` en Java et `atomically` en mémoire transactionnelle ?

2 Exclusion mutuelle (4 pt)

On propose l'algorithme d'exclusion mutuelle suivant, où **moi** est l'identité propre du processus (chaque processus possède une identité distincte des autres) :

```
tour : global 0..N-1;
occupé : global boolean := false;
répéter
     répéter
          tour ← moi;
     tant que (occupé);
     occupé ← true;
tant que tour ≠ moi;
     section critique
occupé ← false;
```

1. Cet algorithme vérifie-t-il la sûreté de l'exclusion mutuelle, c'est-à-dire, qu'à tout moment, au plus un processus est en cours d'exécution d'une section critique ?
2. Cet algorithme vérifie-t-il la vivacité faible de l'exclusion mutuelle, c'est-à-dire que lorsqu'il y a (au moins) une demande, un des demandeurs finit par obtenir l'accès exclusif ?
3. Cet algorithme vérifie-t-il la vivacité forte de l'exclusion mutuelle, c'est-à-dire que lorsqu'un processus demande, ce processus finit par obtenir l'accès exclusif ?

Note : si vous répondez « oui » à une question, vous devez argumenter (preuve informelle) ; si vous répondez « non », un contre-exemple suffit.

3 Sémaphore (5 pt)

Un centre de rééducation permet à des sportifs de se soigner suite à une blessure. Le centre est utilisé par les joueurs de deux équipes rivales A et B. Un joueur se rend de lui-même au centre de rééducation quand il est blessé. Un joueur invoque les primitives suivantes (vous pouvez ajouter les paramètres que vous souhaitez, par exemple identité du joueur ou de son équipe) :

- `Entrer(...)` pour entrer dans le centre ;
- `Sortir(...)` pour sortir du centre.

Ces primitives sont potentiellement bloquantes car il faut respecter deux contraintes :

- le centre a une capacité maximale de 20 joueurs ;
- s'il y a dans le centre des joueurs des *deux* équipes, il faut garantir que le nombre de joueurs de l'équipe A n'est pas plus que le double du nombre de joueurs B, et inversement (pour éviter les bagarres).

Hypothèses : les joueurs se comportent correctement, en respectant l'enchaînement `Entrer(...);Sortir(...)` : pas de sortie sans entrée ; pas deux entrées consécutives sans sortie intermédiaire pour un joueur donné ; tout joueur entré finira par demander à sortir.

Proposez une solution à base de sémaphores. Votre solution ne devra pas engendrer d'attente superflue, mais n'a pas à être exempte de famine. Si votre solution présente un risque de famine, vous *devez* préciser dans quelle(s) situation(s) cette famine peut avoir lieu.

4 Moniteur (5 pt)

Proposez une solution à base de moniteur au problème précédent en déroulant clairement les sept étapes de la méthodologie de construction d'un moniteur. Analysez de même les situations de famine.

5 Synchronisation à base de messages (3 pt)

Linda fournit un espace partagé avec deux primitives d'accès :

- une opération de dépôt : `out`, qui permet de déposer un tuple de valeurs. Par exemple `out(3,"coucou",18)` dépose le tuple `(3,"coucou",18)` dans l'espace partagé. Cette opération est toujours non bloquante.
- une opération de retrait : `in`. Cette opération prend en paramètre des valeurs ou des variables libres, par exemple `in(3,?x,?y)`. Cette opération bloque jusqu'à ce qu'un tuple adéquat soit trouvé (ici un tuple de trois éléments dont le premier vaut 3), le tuple est alors retiré de l'espace et les variables affectées avec les éléments du tuple.

On souhaite implanter avec Linda une barrière pour N processus. Une barrière est un point d'attente qui est bloquant tant que les N processus ne l'ont pas atteint et qui devient passant quand les N sont là.

Le code d'un processus utilisateur est :

```
processus travail(i) // i ∈ [1,N] = l'identification du processus
   code applicatif
   code de synchronisation pour franchir la barrière
   code applicatif
```

Donnez le code de synchronisation en utilisant les opérations `in` et `out`.