Algorithme de Dekker et algorithme de Peterson - Introduction à la synchronisation

Ensimag 2A

1 Problème à résoudre

Le but de ce TD est d'introduire la programmation concurrente en vous amenant à penser avec plusieurs flots d'exécution simultanés. En particulier, on s'intéressera dans ce premier TD à l'implantation d'une exécution en **exclusion mutuelle** telle qu'ont dû la réaliser les utilisateurs des premières machines parallèles au début des années 60. Ces premières machines parallèles n'avaient pas de dispositifs matériels pour gérer la concurrence, elles étaient juste composées par la juxtaposition de deux unités de calcul séquentiel manipulant la même mémoire partagée.

1.1 Modèle de machine

Dans tout le TD, on considère une machine composée de deux processeurs identiques mais exécutant chacun son propre flot d'instructions. Un flot d'exécution ne peut pas être déplacé d'un processeur à l'autre. Les deux processeurs accèdent directement à la même mémoire physique et il n'y a pas de mémoires caches.

1.2 Nécessité d'isoler

Question 1 *Quel est le code* pseudo-assembleur *de l'instruction* `i++` *?*

Dans le cas général, si l'un des processeurs fait `i++` alors la valeur de la variable est incrémentée de 1. Si chacun des deux processeurs fait `i++`, on s'attend à ce que la variable soit incrémentée de 2.

Question 2 *Quel scénario d'exécution de* `i++` *produit un résultat incohérent ?*

1.3 Exclusion mutuelle

Question 3 *Quelle solution envisager pour éviter ce problème ?*

On nomme une zone de code qui ne doit être exécutée que par un seul processeur à la fois une **section critique** et on parle alors d'une exécution en **exclusion mutuelle**.

Dans la suite de ce TD, nous allons chercher à produire une exécution en exclusion mutuelle pour une section critique comme par exemple `i++`. Pour cela, nous allons implanter les fonctions `entreeSC` et `sortieSC` protégeant l'exécution de la section critique :

```
entreeSC();
... // section critique
sortieSC();
```

1.4 Pas de `while`

Pour bien comprendre ce qu'il se passe à bas niveau, nous utiliserons uniquement la structure `goto` dans la suite de ce TD. Autrement dit, nous n'utiliserons pas de boucle `while`.

Question 4 *Quel est l'équivalent d'une structure* `while` *en* `goto` *(on s'autorise à utiliser* `if`*) ?*

2 Pour commencer, trichons !

Pour cette seule question, parce que c'est triché, nous utiliserons une fonction `scUsed()` tombée du ciel dont le code s'exécute déjà en exclusion mutuelle si plusieurs processeurs l'appellent en même temps. `scUsed()` renvoie `false` la première fois qu'elle est appelée puis `true` pour tous les appels suivants. La fonction `resetScUsed()` permet de réinitialiser l'état interne de la fonction `scUsed()` afin que celle-ci renvoie à nouveau `false` lors du prochain appel.

Question 5 *Écrire* `entreeSC()` *et* `sortieSC()` *en utilisant* `scUsed()` *et* `resetScUsed()`*.*

Question 6 *Quel est le nom de la stratégie utilisée dans* `entreeSC()` *de la question précédente ?*

Nous allons maintenant implanter l'exclusion mutuelle en utilisant différentes stratégies, et sans tricher ! Sans tricher veut dire ici sans utiliser la fonction `scUsed`. Pour chacune des stratégies vous devez suivre le cahier des charges concernant les fonctions `entreeSC` et `sortieSC`. Nous étudierons les avantages, les inconvénients, ainsi que les erreurs de chaque stratégie.

3 Booléen d'occupation

Dans la première stratégie, un processeur qui rentre dans la section critique va modifier une variable booléenne nommée `occ` pour indiquer qu'il est dans la section critique.

Question 7 *Écrire `entreeSC` et `sortieSC` en utilisant cette stratégie.*

Question 8 *Cette stratégie permet-elle d'avoir une exécution en exclusion mutuelle ? Justifier.*

4 Chacun son tour

Dans cette stratégie, un processeur qui veut rentrer dans la section critique doit attendre que ce soit son tour, indiqué par une variable entière `tour`. Une fois la section critique exécutée, il donne la main à l'autre en changeant la valeur de la variable `tour`.

Pour implanter cette stratégie nous avons besoin de différencier les processeurs lors de l'exécution du code. Pour cela, nous utiliserons les fonctions `myID()` et `otherID()` renvoyant respectivement l'identifiant du processeur qui exécute le code et l'identifiant de l'autre processeur.

Question 9 *Écrire `entreeSC` et `sortieSC` en utilisant cette stratégie.*

Question 10 *Cette stratégie permet-elle d'avoir une exécution en exclusion mutuelle ? Existe-t-il d'autres problèmes dans cette stratégie ? Justifier.*

5 Annonce volonté d'entrer

Dans cette stratégie, les processeurs signalent leur souhait d'entrer en section critique. Pour cela, nous utiliserons un tableau de 2 booléens indexé par $myID()$ et $otherID()$. Un processeur souhaitant entrer en section critique attend que l'autre processeur n'ait plus la volonté d'entrer avant d'y entrer vraiment.

Question 11 *Écrire entreeSC et sortieSC.*

Question 12 *Cette stratégie permet-elle d'avoir une exclusion mutuelle ? Existe-t-il d'autres problèmes dans cette stratégie ? Justifier.*

6 Annonce volonté et renonce

Comme pour la stratégie précédente, un processeur annonce sa volonté de rentrer dans la section critique. Cependant, pour cette stratégie, un processeur n'ayant pas réussi à rentrer car l'autre a également fait l'annonce de sa volonté de rentrer annule son annonce et recommence.

Question 13 *Écrire entreeSC et sortieSC.*

Question 14 *Cette stratégie permet-elle d'avoir une exclusion mutuelle ? Existe-t-il d'autres problèmes dans cette stratégie ? Justifier.*

7 Annonce et renonce + tour

Dans les sections suivantes, nous utiliserons à la fois la notion de demande via un tableau de booléens et la notion de tour. Le processeur fait une demande pour entrer en section critique et renonce si l'autre processeur a également fait la demande et qu'il ne s'agit pas de son tour. Sinon, le processeur rentre.

Question 15 *Écrire entreeSC et sortieSC.*

Question 16 *Cette solution permet-elle d'avoir une exclusion mutuelle ? Existe-t-il d'autres problèmes dans cette stratégie ? Justifier.*

8 Dekker 1966

Les deux points importants de la solution proposée par Dekker en 1966 sont :

- si un processeur renonce, il attend son tour avant de redemander ;
- si c'est son tour, le processeur attend que l'autre ait renoncé avant d'entrer.

Question 17 *Écrire entreeSC et sortieSC.*

9 Peterson 1981

En 1981, Peterson a proposé une solution avec un code plus "simple". L'idée est d'utiliser une variable `dernier` contenant l'identifiant du dernier processeur à vouloir entrer dans la section critique. C'est alors à lui d'attendre. Nous nous détachons de la notion de tour.

Question 18 *Écrire entreeSC et sortieSC.*

10 Implantation

Question 19 *D'un point de vue implantation (produire un code C ou C++ qui fait le pseudo code vu plus tôt), remarquez-vous des éléments qui peuvent être problématiques ?*