TD 4 Synchronisations et sémaphores

Dans l'ensemble de ces exercices, les sémaphores sont supposés FIFO.

1 Producteurs/Consommateurs

Le problème est identique à celui déjà vu avec les moniteurs.

Plusieurs processus producteurs produisent des messages qui sont consommés par plusieurs processus consommateurs. Ces messages de taille fixe sont déposés dans un tampon d'échange de taille N. Les messages sont retirés dans l'ordre des dépôts (tampon FIFO).

- Quand le tampon est plein, un processus producteurs déposants sera bloqué jusqu'à ce qu'une place dans le tampon se libère pour lui.
- Quand le tampon est vide, un processus consommateurs sera bloqué jusqu'à ce qu'un message lui soit destiné.

Question 1 *Comment gérer l'exclusion mutuelle avec un sémaphore ? Comment gérer le comptage des messages déposés avec un sémaphore ? Comment gérer l'espace libre dans le tampon avec un sémaphore ? Du coup combien de sémaphores utiliseriez-vous ?*

Soit deux fonctions fournies `insérerTampon(message)` et `enleverTampon(message)` qui déposent et retirent les messages dans du tampon, sans synchronisations, ni gestion de la concurrence.

Question 2 *Proposer le code des fonctions* `inserer(message)` *et* `retirer(message)` *gérant les accès concurrents.*

2 Philosophes

Plusieurs philosophes sont attablés ensemble autour d'assiettes de spaghettis[1]. Il y a une assiette par personne. Mais pour pouvoir manger un philosophe aura besoin d'avoir deux fourchettes. Malheureusement il

[1]Image de Benjamin ESHAM sous Licence Creative Common BY-SA, voir le détail des attributions à `http://commons.wikimedia.org/wiki/File:Dining_philosophers.png`

n'y a qu'une seule fourchette par philosophe.

Un philosophe sera donc une activité exécutant un code de la forme

```
Philosophe(int i)
{
  while(1)
  {
     pense(i);
     prendre_fourchettes(i);
     mange(i);
     poser_fourchettes(i);
  }
}
```

Le but de l'exercice est donc de d'arbitrer les prises de fourchettes (écrire les fonctions `prendre_fourchettes` et `poser_fourchette` pour permettre aux philospohes de manger lorsqu'ils ne pensent pas.

Question 3 *Si il n'y avait qu'une seule fourchette, comment gérer cette fourchette avec un sémaphore ? Pourquoi cela ne fonctionne-t'il pas avec N fourchettes et N philosophes ($N \geq 2$). Quel est le problème ? Dans quel cas arrive-t-il exactement ?*

2.1 Sémaphore privé : prendre exactement deux fourchettes

Pour synchroniser les philosophes entre eux, on peut utiliserer un sémaphore pour chacun des philosophes (sémaphore privé). Les synchronisations sont organisées en fonction de l'état des philosophes

Question 4 *Un philosophe aura plusieurs états logiques, notamment il* `PENSE` *et il* `MANGE`*. Mais il y a, en plus, un autre état. Lequel ? Combien de fourchettes sont utilisées par un philosophe dans chacun de ces états ?*

Question 5 *Indiquez vos structures de données et vos variables partagées permettant de coder la prise et le dépôt de fourchette ?*

Question 6 *En fonction de l'état du philosophe i et de celui de ses voisins, écrire une fonction* `test_mange(i)` *qui débloque le philosophe i et le fait passer dans l'état* `MANGE` *si il le peut. Cette fonction ne s'occupe pas de la concurrence d'accès aux données.*

Question 7 *Comment gérer l'exclusion mutuelle sur les données avec un sémaphore ?*

La difficulté est qu'il faut gérer en même temps l'exclusion mutuelle sur les données et le blocage des philosophes. Il faut donc prendre garde à ne pas bloquer un philosophes pendant qu'il possède l'accès exclusif aux données.

Question 8 *Écrire les fonctions* `prendre_fourchette` *et* `poser_fourchette`*.*

Question 9 *Cette solution a un problème. Lequel ? Dans quel scénario ?*

2.2 Ordre des ressources : introduire un gaucher

Il n'y a pas d'interblocage si les ressources sont prises dans le même ordre par tous les philosophes.

Question 10 *En utilisant un sémaphore pour chaque fourchette et en fixant un ordre sur la prise de ces sémaphores, codez une solution qui ne provoque pas d'interblocage, contrairement à la question 3.*

2.3 Limiter le nombre de prises partielles des ressources

Il n'y a pas non plus d'interblocage si un seul philosophe à la fois peut essayer de prendre les fourchettes. Il suffit pour cela de coder une exclusion mutuelle bloquante autour de la prises des deux fourchettes.

Question 11 *En utilisant un sémaphore pour chaque fourchette et n'autorisant qu'un seul philosophes à la fois à essayer de prendre les fourchettes, codez une solution qui ne provoque pas d'interblocage, contrairement à la question 3.*

En fait, pour ce problème il n'y a pas d'interblocage si la totalité des philosophes n'essaient pas de prendre une fourchette en même temps.

Question 12 *En modifiant légèrement la solution de la question précédente, codez une solution qui ne provoque pas d'interblocage mais qui autorise nettement plus de concurrence.*

3 Lecteurs/Rédacteurs

Le problème est identique à celui déjà vu avec les moniteurs.

Plusieurs processus/processeurs, souhaitent accéder à une ressource (qui est en lecture et écriture). Pour assurer la cohérence :

- Quand la ressource est en lecture, plusieurs processus/procresseur peuvent accéder à la ressource en lecture. Dans ce cas aucune écriture ne peut avoir lieu.
- Quand la ressource est en écriture, un seul processus/processeur peut à la ressource en écriture. Dans ce cas aucune lecture ne peut avoir lieu.

Question 13 *Combien de sémaphores faut-il pour assurer l'exclusion mutuelle sur la base de données ? Que faudra-t-il compter en plus pour assurer la cohérence de la basse de donnée ? Comment assurer la cohérence du compteur ?*

Question 14 *Codez les fonctions* `debut_lire`*,* `debut_ecr`*,* `fin_lire` *et* `fin_ecr`*.*

Question 15 *Quel est le problème de cette solution ? Comment corriger ce problème ?*

Question 16 *Codez une solution qui n'autorise que des accès dans un ordre FIFO global (lecteurs et rédacteurs) pour les demandes d'accès (*`debut_lire` *et* `debut_ecr`*).*

4 Moniteur en sémaphore

Dans cet exercie nous allons voir comment coder un moniteur en utilisant des sémaphores. Ce moniteur donnera la priorité aux processus réveillés (c.a.d. le `signal` est bloquant)

Notre moniteur sera composé de deux fonctions `f()` et `g()`.

Question 17 *Comment mettre en place l'exclusion mutuelle pour les exécutions de* `f()` *et* `g()` *?*

Question 18 *Comment bloquer un processus ? Comment le réveiller ? Que se passe-t-il si il n'y a aucun processus à réveiller ? Que faut-il donc faire lors des blocages et des réveils ?*

Après un réveil il faut éviter que le processus qui réveille et le processus réveillé ne s'exécutent en même temps.

Question 19 *Comment faire en sorte que le processus qui réveille se bloque ? que faut-il faire en plus si le processus réveillé signale lui-aussi un autre processus ?*

Question 20 *Que faut-il donc faire lorsqu'un processus termine une fonction du moniteur afin de gérer correctement les réveils ?*

Question 21 *Réécrire de manière systématique le petit exemple de moniteur suivant :*

```
Moniteur:
cond c;
cond d;

f()
{
  if (test)
     c.attendre;

  d.signaler;
}

g()
{
  if (test)
     d.attendre

  c.signaler;
}
```