Synchronisation - Mutex et Moniteurs

Ensimag 2A

1 Introduction

1.1 Présentation

Lors du TD précédent (Dekker-Petterson), vous avez découvert des propriétés relatives à la synchronisation entre plusieurs processus :

- **Exclusion Mutuelle** : protection de ressources partagées d'un système pour qu'elles ne soient pas accédées en meme temps.
- **Famine** : un processus n'accèdent jamais à la section critique.
- **Interblocage** ou **Dead-lock** : plusieurs processus se bloquent pour l'éternité.

Les prochains TD ont pour but de pratiquer les outils de synchronisations à la disposition des programmeurs.

1.2 Modèle de machine et d'exécution

La machine est composée de une ou plusieurs unités de calculs (cœurs, CPU, cores, processeurs). À un instant t, plusieurs flots d'exécution sont exécutés sur les unités de calcul, un par unité.

Dans ce TD, tous les flots appartiennent à un même espace de mémoire virtuelle (le même processus). Ils sont nommés *Threads* (ou processus légers, fils d'exécution, lightweight process).

1.3 Outils : Mutex, Moniteurs et Sémaphores

Les 3 outils que nous allons utiliser au cours des différentes séances de TD sont :

- **Mutex** : primitive de synchronisation pour obtenir de l'exclusion mutuelle.
- **Moniteur** : classe thread-safe avec queue d'attente et de réveil.
- **Sémaphore** : compteur private et 2 méthodes d'accès en exclusion mutuelle. Système de queue d'attente.

2 Mutex

2.1 Présentation

Objet utilisé pour faire de l'exclusion mutuelle. Cet objet est généralement réalisé en combinant des techniques logicielles et matérielles.

Initialisation :

```
mutex m;
```

Utilisation :

```
m.lock();
  -- Section Critique
m.unlock();
```

Attention aux problèmes vues dans le TD précédent : Deadlock et Famine.

Par ailleurs, l'exécution d'une section critique est généralement coûteuse : manipulation du verrou et restriction du parallélisme.

2.2 Utilisation simple

Considérons une classe CompteEnBanque qui gère l'état d'un compte en banque. On supposera que l'utilisateur n'a pas de limite sur son découvert.

Question 1 *Ecrire la classe compte dans le cas d'un seul thread.*

```
retirer(int n);
ajouter(int n);
```

Question 2 *Sécurisez cette classe pour le cas de n threads. (Utiliser un mutex)*

3 Moniteurs

Un moniteur est un outil de synchronisation. Il est proche d'un objet avec ses attributs (variables) et ses méthodes (fonctions) utilisant ces attributs. Il est composé :

- de variables d'état décrivant le système (les attributs d'un objet).
- d'un mutex pour réaliser l'exclusion mutuelle nécessaire
- de méthodes exécutées en exclusion mutuelle. Elles auront donc toutes la forme :

```
mutex m;
f(...) {
  m.lock();
  -- Travail de la méthode
  m.unlock();
}
```

— Des variables de condition, c'est à dire des files d'attente pour les threads.
  — `cond c0` : Initialisation d'une condition.
  — `c0.wait(m)` : Attente du thread dans la condition, en libérant le mutex `m` au blocage et en le reprenant au réveil.
  — `c0.signal()` : Réveil d'un thread en attente dans la condition.
  — `c0.broadcast()` : Réveil de tous les threads en attente dans la condition.

Il existe 2 sémantiques de réveil pour les threads. Cela peut modifier la façon dont l'on code le moniteur :

— Hoare : Le thread qui appelle signal se bloque, donne immédiatement la main et le mutex au thread réveillé et reprendra la main ultérieurement. Et donc l'opération broadcast n'a pas de sens.
— POSIX (ou MESA, dans la vraie vie) : Un thread qui appelle signal ou broadcast conserve le mutex.

Sauf mention contraire explicite, nous utiliserons la sémantique POSIX.

3.1 La barrière ou le RDV

Question 3 *Écrire un moniteur qui bloque tous les threads tant que N threads ne sont pas arrivés.*

3.2 Producteur-Consommateur

On dispose à nouveau de deux types de threads qui s'échangent des messages par le biais d'un buffer. Les producteurs écrivent des messages dans le buffer alors que les consommateurs les suppriment. Les règles sont les suivantes :

— Un seul thread modifie la queue à la fois.
— Le tampon est de taille bornée.
— Si il n'y a pas de messages dans la queue, les consommateurs se mettent en attente.
— Si la queue est pleine, les producteurs se mettent en attente.

Question 4 *Ecrire un moniteur pour sécuriser un producteur-consommateur.*

```
-- nombre de cases dans le tampon
const int N = 100;
-- buffer d'écriture
Messages tampon[N];
void producteur(Message mess);
void consommateur(Message *mess);
```

Question 5 *Suivant la politique de réveil (Hoare vs POSIX), le moniteur producteur-consommateur est-il FIFO ?*

3.3 Lecteurs-Rédacteurs

On dispose de deux types de threads voulant accéder à une ressource partagée (BDD, fichier...). Les lecteurs veulent accéder à la donnée et les rédacteurs (aussi appelés écrivains) veulent la modifier. On veut optimiser la politique d'accès. les threads doivent respecter ces règles :

— Il ne peut y avoir au plus qu'un seul thread à la fois qui écrit la donnée.
— Aucun thread ne peut lire pendant une écriture.
— Plusieurs lectures sont possibles en meme temps.

On peut représenter le système par le code suivant :

```
lecteur() {
  debut_lire();
  BDD();
  fin_lire();
}
redacteur() {
  debut_redac();
  BDD();
  fin_redac();
}
```

Question 6 *Ecrire un moniteur pour la gestion des threads lecteur-redacteur. On donnera, si besoin, la priorité aux lecteurs. On vous demande donc d'intégrer au moniteur les méthodes suivantes :*

```
debut_lire(); debut_redac();
fin_lire(); fin_redac();
```

Question 7 *Quel problème peut apparaître avec la politique de priorité aux lecteurs ?*

Question 8 *Ecrire un nouveau moniteur qui donne cette fois-ci la priorité aux écrivains. On comptera les lecteurs et les rédacteurs qui souhaitent accéder à la base de donnée. On essayera de faire le moins de signal inutile possible. Y-a-t-il encore des problèmes ?*

3.4 L'allocateur mémoire

Question 9 *Ecrire un allocateur mémoire pour 1 seul thread. Renvoyer false si il n'y a pas assez de mémoire disponible.*

```
int nblibre = N;
bool alloc(int n);
void free(int n);
```

Question 10 *En utilisant un moniteur, écrire un allocateur mémoire pour p threads, avec limitation de mémoire. Un processeur n'obtenant pas la mémoire demandée attend que celle ci soit disponible. On utilisera la politique de Hoare.*

```
int nbLibre = N;
void alloc(int n);
void free(int n);
```

Question 11 *Que se passe-t-il lorsque l'on utilise ce code de la politique de Hoare dans un cas réel ? On pourra expliciter un exemple d'exécution invalide.*

Question 12 *Modifiez votre code d'allocation de la politique de Hoare pour qu'il fonctionne dans un système usuel. On utilisera l'opérateur* `broadcast`.

3.5 Problème des philosophes

Un groupe de N philosophes est attablés autour d'une table ronde et ils vont manger des pâtes. Pour manger des pâtes, chaque philosophe a besoin de deux fourchettes. Malheureusement ils n'ont qu'une fourchette par personne.

Chaque fourchette est donc posée entre deux philosophes. Le but va être de synchroniser les philosophes avec deux fonctions *prendre_fourchettes* et *poser_fourchettes*.

Le philosophe i exécute le code suivant :

```
while(1) {
  penser();
  prendre_fourchettes(i);
  manger();
  poser_fourchettes(i);
}
```

Question 13 *Combien d'états logiques a un philosophe ?*

Question 14 *Ecrire un moniteur pour le problème des philosophes avec une file d'attente par philosophe.*

Question 15 *Ecrire un moniteur pour le problème des philosophes avec une file d'attente globale.*

Question 16 *Quel problème de synchronisation existe-t-il pour les deux solutions ?*

Question 17 *Considérez le code suivant. Pourquoi cette solution ne provoque-t-elle pas d'interblocage dans la sémantique de Hoare ?*

```
mutex m;
bool fourchette_prise[N] = { false, ..., false };
cond file_fourchette[N];

prendre_fourchettes(i) {
  m.lock();
  if (fourchette_prise[i])
    file_fourchette[i].wait(m);
  fourchette_prise[i] = true;

  if (fourchette_prise[(i+1)%N])
    file_fourchette[(i+1)%N].wait(m);
  fourchette_prise[(i+1)%N] = true;
  m.unlock();
}

poser_fourchettes(i)
{
  m.lock();
  fourchette_prise[i] = false;
  fourchette_prise[(i+1)%N] = false;
  file_fourchette[i].signal();
  file_fourchette[(i+1)%N].signal();
  m.unlock();
}
```