Synchronisation - Mutex et Moniteurs

1 Introduction

1.1 Présentation

Lors du TD précédent, nous avons découvert des propriétés relatives à la synchronisation entre plusieurs processus/threads :

- **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.

Nous allons maintenant pratiquer les outils de synchronisation à la disposition des programmeurs.

1.2 Modèle machine/exécution

La machine est composée de plusieurs unités de calcul. 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 mémoire virtuelle (le même processus), ce sont donc des *threads*.

1.3 Outils

Nous utiliserons deux outils de synchronisation :

- **mutex** : exclusion mutuelle ;
- **moniteur** : exclusion mutuelle et la possibilité d'attendre une certaine condition.

2 Mutex

2.1 Présentation

Outil utilisé pour faire de l'exclusion mutuelle et généralement réalisé en combinant des techniques logicielles et matérielles. Utilisation d'un mutex :

```
mutex m;  // initialisation
lock(&m);
// section critique
unlock(&m);
```

Attention, l'exécution d'une section critique est coûteuse à cause de la manipulation du verrou et restreint du parallélisme.

2.2 Échauffement

Considérons la gestion de l'état d'un compte en banque représenté par une variable de type entier partagée entre plusieurs threads. On supposera que l'utilisateur n'a pas de limite sur son découvert.

Question 1 *Écrire les deux fonctions suivantes dans le cas où l'on considère qu'il n'y a qu'un seul thread qui les utilise.*

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

Question 2 *Sécuriser ces fonctions à l'aide d'un mutex pour le cas de n threads.*

3 Moniteurs

3.1 Présentation

Un moniteur fournit à la fois de l'exclusion mutuelle et la possibilité d'attendre une certaine condition et se compose :

- de variables d'état décrivant le système ;
- d'un mutex `m` pour réaliser l'exclusion mutuelle nécessaire ;
- de fonctions exécutées en exclusion mutuelle qui auront donc toutes la forme :

```
mutex m;
f(...) {
  lock(&m);
  // corps de la fonction
  unlock(&m);
}
```

- de variables de condition, c'est à dire des files d'attente pour les threads.

Une variable de condition fournit :

- `cond c` : initialisation d'une condition ;
- `wait(&c, &m)` : attente du thread dans la condition, en libérant le mutex `m` au blocage et en le reprenant au réveil ;
- `signal(&c)` : réveil d'un thread en attente dans la condition ;
- `broadcast(&c)` : réveil de tous les threads en attente dans la condition.

Il existe 2 sémantiques différentes pour les fonctions `signal` et `broadcast` :

— *POSIX* (donc ce que nous utiliserons en TP) : le thread qui appelle `signal` ou `broadcast` conserve le mutex.

— *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.

La sémantique choisie peut modifier la façon dont on code le moniteur. Sauf mention contraire explicite, nous utiliserons la sémantique POSIX.

3.2 La barrière ou le RDV

Question 3 *Écrire un moniteur qui bloque tous les threads tant que N threads ne sont pas arrivés en implémentant la fonction suivante.*

```
void barriere();
```

3.3 Producteurs-Consommateurs

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

— un seul thread modifie le tampon à la fois ;

— le tampon est de taille bornée ;

— si il n'y a pas de messages dans le tampon, les consommateurs se mettent en attente ;

— si le tampon est plein, les producteurs se mettent en attente.

Question 4 *Écrire un moniteur producteur-consommateur respectant les règles ci-dessus en implémentant les fonctions suivantes*

```
int N = 100; // nb cases dans le tampon
struct message tampon[N]; // état
void produire(struct message mess);
void consomme(struct message *mess);
```

Question 5 *Suivant la sémantique de réveil, Hoare ou POSIX, le moniteur producteur-consommateur que vous avez écrit est-il FIFO ? Autrement dit, est-ce que les threads producteurs respectivement consommateurs sont exécutés dans le même ordre que leur ordre d'arrivée dans le moniteur via la fonction produire respectivement consomme ?*

3.4 Lecteurs-Rédacteurs

On dispose à nouveaux de deux types de threads voulant accéder à une ressource partagée, par exemple une base de données ou un fichier. Les lecteurs veulent accéder à la ressource pour la consulter alors que les rédacteurs veulent la modifier. Les threads doivent respecter les règles suivantes :

— il ne peut y avoir au plus qu'un seul thread rédacteur à la fois

— aucun thread lecteur ne peut consulter la ressource pendant qu'un rédacteur la modifie

— plusieurs lectures sont possibles en même temps

On peut représenter le système par le code suivant :

```
// ressource partagee
struct ressource rsrc;
lecteur() {
  debut_lire();
  // opérations lisant rsrc
  fin_lire();
}
redacteur() {
  debut_redac();
  // opérations lisant et modifiant rsrc
  fin_redac();
}
```

Question 6 *Écrire un moniteur pour la gestion des threads lecteurs-redacteurs avec priorité au lecteur en implémentant les fonctions suivantes.*

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

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

Question 8 *Écrire un nouveau moniteur qui donne cette fois-ci la priorité aux rédacteurs. On essayera de faire le moins de signal inutile possible. Y-a-t-il encore des problèmes ?*

3.5 L'allocateur mémoire

Question 9 *Écrire un allocateur mémoire utilisable uniquement dans un context mono-thread en implémentant les fonctions suivantes. Pour se concentrer sur les aspects synchronisation, alloc renvoie simplement false si il n'y a pas assez de mémoire disponible et true sinon.*

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

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

Question 11 *Que se passe-t-il lorsque l'on utilise ce code en considérant non plus la sémantique de Hoare mais POSIX ? On pourra expliciter un exemple d'exécution invalide.*

Question 12 *Modifier le moniteur pour qu'il fonctionne avec la sémantique POSIX. On utilisera l'opérateur* `broadcast`.

3.6 Problème des philosophes

Un groupe de N philosophes se trouve autour d'une table ronde pour manger des pâtes. Pour manger ces 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 *Écrire un moniteur pour le problème des philosophes avec une file d'attente par philosophe.*

Question 15 *Écrire 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 filea_fourchette[N];

prendre_fourchettes(i) {
  lock(&m);
  if (fourchette_prise[i])
    wait(&filea_fourchette[i], &m);
  fourchette_prise[i] = true;
  if (fourchette_prise[(i+1)%N])
    wait(&filea_fourchette[(i+1)%N], &m);
  fourchette_prise[(i+1)%N] = true;
  unlock(&m);
}

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

3.7 WaitGroup

Dans ce schéma de synchronisation, un groupe de threads, les *waiters*, attends la fin des calculs d'un nombre variable N de DONE, souvent faits par d'autres threads, les *workers*. Ce schéma de synchronisation est utilisé en Go, par exemple. Chaque WaitGroup est à usage unique. Il est possible d'augmenter, ou diminuer N, tant qu'il n'est pas tombé à 0. Chaque worker appelle la fonction *done* quand il a fini. cette fonction décrémente N de 1. Chaque waiter attends, bloqué dans la fonction *wait* tant que tous les worker n'ont pas indiqué avoir fini.

Cet exercice utilise une structure contenant le mutex et la variable de condition du WaitGroup, dans une modélisation objet. Les exercices précédents ont utilisé uniquement des variables globales.

```
typedef struct waitgroup {
  int N;
  bool fini;
  mtx_t m;
  cnd_t cg;
} WaitGroup;

init(WaitGroup *);
add(WaitGroup *);
done(WaitGroup *);
wait(WaitGroup *);
```

Question 18 Implanter les fonctions `init()`, `add()`, `done()`, et `wait()`.