

Prolongations sur les threads

Objectifs du TP :

1. Créer des programmes multi-threadés.
2. Manipuler sémaphores, mutex et conditions.

1 Les conditions

Durant les TP précédents, nous avons vu les mutex, ainsi que les sémaphores qui sont une simple généralisation des mutex. Penchons-nous maintenant sur les conditions. Les conditions sont très proches des mutex. Alors que les mutex sont plutôt destinés à créer des sections critiques, les conditions permettent d’attendre via la fonction `pthread_cond_wait` qu’une condition soit réalisée, le thread attendant cette condition étant réveillé par la fonction `pthread_cond_signal` ou par `pthread_cond_broadcast` (s’ils sont plusieurs à attendre). `pthread_cond_wait` prend en argument une condition et un mutex, ce dernier étant libéré durant l’attente, et repris lors du réveil, le tout de façon atomique. N’oubliez pas de consulter les pages de manuel à propos de ces fonctions. Voici un petit exemple de code utilisant les conditions :

```
pthread_mutex_t mutex;
pthread_cond_t condition;
int ready_to_go = 1;
pthread_mutex_init(&mutex);
pthread_cond_init(&condition, NULL);
void MyWaitOnConditionFunction() {
    pthread_mutex_lock(&mutex); /* on commence par prendre le mutex */
    while(ready_to_go == 0) {
        pthread_cond_wait(&condition, &mutex); /*le mutex est libéré */
    }
    ready_to_go = 0;
    pthread_mutex_unlock(&mutex);
}
void SignalThreadUsingCondition() {
    pthread_mutex_lock(&mutex);
    ready_to_go = 1;
    pthread_cond_signal(&condition);
    /* c'est bon, on peut réveiller le thread attendant la condition */
    pthread_mutex_unlock(&mutex);
}
```

Dans cet exemple, quelque soit l’ordre de départ des threads, ils s’exécuteront en section critique, et `MyWaitOnConditionFunction` attendra le signal de `SignalThreadUsingCondition` pour finir.

2 Lecteurs et écrivains

Dans ce problème classique de programmation multi-threads, il y a une seule donnée partagée qui s’appelle *donnee* (!), utilisée par des threads lecteurs et des threads écrivains. Les threads écrivains incrémentent la variable partagée de différentes valeurs. Quand cette variable atteint une valeur maximum, donnée par *valmax*, alors tous les threads s’arrêtent, aussi bien les lecteurs que les écrivains.

L’application doit fonctionner de la façon suivante :

1. le premier lecteur interdit l’accès aux éventuels écrivains suivants, mais pas aux autres lecteurs,

2. un écrivain interdit tout autre accès, que ce soit en lecture ou en écriture,
3. le point 1 peut provoquer la famine pour les écrivains : si un écrivain arrive, il doit attendre la fin du flot des lecteurs avant d'accéder à la ressource. Pour éviter ce dysfonctionnement, qui peut se traduire par la famine pour les écrivains, l'arrivée d'un écrivain doit bloquer tous les lecteurs suivants. Voici le comportement qui sera adopté : lorsqu'un écrivain se présente, il attend la sortie des lecteurs courants, les lecteurs qui le suivent sont bloqués. Lorsque tous ces lecteurs sont sortis, l'écrivain modifie la donnée, et on sert le (ou les) écrivain(s) qui seraient arrivés entre-temps.
4. Lorsque un écrivain a fini une mise à jour, il donne d'abord accès aux écrivains qui attendent, puis, si il y en a, les lecteurs passeront après.
5. Chaque acteur prend un temps aléatoire (quelques millisecondes) pour effectuer une opération, lecture ou écriture, et se repose un peu entre deux opérations successives.

Question 2.1. *Écrire le programme C simulant l'activité des écrivains et des lecteurs. Vous ferez bien attention à respecter toutes les contraintes, et à ce que chaque écriture se passe sans problème. Vous trouverez un canevas du programme ici : `/home/mgallet/enseignement/ASR2/TP6/`. Je vous conseille de commencer par faire fonctionner les threads avant de vous attacher aux conditions et autres sémaphores.*

3 Le problème des philosophes.

Ce problème a été énoncé par Edsger Dijkstra La situation est la suivante :

- cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ;
- chacun des philosophes a devant lui un plat de raviole ;
- à gauche de chaque assiette se trouve une fourchette.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé ;
- être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :

- quand un philosophe a faim, il va se mettre dans l'état « affamé » et attendre que les fourchettes soient libres ;
- pour manger, un philosophe a besoin de deux fourchettes : les deux fourchettes qui entourent sa propre assiette ;
- si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

Question 3.1. *Ecrire un programme permettant de modéliser le comportement des philosophes.*

Question 3.2. *Lister quelques cas problématiques.*

4 Un programme utile : la multiplication de matrices

Dans cet exercice, nous considérons deux matrices carrées A et B de taille n . Le but est d'écrire un programme de multiplication de matrices utilisant plusieurs threads.

Question 4.1. *Quels sont les découpages possibles ? Combien de threads faudrait-il utiliser ?*

Question 4.2. *Programmez cette multiplication de matrices (on supposera que les coefficients sont des double). Évaluez le temps en fonction du nombre de threads. Est-ce que ce nombre dépend de la taille de la matrice ?*