Théorie et pratique de la concurrence – Master 1 Informatique
TP 2 : Programmation concurrente en C (suite)

Les mutexs

Les *mutexs* permettent de réaliser facilement l'exclusion mutuelle. Ils appartiennent à la bibliothèque `Pthreads`. On les appelle également *verrous*. Pour les utiliser, il faut, bien entendu, inclure la bibliothèque :

```
#include <pthread.h>
```

Pour initialiser un mutex, on procède comme suit :

```
pthread_mutex_init (&lock , NULL);
```

Le premier argument correspond au verrou que l'on souhaite initialiser, quand au second, qui est mis à `NULL`, il précise les attributs du verrou.

Pour "prendre un verrou", on fait :

```
pthread_mutex_lock (&lock);
```

Et pour le libérer :

```
pthread_mutex_unlock (&lock);
```

Pour compiler vos programmes, n'oubliez pas de mettre l'option `-lpthread`

Exercices

Exercice 1: *Le dîner des philosophes*

Le problème du diner des philosophes est un problème classique de partage de ressources en programmation concurrente. Ce problème peut être résumé de la façon suivante. Il y a n philosophes qui se trouvent autour d'une table. Chaque philosophe a devant lui une assiette de riz. Directement à gauche de chaque assiette se trouve une baguette pour manger (il y a donc n baguettes). Un philosophe fait deux choses : penser et manger. Pour manger, il a besoin des deux baguettes qui sont à coté de son assiette. Chaque philosophe agit de la façon suivante : il pense, ensuite quand il a envie de manger il prend d'abord la baguette à sa gauche, *ensuite* la baguette à sa droite. Quand il termine de manger, il rend les deux baguettes et il pense, etc.

Le but de cet exercice est de modéliser ce problème du dîner des philosophes par un programme concurrent C. On modélisera les baguettes par des mutex. Chaque philosophe sera implémenté par un thread.

1. Testez votre programme avec 5 philosophes.
2. Essayez d'obtenir un deadlock.
3. Proposez une solution pour résoudre ce problème d'inter-blocage.

Exercice 2: *Encore producteur-consommateur*

Comme dans le Tp précédent, on programmera un producteur et plusieurs consommateurs, de façon cette fois de garantir l'exclusion mutuelle, à l'aide des mutex. A un moment choisi au hasard, un producteur ajoute un entier choisi au hasard à une file (FIFO). Chaque consommateur attend que la file ne soit pas vide, et ensuite il enlève l'entier de la tête de la file et il l'affiche, et il recommence du début.

Observez le comportement de votre programme. Est-ce que la famine peut encore se produire ?

Exercice 3: *Problème du pont à voie unique*

Des voitures arrivant du nord ou du sud doivent passer un pont à voie unique. Les voitures ayant la même direction peuvent passer le pont au même moment, mais les voitures ayant des directions opposées ne le peuvent pas. Chaque voiture sera un thread différent, qui, une fois passée sur le pont, recommence toujours dans la même direction.

1. Programmez le code des processus voiture, ainsi que le code nécessaire à leur synchronisation à l'aide des mutex. Vérifiez la possibilité de famine. Est-ce votre programme peut produire un inter-blocage ?
2. Modifiez votre solution pour permettre au plus 4 voitures d'être sur le pont.
3. On souhaite maintenant garantir qu'il y aura une alternance entre voiture des deux sens (si il y a régulièrement des voitures qui veulent entrer dans les deux directions). Pour cela, développez une solution qui laisse au plus entrer 4 voitures sur le pont, si une voiture qui va dans la direction opposée veut entrer.

Exercice 4: *Le train qui attend*

Trois cars au départ de Marolles, de Villecresnes et de Santeny amènent les étudiants à la gare de Boissy Saint Léger. Là, un RER spécial les attend, qui partira seulement après l'arrivés des trois cars, pour les amener à l'amphi de IF1. Réalisez ce scénario avec des threads, et utilisez des mutex pour la synchronisation.