Théorie et pratique de la concurrence – Master 1 Informatique
TP 4 : Programmation concurrente en C (suite)

Les mutexs

Les *mutexs* sont des sémaphores binaires (ne pouvant prendre que la valeur 0 ou 1) de la bibliothèque `Pthreads`. On les appelle également verrous. Pour les utiliser, il faut inclure la bibliothèque :

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

La déclaration d'un *mutex* se fait de la façon suivante :

```
pthread_mutex_t verrou;
```

Pour initialiser un *mutex*, on procède comme suit :

```
pthread_mutex_init (&verrou , 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" (i.e. faire P sur le verrou), on fait :

```
pthread_mutex_lock (&verrou);
```

Et pour le libérer :

```
pthread_mutex_unlock (&verrou);
```

Par conséquent, une section critique pourra être faite de la façon suivante :

```
pthread_mutex_lock (&verrou);
/*section critique*/
pthread_mutex_unlock(&verrou);
```

Les variables de condition

La bibliothèque `Pthreads` propose également un autre outil de synchronisation entre processus que l'on appelles les variables de condition. Une variable de condition est déclarée comme suit :

```
pthread_cont_t (varcond);
```

Pour l'initialiser, on procède ainsi :

```
pthread_cond_init (&vacond,NULL);
```

Pour utiliser les variables conditionnelles, on dispose de trois fonctions qui doivent être appelées au sein d'une section critique créée grâce à un *mutex*. La première fonction bloque le *thread* appelant et libère le verrou. Lorsque le *thread* bloqué est ensuite réveillé, il reprend la main sur le verrou. Elle s'appelle de la façon suivante :

```
pthread_cond_wait (&varcond,&verrou);
```

Le deuxième argument correspond au `mutex` utilisé pour la section critique. Un code utilisant cette fonction aura donc l'allure suivante :

```
pthread_mutex_lock(&verrou);
pthread_cond_wait (&varcond,&verrou);
pthread_mutex_unlock(&verrou);
```

Pour réveiller un *thread* bloqué sur une variable de condition, un autre thread peut faire :

```
pthread_cond_signal(&varcond);
```

Là aussi il est **recommandé** d'appeler cette fonction au sein d'une section critique protégée par le *mutex* utilisé par le thread en attente (dans l'exemple précédent il s'agit du *mutex* `verrou`). On peut aussi réveiller tous les *threads* en attente grâce à la fonction :

```
pthread_cond_broadcast(&verrou);
```

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

Vous trouverez un exemples de programmes utilisant les fonctionnalités présentées sur la page web des TD/TP :
`http://www.liafa.jussieu.fr/~sangnier/enseignement/concurrence.html`

Exercices

Exercice 1: *Salle de bain unisexe*

Supposez qu'on dispose d'une seule salle de bain qui peut être utilisée par des hommes ou des femmes mais pas au même moment.

1. Programmez en utilisant les sémaphores une solution à ce problème. Il faut permettre un nombre quelconque d'hommes OU de femmes être dans la salle de bain au même moment. La solution doit assurer l'exclusion mutuelle demandée et l'absence d'inter-blocage.
2. Modifiez votre solution pour permettre au plus 4 personnes (du même sexe) en même temps.
3. On souhaite maintenant garantir qu'il y aura une alternance entre hommes et femmes (si il y a régulièrement des hommes et des femmes qui font la queue). Pour cela, développez une solution qui laisse au plus entrer 4 femmes [respectivement 4 hommes] dans la salle de bains, si un homme [resp. une femme est en train d'attendre].

Exercice 2: *Programmation des sémaphores avec les variables de condition*

Dans cet exercice il vous est demandé de programmer vos propres sémaphores. Pour ce faire, vous pouvez créer un type sémaphore de la forme suivante :

```
typedef struct sema
pthread_mutex_t verrou;
pthread_cond_t varcond;
int compteur;
 semaphore;
```

Intuitivement le verrou sert à protéger la valeur du compteur associée au sémaphore et la variable de condition est utilisée pour bloquer les *threads* qui tentent de prendre le sémaphore alors que la valeur du compteur est à 0.

1. Programmez les fonctions `int semaphore_init(semaphore *sem,int val)`, `int semaphore_post(seamphore *sem)` et `int semaphore_wait(semaphore *sem)` de ce sémaphore (qui correspondent aux fonction équivalentes pour les sémaphore de la libraire `Pthreads`).

2. Testez votre version des sémaphores en les utilisant pour le premier exercice du TP précédent.

Exercice 3: *Problème de la liste partagée*

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.

1. Programmez le code des processus voiture, ainsi que le code nécessaire à leur synchronisation.