Série 3 : La synchronisation inter-threads

avec la librairie pthread (mutex, variables conditions et barrières)

La documentation

a- Mutex , le semaphore d'EM de la librairie pthread

Pour créer un nouveau sémaphore d'exclusion mutuelle (mutex) avec la librairie pthread et l'initialiser :

```
pthread_mutex_t  mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

Pour verrouiller le mutex :

```
pthread_mutex_lock (&mutex1);
```

Pour le déverrouiller après la SC (par le même thread qui a verrouillé!)

```
pthread_mutex_unlock (&mutex1);
```

Pour detruire le mutex a la fin du programme:

```
pthread_mutex_destroy(&mutex1) ;
```

b- Les variables condition (équivalant des sémaphores privés)

Examinons la situation suivante : un thread dans un programme donne ne peut exécuter un morceau de code sauf si un événement particulier arrive (disponibilité d'une donnée par exemple, écrite par un autre thread). Au lieu de faire une attente active qui consiste a dormir x temps, puis re-vérifier si la condition est réalisée, on utilise un sémaphore qui ferait bloquer le thread jusqu'à ce qu'un autre thread (celui qui dépose la valeur) le réveille avec un signal sur le même sémaphore. Ce type de sémaphores est souvent appelé sémaphore privé (un seul ferait un P ou wait (celui qui attend la condition) et tout les autres signal V). Dans la librairie Pthread il existe un type variable condition déclaré comme suit.

Déclaration d'une variable condition :

```
pthread_cond_t  cv;
pthread_condattr_t cattr; //les attributs on peut utiliser par defaut
int retour;
```

Initialisation d'une variable condition (operation impossible si la variable est deja en utilisation par un autre thread):

```
/* initialisation avec des attributs par defaut */
retour = pthread_cond_init(&cv, NULL);  //équivalant a :
cv=PTHREAD_COND_INITIALIZER ;
/* initialisation avec des attributs */
ret = pthread_cond_init(&cv, &cattr);
```

Attente d'une variable condition :

```
/* fonction wait sur la condition * /

pthread_cond_wait(&cv, &mutex1)
```

On donne l'adresse du mutex détenue par le même thread pour qu'il le relâche, avant de se bloquer sur la condition puis il le fermera a nouveau (lock) a la sortie de la condition. Cela suppose aussi qu'on doit toujours verrouiller le mutex avant d'appeler wait.

Signaler une condition :

```
pthread_cond_signal(&cv) ;
```

Il convient aussi de toujours verrouiller le mutex avant de signaler une condition, puis de le relâcher toute de suite après. Cela permettra au thread signale d'être le premier a détenir le mutex juste après son réveil et donc il pourra procéder a l'exécution de la section critique.

Pour détruire la variable condition a la fin du programme :

```
pthread_cond_destroy(&cv) ;
```

c- Les barrières de threads : continuer uniquement si tout les threads ont atteint la barrière !

Declaration d'une barriere de threads :

```
pthread_barrier_t br ;
```

Initialiser la barrière avec le nombre de threads devant atteindre la barrière (et qui appelleront tous pthread_barrier_wait)

```
pthread_barrier_init(&br, NULL, 4) ; //ici 4 threads
```

Mise en attente des autres threads : (blocage du thread appelant jusqu'à ce que tout les threads atteignent la barriere) :

```
pthread_barrier_wait(&br) ;
```

Détruire la barrière a la fin du programme :

```
pthread_barrier_destroy(&br) ;
```

Documentation supplémentaire sur les sémaphores de la norme POSIX

(- qui ne font pas partie de pthread)

La librairie pthread ne définit pas de sémaphores généralisé (autres que les types présentés dans la section précedante). En complément, on pourrait cependant utiliser les fonctions fournies par une autre norme POSIX et qui sont définies dans **semaphore.h** pour la synchronisation entre threads et entre processus lies avec des sémaphores.

Les principales fonctions :

- Pour définir un nouveau sémaphore : sem_t *sem_name*;
- Pour initialiser un sémaphore :

```
int sem_init(sem_t *sem, int pshared, unsigned int value);

Exp : sem_init(&sem_name, 0, 10);
```

- Pour attendre un sémaphore : int sem_wait(sem_t *sem);
  - Exp. sem_wait(&sem_name);
  - Si la valeur du sémaphore est <0, le processus appelant va se bloquer, un appel a sem_post va bloquer un des processus bloques dans la fille d'attente du sémaphore.

- Pour incrémenter la valeur d'un sémaphore : int sem_post(sem_t *sem);
  - Exp. sem_post(&sem_name);
  - sem_post décrémente la valeur du sémaphore sem_name et réveil un des processus bloques (ceux qui ont appele sem_wait).

- Pour récupérer la valeur courante du sémaphore et mettre la valeur dans l'endroit pointe par val

  int sem_getvalue(sem_t *sem, int *valp);

  - **Exp :**

```
int value;
sem_getvalue(&sem_name, &value);
printf("La valeur du semaphore est %d\n", value);
```

- Pour détruire un sémaphore : int sem_destroy(sem_t *sem);

Exercice 1 : Le partage de données entre threads (Mutex + variables conditions)

Le code suivant décrit un programme qui simule un magasin de produits (pour simplifier, un seul type de produits disponible en «INITIAL_STOCK» exemplaires) et un ensemble de clients (des threads) qui viennent retirer un nombre variable d'instances du produit (décrémenter la variable stock de la structure **store**). La taille du stock ne peut dépasser les 20 articles, le magasinier (un autre thread) rapprovisionne son stock des que le nombre d'articles restant atteint 0.

Q1- Exécuter ce programme avec des nombres différents de clients et observer l'enchainement des opérations.

Q2- Il arrive parfois que le stock restant soit une valeur négative malgré que le magasinier réapprovisionne le stock quand il atteint 0 . Régler ce problème en utilisant un mutex (pour protéger la variable partagée). Est ce que le problème du stock <0 disparaît ?

Q3- Utiliser une variable condition pour alerter le magasinier des que le stock est fini ou pas assez pour servir un lient, aussi empêcher les clients de retirer des articles quand le stock n'est pas suffisant.

Q4- Que se passera t il si le thread principal appel pthread_cancel sur une partie des threads clients? Essayez. Pour résoudre le problème changer l'attribut **cancelstate** (utiliser *int pthread_setcancelstate (int state, int * etat_pred)* de chaque thread avant d'entrer dans la section critique puis remettez le a la fin (après la libération du mutex).

NB. La documentation pour la fonction pthread_setcancelstate peut etre obtenue en tapant :

man 3 pthread_setcancelstate.