TD - Programmation répartie

Zone critique – barrière

Exercice 1 : notion de zone critique

Une banque utilise des threads qui mettent à jour les avoirs sur les comptes des particuliers. L'opération effectuée est simplement

```
somme = somme + valeur ;
```

`somme` est le montant sur le compte courant, et valeur est la valeur en plus ou en moins correspondant au crédit ou débit sur le compte.

1. Détaillez les opérations effectuées (à l'échelle de l'assembleur mais sans la syntaxe).
2. L'accès concurrent des threads sur la variable `somme` pose-t-elle un problème ?
3. Expliquez avec un exemple.

Exercice 2 : zone critique – mise en œuvre

Vous devez mettre en œuvre des solutions pour éviter les problèmes d'accès dans une zone critique. Plus précisément, vous devez garantir qu'un seul processus/threads exécute la zone critique à un instant donné et qu'il n'est pas interrompu par un autre processus concurrent exécutant la même zone critique jusqu'à sa fin.

Nous utilisons les conventions suivantes :

```
pthread_mutex monMutex ; //déclare un mutex monMutex
pthread_mutex_lock(&monMutex) ; //vérouille le mutex
pthread_mutex_unlock(&monMutex) ; //dévérouille le mutex
```

Il faudra penser à initialiser vos mutex dans vos codes mais pas demandés en TD. Voici le code du thread (qui est lancé plein de fois au travers de plein de threads différents):

```
void calculFactoriel()
{
       terme++; //terme est une variable globale initialisée à 1
       factoriel=factoriel*terme;  //factoriel  est  une  variable  globale
initialisée à 1
}
```

Questions :

1. Le code du thread comporte-t-il une zone critique ? Si oui, quelle est-elle?
2. Complétez le code de manière à protéger la ou les zones critiques avec des sémaphores à exclusion mutuelle ?
3. On souhaite limiter le calcul du factoriel à 100. Complétez le code de manière à ce que le calcul ne s'effectue pas lorsque terme>100.
4. Complétez le code avec des sémaphores posix :
   - `sem_t monSemaphore; //Création`

- `int sem_init(sem_t *sem, int pshared, unsigned int value); //Initialisation`
- `int sem_wait(sem_t *sem); //Décrémente le sémaphore`
- `int sem_post(sem_t *sem); //Incrémente le sémaphore`
- `int sem_destroy(sem_t *sem); //Détruit le sémahore`

Exercice 3 : Barrière

Nous supposons qu'un certain nombre de threads, implémentant la même fonction ici, doivent se synchroniser à un point dans le code.

Le code du thread est le suivant :

```
void codeThread()
{
      instruction1 ;
      instruction2 ;
      //Les threads doivent s'attendre mutuellement ici.
      instruction3 ;
}
```

Nous supposons qu'il y a N threads (N pourra être une variable globale). Donnez le code pour mettre en œuvre cette barrière avec :

1. Des sémaphores. Il faudra utiliser aussi un mutex et une variable globale indiquant le nombre de threads arrivé à la barrière.
2. Des variables de conditions :

    - `pthread_cond_t condition;`
    - `pthread_cond_wait(&condition, &mutexCond);`
    - `pthread_cond_broadcast(&condition);`

Remarque importante : le `pthread_cond_wait()` doit être forcément combiné à l'utilisation d'un mutex. Celui-ci doit être verrouillé au préalable. Le pthread_cond_wait déverouille le mutex et se met en attente du signal. A la réception du signal le mutex est reverrouillé par `pthread_cond_wait()`.