Systèmes d'exploitation

TP2 : Threads et synchronisation

Objectifs du TP

Implémenter des simulations de gestion des threads et leur synchronisation.

Consigne

Lisez bien l'énoncé, il contient beaucoup d'informations et chaque mot est utile!

N'hésitez pas à chercher des informations sur les fonctions dans le cours et sur Internet, un bon informaticien doit savoir chercher des informations en ligne.

Environnement de travail

Le cours étant surtout basé sur des exemples Linux, nous allons travailler dans un environnement Linux. Vous pourrez éditer vos fichiers C avec le programme `Gedit`.

---

1 Création des threads

Sous UNIX, la bibliothèque permettant de gérer les threads est **`<pthread.h>`**. Un exemple de programme créant un thread appelant une fonction est disponible sur Moodle (`TP2.c`).

1- Ecrivez un programme qui crée deux threads avec la primitive **`pthread_create`**. Le premier thread affiche les entiers de 1 à 50. Le second thread affiche les entiers de 51 à 100. Qu'observez-vous?

2- Si ce n'est pas déjà fait, ajoutez la méthode **`pthread_join`** pour vos deux threads à la fin de votre programme. Cette méthode permet d'attendre la fin des threads (sans cela le programme s'arrête avant la fin).

3- Ecrivez à présent un programme qui crée n threads, la valeur n étant passée en paramètre du lancement du programme principal. Chaque thread doit écrire un message à l'écran.

4- Ajouter les éléments suivants :

— Chaque thread affiche en plus son PID

— Chaque thread affiche son numéro d'ordre (le premier thread affiche 0, le deuxième 1, etc.).

Pour le passage de paramètre, vous pouvez vous inspirer de l'exemple *thread-param.c* présent sur Moodle.

Qu'observez-vous à l'exécution de votre programme?

Trouvez un moyen de corriger l'erreur éventuellement observée.

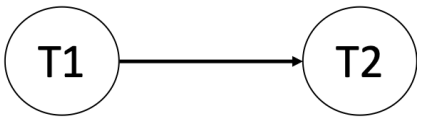
2 Synchronisation des threads et concurrence

5- Ecrivez un nouveau programme créant deux threads. Le premier thread possède une boucle allant de 1 à 1 million et incrémente de 1 une variable globale à chaque itération. Le second thread affiche la valeur de cette variable globale (résultat de l'incrémentation dans la boucle). Qu'observez-vous ?

6- Nous nous trouvons dans un cas où le thread 2 a besoin que le thread 1 soit terminé pour s'exécuter. Autrement dit, nous souhaitons respecter le graph de dépendance ci-dessous. Pour s'en assurer, nous allons utiliser les sémaphores. En C, l'utilisation des semaphores se fait grâce à la bibliothèque **`semaphore.h`** :

— La bibliothèque définit le type semaphore **`sem_t`**.
— La méthode **`sem_init(&sem_t, int shared, int value)`** permet d'initialiser le semaphore à **`value`** (la variable **`shared`** est 0 par défaut).
— La méthode **`sem_wait(&sem_t)`** permet d'utiliser le semaphore (opération **`P()`** vue en cours).
— La méthode **`sem_post(&sem_t)`** permet de relâcher le semaphore (opération **`V()`** vue en cours).

Ajoutez les sémaphores dans votre code pour assurer que le graphe de dépendance soit respecté (le thread T1 doit être terminé avant de lancer T2).



7- Ecrivez un programme avec les éléments suivants :
— Création de *n* threads avec *n* passé en paramètre du programme
— Déclaration d'une variable globale
— Chaque thread doit dans une boucle additionner 1 000 000 de fois son numéro d'ordre à la variable globale. Ex : Pour 5 threads numérotés de 0 à 4, on doit obtenir (0+1+2+3+4) * 1000000 = 10000000
— Le programme principale affiche la valeur de cette variable globale à la fin

Corrigez votre programme si le résultat affiché n'est pas celui souhaité (10000000). Vous pourrez notamment utiliser les mutex : **pthread_mutext_t** avec les méthodes *lock()* et *unlock()*.