

Programmation Concurrente

Mutex et barrière de synchronisation

Exercice 1

Le coeur des 3 threads ci-dessous (appelés en boucle) peuvent avoir grand nombre d'occurrences chacun. Les mutex sont déjà initialisés correctement. Les appels aux ressources X(), Y() et Z() ne sont pas thread-safe.

Coeur du thread A :

```
mutex_lock(&mutex_x);  
use_X();  
mutex_unlock(&mutex_x);
```

Coeur du thread B :

```
mutex_lock(&mutex_y);  
use_Y();  
mutex_unlock(&mutex_y);
```

Coeur du thread C :

```
mutex_lock(&mutex_z);  
use_Z();  
mutex_unlock(&mutex_z);  
  
avec :  
use_X() {  
    X();  
    mutex_lock(&mutex_y);  
    Y();  
    mutex_unlock(&mutex_y);  
}  
  
use_Y() {  
    Y();  
    mutex_lock(&mutex_z);  
    Z();  
    mutex_unlock(&mutex_z);  
}  
  
use_Z() {  
    Z();  
    mutex_lock(&mutex_x);  
    X();  
    mutex_unlock(&mutex_x);  
}
```

- 1) Mais cette solution peut présenter un dead-lock . Démontrez-le de façon explicite.
- 2) Est-ce que la solution proposée ci-dessus fonctionnerait si seulement les version A et B des threads étaient utilisés (à plusieurs exemplaires) ?
- 3) Corrigez la solution complète , **en conservant des sections critiques les plus courtes possibles.**

Exercice 2

On aimerait implémenter notre propre version de barrière de synchronisation pour N threads à l'aide de mutexes **sécurisés**. Votre implémentation de barrière devra avoir un comportement similaire aux barrières POSIX vu en cours, à l'exception de la réinitialisation de la barrière une fois tous les threads passés (cela n'est pas demandé). Chaque thread doit signaler son arrivée à la barrière à l'aide d'un appel à une fonction dédiée, tout comme pour l'implémentation dans la librairie Pthreads.

On se propose d'implémenter l'interface ci-dessous :

- Un nouveau type barrière :
`barrier_t`
- Une fonction d'initialisation d'un objet barrière :
`void barrier_init(barrier_t *b, int count)`
- Une fonction d'attente à la barrière :
`void barrier_wait(barrier_t *b)`
- Une fonction de destruction de la barrière :
`void barrier_destroy(barrier_t *b)`

Réalisez un programme de test permettant de montrer que votre implémentation fonctionne correctement.

- 1) Que pouvez-vous conclure de votre implémentation au niveau de l'utilisation processeur ?

Exercice 3

a) On désire réaliser une petite librairie implémentant une pile pour des entiers. La taille de la pile sera statique et déterminée au moment de sa création.

Afin de garantir un comportement cohérent et déterministe dans le cas d'une exécution multi-threadée, la pile devra être thread-safe. Vous utiliserez les primitives d'exclusion mutuelle appropriées pour cela.

L'interface des fonctions à implémenter se présente comme suit :

- Crée une pile de taille déterminée et renvoie un booléen indiquant si la création a réussi :
`bool stack_create(stack_t *s, int max_size);`
- Détruit une pile :
`void stack_destroy(stack_t *s);`
- Empile une valeur :
`void stack_push(stack_t *s, int val);`
- Dépile une valeur :
`int stack_pop(stack_t *s);`
- Teste si la pile est vide :
`bool stack_is_empty(stack_t *s);`

A vous de décider ce que contiendra la structure `stack_t` définissant un objet de type pile. Un programme de test de la stack vous est fourni (*Ex2_etu.c*): il utilise plusieurs threads `test_stack()` qui font des push et des pop (sans se soucier du contenu récupéré lors des pop). Utilisez-le pour tester votre version de la stack.

b) On aimerait maintenant s'assurer que toutes les tâches `test_stack()` démarrent en même temps. Ajoutez le mécanisme nécessaire pour cela.

Important

Pensez à insérer des assertions (voir « `man assert` ») dans le code aux endroits nécessaires.