Programmation Concurrente

Mutex et barrière de synchronisation

Exercice 1

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.

Question

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

Exercice 2

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 à 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 : il utilise plusieurs threads 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.

Important

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

Exercice 3

On aimerait effectuer le calcul suivant:

$r(k)=\sum_{i=0}^{M-1} A\,x(k,i)$ où $x(k,i)=\sin(k+i)$ si $\sin(k+i)>0$ ou

$x(k,i)=0$ dans les autres cas

A=100

r(k) représente un résultat du calcul (soit une itération k avec $k\in 0,1,2,\ldots,I-1$).

Commencez par réaliser une version séquentielle du programme. Elle permettra entre autre de valider les futurs résultats de la version multi-threadée.

Ensuite, implémentez une version multi-threadée dont le but est de diminuer le temps d'exécution du programme. Pour cela, vous pourrez utiliser un nombre de threads qui va de 2 à N pour calculer le résultat de chaque itération r(k), où 1<N<M, M étant la longueur du vecteur x(k).

2 est la valeur minimum pour N, car pour chaque itération k, il faudra au minimum un thread pour calculer la somme (phase B) et un autre pour calculer les arguments $A\,x(k,i)$ (phase A). Toutefois, des threads supplémentaires peuvent être créés pour participer au calcul de ces arguments.

Dans le but d'accélérer le calcul, utilisez un double vecteur : ainsi l'un peut servir à calculer la phase A du calcul pendant que l'autre et «libéré» pour calculer la phase B.

Il faudra bien sûr répartir la charge de calcul au mieux parmi les threads calculant la phase A et être capable d'enchaîner plusieurs itération k du calcul.

Le programme multi-threadé prendra les 3 arguments suivants :

<M> <N> <I>

Vous êtes libres d'utiliser les primitives de synchronisation ou d'exclusion mutuelle de votre choix, mais l'attente doit être passive.

Enfin, affichez le résultat r(k) obtenu après chaque itération, ainsi que le temps obtenu après

l'exécution complète du programme.

Exécutez votre programme avec différents nombre de threads et avec M assez grand (par exemple 3'000'000).

Vérifiez avec les mêmes nombres I et M, que le résultat soit toujours identique quel que soit N.

Important : le code ci-dessus requiert de lier (*link*) votre exécutable à la librairie `rt`. Pour ce faire, passez `-lrt` en dernier argument à `gcc` au moment de l'édition des liens.