Projet système d'exploitation

Avril 2022

Instructions :
— Le rendu se fera sur la page du cours Moodle, sur l'ENT.
— Un seul rendu par binôme/trinôme
— le rendu devra se composer de :
  — Un rapport qui commencera par le nom de TOUS les membres du binôme/trinôme
  — Le rapport contiendra une description brève des fonctions que vous avez implémentés.
  — Votre code source (et éventuel Makefile)
  — Si vous n'avez pas de Makefile, ajouter la commande de compilation pour chaque exercice dans le rapport. (par exemple : gcc -Wall -pthread syracuse.c -o exe)

1 Exercice 1 - Threads parallèles

Nous allons écrire un programme qui, à l'aide de plusieurs threads, calcule la durée de vol de la suite de Syracuse de tous les entiers allant de 1 jusqu'à n. La suite de Syracuse d'un entier k est définit par récurence comme suit :
$u_O = k$
$u_{n+1} = \frac{u_n}{2}$ si u_n est pair,
$u_{n+1} = 3u_n + 1$ si u_n est impair. Par exemple, la suite de Syracuse de 5 est :

u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	
10	5	16	8	4	2	1	4	2	1	...

On appel durée de vol, le nombre d'étape nécessaire avant d'atteindre le nombre 1. Ainsi, la durée de vol de 10 est 6, car il faut 6 étapes pour atteindre 1 depuis le nombre 10.

1. En partant du fichier syracuse.c fournit, écrivez un programme qui, à l'aide de NB_THREAD threads, calcule la durée de vol de tout les nombres de 1 à $SIZE$.
   Chaque thread calculera uniquement le temps de vol d'une petite partie des nombres. Ainsi, si on utilise 3 threads pour calculer la durée de vol des nombres de 1 à 30, le premier thread calculera pour les nombres de 1 à 10, le second pour les nombres de 11 à 20, et le troisième pour les nombres de 21 à 30.
   La durée de vol de chaque nombre sera stocké dans le tableau nommé syracuse. Ainsi syracuse[10] devra être égale à 6.
   Indications :
   — Pour utiliser les thread en C, il est nécessaire d'utiliser la librairie pthread.h : #include <pthread.h>
   — Il est aussi nécessaire de compiler avec l'option de compilation -pthread : gcc -pthread syracuse.c -o mon_exe
   — Utiliser les fonctions pthread_create (pour créer un thread), et pthread_join (pour attendre que l exécution d'un thread soit terminé).

1.1 N'oubliez pas de tester vos résultats.

2. Améliorer la fonction qui calcul la durée de vol de Syracuse pour qu'elle utilise les valeurs déjà calculé.
   Par exemple lorsque je calcul la durée de vol de 10, à l'étape $u_2 = 16$, si je connais la durée de vol de 16 (qui est de 4), je peux directement obtenir la durée de vol de 10 qui est donc de 4 (durée de vol de 16) + 2 (qui est le nombre d'étape faites par 10 pour arriver à 16) = 6.
   Indications :

— Il sera nécessaire d'avoir initialiser le tableau syracuse à l'aide de valeur (0 ou -1 par exemple) afin de pouvoir différencier les valeurs dont on connait déjà la durée de vol, de celle qui doivent encore être calculé.
— Attention, il se peut aussi, que lors de votre vol vous dépassiez $SIZE$, n'essayer alors pas de consulter le tableau.

2.1 Modifier votre code pour qu'il affiche un message lorsque l'amélioration est utilisé. Faites que le message affiche le nombre dont on calcul la durée de vol, et le nombre dont la durée de vol était déjà connu et qui a été utilisé pour arriver directement au résultat.
Indications :
— Vous devriez remarquer que la durée de vol de certains nombres utilisent des résultats calculés par d'autres threads. Cela est possible car la mémoire est partagé entre les différents thread.

Si vous exécutez plusieurs fois votre programme, vous devriez remarquer que les messages affichés ne sont pas toujours les mêmes, ni dans le même ordre. Expliquez pourquoi cela se produit ?

2 Exercice 2 - Thread, Sémaphore et Gestion de mémoire

Nous allons gérer nous mêmes l'allocation et la désallocation de mémoire d'un programme sur plusieurs threads.

1. Reprendre l'implémentation de la gestion de mémoire simple faites pour l'exercice 2 du TP2.
2. Dans la fonction creerMemoire, ajouter la création d'un morceau de mémoire de taille $tailleMot \times nbMots$. Ce morceau représente la mémoire que l'on gère et qui sera utiliser par notre programme multithreadé.

   Toutes les allocations mémoires dynamique de notre programme doivent se faire avec la fonction allocationMemoire. La mémoire devra être libéré avec la fonction liberationMemoire.
   Cela signifie, pas de tableau statique, ni de malloc autre que celui dans creerMemoire.

Le programme que nous allons utiliser va calculer la n-ème ligne du triangle de Pascal.
La n-ème ligne du triangle de Pascal se calcul à l'aide de la $n-1$-ème ligne de la manière suivante : L'élément i de la ligne n est égale à la somme l'élément i de la ligne $n-1$ et de l'élément $i-1$ de la ligne $n-1$. Plus formellement, le i-ème élément de la n-ème ligne noté $T_{n,i}$ vaut : $T_{n,i} = T_{n-1,i} + T_{n-1,i-1}$.
La tableau suivant représente les 9 premières lignes du triangle de Pascal :

1								
1	1							
1	2	1						
1	3	3	1					
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1

Une façon de voir les règles de calculs est de dire que chaque case du tableau est égal à la case qui est au dessus d'elle, et à la case à gauche de celle-ci. Ainsi, la case rouge est égale à la case bleu plus la case marron.

On va écrire un programme qui calcule la n-ème ligne du triangle de pascal (qui sera passé en argument de ligne de commande) de manière parallèle à l'aide de threads pour être plus rapide.
Afin d'économiser la mémoire, nous n'allons pas stocker tout le triangle, mais uniquement la dernière ligne que nous viendrons de calculer. Cette ligne sera stocké dans un tableau (int *lignePascal).

3.1 Appelé la fonction creerMemoire avec comme paramètres : $4n$ mots, et des mots de la taille d'un int.

Cela sera largement suffisant si les fonctions allocationMemoire et liberationMemoire fonctionnent correctement.
Rappel : n est passé en argument de la ligne de commande.

3.2 Écrivez une fonction calculLigneSuivante qui prend en argument k, et qui lance NB_THREAD threads pour calculer la k-ème ligne du triangle de pascal.
On suppose que lignePascal contient la $k1$-ème ligne du triangle de pascal. Une fois tous les threads terminés, la fonction qui aura récupéré les résultats de chaque thread, met à jour la variable lignePascal pour qu'elle contienne désormais la n-ème ligne du triangle de pascal.
Indications
— Si NB_THREAD est plus grand que k, on ne ne lancera que k threads.
— Sinon, on partage les valeurs à calculer entre les thread. Par exemple, si l'on a 10 thread pour la 50ème ligne, le premier thread calculera les indices de 0 à 4, le second thread de 5 à 9, ... , le 10ème thread de 45 à 49.

Dans l'était actuel de la fonction, il se peut que plusieurs threads utilise la fonction allocationMemoire en même temps. Cela peut poser des problèmes, deux threads différents pourraient voir simultanément le même morceau de mémoire libre, et se l'allouer tous deux.
Ce problème n'est pas inédit. Il se produit à chaque fois que l'on souhaite utilise des threads qui écrivent sur une ressource partagée. Pour gérer cela, il existe un système de verrous que l'on peut utiliser et qui fonctionne de la façon suivante :
Il est possible de fermer ou d'ouvrir un verrou. Si un thread tente de fermer un verrou qui est déjà fermer. Le thread attendra jusqu'à ce que le verrou soit ouvert, le thread fermera alors le verrou et pourra poursuivre. Ce système de verrou est appelé sémaphore. En C les fonctions les semaphores sont implementés dans la librairie : #include <semaphore.h>. Les fonctions des bases sont les suivantes :

— sem_init : Initialise un semaphore
— sem_destroy : Destruction du semaphore
— sem_wait : Verrouillage (fermeture) du semaphore
— sem_post : Déverrouillage (ouverture) du semaphore

Pour plus de détails : man est votre ami.

3.3 Utiliser un semaphore pour sécuriser notre gestion de mémoire.

4 Ecrire une fonction calculLigneN qui calcul la n-ème ligne du triangle de Pascal à partir de rien. Cette fonction appellera plusieurs fois calculLigneSuivante.