Exercices sur les structures de données

Exercice 1

On désire réaliser une petite librairie implémentant une file pour des entiers. La taille de la file sera dynamique. Afin de garantir un comportement cohérent et déterministe dans le cas d'une exécution multi-threadée, la file devra être thread-safe. Vous utiliserez les primitives d'exclusion mutuelle appropriées pour cela. L'interface des fonctions à implémenter se présente comme suit :

- Création d'une file de taille et retour d'un booléen indiquant si la création à réussi:
  ```
  bool queue_create(queue_t **q);
  ```
- Détruire une file et retour de booléen si la destruction a réussi:
  ```
  bool queue_destroy(queue_t *q);
  ```
- Enfiler une valeur et retour de booléen si l'opération a réussi:
  ```
  bool queue_enqueue(queue_t *q, int val);
  ```
- Défiler une valeur et la stocker dans `val`. Cette fonction retourne un booléen pour savoir si l'opération a réussi:
  ```
  bool queue_dequeue(queue_t *q, int *val);
  ```

Décider ce que contiendra la structure `queue_t` définissant un objet de type file.

Remarque: Penser à insérer des assertions dans le code aux endroits nécessaires.

Exercice 2

On désire modéliser, à l'aide de threads, un "workflow" d'opérations réalisées en cascade sous forme "d'étages", un petit peu comme ce qui serait réalisé sur une chaîne de montage. Les étages à implémenter sont les suivants:

- Un premier étage `create` qui va générer un vecteur de valeurs aléatoires [1].
- Un deuxième étage `sinus` qui va appliquer la fonction sinus à chaque valeur du vecteur.
- Un troisième étage `positive` qui va mettre à zéro toutes les valeurs négatives du vecteur.

1. Pour générer les valeurs aléatoires utilisez le générateur réentrant `rand_r()`.

— Un quatrième étage `amplification` qui va multiplier chaque valeur du vecteur par un coefficient d'amplification.
— Enfin, un cinquième étage qui va calculer et afficher la somme du vecteur en provenance de l'étage précédent.

L'exécution des 5 étages représente un cycle d'exécution. Le programme prendra en argument le nombre de cycle à exécuter. Commencer par réaliser une version séquentielle du programme. Ensuite, implémenter une version multi-threadée dont le but est de diminuer le temps d'exécution du programme.

Le programme multi-threadé prendra un argument supplémentaire: le nombre de threads à utiliser. Pour simplifier, on part du principe que la taille du vecteur de chaque étage est divisible par le nombre de threads. Vous êtes libres d'utiliser les primitives de synchronisation ou d'exclusion mutuelle de votre choix.

Attention à joindre tous les threads créés et détruire toute(s) barrière(s) avant la terminaison du programme. Enfin, afficher le temps obtenu après exécution du programme.

Questions 1

Exécuter le programme avec un nombre différent de threads.

1. Quel est le gain obtenu pour votre version multi-threadée par rapport à la version séquentielle?
2. Est-ce que le temps obtenu varie avec le nombre de threads?

Mesures de temps

Les mesures de temps peuvent être effectuées avec la fonction `clock_gettime()` de la librairie `time.h` comme montré ci-dessous:

```
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);

// code à mesurer

clock_gettime(CLOCK_MONOTONIC, &finish);
double seconds_elapsed = finish.tv_sec-start.tv_sec;
seconds_elapsed += (finish.tv_nsec-start.tv_nsec)/1.0e9;
```

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