HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Prof. Yann Thoma
Gregory Medwed
Fiorenzo Gamba

Laboratoire de Programmation Concurrente
semestre printemps 2020

Exclusion mutuelle

Temps à disposition : 2 périodes (travail débutant en semaine 2)

1 Objectifs pédagogiques

— Résoudre le problème d'accès à une ressource partagée par attente active.

2 Récupération des sources

La récupération du code source s'effectue à l'aide de la commande git suivante, exécutée dans votre répertoire git PCO :

```
$ git pull
```

3 Cahier des charges

Le programme qui vous est donné effectue une simple incrémentation d'un nombre un certain nombre de fois. Cette incrémentation est faite par plusieurs threads et ce potentiellement en parallèle (ou pseudo-parallèle). Si vous le lancez et le compilez, vous devriez observer des valeurs étranges pour la valeur finale du compteur.

Le dialogue lancé par l'application vous permet de choisir le nombre de threads à lancer et le nombre d'itérations à effectuer. Une pression sur le bouton *Start* lance les threads. Lorsqu'ils ont tous terminé leur travail, la valeur finale du compteur ainsi que le ratio (valeur finale observée/valeur finale attendue) sont affichées.

Durant le précédent laboratoire vous avez tenté quelques expériences pour améliorer la valeur finale observée. Il n'a pas été facile de trouver un algorithme fonctionnel, et pour cause. Ce problème est loin d'être aisé.

Vous allez maintenant implémenter une solution fonctionnelle en exploitant un algorithme existant. Vous avez le choix entre :

1. Peterson : `https://en.wikipedia.org/wiki/Peterson's_algorithm`
2. Lamport : `https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm`
3. Eisenberg & McGuire : `https://en.wikipedia.org/wiki/Eisenberg_%26_McGuire_algorithm`
4. Szymański : `https://en.wikipedia.org/wiki/Szyma%C5%84ski%27s_algorithm`

Ces algorithmes fonctionnent si les mémoires caches sont cohérentes. Il y a donc fort à parier qu'il soit nécessaire de forcer cette cohérence à certains endroits de votre code. Pour ce faire vous aurez besoin d'une barrière mémoire :

`std::atomic_thread_fence(std::memory_order_acq_rel)`

Ceci permet de synchroniser les mémoires caches, et en le plaçant à des endroits judicieux cela permet de rendre le code plus sûr (bien que moins performant).

Vous aurez aussi vraisemblablement besoin d'utiliser la fonction `std::this_thread::yield()` si le nombre de threads dépasse le nombre de coeurs de votre processeur. A vous de voir si son utilisation est nécessaire et pourquoi.

Pour réaliser ce labo, un nouveau code vous est fourni. Vous pouvez y observer la présence d'une classe abstraite `CriticalSection` ainsi qu'une classe concrète `WonderfulCriticalSection`. A vous d'implémenter cette deuxième classe. La fonction d'initialisation est appelée dans `dialog.cpp`, et un pointeur sur cet objet est passé à la méthode `runTask()`.

4 Bonne pratique

Que pensez-vous du code suivant ?

```
class CriticalSection
{
public:
    virtual ~CriticalSection() = default;

        /**
         * @brief Méthode initialisant la section critique
         * @param nbThreads Le nombre de threads maximal géré par la section critique
         * Cette méthode doit être appelée avant l'utilisation de la section critique
         */
        virtual void initialize(int nbThreads) = 0;

        /**
         * @brief Protocole d'entrée dans la section critique
         * @param index Indice de la tâche appelante
         */
        virtual void lock(int index) = 0;

        /**
         * @brief Protocole de sortie de la section critique
         * @param index Indice de la tâche appelante
         */
        virtual void unlock(int index) = 0;
};
```

5 Travail à rendre

Rendez votre code selon la procédure décrite dans les consignes de laboratoire. Un mini rapport est demandé. Expliquez-y si tout s'est déroulé comme prévu, ou si vous avez dû utiliser des barrières mémoires et/ou des `yield()`.