HEIG-VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Chargés de cours : Florian Vaussard, Rick Wertenbroek

Laboratoire de Programmation Concurrente

semestre automne 2022

Prise en main des threads et observation de l'ordonnancement

Temps à disposition : 4 périodes (2 sessions de laboratoire)

1 Objectifs pédagogiques

— Se familiariser avec l'environnement de programmation.
— Réaliser son premier programme concurrent en C++.
— Observer l'influence de l'ordonnanceur.
— Tester les problèmes de concurrence.

2 Avant de commencer

Ce laboratoire va nous permettre de nous familiariser avec l'environnement de développement, ainsi qu'avec la notion de threads et des problèmes que nous pourrons rencontrer. Il est décomposé en plusieurs parties, chacune mettant en jeu un nouveau projet. Le but y est chaque fois de faire quelques essais, et de répondre aux questions posées dans cet énoncé.

⚠ Avant de commencer il est fortement recommandé de consulter le guide de démarrage Qt disponible sur cyberlearn.

Les commandes suivantes doivent être faites dans un terminal, afin d'installer quelques éléments :

```
sudo pip3 install pandas
sudo pip3 install matplotlib
sudo update_pcosynchro
```

3 Interlacement sur std : :cout

Le dossier `interlacement` contient un programme qui lance 3 threads. Observez la structure de ce programme, et notamment comment les threads sont créés, via la classe `PcoThread`.

Dans la fonction `run()`, ajoutez de quoi afficher un message sur la ligne de commande via `std::cout`, notamment en utilisant le paramètre `tid` pour discriminer les theads.

Qu'observez-vous ?

Remplacez maintenant votre `std::cout` par `logger()`. Pour ce faire il faudra ajouter

```
#include <pcosynchro/pcologger.h>
```

en début de fichier, et

```
logger().setVerbosity(1);
```

au début du `main()`.

Qu'observez-vous ?

Cette fonction `logger()` exploite une classe `PcoLogger`. Qu'est-ce qui a pu être mis en place dans `PcoLogger` pour que l'affichage se comporte ainsi ?

4 Ordre d'exécution

Le dossier `order` contient un programme qui lance 2 threads, chacun modifiant une variable partagée. Le thread 0 place la valeur 0 dans la variable, et le thread 1 place la valeur 1. Notons également que le programme principal, de son côté y place la valeur 2.

Avant le lancement, pouvez-vous prédire la valeur finale de la variable partagée ?

Lancez-le plusieurs fois.

Observez-vous quelque chose de surprenant ?

Qu'en est-il de la reproductibilité des tests ?

Avec ce que vous connaissez, pourriez-vous modifier le code pour faire que les trois modifications de variables se fassent, mais que le résultat final soit toujours 1 ? (Ceci sans déplacer du code, mais en y ajoutant des éléments)

5 Timestamps

Le dossier `timestamps` contient un programme qui lance 3 threads. Chacun de ces threads est composé d'une boucle qui récupère à chaque fois le temps courant en nanosecondes ainsi que l'identifiant du processeur utilisé et stocke ces informations dans un tableau. Après s'être exécuté, le proramme génère un fichier composé des données observées, et lance automatiquement un script python pour les afficher. Il est également possible d'exécuter ce script à nouveau en l'appelant en ligne de commande `./show_graphic_from_exec.py`. Il faut toutefois qu'il soit dans le même répertoire que le fichier de données. Lancez le programme à plusieurs reprises pour observer les différents résultats.

Vous pouvez à nouveau tester votre code en forçant l'usage d'un ou plusieurs coeurs :

— Sous Linux, en ligne de commande, pour un seul coeur : `taskset 1 ./timestamps`

Pour info, l'exécutable `timestamps` se trouve dans le dossier `code/build-timestamps-...` une fois le code compilé dans QtCreator.

Vous pouvez aussi changer le nombre de threads dans le fichier source.

Qu'observez-vous ? Est-ce que cela a du sens lorsque vous forcez à moins de coeurs ? Attention au nombre de coeurs alloués à votre VM, évidemment.

6 Incrémentation de compteur

Le programme présent dans le dossier `counter` 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.

A partir du code qui vous est donné, tentez quelques expériences pour améliorer la valeur finale observée. Pour ce faire, essayez de mettre en place un méchanisme d'attente active en exploitant des variables partagées par l'ensemble des threads. Utilisez des variables de classe (`static`) pour réaliser cela. Le fichier à modifier est `mythread.cpp`. Vous pouvez y observer l'incrémentation d'un compteur partagé.

Durant vos essais, vous pouvez tenter d'utiliser une barrière mémoire, qui va garantir que tous les load et store présents avant elle sont exécutés avant ceux qui sont présents après :

`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) [1].

Enfin, n'hésitez pas à tester votre code en forçant l'usage d'un seul coeur :

— Sous Linux, en ligne de commande : `taskset 1 ./PCO_Labo_1`

L'exécutable se trouve dans le dossier `labo1_counter/build-PCO_Labo_1-...` une fois le code compilé dans QtCreator.

7 Travail à rendre

Il s'agit d'une prise en main. Amusez-vous, posez toutes les questions nécessaires et tentez d'améliorer le comportement de votre programme. Le labo ne sera pas évalué.

A la fin de ce laboratoire, rendez le code du compteur (dernière étape) selon les consignes de laboratoires disponibles sur cyberlearn. Certains codes seront analysés en classe.

1. `https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence`