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

Yann Thoma
Olivier Cuisenaire
Florian Vaussard

Laboratoire de Programmation Concurrente

semestre automne 2024

Thread pool

Temps à disposition : 6 périodes (travail débutant en semaine 12)

1 Objectif pédagogique

Réaliser un pool de threads pour la répartition de charge, via un **moniteur de Hoare**.

2 Enoncé du problème

Lors de traitements conséquents en termes de consommation de temps CPU il peut être intéressant de répartir la charge sur plusieurs coeurs, et donc plusieurs threads. Le lancement d'un thread est toutefois une opération lourde, et il est donc intéressant de disposer d'un pool de threads prêts à exécuter des calculs sans l'overhead du lancement. Afin d'être suffisamment générique, et offrir une solution réutilisable, nous allons réaliser ceci sous la forme d'un mécanisme classique appelé *Thread Pool*.

Avec un thread pool, l'idée est d'allouer dynamiquement des threads au fur et à mesure que cela devient nécessaire dans une "piscine" de threads. La piscine a une taille maximale : une fois sa capacité atteinte, les threads sont "recyclés" pour traiter les requêtes suivantes.

Le système de thread pool limite donc le nombre maximal de threads, notamment pour ne pas surcharger la machine. Si le système est temporairement débordé, la latence des calculs augmente malgré tout, mais avec la garantie qu'il ne va pas s'écrouler sous la demande.

Nous pourrons voir le thread pool comme une sorte de serveur de requêtes, où chaque requête est un calcul particulier à exécuter. Il faut aussi ajouter au thread pool un mécanisme de gestion de surplus de requêtes : en effet, sans limitations, la file d'attente des requêtes peut devenir suffisamment grande pour arriver à la limite de la mémoire du système. Pour cette raison, il s'agit aussi de limiter la file à une taille maximale et rejeter les requêtes entrantes si elle est pleine. Cette mesure produit une expérience temporairement dégradée, mais garantit que le serveur peut récupérer, une fois la "tempête" passée.

Nous aimerions également éviter de laisser des threads en vie trop longtemps s'ils n'ont rien à faire. Ceci permettra de libérer de la mémoire si le thread pool est sous-exploité. Pour ce faire, le thread pool disposera d'un timeout, en millisecondes, fourni au constructeur. Un thread devra donc se terminer s'il a été inactif pendant ce temps (c'est-à-dire qu'il a terminé une exécution de tâche et qu'il est resté en attente pendant cette durée).

Le rôle des threads du thread pool sera donc d'exécuter des tâches particulières. Pour ce faire nous allons définir l'abstraction suivante, dont la définition est proposée ci-dessous :

```
class Runnable {
public:
    /*
     * An empy virtual destructor
     */
    virtual ~Runnable() {}
```

```
    /*
     * Function executing the Runnable task.
     */
    virtual void run() = 0;

    /*
     * Function that can be called from the outside, to ask the ↘
     →cancellation
     * of the runnable.
     * Shall be called by the threadpool if the Runnable is not started.
     */
    virtual void cancelRun() = 0;

    /*
     * Simply retrieve an identifier for this runnable
     */
    virtual std::string id() = 0;
};
```

Et voici le squelette de déclaration pour la classe `ThreadPool` qui utilise cette définition :

```
class ThreadPool
{
public:
    ThreadPool(int maxThreadCount, int maxNbWaiting, std::chrono::↘
    →milliseconds idleTimeout) {...}

    ~ThreadPool() {....}

    /* Start a runnable. If a thread in the pool is available, handle ↘
    →the runnable with it. If no thread is available but the pool can ↘
    →grow, create a new thread and handle the runnable with it. If no ↘
    →thread is available and the pool is at max capacity and there are ↘
    →less than maxNbWaiting threads waiting, block the caller until a ↘
    →thread becomes available again, and else do not run the runnable.
     If the runnable has been started, returns true, and else (the last ↘
     →case), return false. */
    bool start(std::unique_ptr<Runnable> runnable) {...}

    /* Returns the number of currently running threads. They do not need↘
    → to be executing a task, just to be alive.*/
    size_t currentNbThreads() {....}
};
```

L'interface est donc très simple, et le développeur désirant exploiter le thread pool n'aura qu'à créer des sous-classes de `Runnable` embarquant assez d'information pour pouvoir exécuter un traitement pertinent.

Le constructeur prend en paramètres le nombre maximum de threads du pool, le nombre maximum de requêtes pouvant être en attente, et le temps maximum d'inactivité du thread. Une deuxième méthode doit permettre de savoir combien de threads sont actuellement en vie. Celle-ci est uniquement pertinente pour la mise en place des tests.

3 A faire

Implémentez la classe `ThreadPool` qui réalise ce concept d'allocation dynamique et de recyclage de threads, et ce à l'aide d'un **moniteur de Hoare**. Utilisez la classe abstraite `Runnable` pour représenter les requêtes. Notez que le code est censé gérer correctement la destruction du thread pool, en attendant que les threads en cours se terminent tout en évitant de relancer de nouveaux threads.

N'hésitez pas à discuter de vos choix architecturaux avec le professeur ou l'assistant, afin de ne pas vous perdre dans une direction qui ne serait pas la bonne.

Le projet fourni offre cinq scénarios de test qui permettront de vérifier une partie des fonctionnalités de votre implémentation du thread pool.

⚠ Ce n'est pas parce que les scénarios de test sont validés que votre code est forcément bon. Vous avez le droit d'ajouter des scénarios si vous le désirez. Pour ce faire, il suffit d'ajouter une méthode en-dessous de `testCase5()` et de vous inspirer de ce qui vous est fourni.

4 Travail à rendre

— Comme pour les laboratoires précédents, la description de l'implémentation, ses différentes étapes, la manière dont vous avez vérifié son fonctionnement, la réponse aux questions et toutes autres informations pertinentes doivent figurer dans un petit rapport. Les fichiers sources doivent évidemment être correctement documentés et commentés.

— Inspirez-vous du barème de correction pour savoir là où il faut mettre votre effort.

5 Barème de correction

Conception, conformité au cahier des charges et simplicité	45%
Exécution et fonctionnement	10%
Codage	15%
Documentation	20%
Commentaires au niveau du code	10%