Rapport Labo 6

Ali Zoubir & Kenan Augsburger

January 14, 2025

Contents

1. Informations sur le projet

1.1. Répertoire Git (sur GitHub)

mondotosz/pco_lab06°

1.2. Membres

- Ali Zoubir°
- Kénan Augsburger°

1.3. Introduction du problème

Dans ce laboratoire, l'objectif était de concevoir et implémenter un pool de threads capable de gérer efficacement l'exécution de tâches concurrentes tout en respectant les contraintes définies : limitation des threads actifs, gestion de la surcharge via une file d'attente limitée, et recyclage des threads inactifs après un délai donné.

2. Implémentation et choix

2.1. Structure Générale du Code

Runnable Une interface représentant une tâche exécutable :

- `run()` : contient la logique à exécuter.
- `cancelRun()` : permet d'annuler l'exécution.
- `id()` : fournit un identifiant unique pour chaque tâche.

ThreadPool La classe principale, utilisant un moniteur de Hoare pour gérer la synchronisation :

Constructeur :

- `maxThreadCount` : nombre maximum de threads dans le pool.
- `maxNbWaiting` : taille maximale de la file d'attente.
- `idleTimeout` : délai d'inactivité avant qu'un thread ne soit terminé. Un thread séparé (`timer_thread`) est initialisé pour surveiller les timeouts.

Méthodes :

- `start()` : Ajoute une tâche dans la file ou la rejette si elle est pleine. Si un thread est disponible, il traite immédiatement la tâche. Sinon, un nouveau thread est créé (si le pool n'a pas atteint sa capacité maximale).
- `worker()` : Gère l'exécution des tâches pour chaque thread. Attend une tâche ou termine après un délai d'inactivité.
- `timer()` : Surveille les threads inactifs et les termine lorsqu'ils dépassent idleTimeout.
- `Destructeur` : Termine proprement les threads et libère les ressources associées.

2.2. Architecture de la classe

La classe principale `ThreadPool` repose sur un moniteur de Hoare pour gérer la synchronisation. Les threads sont encapsulés dans une structure qui contient l'état de chaque thread (actif, en attente, ou expiré).

Pseudocode pour `ThreadPool` :

```pseudocode
Classe ThreadPool :
  initialiser(max_threads, max_queue, timeout) :
    // Préparer les structures pour la gestion des threads

  démarrer(tâche) :
    si (file pleine) alors rejeter(tâche)
    sinon assigner_à_thread(tâche)

  boucle_thread() :
    tant que (actif) :
      si (délai dépassé) alors terminer_thread()
      sinon exécuter_tâche()
```

2.3. Méthodes worker

- `worker_loop` : Chaque thread attend une tâche ou termine après un délai d'inactivité.

Pseudocode simplifié de la logique :

```pseudocode
si (file non vide) alors :
  tâche = retirer_file()
  exécuter(tâche)
sinon si (temps_inactivité atteint) :
  terminer_thread()
```

2.4. Points Clés sur le Thread de Timeout

Gestion du Timeout dans le Thread Timer :

Le thread **`timer_thread`** parcourt les threads actifs. Si un thread est inactif et dépasse `idleTimeout`, il est marqué comme expiré (`timed_out`) et réveillé via un signal. Les threads expirés sont joints (avec join()) et supprimés de la liste. Découplage entre le Timer et les Workers :

Les threads "workers" ne gèrent pas eux-mêmes leur timeout, ce qui simplifie leur logique. Le thread timer agit comme un observateur, surveillant les états des threads et agissant en conséquence.

2.5. Subtilités concurrentielles

L'implémentation évite les blocages grâce à l'utilisation de conditions de synchronisation, garantissant un accès sécurisé et cohérent aux données partagées. La destruction du pool est gérée avec rigueur pour prévenir toute fuite de threads. Une attention particulière a été portée au rejet des tâches impossibles à exécuter et à la terminaison propre des threads, notamment pour garantir la réussite des tests.

3. Tests

3.1. Scénarios de test

Les tests incluent :

1. Gestion de 10 tâches avec 10 threads.
2. Surcharge de la file d'attente.
3. Recyclage des threads inactifs.

Pseudocode pour un test :

```pseudocode
pool = initialiser_pool(10, 50, 100ms)
envoyer_tâches(pool, 10)
vérifier(toutes_tâches_finies)
```

3.2. Résultats

Les tests montrent que le pool respecte les limites et fonctionne efficacement dans des scénarios variés en respectant les contraintes de temps.

4. Remarques et conclusion

4.1. Améliorations possibles

- Réduction des ressources consommées par les threads inactifs.
- Ajout de métriques pour surveiller les performances.

4.2. Conclusion

Ce laboratoire a permis de concevoir un pool de threads capable de gérer des tâches concurrentes sous des contraintes strictes ; Telles que la limitation des threads actifs, la gestion d'une file d'attente et le recyclage des threads inactifs. L'utilisation d'un moniteur de Hoare et d'un thread dédié à la gestion des timeouts a assuré une synchronisation sécurisée. Les tests ont confirmé la conformité du système, offrant une solution adaptable à des environnements à charge variable.