Rapport d'OS

Andrius Ezerskis & Moïra Vanderslagmolen & Milan Skaleric

December 6, 2022

1 Introduction

Après une base de données simple, avec des requêtes et du monitoring, il nous a été demandé de réaliser une mise à jour afin de permettre des connexions de multiples clients sur un serveur.
Ces clients doivent pouvoir faire les mêmes requêtes qu'initialement, avec quelques améliorations pour réduire le temps d'exécution des requêtes et pour ne pas surcharger le serveur. Vous trouverez, dans ce rapport, nos choix d'implémentation et les difficultés que nous avons rencontrées au fil de la réalisation de ce projet ainsi que les mécanismes de synchronisation utilisés, une comparaison détaillée entre threads et processus et des propositions d'améliorations relatives aux performances et au fonctionnement.

2 Choix d'implémentation et difficultés rencontrées

2.1 Multi-threading

Pour éviter les problèmes de surcharge de serveur, nous avons limité le nombre de clients à 50.

2.2 Signaux

Lors de l'implémentation des signaux, nous avons eu du mal à fermer les clients lorsque le serveur attrapait un signal tel que Ctrl + C. En effet, le client lit deux files descriptors différents, le stdin et le socket du serveur. Nous écrivions donc dans le socket du serveur un message ('stop') afin que le client se ferme dès qu'il le lit. Seulement, le lecture du stdin est bloquant, le client ne lisait donc dans le socket du serveur que lorsque l'utilisateur écrivait dans le stdin.
Nous avons d'abord essayé les commandes epoll, poll ou select, qui permettent de signaler lorsqu'un file descriptor est prêt, mais nous nous sommes rendus compte que ce n'était pas adapté à notre code, vu que nous n'avions que deux files descriptors, et donc nous avons réglé ce problème plus facilement en rendant les deux files descriptors non-bloquants.

3 Synchronisation

Afin de synchroniser les clients entre eux, nous avons fait l'usage de mutex(mutual exclusion) qui permettent à deux threads de ne pas écrire ou lire en même temps dans la mémoire. En effet, si deux threads écrivent en même temps, il est possible que les écritures se chevauchent et donc que l'étudiant écrit dans la base de données soit incorrect.
Dans le pseudo-code, nous avons 3 mutex : reader registration, writer access et new access. Le mutex writer access permet tout simplement de bloquer la mémoire lors des opérations d'écriture. Le reader registration permet . Le new access est un peu plus complexe. Il permet de faire les accès mémoire dans l'ordre dans lequel elles ont été écrites. En effet, si un premier thread veut effectuer une lecture et un second veut effectuer une écriture, le temps que le premier thread lock le reader registration, le second thread pourrait lock le write access, et donc la lecture serait bloquée alors que c'était le premier thread. Le new access règle ce souci en bloquant l'accès au write access.

4 Processus vs Threads

Le processus est indépendant, et donc même si le processus principal se crashe, les processus enfants peuvent continuer d'exécuter des tâches.

La gestion en mémoire partagée est beaucoup plus simple avec des threads. Les threads sont plus rapides que les processus à la création et à la destruction.

5 Améliorations

Pour améliorer la rapidité de notre programme, nous avons eu plusieurs idées.

La création et destruction de threads est assez coûteuse, nous pourrions donc laisser un thread gérer un certains nombres de clients, par exemple 5 clients. Au lieu d'écrire dans la requête dans un socket et d'écrire le résultat de ce même socket, nous pourrions écrire dans un fichier, pour ne pas être limité en caractères.

Tout d'abord

6 Conclusion

En conclusion, malgré les problèmes survenus, nous avons réussi à implémenter une base de données serveur/-clients qui s'exécute rapidement et efficacement.
Le code est maintenable et séparé en plusieurs fichiers, afin de faciliter la correction de bugs et d'erreurs éventuels, ainsi que la maintenabilité du projet dans son ensemble.