Rapport d'OS - Projet 2

Andrius Ezerskis & Moïra Vanderslagmolen & Milan Skaleric

December 8, 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

Implémentation Nous avons implémenté une database avec relation client-serveur en multi-threading. Pour cela, le serveur lance un thread à chaque nouveau client. Le client envoie sa requête au serveur, et le thread du serveur écrit dans un file lié au socket de son client.

2.1 Fin d'un socket

Lorsque nous avons commencé à lancer les tests, nous avons vu que le socket 'gardait' l'ancienne commande. En effet, si la requête était plus longue que le résultat, le résultat s'affichait et le reste de la requête aussi. Pour régler cela, nous avons rajouté manuellement un '0' à la fin du buffer lu, pour signaler la fin du message au read.

2.2 Multi-threading

Pour éviter les problèmes de surcharge de serveur, nous avons limité le nombre de thread tournant en même temps à 20. À chaque fois que le nombre de thread atteint 20, on attend que tous les threads se terminent et puis on peut de nouveau lancer des threads.

2.3 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 nous avons donc réglé ce problème plus facilement en rendant les deux files descriptors non-bloquants.

2.4 Longueur d'un socket

Avant d'avoir ajouté le flag non-bloquant au socket et au stdin, nous avions limité la pipe à 2048 caractères, ce qui nous posait problème lors du 'select', car il fallait afficher tous les étudiants retournés par le serveur. Lorsqu'on a ajouté le flag bloquant, ce problème s'est réglé, car le serveur écrit dans le file descriptor lié au socket, et le client ne s'arrête pas sur le fgets juste après le read, et continue à lire.

Rendre le pipe non-bloquant nous a aidé sur deux problèmes, mais nous avons rajouté un sleep d'un dixième de seconde, afin d'optimiser le programme.

3 Synchronisation

Afin de synchroniser les clients entre eux, nous avons fait l'usage de mutex(mutual exclusion). Les mutex permettent à deux ou plusieurs 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. Lors d'une lecture et d'une écriture en parallèle, il se peut que la lecture se termine avant l'écriture, ce qui résulte en une lecture faussée.

Dans le pseudo-code, nous avons 3 mutex: reader registration, writer access et new access.

Lors de l'écriture, le thread bloque la mémoire, écrit dedans et puis la débloque.

Les mutex gérant la lecture permettent à plusieurs lecteurs de lire en même temps la mémoire. En effet, ils ne modifient pas la mémoire, ils peuvent donc être à plusieurs. Lorsque le premier thread lit, il bloque la mémoire au niveau de la database, pour éviter que un autre thread écrive en même temps. D'autres threads qui lisent en même temps ne devront pas bloquer la mémoire, car le nombre de readers_c est supérieur à 0. Si le nombre de lecteurs n'est pas égal à 0, la mémoire ne sera pas débloquée.

Le new access est un peu plus complexe. Il permet de faire les accès mémoire dans l'ordre dans lequel les requêtes ont été écrites et ainsi éviter les problèmes de famine. En effet, avec le mutex new-access, les requêtes read-write-read s'exécuteront selon le mode suivant : la lecture commence par bloquer le new_access, bloque donc la mémoire, débloque le new_access et commence ses opérations de lecture. Le thread commençant l'écriture est bloqué par le new_access, et attend que la mémoire se débloque. En attendant, le dernier thread voulant effectuer la lecture est bloqué au new_access. Lorsque le premier lecteur a terminé, il débloque la mémoire, et donc l'écriture peut commencer. Le troisième lecteur est alors bloqué au write_access. Ensuite, l'écriture se termine, et le lecteur peut donc bloquer la mémoire et lire dedans.

Sans le new_access, les requêtes read-write-read se seraient exécutées différemment. Le lecteur aurait tout d'abord bloqué le write_access. Le thread effectuant l'écriture s'arrête au niveau du bloquage de write_access, tandis que le troisième thread (le lecteur) commence à lire en mémoire, car il n'est pas le seul lecteur (et ne doit donc pas bloquer le write_access).

Enlever le write_access peut causer deux problèmes majeurs. Premièrement, cela peut causer des problèmes de famine. En effet, si une requête en lecture se produit, puis une requête en écriture, puis des dizaines de requêtes en lecture, la requête en écriture ne pourra jamais se produire. Deuxièment, dans une base de données, il est important que les requêtes se fassent dans l'ordre dans laquelle elles ont été écrites. Sans le new_access, notre programme ne peut pas assurer l'ordre.

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. Ce n'est pas le cas avec les threads.

La gestion en mémoire partagée est beaucoup plus simple avec des threads. Ceux-ci partagent par défaut la mémoire virtuelle d'un même processus, cela nous évite donc de devoir implémenter la gestion de la mémoire partagée nous-mêmes.

En effet, lors du projet 1, nous avons du implémenter nous-mêmes la gestion en mémoire partagée, à l'aide de diverses librairies comme shmat. shmat permet de partager un file descriptor entre plusieurs processus enfants ainsi que le processus parent. Il y a eu beaucoup de contraintes avec ce file descriptor, notamment le fait qu'un processus enfant ne peut pas augmenter la taille du file descriptor, seulement le parent est autorisé à le faire.

Lors du projet 2, nous avons simplement dû faire usage de mutex, pour éviter que les threads écrivent et/ou lisent dans la mémoire en même temps.

Les threads sont également plus rapides que les processus à la création et à la destruction, et prennent moins de temps pour le changement de contexte, car ils sont plus légers.

La gestion des signaux est très différente entre les processus et les threads. Pour les processus, il faut envoyer un signal à chaque processus enfant et au processus parent, tandis que pour les threads, il suffit d'envoyer un signal au processus principal (et donc au thread principal), et de l'ignorer dans les threads créés.

Les threads prennent aussi moins de place que les processus, car ils partagent la même mémoire, contrairement aux processus.

5 Améliorations

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

Tout d'abord, 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.

Nous pourrions aussi lancer plusieurs serveurs, au lieu d'en avoir qu'un seul, afin de supporter plus de monde.

Nous pourrions aussi créer un index d'étudiants grâce à std::map, ou un tableau d'étudiants, avec l'id en clé afin de vérifier si l'id est déjà dans la database beaucoup plus rapidement.

Nous pouvons aussi prévoir beaucoup plus d'espace pour la base de données au démarrage. Si plusieurs accès en lecture se produisent, nous pouvons les 'rassembler' tout en évitant de provoquer une famine.

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.

Il est possible de le rendre encore plus performant, notamment à l'aide de multiples serveurs. L'utilisation des processus et des threads dépend surtout de l'utilisation qu'on veut en faire, chacun ont leur avantages et leurs inconvénients. Les processus sont plus robustes, et les threads plus rapides.

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.