Rapport Projet OS partie 2

Realisé par: Kevin Issa et Arthur Installé
matricule Kevin: 000514550
matricule Arthur: 000495303
email: kevin.issa@ulb.be et arthur.installe@ulb.be

Décembre 2022

Table des matières

1. Introduction

Maintenant que l'on a pu implémenter notre base de données en utilisant les processus, afin de s'exercer sur le reste de la matière, une partie importante à parcourir reste les threads, les sockets et ce qui les lies.

Dès lors, cette partie repose sur la communication entre plusieurs clients et un serveur qui contiendra cette base de données. Étant donné, que nous devons remplacer les processus par des threads, faire en sorte que le programme fonctionne comme prévu amène naturellement des difficultés propre à ceux-ci.

Ainsi dans ce rapport, nous allons vous expliquer, notre manière d'implémenter ces différentes fonctionnalités ainsi que leur fonctionnement si nécessaire. Par la suite, nous étudierons les différences entre les processus et les threads pour finir sur les problèmes que nous avons rencontré et notre cheminement pour les résoudre.

2. Implémentation des fonctionnalités

2.1. Threads

Afin de créer un thread pour chaque client sans pour autant rendre le code compliqué. Nous avons décide de créer au préalable une structure sockaddr_in contenant l'adresse du client puis de mettre le serveur en attente de la demande de connexion. Lorsqu'un client demande de se connecter au serveur, celui-ci lui attribue l'adresse déjà créer et accepte la demande.

Par la suite, le thread se crée en prenant soin d'initialiser une structure data storage, propre a chaque client, qui contient tout ce dont le thread aura besoin pour gérer les requêtes. Cette structure a une deuxième utilité qui est de permettre a la fonction handle_connection de simuler la possibilité d'avoir plusieurs paramètres.

En effet, étant donné que les fonctions pour les threads ne peuvent avoir qu'un seul argument, il en devient difficile au premier abord d'avoir accès à tout les paramètres nécessaires pour exécuter une requête dans le thread. Dès lors, l'idée de passer par une structure de donnée pour contourner ce problème fut vite évoquer, ce qui nous permet de faciliter énormément le code.

2.1.1. Représentation du code

```
struct sockaddr_in client_adrr;
size_t addrlen = sizeof(client_adrr);
int client_socket = accept(*socket_server, (struct sockaddr *)&client_adrr, (sock

pthread_t client_thread;
data_storage* client_data=(data_storage*)malloc(sizeof(client_data));

...initialisation des données

pthread_create(&client_thread,NULL,handle_connection,client_data);
```

3. Différences entre les threads et les processus

3.1. Avantages et inconvénients des processus

Pour commencer, d'après les observations que nous avons pu tirer durant ces 2 parties du projet et en reconsultant le cours, nous avons remarqué que les processus n'avaient que très peu d'avantage. En effet, l'avantage principal est que la mort d'un processus n'entraîne pas celle des autres, contrairement aux threads où la mort du processus entre la mort de tout les threads, et qu'ils sont chacun indépendant avec leur propre zone mémoire, ce qui n'entraîne pas de concurrence lorsque l'on modifie des données.

Cependant, cette séparation dans le partage des données, entraîne des difficultés supplémentaires lorsqu'il s'agit de les faire communiquer entre eux afin de modifier de manière synchroniser cette donnée. Pour cela, il faut utiliser une mémoire partagée entre les processus, qui n'est pas forcément évident à mettre en place. De plus, il sera nécessaire de créer des pipes pour relier chaque processus secondaire au processus principale, ce qui complexifie le code et peut entraîner plusieurs bug. Par exemple, si un pipe se ferme mal en lecture ou en écriture ou si celui-ci reçoit trop de données étant donné qu'il a une faible capacité.

Enfin, la création d'un nouveau processus nécessite de copier la mémoire et le heap du processus actuel, en passant par le système, et de lui adresser une zone mémoire propre à lui, ce qui prend du temps et alourdit la mémoire. Dès lors, nous pouvons voir que les processus ont beaucoup de plus d'inconvénients que d'avantages.

3.2. Avantages et inconvénients des threads

Les principaux avantages des threads c'est qu'ils ont la même mémoire commune, ce qui provoque également un problème dont nous parlerons plus tard, et qu'ils sont beaucoup moins dur à mettre en place . En effet, un thread contrairement à un processus n'est pas un nouveau programme mais uniquement une partie du processus en cours d'exécution.

Dès lors, les threads ont uniquement que le stack qui est différent entre chaque thread, tout le reste est partagé que ce soit la zone mémoire, le file descriptor ou le heap. Par conséquent, ils sont beaucoup plus rapide et léger à créer étant donné que l'on doit juste leur attribuer un stack dédié.

De plus, ils ont besoin de rien d'autre afin de communiquer entre eux étant donné qu'ils sont tous dans la même zone mémoire lors de leur création, ce qui limite le nombre de problèmes lié à la communication des données. Un autre avantage moins important est qu'ils prennent moins de temps à se terminer.

Cependant, leur utilisation entraîne également quelques difficultés non négligeable, le principal consiste en l'existence d'une concurrence entre les threads. En effet, étant donné que la zone mémoire est commune, la modification d'une donnée par un thread, la modifie pour tout les autres. Dès lors, Durant une modification successive d'une variable, celle-ci se retrouve avec des valeurs aléatoires en fonction de l'ordonnancement des threads lors de l"exécution. Ce problème peut être réglé via des mutex, qui permettent de bloquer

l'accès à une ressource lorsqu'elle est utilisée par un thread. Cette méthode présente cependant des dangers qui seront expliqué en détail dans une autre section.

3.3. Différences de performances

Lors du test de performance entre la version exécute par les threads et celle par les processus, nous avons pu remarqué quelques différences notables. Veuillez notez que ces tests sont exécute sur le même id d'étudiant à savoir l'étudiant 1000 et 1M(sauf pour insert qui est sur la fin de la base de donnée), sur les même requêtes et avec 4 clients connecté au serveur pour la partie thread afin de simuler les 4 processus.

3.4. Sommaire

En résume, sous appuis de la différence entre le nombre d'avantages et d'inconvénients entre les 2 méthodes. Nous en concluant que les threads sont plus pertinent à utiliser que les processus dans le cadre de ce projet ne seraient ce que pour la différence au niveau de la mémoire utilisée et la facilité d'implémentation des threads. En effet, nous avons eu beaucoup moins de choses à penser et a faire attention lors de cette partie que lors de la partie 1.

4. Implémentation de la synchronisation entre threads

En effet, les threads présentent plusieurs problèmes. Le problème de concurrence dont nous avons déjà parlé et une possibilité de famine. Effectivement lorsqu'une requête en écriture est demandé aucune autre requête ne peut s'exécuter. Ainsi, si les requêtes en écriture arrivent en continu, aucune lecture ne sera possible et la requête sera en situation de famine car elle ne sera jamais traitée. Afin de régler ce double problème, l'utilisation des mutex est indispensable afin d'ordonnancer correctement les threads.

Voici le pseudo code de la manière dont est implémenter ces mutex:

```
mutex new_access = unlocked;
mutex write_access = unlocked;
mutex reader_registration = unlocked;
int readers_number = 0;

// if (write_request)
    lock(new_access);
    lock(write_access);
    unlock(new_access);
    // ... WRITE OPERATIONS
    unlock(write_access);
```

```
// if (Read_request)
    lock(new_access);
    lock(reader_registration);
    if (readers_number == 0)
        lock(write_access);
    readers_number++;
    unlock(new_access);
    unlock(reader_registration);
    // ... READ OPERATIONS
    lock(reader_registration);
    readers_number--;
    if (readers_number == 0)
        unlock(write_access);
    unlock(reader_registration);
```

Leur fonctionnement est celui-ci, nous initialisons 3 mutex, et un compteur pour connaître le nombre de requêtes de lecture en cours. Ensuite, nous séparons le cas d'une requête d'écriture ou de lecture puis dans les 2 cas on bloque l'arrivée d'autres requêtes avec leur acces en écriture ou en lecture suivant le type de requête. Le blocage en écriture avant la fin de l'exécution permet 2 choses, éviter que plusieurs écriture se fassent en même temps et empêcher qu'une lecture se fasse au même moment qu'une écriture.

En effet, si write_acces est bloqué et qu'une lecture est demandée, celle-ci sera mise en attente lors du premier if du coté des requêtes de lecture et pourra s'exécuter lors de la fin de cette écriture tout en bloquant l'accès aux autres écritures en rebloquant ce mutex.

Dès lors, le nombre de lecture en cours est incrémentée et l'on ouvre les mutex pour les accès a de nouvelles demandes en lecture étant donné que celle en écriture seront toujours bloquée par write_acces tant qu'elles n'ont pas toutes fini leur tâches. Après la décrémentation de readers_number, de nouveaux select sont acceptés et uniquement le dernier select exécuter autorisera les écritures de s'exécuter.

Cependant, la famine est évitée car étant donné que de nouvelles requêtes sont possibles, si une écriture est demandée, celle-ci bloquera la possiblitée de nouvelles lectures et ainsi la demande en écriture s'éxécutera lorsque les demandes en lectures déja en cours se terminent.

5. Problème bind

Un problème que nous avons eu lors de la mise en place du serveur est qu'il arrivait a se lier au socket une fois puis, il ne le pouvait plus durant plusieurs dizaines de secondes.

La première hypothèse fut que le file descriptor restait utilisé par l'os au cas où on en aurait encore besoin .

En réalité, il s'agissait du compilateur qui devait choisir entre la fonction bind des sockets et la fonction bind de la libraire standard(ces 2 fonctions ne font pas la même chose et n'ont pas le même retour de fonction). Étant donné qu'il ne savait pas lequel choisir, il autorisait la compilation mais le résultat n'était pas stable. Dès lors, il suffisait juste de mettre ::bind afin de régler le problème.

6. Proposition d'amélioration du projet

Une amélioration évidente du projet est la longueur des fonctions et les nombreuses répétitions . En effet, plusieurs fonctions ont entre 40 et 50 lignes en prenant en compte les brackets et les espaces. Dès lors, nous aurions pu essayer de les réduire avec des fonctions intermédiaires.

Cependant, ces fonctions concernant les querys et le problème de répétitions étant le même que lors de la première partie, nous avons préféré garder cette méthode en la rendant la plus lisible possible grâce aux espaces au lieu de prendre du temps supplémentaires afin de modifier cela alors que nous l'avons déjà fait durant la partie 1.

7. Conclusion

En conclusion, ce projet nous aura permis de cerner tout les aspects du cours, de les comprendre dans leur application, leurs avantages, leurs défauts et d'apprendre une méthode de programmation plus moderne qu'est le multithreading.