Université Libre de Bruxelles

INFO-F-201
Projet de programmation système

SmallDB

Étudiants :
Ethan Bebelambou
Abderrahmane Faher

Enseignants :
J. Goossens
C. Hernalsteen
H. Callebaut
N. Fernandez Lojo
A. Leponce

17 décembre 2022

Table des matières

1 Introduction

Smalldb est la version client/serveur de tinydb. Un programme serveur gère la base de données et traite les requêtes envoyées par le client qui, à son tour, reçoit le résultat et l'affiche. Cette fois les contraintes ont changé ; Nous allons utilisé du multi-threading au lieu d'un système multi-processus. Nous devons aussi gérer la synchronisation et permettre l'exclusion mutuelle à certains types de requêtes à travers l'implémentation de mutex/sémaphores.

2 Smalldb

Pour des raisons de performance et de compléxité, nous avons été obligés de faire des choix d'implémentations précis dans différentes situations.

2.1 Envoi et réception de messages

En utilisant des strings (valeurs de retour de la fonction *student_to_str*), la communication entre le client et le serveur ne marchait pas bien, de telle façon que la lecture des données était erronée. Ce qui nous a amené à utiliser la fonction **c_str** fournie par le langage C++, permettant la conversion d'un type string en char* pour résoudre ce problème.

2.2 Signaler la fin d'un message écrit

Un autre problème rencontré était de détecter la fin du message écrit. Cette fois, l'idée était d'envoyer un mot-clé "STOP" juste après la fin du message de telle façon, qu'il doit arrêter la lecture une fois tombé dessus.

2.3 Gestion de plusieurs clients

Nous avons été amené à implémenter un système multi-clients capable de se lancer au même temps et exécuter des tâches de la même façon. Pour se faire, nous avons décidé de stocker dans un vecteur, les file descriptor (valeur de retour de la fonction accept) de chaque client qui vient de se connecter pour les utiliser ultérieurement au niveau de la déconnexion par exemple. Par contre, si nous voulions stocker les file descriptor dans une seule variable, il n'y aura que le dernier client connecté qui pourra s'exécuter et les plus anciens seront bloqués.

2.4 Fermeture du serveur

Gérer la fermeture du serveur avec un signal n'est pas très compliqué, mais la difficulté résidait dans le cas où il fallait fermer le serveur alors que les clients sont encore connectés sur celui-ci. Pour palier à ce problème, nous avons fait en sorte qu'une fois que le serveur reçoit un signal indiquant qu'il doit s'arrêter, il envoie un message d'arrêt à chaque client puis ferme les sockets de communication pour chacun d'entre eux. Après cela, le serveur attend que toutes les requêtes en cours d'exécution se finissent puis coupent les connexions avec les clients avant de se fermer.

2.5 Affichage des adresses IP (partie bash)

La commande **sync** a pour but d'afficher les adresses IP de chaque client connecté sur le serveur. Cependant, la fonction utilisé pour récupérer les adresses renvoie un tableau d'informations dont certaines ne sont pas utiles (pour le sync). Pour récupérer seulement les adresses IP, il y a deux choses à faire :

1. ajouter le paramètre "*–no-header*" pour retirer les titres du tableau
2. à la fin de la commande, ajouter "*| awk '{print $5}'*" qui va afficher seulement les éléments dans cinquième colonne, qui sont les adresses IP des clients

```
1 #!/bin/bash
2 ss --no-header --ipv4 --tcp src :28772 | awk '{print $5}'
```

Listing 1 – Commande pour récupérer les adresses IP des clients

3 Synchronisations

Une requête peut agir de deux manières différentes sur la base de données :

— **récupérer** des éléments de la db

— **ajouter/retirer** des éléments de la db

Quand plusieurs d'entre elles sont réceptionnées par le serveur, il faut veiller à ne pas avoir une lecture ("select" par exemple) et une écriture ("delete" par exemple) qui sont exécutées en même temps pour éviter des erreurs de mémoire.

Le pseudo-code ci-dessous montre la stratégie utilisée pour faire en sorte que les requêtes qui vont modifier la db puissent être seules pendant leurs exécutions, et que les requêtes qui vont lire la db puissent s'exécuter en parallèle.

```
1  mutex new_access = unlocked;
2  mutex write_access = unlocked;
3  mutex reader_registration = unlocked;
4  int readers_c = 0;
5
6  // Writer
7  lock(new_access);
8  lock(write_access);
9  unlock(new_access);
10 // ... WRITE OPERATIONS
11 unlock(write_access);
12
13 // Reader
14 lock(new_access);
15 lock(reader_registration);
16 if readers_c == 0
17     lock(write_access);
18 readers_c++;
19 unlock(new_access);
20 unlock(reader_registration);
21 // ... READ OPERATIONS
22 lock(reader_registration);
23 readers_c--;
24 if readers_c == 0
```

```
25      unlock(write_access);
26  unlock(reader_registration);
```

Listing 2 – Synchronisation des requêtes lisant/modifiant la base de données

Pour avoir une exclusivité lors de l'exécution d'une requête, on utilise trois mutex et un entier :

- **new_access** : mutex qui "donne l'autorisation" d'agir sur la base de données (lire ou écrire)
- **write_access** : mutex qui donne l'autorisation d'écrire dans la db
- **reader_registration** : mutex qui donne l'autorisation de lire la db
- **reader_c** : contient le nombre de requêtes en lecture en cours d'exécution sur la base de données

Que ce soit pour une lecture ou une écriture, il faudra toujours bloquer le **new_access** en premier car c'est lui qui permet d'accéder à la base de données pour effectuer les opérations demandées par le client, et donc de bloquer soit *read_access*, soit *write_access*. Ne pas bloquer celui-ci implique qu'une autre requête manipule les mutex pour pouvoir intéragir avec la db.

REMARQUE : dans le code du projet, la déclaration, l'initialisation et l'utilisation des mutex ne se font pas au même endroit mais le principe reste le même.

3.1 Lecture de la db

Pour pouvoir lire les données dans la db, il faut impérativement qu'il n'y ait aucune écriture en cours. Par conséquent, dès qu'une requête nécessitant une lecture sera réceptionnée par le serveur, elle va bloquer **reader_registration** et **new_access** pous s'assurer qu'aucune autre requête ne vienne interférer son traitement, puis va essayer de bloquer **write_access**. Il y a trois scénarios possibles :

1. Cette requête est la première recpetionnée par le serveur donc elle bloque le mutex d'écriture pour empêcher qu'un "update" ne puisse se lancer par exemple.
2. Une requête de lecture est en cours d'exécution donc le mutex d'écriture est déjà bloqué. De ce fait, elle peut tout simplement commencer son traitement en parallèle.
3. Une requête d'écriture est en cours d'exécution, ce qui implique qu'elle ne peut pas commencer son traitement car **write_access** est déjà bloqué. Donc elle devra attendre la fin de la requête d'écriture pour pouvoir commencer son exécution.

3.2 Écriture dans la db

Dans ce cas de figure, la seule chose à faire est de bolquer le mutex de lecture car à lui seul, il va empêcher n'importe quel requête de pouvoir se lancer.

4 Processus vs Threads

Dans le cadre de notre projet, nous sommes amenés à utiliser du multi-threading au lieu du multi-processing, de telle sorte qu'un nouveau thread est créé à chaque connexion de client.

4.1 Multi-Processing

Le **multi-processing** est l'utilisation de plusieurs processus indépendants les uns des autres. Chacun possède son propre espace de mémoire (pas de mémoire partagée), comme ce qui était le cas pour le premier projet "tinydb", dans lequel chaque type de requête devait être gérer par un processus (4 au total). Pour des raisons de synchronisation, il fallait gérer aussi l'accès partagé en mémoire pour mettre à jour la base de données à chaque fois qu'un de ces processus y entraîne des changements. Ce système demande souvent des changements de contexte qui consiste à sauvegarder l'état environnemental du processus qui va être suspendu par l'ordonnanceur, charger des registres et charger/retirer des pages de mémoires afin de pouvoir continuer son exécution ultérieurement.
Ce changement de contexte est très coûteux en terme d'espace mémoire et de temps CPU.

4.2 Multi-threading

Le **multi-threading** consiste à avoir plusieurs threads dans le même processus. Ainsi, ils partagent tous le même espace d'adressage (heap précisément). Leur ordonnancement est aléatoire et ils s'exécutent tous en même temps, ce qui demande aussi un changement de contexte entre les différents threads mais il est généralement beaucoup plus rapide que celui entre des processus. La raison derrière est que tant qu'ils ont une mémoire partagée, l'échange des pages mémoires virtuelles, considéré comme l'une des opérations les plus coûteuse lors d'un changement de contexte, n'est pas nécessaire. Ce qui nous fait gagner du temps CPU et de l'espace mémoire.

En général, il vaut mieux utiliser du multi-threading pour économiser de l'espace mémoire surtout que l'ensemble de données gérées est important (contenu de la db). le multi-processing sera donc un bon choix dans le cas où l'espace mémoire utilisé est grand ou si les données manipulées ne sont pas si importantes en terme de taille.

5 Améliorations

5.1 Fermeture du serveur

Actuellement, lors de la fermeture du serveur, il est obligé d'attendre la terminaison des requêtes en cours d'exécution pour pouvoir s'arrêter. La meilleur manière de faire serait de directement stopper l'exécution des requêtes et couper les connexions avec les clients.

5.2 Commande Sync

La commande Sync, permet d'effectuer la synchronisation des données de la database en envoyant un signal SIGUSR1. Son handler fait en sorte de bloquer l'accès à la base données impliquant que si une requête est envoyée par le client, elle ne sera pas traitée directement, elle sera mise en attente jusqu'à ce que la sauvegarde de la base de données s'achève. Comme le temps d'attente est proportionnel à la taille de la database, plus elle est volumineuse, plus le temps d'attente est important. Et la situation devient de plus en plus grave avec l'augmentation du nombre de clients en attente.

L'une des améliorations possible est de faire en sorte de lancer la commande sync que si nous sommes sûrs qu'il y aura pas de requêtes envoyées par un ou plusieurs clients pour empêcher l'utilisation d'un espace mémoire supplémentaire que pour sauvegarder les requêtes en attente.

6 Conclusion

Après avoir fini ce projet, nous pouvons dire que pour la gestion de données, il est préférable d'utiliser le multi-threading car il ne faut plus se préoccuper du partage de mémoire.

L'ajout du serveur/client et de la synchronisation nous a permis de bien comprendre comment manipuler plus données en même temps, c'est-à-dire traiter certaines informations et en laisser d'autres en suspens, et communiquer sur plusieurs machines à l'aide de sockets.

Au final, pour pouvoir mener ce projet à bien, nous avons utilisé toutes nos compétences pour avoir un résultat correct.