Rapport d'OS

Andrius Ezerskis & Moïra Vanderslagmolen & Hasan Yildirim

November 3, 2022

1 Introduction

Il y a peu de temps, notre université a décidé qu'il était temps de changer notre base de données des étudiants afin d'améliorer l'accessibilité et le confort du système, ce qui permettra de fournir plus d'informations et une meilleure entrée à nos camarades étudiants. La base de données précédente ne répondait plus aux besoins depuis que le nombre d'étudiants a considérablement augmenté ces dernières années. Il était donc nécessaire que notre université s'occupe de ce problème de croissance de la base de données. Nous avons demandé ce projet pour gérer la création d'une nouvelle base de données. Notre groupe est composé de 3 étudiants qui ont joué des rôles différents dans ce projet mais en coopération. L'ancienne base de données donnait beaucoup d'échecs. Les répétitions de noms, les informations manquantes sur les étudiants, ou les comptes supprimés étaient les quelques exemples parmi tant d'autres survenus pendant l'utilisation active de la base de données. Notre projet a été divisé en 8 parties différentes. Les parties suivantes sont les parties de notre projet.

2 Stockage de la base de données

Afin de stocker les données de milliers d'étudiants, plusieurs choix se sont offerts à nous. Tout d'abord, nous avons utilisé std::map, comparable à un dictionnaire en python, dont les clés ont été l'identifiant de l'élève et les valeurs la struct student. Seulement, cela prenait beaucoup de place en mémoire et nous avons donc délaissé l'idée. Nous avons donc décidé de faire un tableau.

3 Fin d'une query

Pour signaler la fin d'une query, nous avons implémenté une message queue grâce à une pipe. Lorsqu'on envoie une query à un processus, nous incrémentons une variable globale operation in progress qui indique le nombre d'opérations en cours. Lorsque qu'un processus termine sa query, il écrit dans sa pipe "SUCCESS". Le processus principal va lire cette pipe et va décrémenter la variable globale au fur et à mesure. Le pipe est non-bloquant, ce qui permet au processus principal de continuer même si il ne lit rien dans la pipe. Lors d'une transaction, on limite operation in progress à 1.

4 Gestion de signaux

Plusieurs choix se sont offerts à nous afin de terminer un programme. Nous avons d'abord pensé à 4 pipes qui indiquent au programme principal la fin d'une requête, mais ce n'était pas optimal. Nous avons repris la message queue qui traite les query, et dans cette message queue, le processus principal écrit un message, "kill". Les processus enfants le liront après avoir terminé leur requêtes, et le kill arrêtera la boucle et exit le programme. Seulement, lorsque le processus enfant se termine, le processus parent n'est pas au courant de la mort de l'enfant. Nous avons donc eu beaucoup de processus zombies. Pour régler ce souci, nous avons tout simplement ignoré le signal SIGCHLD.

5 Checker l'identifiant

Pour voir si l'identifiant est déjà présent dans la base de données, nous avons pensé à mettre chaque étudiant dans son emplacement "correspondant" dans le tableau (étudiant id=500 va à l'emplacement db[500]). La recherche par id et le delete auraient été rapides, et on aurait pu rapidement savoir si un étudiant est déjà présent dans le tableau ou non. Seulement, si l'utilisateur introduit un id très grand, le stockage de la data base sera très grand, pour un seul étudiant. Nous avons donc essayé d'implémenter

une boucle for, qui vérifiait si l'id de l'étudiant n'était pas encore présent en itérant dans toute la base de données. Cela prenait beaucoup trop de temps. Nous avons donc décidé d'avoir un tableau trié d'étudiants. Pour cela, lorsque nous ajoutons un étudiant, nous vérifions si l'étudiant à l'emplacement du tableau -1 est plus petit que lui. Si il est plus petit que lui, nous l'ajoutons juste derrière. Sinon, nous décrémentons la variable au fur et à mesure, jusqu'à ce que l'étudiant aie un étudiant + petit que lui. Lorsqu'il trouve cet étudiant, il s'insère juste derrière lui et décale toute la database afin de s'insérer en plein milieu. Lors de l'update, nous avons remarqué que l'utilisateur maladroit pouvait par exemple écrire update fname=Mario set id=88. Pour s'assurer que cela n'arrive pas, nous vérifions si l'id n'est pas déjà présent dans la database dans la fonction update. Nous copions l'étudiant duquel on veut changer l'id, nous changeons l'id du nouvel étudiant et nous l'ajoutons à la database. Si l'id n'est déjà présent, nous ajoutons le nouvel étudiant grâce à la fonction db add, qui trie les étudiants en les ajoutant. Nous arrêtons immédiatement la boucle dans si l'id est déjà présent ou non pour des soucis de performance(deux étudiants ne peuvent pas avoir le même id, donc tous les étudiants s'appelant Mario ne peuvent pas avoir le nouvel id).

6 Classes

Nous avons mis la database_t et le query_result_t dans une classe, afin de rendre le code plus lisible. Par conséquent, nous avons implémenté un constructeur dans query_result_t, au lieu du query_result_init, pour que le code crée directement le query_result_t, sans devoir appeller l'initialisation à chaque création d'objet. Dans query result t, nous avons aussi ajouté log query, pour pouvoir mettre les attributs en privés. Nous n'avons pas mis student_t en classe, parce que les attributs d'un student sont souvent accédés par la classe query result t. et nous n'aurions donc pas su mettre les attributs en privés.

7 Améliorations possibles

7.1 std::map

Nous avions comme idée de stocker, dans un structure de données std::map (équivalent d'un dictionnaire en Python), les identifiants des étudiants en clés ainsi que leur indice de position dans la base de données en valeurs de ces clés.
Selon nous, cela aurait permis de faciliter et d'améliorer l'efficacité de la vérification de si un étudiant est déjà dans la base de données ou non.
En effet, lors de l'ajout d'un étudiant, nous aurions juste à vérifier si l'identifiant est présent dans le map pour éviter d'avoir des doublons dans la base de données.

7.2 Threads

Au lieu d'utiliser des processus, nous aurions pu utiliser des threads. Seulement, si le processus main crashe, alors tous les threads crashent aussi. L'avantage des processus est leur indépendance, ce qui les rend plus stable que des threads.

8 Process vs Thread

Tout d'abord, nous allons définir ce qu'est un processus et un thread, et finalement les avantages des processus.

Donc, un processus est l'exécution d'un programme qui permet d'effectuer les actions appropriées spécifiées dans un programme. Il peut être défini comme une unité d'exécution où s'exécute un programme. Le système d'exploitation vous aide à créer, planifier et terminer les processus utilisés par l'unité centrale. Les autres processus créés par le processus principal sont appelés processus enfants.

Ensuite, un thread est une unité d'exécution qui fait partie d'un processus. Un processus peut avoir plusieurs threads, tous exécutés en même temps. Il s'agit d'une unité d'exécution dans la programmation concurrente. Un thread est léger et peut être géré indépendamment par un planificateur. Il vous aide à améliorer les performances de l'application en utilisant le parallélisme.

Nous avons utilisé des processus pour réaliser notre projet, comme précisé dans les consignes. Les avantages des processus qui nous ont aidés à réaliser ce projet sont les suivants :

- La création de chaque processus nécessite des appels système distincts pour chaque processus.
- Il s'agit d'une entité d'exécution isolée qui ne partage pas les données et les informations.
- Les processus utilisent le mécanisme IPC(Inter-Process Communication) pour la communication qui augmente considérablement le nombre d'appels système.
- La gestion des processus nécessite plus d'appels système.
- Un processus a sa pile, son tas de mémoire avec la mémoire, et sa carte de données.