Rapport d'OS

Andrius Ezerskis & Moïra Vanderslagmolen & Hasan Yildirim

November 7, 2022

1 Introduction

La base de données des étudiants à l'ULB étant actuellement surchargée et lente, l'université nous a demandé à nous, informaticiens, de créer un nouveau système de stockage des étudiants plus efficace et plus rapide, pour permettre à l'ULB de continuer à accueillir de plus en plus d'étudiants chaque année. L'utilisateur peut effectuer plusieurs requêtes: il peut supprimer, rajouter, choisir et mettre à jour des étudiants. Lors de l'implémentation de ce projet, nous avons eu quelques difficultés auxquelles nous avons proposé plusieurs solutions. Dans les prochaines parties, nous décrirons donc nos choix d'implémentation et leur raisons, ainsi que les problèmes survenus.

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 avions utilisé std::map, comparable à un dictionnaire en Python, dont la clé était l'identifiant de l'élève et la valeur la structure student_t. Cela aurait permis de vérifier facilement et rapidement si l'id d'un étudiant est déjà présent dans la base de données (la fonction find de std::map est en log(n)). Seulement, cela prenait beaucoup de place en mémoire et ce n'était pas évident à mettre en place, nous avons donc délaissé l'idée.

Nous avons ensuite pensé à utiliser des pointeurs, chaque étudiant ayant son student.next et son student.precedent. Le problème de cette implémentation était que la structure de l'étudiant faisait exactement 256 bytes. Ajouter un next.student et un precedent.student a modifié la taille de la structure de student, ce qui n'allait pas avec le fichier students.bin, qui fait exactement 256 bytes par ligne. Nous avons donc décidé de faire un tableau d'étudiants, que nous avons triés pour des raisons que nous allons voir dans la section "Checker l'identifiant".

3 Fin d'une requête et transaction

Pour signaler la fin d'une requête, 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 donc implémenté une message queue grâce à une pipe. Lorsqu'on envoie une requête 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 requête, il écrit dans sa pipe "SUCCESS". Le processus principal va lire cette pipe et décrémenter la variable globale au fur et à mesure.

Le problème majeur de cette implémentation est que la pipe est bloquante, c'est-à-dire que le processus va s'arrêter jusqu'à ce qu'il lise un message écrit dans sa pipe. Nous avons donc rendu le pipe non-bloquant, ce qui permet au processus principal de poursuivre son exécution même lorsqu'il ne lit rien dans la pipe. Lors d'une transaction, nous limitons operation_in_progress à 1.

4 Gestion de signaux et fin de processus

Pour terminer notre programme, nous avons repris la message queue qui traite les requêtes, et dans cette message queue, le processus principal écrit un message, "kill". Les processus enfants liront le message "kill" après avoir terminé leurs requêtes, et le kill arrêtera la boucle de chaque processus enfant et mettra fin au programme. Seulement, lorsque le processus enfant se termine, le processus parent n'est pas mis 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. Lorsque l'enfant termine son exécution, un signal SIGCHLD est envoyé au parent. Si on ignore ce signal, le système l'ignorera aussi et le processus enfant sera supprimé de la table des processus.

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 (par exemple, l'étudiant id=500 va à l'emplacement db->data[500]). La recherche par id et le delete auraient été rapides, et nous aurions pu rapidement savoir si un étudiant est déjà présent dans la base de données. Seulement, si l'utilisateur introduit un id très grand, le stockage de la base de données serait 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 déjà 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. S'il est plus petit que lui, nous l'ajoutons juste après. Sinon, nous décrémentons la variable au fur et à mesure, jusqu'à ce que l'étudiant à insérer aie un étudiant plus petit que lui. Lorsqu'il trouve cet étudiant, il s'insère juste après lui et décale toute la base de données située après lui.

Lors de l'update, nous avons remarqué que l'utilisateur maladroit pouvait introduire la requête "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 base de données dans la fonction update. Nous copions l'étudiant auquel nous voulons changer l'id, nous changeons l'id du nouvel étudiant et nous l'ajoutons à la base de données. Si l'id n'est pas déjà présent dans la base de données, nous ajoutons le nouvel étudiant grâce à la fonction db_add, qui trie les étudiants avant de les ajouter. Nous arrêtons immédiatement la boucle dans db_add si l'id est déjà présent ou non pour des soucis de performance (deux étudiants ne peuvent pas avoir le même id, seulement un étudiant s'appelant Mario peut avoir le nouvel id).

6 Classes

Nous avons mis la base de database_t et l'objet requête 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 la requête 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 privé. 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 que nous n'avons pas beaucoup de méthodes pour student_t.

7 Gestion en mémoire partagée

Nous avons d'abord seulement partagé la mémoire avec db-¿data, mais nous avons vite remarqué que les processus ne partagaient pas la même base de données. Nous avons donc partagé la mémoire aussi avec

l'objet database, à l'aide de mmap. Malheureusement, lorsque nous effectuions les tests, ils affichaient une erreur. En effet, à chaque fois que le processus enfant s'occupant de insert augmentait la mémoire, les autres processus n'y avaient plus accès. Nous avons donc essayé d'augmenter la mémoire uniquement avec le processus principal, en utilisant le db-¿upsize avant de faire le processus insert, ce qui ne fonctionnait toujours pas. Nous avons alors ajouté un file descriptor au mmap. Lorsqu'on augmente la taille du fichier et que sa place en mémoire change, le programme nous donnait un segmentation fault. Effectivement, les processus enfants ne remappaient pas la nouvelle mémoire crée, et par conséquent, il n'avaient plus la permission d'accéder à la nouvelle mémoire. Suite à ça, nous avons écrit une nouvelle fonction qui remappe chaque processus enfant lors d'un changement lié à la mémoire.

8 Conclusion

En conclusion, nous avons beaucoup appris avec ce projet, par exemple sur la communication inter-processus ou bien la gestion de signaux et nous avons réussi à implémenter une base de données plus efficace que la précédente. Nous permettons ainsi à l'université d'accueillir de plus en plus d'étudiants chaque année, sans se soucier de la gestion de ceux-ci par la base de données. Le code est également maintenable, permettant aux informaticiens de l'ULB de résoudre des bugs et problèmes éventuels facilement.