INFO-F201 : Projet 2 système exploitation

Oudahya Ismaïl

Table des matières

1.Introduction

Pour ce projet de système d'exploitation , on nous demande d'implémenter un projet de base de données qui a été laissé à l'abandon. Cette base de données doit être capable de lancer plusieurs requêtes simultanément notamment sélectionner , supprimer , ajouter et modifier. Cette base de données doit être programmer avec le langage C tout en utilisant la librairie standard et l'API POSIX. Dans ce projet j'ai notamment utilisé les librairies standards et les API POSIX qui sont <sys/time.h> , <pthread.h> , <time.h> et <signal.h>.

- Concernant <sys/time.h> , il me permet de calculer la durée de chaque requête et sont stocker dans les logs
- <pthread.h> permet d'utiliser les threads mais on en parlera un peu plus tard
- <time.h> permet de gérer tout ce qui concerne le temps ici , cela m'a servi à utiliser dans la struct de la base de données.
- <signal.h> m'a permis de jouer avec le ctrl-c pour pouvoir arrêter le programme tout en sauvegardant la base de données.

L'utilisation de l'allocation dynamique était aussi importante pour se projet. A chaque allocation dynamique il faut un free() pour libérer la mémoire et avec valgrind ont peut détecter quel mémoire à été vider et celle qui se retrouve encore en mémoire.

On utilise une liste de pointeur de struct database_t pour récupérer toutes les informations concernant la base de données. On pouvait aussi utiliser les linked list mais j'ai préféré utiliser une liste de pointeur.

En termes de temps ici le delete est le plus lent car après avoir supprimer un étudiant , il doit aussi arranger la liste de pointeur de database_t pour ne pas avoir de trou et avoir des erreurs de parcours. Et le plus rapide reste l'ajout qui est insert car il lui suffit de charger la base de données pour ensuite le rajouter à la fin.

INFO-F201 : Projet 2 système exploitation

Oudahya Ismaïl

2.Architecture

Tout d'abord , le code se compose de quartes fichiers , la db.c et son header qui gère toute la base de données donc insert,delete,update,load,select. Il utilise de l'allocation dynamique pour ajouter de la mémoire pour la base de données en revanche malgré l'implémentation de thread la commande delete prend beaucoup de temps d'exécution lorsqu'on la lance sur une base de données à plusieurs milliers d'étudiants, une solution serait d'utiliser une linked list.

Ensuite , nous avons le fichier parsing.c et son header qui lui gère à séparer le fichier texte entrée par un fichier donc par stdin ou par la commande émise par l'utilisateur dans le terminal. Le parsing servira principalement pour récupérer la commande donnée et de séparer les éléments essentiels.

Le fichier student qui possède la structure de chaque étudiants c'est à dire le nom ,prénom ,id ,section et date d'anniversaire.

Pour finir, la main.c qui se compose de tous les appels de fonction et de création de thread ainsi que les entrées en commande par l'utilisateur et par une entrée de fichier grâce à un fgets prenant en compte un buffer et stdin.

L'utilisateur à deux choix , soit il lance le programme en donnant aucune database et donc le programme va crée une base de données , et l'autre choix est quand le l'utilisateur entre la base de données.

2.1 Les signaux

Pour le signal , le ctrl-v arrête bien le fgets pour arrêter le parcours de commande , concernant le ctrl-c il n'était pas possible d'arrêter le fgets avec un ctrl-c donc j'ai préféré utiliser une variable globale volatile , j'ai utilisé une variable volatile pour prévenir le compilateur que cette variable pouvait être modifié par une commande extérieure du programme donc notamment, ici le ctrl-c.

Le handler récupère le signal donné par ctrl-c et modifie cette variable volatile qui aura un impact sur le fgets car j'ai notamment ajouté à la condition while que le ctrl-c n'a pas modifié cette variable.

En revanche pour pouvoir la validé comme étant un ctrl-c il faut la donner comme si c'était un paramètre donc l'utilisation est de lancer ctrl-c et d'appuyer sur entrées.

2.2 Log et gestion du temps

Pour le log on part du fait que le dossier logs existe , pour gérer la création du fichier log et l'écriture dans le fichier , j'ai décidé de crée deux fichiers dont un qui est temporaire , il me permet d'écrire dans le fichier temporaire les étudiants trouvé grâce aux commandes pris par le fgets() et de l'écrire après avoir récupérer le temps d'exécution de l'appel de la fonction pour trouver ce que l'utilisateur demande.

Concernant la gestion du temps j'ai utilisé clock_gettime() pour avoir une valeur précise de temps entre l'action d'appel de fonction que fait le CPU. J'ai du changé pour que ce soit en millisecondes.

3.Threads

Après avoir réfléchit à l'implémentation de threads , j'ai réalisé qu'en recevant les requêtes une par une via un fichier ou l'entrée d'un utilisateur on pouvait que faire des threads séquentiels. Et que cela n'apportait pas une grande différentes quant à la rapidité d'exécution.

J'ai aussi divisé la database_t en 4 parties pour pouvoir l'implémenter pour la commande select , celle-ci aura 4 threads qui lanceront chacun leurs threads avec différents indices de la database_t

Taille thread 1	(taille database_t / 4) *1
Taille thread 2	(taille database_t / 4) *1
Taille thread 3	(taille database_t / 4) *1
Taille thread 4	(taille database_t/4)*4 + modulo taille_databse_t

Cependant cela n'a pas eu beaucoup d'impact dans le temps donc j'ai décidé de rester sur du séquentielle.

3.1 Amélioration possible

Cependant si on part du principe que l'ont reçois toutes les requêtes en une fois on peut utiliser les threads en parallèle mais on devra faire attention aux commandes qui pourrait s'entremêler.

Une des solutions que l'ont pourrait utiliser se sont les mutex , quand par exemple on a une commande delete en même temps qu'une commande update on pourrait lock une des deux pour pouvoir modifier ou supprimer car si l'update tente de modifier la database_t et qu'en plein milieu de son exécution le delete supprime on aurait une erreur et ce n'est pas ce que l'ont recherche ici donc un lock d'une des deux serait envisageable.

4.Conclusion et problème rencontrer

Pour conclure , les difficultés rencontrer étaient plutôt les erreur de segmentation , les double free or corruption**.** Mais grâce au debugger Valgrind j'ai réussi à retrouver mes erreurs , c'était principalement du au fait d'oublier de faire des free() ou d'essayer d'accéder à une mémoire qui n'existe pas. J'ai aussi rencontré quelque souci avec strcpy qui parfois remplaçait une lettre par une phrase par exemple : section=pharma fname=pharmacien du coup la section devenait pharmapharmacien , donc j'ai décidé de ne pas utiliser strcpy.

Le projet n'est pas trop compliqué, c'est juste la découverte du langage C qui nous entrave comme par exemple pour comparer deux strings , pouvoir séparer un string. Toute utilisation de chaine de caractère se voit se complexifié un peu trop, donc l'utilisateur est obligé de s'occuper de petits détails comme ceux-là. Et sans oublier une gestion d'erreur très simple.