LSINF 1252 - Système informatique - 2019

Projet Password cracker

Misonne Thibaud 5314-17-00
Pironnet Justine 6788-17-00

Remise le : 10 mai 2019

1 Introduction

Il y a quelques semaines, il nous a été demandé de créer un programme *cracker*. Nous devions donc implémenter une architecture résolvant un problème de calcul intensif grâce à la parallélisation. Cette parallélisation peut se faire via des *threads*.
Ce rapport contient donc plusieurs points importants de notre projet afin de bien comprendre ce que nous avons fait et pourquoi nous l'avons fait. Il est donc composé en 4 sections. La première consiste en une explication de *l'architecture à haut niveau*. La deuxième partie contient nos *choix de conceptions* qui nous paraissaient important d'expliquer. La troisième section comporte nos *stratégie de tests* et enfin, une *évaluation quantitative*.

2 Architecture haut-niveau

Notre architecture est "divisée" en trois fonctions de base. La fonction *lire()* qui a pour but de lire dans les différents fichiers passés en argument, les mots de passe sous forme binaire. Ensuite il y aura la fonction *decrypteur()* dont le rôle est d'appliquer la fonction *reversehash* (déjà implémentée pour nous) afin de trouver un inverse ou non à ce mot de passe. Et enfin la troisième fonction de base est la fonction *ecrire()* qui va écrire dans un fichier les candidats trouvés en fonction des critères de sélection.

Pour pouvoir paralléliser ces trois fonctions et utiliser les *threads*, nous avons utilisé le problème du producteurs/consommateurs. En effet, nous recevons donc en entrée plusieurs fichiers, qu'on va lire avec un thread de lecture. Ce sera donc le premier producteur, qui placera dans le *buffer1* de 3 places, les mots de passe (sous forme de hash) lus.
Ensuite, les consommateurs 1, qui sont les threads de **calculs**. Ils prendront donc dans le *buffer1* les différents mots de passe et leur appliqueront la fonction *reversehash*. Si cette fonction a trouvé un inverse au mot de passe, alors il est placé dans le *buffer2* de 5 places. Ils seront donc aussi les producteurs 2.
Et enfin, dans la dernière fonction, qui représente les consommateurs 2, les *threads* iront chercher dans le *buffer2* les différents mots de passe et regarderont s'il est candidat ou non. S'il l'est, il est écrit, par défaut, dans la console, mais peut également être écrit dans un fichier de sortie.
Les différentes threads communiquent donc grâce au *buffer1* et au *buffer2*.

Afin de garantir la "sécurité" du programme et que plusieurs *threads* ne fassent pas la même commande en même temps, nous avons utilisé des sémaphores et des mutex aux endroits critiques.

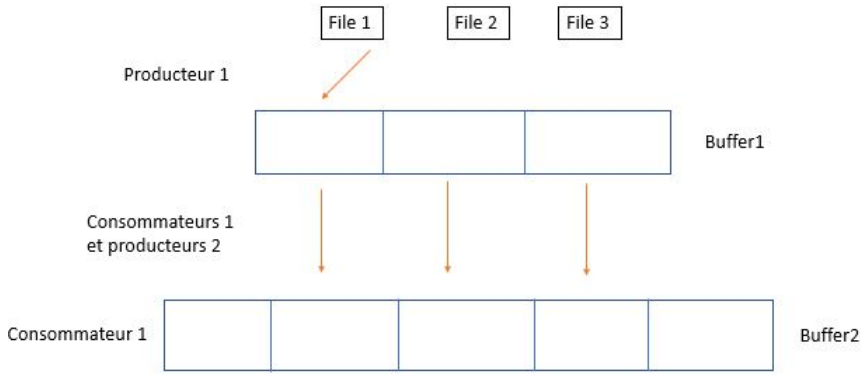

FIGURE 1 – Architecture

3 Choix de conception

3.1 Variables *debut1, fin1, debut2, fin2*

Afin que buffer1 et buffer2 (nos tableaux permettant aux différents threads de communiquer) fonctionnent correctement, il est nécessaire que les producteurs et les consommateurs sachent constamment à quelle place du tableau ils peuvent insérer et prendre un élément. Pour ce faire, nous avons déclaré deux entiers par tableau,

soit quatre entiers au total. Pour un tableau, les deux variables sont initialisées à "0". Lorsque le producteur insère son élément, il le fera en position "debut1" et celui-ci avancera d'une case dans le tableau. Et lorsque le consommateur y lit un élément, il le fera en position "fin1" qui se avancera également d'une case dans le tableau. De cette manière, les deux variables représentent le début et la fin (d'où leur nom) d'un espace libre dans le tableau, et permettent donc à elles seules de constamment connaître l'état de celui-ci.

3.2 Structure *list*

Afin de stocker les candidats avant leur écriture dans la console ou le fichier de sortie, nous avons créé deux structures permettant de les stockers de manière dynamique. La première, *node*, représente des noeuds pointant chacun vers, au choix, un autre noeud ou NULL, et stockant les mots de passe candidats. La deuxième structure : *list*, possède un élement *first* représentant le début de la liste. Cela nous semblait la manière la plus simple à implémenter car lorsque la variable *test*[1] est plus grande que la variable *max*[2], il est facile de "reset" la liste en attribuant au noeud *first* le nouveau noeud dans lequel est stocké le nouveau premier candidat avec un nombre de consonnes/voyelles plus important.

3.3 Les boucles *while*

Dans le but de pouvoir lire autant de fichiers que demandés lors de l'exécution du programme, nous avons du mettre à jour les conditions des boucles *while* dans la fonction *decrypteur()* et *ecrire()*. En effet, celles-ci sont maintenant des boucles infinies qui vont "*break*" lorsque plusieurs conditions seront réunies.

- **Pour la fonction *decrypteur***
    1. La variable *lu* doit être <= 0
    2. *Début1* doit être égal à *fin1* (ce qui indique que le buffer est vide)
- **Pour la fonction *ecrire***
    1. La variable *decrypte* doit être égale à -1

Lorsque la fonction *lire* est finie, *lu* est décrémenté du nombre de threads de lecture pour que *lu* arrive à 0. La fonction *decrypteur*, quant à elle, va fonctionner grâce aux threads de calculs.Tous les threads rentrent dans le boucle infini, et dans cette boucle, 2 options sont possibles : soit le buffer est vide et il n'y a plus de mot de passe à reversehash, soit le buffer n'est pas vide et les threads doivent encore travailler.

1. Si le buffer n'est pas vide : après que le thread aie fait la fonction *reversehash*, un *if* va regarder si le buffer est vide avec *debut1 et fin1* et si *lu* est plus petit ou égal à 0 (ce qui veut donc dire que *lire* a fini sa fonction). Si c'est le cas, *lu* est décrémenté, et le thread passe dans la condition d'après qui dit que si *lu* est **strictement** plus petit que 0, le thread s'arrête avec la commande *break*. Si le thread ne passe pas dans la première instruction conditionnelle, le thread recommence la boucle.
2. Si le buffer est vide : *lu* est décrémenté et le thread sort de la boucle grâce à *break*.

Lorsque *lu* arrive à la valeur *-tvalue* qui correspond au nombre de threads de calculs, la variable *decrypte* passe à -1, ce qui stop la boucle *ecrire* grâce à une instruction *if* et un *break*.

4 Stratégie de tests

Afin de tester notre travail, nous avons commencé par tester les deux sous-fonctions que nous utilisons. Pour tester la fonction *count*, nous l'avons écrite dans un fichier *count.c* et exécuté dans un fichier *count.o*. Il est alors possible d'exécuter count.o en lui demandant ce compter les consonnes d'un mot avec l'option *-c* ou les voyelles par défaut, et d'imprimer le résultat dans la console. Nous constatons alors, que pour plusieurs mots, elle fonctionne dans les deux cas. Nous avons également testé notre deuxième sous fonction, *check*, permettant de tester si le buffer intermédiaire de *lire* et *décrypter* est vide. Nous l'avons donc écrite dans un fichier *check.c* et exécuté dans un fichier *check.o*. Automatiquement, il va alors appeler la fonction une première fois avec un tableau vide et ensuite avec un tableau rempli et imprimer le résultat.

1. qui sert à compter le nombre de consonnes/voyelles en fonction du critère de sélection
2. qui représente le maximum de consonnes/voyelles trouvées jusqu'à présent

Nous avons ensuite testé le programme complet, en l'appelant dans différents cas. Il s'exécutera alors avec des nombres de threads différents, parfois plus grand et parfois plus petit que le nombre de mot de passes à lire dans le fichier, il comptera parfois les voyelles, parfois les consonnes, écrira dans la console ou dans un fichier de sortie grâce à l'option -o, de manière à vérifier tous les cas. Nous pouvons alors vérifier que tous les outputs sont corrects. Pendant que nous codions, nous imprimions les différents mots de passe lus ainsi que différents éléments nous permettant de voir que tout se passait correctement. Bien sûr, cela a été supprimé puisque ce n'était pas demandé.

5 Évaluation quantitative

Nous avons fait plusieurs tests de "vitesse" afin de bien voir l'utilité de la parallélisation de plusieurs threads. Ces tests ont été réalisés sur un ordinateur de la salle Intel. Vous pouvez retrouver ci-dessous plusieurs tableaux. Le premier correspond à la vitesse en fonction du nombre de threads de calculs, avec le fichier contenant 1000 mots de passe de maximum 4 lettres avec le critère de sélection voyelles.

Nombre de threads	1	2	3	5	10	50	100	500	1000
Temps en secondes	316	157	105	86	85	85	85	85	85

Grâce à ce tableau nous pouvons donc voir très clairement qu'à partir de 5/10 threads de calcul, le programme ne sera pas plus rapide. Nous avons donc essayé d'augmenter la taille des deux buffers, mais le temps restait le même à une seconde près.