LSINF1252

Password Cracker: Rapport Final

Groupe 8

Marques Mathias (13181700) - Grognard Simon (37811700)

1 Avril 2019

1 Introduction

Dans le cadre du cours de Système informatique, il nous a été demandé de créer un programme en C capable de cracker une liste de mots de passes. Ces derniers seront hashés au préalable et écrit dans des fichiers d'input. Notre objectif est de décryptés ces mots de passe grâce à une fonction données et d'écrire sur la sortie standard ceux avec la plus grande occurrence de voyelles ou de consonnes selon la valeur placée en argument(voyelles par défaut). Un deuxième argument peut être placé, il s'agit du nombre de threads, il sera initialisé à 1 par défaut (si on ne place pas d'argument) sinon on nombre de threads souhaités. Les autres arguments seront les fichiers d'input et celui d'output (par défaut notre programme affiche les mots de passe sur la sortie standard si aucun fichier output n'a été placé en argument).

2 Architecture

Lors de de l'exécution du code, ce dernier va chercher quels arguments sont placés dans l'exécutable. Il regardera si le premier argument est '-c' ou '-t nbThread' ou encore '-o filename' tout d'abord l'argument '-c' nous indiquera qu'il faut compter les consonnes et non les voyelles. Ensuite l'argument '-t nbThread' nous indique le nombre de threads que nous pouvons allouer au décryptage des mots de passe. Et pour finir l'argument '-o' nous donne un fichier d'output pour écrire nos mot de passe. Une fois que nous avons récupéré toute ses informations nous pouvons lancer nos divers threads. Tout d'abord le producteur va prendre en argument le nom d'un fichier et va lire son contenu par groupe de 32 bytes pour ensuite les placer un par un dans un buffer (de type u_int8_t*). Pendant ce temps les threads consommateur vont prendre les mots de passe cryptés dans ce buffer et vont appeler la fonction reversehash qui a été donnée dans le cadre de ce projet (cette fonction permet de retourner le mot de passe non crypté depuis son mot de passe crypté). Afin de savoir à quel endroit dans le buffer le consommateur doit prendre le mot de passe crypté nous avons une variable qui nous indique le dernier élément ajouté par le producteur, le consommateur prend alors ce dernier élément. Bien évidemment le buffer et la place dans ce dernier sont protégé par des mutex et des sémaphores(2 mutex (le premier pour la protection du buffer et le second pour la stack) et 2 sémaphore). Ces derniers sont disposés de manière à régler le problème des producteurs et consommateurs. Mais l'appelle de la fonction reversehash n'est pas l'unique fonction que nous donnons à nos consommateurs, ils sont aussi chargés de placer nos mots de passe dans une stack (les threads consommateurs ne place pas leur mot de passe décrypté dans cette stack uniquement lorsque le nombre de consonnes ou voyelle est supérieur ou égal au nombre de consonne ou voyelle du dernier mot de passe dans la stack) ainsi une fois tous les threads consommateurs terminés nous pouvons retourner les mots de passe ayant la plus grande occurrence de voyelles ou consonnes. Ce choix de conception est un peu particulier et sera controversé dans la partie suivante. Mais viens alors la question de savoir les conditions d'arrêt de nos différents threads. Pour les producteurs c'est assez trivial il suffit de terminer le fichier. Par contre pour les consommateurs c'est un peu plus complexe, il faut que tous les threads producteurs soit terminés (=fin de tous les fichiers) et que notre buffer soit vide. Pour la deuxième condition aucun problème, pour la première en revanche c'est un peu plus compliqué. Nous avons finalement décidé de donner une certaine valeur à un entier une fois que tous les threads producteurs se sont join

(pthread_join) dans la main (Mais cette solution pose quelques problèmes que nous expliqueront dans la section suivante).

3 Choix de conceptions et implémentations

Les choix de conceptions qui peuvent nous distinguer par rapport aux autres sont principalement l'utilisation de la stack. Nous remplissons cette dernière à l'aide de nos threads consommateurs. Ce qui nous semblait à première vue une excellente idée car nous parallélisons correctement le problème et nous ne conservons pas les mots de passe qui ne nous intéressent plus (dans une certaine mesure, nous expliquerons le fonctionnement de notre stack par la suite). Mais après de nombreuses discussions avec les assistants et étudiants nous nous sommes rendu compte de l'utilité de séparer le décryptage des mots de passe et leur stockage dans la stack, nous avons alors décidé de séparer ces 2 étapes mais malgré de nombreux essais ces derniers sont restés infructueux. Nous sommes donc resté sur notre idée de départ. Ce qui malheureusement rend notre programme un peu plus lent que ceux qui ont réussi à scinder ces 2 étapes. Parlons maintenant de nos conditions d'arrêt des consommateurs comme indiqué ci-dessus. Nous avons une variable qui indique la fin de la lecture des fichiers et nous vidons le buffer, une fois que ces 2 conditions sont remplies nous quittons consommateur. Le problème que nous rencontrons est que cette méthode se joue sur la vitesse à laquelle la variable qui indique que tous les fichiers ont été lu change de valeurs assez rapidement car si elle se met à jour après que le placetab (place du mot de passe crypté dans le buffer) soit à 0 alors certains threads consommateur vont rester bloqué à cause des sémaphores. Même avec ce petit jeu sur la vitesse nous ne rencontrons jamais ce problème car notre fonction consommateur est bien plus lente que producteur ce qui permet de donner la bonne valeur et ainsi fermer correctement nos consommateurs. Nous restons tout de même bien conscient de la précarité de cette solution. Malgré de nombreuses tentatives et discussions avec assistant et étudiants nous ne sommes pas parvenus à trouver une meilleure condition d'arrêt. Une autre limite que notre code rencontre est lorsque nous le lançons avec plus de threads que de mots de passe dans le fichier. Ce qui nous montre une fois de plus les limites de nos conditions d'arrêt. Un point important de notre code est la stack dans laquelle nous stockons les mots de passe. Cette dernière est construite de 3 éléments : le premier est le mot de passe en lui-même, le deuxième est le nombre de voyelles ou consonnes que possède ce mot de passe et le dernier élément est un pointeur vers le noeud suivant. Nous ajoutons un noeud à cette stack seulement si le nombre de consonnes ou voyelles est au moins égal au nombre de consonnes ou voyelles du mot de passe se trouvant en haut de la stack. L'inconvénient de cette dernière est que nous gardons des mots de passe qui ne nous seront pas utiles. Mais en contrepartie nous ne devons pop la stack qu'une seule fois ainsi nous diminuons la perte de temps liée au pop de la stack dans le programme. Ainsi nous pouvons pop cette stack à la fin de notre programme après que ce dernier nous ai affiché les mots de passe.

4 Stratégie de test

Comme notre programme utilise diverses fonctions il a fallu les implémenter une à une et les tester au fur et à mesure pour ce faire nous avons écrit ces fonctions dans d'autre fichiers puis tester à l'aide de la fonction "printf" toutes les valeurs qu'elles retournaient. Ensuite nous les avons ajouté une à une dans le fichier cracker.c et une fois que celle-ci marchait nous avons rajouté les threads dans le programme. Ainsi nous avons pu tester tous les résultats intermédiaires. Il ne manque plus qu'à vérifier que notre programme sorte les bons outputs ce que nous avons fait avec le fichier 02_6c_5.bin.

En guise de test pour notre programme final, nous avons créé un Makefile avec plusieurs tests à l'intérieur. Nous récupérons les fichiers donnés dans le cadre du cours (cfr Exemple d'input pour le problème). Nous créons 4 cibles différentes pour chaque test :

— tests1 : cracker tests/02_6c_5.bin tests/01_4c_1k.bin
./cracker -t 100 tests/02_6c_5.bin tests/01_4c_1k.bin

— tests2 : cracker tests/02_6c_5.bin
./cracker -t 2 tests/02_6c_5.bin

— tests3 : cracker tests/01_4c_1k.bin
./cracker -t 100 -c tests/01_4c_1k.bin

— tests4 : cracker tests/01_4c_1k.bin
./cracker -t 100 -o tests/fichierOut.txt tests/01_4c_1k.bin

Pour exécuter ces tests, nous pouvons faire la commande 'make tests' (tests : tests1 tests2 tests2 tests3 tests4) ou alors 'make tests1', 'make tests2', 'make test3', 'make tests4'. En focntion du test, il écrira les mot de passes soit sur la sortie standard ou dans le fichier 'fichierOut.txt'.

5 Evaluation quantitative

Pour l'évaluation quantitative du programme, nous avons fait des graphes des deux premiers tests (test1 et test2) correspondant au temps d'exécution en fonction du nombre de threads. L'axe des abscisses correspond au nombre de threads et l'axe des ordonnées correspond au temps d'exécution en minutes.secondes.

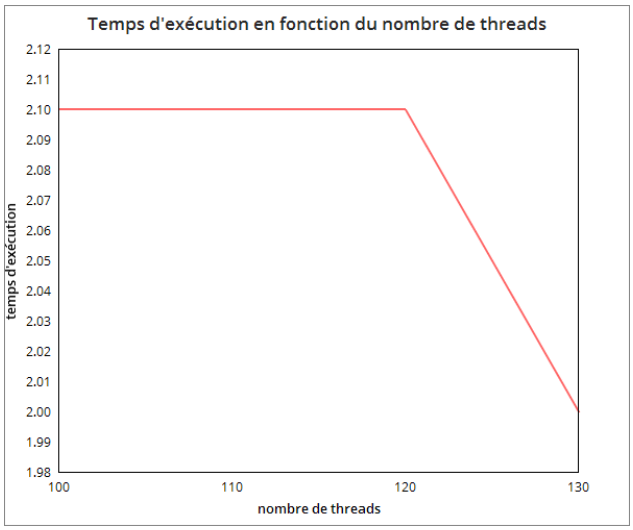

FIGURE 1 – Graphique du temps d'exécution en fonction du nombre de threads pour le test1

Pour le premier graphe, il s'agit du tests1, on peut remarquer qu'en augmentant de 10 threads à chaque exécution, notre programme gardait le même temps d'exécution. Sauf quand on augmente le nombre de threads à 130, le programme s'exécute 2 secondes plus rapidement.

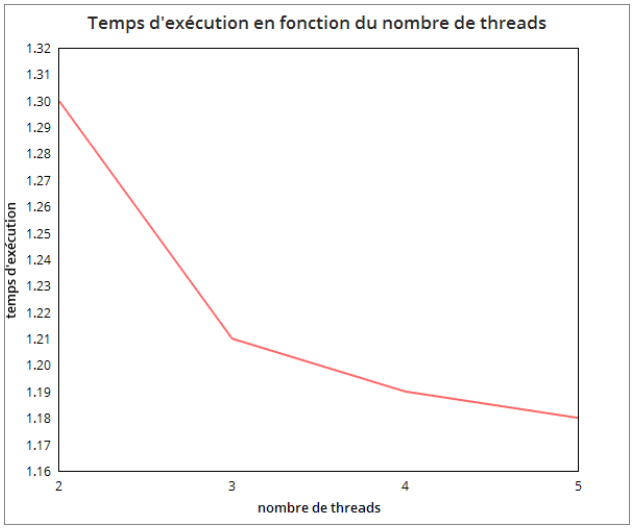

FIGURE 2 – Graphique du temps d'exécution en fonction du nombre de threads pour le test2

Pour le tests2, on remarque mieux la différence du temps d'exécution même si la différence de temps restent minime. Pour 2 threads il lui faut 1 minute 30 secondes pour trouver les deux mots de passes avec la plus grande occurrence de voyelles. Tandis qu'avec 5 threads, il lui faut plus qu'une minute et 18 secondes pour les trouver.

6 Conclusion

Pour conclure ce projet, nous avons appris à coder un programme avec des threads qui permet d'augmenter l'exécution du code comme vous pouvez le voir sur les graphiques de l'évaluation quantitative. Nous savons que notre code n'est pas encore très optimisé à cause de notre mauvaise gestion du timing. La fonction consommateur pourrait être plus efficace si