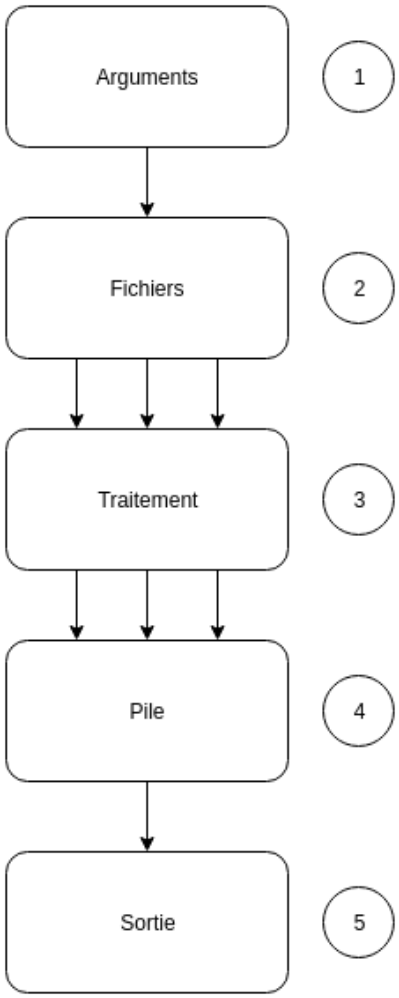
Systèmes Informatiques : Projet Threads

Clément Chardon Alexandre Serrao

1 Introduction

Dans le cadre du cours de Systèmes Informatiques, il nous a été demandé de réaliser un programme complet en language C dont le but était d'inverser les hash de nombreux mots de passe hashés présents dans un fichier en entrée et d'en imprimer un certain nombre de candidats en sortie. Le but de ce projet étant de nous familiariser avec la parallélisation d'un programme, la fonction *reversehash()* qui nous était fournie pour réaliser notre projet était une fonction très lourde, susceptible de bloquer inutilement l'exécution du programme pendant un long moment. Nous devions donc utiliser des threads pour paralléliser le programme un maximum, tout en gérant les éventuels problèmes d'exclusion mutuelle qu'ils causent.

2 Architecture



1. La toute première partie de notre code contient la gestion des arguments. Ceux-ci définissent les options avec lesquelles le programme va se dérouler. Toutes les possibilités d'argument proposées dans l'énoncé du projet sont prises en compte par notre programme.
2. Ensuite, nous devons gérer les fichiers binaires en entrée. Tous les fichiers terminés par ".bin" dans les arguments sont considérés comme des fichiers à ouvrir et, si leur format ne convient pas, une erreur sera renvoyé et le programme s'arrêtera. Dans notre code, nous utilisons un tableau contenant le nom de chaque fichier à ouvrir. Une fois le tableau entièrement parcouru, le programme passe à la phase (5).
3. La parallélisation commence ici. Nous avons construit notre programme de manière à écrire un seul et même procédé à suivre pour tous les threads utilisés pour ce programme. Chaque thread lit un hash sur le fichier binaire (tant qu'il en reste), appelle la fonction *reversehash()*, puis place le mot de passe obtenu sur la pile en phase (4).
4. La pile contient tous les mots de passe hashés avec le plus grand nombre de voyelles (ou de consonnes, en fonction de l'option choisie). Plus d'informations sur cette pile seront données dans la section Choix de conception. Une fois le mot de passe placé, le thread recommence le processus.
5. Une fois tous les fichiers lus, la pile est imprimée soit sur la sortie standard, soit sur un fichier placé en argment.

3 Choix de conception

Dans cette section, nous allons aborder différents aspects de notre code qui pourraient le rendre plus efficace ou le différencier des autres.

3.1 Parallélisation

Tout d'abord, notre choix d'opter pour un même processus à suivre pour tous les threads n'est pas du au hasard. D'abord, il permet de ne pas devoir passer par une file intermédiaire par laquelle deux threads qui exécuteraient deux procédés différents devraient se donner des informations. Ceci nous évite de causer une occupation de mémoire inutile et de devoir gérer un problème de producteur-consommateur parfaitement évitable. Cela assure aussi une répartition du travail équitable entre les threads et utilise chaque thread de la meilleure des façons.

3.2 Pile

La pile que nous avons implémentée est constamment mise à jour par les threads qui la remplisse. Lorsqu'un thread place un mot de passe dans cette pile (lorsqu'il appelle la fonction *place()*), il ne fait pas que simplement poser ce mot de passe sur la pile. Il compare le nombre de voyelles (ou de consonnes) que contient son mot de passe avec le nombre de voyelles maximum (ou de consonnes) qui est une variable de la structure Stack (pile). Si ce nombre est inférieur, il libère le mot de passe et l'espace qu'il prenait. Si ce nombre est égal, il place le mot de passe sur la pile. Si ce nombre est supérieur, il retire tous les éléments de la pile, place son mot de passe et met à jour la variable du nombre voyelles (ou de consonnes) maximum de la pile. Ainsi, les mots de passes sur la pile sont toujours ceux avec le plus de voyelles (ou de consonnes) et la mémoire occupée par la pile est ainsi optimisée.

3.3 Exclusion mutuelle

Le problème bien connu de l'exclusion mutuelle causée par l'utilisation de threads a été gérée par deux Mutex. Un Mutex protège l'accès au fichier ouvert et à la variable utilisée pour compter le nombre de hash restant à lire dans ce fichier. Ce Mutex permet d'éviter que cette variable soit modifiée en même temps par deux threads et ne cause un problème dans la lecture du fichier. Le deuxième mutex est utilisé pour protéger la pile. Puisque la pile est imprimée à la fin du procédé, cette situation n'est pas un cas de producteur-consommateur et ne nécessite pas l'utilisation d'un Mutex et d'une Sémaphore supplémentaire.

4 Evaluation qualitative

Pour cette évaluation qualitative, nous avons effectué deux différents tests visant à montrer les avantages d'un programme multi-threadé ainsi que ses limites.

Le premier test a été réalisé avec un fichier en entrée contenant un grand nombre de mots de passe hashés, mais dont la fonction *reversehash()* était relativement faible en temps.

On peut observer sur la figure 1 que le temps d'exécution baisse de manière générale en fonction du nombre de threads. Cependant, on peut voir qu'il reste plus ou moins constant au-delà de 2 threads utilisés. Ce résultats n'est pas surprenant puisque l'ordinateur utilisé pour réaliser ces tests contient un processeur bi-coeur. Il ne peut donc exécuter simultanément que 2 différents threads. Les quelques variations du temps d'exécution en fonction du nombre de threads au-delà de 2 sont la conséquence de l'exclusion mutuelle au niveau du fichier d'entrée et de la pile. En effet, certains threads se voient bloqués dans leur avancement par un appel à *pthreadmutexlock()* refusé et ceci ralenti donc leur progression.

Le deuxième test consistait à donner un entrée un fichier binaire avec peu de mots de passe, mais dont la fonction *reversehash()* est très lourde en temps.

On peut voir sur la figure 2 que le deuxième test confirme bien les résultats du premier. On observe encore la baisse de temps entre 1 et 2 threads ainsi que les légères variations de temps d'exécution au-delà de 2 threads.

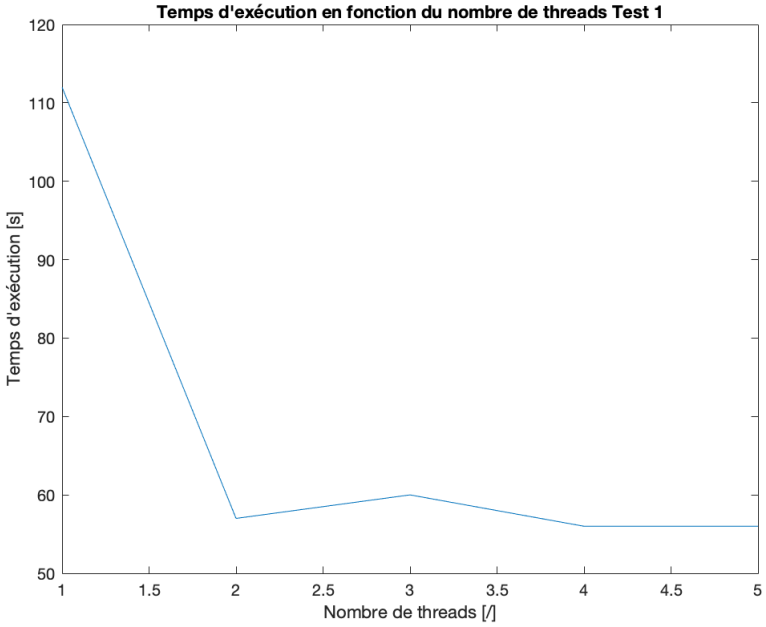


FIGURE 1 – Résultats du premier test

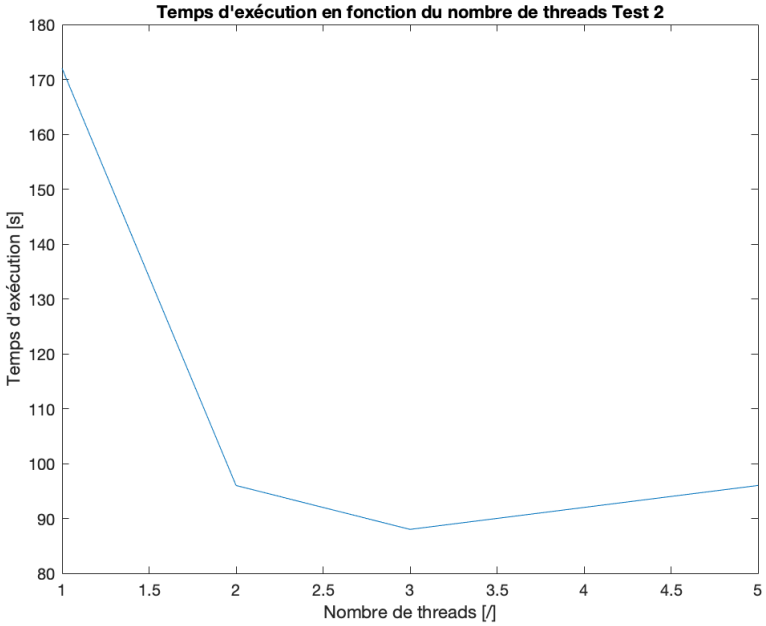


FIGURE 2 – Résultats du deuxième test

5 Tests

Lorsque nous lançons la commande *make test* dans le terminal, nous lançons trois scénarios. Le premier est un scénario avec deux fichiers binaires en entrée pour montrer le bon fonctionnement du programme face à plusieurs fichiers. Le deuxième est un cas où le programme doit écrire les mots de passe candidats obtenus sur un fichier texte. Le dernier est un scénario dans lequel le fichier binaire en entrée n'existe pas. Il a pour but de montrer le caractère défensif de notre code.

6 Conclusion

Pour conclure, nous pouvons dire que notre code est simple et robuste. Il optimise correctement la mémoire utilisée par le programme et ne nécessite pas d'utilisation superflue de nombreux Mutex et Semaphores qui compliquent souvent le code. Il nous montre aussi les gains de temps que peuvent apporter une programmation multi-threadée mais aussi les limites causées par le nombre de coeurs qu'un processeur peut contenir.

Notre code a aussi été construit en programmation en binôme, méthode qui nous permettait d'avancer rapidement sans faire d'erreurs. Nous avons opté pour une programmation défensive et communicative pour permettre à un lecteur qui ne connaît pas le code de bien comprendre les éventuelles raisons d'un disfonctionnement du code causé par un mauvais argument en entrée, par exemple.