Rapport de fin de projet

Thibault Blanpain 21361700
Natacha Goux 51791700

2019

1 Introduction

Le monde évolue sans cesse, et sans pitié. N'importe quelle espèce non-adaptée à cette évolution inflexible est vouée à disparaître sans autre forme de procès. Ainsi, afin de survivre dans cet environnement impitoyable, l'homme dut s'adapter. C'est ainsi que naquirent moult technologies, dont l'informatique. Quiconque désire participer à la construction de cet édifice technologique se doit d'en maîtriser certains concepts sur le bout des doigts. Parmi ces concepts se trouve notamment la course à la vitesse. Cependant, après avoir optimisé du mieux possible les composants des ordinateurs, les informaticiens n'eurent d'autre choix que d'imaginer de nouvelles techniques pour permettre à leurs programmes de suivre un rythme de calcul encore plus soutenu. Grâce aux efforts conjugués de plusieurs savants, l'art du multithreading vit le jour ! Cette méthode, consistant en la parallélisation d'opérations de calculs, permit d'améliorer grandement la vitesse des logiciels.

C'est pourquoi, dans le cadre du cours LSINF1252, le groupe composé de Natacha Goux et Thibault Blanpain décidèrent de tenter d'approcher l'art complexe de la programmation mettant en oeuvre plusieurs threads. Ce rapport de fin de projet a pour objectif de présenter quatre sujets. Tout d'abord, l'architecture de haut niveau du programme sera présentée. Ensuite, vient une discussion des choix de conception, qui permettraient de différencier ce projet des autres. Par après, la stratégie de test utilisée pour valider le programme sera introduite. Ce rapport se finit sur une évaluation quantitative des performances réelles du programme.

2 Architecture du programme

Le programme se présente relativement simplement : il suit le schéma du producteur-consommateur. En effet, avant de pouvoir afficher les mots de passe décryptés, il fallait tout d'abord passer une série d'étapes.

Pour débuter, le programme commence, dans la fonction *main()*, par lire les options entrées lors de l'appel au programme. Par après, un thread de lecture — par type d'entrée — a pour travail de lire les fichiers dont les noms sont donnés en argument à l'exécutable *cracker.o*. Cette opération de lecture se déroule de la manière suivante : les fichiers, un par un, sont ouverts, et, ligne par ligne, les hashs sont copiés dans un tableau, dont l'accès est régulé par des sémaphores. Ces mêmes sémaphores seront discutés dans la section suivante. Dans le schéma du producteur-consommateur, ces threads de lecture représentent les threads producteurs.

Une fois que le tableau contient au moins un élément, une quantité précise (soit spécifiée en option, soit un seul par défaut) de thread sont crées. Ces *nthread* ont pour but d'effectuer l'opération de décryptage, coûteuse en CPU. Pour ce faire, lorsque le tableau est non vide, ces threads y accèdent afin de récupérer un hash de mot de passe. Ainsi, chacun de ces threads de calcul va produire, si possible, un mot de passe clair, à partir des hashs récupérés dans le tableau. Ces threads de reversehash, ou de calcul, sont représentés par les threads consommateurs dans le modèle du producteur-consommateur.

Dans l'optique de stocker au fur et à mesure ces mots de passe décryptés, une liste chaînée est créée. Ainsi, au fur et à mesure que les consommateurs décryptent des mots de passe, des structures *Candidats* sont ajoutées à la liste.

Cependant, tous ces *Candidats* ne sont pas d'office des mots de passe éligibles. C'est là que la fonction de tri des mots de passe décryptés intervient. Celle-ci parcourt la liste, et retire tous les éléments ne vérifiant pas le critère de sélection, en l'occurrence, le nombre de voyelles (ou de consonnes, si précisé dans les options).

Enfin, c'est ici qu'intervient la dernière fonction, dont le seul travail est d'à nouveau parcourir la liste chaînée (triée, cette fois-ci), et d'en afficher les mots de passe décryptés sur la sortie standard — ou dans un fichier dont le nom a été spécifié en option.

3 Choix de conception

Cette section a pour objectif de décrire certaines particularités du programme, dont le fonctionnement pourrait être soit légèrement obscur, soit intéressant à discuter.

Le premier point intéressant à discuter consiste en la partie très importante du programme : la gestion de l'accès au tableau, contenant les hashs des mots de passe, par la paire producteur-consommateur. En effet, en cas de mauvaise gestion des accès, il se peut que certaines données soient perdues, ou bien certains calculs effectués plusieurs fois, ou bien même que le programme plante, tout bonnement. Afin de coordonner les threads consommateurs entre eux, et le couple de production-consommation, comme précisé dans la section précédente, l'usage de sémaphores a été réalisé. Concrètement, deux sémaphores sont initialisés : *semHashBufEmpty* et *semHashBufFull* dont les rôles sont respectivement d'indiquer si le tableau de hashs possède au moins une ligne de vide, et si le tableau de hashs possède au moins une ligne de remplie. Ceci permet déjà d'empêcher les threads producteurs d'essayer de remplir le tableau lorsqu'il est plein, mais aussi d'éviter que les threads de calculs ne tentent de décrypter une ligne vide. Cependant, malgré le fait que les threads savent maintenant accéder de manière plus ou moins sécurisée au tableau, il faut encore régler le problème de la navigation dans ce-dit tableau. Il a été décidé d'y naviguer via un index global, dont l'accès est protégé par un mutex. Cet indice est augmenté d'une valeur 1 à chaque fois qu'un producteur ajoute un élément dans le tableau et diminué de la même valeur lorsque qu'un consommateur accède au tableau. Pour éviter que divers threads ne modifient en même temps la valeur de l'index, celui-ci, lors de chaque utilisation, est protégé par un mutex. Voilà ce qui pouvait être dit quant à la régulation de l'accès au tableau de hashs.

Un autre choix de conception qui mérite d'être abordé est celui de la liste chaînée. En effet, la liste de candidats aurait pu être gérée de quantité de manières différentes ; trier la liste de manière régulière et aléatoire via un thread de tri, utiliser un tableau à la place d'une liste chaînée... Cependant, le choix d'une liste chaînée semblait s'imposer assez aisément, au vu de ses nombreux avantages : d'une part, il est très facile de manipuler les éléments d'une liste, d'en supprimer certains, ajouter d'autres, etc. D'autre part, une liste chaînée permet de stocker plusieurs paramètres pour chaque élément, via une structure. Ici, il a été jugé utile de décrire chaque *Candidats* comme détenteurs de trois critères : un pointeur vers le candidat suivant, le mot de passe au clair ainsi que le nombre d'occurrence du critère de sélection. Ces choix permettent, à l'aide d'une seule liste, d'effecteur plusieurs opérations très importantes, telles que la création de la liste en tant que telle, le tri des candidats et l'impression/stockage des candidats retenus. Le seul inconvénient pouvant être cité serait le fait que gérer une liste chaînée a demandé l'implémentation de nouvelles fonctions afin de pouvoir créer facilement de nouveaux noeuds, les ajouter à la liste et calculer le nombre d'occurrence de voyelles.

Ensuite, il est intéressant de remarquer que la structure de *Candidats* n'admet que trois critères, dont *nbrOccurence*. Cet élément de la structure ne représente pas obligatoirement le nombre de voyelles du mot de passe associé, malgré que ce soit le critère de tri par défaut. Il a été décidé de créer une fonction dont le seul but était de calculer le nombre d'occurrence de voyelles. Cependant, si le paramètre *consonne* est passé à *true*, le nombre de consonnes est calculé en effectuant tout simplement le calcul $nombre\_Occurrence = nombre\_Lettre\_Mdp - nombre\_Voyelles$. Ensuite seulement, le nombre d'occurrences est associé à l'élément *nbrOccurence*. Ceci permet d'éviter de devoir calculer et stocker à la fois le nombre de voyelles et à la fois celui de consonnes dans la structure *Candidats*.

4 Stratégie de tests

Concernant la stratégie de test suivie pour valider le programme, celle-ci se déroule en deux étapes majeures : tout d'abord, des sanity checks sont effectués, et par après, des tests plus complets prennent place.

Les sanity checks permettent de vérifier qu'en cas d'utilisation très simpliste du programme, l'output de celui-ci reste conforme à ce qui est attendu. Ainsi, le premier test se contente d'appeler l'exécutable, sans aucun argument. Comme prévu, la sortie affiche un message d'erreur précisant qu'il est nécessaire qu'au moins un argument soit présent pour le fonctionnement du programme. Cet argument minimal est un nom de fichier. Le second sanity

check, dans la lignée du précédent consiste à appeler l'exécutable avec un seul argument ; un nom de fichier vide. La sortie standard affiche n'affiche simplement aucun candidat, comme attendu.

Ensuite, maintenant que le fonctionnement du programme dans des cas minimalistes a été vérifié, des tests plus complets et complexes peuvent être exécutés. Le premier de ceux-ci consiste en l'appel de l'exécutable avec comme argument un tout petit fichier "4hash.bin" contenant 4 hashs de mots de passe. Ceci permet de vérifier simplement et rapidement que la grande majorité des fonctions implémentées sont fonctionnelles. Le test suivant diffère du précédent par la taille du fichier à lire : "01_4c_1k.bin". Les tests suivants traitent le même fichier, mais permettent de vérifier le bon fonctionnement du traitement des options. Pour ce faire, le prochain test a pour objectif de vérifier que le paramètre du nombre de threads est bien implémenté, en appelant *cracker.o* avec l'option **-t 6**, qui doit faire en sorte de créer 6 threads de calcul. Un test semblable permet de vérifier le paramètre de tri, en utilisant l'option **-c**, qui indique au programme que les candidats doivent être triés selon leur nombre de consonnes. Un test final permettant de valider le programme appelle *cracker.o* avec comme options **-t 12 -c** et comme fichier de hashs "01_4c_1k.bin" et "02_06_5.bin". En effet, ce test combine toutes les options possibles, mais aussi la lecture, le traitement d'un grand nombre de hashs correspondant à des mots de passe de seulement quatre caractères, mais aussi d'un petit nombre de hashs correspondant à des mots de passe de 6 caractères, dont l'opération de reversehash est très coûteuse en CPU.

5 Évaluation quantitative

N'ayant pas réussi à faire en sorte que notre programme sorte les mots de passe triés, à cause d'une segfault qui n'a pas su être traitée, effectuer une évalution quantitative des performances du programme ne rime à rien. Cette section détaillera donc plutôt quels objectifs ont été atteints, quels objectifs ne le furent point, ainsi que différentes pistes d'améliorations du programme.

Tout d'abord, il est facile de remarquer que le programme, même non-fonctionnel, fait bien utilisation des threads selon le critère du non-déterminisme. Le programme n'est pas séquentiel. En effet, après avoir compilé le programme avec **gcc -Werror -Wall -lpthread** comme options et lancé l'exécutable — avec Valgrind, ou GDB, cela se vérifie également, - une variation dans l'ordre des messages apparaissant sur le terminal est observable. Ceci est bien la preuve que le programme est non-séquentiel.

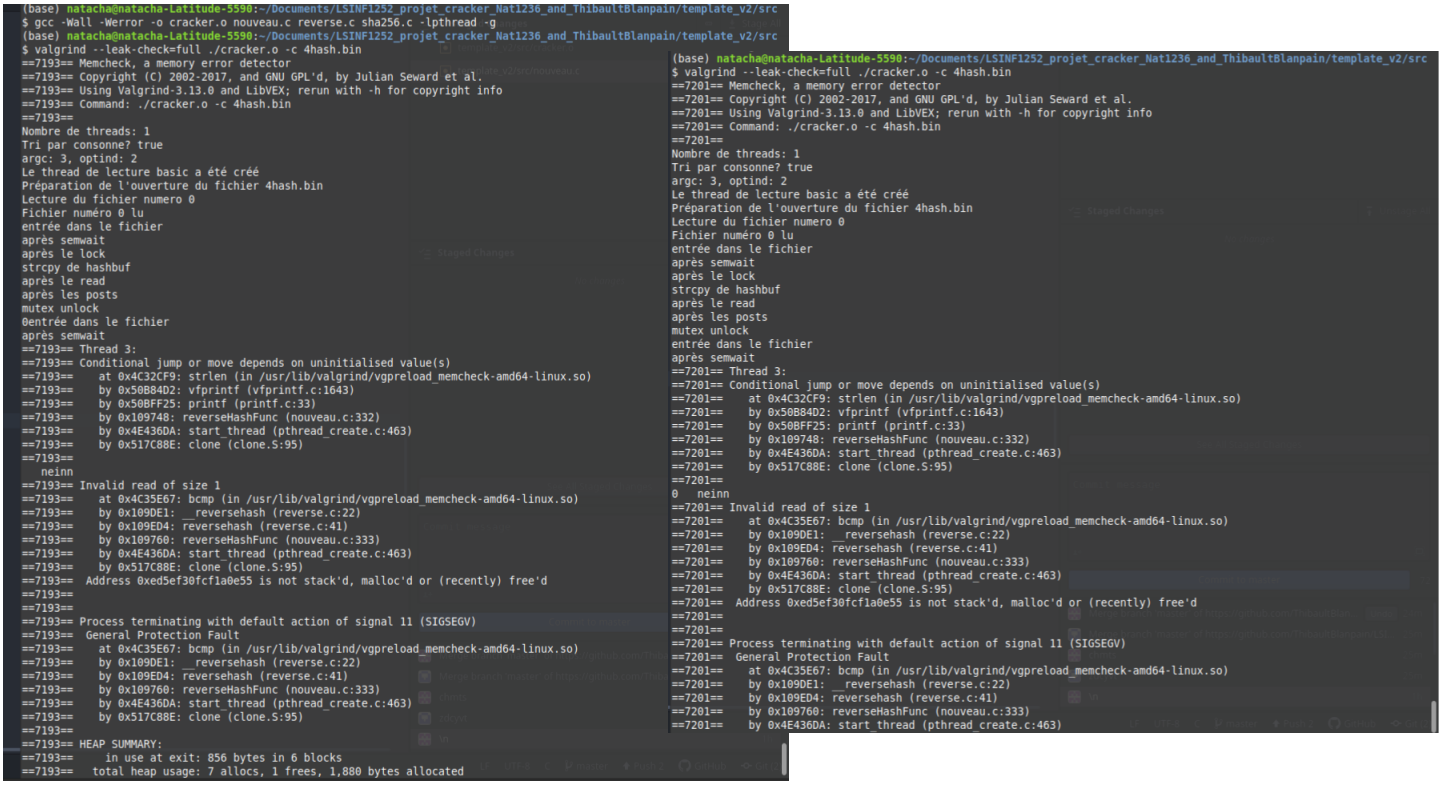

FIGURE 1 – Mise en évidence du non-déterminisme

Afin de décrire le plus précisément possible ce que le programme n'est pas capable de faire, il est très intéressant de se pencher de près sur l'endroit supposément fatidique du programme. L'on pense que le problème majeur se situe dans la fonction associée au thread de lecture, à l'endroit où celui-ci tente de remplir *hashBuf[indexG]* (tel que *hashBuf* est de type *uint8_t ***) avec *buf* qui est de type *(uint8_t *)*, qui est lui-même rempli avec la fonction *read(fd, buf, size)*. Cependant, il est aussi possible que le problème ne vienne pas de là, mais plutôt de la fonction associée au thread de calcul. Il est en effet possible que celle-ci tente de récupérer le hash dans *hashBuf[indexG]*, alors que cette position pointerait vers un espace mémoire indéfini, ce qui rendrait totalement impossible d'utiliser la fonction *reversehash(localHash, candid, 16)*. Ce sont deux suppositions qui ont été analysées du mieux possible, ave l'aide de Valgrind et GDB, sans que l'erreur ait su être gérée convenablement. En résumé, le programme ne sait pas calculer l'inverse d'un hash, très probablement à cause d'un mauvais accès (en production ou consommation) au buffer *hashBuf*.

Enfin, il peut être utile de proposer des pistes d'amélioration du programme. Il aurait été en effet possible d'implémenter la fonction de tri et la fonction d'affichage en une seule fonction parcourant la liste chaînée, et n'affichant que les mots de passe optimaux. Ceci aurait permis d'éviter de parcourir cette liste chaînée une fois de trop.