ÉCOLE POLYTECHNIQUE DE LOUVAIN

UCLouvain

LSINF 1252 - Systèmes informatiques - 2019

Password cracker

Auteurs

Camberlin	Merlin	0944-17-00
Pisvin	Arthur	3711-17-00

Remis le : 10 mai 2019

Professeur :
Bonaventure O.

Assistant :
De Coninck Q.

1 Architecture à haut niveau

1.1 Fonctionnement général et structures de données utilisées

Le fonctionnement de base de notre programme repose sur l'utilisation de deux producteurs-consommateurs. La FIGURE 1 ci-après représente le fonctionnement en 4 étapes de notre programme :

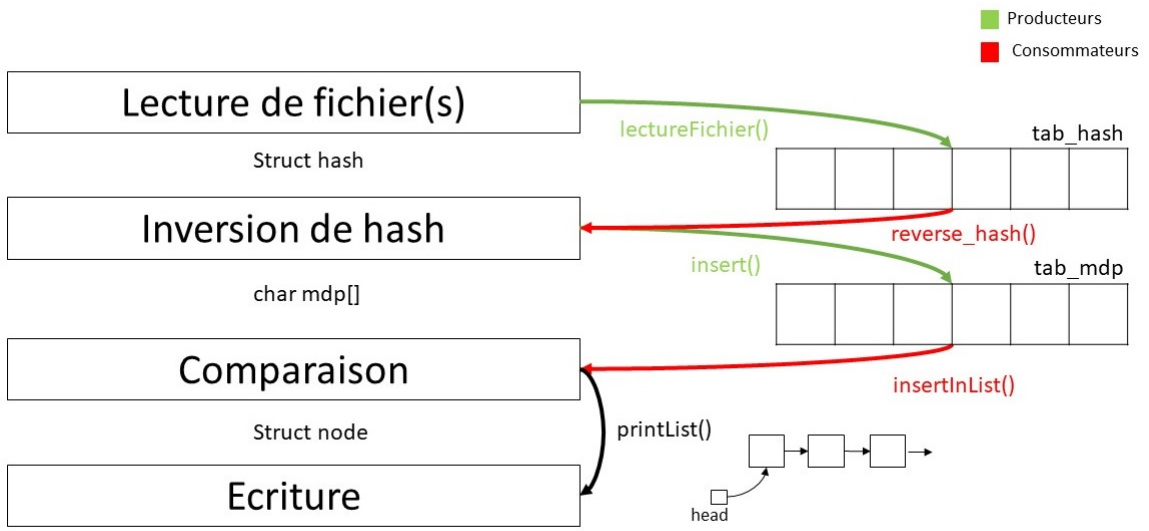

FIGURE 1 – Fonctionnement général

1. La première étape consiste à lire le(s) fichier(s) binaire(s) par 32 octets pour en extraire les hash à inverser et à les insérer dans le premier buffer appelé `tab_hash`. Les hash sont repésentés via une structure de données nommée `hash` contenant un tableau de 32 `char` tandis que le buffer `tab_hash` est un tableau de pointeurs vers les hash lus.

```
typedef struct hash
{
  char hash[32];
}hash;
```

2. La seconde étape consiste à inverser au fûr et à mesure les hash contenu dans `tab_hash`. Si la fonction `reversehash()` trouve un inverse, alors le mot de passe en clair trouvé est inséré dans le second buffer nommé `tab_mdp`. Le mot de passe en clair est représenté par un tableau de caractères nommé `mdp`, tandis que le buffer `tab_mdp` est représenté par un tableau de pointeurs vers ces mots de passe en clair.

   Tout comme pour le premier buffer, l'avantage d'utiliser un tableau de pointeurs permet de diminuer la mémoire allouée par notre programme. En effet, au lieu de stocker des hash complets, c'est-à-dire 256 bits, on ne stocke que des pointeurs de hash qui occupent moins de place en mémoire.

3. La troisième étape consiste à comparer les différentes inversions des hashs suivant le critère de sélection et à insérer les mots de passe vérifiant ce critère dans une liste simplement chaînée représentée par un pointeur vers sa tête nommé `head**`. La liste simplement chainée est consistutée de noeuds représenté par la structure de données `node` contenant un pointeur vers le noeud suivant ainsi que le mot de passe en clair.

```
typedef struct node
{
  char mdp[16];
  struct node next;
}node;
```

4. La quatrième et dernière étape consiste à parcourir la liste chainée pour afficher son contenu sur la sortie standard ou pour l'écrire dans un fichier externe.

1.1.1 Types de threads entrant en jeu

Pour l'implémentation, des threads `Posix` seront utilisés. Leurs tâches seront diverses. Un thread sera chargé d'exécuter la fonction principal. Deux autres s'occuperont respectivement de la lecture et de la comparaison. Les autres threads s'occuperont du calcul de l'inverse de hash.

1.1.2 Méthode de communication entre les threads

Les buffers représentent des ressources partagées entre plusieurs threads. Pour coordonner les threads entre eux et éviter qu'ils ne violent la zone critique que représente chaque buffer, deux sémaphores et un mutex seront utilisés pour chaque buffer. Le premier sémaphore sera chargé de compter le nombre de slots disponibles, tandis que le suivant ceux remplis. Quant au mutex, celui-ci *"lockera"* ou *"unlockera"* la ressource respectivement avant et après chaque utilisation de celle-ci par un thread.

1.1.3 Informations communiquées entre les threads

Les threads communiquent entre-eux via des variables globales.

- `CalculExecution` - compte le nombre de threads qui sont en cours d'exécution de la fonction `reversehash()`.
- `fin_de_lecture` - indique si la lecture de tous les fichiers binaires a été terminée ou non.
- `nbreSlotHashRempli` - compte le nombre de hash qui sont dans le buffer `tab_hash`.
- `nbreSlotMdpRempli` - compte le nombre de mots de passe qui sont dans le buffer `tab_mdp`.

1.2 Lecture de fichier(s)

L'étape de lecture de fichier(s) est jouée par la fonction `lectureFichier()`. Celle-ci lit par 32 octets de données les hash d'un même fichier avant de passer au fichier suivant. Au fûr et à mesure, les hash sélectionnés seront introduits dans le premier buffer nommé `tab_hash`.

1.3 Inversion de hash

L'étape d'inversion des hash est jouée par la fonction `reverse_hash()` qui calcule l'inverse d'un hash via l'opération coûteuse `reversehash()` et place ensuite le résultat dans le second buffer nommé `tab_mdp`. L'opération `reversehash()` est le consommateur du premier buffer contenant les hashs. Tandis que l'insertion du mot de passe en clair dans `tab_mdp` réalisée par `insert_mdp()` est le producteur du second buffer.

1.4 Comparaison

L'étape de comparaison est jouée par la fonction `insertInList()` qui joue le rôle de consommateur du second buffer en déterminant si un mot de passe en clair doit être inséré dans la liste simplement chaînée ou doit être ignoré selon le critère de selection.

Comme le nombre de mots de passe susceptible de remplir le critère de sélection ne peut pas être déterminé à l'avance, l'utilisation comme structure de données d'une liste simplement chainée était préférable à l'utilisation d'un tableau, nécessitant de réverser une certaine quantité de mémoire à sa création.

1.5 L'écriture

L'étape d'écriture est jouée par la fonction `printList()` qui affiche sur la sortie standard ou dans un fichier externe le contenu de la liste simplement chainée.

2 Choix de conception et d'implémentation

2.1 Thread de lecture

Si on souhaite exécuter un thread de lecture par fichier, alors, dans la situation de deux fichiers sur le même disque dur, ce choix d'implémentation nuirait aux performances temporelles du programme. En effet, comme l'emplacement en mémoire des fichiers sur le disque durs sont différents, utiliser un thread par fichier reviendrait à déplacer successivement la tête de lecture vers l'emplacement du fichier 1 et ensuite vers l'emplacement du fichier 2. Or, le déplacement d'une tête de lecture dans différents secteurs d'un disque dur est une opération coûteuse qui prend un temps de l'ordre de la milliseconde.

Un choix plus judicieux serait d'exécuter un thread par **type** de fichier. Cette fois, dans la situation de deux fichiers sur le même disque dur, un seul thread ne serait exécuter tandis que dans le situation d'un fichier sur disque dur et un sur réseau, deux threads seraient utilisés simultanément.

Même si utiliser un thread par fichier d'origine différente (disque dur, réseau,...) optimise la rapidité de lecture des fichiers, nous avons préféré lire chaque fichier complètement avant de passer au suivant car le gain de temps de lecture gagné en utilisant l'autre solution n'aurait pas pu être persceptible. En effet, comme le producteur lisant les fichiers binaires est beaucoup plus rapide que le consommateur qui calcule l'inverse des hash, une fois que le buffer `tab_hash` est rempli, les threads de lecture sont en attente et doivent attendre que le consommateur vide le buffer avant de reprendre la lecture des fichiers. Ce temps d'attente occassionné compense le retard de la lecture individuelle de chaque fichier.

2.2 Structure `hash`

Dans la structure `hash`, on retrouve un tableau de 32 caractères. En réservant de la mémoire pour un tableau de 32 caractères, on réserve 32 octets de mémoire puisqu'un caractère est stocké sur 1 octet. Cette réservation de mémoire est donc adéquate pour y placer les 32 octets de données que représente un hash.

2.3 Choix de la taille des buffers

Pour déterminer le nombre de slots dans chaque buffer, nous avons jugé que prendre le double du nombre de threads de calcul comme nombre de slots du buffer était la taille adéquate pour permettre un bon compromis entre l'utilisation des producteurs et des consommateurs.

2.4 Fonctionnement de la fonction `insertInList()`

La fonction `insertInList()` se comporte différemment dans 3 situations.

- Si le mot de passe passé en argument contient plus d'occurences de voyelles ou consonnes que les mots de passe précédents, alors `insertInList()` libère la liste chaînée maintenue jusqu'ici via la fonction `freeLinkedList()` et ensuite, crée une nouvelle liste chaînée pour y placer le mot de passe avec le plus d'occurences de voyelles ou consonnes.
- Si le mot de passe passé en argument contient autant d'occurences de voyelles ou consonnes que les mots de passe précédents, alors `insertInList()` ajoute le nouveau mot de passe en tête de liste.
- Si le mot de passe passé en argument contient moins d'occurences de voyelles ou consonnes que les mots de passe précédents, alors `insertInList()` ignore le mot de passe.

2.5 Compteur du nombre de threads en cours d'exécution de reversehash()

A chaque fois d'un thread commence ou termine le calcul de `reversehash()` le compteur `CalculExecution` est respectivement incrémenté et décrémenté. Cette manipulation permet de palier un problème que l'on rencontre lorsque le nombre de threads de calcul est égal au nombre de hash à inverser. Sans vérifier le nombre de threads en train d'effectuer le calcul de `reversehash()`, la situation suivante pourrait se produire.

Considérons 3 threads de calculs et 3 hash à inverser. Selon notre implémentation, nous accordons 6 slots dans chaque buffer. Comme la lecture est rapide comparé à l'opération d'inversion, les 3 hash sont presque immédiatement insérés dans le premier buffer. Les 3 threads de calcul peuvent alors calculer l'inverse de ces

hash et vider complètement le premier buffer. Au moment où le premier buffer a été vidé mais où les résultats de l'inverse des hash n'a pas encore été calculé, la lecture est terminée et les buffers sont tous vides et par conséquent notre implémentation considérerait que le programme a fini son exécution. Vérifier la valeur de `CalculExecution` permet de palier ce problème.

3 Tratégie de tests

Pour nos tests, nous avons procédé en deux parties.

Premièrement, nous avons effectué des tests unitaires chargés de veiller au bon fonctionnement des trois principales fonctions (`lectureFichier`, `reverse_hash()`, `insert()`) indépendamment. Nous avons lancé chaque fonction séquentiellement pour comparer le résultat avec ce qu'on attendait.

Deuxièmement, nous avons effectué un test global de notre programme. Pour ce faire, nous avons créé notre propre fichier binaire contenant les hashs des mots suivants : (oui, non, sac, porc, ours) et lancer `cracker` avec ce fichier binaire en entrée.

3.1 Remarques

- Nos test CUnit n'ont volontairement pas été insérés dans le `makeFile` car en essayant de les lancer sur les machines de la salle intel, une erreur de librairie apparaissait. Après recherche, il semblerait que l'erreur se produit car l'utilisation des librairies `CUnit` nécessite une installation au préalable sur ces machines. Toutefois, en lançant ces tests sur nos machines, leur résultat était correct et correspondait aux résultats attendus.
- Lorsque l'on lance le test de 1000 hashs fournis avec le template avec 1000 threads, il arrive parfois que le programme se bloque au moment décrit ci-après. Le *deadlock* apparait lorsque la lecture des fichiers est terminée, que le buffer `tab_hash` ainsi que le buffer `tab_mdp` sont vides, que les mots de passes ont tous été traduits mais qu'il reste 1 thread en cours d'exécution (`CalculExecution =1`). Jusqu'à ce jour, nous cherchons toujours à comprendre la raison de ce blocage. Pour notre défense, jusqu'à ce jour, en testant le programme avec des fichiers contenant moins de hash (<1000), aucun *deadlock* n'était apparu.
- Lorsque nous avons essayé de lancer notre programme en spécifiant l'option `-o "FichierOut"` sur les machines de la salle *intel*, une erreur apparaissait alors que sur nos propres ordinateurs, elle ne se produisait pas. Nous supsectons un problème de permission comme cause de ce problème.

4 Evaluation quantitative

Sur un ordinateur de processeur Intel Core i7-6500U Dual-Core de fréquence de CPU de 2.5 GHz avec 8Gbytes de mémoire RAM, en DDR3, on observe la variation du temps d'exécution en fonction du nombre de threads représenté sur la FIGURE 2 avec comme fichier d'entrée `02_6c_5.bin`

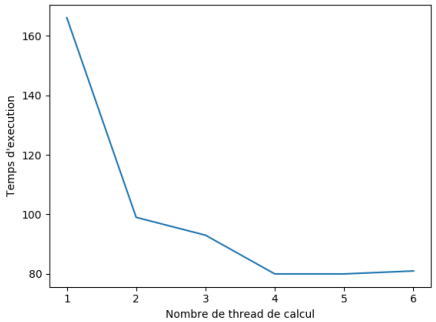

FIGURE 2 – Graphe du temps d'exécution en fonction du nombre de threads de calcul

On remarque que le temps d'exécution diminue significativement pour les premiers threads mais qu'ensuite, les gains de temps ne sont plus autant marqués.