# LSINF1252

Jérôme Delforge, Corentin Denis

3 avril 2019

## 1 Architecture

### 1.1 Structures de données

La représentation de nos données a été principalement dictée par la méthode reversehash, qui prend en argument un pointeur uint8_t* et écrit le résultat dans un pointeur char*. Les hashs après lecture du fichier seront donc sous la forme uint8_t et les mots de passe donnés par reversehash seront sous la forme char*.

### 1.2 Types de threads

Nous avons définis trois types de threads différents :

— Le premier type de thread (T1) aura pour fonction de lire le fichier, séparer les différents hashs et les stocker dans un tableau de uint8_t*. Notre architecture ne prévoit qu'un seul thread de ce type. Ce sera géré par le thread principal.

— Le second type de thread (T2) parcourera le tableau de uint8_t* rempli par le premier thread et passera chaque élément du tableau dans la fonction reversehash. Le mot de passe ainsi obtenu sera stocké dans un tableau de char*. Le nombre de threads de ce type sera à préciser dans l'appel de notre méthode. Ces threads étants les plus lents, ce seront eux qui détermineront la vitesse de notre programme.

— Le troisième et dernier type de thread (T3) parcourera le tableau de mots de passe, fera une comparaison entre chaque élément et gardera les plus pertinents en fonction du choix émis à l'appel de la méthode (mdp avec le plus de voyelles ou de consonnes). Ces mots de passe seront stockés dans une liste chaînée. Nous ne prévoyons également qu'un seul thread de ce type-là.

### 1.3 Méthode de communication entre threads

Dans l'implémentation telle qu'elle est prévue, le thread T1 communique avec les threads T2 et les threads T2 communiquent avec le thread T3. Il n'y a à prioris aucune raison pour que T1 et T3 doivent échanger des informations.

Les threads T2 présentent deux sections critiques, la première quand ils doivent rechercher un hash dans le tableau rempli par T1, la seconde quand ils doivent écrire un mot de passe dans le tableau utilisé par T3. Un autre problème étant la lenteur de la méthode reversehash.

Afin de résoudre ces problèmes, voici la méthode que nous avons décidé d'adopter. Le tableau rempli par T1 (tab1) serait composé de N cases si nous avons N threads T2, il en va de même pour le tableau rempli par les threads T2 (tab2). C'est à dire qu'un thread T2 n'aurait accès qu'à certaines cases de tab1, ainsi les différents threads T2 ne pourraient jamais vouloir chercher la même information au même moment. Vu la lenteur des threads T2, il serait inutile de créer des tableaux plus grands.

### 1.4 Informations communiquées entre threads

Les informations communiquées entre les threads seront stockés dans deux tableaux différents :

— Premièrement tab1 de type uint8_t**, le thread T1 lira les hash et les stockera dans ce tableau. Les threads T2 récupéreront les hashs dans ce tableau pour les traduire.

— Deuxièmement tab2 de type char**, les Threads T2 stockeront les mots de passe traduits dans ce tableau, ceux-ci seront récupérés par le thread 3 pour les comparer.

Une fois triés, les mots de passe sont stockés dans une liste chaînée, cependant le seul thread pouvant y accéder est le thread T3.

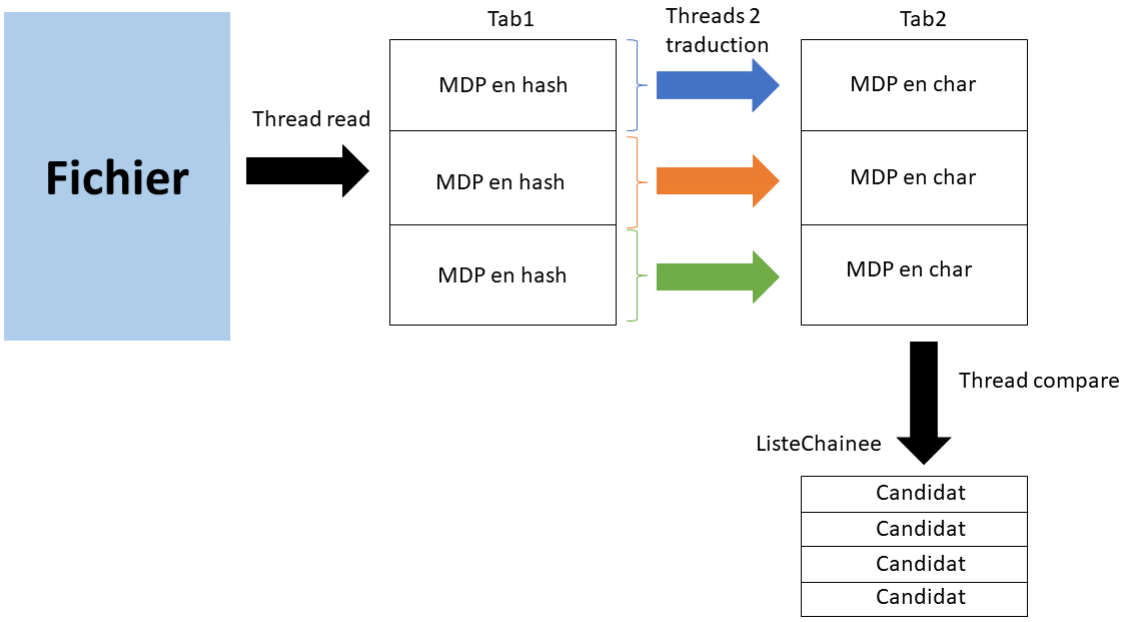


FIGURE 1: Architecture

## 2 Choix de conception

La gestion des threads ce fait principalement via 5 sémaphores :

— Deux sémaphores liant T1 et les T2, caractérisants l'un le nombre de places libre dans tab1, l'autre le nombre de places occupées. Ainsi, à l'entrée du thread T1, si tab1 est rempli, le thread attendra qu'une place se libère avant de pouvoir écrire dedans. Les threads T2 attendront également si tab1 est vide, qu'un hash soit écrit dedans par T1.

— Deux autres sémaphores liants les threads T2 et T3 utilisants la même stratégie. Si tab2 est rempli, les threads T2 attendrons qu'une place se libère pour pouvoir commencer la traduction et le thread T3 attendra qu'il y aie des mots de passe dans tab2 pour pouvoir les comparer.

— Un dernier sémaphore surviens à la fin pour que la fonction main attende que l'impression des résultats par le thread T3 soit finie pour pouvoir arrêter le programme.

## 3 Stratégie de test

Par manque de temps, nous n'avons pas élaboré de tests très poussés. Nous lançons simplement le programme plusieurs fois avec un nombre de threads différent.

## 4 Évaluation quantitative

Nous avons fait une série de tests afin d'évaluer les performances de notre programme, voilà ce qui en est ressorti :

Notons tout d'abord que notre programme ne fonctionne pas avec un seul thread. En effet, le programme se bloque une fois arrivé à la fin du fichier, cela est sans doutes du à une mauvaise gestion des sémaphores. Cependant, étant donné que notre programme fonctionne correctement avec plus de 1 thread, nous avons décidé de ne pas nous attarder sur ce point.

Le programme ne fonctionne également pas avec le fichier de 2000 petits mots de passe, Nous pensons que cela est du à un problème d'allocation de mémoire, mais nous n'avons pas réussi à le régler.

Voici les courbes du temps que prend notre programme en fonction du nombre de threads :

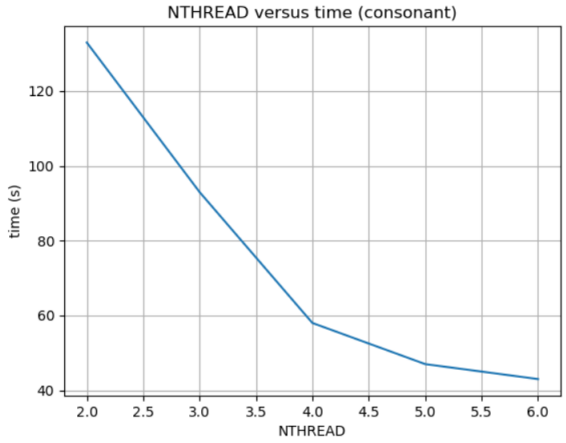


FIGURE 2: Architecture

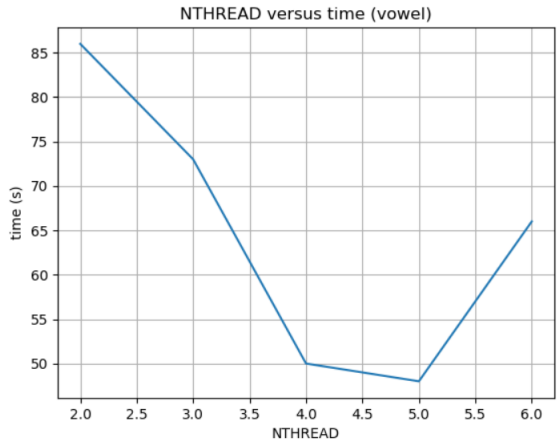


FIGURE 3: Architecture