Programmation système avancée
TP n° 8 : Mapping de fichier et table de hachage persistante

Le but de ce TP est d'utiliser un format très simple de table de hachage **persistante**, c'est-à-dire stockée dans un fichier régulier, avec des programmes pour créer une nouvelle table, y ajouter un élément ou y rechercher un élément. Le but est la simplicité, ainsi, on ne considérera pas la suppression d'entrées.

Il est obligatoire d'utiliser les structures et signatures du fichier `hache.h` fourni, et interdit d'utiliser `malloc` ni des fonctions de lecture/écriture en fichier (`read` etc.) : le code devra impérativement accéder à la table par une projection mémoire (`mmap`).

Le format de fichier est *little endian*[1] : tous les entiers dont il est question dans le format sont signés, sur 32 bits, petit-boutistes. Le format de fichier consiste en 6 zones (ou champs) :

1. Le nombre `n` de cases. Il est fixé à la création et ne peut changer ensuite
2. L'adresse `offset_libre` du début de l'espace libre
3. La taille `taille_libre` de l'espace libre
4. La table de hachage elle-même : un tableau `table` de `n` cases de type`uint32_t`
5. L'espace occupé, s'étendant de la dernière case jusqu'à `offset_libre` exclus
6. L'espace libre, s'étendant de `offset_libre` jusqu'à `taille`

Une **entrée de la table** consiste en

- l'adresse de l'entrée suivante, ou -1 si on est en bout de chaîne
- la longueur `len` de l'objet stocké, en octets
- l'objet stocké lui-même. On peut stocker n'importe quoi même si le TP considérera uniquement des chaînes de caractères

L'espace occupé est occupé par des entrées de table. Une case de la table (en zone 4) pointe une entrée de table (en zone 5), qui elle-même pointe l'entrée suivante, etc. Cela forme une liste chaînée. La liste chaînée de `table[h]` est constituée des objets dont le hashcode vaut `h`. `table[h] = -1` s'il n'y en a aucun.

Exercice 1 :

Écrire la fonction `hache_open`. Elle prend en paramètre un nom de fichier. Elle l'ouvre, obtient sa taille par `fstat`, fait un *mapping* en lecture/écriture sur ce fichier, puis ferme le fichier[2].
Elle renvoie l'adresse du mapping, converti en `struct hache_table *`. Vous noterez que le type `struct hache_table` est incomplet (on ne peut pas l'allouer ni connaître sa taille) mais il est pratique d'utiliser ses champs correspondant aux zones 1 à 4 de la table.
Toutes les fonctions utilitaires à écrire le seront dans un fichier `hache.c`, de sorte que les quatre programmes à écrire utilisent tous ce fichier-là ainsi qu'un autre ne contenant qu'un `main`.

1. à ceux qui diraient que ce n'est pas portable, on peut répondre que le format GIF, sans lequel le Web ne serait pas le Web, est *little endian*
2. on rappelle que les mappings restent utilisables quand le descripteur est fermé, et il est conseillé de le fermer tôt pour éviter d'y faire des `read` ou `write` risquant l'incohérence avec le mapping.

Exercice 2 :

Écrire la fonction `hache_close` qui se contente de fermer le mapping.

Exercice 3 :

Écrire un programme `hache_inspecte` qui prend en paramètre un nom de table existante ; l'ouvre par `hache_open` ; puis affiche

- son nombre de cases,
- la taille de l'espace libre,
- et la taille de l'espace occupé (la calculer par différence entre la dernière case de la table et le début de l'espace libre) ;

et le referme par `hache_close`.
Testez votre programme sur les deux tables données en exemple.

Exercice 4 :

1. implémenter la fonction
   `int32_t hache_search(struct hache_table *boucherie, void *obj, int32_t len)`
   Pour cela il faut
   - calculer le hashcode `h` de `obj`, en utilisant le hashcode de Fowler–Noll–Vo (cf `fnv.c`) modulo le nombre de cases :
     `uint32_t h = fnv(obj, len) % boucherie->n`
   - puis parcourir la liste d'entrées dont la tête est `table[h]` en testant l'égalité de l'objet pointé et de l'objet recherché par `memcmp`.

   `hache_search` renvoie -1 si absence, l'adresse de l'entrée correspondante en cas de présence.
2. Écrire un programme `hache_search` qui prend en paramètre un nom de table existante et une chaîne de caractères, et dit si cette chaîne est présente dans la table ou pas. L'utilisez pour chercher les mots HACHAGE HASHING HACHAT HACHETEUR dans `scrabble.hache`.

Exercice 5 :

Écrire un programme `hache_init` qui prend en paramètre un nom de fichier, un nombre de cases et une taille d'espace libre, puis crée un fichier (en écrasant un éventuel fichier préexistant) correspondant à une table vide : toutes les cases doivent être à -1 et il n'y a aucune entrée. Bien additionner les longueurs des 6 zones (la zone 5 étant pour l'instant de longueur nulle) pour avoir la taille à donner à `ftruncate`.
L'utiliser pour créer une table de 100000 entrées et 10Mo d'espace libre.

Exercice 6 :

1. implémenter la fonction `hache_insert` . On utilise toujours le hash `h=fnv(obj, len) % boucherie->n`.
   Le plus simple est l'insertion en tête : la nouvelle entrée est créée au début de l'espace libre (qui diminue d'autant), et `hable[h]` pointe cette nouvelle entrée qui elle-même pointera l'ancienne tête. Il faut utiliser `hache_search` avant pour refuser de mettre deux fois une valeur dans la table, et dans ce cas on retourne -1 (0 en cas d'insertion réussie).

2. Écrire un programme `hache_insert` qui prend en paramètre un nom de table existante et insère chaque ligne lue sur son entrée standard dans cette table. Les tables en exemple ont été créées par

```
./hache_init mini.hache 3 100
echo Hello | ./hache_insert mini.hache
echo World | ./hache_insert mini.hache
echo Collistion | ./hache_insert mini.hache
./hache_init scrabble.hache 100000 10000000
./hache_insert scrabble.hache < dictionnaire_officiel_scrabble_2012.txt
```

Essayer d'obtenir le même résultat (vos fichiers devraient être identiques à l'octet près à ceux fournis)