BIBLIOTHÈQUES

Motivations

- Certaines parties de codes, utilisées par un grand nombre de développeurs, sont écrites et compilées une seule fois:
  - Rassemblées dans une ou plusieurs bibliothèques
- Les programmes utilisent les fonctions situées dans ces bibliothèques sans connaître (ni avoir) leur code source
  - Seule la signature d'une fonction doit être connue pour pouvoir l'utiliser dans un programme
  - C'est le cas de toutes les fonctions de la bibliothèque standard C (`printf`, `scanf`, `strcmp`, etc)

Deux types de bibliothèques

- **Bibliothèques statiques:**
  - Le code des fichiers objets est **rassemblé** dans une archive (extension `.a`)
  - Lors de l'édition des liens, le code (binaire) d'une fonction est **copié** dans l'exécutable.
  - Lors de l'exécution, la présence de la bibliothèque statique **n'est pas requise**.
- **Bibliothèques dynamiques:**
  - Le code des fichiers objets **subit une édition des liens** et donne lieu à un fichier d'objets partagés (extension: `.so`, `.dll`, `.dylib`, etc)
    - L'extension dépend du système d'exploitation
  - Lors de l'édition des liens, le code (binaire) d'une fonction est **référencé** dans l'exécutable.
  - Lors de l'exécution, la présence de la bibliothèque dynamique **est nécessaire** pour le fonctionnement du programme.

Bibliothèques Statiques

- **Avantages:**
  - L'exécutable devient plus portable et peut fonctionner même sur un système d'exploitation où la bibliothèque n'est pas installée
  - L'exécution est légèrement plus rapide
- **Inconvénients:**
  - La taille de l'exécutable est plus grande
  - Dans certains cas (compilation statique contre la **libc**), on perd la portabilité
    - Arrive quand on compile contre une bibliothèque très récente qui utilise des appels systèmes très récents.
    - Solution: toujours compiler en statique sur un ancien système (et une ancienne version de la bibliothèque) et profiter ainsi de la compatibilité descendante souvent garantie.

Bibliothèques Statiques: Construction

- Commandes:
  `ar rcs libxxxxxx.a fich`$_1$`.o …. fich`$_n$`.o`
  `ranlib libxxxxxx.a`

- Options de `ar`:
  `r` : insère les fichiers objets en remplaçant toute version antérieure
  `c` : crée une nouvelle archive si cette dernière n'existe pas
  `s` : crée un index pour l'archive (nécessaire pour utiliser la bibliothèque)

- `ranlib` : crée un index pour l'archive
  – Théoriquement pas nécessaire si on a spécifié l'option `s` de `ar`
  – Mais dans certains systèmes, elle est nécessaire quand même

- **Remarque importante:** Il est obligatoire que la bibliothèque s'appelle **`lib`**`quelquechose`**`.a`** pour que le compilateur GCC puisse l'utiliser lors de l'édition des liens.

Bibliothèques Statiques: Edition des liens

- Commande:

```
gcc –o nom_executable f1.o … fn.o
 -L/dossier/contenant/la/bibliotheque -lnombib
```

- `-L` : dossier contenant la bibliothèque statique
- `-l` : nom de la bibliothèque sans le préfixe `lib` et sans l'extension `.a`

Bibliothèques Dynamiques

- **Avantages:**
  – La taille de l'exécutable est plus réduite
  – Plusieurs exécutables en cours d'exécution nécessitent une seule instance de la même bibliothèque dynamique présente dans la mémoire
- **Inconvénients:**
  – L'exécutable ne peut s'exécuter qu'en présence de la bibliothèque dynamique sur le système
    - Avec une version compatible
  – L'exécution est un peu plus lente
    - Temps supplémentaire de chargement de la bibliothèque

Bibliothèques Dynamiques: Construction

- Compiler tous les fichiers objets avec `-fPIC` (Position Indemendant Code) pour produire un code binaire dont les sauts utilisent des adresses relatives (plutôt qu'absolues):

```
gcc -c -fPIC fichier.c
```

- Créer la bibliothèque:

`gcc -shared libxx.so fichier`$_1$`.o … fichier`$_n$`.o`

Bibliothèques dynamiques: Edition des liens

- Commande:
  ```
  gcc –o nom_executable f1.o … fn.o
     -L/dossier/contenant/la/bibliothèque -lnombib
  ```
- `-L` : dossier contenant la bibliothèque statique
- `-l` : nom de la bibliothèque sans le préfixe `lib` et sans l'extension `.so`
- Possibilité de faire pointer l'exécutable (en dur) vers une bibliothèque particulière avec `-Wl,-rpath`

  `gcc -Wl,-rpath,'${ORIGIN}/../lib' -o nom …`

  `-Wl,-rpath,'chemin'`: dossier contenant la bibliothèque
  - `${ORIGIN}`: Dossier contenant l'exécutable
  - `${LIB}`: vaut `lib` ou `lib64` selon l'architecture
  - `${PLATFORM}`: Type du processeur (eg. `x86_64`)

Bibliothèques Dynamiques: Exécution

- Lors de l'exécution, le système (**ld-linux.so**) cherche les bibliothèques en suivant l'ordre suivant:
  1. (*ELF Seulement*): Les chemins spécifiés dans la section **DT_RPATH** de l'exécutable, si présente
     - si pas de section **DT_RUNPATH** existante
     - obsolète
  2. Les dossiers pointés la variable d'environnement **LD_LIBRARY_PATH**
     - Pas pour les exécutables **suid/sgid**
  3. (*ELF Seulement*): Les chemins spécifiés dans la section **DT_RUNPATH** de l'exécutable, si présente
  4. Les bibliothèques pointées par `/etc/ld.so.cache`
     - Configurable avec la commande **ldconfig**
  5. Les dossiers `/lib` puis `/usr/lib`
- Plus de détails sur: https://www.systutorials.com/docs/linux/man/8-ld-linux.so/

Commandes Utiles

- `ar`: crée une bibliothèque statique à partir de fichiers objets
- `ranlib`: crée un index pour une bibliothèque statique
- `nm`: liste les symboles (variables globales, fonctions) utilisés dans un exécutables
- `ldd`: liste les bibliothèques dynamiques dont dépend un exécutables et leur chemins sur le système courant.

Variables d'environnement

- **LD_LYBRARY_PATH**: Liste de dossiers dans lesquels le système cherche les bibliothèques dynamiques requises par un exécutables (ordre de priorité 2)
- **LD_PRELOAD**: Liste de bibliothèque dynamiques qui seront chargées avant toute autre bibliothèque dynamique et dont les symboles exposés seront utilisés à la place d'autres ayant le même noms.

  `LD_PRELOAD=libma_bib.so ./mon_exe`

  – Si `libma_bib.so` contient une autre définition de la fonction `malloc` par exemple et que `mon_exec` appelle la fonction `malloc`, c'est la fonction définie dans `libma_bib.so` qui sera appelée.

Exemple: tracer l'appel à `malloc` : trace.c

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void * (*real_malloc) (size_t) = NULL;

void *malloc (size_t size) {

   if (!real_malloc) {
      fprintf (stderr,   "Chercher malloc!\n");
      //printf ou fprintf to stdout cause un seg fault
      real_malloc = dlsym (RTLD_NEXT, "malloc");
      // dlsym cherche le symbole suivant dans la table des symbole ayant le nom donné
   }

   fprintf (stderr,"Tentative d'allocation de %ld octets\n", size);
   void *resultat = (*real_malloc) (size);

   return resultat;
}
```


Exemple: compilation et exécution

- Compilation:

```
gcc -c -fPIC trace.c
gcc -o libtrace.so -shared trace.o
```

- Exécution:

```
LD_PRELOAD=./libtrace.so ./nom_exec
Tentative d'allocation de 64 octets
Tentative d'allocation de 49 octets
…
```


Exercice: `printf` en couleur

- Ecrire une bibliothèque dynamique `libcouleur.so` qui cause l'affichage en rouge de tout texte donné à `printf` dans tout exécutable utilisant cette bibliothèque en `LD_PRELOAD`.
- **Etapes:**
  1. Chercher le code source de `printf` et identifier la fonction à utiliser pour contourner l'exécution de `printf`
  2. Chercher comment cause un affichage en couleur (**ANSI Escape Codes C**)
  3. Ecrire le code nécessaire

TABLES DE HACHAGES
FONCTIONS DE HACHAGE

Tableaux Associatifs

- Collection de paires (clef, valeurs) non ordonnées
  - Les clefs doivent être **uniques** et d'un type **non muable**
  - Les valeurs peuvent être de n'importe quel type
- Expose les fonctions suivantes:
  1. `search(a, k)`: renvoie la valeur v associée à la clef k du tableau associatif a, ou NULL si la clef n'existe pas.
  2. `insert(a, k, v)`: stocke la paire (k,v) dans le tableau associatif a.
  3. `delete(a, k)`: supprime la paire (k,v) associée à la clef k si k existe, ne fait rien si k n''existe pas.

Implantation possibles

- Tableau classique:
  search: O(n)
  insert : O(1)
  delete : O(n)
- Table de hachage:
  search: O(1) dans la plupart des cas
  insert : O(1) dans la plupart des cas
  delete : O(1) dans la plupart des cas
- Dans la suite, nous utilisons des tables de hachage dont les clefs et les valeurs sont des chaines de caractères.
  - Mais la généralisation est possible pour d'autres types

Structure d'une table de hachage

- Paire Clef/Valeur

```
typedef struct {
    char* key;
    char* value;
} ht_item;
```

- Table:

```
typedef struct {
    int size;  // Taille maximale
    int count; // Nombre d'éléments
    ht_item** items;
} ht_hash_table;
```


Allocation

- Paire de clef/valeur

```
ht_item* ht_new_item (const char* k, const char* v) {
    ht_item* i = malloc(sizeof(ht_item));
    i->key = strdup(k);
    i->value = strdup(v);
    return i;
}
```

- Table

```
ht_hash_table* ht_new() {
    ht_hash_table* ht = malloc(sizeof(ht_hash_table));

    ht->size = 53;
    ht->count = 0;
    ht->items = calloc((size_t)ht->size, sizeof(ht_item*));
    return ht;
}
```

Destruction

- Paire de clef/valeur

```
void ht_del_item(ht_item* i) {
    free(i->key);
    free(i->value);
    free(i);
}
```

- Table

```
void ht_del_hash_table(ht_hash_table* ht) {
    for (int i = 0; i < ht->size; i++) {
        ht_item* item = ht->items[i];
        if (item != NULL) {
            ht_del_item(item);
        }
    }
    free(ht->items);
    free(ht);
}
```


Fonctions de hachage

- **Motivation:** pour avoir un temps constant de recherche, on est tenté de pouvoir indexer un tableau par les clefs d'une table de hachage
  - Possible uniquement si les clefs sont d'un type discret
- **Solution:** utiliser une fonction (fonction de hachage) qui, à partir d'une clef, calcule un indice entier et utiliser cet indice pour enregistrer le couple clef/valeur
  - **Problème potentiel:** que faire si deux clefs différentes donnent le même indice?
  - **Début de solution:** essayons d'abord de rendre ce genre d'accidents rare

Pseudo code d'une fonction de hachage

```
function hash(chaine, a, taille_tableau):
    hash = 0
    len = length(chaine)
    for i = 1, ..., len:
        hash += (a ** (len - i - 1))) *
                    char_code(chaine[i])
    hash = hash % taille_tableau
    return hash
```

- Paramètres:

  `chaine` : chaine à hacher

  `a` : nombre premier plus grand que la taille de l'alphabet

  `taille_tableau` : taille de la table de hachage

Implantation en C

```
int ht_hash(const char* s, const int a, const int m) {
    long hash = 0;
    const int len_s = strlen(s);
    for (int i = 0; i < len_s; i++) {
        hash += (long)pow(a, len_s - (i+1)) * s[i];
        hash = hash % m; // Pour éviter le débordement
    }
    return (int)hash;
}
```

Données pathologiques

- Une fonction de hachage idéale distribue les clefs uniformément sur les hash codes
  - Malheureusement, une telle fonction n'existe pas
  - Il existe des fonctions de hachages dites minimales parfaites qui garantissent l'unicité des codes mais leur définition dépend des valeurs des clefs
- Pour toute fonction de hachage, il existe un ensemble d'entrées **pathologique** qui cause des collisions lors du calcul des hash codes.
  - Complexité des fonction de la table passe à O(n).

Traitement des collisions: Chaînage

- Chaque valeur de la table de hachage constitue une liste chainée. Lorsque les éléments entrent en collision, ils sont ajoutés à la liste courante.
  - Insertion: hacher la clé pour obtenir l'indice dans le tableau. S'il n'y a rien, stocker l'objet. S'il y a déjà un élément, ajouter l'élément à la liste liée.
  - Recherche: hacher la clé pour obtenir l'indice dans le tableau. Parcourir la liste chainée en comparant la clé de chaque élément à la clé de recherche. Si la clé est trouvée, renvoyer la valeur, sinon retourner «NULL».
  - Supprimer: hacher la clé pour obtenir l'indice dans le tableau. Parcourir la liste chainée en comparant la clé de chaque élément à la clé de recherche. Si la clé est trouvée, supprimer l'élément de la liste liée. S'il n'y a qu'un seul élément dans la liste liée, mettre un pointeur NULL

Traitement des collisions: Adressage Ouvert

- Quand des collisions se produisent, l'élément en collision est placé dans un autre indice du tableau. Il existe 3 méthodes pour calculer l'indice du tableau dans lequel stocker la valeur en collision.
  - Sondage linéaire
  - Sondage quadratique
  - Double hachage

Sondage Linéaire

- Lorsqu'une collision se produit, l'index est incrémenté et l'élément est placé dans le prochain indice disponible dans le tableau.
  - Insertion: hacher la clé pour obtenir l'indice dans le tableau. S'il n'y a rien, stocker l'objet. S'il y a déjà un élément, incrémenter l'indice plusieurs fois jusqu'au prochain emplacement vide et y insérer l'élément.
  - Recherche: hacher la clé pour obtenir l'indice dans le tableau, incrémenter l'indice jusqu'à ce que la clef de l'élément courant soit égal à la clef de recherche ou à «NULL». Retourner la valeur correspondante.
  - Supprimer: hacher la clé pour obtenir l'indice dans le tableau, incrémenter l'indice jusqu'à ce que la clef de l'élément courant soit égal a la clef de recherche ou a «NULL». Supprimer la valeur correspondante. La suppression de cet élément brise la chaîne, il faut donc réinsérer tous les éléments de la chaîne après l'élément supprimé.

Sondage Quadratique

- Similaire au sondage linéaire, mais au lieu de mettre l'élément en collision dans le prochain indice disponible, on essaye de placer l'élément selon la séquence d'indices suivants : i, i + 1, i + 4, i + 9, i + 16 , ... , où i` est le hach original de la clé.
  - Insertion: hacher la clé pour obtenir l'indice dans le tableau. S'il n'y a rien, stocker l'objet. S'il y a déjà un élément, suivre la séquence de sondage jusqu'à ce qu'un emplacement vide soit trouvé et y insérer l'élément.
  - Recherche: hacher la clé pour obtenir l'indice dans le tableau, suivre la séquence de sondage jusqu'à ce que la clef de l'élément courant soit égale à la clef de recherche ou à «NULL», retourner la valeur correspondante.
  - Supprimer: Comme nous ne pouvons pas dire si l'élément que nous supprimons fait partie d'une chaîne de collisions, nous ne pouvons donc pas supprimer l'élément. Au lieu de cela, nous le classons simplement comme supprimé.

Double Hachage

- Le double hachage vise à résoudre les problèmes des sondages linéaire et quadratique.
- Pour le mettre en place, nous utilisons une deuxième fonction hash pour choisir un nouvel indice pour l'élément. Après i collisions:

```
index = hash_a(string) + i * hash_b(string) % taille
```

- Pour éviter que `hash_b` retourne 0

```
index = (hash_a(string) + i * (hash_b(string) + 1)) % taille
```


Implantation en C

```
int ht_get_hash(const char* s,
                const int taille,
                const int attempt)
{
    const int hash_a = ht_hash(s, HT_PRIME_1, taille);
    const int hash_b = ht_hash(s, HT_PRIME_2, taille);

    return (hash_a + (attempt * (hash_b + 1))) % taille;
}
```


Insertion: Version 1

```
void ht_insert(ht_hash_table* ht,
               const char* key,
               const char* value)
{
    ht_item* item = ht_new_item(key, value);
    int index = ht_get_hash(item->key, ht->size, 0);
    ht_item* cur_item = ht->items[index];
    int i = 1;
    while (cur_item != NULL) {
        index = ht_get_hash(item->key, ht->size, i);
        cur_item = ht->items[index];
        i++;
    }
    ht->items[index] = item;
    ht->count++;
}
```

Recherche: Version 1

```
char* ht_search(ht_hash_table* ht, const char* key)
{
    int index = ht_get_hash(key, ht->size, 0);
    ht_item* item = ht->items[index];
    int i = 1;
    while (item != NULL) {
        if (strcmp(item->key, key) == 0) {
            return item->value;
        }
        index = ht_get_hash(key, ht->size, i);
        item = ht->items[index];
        i++;
    }
    return NULL;
}
```


Supression: Version 1

```
static ht_item HT_DELETED_ITEM = {NULL, NULL};

void ht_delete(ht_hash_table* ht, const char* key) {
    int index = ht_get_hash(key, ht->size, 0);
    ht_item* item = ht->items[index];
    int i = 1;
    while (item != NULL) {
        if (item != &HT_DELETED_ITEM) {
            if (strcmp(item->key, key) == 0) {
                ht_del_item(item);
                ht->items[index] = &HT_DELETED_ITEM;
            }
        }
        index = ht_get_hash(key, ht->size, i);
        item = ht->items[index];
        i++;
    }
    ht->count--;
}
```


Insertion: Version 2

```
void ht_insert(ht_hash_table* ht,
               const char* key,
               const char* value) {
    // Même Code…
    while (cur_item != NULL &&
           cur_item != &HT_DELETED_ITEM)
    {
        // Même Code…
    }
    // Même Code…
}
```


Recherche: Version 2

```
char* ht_search(ht_hash_table* ht,
                const char* key)
{
    // Même Code…
    while (item != NULL) {
        if (item != &HT_DELETED_ITEM) {
            if (strcmp(item->key, key) == 0) {
                return item->value;
            }
        }
        // Même Code…
    }
    // Même Code…
}
```

Mise à jour d'un couple clef/valeur

- La table de hachage précédente ne supporte pas la mise à jour de la valeur d'une clé.
  - Une insertion de deux éléments avec la même clef sera considéré comme une collision.
  - Le deuxième élément sera inséré dans le prochain indice disponible.
- Lorsque l'on recherchera la clé, la clé d'origine sera toujours trouvée
  - Impossible d'accéder au deuxième élément.
- Possible de corriger cela en modifiant `ht_insert` pour faire en sorte de supprimer l'élément précédent et insérer le nouvel élément à son emplacement.

Insertion: Version 3

```
void ht_insert(ht_hash_table* ht,
               const char* key,
               const char* value)
{
    // Même Code…
    while (cur_item != NULL) {
        if (cur_item != &HT_DELETED_ITEM) {
            if (strcmp(cur_item->key, key) == 0) {
                ht_del_item(cur_item);
                ht->items[index] = item;
                return;
            }
        }
        // Même Code…
    }
    // Même Code…
}
```


Redimensionnement d'une table de hachage

- La table précédente est de taille fixe.
- Plus on insère d'éléments, plus la table se remplit, plus cela génère les problèmes suivants :
  1. La performance de la table de hachage diminue avec des taux élevés de collisions
  2. La table ne peut stocker qu'un nombre fixe d'éléments.
     - Si nous essayons de stocker plus que cela, la fonction d'insertion échouera.
- Pour pallier à cela, nous pouvons augmenter la taille du tableau de l'article quand il est trop rempli. Par exemple:
  - Si le taux de remplissage est > 0.7, on l'agrandit
  - Si le taux de remplissage est < 0.1, on le diminue
- Pour redimensionner, nous créons une nouvelle table de hachage ou l'on insère tous les éléments non supprimés.

Caractéristiques de la nouvelle table

- La nouvelle taille doit être un nombre premier correspondant à peu près à deux fois (agrandissement) ou à la moitié de la taille actuelle (réduction).
- Trouver la nouvelle taille de tableau n'est pas trivial. Mais n'est pas difficile Pour ce faire nous :
  - Définir une taille initiale de base (commençons par 50)
  - Définir la taille réelle comme le premier nombre premier plus grand que la taille de base.
  - **Agrandir**: doubler la taille de base et trouvons le premier nombre premier plus grand
  - **Réduction**: réduire la taille de moitié et trouvons le premier plus grand

Allocation d'une table: Version 2

```
ht_hash_table* ht_new_sized(const int base_size) {
    ht_hash_table* ht = malloc(sizeof(ht_hash_table));
    ht->base_size = base_size;

    ht->size = next_prime(ht->base_size);

    ht->count = 0;
    ht->items = calloc((size_t)ht->size,
                       sizeof(ht_item*));
    return ht;
}

ht_hash_table* ht_new() {
    return ht_new_sized(HT_INITIAL_BASE_SIZE);
}
```


Redimensionnement (1/2)

```
void ht_resize(ht_hash_table* ht, const int base_size) {
    if (base_size < HT_INITIAL_BASE_SIZE) {
        return;
    }
    ht_hash_table* new_ht = ht_new_sized(base_size);
    for (int i = 0; i < ht->size; i++) {
        ht_item* item = ht->items[i];
        if (item != NULL && item != &HT_DELETED_ITEM) {
            ht_insert(new_ht, item->key, item->value);
        }
    }
    ht->base_size = new_ht->base_size;
    ht->count = new_ht->count;

    const int tmp_size = ht->size;
    ht->size = new_ht->size;
    new_ht->size = tmp_size; // Nécessaire pour bien désallouer
    ht_item** tmp_items = ht->items;
    ht->items = new_ht->items;
    new_ht->items = tmp_items; // Nécessaire pour bien désallouer
    ht_del_hash_table(new_ht);
}
```


Redimensionnement (2/2)

```
void ht_resize_up(ht_hash_table* ht)
{
    const int new_size = ht->base_size * 2;
    ht_resize(ht, new_size);
}

void ht_resize_down(ht_hash_table* ht)
{
    const int new_size = ht->base_size / 2;
    ht_resize(ht, new_size);
}
```


Insertion: Version 4

```
void ht_insert(ht_hash_table* ht,
               const char* key,
               const char* value)
{
    const int load =
       ht->count * 100 / ht->size;
    if (load > 70) {
        ht_resize_up(ht);
    }
    // Même Code…
}
```

Suppression: Version 2

```
void ht_delete(ht_hash_table* ht,
               const char* key)
{
    const int load =
       ht->count * 100 / ht->size;
    if (load < 10) {
        ht_resize_down(ht);
    }
    // Même Code…
}
```