LINFO1252 ▷ Systèmes Informatiques
Rapport Projet 0

Bellière Arnaud | 62652000
Cheffert Théo | 25882000

1 Stratégie

Tout d'abord, avant de coder nos différentes fonctions, nous avons réfléchi à la structure de notre mémoire. Elle inclut une structure de métadonnées, qui se compose de :

- `int free` : indique si la zone est libre (1) ou occupée (0).
- `size_t size` : taille du bloc.
- `struct metadonnee* next` : pointeur vers le prochain bloc.
- `struct metadonnee* prev` : pointeur vers le bloc précédent.

Cependant, nous avons constaté que notre structure de métadonnées occupait trop de place inutilement. Nous pouvons donc supprimer `size`, puisqu'il peut être recalculé en utilisant le pointeur `next`, ce qui améliore l'efficacité en termes de temps et d'espace (une donnée à stocker en moins).

Nous avons aussi envisagé de retirer le pointeur `prev`, qui est surtout utile pour fusionner deux blocs libres adjacents lors de la libération de mémoire `my_free`. Supprimer ce pointeur réduirait la taille de la structure mais augmenterait le temps d'exécution de la fonction `my_free`. En effet, celle-ci devrait alors parcourir les pointeurs `next` jusqu'à retrouver le bloc actuel et vérifier si le bloc précédent est libre, ce qui ferait passer la complexité de `my_free` de $\mathcal{O}(1)$ à $\mathcal{O}(n)$. Nous n'avons donc pas opté pour cette méthode.

2 Structure

Notre mémoire est constituée de métadonnées, de 4 octets pour le flag `free` et 8 octets pour les pointeurs, ainsi que les données en elles-mêmes de taille quelconque puisque nous n'avons pas de pas d'alignement.

Notre bloc de métadonnées contient un entier `free` (indiquant si le bloc suivant est libre (1) ou non (0)), un pointeur vers le prochain bloc de métadonnées `next`, et un pointeur vers le bloc de métadonnées précédent `prev`. Cette structure nous permet d'avoir une complexité temporelle pour `my_malloc` de $\mathcal{O}(n)$ (parcours de n blocs) dans le pire des cas, et pour `my_free` de $\mathcal{O}(1)$ ou, plus précisément, $\Theta(1)$.

3 Fonctionnement de l'algorithme

`struct metadonnee` : int libre, struct metadonnee *next , struct metadonnee *prev.

`init()` : initialise la première métadonnée au début de `My_Heap`, avec free = 1 ainsi que `next` et `prev` à *null*. Cela se fait en complexité temporelle de $\mathcal{O}(1)$.

`my_malloc(size_t size)` : parcours les pointeurs `next` à partir de `My_Heap` jusqu'à trouver un emplacement libre de taille suffisante. Si le bloc trouvé n'est pas de taille parfaite, donc plus grand que la taille demandée (`size + sizeof(metadonnee)`), on scinde le bloc en 2. Les pointeurs `next` et `prev` sont mis à jour en fonction de la position du bloc alloué (suivant ou précédent), tout en initialisant les pointeurs de ce nouveau bloc alloué. La complexité de cette fonction est en $\mathcal{O}(n)$ pour un parcours de n blocs en mémoire.

`my_free(void *pointer)` : libère le bloc pointé et fusionne le/les blocs adjacents, s'ils sont libres, grâce à `prev` et `next`. La complexité de notre `my_free` se fait en $\Theta(1)$.

4 Diagrammes

Dans nos diagrammes, le bloc **FREE** est de 4 octets (*int*) et les blocs pointeurs **NEXT** et **PREV** sont de taille 8 octets.

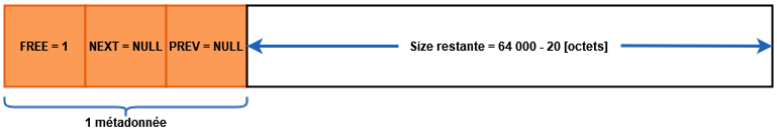

FIGURE 4.0.1 – Initialisation de la mémoire. Vierge de toute exécution de `malloc` ou `free`.

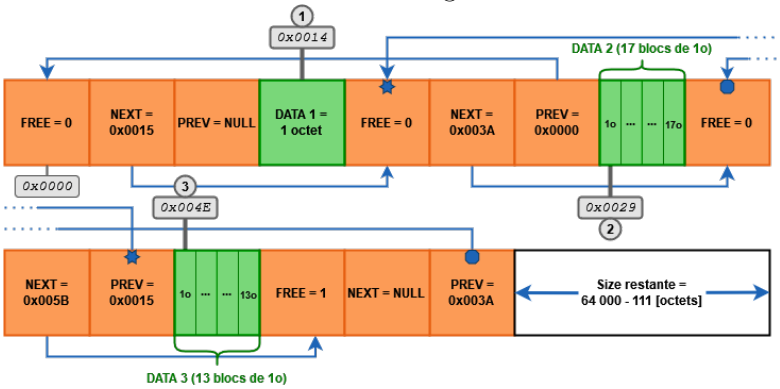

FIGURE 4.0.2 – État de la mémoire après un `malloc` de 1, 17 et 13 octets. Les adresses chiffrées sont les retours respectifs des 3 mallocs.

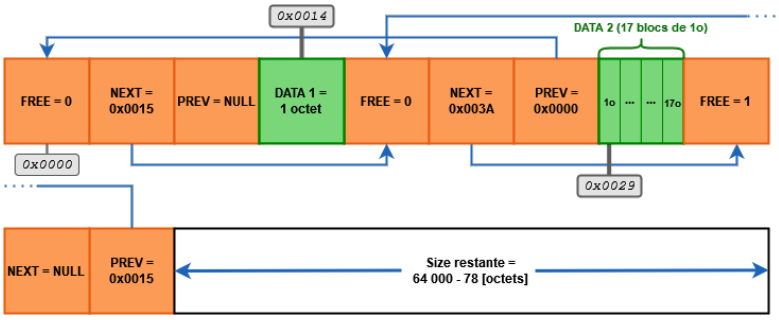

FIGURE 4.0.3 – État de la mémoire après exécution de `free`