Allocateur Mémoire
AISE

Réalisé par :
Hamza ELBARAGHI
Sofiane BOUZAHER
Atef DORAI

15 mars 2020

Sommaire

1 Introduction

Le but de ce projet est de développer un allocateur mémoire capable d'être lancé en remplacement de l'allocateur système pour n'importe quel type d'application ou de bibliothèque. chaque Allocateur de memoire contient les fonctions suivantes :

1.1 malloc

La fonction standard **malloc** a comme paramétre la taille de memoire à alloué en bytes et comme retour un pointeur vers la zone allouée ou 0 si la taille entré est 0. En notant que la memoire alloué n'est pas initialisé.

1.2 calloc

La fonction standard **calloc ()** prend en entrée deux parametres 'nmemb' et 'size' pour allouer de la mémoire un tableau de 'nmemb' éléments et de taille 'size' octets chacun ,et renvoie un pointeur sur la mémoire allouée on dirait malloc mais avec un mise à zéro de la mémoire. Si nmemb ou size est 0, calloc () renvoie soit NULL, soit une valeur de pointeur unique qui peut plus tard réussir entièrement passé à free ().

1.3 realloc

La fonction standard **realloc ()** prend comme parametres d'entreé un 'pointeur' et un 'size' afin de modifier la taille du bloc de mémoire pointé par 'ptr' en 'size' octets. Si la nouvelle taille est plus grande que l'ancienne taille, la mémoire ajoutée ne sera pas initialisé. Si ptr est NULL, alors l'appel est équivalent à malloc (size), si la size est égale à zéro et ptr n'est pas NULL, alors l'appel est équivalent à free(ptr). Si la zone pointée a été déplacée, un free (ptr) est effectué.

1.4 free

la function standard **free** libere l'espace memoire , un appel précédent à malloc (),calloc () ou realloc () est obligatoire pour retourne le ptr qui sera utiliser par free. Si ptr est NULL, aucune opération n'est effectuée.

2 Déscription du code

2.1 Recyclage de blocs et metadonnées

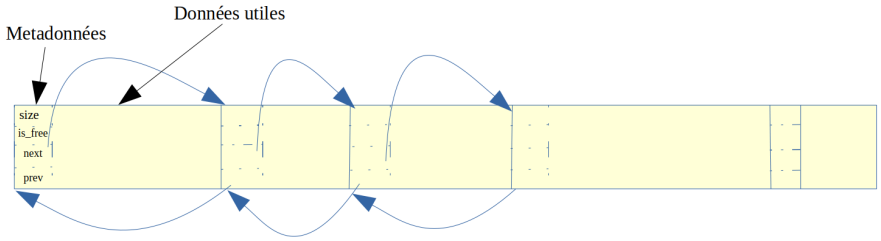

FIGURE 1 – Recyclage de blocs

Pour notre implémentation, nous avons choisi une structure de donnée contenant les données utiles pour les blocs alloués/désalloulés, à savoir : la taille du bloc, son état (alloué ou désalloué), un pointeur vers le bloc suivant, et un poineur vers le bloc précédent.

```
typedef struct bloc_meta_data{
        size_t size     ;  // contient la taille demandé par l'utilisateur.
        char   is_free  ;  // either the bloc is freed(1) or not(0) .
        struct bloc_ *next ;
        struct bloc_ *prev ;
} meta_data ;


typedef struct bloc_ {
        meta_data m_data ;
        void *addr_      ;
} bloc ;
```

FIGURE 2 – Structure metadonnées

Metadonnées : Les métadonnées représentent des informations sur le bloc alloué, et sur le bloc suivant et précédant ; cela permet d'optimiser le nombre d'appel à la fonction mmap : un appel au malloc ne fait pas nécessairement un appel à mmap.

Recyclage de blocs :

- **Première allocation :** première mapping = appel mmap d'une page de 4KB
- **Prochaines allocations :** dépend de la taille demandée :
  - Si la taille demandée est supérieure à 4KB (taille d'une page) => un autre appel à mmap est nécessaite
  - Si la taille demandée est inférieur à 4KB => On parcours les pages déjà aloués et on cherche le premier bloc free et dont la size est supérieure ou égale à la taille demandé, on récupère la zone suffisante dans ce bloc, et on modifie les données de métadonnées .
- **Reallocation :**
  Pour réallouer une zone, on vérifie si le bloc suivant (s'il existe) est suffisant, si c'est le cas on merge les deux blocs et on modifier les informations sur les métadonnées des deux blocs, sinon on fait appel à malloc, et on copie les données initiales dans l'adresse retourné.
- **Free :**
  On modifie les métadonnées de l'adresse à libéré ; on met is_free = 1 pour signaler que cet zone est libre pour les prochaines malloc .
  En plus, on merge avec le bloc suivant s'il est free pour construire un seul bloc free, de meme pour le bloc précédent .
- **Alignement :**
  Avant d'allouer un blocon aligne d'abord la taille demandé en plus de la taille des métadonnées (sizeof(meta_data)) sur le mot machine pour optimiser les accès mémoire en lecture et écriture.
- **Multithreading :**
  Pour une application qui lance plusieurs threads, et chaque thread essaye d'allouer une zone mémoire, il faudra protéger les zones allouer pour chaque thread, pour cela nous avons ajouter une mutex dans notre implémentation, et à chaque fois on accéde à une section critique on la protége avec les deux fonctions : pthread_mutex_lock et pthread_mutex_unlock

3 Les tests

Pour tester notre bibliothèque nous avons imlémenté une boucle dont on fait appel à malloc/realloc/free et on mesure on le temps écoulé :

Premièrement une boucle de plusieurs malloc de tailles aléatoires, puis une autre boucle pour free des zones aléatoires parmi les zones alloués précédément. et en suite une boucle pour réalloc .

4 Conclusion

Nous avons implémenté notre propre bibliothèque d'allocation mémoire en utilisant le principe de recyclage de bloc, et la méthode : implicit free list pour allouer un bloc parmi les blocs désalloués, ainsi qu'on a implémenté pour la fonction free une technique qui permet de rassembler les free zones qui sont succcessives, en plus on a protégé les zones critique par des mutex pour mettre notre bibliothèque thread-safe .

Nous pouvons améliorer notre implémentation, par exemple en adoptant la méthode : best fit free list qui permet de chercher pour une allocation donnée le bloc optimal, ou la méthode segregated free list qui cherche le bloc optimal parmi une liste de blocs triés par tailles, ce qui permet de minimiser le temps de recherche de blocs .

Par contre, nous avons rencontrés des difficultés au niveau de l'intreposition de notre bibiothèque avec la libc .

Références

[1] UNIX. Le manuel du programmeur.

[2] Université Utah. Memory management with mmap. https ://my.eng.utah.edu/ cs4400/malloc.pdf.