Rapport TP Mémoire (Mnemosyne)

Moissonnier Shane et Bournat-Querat Jean-Edouard

Octobre 2021

Contents

1 Résumé

Nous avons implémenté durant ce projet les fonctions **mem_init**, **mem_show**, **mem_alloc**, **mem_free**, **mem_realloc** ainsi que les trois stratégies d'allocations **mem_first_fit**, **mem_best_fit** et **mem_worst_fit**. Nous avons testé notre implémentation avec les tests fournis, ainsi qu'avec des tests personnels, cependant nous n'avons pas réussi à utiliser les wrappers fournis pour utiliser notre implémentation avec les commandes ls et ps.

2 Principes d'implémentation

Au cours de ce projet nous avons mis en place deux versions différentes de l'allocateur mémoire, dans cette section nous expliquerons les différences entre ces deux versions et les perspectives d'optimisation qui en découlent.

2.1 Première version de l'allocateur:

Pour commencer nous avons décidé d'implémenter deux listes chaînées (une liste chaînée des **zones libres** et **zones occupées**).
Ces deux listes partageaient la même structure de bloc (voir **Figure 1**). La liste chaînée des zones occupées était utilisée pour détecter et vérifier les libérations de mémoire à des adresses invalides ou les doubles libérations de blocs.
Ce choix d'implémentation impliquait des coûts en matière de performance (voir section **Performance**).
De plus utiliser deux listes chaînées nous obligeait à réordonner à chaque allocation et libérations

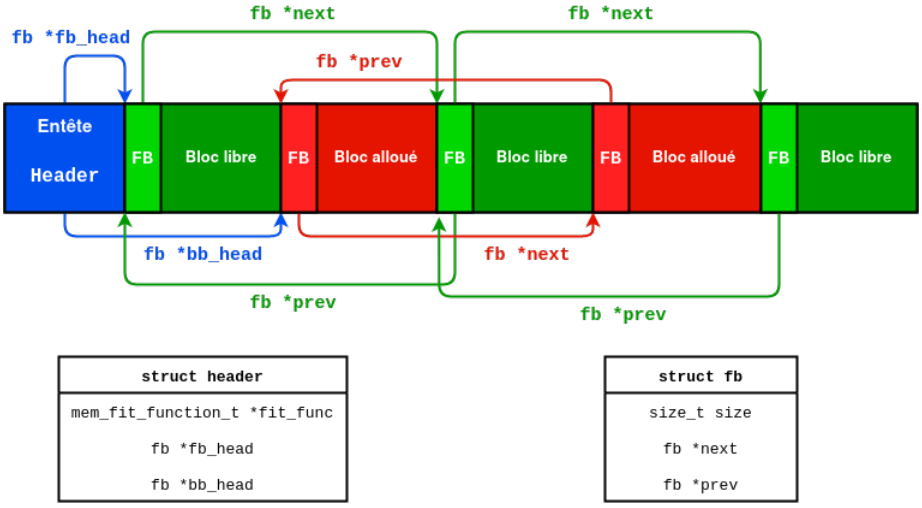

Figure 1: Schéma d'organisation de la mémoire (Première version allocateur)

nos éléments des deux listes pour éviter les erreurs de chaînage. Cela rajoutait un effort conséquent pour le débogage effectif du programme.

2.2 Version finale de l'allocateur:

Afin d'optimiser notre allocateur nous avons revu notre implémentation. Dans un premier temps nous avons séparé les entêtes des deux types de bloc (libre et occupé) en deux structures (struct fb et struct bb). Les entêtes des zones libres contiennent leur taille totale (taille de l'entête + taille de la zone utilisateur) et le prochain élément de la liste chaînée. Les entêtes des zones occupées quant à elles contiennent aussi leur taille totale et un pointeur donnant l'adresse de début du bloc occupé. Cet ajout nous permet de supprimer la liste chaînée des zones occupées, la vérification des libérations de mémoire invalide se faisant de la manière suivante :
Lors de la libération d'une adresse **a**:

1. On vérifie que **a - tailleEnteteBb** se situe dans l'espace mémoire de notre tas.
2. On récupère la valeur du pointeur ptr contenu dans l'entête bb (du bloc (supposé) se situant à l'adresse **a - tailleEnteteBb**) et on verifie que cette valeur est bien égale à l'adresse de début du bloc (**a - tailleEnteteBb**) (voir **Figure n°2**).

Concernant le reste du programme nous avons suivi les instructions données en cours. Lors de l'allocation nous vérifions que la taille restante de la zone libre peut contenir un entête de zone libre plus un entier, si c'est le cas on scinde notre bloc sinon on agrandit le bloc que l'on est en train d'allouer, enfin nous réordonnons notre liste chaînée.

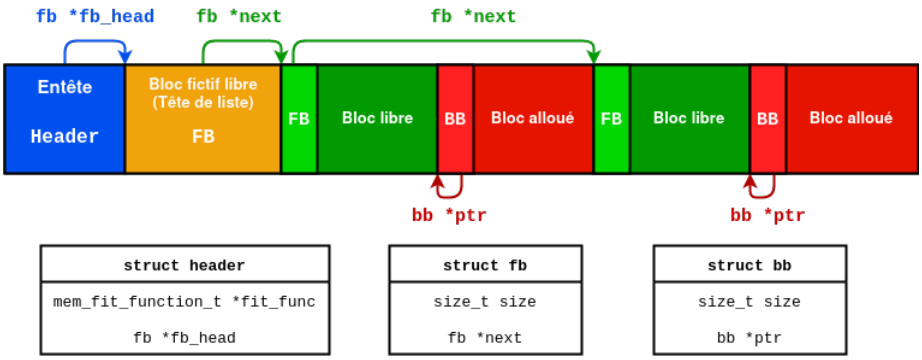

Figure 2: Schéma d'organisation de la mémoire (Version finale allocateur)

3 Performances

Dans cette section nous réaliserons une comparaison des différentes politiques d'allocations implémentées, puis nous donnerons une estimation du coût mémoire des structures choisies, enfin nous mettrons en évidence les différences de performances pour nos deux versions d'allocateurs.

3.1 Comparaison des politiques d'allocations

Figure 3: Cas n°1

Figure 4: Cas n°2

Pour chacun des cas nous faisons abstraction de l'existence des entêtes:

Cas où la stratégie best_fit créer une fragmentation minimale par rapport aux autres stratégies:

Dans le cas n°1 si l'utilisateur demande une taille de bloc de **15 octets** alors la stratégie **first fit** choisira la première zone **A**. Si dans un second temps l'utilisateur demande une taille de **34 octets** alors l'allocateur ne pourra répondre à sa demande (**33 octets** restants dans la zone **A** et **16** dans le B). Sinon dans le cas où la stratégie choisie serait le **worst fit** alors nous nous retrouverions face au même problème. La stratégie **best fit** est donc meilleure dans ce cas.

Cas où la stratégie worst_fit créer une fragmentation minimale par rapport aux autres stratégies:

Dans le cas n° 2 si l'utilisateur demande une taille de bloc de **11 octets** alors la stratégie **first fit** choisira la première zone **A**. Si dans un second temps l'utilisateur demande une taille de **15 octets** l'allocateur choisira la zone **B** or si l'utilisateur refait une demande de **12 octets** alors l'allocateur ne pourra pas répondre à sa demande (il reste 5 octets dans la zone A et 10 dans la zone B). Si la stratégie était **best fit** on se retrouve dans la même situation. Dans ce cas le **worst fit** est une meilleure solution.

3.2 Estimation du coût mémoire des structures choisies

Pour réaliser cette estimation nous avons légèrement modifié le test fourni **"test_frag"** afin de récupérer le nombre de zones libres crées. Puis nous avons lancé plusieurs exécutions (une vingtaine). Pour un tas mémoire de taille **128 000 octets** et avec une stratégie par défaut de type first fit nous avons déterminé que les structures de gestion (entêtes) occupés en moyenne **19%** de l'espace total de notre tas mémoire.

3.3 Comparaison des deux versions d'allocateur mémoire:

Pour montrer les différences de performance entre nos deux versions d'allocateur nous analysons les principes d'implémentations choisis :

Concernant la première version le choix d'avoir deux listes chaînées impliquait l'existence de deux boucles **while** pour chaque fonction d'allocation et de libération (**mem_alloc** et **mem_free**). Ce qui en matière de performance pouvait se révéler problématique. Afin de remédier à ce problème nous avons eu l'idée d'implémenter une double liste chaînée en insérant dans la structure d'entête des blocs (**occupés** et **libres**) un élément précédent. Bien que performante cette solution nous confronte au compromis **performance/coût mémoire**, nos entêtes étant de ce fait plus grand pour chaque bloc.

Dans la version finale de notre allocateur la suppression de la liste chaînée des zones occupées ainsi que l'utilisation d'un élément fictif au début du tas pour notre liste chaînée de zone libre nous permettent de supprimer la quasi-totalité des boucles while de notre première version. Dans le cas de la reconstruction de notre liste chaînée de zones libres après libération (**mem_free**) d'un bloc précédemment alloué nous utilisons une seule boucle while afin de récupérer la zone libre précédant le bloc que l'on vient de libérer (dans le cas de la fonction **mem_alloc** les différentes stratégies

nous retournent l'élément précédent la zone libre trouvée).

4 Structure du code

Le code du projet est structuré de la manière suivante:

1. **mem.h**: Contient les différentes structures utilisées par l'allocateur (l'entête de début de tas, l'entête des zones libres et occupées)
2. **mem.c**: Contient toutes les fonctions d'allocations, de libérations, de stratégies et fonctions annexes (**mem_alloc**, **mem_free**, **mem_realloc** etc...).
3. **Dossier cutest**: Contient la librairie CuTest ainsi que les différents tests unitaires (**AllTests.c**).

5 Tests effectués

Pour faciliter l'écriture et l'implémentation de nos tests nous avons utilisé la librairie CuTest (librairie de test unitaire). Voici la liste des différents tests réalisés par nos soins:

Tests **mem_alloc** et **mem_free**:

1. Allocation d'une zone mémoire avec écriture de données puis libération de la zone (vérification effacement données utilisateurs)
2. Allocation d'une zone de taille négative
3. Allocation d'un nombre n de blocs puis libérations de k blocs au milieu de ces allocations
4. Allocation d'un nombre n de blocs puis libérations de k blocs à la fin de ces allocations

Tests **mem_realloc**:

1. Allocation de trois blocs mémoire de taille k puis reallocation du premier bloc avec une taille supérieure à sa taille initiale
2. Allocation de trois blocs mémoire de taille k puis reallocation du premier bloc avec une taille inférieure à sa taille initiale
3. Allocation de trois blocs mémoire de taille k puis reallocation du premier bloc avec une taille identique à sa taille initiale

6 Compilation et exécution

Afin de compiler le projet :

```
$ make clean all
$ ./memshell
```

Pour lancer tous les tests :

```
$ make test
```

7 Extensions

Voici les différentes extensions implémentées dans ce projet:

1. La fonction **mem_realloc** qui suit la spécification de la fonction realloc de la libc.
2. Un système d'effacement des données utilisateurs après libérations d'un bloc mémoire pour éviter l'accès à des données supprimées (fonction **mem_remove_data**).
3. Prise en charge de certains cas d'utilisation erroné de l'allocateur (accès invalide et multiple libération d'un même bloc).
4. Une nouvelle interface plus lisible pour afficher l'état de la mémoire (commande **"s"** dans **memshell.c**) fortement inspirée du simulateur d'allocateur fournit.