TP4 – API – Allocateur Mémoire

Introduction :

Dans ce TP, nous devions implémenter un allocateur mémoire réaliste. Ce dernier, dont la structure (conseillée) nous était fournie dans les fichiers originaux, devait fonctionner à la demande, dans un certain espace mémoire prédéfinit. Nous devions donc implémenter les fonctions d'allocation et de libération de la mémoire allouée, en plus de certaines politiques d'allocation et d'autres extensions si le temps nous le permettait.

Implantation :

Nous avons donc travaillé comme le conseillaient les enseignants, c'est à dire avec une répartition par zones dans la mémoire, chaque zone serait représentée dans la mémoire par un bloc de méta-données ainsi que de son contenu (sauf pour les méta-données globales qui tiennent toujours en début de mémoire), voici un schéma récapitulatif :

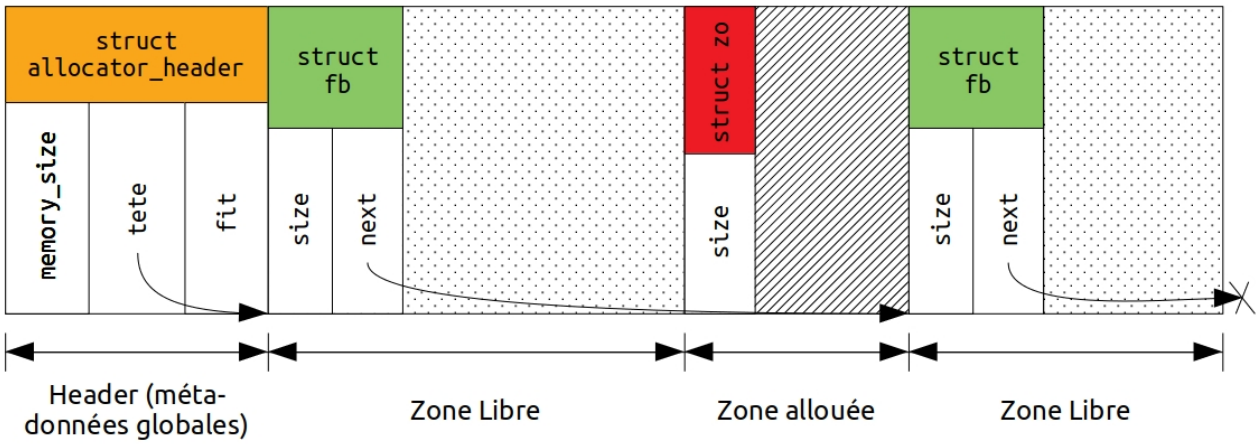

Illustration 1: Schéma de l'état de la mémoire dans une situation possible

Comme nous pouvons le voir, dans notre code C, nous avons implanter les méta-données globales en début de mémoire, à travers une structure C appelée `allocator_header`, contenant 3 champs :

- `memory_size`, contenant la taille de la totalité de la zone mémoire.
- `tete`, qui est un pointeur vers la premier zone libre de la mémoire s'il en existe.
- `fit`, qui est un pointeur vers la fonction qui définit la politique d'allocation utilisé par l'exécution en cours du programme.

Nous avons ensuite deux types de zones distinctes. Les zones libres, `fb` (pour freeblock), contenant des méta-données dans une structure C :

- `size`, la taille de la zone-libre (y compris les méta-données).

- `next`, l'adresse de la prochaine zone libre, si elle existe (sinon on illustre le cas échéant par une croix la prochaine zone inexistante (pointeur vers `NULL` en C)).

Et les zones occupées (ou allouées), contenant « des » méta-données, avec le champ `size` de la structure C « `zo` », ainsi que les données stockées dans cette zone (partie hachurée).

Dans l'ensemble nous avons donc suivis l'implantation prédéfinis par l'enseignement, mais nous avons **quelques différences**, comme par exemple le champs « `fit` » du header, ou la présence d'une structure dans une zone occupée. **Ces deux choix se justifient**, le premier car nous ne voulions pas plus d'une variable globale dans notre programme (celle de l'adresse mémoire principale). Et la deuxième car nous pensions à la maintenabilité du code. En effet, si les méta-données de la zone occupée viendraient à grossir, on pourrait simplement rajouter un champs dans la structure `zo`, au lieu de réécrire chaque ligne de code concernant ces zones.

Fonctionnalités et limites :

Notre allocateur mémoire peut donc :

- Allouer une zone mémoire à partie d'une taille en octet demandée.
- Libérer une zone mémoire précédemment allouée.
- Changer de politique d'allocation pour pouvoir définir quelle zone est à allouer plutôt qu'une autre.
- Être maintenu facilement, de part son implantation et son interface.

En revanche, il ne peut :

- Gérer les bonnes écritures dans ses zones mémoire (une fois une adresse allouée on ne vérifie pas ce que l'utilisateur va entreprendre dans notre mémoire).
- Gérer les libérations sur des bonnes adresses de zones occupées (l'utilisateur peut potentiellement libérer une adresse non-allouée).

Tests :

Nous avons réaliser plusieurs tests, dans lesquels l'utilisateur n'utilise que la politique « first fit », et n'exerce que des allocations, ainsi que des libérations sur les bonnes adresses, il en est ressorti à travers ces tests que l'implémentation des fonctions de l'interface mem.h était correcte et fonctionnelle. Ces tests peuvent être lancés et observés avec la commande `make tests` depuis le dossier racine du TP, une description de chacun des tests est lisible au lancement.

Conclusion :

Dû au manque de temps et aux autres TP de cette fin de semestre, nous n'avons pas eu le temps d'implémenter toutes les politiques d'allocation, ni de mettre en place d'extensions particulières, nous nous sommes donc contenter d'implémenter le nécessaire et de le tester. Nous considérons cela dit ce TP comme une bonne expérience et regrettons de ne pas pouvoir aller plus loin tout de suite.