Allocations mémoire

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `alloc` contenant une source en C et un makefile :

```
curl cdn.sci-phy.org/mp2i/tp9-alloc.tgz | tar xvz
```

Le fichier à compléter est le fichier `alloc.c`. On y trouvera toutes les fonctions que l'on souhaite créer, mais beaucoup sont incomplètes (mention « `A COMPLETER !` »). Pour ne pas gêner la compilation, les fonctions incomplètes qui doivent retourner un pointeur contiennent « `return NULL;` ». Bien évidemment, cette ligne pourra être modifiée!

Il s'agit du seul fichier dans ce projet, mais le répertoire contient également un `makefile` pour compiler le compiler comme d'habitude (On rappelle les commandes « `make all` » pour compiler le programme, « `make run` » pour le compiler et l'exécuter si la compilation a réussi).

1.2 Objectifs

Le but de cette séance est d'étudier le mécanisme d'allocation et de libération de la mémoire (et de s'entraîner à utiliser des listes chaînées!)

Il n'est pas possible de prendre simplement le contrôle de la mémoire, aussi va-t-on ruser un peu. Le programme va demander au tout début une « grande » quantité de mémoire (de quoi contenir 1024 entiers). On va ensuite s'efforcer d'écrire deux fonctions `ialloc` et `ifree` permettant de réserver et de libérer une partie de cette mémoire.

La fonction `ialloc` devra s'efforcer de trouver une portion contiguë des 1024 entiers qui n'a pas encore été réservée, et retourner l'adresse de début de cette portion. La zone ainsi allouée ne pourra plus être fournie à un autre demandeur tant qu'elle n'a pas été libérée par un appel à `ifree`.

Ce TP est particulièrement difficile, car il fait intervenir de nombreux pointeurs (et parfois des pointeurs de pointeurs) et des insertions et suppressions dans des listes chaînées. Il faut du temps pour être à l'aise avec de telles structures. Par ailleurs, il est difficile de trouver ce qui cloche, car les erreurs conduiront souvent à un comportement indéfini (*undefined behavior*) du programme, et donc des erreurs qui semblent apparaître et disparaître au hasard.

Pour l'instant, efforcez-vous surtout de comprendre au mieux ce que l'on cherche à faire et comment. n'hésitez pas à faire des dessins pour écrire vos algorithmes. Même si traditionnellement en C on utilise plutôt des itérations (**`for`** et **`while`**), il est probable qu'ici des approches récursives puissent vous faciliter la vie. Essayez au moins d'envisager les deux possibilités avant de tenter de programmer quoi que ce soit.

1.3 Blocs

Un « *bloc* » désignera dans la suite un morceau contiguë de ces 1024 entiers. Pour pouvoir gérer de tels blocs, on définit le type « **`struct`** `block` » suivant :

```c
struct block {
    int* p_start;
    int size;
    struct block* p_next;
};
```

Dans cette structure, le champ `.p_start` désigne l'adresse du premier des entiers du bloc, et `.size` le nombre d'entiers consécutifs qu'il contient.

L'ensemble des blocs disponibles pour une allocation, de même que l'ensemble des blocs alloués à un instant t, seront rangés dans des listes chaînées de blocs. On a donc ajouté un champ `.p_next` qui permettra, pour tout bloc, de désigner le bloc suivant dans la liste chaînée (et `NULL` si c'est le dernier bloc de la liste).

1.4 Gestion des blocs

Pour mémoriser la liste des blocs disponibles et la liste des blocs alloués, on crée une structure « **`struct`** `pool` » :

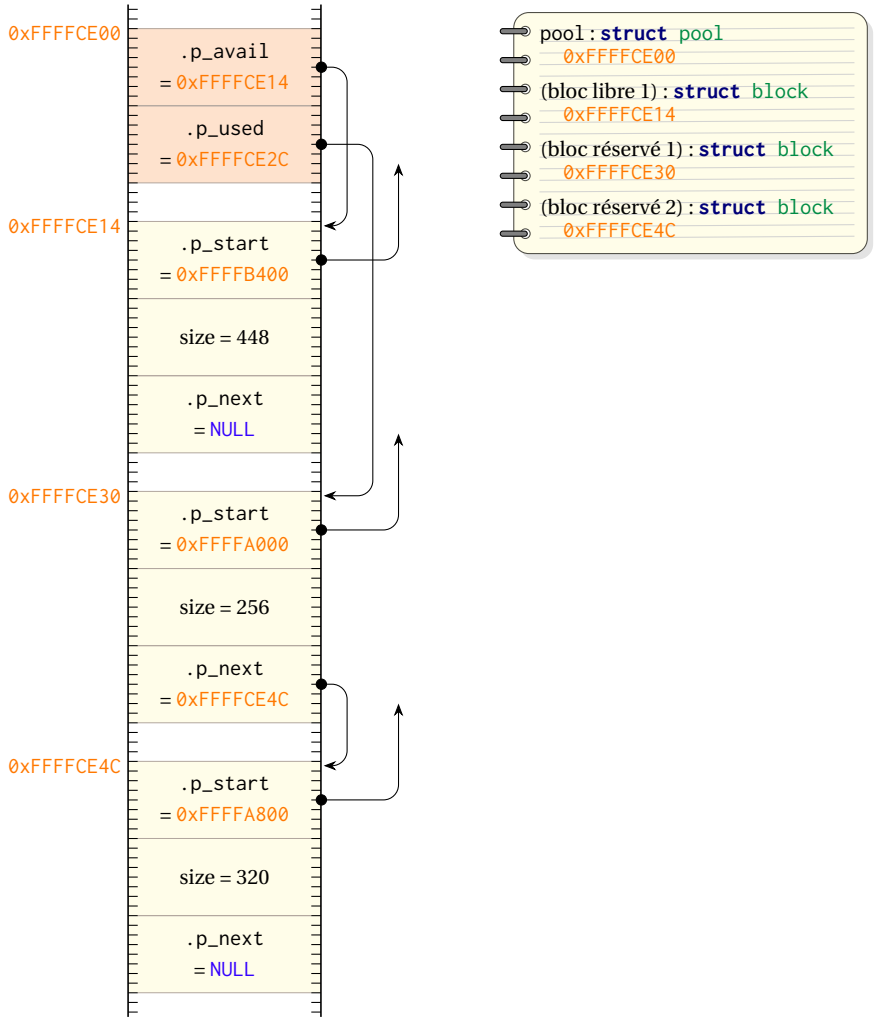
```c
struct pool {
    struct block* p_avail;
    struct block* p_used;
};
```

Le champ `.p_avail` sert à mémoriser la liste chaînée des blocs disponibles (il contient l'adresse de la structure décrivant le premier de ces blocs disponibles). De même, le champ `.p_used` mémorise la liste des blocs actuellement utilisés.

L'initialisation de cette structure (allocation des entiers et création des deux listes) est faite par la fonction `new_pool` qui retourne une telle structure. La liste des blocs utilisés est initialement vide, et celle des blocs disponibles ne contient qu'un seul bloc, donc la taille contient la totalité des entiers alloués.

Pour illustrer un peu l'utilisation des structures, supposons que l'on a réservé 1024 entiers (par exemple 8192 octets si les entiers sont sur 64 bits) de la mémoire, aux adresses `0xFFFFA000` à `0xFFFFBFFF`, puis qu'ont été réservé deux blocs, l'un de 256 entiers, l'autre de 320 entiers. L'allocateur utilisé les adresses `0xFFFFA000` à `0xFFFFA7FF` pour le premier bloc, les adresses `0xFFFFA800` à `0xFFFFB3FF` pour le second bloc. les adresses `0xFFFFB400` à `0xFFFFBFFF` n'ont pas encore été réservées.

Les données de l'allocateur en mémoire ressemblent alors à cela [1] :



Mémoire

1. Les positions exactes des différents éléments en mémoire ne sont pas totalement réalistes.

2 Premières allocations

2.1 Recherche d'un bloc adéquat

La première chose que l'on va chercher à faire est de compléter la fonction `struct block* get_first(struct block* p_lst, int size)` qui prend en argument une liste chaînée de blocs (disponibles) sous la forme d'un pointeur vers le premier maillon, et retourne l'adresse de la structure décrivant le premier bloc de la structure qui soit suffisament grand, et NULL s'il n'y en a aucun dans la liste qui convient.

Aucune modification n'est faite à la liste, on se contente de la parcourir et de retourner l'adresse d'une structure qui pourrait convenir.

1. Compléter la fonction `get_first`.

2.2 Réservation du bloc

Une fois le bloc identifié, il faut le marquer comme utilisé. Pour ce faire, on veut le retirer de la liste des blocs disponibles, et le rajouter dans la liste des blocs utilisés. On change l'objet de type « `struct block` » de liste, donc on ne crée ni ne supprime aucun bloc.

On commencera par compléter la fonction `remove_block_from_list` qui vise à retirer un objet de type « `struct block` » dont on connait l'adresse (passée en second argument) d'une liste chaînée qui le contient. **Attention, si le bloc se trouve en première position, le pointeur (typiquement `pool.p_avail`) qui désigne le premier élément de la liste devra voir sa valeur changer!**.

Par conséquent, on transmet l'*adresse* de la variable qui contient le pointeur vers le premier bloc, pour qu'il soit possible de changer cette adresse. Le premier argument est donc de type `struct block**`!

2. Proposer une implémentation de `remove_block_from_list`.

On souhaite ensuite ajouter le bloc à l'autre liste (par exemple en tête de la liste), et pour ce faire on utilisera une fonction `add_block_to_list`. Le premier argument est de nouveau un `struct block**` car là encore, le pointeur (typiquement `pool.p_used`) qui pointe vers le premier bloc utilisé *va* voir sa valeur modifiée.

3. Proposer une implémentation de `add_block_to_list`.

4. Enfin, on complétera la fonction `ialloc`. La stratégie est la suivante :
- on cherche un bloc qui convient dans ceux disponibles
- on retire ce bloc de la liste des blocs disponibles
- on ajoute ce bloc dans la liste des blocs utilisés

Pour le moment, on supposera qu'on allouera le bloc dans son intégralité même s'il est trop grand.

2.3 Premier test

Normalement, le programme devrait être en état d'être exécuté. On testera une première séquence d'allocation (séquence 1), qui demande successivement des réservations de tailles 1536, 256, 512, 128, 512, 64, 64 et 64. En principe, le résultat devrait être le suivant :

- la première demande d'allocation échoue (on demande plus que le nombre d'entiers disponibles) ;
- la seconde demande d'allocation réussit ;
- toutes les demandes suivantes échouent, car on a alloué la totalité du bloc à la première demande

Par ailleurs, le programme affiche le contenu des deux listes (blocs disponibles et bloc alloués) avant de s'arrêter, ainsi qu'avant chaque opération lorsque l'on active le mode « debug » (en ajoutant l'option `--debug` en lançant le programme, ou en utilisant `make debug` à la place de `make run`). Normalement, il devrait ne plus y avoir de bloc disponible, et un seul bloc alloué de taille 1024.

5. Compiler (et corriger) le programme jusqu'à avoir le comportement attendu.

2.4 Coupure de bloc

Bien évidemment, on ne veut pas allouer *tous* les entiers dès la première demande. Il va falloir donc *couper* le bloc s'il est trop grand. On va donc modifier la fonction `ialloc` de la façon suivante :

- si la taille demandée est égale à la taille du bloc que l'on a identifié comme adéquat, on procède comme auparavant ;
- si la taille demandée est plus petite, on réserve le début de la zone de la façon suivante, et on marque libre le reliquat, de la façon suivante :
  — on crée (par une allocation dynamique) un *nouveau* objet « `struct bloc` » dont l'adresse de début est celle du bloc identifiée et la taille est la taille demandée
  — on ajoute ce nouvel objet dans la liste des blocs utilisés
  — on *modifie* l'adresse de début et la taille du bloc identifié comme adéquat pour correspondre au reliquat (la taille est réduite de la taille qui a été allouée, l'adresse de début est celle qui suit immédiatement tout juste la zone allouée).

6. Modifier la fonction `ialloc` selon les règles précédentes.

7. Tester le programme avec la séquence 1. La première demande doit toujours échouer, les trois suivantes réussir, la cinquième échouer, les deux suivantes réussir, et la dernière échouer. Par ailleurs, il ne doit plus y avoir de bloc disponible à l'issue de la séquence (en particulier, on s'assurera qu'il ne reste pas un bloc de taille nulle !) et on doit retrouver les cinq blocs dont l'allocation s'est passée correctement parmi les blocs alloués.

3 Première approche pour la libération

À présent, il nous faut voir comment nous pouvons libérer la mémoire qui a été allouée lorsqu'elle n'est plus utile. Cela se fait avec un appel à `ifree` qui prend en argument un pointeur de type `int*` retournée par un appel précédent à `imalloc`. Si on passe à `ifree` un pointeur nul, la fonction retourne sans rien faire. Cela permet de passer sans risque un résultat de `imalloc` même si l'allocation n'a pu avoir lieu.

La première étape est d'identifier le bloc alloué que l'on essaie de libérer, puisque l'on ne passe en argument qu'une adresse (`int*`).

8. Compléter la fonction `get_ptr` qui recherche et retourne, dans une liste de blocs, le premier bloc dont le champ `.start` coïncide avec l'adresse fournie en paramètre.

Une fois le bloc identifié, `ifree` retire ce bloc de la liste des blocs alloués et le replace dans la liste des blocs disponibles.

9. Compléter la fonction `ifree` pour effectuer cette opération. On s'inspirera de ce qui a été fait avec `imalloc` (il n'est ici pas question de découpe, la fonction devrait plutôt ressembler à la première version de `imalloc`).

10. Modifier la fonction `main` pour qu'elle utilise dorénavant la séquence 2 pour les tests, et tester le programme. Le comportement doit être le suivant :

- les quatre premières allocations doivent réussir
- les allocations sont ensuite libérées
- les trois allocations suivantes doivent réussir, mais celle qui suit doit échouer (pas de bloc assez grand)
- les allocations sont à nouveau libérées
- les deux dernières allocations doivent échouer.

Par ailleurs, à l'issue du programme, il doit y avoir huit blocs (de tailles 64, 128, 64, 256, 256, 128, 64 et 64) disponibles.

4 Une allocation plus intelligente

Dans le test précédent, il aurait été possible de satisfaire davantage d'allocations si l'on avait été plus malins dans le choix des blocs, plutôt que de prendre le premier disponible.

11. Compléter la fonction `get_largest` qui, contrairement à `get_first`, cherche le *plus grand* bloc qui puisse convenir (on pourra choisir librement en cas d'égalité).

12. Même chose avec la fonction `get_smallest` qui cherche le *plus petit* bloc qui puisse convenir.

13. Modifier la fonction `imalloc` pour qu'elle fasse appel à l'une, puis à l'autre, de ces fonctions plutôt qu'à `get_first`.

14. Quelle stratégie fonctionne mieux ? Pourquoi ?

5 Une libération plus intelligente

Reste que la dernière demande d'allocation n'est toujours pas satisfaite, alors qu'en principe il devrait être possible de le faire. La raison est simple : `ialloc` découpe les blocs, qui deviennent des blocs de petite taille, mais rien ne permet de les recoller. Nous allons à présent tenter d'y remédier.

Pour ce faire, plutôt que de remettre les blocs dans la liste des bloc disponible n'importe où, on va garantir que les blocs sont ordonnés *par adresses de début (*`.p_start`*) croissantes*.

15. Compléter la fonction `insert_block_into_list` pour effectuer cette insertion.

16. Proposer une fonction `consecutive` prenant en argument deux pointeurs vers deux blocs (supposés provenant du même « pool » de mémoire), et retournant un booléen indiquant si le second suit immédiatement le premier en mémoire.

17. Proposer une fonction `clean_list` qui prend une liste de blocs disponibles, et effectue des fusions de blocs consécutifs (on prendra garde à bien appeler `free` pour supprimer la `struct block` qui disparaît) tant qu'il y a des blocs consécutifs dans la liste.

18. Tester à nouveau le programme avec la séquence 2. Toutes les allocations doivent dorénavant réussir !

19. Trouver une séquence d'allocations et de désallocations telle qu'il n'y a jamais plus de 1024 entiers requis à un quelconque instant, et qui pourtant conduit à des allocations impossibles.