TP5 : Gestion de la mémoire

Matthieu Boutier *(boutier@pps.univ-paris-diderot.fr)*

Le but de ce TP est de faire une alternative à `malloc(3)`.

1 Allocateur mémoire naïf

Cet allocateur naïf ne permet pas de libérer la mémoire.

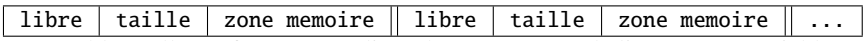
1. Créez trois fichiers `alloc.c`, `alloc.h` et `main.c`.
2. Écrivez une fonction `void *malloue(int taille)` qui utilise l'appel système `sbrk(2)` pour obtenir de la mémoire auprès du système.

2 Un vrai allocateur mémoire

Afin de libérer la mémoire, ou plus exactement des parties de la mémoire, il est nécessaire de pouvoir identifier ces différentes parties. Nous allons programmer une version plus proche de la façon donc `malloc()` travaille, avec une structure de bloc alloué, constituée de trois champs :

— Un flag disant si le bloc est libre ou occupé (codé sur un int),
— La taille du bloc (codé sur un int),
— La zone mémoire elle-même (de taille variable).

La mémoire ressemblera à cela :



Les blocs alloués forment une liste chaînée. La tête de liste est une variable statique affectée lors du premier appel à `malloue()`. On peut ensuite parcourir la liste des zones mémoires grâce au champ `taille`, jusqu'à ce qu'on dépasse l'espace alloué par le système. Cette limite, obtenue par sbrk(0), sera stockée dans une autre variable statique (pour éviter trop d'appels système).

1. Créez une variable statique `void *mem_dyn`, qui représente votre mémoire dynamique, et `void *fin` qui représente l'adresse après le dernier bloc alloué (à tout moment, `fin == sbrk(0)`).
2. Écrivez `void *search_free_place(int x)` qui cherche dans la liste des zones mémoires allouées un emplacement libre d'au moins `x` octets. S'il n'y en a pas, cette fonction retournera `NULL`.
3. Modifiez `malloue()` pour qu'elle utilise la fonction précédente. Attention de renvoyer l'adresse de début de mémoire libre (pas du bloc) !
4. Écrivez une fonction `void mlibere(void *)`, notre version de `free(3)`. Attention au fait que l'adresse connue du programmeur n'est pas l'adresse de début du bloc, mais de la zone mémoire libre (il faut retrancher la taille de l'entête).

5. La fonction `search_free_place()` a tendance à fragmenter la mémoire. Écrivez une fonction `void *search_best_free_space(int x)` qui cherche la plus petite zone mémoire contenant `x` octets.
6. Écrivez `mrealloue()` qui se comporte comme `realloc(3)`. Si à la suite du bloc à réallouer se trouve miraculeusement un ou plusieurs blocs libres que l'on peut récupérer, c'est parfait on les concatène. Sinon il faut allouer un nouveau bloc, copier le contenu existant, et libérer l'ancien bloc.
7. BONUS : Faites en sorte que libérer la mémoire fusionne toujours avec les zone mémoires libres adjacentes.
8. BONUS : À l'inverse, que l'allocation de mémoire fragmente le bloc mémoire utilisé s'il est assez gros pour cela.
9. BONUS : Pour utiliser moins de mémoire quant-à l'en-tête, codez le flag "libre" en utilisant le bit de signe du champ taille.