Allocateur de mémoire en C

Projet 1ère partie

Un allocateur de mémoire est un programme qui gère le tas [1], la zone de mémoire réservée aux données non statiques et non locales aux fonctions. En C, ce programme est codé dans la bibliothèque standard `stdlib`. Un allocateur de mémoire permet de :

réserver un bloc de mémoire dont la taille est donnée en argument de la demande ; en C, ceci est réalisé grace à l'appel à la fonction `void *malloc(size_t size)`,

libérer un bloc de mémoire préalablement réservé ; en C, ceci est réalisé par l'appel à la fonction `void free(void *ptr)`,

redimensionner un bloc mémoire déjà réservé à la taille donnée en argument ; en C, ceci est possible avec la fonction `void *realloc(void *ptr, size_t size)`.

La réservation et le redimensionnement renvoient l'adresse de début d'une zone de mémoire dans le tas ou 0 (NULL) si l'opération n'a pas pu être effectuée (à cause de l'épuisement de la mémoire disponible, par exemple).

L'objectif de ce projet est de vous faire programmer un allocateur de mémoire en C, qui pourrait remplacer celui de la bibliothèque `stdlib`. Vous devez coder votre projet dans une bibliothèque `myalloc` contenant deux fichiers : `myalloc.c` avec le code des fonctions `malloc`, `free` et `realloc`, `myalloc.h` avec la déclaration de ces fonctions pour l'utilisation dans les programmes « clients » écrits en C.

L'allocateur de mémoire utilisera deux politiques différentes en fonction de si le bloc de mémoire demandé est « petit » ou « grand ». Dans les deux premières parties, on s'intéressera à une première version de l'allocateur de petits blocs. La troisième partie concernera les blocs de grande taille.

Ce document décrit la seconde partie qui est à rendre pour le 17 octobre.

Partie 2 : blocs de petite taille, version 2

Spécification

La version de l'allocateur proposée en première partie souffre de deux défauts principaux : elle ne permet pas d'allouer de trop grande taille, et elle n'est pas très efficace. Cette seconde version vise à régler le problème d'efficacité, toujours pour des allocations de petite taille.

1. En faite une partie du tas, car certaines parties peuvent être réservées en dehors de l'allocateur.

Dans la première version, lors d'une demande d'allocation, le tableau devait être parcouru pour trouver une potentielle case libre, et l'allocation de mémoire a donc une complexité linéaire dans le pire cas. Une première idée pour pallier ce problème serait de retenir la case du prochain bloc libre, mais il faut quand même rechercher un autre bloc libre quand celui retenu est alloué, donc la complexité reste linéaire.

La solution proposée dans la deuxième partie de ce projet consiste à utiliser l'entête d'un bloc libre comme un pointeur vers une autre case libre du tableau. Ainsi, les blocs libres sont répertoriés dans une liste dont on garde en mémoire l'adresse du premier élément. La complexité de l'allocation devient constante car on garde toujours en mémoire un accès direct vers une case libre.

Plus précisément, on aura en plus du tableau de petits blocs, un pointeur qui indique le « premier » bloc libre (au sens du premier qu'on allouera, pas du premier dans l'adressage de la mémoire). Lorsqu'on alloue le bloc vers lequel ce pointeur pointe, l'en-tête du bloc nous indiquera la position du bloc suivant libre. Pour cela, la liste des blocs libres doit être initialisée avant le fin du premier appel à `malloc` ou `free`. Lorsqu'on libère un bloc, ce bloc est ajouté en début de la liste de blocs libres, afin d'obtenir une complexité en temps constant pour l'opération `free`.

Tests et debug

Pour garantir un standard de qualité du code, il sera attendu que votre code compile avec les options `-Wall -Wextra -Werror -Wshadow -ansi -pedantic` de gcc.

En plus du code destiné à être exécuté par des utilisateurs de l'allocateur, il est important d'écrire du code supplémentaire :

- Déjà, des fonctions de debug, dont le but est de faire des vérifications en plus afin de trouver les inévitables bugs dans votre code. Ceci inclut des fonctions permettant de visualiser l'état de la mémoire gérée par votre allocateur, des fonctions d'accès à la mémoire vérifiant qu'on accède bien uniquement à de la mémoire allouée, etc.
- Ensuite, des tests de correction qui appellent les fonctions de la librairie et vérifient que leur comportement est correct (par exemple qu'un bloc renvoyé par `malloc` était bien libre avant et occupé après, qu'un bloc `free` est bien libre après, que `malloc` renvoie bien NULL si la mémoire est pleine ou que la taille demandée est trop grosse, etc.)
- Enfin (en option si vous avez le temps), des tests de performance. En utilisant la fonction `time`, comparez les temps d'exécution de votre programme avec la librairie standard (et surtout, lors des parties suivantes du projet, constatez le gain de performances). Il est utile pour ce genre de tests de tester votre librairie sous pression (*i.e.*, d'avoir beaucoup de blocs alloués en même temps et de faire beaucoup d'opérations).