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.

Ce projet sera divisé en 3 parties, la première partie est à rendre pour le mardi 8 octobre. L'allocateur de mémoire utilisera deux politiques différentes en fonction de si le bloc de mémoire demandé est « petit » ou « grand ». Dans la première partie, on s'intéressera à une permière version de l'allocateur de petits blocs.

Partie 1 : blocs de petite taille, version 1

Spécification

L'allocateur de petits blocs utilise un tableau statique `small_tab` contenant `MAX_SMALL` blocs de 128 octets (on pourra prendre par exemple `MAX_SMALL` égal à 100). Chaque bloc est composé :

d'un entête représenté par un entier [2] et qui a deux fonctions : le bit de poids faible indique si le bloc est libre (0) ou occupé (1), les bits de poids fort seront utilisés dans la version 2 ;

1. En réalité une partie du tas, car certaines parties peuvent être réservées en dehors de l'allocateur.
2. Cet entier est voué à stocker des pointeurs, et sera donc un `size_t`

d'un corps représenté par un tableau d'octets de taille `SIZE_BLK_SMALL` égal à `128-sizeof(size_t)` octets, dans lequel se trouve l'information stockée par l'utilisateur de l'allocateur si le bloc est occupé.

Lors d'une demande d'allocation, si la taille du bloc demandé est inférieure ou égale à `SIZE_BLK_SMALL` alors la fonction `malloc` cherche, dans le tableau `small_free` un bloc libre. Si au moins un bloc est libre, alors le premier bloc libre b est réservé et la fonction `malloc` renvoie au demandeur l'adresse `(size_t *)b + 1`, qui est le début du corps du bloc b.

Lors d'une demande de libération, la fonction `free` s'assure que la mémoire à libérer a une adresse valide (*i.e.*, est le début du corps d'un bloc occupé). Si c'est pas le cas, un message d'erreur est affiché. Si l'adresse à libérer est correcte, le bloc de cette adresse est marqué libre.

Lors d'une demande de redimensionnement, la fonction `realloc` s'assure que l'adresse en argument correspond à un bloc occupé et que la nouvelle taille est inférieure à `SIZE_BLK_SMALL`. Si ce n'est pas le cas, la fonction renvoie `NULL` (dans cette première version).

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 librarie 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 librarie sous pression (*i.e.*, d'avoir beaucoup de blocs alloués en même temps et de faire beaucoup d'opérations).