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 troisième partie est à rendre pour le jeudi 7 novembre.

Partie 3 : blocs de grande taille

Pour des blocs de « grande » taille, l'allocateur va gérer l'espace alloué directement dans le tas, et demander au besoin au système d'exploitation de rallonger la taille du tas. De plus, les blocs gérés ne seront plus de taille fixe, mais s'adaptent à la taille demandée.

Spécification

Chaque bloc est composé :

d'un entête représenté par une paire entiers. Le premier entier encode l'état du bloc (libre/occupé, sur le bit de poids faible) et l'adresse du bloc libre suivant, s'il existe. Le second entier est la taille du bloc en nombre d'octets. On note h la taille de

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

l'entête (2∗`sizeof(size_t)`). On s'assure que les tailles des blocs sont multiples de `sizeof(size_t)` (pour garder l'alignement en mémoire des blocs) et supérieures à `SIZE_BLK_SMALL`. Si la taille demandée ne satisfait pas ces contraintes, l'allocateur réserve un bloc de taille égale au plus petit entier multiple de `sizeof(size_t)` et plus grand que la taille demandée plus h (pour l'entête).

d'un corps contenant un nombre d'octets égal à la taille donnée dans l'entête moins la taille de l'entête.

Le programme maintient une variable globale `big_free` donnant le début de la liste de grands blocs libres. Initialement, l'allocateur met dans cette liste un premier bloc, de taille `SIZE_FIRST_BLK_LARGE` (à choisir, par exemple 1024), en utilisant l'appel système `sbrk(SIZE_FIRST_BLOCK_LARGE)` [2].

Lors d'une demande d'allocation, si la taille demandée ℓ est supérieure à `SIZE_BLK_SMALL` alors la fonction `malloc` cherche, dans la liste `big_free` un bloc libre pouvant satisfaire cette demande (dont la taille est au moins $\ell + h$).

- Si un tel bloc n'existe pas, l'allocateur fait une demande au système un bloc de mémoire de taille au moins $\ell + h$ en utilisant l'appel à `sbrk`.
- Si le bloc existe et que sa taille est assez proche de celle demandée (par exemple $< \ell + h +$`SIZE_BLK_SMALL` octets), tout le bloc est déclaré occupé et la liste des blocs libres est mise à jour pour éliminer ce bloc.
- Si un bloc b existe avec une taille ℓ_b plus grande que $\ell + h +$`SIZE_BLK_SMALL` bytes, alors le bloc est divisé en deux parties :
  - au début, un bloc b' qui reste libre mais dont la taille est redimensionnée pour exclure les derniers $k \geq \ell + h$ octets (avec k multiple de `sizeof(size_t)`),
  - à la fin, un bloc b'' dont la taille est k et qui sera occupé.

La fonction `malloc` renvoie au demandeur l'adresse du debut du corps du bloc alloué.

Lors d'une demande de libération, la fonction `free` marque le bloc contenant l'adresse libérée a comme étant libre et l'insère dans la liste `big_free`. Notez que l'allocateur doit retrouver à l'adresse $a - h$ une valeur entière qui a le bit de poids faible à 1 et à l'adresse $a-$`sizeof(size_t)` un entier qui dénote la taille du bloc.

Une variante de cette fonction permet de lutter contre la fragmentation de la mémoire (création de blocs libres adjacents mais de taille individuelle inférieure au besoin des clients). Lors de la libération d'un bloc, si la liste `big_free` contient un bloc adjacent (à gauche ou à droite) avec le bloc qui vient d'être libéré, alors la jonction des deux blocs est faite et un seul bloc libre est insérée dans la mémoire.

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 celle du bloc. Si c'est pas le cas, la fonction cherche un nouveau bloc libre de la taille demandée dans la liste des blocs libres, et procède à une séquence `malloc-copy-free`.

2. plus de détails à propos de `sbrk` en fin de sujet

Si la nouvelle taille est « suffisamment » plus petite que l'ancienne, on peut également séparer le bloc en deux et ajouter la deuxième moitié à la liste de blocs libres, voire déplacer le bloc dans la partie des blocs de petite taille.

L'appel système `sbrk`

`sbrk` est inclus via le fichier `unistd.h`, et a la signature suivante : `void *sbrk(intptr_t increment)`. Un appel à `sbrk(n)` déplace le program break (la fin du tas) de `n` octets. Ainsi, si `n` est positif, de la mémoire est allouée, et si `n` est négatif, de la mémoire est libérée. La valeur renvoyée est le program break précédent. Ainsi, un appel à `sbrk(0)` permet d'obtenir la valeur du program break, tandis que pour une allocation ($n > 0$) la valeur renvoyée est un pointeur vers une zone de mémoire libre (mais non initialisée) de la taille demandée. `sbrk` peut également renvoyer `(void *) -1` lors d'une erreur (si le système ne peut pas allouer la quantité de mémoire demandée).

Précautions avec l'usage du tas (à lire si vous avez des bugs vraiment bizarres)

L'usage du tas alloué par le système au démarrage d'un processus est une vaste jungle, et tend d'ailleurs à être déprécié (pour ceux qui veulent regarder, un allocateur moderne alterne entre l'utilisation du tas initial pour des questions de performance, et la gestion d'autres tas alloués avec `mmap`). Si on ne vous demande pas dans ce projet de comprendre la virtualisation de la mémoire, et que `sbrk` n'a que peu de chances de renvoyer une erreur (sauf si vous essayez d'allouer plusieurs centaines de Go...), il faut néanmoins être conscient que certaines fonctions auxquelles vous êtes habitués utilisent `malloc` en interne, et malheureusement pour vous, `printf` en fait partie. Cela veut dire deux choses : si vous remplacez les fonctions d'allocation de la librairie standard, le bon fonctionnement de `printf` dépend du bon fonctionnement de votre programme, et vous n'êtes probablement pas les seuls à utiliser le tas dans tous les cas. Ainsi, on ne peut compter ni sur le fait que le program break n'ait pas bougé entre deux de vos appels à `sbrk`, ni sur le fait que deux blocs soient contigus sans l'avoir explicitement vérifié. Pour ceux qui veulent limiter ce genre de problèmes, vous pouvez utiliser les fonctions d'affichage plus bas niveau (`puts`, `putc`, ...), qui n'allouent rien.

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, ect.)

— 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).

Extensions possibles

Voici quelques pistes pour aller plus loin (elles ne sont pas spécialement rangées par difficulté d'implémentation).

— Rechercher dans la liste de blocs libres peut être long, n'hésitez pas à faire des tests qui font exprès de faire grossir la liste de blocs libres afin de comparer la performance de votre code avec celle de malloc. Pour remettre cela en contexte, comparez aussi la vitesse de votre code et de malloc dans des cas simples (où malloc d'a pas d'avantage algorithmique sur votre implémentation).

— Comme mentionné plus haut, lors de la libération d'un bloc, on peut vérifier si les blocs adjacents sont aussi libres, auquel cas on fusionne lesdits blocs avant de réinsérer dans la liste de blocs libres. Si on fait cela, on essaie de maintenir l'invariant que deux blocs libres dans la liste ne peuvent être contigus.

— Vous l'avez peut-être remarqué, mais actuellement, on n'utilise qu'un nombre constant de petits blocs, et on renvoie `NULL` si l'utilisateur essaie d'en allouer plus. Une première solution pour cela est de créer de petits blocs supplémentaires de la même manière que les grands blocs si besoin (attention, cela cause beaucoup de soucis de fragmentation). On peut aussi allouer beaucoup de petits blocs d'un coup comme un gros bloc et les ajouter à la liste de petits blocs libres (moins de fragmentation, mais il est difficile de détecter si notre nouveau tas de petits blocs est entièrement libre lors d'un free, et en plus il n'est pas nécessairement souhaitable de le désallouer immédiatement dans ce cas, car l'allouer prend un certain temps (pour chaîner tous les petits blocs à l'intérieur)).

— On peut lors de la libération et fusion de blocs, regarder si le nouveau bloc libre est bien le dernier bloc du tas (`sbrk(0)` pointe le premier octet que le bloc n'utilise pas). Dans ce cas, et si le bloc est supérieur à une limite (par exemple 16*1024), on libère de la mémoire via `sbrk` de sorte que le bloc revienne à une taille donnée (par exemple 1024).