Allocateur de mémoire en C

Programmation 1 (2026 - 2027)

Vincent Lafeychine
vincent.lafeychine@ens-paris-saclay.fr

La première version de l'allocateur fonctionne, mais présente de nombreux défauts. Le principal défaut réside dans la fragmentation des *chunks*. En effet, lorsqu'un *chunk* est libéré, il faudrait que le *chunk* regarde si ses voisins sont également libérées afin de former un seul *chunk* plutôt que plusieurs *chunk* libérés consécutifs.

Cependant, étant donné un *chunk*, ce dernier n'a uniquement la possibilité de récupérer son prochain voisin. Une première idée consisterait à stocker également la taille du *chunk* précédent, mais cela augmenterait la taille des métadonnées, ce qui doit être évité.

Dans cette partie, nous nous intéresserons à la fusion de *chunk* libres, ce qui permettra notamment de déplacer le *program break* « négativement » afin de redonner de la mémoire au système.

1. Modalités de rendu

Il est attendu que vous **modifiiez** les fichiers que vous avez déjà rendus lors de la première partie. De plus, une plus grande autonomie de votre part est attendue : vous devez vous assurer de l'intégrité de vos structures. N'hésitez pas à utiliser des assertions pour vous aider.

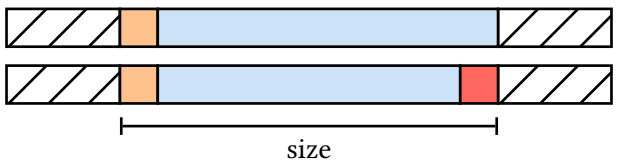
À partir de maintenant, votre projet **échouera** la compilation si des *warnings* sont présents. Il est vivement conseillé d'exécuter la commande suivie de pour vérifier qu'aucun avertissement subsiste. Vous pouvez également opter pour une solution plus radicale en ajoutant l'option de compilation dans le fichier , dans les lignes 17 et 18 concernant les et les , ce qui aura pour effet d'empêcher la compilation au moindre avertissement.

2. Récupération du voisin précédent

Lorsqu'un *chunk* devient libre, le payload autrefois utilisé par l'utilisateur peut être maintenant réutilisé par l'allocateur. D'après l'alignement de 16 octets et de 8 octets, le payload fait au moins 8 octets de mémoire.

Pour résoudre le problème d'accès au voisin précédent, nous pouvons considérer qu'il est possible de récupérer la taille du *chunk* précédent depuis son payload dès lors qu'il est libéré.

Ainsi, un *chunk* utilisé se compose toujours d'un *header* en orange et d'un *payload* en bleu. Dès lors qu'il n'est plus utilisé, un *chunk* se compose d'un *header* en orange, d'un *payload* en bleu, mais également d'un *footer* en rouge contenant la taille du *chunk*.



Le *footer* étant présent en fin de *chunk*, et les *chunks* étant contigus en mémoire, il suffit à un *chunk* de lire la donnée à côté de ses métadonnées pour connaître la taille du *chunk* précédent.

Il ne reste donc plus qu'à déterminer quand le chunk précédent est libre pour s'assurer de la présence du *footer*. Pour ce faire, le flag est renommé et indique désormais si le *chunk* précédent est libre.

En plus d'appliquer les changements nécessaires dans votre projet (impactant , et), vous allez écrire deux nouvelles fonctions dans le fichier :

Avant tout, il est recommandé de modifier la structure et demandez-vous si doit être modifiée. De plus, faites bien attention au *top chunk*.

La fonction inscrit la taille du *chunk* passé en paramètre dans son *footer*. En opposition, la fonction récupère le *chunk* précédent en récupérant sa taille dans le *footer*.

Pour écrire ces fonctions, il peut être pertinent d'ajouter des assertions.

3. Fusion des blocs libres

Maintenant qu'un *chunk* peut obtenir l'ensemble de ses successeurs, ainsi que l'ensemble de ses prédecesseurs sous réserve qu'ils soient libérés, la fusion de blocs peut maintenant être écrite.

Vous allez écrire deux nouvelles fonctions dans le fichier :

Ces fonctions essaient de fusionner le bloc qui précède () et le bloc qui suit () le *chunk* passé en paramètre, en analysant les *flags*. Si un *flag* ne permet pas de faire la fusion, le *chunk* en paramètre est renvoyé. Si une fusion est appliquée, le *chunk* fusionné est renvoyé.

Encore une fois, faites bien attention au *top chunk* : il est autorisé d'utiliser les fonctions du fichier d' dans pour ces deux fonctions.

4. Libération du top chunk

Maintenant que les blocs mémoires avant le *top chunk* peuvent fusionner avec ce dernier, l'allocateur est maintenant capable de redonner de la mémoire au système.

Vous allez écrire une nouvelle fonction dans le fichier :

La fonction vérifie que le *top chunk* fait au moins deux fois la taille d'une page. Si tel est le cas, la fonction déplace le « curseur » du *program break* vers la gauche afin de libérer de la mémoire et de ne laisser au top chunk qu'une taille de . Cette fonction ne peut pas échouer.

5. Réallocateur plus intelligent

Dans la première partie, le réallocateur appelait directement , suivi d'une copie de données, puis appelait . Cette procédure, nommée « *malloc-copy-free* », est très coûteuse en termes de performances et doit être évitée.

Nous vous proposons d'écrire un algorithme de réallocation qui évite la procédure « *malloc-copy-free* » dans plusieurs situations.

Lorsqu'un appel à demande une nouvelle taille qui :

1. est déjà la taille du *chunk*: il suffit de renvoyer le bloc.
2. est plus petite que la taille du *chunk*: il faut *split* le *chunk*, mettre comme libre le nouveau *chunk* obtenu par le *split*, et renvoyer le *chunk* réduit.
3. est plus grande que la taille du chunk et que le prochain *chunk* est le *top chunk*: il faut *split* le *top chunk* en augmentant sa taille si nécessaire et renvoyer le nouveau *chunk*.
4. est plus grande que la taille du chunk et que le prochain *chunk* est un *chunk* libre d'une taille assez grande: il faut *split* si nécessaire le prochain *chunk* et fusionner les *chunks*.
5. dans tous les autres cas: il faut effectuer la procédure « *malloc-copy-free* ».

6. Écriture de tests avancée

Pour vérifier qu'une fonction affiche bien la sortie prévue (notamment la fonction), et sont mis à votre disposition. Pour *chaque* nouvelle capture, vous devez entourer la fonction par pour commencer l'enregistrement, puis par pour récupérer le contenu de l'enregistrement.

À titre d'exemple, voici un exemple de capture :

Pour certains de vos tests, il peut être intéressant de vous assurer qu'une fonction appelle bien votre assertion. Pour ce faire, la macro prend une expression à tester ainsi qu'un *pattern* (mettez la chaîne de caractère).

Par exemple, la fonction peut vérifier que l'argument ne soit pas le pointeur à l'aide d'une assertion :