Rapport du TP 2 :
Malloc et comptage de références

Par
Steven Han, 20129462

Département d'informatique et de recherche opérationnelle

Travail présenté à Érick Raelijohn
Dans le cadre du cours IFT-2035
section A Concepts des langages de programmation

1 Août 2019

1. Expérience

Préambule : Les problèmes expliqués sont un processus linéaire de l'implémentation de *mymalloc*, *myfree* et *refinc*. Les solutions ne sont pas dans cette section, mais plutôt dans la section 2, où on expliquera en détail notre algorithme ainsi que chacune des fonctions *mymalloc*, *myfree* et *refinc*. Je dois vous avouer que l'implémentation de ces trois fonctions n'est vraiment pas évidente et difficile.

Problème général :

La plus grosse difficulté état le début : par où commencer ? On a tous un petit bagage au niveau des modèles mémoire comme le *stack*, le *heap*, les listes chaînées, etc. Parfois on utilisait des fonctions prédéfinis et c'était facile. Mais cette fois-ci, on jouait avec la mémoire pour de vrai. Pour nous stimuler, nous n'avons fait aucun appel à *malloc* et *free* de la librairie standard. Donc, pour résoudre ce problème, nous avons crée un tableau de char qui contient 4ko d'éléments. Chaque ko est 1024 octets, donc on devait allouer un tableau de char avec une grandeur de 4096.

Maintenant que nous avons notre bloc de mémoire, comment l'arranger ?

Il y avait aussi le problème des pointeurs : gérer des pointeurs est relativement difficile. Le concept est assez simple et facile à comprendre, mais quand on les manipule, ils peuvent avoir des comportements très étranges. Par exemple, si on a un pointeur en argument et qu'on prend ce pointeur et qu'on lui assigne la valeur de NULL, lorsqu'on sort de la fonction le pointeur n'est pas vide. Normalement, le passage par référence change la valeur de la variable.

Problème dans *mymalloc* :

Le premier problème de cette fonction est l'optimisation de la mémoire : donner assez de mémoire sans trop en perdre. Le programmeur peut demander une taille de 1 comme il peut demander une taille dans les millions. Il faut se préparer à ce genre de demande. Comment peut-on donner juste assez ? Après plusieurs réflexions, nous avons suivi la gestion mémoire du *malloc* et *free* de l'énoncé : nous aurons une cartographie d'un bloc de 4ko. Cela optimise la perte de mémoire, mais ne supporte pas les grandes tailles demandées par le programmeur.

Ensuite, le programmeur demande une allocation mémoire d'une certaine taille. Comment vérifier cette taille ? Si le programmeur donne une taille exacte à la taille de notre bloc mémoire, c'est facile. Mais s'il demande moins que 4ko (ce qui arrive fréquemment), comment va-t-on séparer le bloc ?

Une fois la mémoire allouée, comment peut-on tester que la mémoire a été alloué pour de vrai ? On peut seulement tester la taille de chaque bloc, mais comment peut-on savoir que le prochain pointe réellement ce à quoi on s'attend ?

Problème dans *myfree* :

Le plus gros problème de cette fonction est qu'on ne peut pas vérifier si *free* a réellement libérer le pointeur. Selon plusieurs sources d'internet, il faut faire confiance au compilateur pour la désallocation de pointeur. Cela se voit dans les tests du *main* de notre fichier *mymalloc.c* : lorsqu'on free un pointeur plusieurs fois, on peut toujours imprimer sa valeur. Pourtant, la mémoire a été désallouée (le test du fichier *tests.c* le prouve). Le pointeur pointe toujours la valeur qui lui a été assigné. Aussi, pour que *myfree* se comporte comme *free* de la librairie standard, il faut implémenter un *garbage collector* et de la défragmentation. Si ce n'est pas là, la mémoire ne sera jamais libérée réellement (on pourra toujours l'accéder). Un autre point important est que si on veut utiliser *free* de la librairie standard pour pouvoir libérér la mémoire, il faut absolument que le pointeur provient du *malloc* de la librairie standard.

Tous ces problèmes n'étaient que des introductions à la complexité de *malloc* et *free*. À la fin du rapport, il y aura d'autres problèmes (provenant de l'énoncé) où on y répondra.

2. Documentation

Avant de documenter chacune des fonctions et de les expliquer, nous allons vous expliquer
l'idée derrière *malloc* et *free*.

2.1 Algorithme :

Notre algorithme fonctionne comme dans la section 3.3 de l'énoncé. Voici une image qui représente la structure de notre bloc crée par *malloc* :

4.2 How to represent block information

So what we need is a small block at the begining of each chunk containing the extra-information, called meta-data. This block contains at least a pointer to the next chunk, a flag to mark free chunks and the size of the data of the chunk. Of course, this block of information is before the pointer returned by `malloc`.

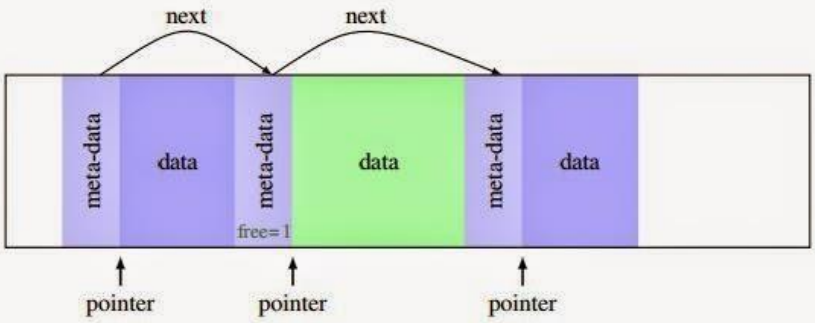

Figure 3: Heap's Chunks Structures

Figure 3 presents an example of heap organisation with meta-data in front of the allocated block. Each chunck of data is composed of a block of meta-data followed by the block of data. The pointer returned by `malloc` is indicated in the lower part of the diagram, note that it points on the data block, not on the complete chunk.

L'idée est que nous avons une grosse carte de mémoire (représenté par le plus gros rectangle dans l'image). Dans notre cas, nous avons choisi 4ko. Dans cette cartographie, chaque bloc d'information (bloc *data*) contient un index (*meta-data*). Chaque index pointera vers un prochain index, qui contiendra un autre bloc d'information. Le pointeur initial pointe le début de la cartographie et lorsqu'on fait appel à la fonction *refinc*, le pointeur avance au prochain index. Si on fait appel à la fonction *myfree*, on «effacera» l'index où le pointeur se trouve ainsi que le bloc d'information. Le pointeur reculera d'un index. Il est à noter que l'index et le bloc d'information doivent être gardé en mémoire côte à côte. Cette structure est une sorte de liste chaînée et requise, car nous avons besoin de savoir si un bloc d'information a été libéré ou s'il est occupé et pour savoir la taille du bloc du prochain index. Notez aussi que dans notre code, on réutilisera cette cartographie pour les allocations futurs de mémoire.

2.1.1 Avantages :

Nous avons choisi cet algorithme, car elle est simple à comprendre, permet de réutiliser des blocs d'information désalloué et que c'est la méthode la plus utilisé sur le web (*stack et heap*). De plus, en mettant une limite d'utilisation de mémoire, cela évitera les pertes de mémoire inutile.

2.1.2 Inconvénient :

Par contre, il y a un inconvénient au niveau de cette structure. Notre bloc a une taille fixe de 4ko, donc si un programmeur veut une taille plus grande que 4096 octets, on devra lui refuser. Notre code pourrait avoir une taille plus grande (en changeant simplement la taille fixe dans notre code à la taille désirée), mais cela pourrait gaspiller de la mémoire inutile.

2.2 Documentation des fonctions:

On documentera seulement les fonctions *mymalloc, myfree* et *refinc*. Pour le reste, il y a des commentaires pour vous guidez.

2.2.1 mymalloc

L'idée est d'initialiser un bloc mémoire de 4ko si ce n'est pas déjà fait. On a déjà un tableau de char comprenant 4096 éléments et un pointeur vers le début de ce bloc. Il suffit d'affecter les champs tailleBloc, libre et prochain. On doit aussi garder une référence vers un pointeur actuel (servira à traverser tous les index du bloc initialisé) et son précédent (le précédent sert d'ajustement des pointeurs).

Ensuite, on doit chercher une place disponible pour l'allocation de mémoire. Pour ce faire, on vérifie chaque index du bloc et on regarde s'il est disponible ou pas. S'il n'est pas disponible, on ajuste les pointeurs (le pointeur précédent va pointer au prochain du pointeur qui n'est pas disponible).

Après avoir trouvé le pointeur libre, on doit regarder la taille que le programmeur demande. Il y a 3 cas :

- S'il est exactement la taille de notre pointeur, alors on lui alloue la mémoire et on lui fait signe que la taille est exactement le taille que le bloc d'information peut avoir.
- Si la taille demandée par le programmeur est plus petite que la taille du bloc où le pointeur est, il faudra couper le bloc où le pointeur est. On retournera ensuite le pointeur qui pointe au premier index de la mémoire alloué. Cette méthode est un peu plus de travail, mais ne perd pas de la mémoire inutilement.
- Si la taille demandée par le programmeur est plus grande que la taille du bloc, on devra refuser et retourner le pointeur vide.

2.2.2myfree

L'idée est que le pointeur reçu en argument est un pointeur retourner par la fonction *mymalloc*. Donc, il suffit de vérifier si l'adresse du pointeur est dans l'intervalle de l'adresse du bloc de 4ko. C'est-à-dire entre 0 et 4096. Si c'est le cas, on met le champ libre à 1 indiquant qu'il est libre et on recule de pointeur. On appel ensuite la méthode *misAJourPointeur* pour mettre à jour les références. Il agit comme un défragmenteur. Si vous voulez plus de détail sur cette fonction, vous pouvez aller voir le code.

2.2.2refinc

Similaire à *myfree*, *refinc* prend un pointeur retourné par *mymalloc*. On vérifie ensuite si l'adresse du pointeur réside dans l'intervalle d'adresse de notre bloc de 4ko. Si c'est le cas, on n'a qu'à prendre la taille du bloc d'information où le pointeur est et retourner cette valeur. C'est ainsi, car on pointe à un certain index et si on veut accéder au prochain, on n'a qu'à retourner la taille du bloc d'information pour savoir où pointe le prochain bloc.

2.2.3*malloc* et *free* de la librairie standard :

Comme mentionné dans la partie 1, nous n'avons pas fait appel à *malloc* et *free* de la librairie standard. Si on avait appelé *malloc* de la librairie standard, il n'y aurait pas eu tous les cas de la fonction *myfree* et il y aurait seulement une ligne de code. Lorsqu'on fait appel à *malloc* de la librairie standard, le travail est fini et donc quel serait la difficulté d'implémenter *mymalloc* puisque *malloc* fait tout le travail ?

L'appel à *free* de la librairie standard aurait été intéressant, car il y a un *garbage collector*. Cependant, étant donné que nous n'avons pas fait appel à *malloc* de la librairie standard, on ne peut pas appeler la fonction *free* de la même librairie. Cela causerait un *undefined behaviour* et le programme plantera. Ceci est à cause que *free* accepte seulement les pointeurs retournés par *malloc* de la librairie standard et rien d'autres.

2.3 Solutions/explication à certains problèmes :

Cette section répondra aux questions de l'énoncé.

1. Cet algorithme ne couvre pas le cas où le programmeur demande d'un seul coup plus que 4 Ko de mémoire.
   C'est vrai. Respecter la taille minimum que le programmeur demande et optimiser la mémoire est assez difficile. Donc, on ne fait pas confiance au programmeur et on a limité le programmeur à 4ko pour ne pas qu'il gaspille de la mémoire inutilement. Si le programmeur veut plus de mémoire, il n'a qu'à prendre notre code et changer la taille du champ blocMemoire et tous les 4096 à la taille désirée.

2. **Lorsque l'utilisation mémoire requiert plusieurs blocs de 4 Ko, il faut que l'algorithme gère la**
   cartographie de plusieurs blocs de 4 Ko.
   Nous avons pensé à implémenter cette idée, car cela réglerait le problème du programmeur qui demande une taille immense à *mymalloc*. Cependant, cette implémentation est assez difficile. Nous avons essayé de construire des nouvelles cartographies qui se pointe un à la suite de l'autre comme une liste chaînée simple. Pour implémenter cette idée, nous avons utilisé une boucle qui itère sur la taille restante après la soustraction de 4096. Dans chaque boucle, on prenait le pointeur actuel et on le pointait à une nouvelle cartographie de 4ko que nous avions créée peu avant. Un pointeur au début de tous ces blocs enchaînés était retourné.
   Cependant, cela ne marchait pas quand on faisait le test 4 du fichier *test.c*. Le programme plantait.
   On sait que c'est ça qu'il faut faire (par internet et la logique), mais nous n'avons pas réussi.

3. **La mémoire n'est jamais retournée via le free de la librairie standard. Si il y a un pic d'utilisation à 2 Go, vous aurez toujours tous les blocs de 4 Ko correspondants même après les appels à free.** De toute façon, la fonction *free* de la librairie standard n'efface même pas les pointeurs qui ont été libéré : on peut toujours accéder à leur valeur, mais c'est du charabia. Mon point est que même si nos blocs de 4 Ko sont toujours disponibles, *free* ne peut même pas effacer entièrement les pointeurs, donc je ne vois en quoi avoir nos 4ko malgré l'appel à *myfree* est un problème. C'est vrai que c'est de la mémoire gaspillée, mais que peut-on faire ?
   De plus, notre code n'a pas ce genre de problème, car il y a seulement une seule cartographie de 4ko.

4. **Les adresses retournées par votre libraire sont-elles toujours alignées sur un multiple quelconque ?**
   Non, ils sont alignés selon la taille de chaque bloc et chaque bloc a une taille différente. Notre bloc de 4ko n'est pas coupé en sous bloc de taille fixe, le code s'adapte à ce que le programmeur demande.
   Si nous avions réussi la liste chaînée des cartographies, nos adresses seront alignées en multiple de 4ko.

5. **Que se passe-t-il si refinc reçoit un pointeur qui n'a pas été retourné par votre malloc ? Quel est alors sa valeur de retour ?**
   Elle sera 0. Si le pointeur n'est pas retourné par *mymalloc*, notre fonction imprimera un
   message d'erreur et la valeur de défault retourné est 0.

6. **Que se passe-t-il si refinc reçoit un pointeur qui a été retourné par votre malloc, mais qui a déjà été désalloué car son compteur était à zéro ? Quel est alors sa valeur de retour ?**
   Comme la valeur de chaque pointeur est accessible, même après *free*, la valeur de retour sera la taille du bloc d'informations. En effet, la mémoire a été désalloué,

mais la valeur est encore accessible. Donc notre fonction *refinc* prendra la valeur du pointeur comme si la mémoire n'avait pas été désalloué.

7. **Où se trouve en mémoire la liste qui contient l'ensemble des adresses retournées par votre librairie ainsi que leur taille associée et leur compteur ? Si à chaque fois que vous ajoutez un élément vous faites appel au malloc de la librairie standard et à chaque fois que vous enlevez un élément vous faites appel au free, vous n'êtes pas en train de minimiser ces appels.**
   Il y a une variable globale qui s'appelle blocMemoire qui est dans le fond notre bloc de 4ko et une autre variable globale qui s'appelle listeVide qui pointe au début de notre bloc mémoire. Donc, si on a besoin d'accéder à un bloc d'informations, on n'a qu'a itérer sur chaque élément de notre bloc de 4ko.

- **Votre algorithme réserve-t-il toujours la taille exactement demandée par le programmeur ou arrondit-il à un nombre d'octets près (4 ou 8 par exemple) ?**
  Il réserve exactement la taille demandée. Si la taille demandée est plus petite que la taille du bloc d'information, on coupe le bloc d'information et on donne ce que le programmeur a besoin. Si la taille est plus grande que 4ko, on refusera l'allocation de mémoire.

- **Quel est la quantité de mémoire demandée à malloc de la librairie standard ? Que se passe-t-il si le programmeur demande 100 Mo de mémoire à votre malloc d'un coup ?** Notre code ne fait pas appel à la librairie standard. C'est comme ci on avait implémenté la vraie librairie standard. Si le programmeur demande 100 Mo d'un coup, on refusera cette demande et on retournera le pointeur vide. Le pourquoi est expliqué dans la question 1.

- **Y a-t-il des cas mal gérés, des effets de bords spéciaux ?**
  Il y a le cas de la mémoire et on l'a assumé plusieurs fois : il y a une limite de demande de mémoire (4ko). Il y a aussi le fait que notre *free* ne fonctionne pas comme la librairie standard et c'est dû au fait qu'on n'a pas intégrer un *garbage collector*. Quand on libère de la mémoire déjà libéré, notre programme fonctionne très bien. Or, le programme devrait exploser. Aussi, lorsqu'on libère un pointeur, la valeur du pointeur devrait être du n'importe quoi (*undefined behaviour*). Or, notre programme imprime toujours la valeur du pointeur qui lui était associé. Cependant, la mémoire est vraiment désalloué. Notre fonction *myfree* fait ce qu'il est sensé de faire : désalloué la mémoire. Il n'a juste pas le même comportement que *free* de la librairie standard, mais est très fonctionnelle.

- **Que se passe-t-il si free reçoit une adresse jamais retournée par malloc ?**
  Dans notre code, il fait juste imprimer un message d'erreur et cela fini là (le code ne plante pas). On sait que dans le *free* de la librairie standard, il devrait provoquer un *undefined behaviour* (brièvement mentionné dans la section 2.2.3). C'est lorsque le compilateur est libre d'assumer ce qu'il veut, car le bout de code n'est pas définit

par le langage C# standard. Le compilateur ne fait qu'exécuter des instructions à la lettres et lorsqu'il voit

du code, normalement, il sait comment réagir. C'est comme ci en voyant un *free* avec un pointeur non retourné par *malloc*, le compilateur ne sait pas quoi faire.

- **Avez-vous fait des tests unitaires ? Comment êtes-vous certains que votre code ne contient pas de bug ?**
  Il y a des tests dans *main* qui tests diverse choses. Le mieux qu'on puisse faire pour les *bugs* est de tester tous les aspects de chaque fonction. Il y a aussi des tests par un fichier nommé *tests.c*. Dans ce fichier, on a réussi 3 tests sur 6 (car on ne peut pas gérer les grosses demandes de mémoire) et si on augmente notre cartographie à une très très grande valeur, on a réussi 5 tests sur 6 (le 6^{e} est fautif pour une raison inconnue).

3. Conclusion

L'idée de notre algorithme et de notre structure mémoire est très bon. C'est la même structure que le professeur nous a conseillé.
Notre fonction *mymalloc* fonctionne bien et optimise la mémoire, mais cela coûte un prix : il y a une limite de mémoire que le programmeur peut emprunter (4ko). Cependant, on peut faciliter changer la limite.
Notre fonction my*free* fonctionne très bien et fait ce qu'il est supposé de faire : désallouer de la mémoire. Cependant, cette fonction n'a pas le même comportement que *free* de la librairie standard. C'est à cause du manque d'implémentation d'un *garbage collector*. Les deux comportements différents à la librairie standard sont :

1. Lorsqu'une mémoire est libérée et qu'on veut la relibéré, le programme devrait exploser ; le nôtre n'explose pas.
2. La valeur d'un pointeur libéré devrait être du charabia ; la valeur du pointeur dans notre code reste toujours le même.

Notre fonction *refinc* fonctionne bien selon nous.