Université de Montréal

IFT 2035 – Concept de langages de programmation

TP 2 - Mymalloc

par :
Jean Laprés-Chartrand
André Lalonde

18 juillet 2017

Expérience de développement

L'expérience de développement a été beaucoup plus agréable pour mymalloc que pour mini-haskell. Étant donné que nous n'avions aucun canvas de base pour le code, nous avons d'abord du nous renseigner sur le fonctionnement de malloc. Puisque le TP consiste à effectuer de la gestion mémoire, alors il était impératif de commencer par **sys/mman.h** et de porter une attention particulière à mmap().

Une fois familier avec le sujet, nous avons prototypé plusieurs idées d'implémentation avant d'arriver au choix que nous avons fait. Puisque nous n'étions pas familier avec le langage C, nous avons choisi une méthode qui rendra facile l'implémentation de myfree() afin de simplifier grandement la tâche de développement du code. Avec une bonne idée en tête de ce que nous allions faire, nous sommes passés à la troisième étape : le pseudocode. Chacun de notre côté nous avons développer un code, et bien que l'un était plus en forme de code, nous avons décidé de suivre l'idée qui était écrite en texte plutôt qu'en code puisque la cohérance était mieux justifiée.

Nous avons fait face à plusieurs obstacles lors de la création du code. Premièrement, puisque notre méthode utilise une case mémoire de plus pour enregistrer la taille libre ou occupé qui la suit au début et à la fin de chaque page, ainsi qu'après chaque block mémoire écrit. Le problème étant la taille d'un int qui diffère d'une architecture à l'autre, nous avons décidé de faire un storage dans un short qui a une valeur maximale de 32665. Par contre, si les demandes sont grandes, nous ne les traitons pas. Puisque quelques bytes ne font pas une grande différence lorsqu'on effectue d'énormes demandes, nous avons du adapter notre code afin de traiter ces cas. Nous avons alors séparer la tâche en 2 grandes parties : les mallocs de petits blocs et les mallocs de grands blocs.

La partie qui constitue les demandes de petites tailles fut asser simple. Un seul grand problème est survenu lors de son implémentation (qui était avant myfree) : la valeur du short qui dit la taille libre (de la première page vide, soit 4080) fut changer à 4 lors des tests. Le problème étant le test 2 qui affecte cette valeur à la page, l'inexistance de myfree() ne libère jamais et donc la valeur reste celle là. Cependant, l'erreur était plutôt que la case a été modifié. Cela voulait donc dire que l'on passait en argument une case qui ne devait pas aparaître (il manquait la taille d'un short dans l'adresse). Après quelques modifications, nous sommes passé au cas suivant.

Pour les grandes tailles, nous avons simplement ajouter un type long au début du bloc et nous nous assurons de retourner un bloc complet (même si l'on peut avoir un peu de mémoire gaspiller) pour des raisons de simplifications (bien qu'il y ait d'autres avantages). Le principe étant le même ce ne fut pas une étape difficile.

Tel que prévu au départ, l'implémentation de myfree a été simple. Pour les grands blocs, la solution est triviale. Puisque l'on alloue des blocs complets, nous vérifions si l'adresse demander est un pointeur valide. Le cas échéant, on libère tout le bloc. Pour les myfree de petits blocs, étant donné que chaque bloc de donnée est précédé par un pointeur qui donne la taille de la doonnée, on connais donc le nombre d'espace à libérer. Suffit de modifier le nombre de cases libres pour pouvoir réinsérer des objets dans la page. Une amélioration potentielle est de vérifier si la page est vide après un free, pour la libérer entièrement de la mémoire. Avant de développer sur l'algorithme, voici un example des plages mémoires.

		ptr_a															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
		-13														0	NULL	blockCourt = ptr_a
	a = 2	2	a	a	-10											0	NULL	blockCourt = ptr_a
	b = 3	2	a	a	3	b	b	b	-6							0	NULL	blockCourt = ptr_a
	c = 1	2	a	a	3	b	b	b	1	c	-4					0	NULL	blockCourt = ptr_a
	d = 4	2	a	a	3	b	b	b	1	c	2	d	d	-1		0	NULL	blockCourt = ptr_a
	retirer c	2	a	a	3	b	b	b	-1		2	d	d	-1		0	NULL	blockCourt = ptr_a
	retirer b	2	a	a	-5						2	d	d	-1		0	NULL	blockCourt = ptr_a
	e = 6	2	a	a	-5						2	d	d	-1		0	ptr_b	blockCourt = ptr_a
	retirer d	2	a	a	-10									-1		0	ptr_b	blockCourt = ptr_a
	retirer a	-13												-1		0	ptr_b	blockCourt = ptr_a
	supression																	blockCourt = ptr_b
		ptr_b																
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	génération	-13														0	NULL	
	e = 6	6	e	e	e	e	e	e	-6							0	NULL	

Algorithme

L'algorithme choisi pour implémenter mymalloc et myfree est une méthode originale basé sur aucun modèle que nous avons trouvé. Par conséquent, nous allons expliquer toutes les parties du code dans la documentation et suggérer des améliorations potentielles.

Structure de page

Malheureusement, nous étions trop avancer dans le temps pour modifier le modèle et incorporer une structure pour les pages plûtot que de les gérer manuellement. La fonction de **generateur_new_page()** s'occupe donc de créer et d'assigner le modèle de base lorsque l'on crée une page. La structure est comme-ci :

1. Chaque page a une taille de 4096ko.
2. Les 2 premiers octets sont un short qui contient un pointeur dont la valeur stocké est initialement le nombre de cases libres dans la page en tant que nombre négatif (4084k).
3. Les 8 derniers octets sont un void pointer qui pointe vers la prochaine page (si il n'existe pas de prochaine page, la valeur est mise à null).
4. Les 2 octets précédent le void pointer est un short pointeur avec une valeur de 0 (cette valeur indique que l'on est à la fin d'une page, utilisé par l'algorithme de recherche de mémoire).
5. Lorsque l'on insère un bloc d'information dans la page, on met un pointeur devant le bloc indiquant la grosseur du bloc, et un pointeur après étant l'espace libre restant.

Un problème ici est le manque d'une définition de structure. Rien ne nous garanti que la page est bel et bien construite comme on le souhaite et qu'elle garde la structure désirée tout au long de son utilisation. Il est donc beaucoup plus facile de briser la structure en modifiant une mauvaise donnée. Une solution serait de créer une structure et de re-définir une majorité des fonctions, qui reviens à pratiquement faire un nouveau code en se servant de celui-ci comme "pseudocode".

Différence pour une structure de page d'un grand bloc

Pour un bloc d'une grandeur supérieure à 3ko, nous utilisons une structure légèrement différente. Bien qu'on pourrait faire une énumération du type dans un struct pour voir la différence du type de structure, nous simulons ceci à l'aide d'une modification simple. La première donnée de la page est un long pointeur qui prend la taille totale du bloc comme donnée. La deuxième est un short pointer, d'une valeur de -4099 et la dernière est un void pointer qui pointe vers la prochaine page de format de gros bloc. Le short de -4099 est là pour pouvoir différencier un gros bloc avec un petit bloc.

mymalloc

La fonction de malloc est en fait un embranchement qui détermine comment est-ce que l'on va alouer la mémoire. On prends les cas inférieur à 3ko et les cas supérieurs à 3ko comme étant deux divisions possibles. La raison pour le 3ko est un treshold de 75%. qui est un compromis entre le gaspillage

d'espace mémoire et la recherche d'espace libre. Noter que cette valeur est arbitraire et peut être modifiée selon les besoins de l'utilisateur. Dans le cas de petites pages, nous allons donc voir mypetitmalloc et dans le cas de grandes pages mygrandmalloc.

myPetitMalloc

Le bût de cette fonction est de minimiser le nombre d'appels à mmap() puisque celles-ci sont plutôt lentes. Par conséquent, nous allons appeler mmap uniquement lorsque nous allons avoir besoin de créer une nouvelle page (espace joint trop petit dans les pages existentes). Le premier appel d'un petit malloc va créer une page et inscrire son adresse dans un pointeur global. Ce pointeur sera utile comme étant la tête d'une liste simplement chaînée. Puisque nous voulons éviter d'apeller mmap autant que possible, nous allons parcourir les pages une à une, sautant d'un pointeur à l'autre en vérifiant s'il y a asser d'espace continue pour subvenir aux besoins de la requête. En l'absence de page suivante, on en crée une nouvelle, modifi le pointeur de la dernière page pour qu'elle pointe vers la nouvelle, et puisqu'elle contiendra forcément asser d'espace (4ko >MAX(3ko)) nous pouvons effectuer la recherche à nouveau en utilisant le pointeur vers la nouvelle page.

smallSearch

La fonction de searchSmall qui est appelée par myPetitMalloc est une fonction récursive qui saute d'un pointeur à l'autre. Elle utilise l'implémentation des pointeurs avant/après les blocs de mémoires ainsi qu'au début/fin de page afin de sauter d'un bloc à l'autre. Dans ce cas, le temps de recherche croît linéairement avec le nombre d'allocations de petits mallocs. Une optimization possible de récupération de mémoire de cette partie est une gestion des pages (par examples on enlève une page vide en modifiant les pointeurs nécéssaire, réallocation de blocs utilisé fréquemment dans des trous libéré au début de la liste, permutation et placement de blocs dans l'espace par un niveau de priorité demandé au malloc, inclusion de blocs static : si l'on sait qu'on les conserves jusqu'à la fin de l'exécution du programme, alors on les met au début de la liste et on considère la liste comme débutant après eux dans la liste de recherche).

myGrosMalloc

La gestion des gros blocs mémoires est différente. Il y a cependant une similarité : nous gardons en mémoire un pointeur dans la dernière case du bloc qui pointe vers le prochain. Lors de l'allocation, la première donnée est un long qui contient une valeur égal à la grandeur du bloc. Le short suivant est un 0 utilisé pour l'identification du type de bloc lors d'un free. L'implémentation nécéssite un appel à mmap() par bloc demandé, ce qui est un stricte minimum. Le parcours pour trouver un bloc libre se fait

par saut de pointeur de bloc en bloc.

myfree

Comme la fonction mymalloc, la fonction myfree sert simplement à choisir quel type de free qu'on va faire. En se servant du pointeur passer en paramètre, on vérifie si on a le cas d'un gros bloc ou d'un petit bloc dans une page. Celui-ci est effectuer grâce à la valeur qu'on a mis dans les short, ainsi que le int existant dans les gros blocs. On appelle ensuite la version myfreePetit ou bien myfreeGros en conséquence pour libérer la mémoire.

myfreePetit

Pour libérer l'espace, on effectue une recherche aux travers des pages en se basant sur la valeur du pointeur. Puisque toutes les pages ont la même taille, nous pouvons savoir quand nous sommes sur la bonne page. Le cas échéant, nous sautons de pointeur en pointeur jusqu'à ce que l'on arrive à celui voulu. Une fois rendu, on inverse la valeur à l'intérieur pour dire que les espaces suivants seront libre. Pour des fins pratiques, nous effectuons une autre recherche pour les pointeurs précédent et suivant celui-ci. Si le pointeur précédent est négatif, alors on ajoute à sa valeur le nombre de cases qu'on viens de libérer plus la valeur d'un short. Si le suivant est négatif, alors on ajoute sa valeur ainsi que la valeur d'un short à la valeur storer dans la case du pointeur initiale. Ceci est fait par la fonction reuniteEmptySpace. Si la demande n'est pas sur les pages, alors l'algorithme ne fait rien.

myfreeGros

Premièrement, le myfreeGros débute au premier bloc et saute de bloc en bloc jusqu'à ce qu'il arrive au bloc demandé. Il utilise un pointeur temporaire pour se souvenir du bloc précédent (afin de pouvoir modifier le pointeur de la dernière case qui pointe au bloc suivant). Lorsque l'on arrive au bloc demandé, on modifie le pointeur de la dernière case du bloc précédent pour pointer vers celui d'après puis on appel munmap pour libérer ce bloc. Un des avantages du stockage de pointeur est la facilité de retourner l'espace au système.

Known bugs

Il y a présentement un bug dans le code remis. Celui-ci (qui aurait pu être éviter par l'utilisation d'un struct) survient lors de la ré-allocation de la mémoire dans un petit malloc. Lorsque l'on fait appel à myfree et que l'on modifie la valeur des pointeurs dans les shorts pour s'assurer de bien compter le nombre de données, une erreure est insérer. Étant donné que l'on ne garde pas la bonne valeur en mémoire, lorsque l'on attribut un premier objet dans une case libérer tout va bien. Par contre, la deuxième réallocation ne s'effectue pas correctement (soit par manque d'espace, ou bien parce que l'on arrive pas sur un bon short). Par conséquent, la ré-allocation de mémoire de petite taille est défectueuse.

Une piste de solution pour ce problème est de s'assurer que les bonnes valeurs sont insérer dans les cases lors de la libération.