Concepts des langages de programmation

TP 2

Michael Marchand p1009840
Nahuel Londono p1169038

1er août 2019

Table des matières

1 Expérience de développement : Michael

Commençons par le positif : ce n'était pas du Haskell ! En ce qui me concerne, je crois que l'utilité principale du tp1 fut d'augmenter mon appréciation pour tous les autres langages de programmation. Le TP2 fut ma première vraie utilisation du langage C et je dois dire que ce n'était pas aussi pénible qu'anticipé. La matière est intéressante et les problèmes que nous avions à résoudre demandent de la réflexion plutôt que de seulement être longs et mécaniques (je te regarde, architecture des ordinateurs !).

Malheureusement, nous avons manqué de temps et nous avons donc été forcés de couper les coins à plusieurs endroits (plus de détails dans la suite de ce rapport). L'extension de la remise fut vraiment appréciée cela m'a permis d'étudier convenablement pour les examens.

En ce qui concerne l'implémentation, mon inexpérience en matière du langage C (et de programmation en général, pour être parfaitement honnête) c'est fait fortement ressentir. Comme dans le TP1(mais beaucoup moins pire), la pensée qui résume le mieux l'expérience de développement pour moi est : "je sais ce que je dois faire, mais je n'arrive simplement pas à le traduire correctement en code". La possibilité d'aller chercher une bonne partie des points quand même avec l'aide de ce rapport est vraiment appréciée.

Notre programme a eu une crise lorsque les tests unitaires furent mis disponible. Nous avions une version complète qu'on croyais fonctionelle (nous avions crée nos propres tests ainsi qu'une fonction permettant de voir l'état de notre mémoire) hélas elle ne passait pas les tests unitaires donnés par le professeur. Au final on a tout refait du début pour aboutir à une version qui elle passe les tests (ou plutôt, passerait les tests si on les rouleraient pendant plusieurs cycles de vie de l'univers, plus de détails plus bas). Cela dit, une chance qu'on nous a donné plusieurs tests, on s'en allait remettre une version qui clairement ne remplissait pas les exigences.

Pour conclure, je garde une bonne mémoire de ce TP. Bien que j'aurais vraiment aimé avoir plus de temps pour remettre une meilleure version, j'ai tout de même l'impression d'avoir beaucoup appris sur des concepts pertinents ainsi que sur C.

2 Expérience de développement : Nahuel

Pour ma part comme avec tout nouveau projet, la première difficulté rencontrée a été le choix d'IDE et le téléchargement des logiciels nécessaires. Après plusieurs recherches j'ai trouvé une solution convenable à mon environnement Windows.

Cette tâche frustrante accomplie, je me suis lancé sur le développement en suivant les conseils du professeur. J'ai implémenté ma propre version des sections 3.2 et 3.3 dans ce même ordre, afin de bâtir des bases solides et m'assurer de bien comprendre ce que je devais faire.

Pour être honnête, les concepts d'allocation de mémoire dynamique et de pointeurs ne m'étaient pas tant familiers. En dehors de ce qu'on avait vu en cours bien sûr. Ceci dit, après mon implémentation très simple de la section 3.3 je me suis retrouvé perdu. Cependant, mon coéquipier a partagé avec moi plusieurs sources d'information sur internet à ce sujet et j'ai pu me renseigner sur les différentes approches qu'on peut adopter face à ce genre de problème (mymalloc/myfree).

Au début, on a essayé d'implémenter mymalloc avec une liste doublement chaînée, ce qui a marché bien pour nous. À vrai dire on avait fait une implémentation assez complète. Après plusieurs tests effectués, pas très sophistiqués (avec du recul), on pensait avoir fini le travail.

Cependant, trois jours avant la remise le professeur a mis en ligne un fichier de tests unitaires qui nous à tout fait remettre en question. En effet, selon le fichier de tests notre implémentation n'était pas très compétente. On a donc dû dans un temps très restreint tout changer et implémenter mymalloc et myfreee d'une autre forme.

On en est donc arrivés avec une version moins robuste que celle qu'on avait avant, mais celle-ci passe tous les tests. Ne m'interprétez pas mal, notre nouvelle implémentation est fonctionnelle et bonne, mais on est conscients de ses lacunes et qu'est-ce qu'on aurait pu faire pour l'améliorer.

Ceci dit, j'en ressors avec une meilleure compréhension théorique et pratique de cette gestion de mémoire dynamique qui nous a tant donné de frustrations. Néanmoins, on est reconnaissants d'avoir eu la chance de tout pouvoir apprendre cela et de pouvoir vous présenter notre propre implémentation de malloc() et free().

3 Documentation : l'algorithme

Pour notre algorithme, nous avons opter pour une stratégie first-fit avec une liste explicite simplement chaînée. La liste est explicite, car nous gardons un pointeur vers le prochain bloc de la liste plutôt que de la traverser en utilisant les tailles des blocs. Idéaallement, pour accélérer la recherche, nous aurions aussi un pointeur pour le prochain bloc vide, cela nous permetterais de passer par-dessus les blocs utilisés lorsqu'on fait appel à malloc. Malheureusemet, notre implémentation utilisant cette méthode avait des problèmes de pointeurs que nous n'avons pas réussi à régler. Nous avons donc opter de remettre une version sans les liens au prochain free bloc, ralentissant grandement notre algorithme. En effet, si nous devons ajouter n noeuds à la fin de la liste nous allons prendre un temps d'ordre n^2 pour le faire, ce qui rend l'algorithme trop lent en pratique si le nombre de requête est très élevée. Un avantage mineur de cette implémentation est que nous n'avons pas besoin d'espace pour le pointeur vers le prochain bloc libre, sauvant de l'espace.

Nous avons choisi la stratégie first-fit pour son excellent avantage principale : sa facilité d'implémentation. Selon nous, un avantage qui rend toute autre considération négligeable (dans le contexte d'un tp qui fait compétition dans le temps avec la préparation aux examens). Les autres avantages de cette stratégie est qu'elle est plus rapide qu'une stratégie best-fit et donne une liste qui est moins fragmentée qu'une stratégie next-fit. Le désavantage principal de cette stratégie est que la liste est souvent très fragmentée au début.

Nous avons une variable globale upgradeSize qui détermine la taille de l'appel du malloc de la librarie standard que notre programme va faire. Initialement, cette valeur est de 4ko. Cependant, nous avons décidé de doubler cette valeur après chaque appel de malloc que nous exécutons pour avoir un bon compromis entre le nombre d'appels à malloc nécessaire et avoir trop d'espace inutilisé causé par de trop grands appels à malloc. Si l'utilisateur demande un bloc plus grand que upgradeSize d'un coup, alors on assigne directement upgradeSize égal à la demande de l'utilisateur pour pouvoir gérer la requête.

Pour déterminer quels blocs sont alloués et lesquels sont vides, nous gardons dans chaque bloc un compteur indiquant le nombre de références vers cet emplacement mémoire l'utilisateur a demandé. Lors du premier appel à mymalloc, nous assignons le nombre de références à un. Lorsque l'utilisateur utilise refinc avec le pointeur que nous lui avons retourné, on incrémente le compteur de 1. Similairement, lorsque l'utilisateur utilise free avec ce pointeur, on décrémente le compteur de référence de un (jusqu'à un minimum de 0). Un bloc ne contenant aucune référence sera considéré par notre algorithme comme un bloc vide.

Pour ce qui est de la mémoire qui a été libérée par free(), nous ne redonnons jamais le moindre bit de mémoire au système d'exploitation. C'est notre mémoire, on la garde. Nous préférons la réutiliser lors de nouveaux appels de mymalloc.

En ce qui concerne l'alignement, nous avons décidé de tout aligner sur des multiples de 16 pour améliorer le temps d'exécution de certaines opérations système.

4 Documentation de mymalloc

mymalloc prends comme argument un size_t représentant la taille demandée par l'utilisateur et il doit retourner un pointeur void* vers une plage mémoire allouée que l'utilisateur peut utiliser. Notre implémentation commence par prendre la taille passée en paramètre et l'arrondir sur un multiple de 16. Ensuite, on regarde si c'est le premier appel à mymalloc. Si c'est le cas, nous devons créer le premier noeud de la liste, root, contenant la plage mémoire demandée par l'utilisateur et pointant vers un noeud vide contenant le reste de la mémoire qu'on a demandé en trop. Le compteur des références de root est initialisé à un et celui du noeud vide est initialisé à zéro.

Pour tous les autres appels à mymalloc, nous allons traverser notre liste chaînée à la recherche d'un emplacement vide (c'est-à-dire, un où le compteur de références est égal à zéro) assez grand pour contenir la plage mémoire demandée par l'utilisateur. Ici la fonction se divise en deux cas : le cas où le premier élément libre assez grand est dans la liste et le cas où il n'y a pas de bloc libre assez grand pour la demande de l'utilisateur.

Dans le premier cas, on regarde la taille du bloc trouvé. Si la taille du bloc est exactement la bonne, on augmente le compteur de référence de 1(indiquant que le bloc n'est plus libre) et on retourne un pointeur vers la plage mémoire associée à ce bloc. Si le bloc est plus grand que la taille demandée, alors on sépare le bloc en deux : une partie contenant le bloc avec la plage mémoire demandée par l'utilisateur et un nouveau bloc vide contenant le reste de l'espace libre.

Dans le deuxième cas, nous n'avons pas trouvé de bloc assez grand pour la requête. Nous devons donc faire un nouveau appel au malloc de la librairie standard. L'appel se fait avec la taille de upgradeSize (une variable globale), qui sera alors doublée après l'appel de malloc. Comme dans le cas de l'initialisation, on crée deux blocs : l'un contenant la plage demandée par l'utilisateur et l'autre un bloc vide contenant ce qu'on n'a pas utilisé de notre appel à malloc.

Le seul cas spécial que nous devons gérerr avec un appel à mymalloc est le cas où la taille demandée est plus grande que la taille de upgradeSize. Dans ce cas, on met simplement upgradeSize égal à la taille de la requête et on continue normalement. À noter que cette stratégie implique que tous les prochains appels à malloc de la librarie standard vont être au moins aussi gros que la taille de la plus grande requête, cela ne devrait pas être un problème dans la grande majorité des cas puisque de toute façon nous pouvons utiliser l'espace en trop reçu par malloc pour de plus petites valeurs en faisant la séparation en deux d'un bloc vide trop gros.

5 Documentation : myfree

myfree prend comme argument un pointer void*, idéalement l'un que nous avons retourné avec mymalloc, mais cela n'est pas vérifié. S'il advenais que l'utilisateur appelle myfree avec un pointeur que nous n'avons pas retourné avec mymalloc, tout ce qui se passerait c'est qu'on rechercherait le pointeur dans notre liste sans succèss. Si l'appel est fait avec le pointer NULL, alors on retourne immédiatement sans rien changer et sans parcourir inutilement la liste.

myfree ne retourne rien à l'utilisateur. Par contre, lorsque appelé avec un pointeur que nous avons retourné avec mymalloc, myfree va parcourir notre liste simplement chaîné pour retrouver le bloc contenant le metadata associé à ce pointeur. Lorsque ce bloc est trouvé, on décrémente le nombre de références associées à bloc puis on retourne (pas besoin de parcourir le reste de la liste).

Si lorsqu'on décrémente le nombre de références celui-ci tombe à zéro, ce-dernier est alors considéré comme vide par notre algorithme. Nous allons ensuite regarder si le prochain bloc est lui aussi vide. Si c'est le cas, nous allons combiner les deux blocs pour nous retrouver avec un bloc vide de taille égale à la somme des deux. Puisque nous utilisons une liste simplement chaîné et qu'on regarde seulement par en avant, le cas où la racine de la liste tombe à zéro n'est pas problématique (on ne risque pas d'accidentellement fusionner la racine avec le deuxième noeud et d'ainsi perdre la racine). Le cas où les références du dernier noeud tombent à zéro est un cas spécial, on n'essaye pas de fusionner avec le prochain bloc (ceci causerait un crash puisqu'on essayerait d'accéder au champs référence du pointer égal à NULL).

Un autre cas qui aurait pu causer problème est celui d'un appel à myfree() avec un pointeur vers un bloc qui est déjà vide (références == 0). Pour ne pas avoir ce problème, on évite de décrémenter le champs références si celui-ci est déjà a zéro.

Le fusionnement se fait simplement en ajustant le pointeur next du noeud actuel vers le noeud next du prochain noeud et en ajoutant la taille du noeud actuel la taille du prochain noeud (celui qui se fait absorber par le noeud actuel).

6 Documentation : refinc

refinc prend comme argument un pointeur void* et retourne un int représentant le nombre de références que le bloc associé au pointeur possède après avoir été augmenté de 1. Si l'utilisateur appelle refinc avec un pointeur que nous n'avons pas retourné avec mymalloc, on traverse notre liste sans trouver le bloc associé au pointeur. Puis, lorsqu'on arrive au bloc NULL, on retourne l'int 0 (une valeur représentant que quelque chose a mal tourné, puisque sous utilisation normale de refinc, refinc devrait toujours retourner au moins 1)

Si l'appel est fait avec le pointeur NULL, alors on retourne immédiatement sans rien changer et sans parcourir inutilement la liste. Encore une fois en retournant l'int 0.

Le cas où l'utilisateur appelle refinc avec un pointeur qui a déjà été libéré avec free() est un cas spécialement important puisque le bloc associé avec le pointeur pourrait exister ou pas (dans le cas où il a été fusionné avec le bloc précédent). Dans les deux cas, on ne veut pas augmenter le nombre de références du bloc puisqu'utiliser refinc sur un pointeur qui a déjà été libéré est une erreur de l'utilisateur. On se contente de retourner l'int 0 si le bloc trouvé par refinc contient déjà zéro références sans rien changer d'autre.

7 Documentation : testing

Pour des fins de débogage, nous avons fait une fonction show() qui traverse l'entièreté de notre liste et output à la console le contenu de chaque noeud (son adresse, l'addresse du prochaine noeud, l'adresse de la plage mémoire réservée pour l'utilisateur et la taille de l'adresse mémoire réservée pour l'utilisateur). Nous avions aussi fait nos propres tests avant que ceux donné par le prof ne soient disponibles. Malheureusement, nos tests passaient à merveille tandis que ceux du prof échouaient, indiquant que nos tests n'étaient pas bons (lesson learned !). Nous aurions aimé en créer de nouveaux plus stricte, mais faute de temps, ceci n'a pas pu être fait.

En ce qui concerne les tests du prof, la brutale inefficacité de notre algorithme rend impossible de vérifier que le test6 fonctionne correctement. Cela dit, lorsqu'on modifie le test : lorsqu'on enlève une multiplication par 1024, le test est capable de finir bien que évidemment avec la mauvaise solution. Nous croyons que si on donne assez de temps à notre programme, quelques fois l'âge de l'univers, le test finirait par passer avec la bonne réponse. Nous ne pouvons évidemment pas être certains de cela mais c'est le mieux qu'on a pu faire.

Nous ne testons pas non plus refinc ainsi que beaucoup de cas spéciaux outre que de manuellement inspecter la liste avec notre fonction show(). Clairement le programme manque de tests. Avec plus de temps ce serait la prochaine étape à améliorer.