TP 2 - Malloc et comptage de références

IFT 2035 - Été 2019

28 juin 2019

Vous devez faire le TP en équipe de deux.
Vous devez remettre votre travail pour le vendredi 26 juillet 23h59.

1 Introduction

La gestion mémoire en C se fait manuellement et très souvent par les fonctions `malloc` et `free`. Ces deux fonctions ne sont pas des primitives du compilateur mais bien des fonctions fournies dans la librairie standard. Il est donc tout à fait possible d'écrire votre propre version de ces fonctions.

Dans le but de se familiariser avec la gestion mémoire en C, vous allez dans ce travail réimplanter les fonctions `malloc` et `free`. Vous allez aussi devoir écrire un rapport décrivant votre travail sous la forme d'une documentation pour les utilisateurs.

2 Malloc et free

Le concept de malloc et free est relativement simple. Lorsque le programmeur fait appel à `malloc`, il désire obtenir une plage mémoire contiguë d'une certaine taille. La taille voulue, en octet, est passée en paramètre à `malloc`. Il revient à `malloc` de trouver un espace en mémoire assez grand pour les besoins du programmeur et de retourner l'adresse qui correspond au début de cette plage mémoire.

Lorsque le programmeur n'a plus besoin de l'espace mémoire, il doit appeler `free` et lui fournir l'adresse qu'il avait reçue de `malloc`. Ceci ne fait qu'indiquer à la libraire que cet espace mémoire n'est plus utilisé. Les librairies malloc ne sont pas obligées de retourner la mémoire libérée au système d'exploitation.

2.1 `malloc`, `mmap` et `brk`

Évidemment une libairie `malloc` ne peut pas se fier sur l'existance de `malloc` pour obtenir de la mémoire. Pour ce faire il existe des primitives fournis par les systèmes d'exploitations, par exemple `mmap` et `brk`. Un appel à ces primitives

est relativement lent, car cela implique beaucoup de travail pour le système d'exploitation. Une bonne librairie minimisera ces appels.

Pour faciliter votre travail, il vous est permis d'utiliser le `malloc` et le `free` de la librairie standard, mais vous devez *minimiser* ces appels comme s'il s'agissait de `mmap` et `brk`.

3 Votre travail

Vous devez implanter une version particulière de la librairie `malloc` qui fournit également une gestion mémoire par comptage de références. Concrètement, vous devez implanter les trois fonctions `mymalloc`, `refinc` et `myfree` défini dans le fichier myalloc.c. Vous n'avez pas à modifier le fichier myalloc.h.

Lorsque le programmeur fait appel à `malloc` votre librairie doit retourner une plage mémoire assez large. De plus, vous savez alors qu'il n'y a qu'une seule référence vers cette plage mémoire. Lorsque le programmeur fait appel à `refinc` avec un pointeur déjà retourné par votre `malloc`, vous devez incrémenter un compteur interne à votre librairie indiquant qu'il y a une autre référence à cette plage mémoire. `refinc` retourne la valeur de ce compteur (après incrémentation). Lorsque le programmeur fait appel à `free` votre librairie décrémente le compteur et libère la mémoire si et seulement si ce dernier est à zéro.

Vous êtes libres de choisir l'algorithme de votre choix pour implanter ces trois fonctions et cela fait partie du TP 2 de devoir s'informer et réfléchir à comment implanter un `malloc` et `free`. Vous devez bien sûr citer dans votre rapport vos références.

Votre algorithme n'a pas besoin d'être très performant, mais votre librairie doit avoir les caractéristiques suivantes :

1. `malloc` doit allouer au minimum la taille demandée.
2. `malloc` retourne une adresse qui pointe vers une plage mémoire correctement allouée par le système d'exploitation.
3. Aucun autre appel à `malloc` retourne une adresse déjà retournée ou une adresse qui est à l'intérieur d'une plage désignée par une adresse déjà retournée sauf si cette adresse a été passée à `free` et que le compteur était à zéro.
4. `malloc` doit limiter l'usage de la mémoire autant que possible.
5. Un appel à `malloc` se termine quoi qu'il arrive et ne plante jamais. Si une erreur survient, `malloc` retourne le pointeur NULL.
6. `free` accepte le pointeur NULL comme argument et cela ne libère aucun espace mémoire utilisé.
7. `free` ne libère la mémoire que si le nombre de références atteint zéro. Toutefois, ceci est du point de vue du programmeur. Votre librairie peut conserver cette plage mémoire pour utilisation future.

8. Votre libraire fait un usage limité du `malloc` et `free` de la libraire standard.

Voici quelques exemples d'algorithmes, dont les deux premiers sont évidemment trop naïfs pour obtenir une bonne note.

3.1 `malloc` naïf

L'idée la plus simple est que chaque utilisation de votre `malloc` fasse un appel à `malloc` de la libraire standard et même chose pour `free`. Votre librairie utilise une stucture de donnée quelconque pour le compteur associé à chaque référence.

Cette solution ne limite pas les appels à `malloc` et `free`. Ceux-ci vous sont autorisés pour faciliter votre travail, mais doivent être vus comme des appels à `mmap` par exemple.

3.2 `malloc` sans free

La solution pour éviter de faire appel à `malloc` de la libraire standard à chaque allocation mémoire est de demander un "grand" bloc de mémoire au système d'exploitation et gérer soi-même le partitionnement de la mémoire à l'intérieur de ce bloc mémoire.

Votre `malloc` demande donc un bloc d'au moins 4k (4 * 1024 octets) au `malloc` de la librairie standard. Votre librairie possède un pointeur qui au début pointe sur le début du bloc. À chaque appel à `malloc`, vous retournez l'adresse du pointeur et vous l'incrémentez par la taille demandée pour que la prochaine demande obtienne un nouvel espace mémoire. Lorsque vous manquez d'espace, vous faites un deuxième appel à `malloc`.

Avec cet algorithme, vous avez limiter les appels à `malloc` de la librairie standard mais la mémoire n'est jamais récupérée.

3.3 `malloc` avec free

Votre `malloc` demande un bloc de 4 Ko (4 * 1024 octets) à `malloc`. Vous pouvez choisir une taille supérieure, mais attention un `malloc` qui fait des appels arbitraires de 10 Mo par exemple ne respecte pas le critère 4 énoncé au début de cette section.

Cette fois-ci vous maintenez en mémoire une liste qui contient l'ensemble des adresses retournées par votre librairie ainsi que leur taille associée et leur compteur. Vous avez donc une cartographie du bloc de 4 Ko. Lorsque le programmeur appelle `free`, vous décrémenter le compteur et possiblement enlevez cette adresse de la liste.

Lorsque le programmeur appelle `malloc`, vous cherchez la première région assez large et retournez l'adresse de cette région. Vous mettez à jour votre liste avec cette adresse et la taille demandée.

Cet algorithme réutilise la mémoire libérée avec `free` au lieu de faire plusieurs appels à `malloc` de la librairie standard. Un nouvel appel à `malloc` de la

librairie standard sera fait uniquement s'il n'y a aucune place disponible dans votre bloc actuel.

Toutefois, plusieurs questions restent en suspens et vous devez faire des choix dans votre TP 2 pour les régler ou volontairement les ignorer (et l'indiquer dans votre rapport) :

Par exemple :

1. Cet algorithme ne couvre pas le cas où le programmeur demande d'un seul coup plus que 4 Ko de mémoire.
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.
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`.
4. Les adresses retournées par votre libraire sont-elles toujours alignées sur un multiple quelconque ?
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 ?
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 ?
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.

3.4 Un autre algorithme

Vous pouvez implanter l'algorithme de la section précédente ou tout simplement un autre de votre choix.

4 Rapport - Documentation

Vous devez rédiger un rapport qui servira également de documentation. Votre rapport doit contenir au moins les points suivants :

1. Expliquer brièvement votre expérience de développement avec le TP 2 (1 à 2 pages). Par exemple, quelles difficultés avez-vous rencontrées ?

2. Vous devez documenter vos fonctions `mymalloc`, `refinc` et `myfree` ainsi que l'algorithme utilisé (3 à 6 pages). Votre documentation s'adresse à un autre informaticien qui a déjà des bases en C. Il ne s'agit pas d'un document marketing. Par exemple :

   - Expliquer pourquoi vous avez choisi cet algorithme et quelles sont ses caractéristiques (forces et faiblesses).
   - Vous pouvez illustrer graphiquement votre algorithme.
   - 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) ?
   - Les adresses retournées seront-elles alignées sur un multiple d'octets (4 ou 8 par exemple) ?
   - 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 ?
   - Y a-t-il des cas mal gérés, des effets de bords spéciaux ?
   - Que se passe-t-il si free reçoit une adresse jamais retournée par malloc ?
   - Comment le code est-il structuré ?
   - Avez-vous fait des tests unitaires ? Comment êtes-vous certains que votre code ne contient pas de bug ?

   Les points ci-dessous ne sont que des exemples. N'hésitez pas à ajouter à votre documentation ce qu'il vous semble essentiel de connaitre pour tout programmeur qui voudra utiliser, comprendre et/ou modifier votre librairie.

5 Évaluation

- Ce travail compte pour 20 points de la note finale du cours. Vous devez faire ce travail en équipe de deux. Vous devez m'avertir rapidement si vous ne trouvez pas de partenaire.
- Votre code sera évalué sur 8 points. Vous devez remettre *un et seulement un* fichier de code source nommé mymalloc.c qui implante le header mymalloc.h fourni avec cet énoncé.

  Votre code doit compiler, être structuré, lisible et commenté lorsque nécessaire. Il est possible que le correcteur fasse passer des tests unitaires à votre code.
- Le rapport sera évalué sur 12 points. Vous devez remettre un fichier PDF. Vous serez évalué sur la qualité de votre algorithme et de sa description (9 pts), votre expérience pour le TP 2 (2 pts) et la qualité du français (1 pt).