TP 2 - Malloc et free

IFT 2035 - Été 2017

20 juin 2017

Vous devez faire le TP en équipe de deux.
Vous devez remettre votre travail pour le mardi 18 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 votre 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 Zone mémoire

Lorsqu'un processus (programme) s'exécute, le système d'exploitation lui attribue différentes zones mémoires. Tel que vu en cours, la pile sert à stocker les variables locales et les paramètres, la zone statique les données constantes du programme et le programme lui-même. Le reste de la mémoire, souvent appelé le tas, est en fait conceptuellement divisé en deux parties. Il y a le heap (tas),

qui comme vue en classe est une grande zone mémoire contiguë. La primitive sous GNU/Linux pour connaitre l'adresse de fin du heap et également pour augmenter sa taille est `brk` (et `sbrk`).

L'autre région de la mémoire également considérée comme le tas est le memory mapping segment, zone où sont chargés en mémoire les fichiers par exemple. La primitive sous GNU/Linux pour demander un bloc mémoire d'une certaine taille dans le memory mapping segment est `mmap`.

Une très bonne explication de ces concepts est disponible sur le blogue de Gustavo Duarte.

2.2 Votre travail

Vous devez implanter les deux fonctions `malloc` et `free`. Il est possible de n'utiliser que `brk` ou que `mmap` pour obtenir les fonctionnalités d'un `malloc` ou une combinaison des deux. Dans ce TP, vous utiliserez *uniquement* `mmap`. De plus, votre librairie ne doit *jamais* utiliser le `malloc` de la librairie standard mais seulement la primitive `mmap`.

Vous êtes libres de choisir l'algorithme de votre choix 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 `malloc` 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`.
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.

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

2.2.1 `malloc` naïf

L'idée la plus simple est que chaque utilisation de `malloc` fasse un appel à `mmap` et chaque utilisation de `free` un appel à `munmap`.

Toutefois, un appel à `mmap` est relativement lent, car cela implique beaucoup de travail pour le système d'exploitation. La principale raison d'existence de `malloc` est d'éviter de faire plusieurs appels à `mmap` ou `brk`.

2.2.2 `malloc` sans free

La solution pour éviter de faire appel à `mmap` 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) à `mmap`. 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 `mmap`.

Avec cet algorithme, vous avez réglé le problème de vitesse par rapport à `mmap` mais la mémoire n'est jamais récupérée.

2.2.3 `malloc` avec free

Votre `malloc` demande un bloc de 4 Ko (4 * 1024 octets) à `mmap`. 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 et la taille associée. Vous avez donc une cartographie du bloc de 4 Ko. Lorsque le programmeur appelle `free`, vous 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 appel à `mmap`. Un nouvel appel à `mmap` 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. Comment gérer la mémoire nécessaire pour retenir les adresses retournées par `malloc` et leur taille associée (vous ne pouvez pas utiliser le `malloc` standard) ?

2. Cet algorithme ne couvre pas le cas où le programmeur demande d'un seul coup plus que 4 Ko de mémoire.

3. Lorsque l'utilisation mémoire requiert plusieurs blocs de 4 Ko, il faut que l'algorithme gère la cartographie de plusieurs blocs.

4. La mémoire n'est jamais retournée via `munmap`. 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`. *Il est permis pour le TP 2 ne négliger cet aspect comme le font certaines librairies malloc*

5. Les adresses retournées par votre libraire sont-elles toujours alignées sur un multiple quelconque ?

6. La fonction `malloc` reçoit en paramètre un nombre dont le type est `size_t` définie dans la libraire `<stddef.h>`. Que se passe-t-il si le nombre passé à malloc est plus grand que le nombre maximal autorisé par `size_t` ? Ce cas peut-être détecté, car un overflow se produit.

2.3 Pour nos amis sur Windows et Mac

La primitive `mmap` n'est pas disponible sur Windows. De plus, il se pourrait aussi que votre code C se comporte différemment sous GNU/Linux et Mac. Ainsi, votre code sera corrigé sur GNU/Linux et doit fonctionner sur les ordinateurs du DIRO.

Vous pouvez néanmoins développer votre code en utilisant le `malloc` de la libraire standard au lieu de `mmap` au début, mais vous aller devoir fournir un travail qui compile et s'exécute sur GNU/Linux sans aucun appel à `malloc`.

2.4 Tests unitaires

Des tests unitaires vous seront fournis pour tester votre implantation. Le succès auprès de tous les tests unitaires est un gage que votre algorithme se comporte bien, mais cela n'exclut pas l'existence de bugs.

3 Rapport - Documentation

Vous devez rédiger un rapport qui servira également de documentation. Votre rapport doit contenir 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 malloc et free 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 mmap ? Que se passe-t-il si le programmeur demande 100 Mo de mémoire à 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 ?
- Comment se comporte votre algorithme dans les tests unitaires ?

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.

4 Évaluation

- Ce travail compte pour 15 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 programme sera évalué sur 6 points. Vous devez remettre *un et seulement un* fichier nommé mymalloc.c qui implante le header mymalloc.h fourni avec cet énoncé.

  Votre libraire malloc doit réussir les tests unitaires (4 pts) et votre code doit être structuré, lisible et commenté lorsque nécessaire (2 pts).
- Le rapport sera évalué sur 9 points. Vous devez remettre un fichier PDF. Vous serez évalué sur la qualité de votre algorithme et de sa description (6 pts), votre expérience pour le TP 2 (2 pts) et la qualité du français (1 pt).
- Un travail qui ne fait qu' « optimiser » les tests unitaires sans vraiment être un `malloc` se verra attribuer la note de 0.