TP2 - Organisation de la mémoire: le tas

Guillaume Girol Matthieu Lemerre Grégoire Menguy

IN201 - Cours de système d'exploitation

Contents

1 Allocation mémoire triviale **2**

2 Allocation mémoire à liste **2**

2.1 Initialisation 2
2.2 Allocation 3
2.3 Libération 3
2.4 Coalescence 4

Le but de ce TP est d'écrire deux fonctions:

```
void * memalloc(int size);
void memfree(void *);
```

Tel que `memalloc()` renvoie un pointeur vers une zone de mémoire contigue fraîche de taille `size`, et `memfree(ptr)` rend possible le réemploi de la zone mémoire commencant par `ptr`.

Note 1. *Ces fonctions fournissent des fonctionnalités similaires aux fonctions `malloc` et `free` de la librairie standard C. Mais n'utilisez pas `malloc` et `free` dans le TP, puisque le but est de les ré-implémenter pour comprendre comment marchent ces fonctions!*

Question 1. *Allouez statiquement une zone mémoire de 16 kilo-octets.*

Ce tampon de mémoire est appellé le *tas* (heap en anglais). À tout moment, chaque octet qu'il contient sera soit:

- *libre*, c'est à dire qu'il peut être renvoyé par une future allocation.
- *alloué* dans le cas contraire.

1 Allocation mémoire triviale

Dans cette section, nous allons réaliser une première version d'un allocateur mémoire, qui ne permet que d'allouer la mémoire, sans jamais la libérer. Un tel allocateur est utile par exemple si on a besoin d'allouer des structures de données au début d'un programme et plus jamais pendant le reste de son exécution.

Question 2. *Implémentez une version de* `memalloc()` *telle que:*

- *À tout moment, l'ensemble des octets libres forme une seule zone contigüe (il n'y a pas de* fragmentation*)*
- *Le programme affiche une erreur et quitte si il n'y a plus de mémoire disponible.*
- `memfree` *ne fait rien.*

2 Allocation mémoire à liste

Nous allons maintenant réaliser un allocateur mémoire complet, supportant l'allocation aussi bien que la déallocation.

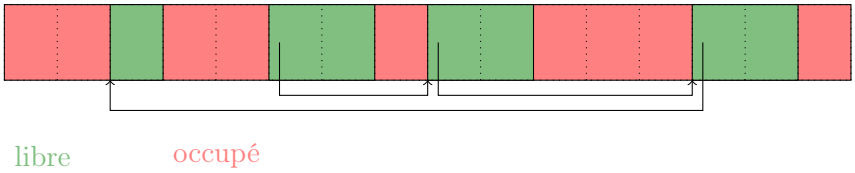
Question 3. *Cachez la première version de votre code au compilateur en utilisant les directives du pré-processeur* `#ifdef` *et* `#endif`*.*

On appelle une *zone* un ensemble contigu d'octets qui sont soit tous libres, soit qui ont été alloués par le même appel à `memalloc(size)`. La succession des opérations d'allocation et de libération font que l'ensemble des zones libres ne forme plus une seule zone contigüe. L'idée de l'allocateur à liste est de constituer une liste chaînée des zones libres (la *free list*). L'allocation consiste alors à parcourir la liste à la recherche d'une zone de taille assez grande pour contenir la taille à allouer; la désallocation consiste à ré-inserer la zone libérée dans la free list.

Pour que l'algorithme fonctionne, il est nécessaire de stocker des informations sur la structure du tas; c'est à dire, pour chaque zone libre, sa taille et un pointeur sur la zone suivante. Dans un allocateur à liste, ces informations pour une zone sont stockées en début de la zone libre.

2.1 Initialisation

Au début du programme, le tas contient une seule grande zone libre contenant tout le tas.



Question 4. *Écrivez une fonction* `void meminit(void)` *qui initialise le tas pour qu'il ne contienne qu'une seule grande zone libre.*

2.2 Allocation

L'allocation d'une taille n consiste à parcourir la liste chainée à la recherche d'une zone supérieur à n. Si la zone trouvée est de taille supérieure à la taille demandée, il faut d'abord la scinder en deux. Dans tout les cas, il faut retirer de la *free list* la portion de la mémoire qui a été allouée.

La stratégie *first-fit* consiste à s'arrêter dès qu'une zone de taille suffisament grande est trouvée.

Question 5. *Implémentez la fonction* `memalloc(size)` *de manière à ce que celle ci réalise une allocation first-fit. Si il n'y a pas de zone de taille suffisamment grande, renvoyez le pointeur 0.*

2.3 Libération

Avant de commencer, notons que `memfree` ne fournit pas la taille de la zone allouée, et que nous avons besoin de cette information pour remplir la free list. En général, nous allons avoir besoin de stocker des *métadonnées* associées à la zone allouée, contenant notamment cette taille. Pour cela, l'idée consiste à modifier l'allocation pour que:

- on alloue une une zone contenant la taille demandée + la taille des métadonnées;
- on stocke les métadonnées au début de la zone allouée;
- on renvoie un pointeur correspondant au début de la zone qui peut être écrite par l'utilisateur (juste après les métadonnées).

Question 6. *Modifier l'allocation first-fit pour permettre le stockage de métadonnées en début de zone, comprenant notamment la taille de la zone allouée.*

Question 7. *Implantez l'algorithme de libération de la mémoire (fonction* `memfree`*):*

1. *Retrouvez le pointeur de début de zone à partir de l'argument de* `memfree`
2. *Insérez la zone libérée dans la* free list.

Que se passe-t-il si vous libérez un pointeur déjà libéré? Ou une zone mémoire qui ne correspond pas à un pointeur alloué?

2.4 Coalescence

Question 8. *Quel problème (appellé fragmentation) pose le fait d'avoir plusieurs zones libres non contigües?*

Question 9. *Implémentez une mitigation simple à ce problème de la manière suivante: lorsque vous liberez une zone, parcourez la liste libre pour trouver si la zone suivante est une zone libre et si oui, fusionnez les deux zones libres contigües.*

Question 10. *Quelles sont les limitations de cette solution, et quelles mesures pourraient être prise pour les améliorer?*

s