UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Master M1 Informatique, CSE
TD & TP Allocation mémoire
Année 2017-2018

Résumé

Cette fiche sera traitée sur deux semaines. Vous traiterez en dehors des séances prévues les exercices et questions non résolues. Les objectifs de cette séance de TD/TP sont les suivants :
— implémenter un allocateur mémoire réaliste
— étudier les différentes stratégies d'allocation possibles
— développer des programmes de tests
— remplacer la bibliothèque standard dans des programmes normaux (`ls`, etc.)

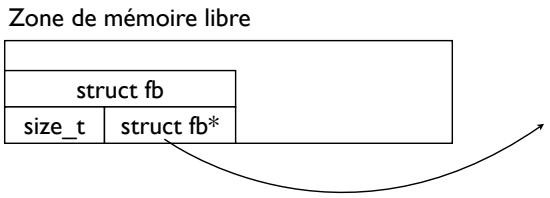
1 Introduction

Le système que nous nous proposons d'étudier possède un espace mémoire de taille fixée à l'initialisation. Cet espace correspond à la mémoire physique utilisable par le système ou au tas, utilisable dans un processus utilisateur. Le problème qui se pose est de fournir un mécanisme de gestion de la mémoire.

La gestion de la mémoire correspond à un suivi des zones de mémoire utilisées ainsi que de celles libres afin de répondre correctement aux demandes :
— d'allocation de nouvelle zone : il faudra dans ce cas choisir un bloc libre ou un morceau de bloc libre afin de le transformer en zone occupée ;
— de libération d'une zone précédemment allouée : il faudra dans ce cas que le zone en question redevienne libre et soit fusionnée, le cas échéant, avec les autres zones libres.

2 Mise en œuvre : chaînage des zones libres et allouées

Un algorithme de gestion de la mémoire repose sur le principe de chaînage des zones libres. À chaque zone libre est associé un descripteur qui contient sa taille et un lien de chaînage vers la zone libre suivante. **Ce descripteur est placé dans la zone de mémoire elle-même.**



L'algorithme d'allocation doit répondre aux demandes de l'utilisateur d'une nouvelle zone de taille `tailleZone` en trouvant parmi les blocs libres un bloc ayant une taille `z` suffisante, choisi selon l'une des politiques suivantes :

première zone libre (first fit) : on choisit la première zone `z` telle que `taille(z) >= tailleZone`. Ce choix vise à accélérer la recherche ;

meilleur ajustement (best fit) : on choisit la zone `z` donnant le plus petit résidu ; autrement dit, on choisit `z` telle que `taille(z) - tailleZone` soit minimal, ce qui impose de parcourir toute la liste ou de la maintenir classée par tailles. Ce choix vise à minimiser la taille du résidu ;

plus grand résidu (worst fit) : on choisit la zone `z` telle que
`taille(z) - tailleZone` soit maximal. Ce choix vise à maximiser le résidu en espérant qu'il puisse constituer une zone assez grande pour une future demande.

Lors de la libération d'une zone, celle-ci est réinsérée dans la liste et fusionnée, s'il y a lieu, avec la ou les zone(s) voisine(s). Cette fusion est facilitée si les zones sont rangées par adresses croissantes.

3 Travail demandé

Il est demandé de réaliser le gestionnaire d'allocation mémoire avec l'algorithme de chaînage des zones libres dans l'ordre des adresses croissantes et la politique **first fit**. **Attention :** vous ne devez pas modifier les fichiers fournis, excepté le `Makefile` que vous pouvez compléter. Votre point d'entrée dans le code fourni est le fichier `mem.c` que vous devez écrire en y plaçant la définition des fonctions déclarées dans `mem.h`.

Au départ, dans votre implémentation, la liste des blocs libres sera constituée d'une seule zone libre correspondant à l'ensemble de la zone mémoire gérée par l'allocateur [1]. La liste des blocs libres évoluera au fur et à mesure des allocations, mais les zones libres et occupées seront toujours à l'intérieur de la zone mémoire gérée par l'allocateur. Le descripteur d'une zone libre, en théorie placé au début de la zone, peut être déclaré en C par :

```
struct fb {  /* fb pour free block */
  size_t size ;
  struct fb *next ;
  /* ... */
};
```

Le type `size_t`, défini dans la bibliothèque `stddef.h`, est synonyme de unsigned long.

▷ **Question 1.** *Y a-t-il besoin de gérer une liste de zones occupées ?*

▷ **Question 2.** *Quelle est la structure d'une zone allouée ?*

3.1 Interface de l'allocateur

L'allocateur comportera les fonctions suivantes, que vous devez écrire dans `mem.c` :

`void mem_init(void *mem, size_t taille);`
Cette procédure initialise la liste des zones libres avec une seule zone correspondant à l'ensemble de la mémoire située en `mem` et de taille `taille` non utilisée par votre allocateur pour son propre usage. Lorsqu'on appelle `mem_init` alors que des allocations et libérations ont déjà été effectuées, l'ensemble de la structure de données est réinitialisé. La fonction de recherche est également (ré)initialisée à `mem_fit_first` par défaut.

`void mem_fit(struct fb* (*fit_function)(struct fb *tete_libres, size_t size));`
Permet de choisir la fonction de recherche.

`void *mem_alloc(size_t size);`
Cette procédure reçoit en paramètre la taille `size` de la zone à allouer. Elle retourne un pointeur vers la zone allouée et NULL en cas d'allocation impossible.

`void mem_free(void *zone);`
Cette procédure reçoit en paramètre l'adresse de la zone occupée. La taille de la zone est récupéré

1. La déclaration de la mémoire vous est fournie dans les fichiers `common.h` et `common.c`. Un programme peut en prendre connaissances avec les fonctions `get_memory_adr` et `get_memory_size`. Les valeurs retournées par ces deux fonctions sont censées être passées comme paramètres à la fonction `mem_init` décrite dans la partie décrivant l'interface avec l'allocateur. Le programme `memshell`, qui fournit une interface d'allocation / libération en ligne de commande vous permettant de tester votre allocateur, initialise la mémoire de cette manière.

en début de zone. La fonction met à jour la liste des zones libres **avec fusion des zones libres contiguës** si le cas se présente.

`size_t mem_get_size(void *zone);`
Cette procédure reçoit en paramètres l'adresse d'une zone allouée et renvoie le maximum d'octets que l'utilisateur peut stocker dans la zone. Elle est utilisée uniquement pour implémenter `realloc` dans le fichier `malloc_stub.c`.

`void mem_show(void (*print)(void *zone, size_t size, int free));`
Cette procédure doit parcourir l'ensemble des blocs gérés par l'allocateur et appeler pour chacun d'eux la procédure `print` donnée en paramètre. Les paramètres de `print` correspondent aux informations concernant le bloc pour lequel elle est appelée, à savoir : son adresse, sa taille, et un booléen indiquant s'il est libre (1) ou occupé (0). La procédure `mem_show` est utilisée uniquement par le programme `memshell`, elle lui sert à afficher l'état de la mémoire sans avoir besoin de connaître les détails d'implémentation de votre allocateur. Vous pouvez trouver dans `memshell.c` plusieurs exemples de procedures correspondant au paramètre `print`, l'une d'elles est :

```
void afficher_zone(void *adresse, size_t taille, int free) {
printf("Zone %s, Adresse : %lx, Taille : %lu\n", free?"libre":"occupee",
(unsigned long) adresse, (unsigned long) taille);
}
```

On pourra appeler `mem_show` par `mem_show(&afficher_zones)` comme cela est fait dans `memshell.c`.

`struct fb* mem_fit_first(struct fb *list, size_t size);`
Fonction renvoyant l'adresse du premier bloc libre de taille supérieure ou égale à `size` présent dans la liste de blocs libre dont l'adresse est `list`. Renvoie `NULL` si un tel bloc n'existe pas. Vous devez écrire cette fonction et c'est cette fonction que vous devez appeler dans `mem_alloc` pour choisir un bloc, sauf si l'utilisateur a indiqué vouloir en utiliser une autre à l'aide de `mem_fit`.

`struct fb* mem_fit_best(struct fb *list, size_t size);`
Fonction renvoyant l'adresse du plus petit bloc libre de taille supérieure ou égale à `size` présent dans la liste de blocs libre dont l'adresse est `list`. Cette fonction est utilisable comme paramètre de `mem_fit` et, dans ce cas, remplace la fonction existante (`mem_fit_first` par défaut).

`struct fb* mem_fit_worst(struct fb*, size_t);`
Fonction renvoyant l'adresse du plus grand bloc libre de taille supérieure ou égale à `size` présent dans la liste de blocs libre dont l'adresse est `list`. De façon analogue au cas de `mem_fit_first` et `mem_fit_best`, cette fonction est utilisable comme paramètre de `mem_fit`.

▷ **Question 3.** *Les adresses 0,1,2 sont-elles valides ? De manière générale, un utilisateur peut-il manipuler toutes les adresses ?*

▷ **Question 4.** *Quand on alloue une zone, quelle est l'adresse retournée à l'utilisateur ?*

▷ **Question 5.** *Quand on alloue dans une zone mémoire libre, il faut faire attention à la procédure de partitionnement. Dans le cas simple, on alloue le début de la zone pour nos besoins et la suite devient une zone de mémoire libre de taille (taille de la zone du début - taille allouée). Toutefois, il est possible que la taille qui reste soit trop petite. Pourquoi ? Comment gérer ce cas ?*

4 Travail à rendre

Vous trouverez sur Moodle une archive qui contient le squelette du programme de l'allocateur. Entre autres, vous avez un Makefile qui vous donne les règles de compilation de base et vous montre comment générer une bibliothèque partagée qui peut être utilisée par des programmes standard.

Vous devez rendre sur la plate-forme moodle **et en binôme** une archive contenant :
— les fichiers initialement fournis **non modifiés**, à l'exception du `Makefile` qui peut être complété ;

— le fichier `mem.c` qui implante les fonctions dont l'interface utilisateur `mem.h` vous est donnée ;
— des programmes de tests pertinents ;
— un `Makefile` tel que `make all` (qui doit être la cible par défaut) compile votre code ainsi que la bibliothèque partagée `libmalloc.so` et que `make tests` lance tous vos tests ;
— un rapport nommé `README.PDF` de **2 pages maximum** qui présente vos choix (justifiés) d'implantation, les fonctionnalités et limites de votre code, les extensions et les tests que vous avez réalisés.

Les questions précédentes ne sont là que pour vous guider. Il n'est pas demandé d'y répondre dans le rapport. La qualité du code fourni (indentation, commentaires, structuration, etc.) devra faire l'objet d'une attention particulière.

5 Pour aller plus loin

Voici quelques extensions possibles pour améliorer les fonctionnalités de votre allocateur mémoire. Implémentez les en fonction de votre temps disponible.

5.1 Débordement mémoire

On pourrait placer autour de chaque zone allouée des gardes. Si [2], lors de la libération, vous constatez qu'une garde est effacée, cela veut dire qu'il y a eu débordement de mémoire.

5.2 Corruption de l'allocateur

Dans le même esprit que l'extension précédente, proposez des moyens pour détecter et signaler des anomalies dans votre allocateur causées par une mauvaise utilisation de celui-ci. A titre d'exemple, la `libc` détecte les libérations multiples d'un même bloc et certains chaînages invalides causés par un écrasement des données de l'allocateur.

5.3 Autres politiques

Implémenter les politiques *best fit* et *worst fit*. Pour limiter les parcours de listes lors de la recherche de zone libre adéquate, on peut introduire deux systèmes de listes correspondant à un ordre par taille et par adresses. Quelles listes faut-il employer pour éviter de multiplier les parcours ? Quelle incidence cela a-t-il sur les performances ?

5.4 Comparaison entre politiques

Pour chaque politique d'allocation implémentée, trouvez un exemple pour lequel cette politique est meilleure que les autres. Remarque : on peut travailler sur cette extension sans avoir implémenté plusieurs politiques.

5.5 Compatibilité avec `valgrind`

Faites en sorte que votre implémentation soit compatible avec `valgrind` : autrement dit, faites en sorte que `Valgrind` soit capable de suivre vos allocations et libérations.

5.6 Autre extension

Proposez votre propre extension originale.

2. mais pas seulement si