ASSI Dima
COURTHIAL Samuel
GOMA MISSAMOU Yoann

Compte-rendu TP allocateur mémoire

Les structures que nous avons décidé d'utiliser sont

```
struct fb {
    size_t size;
    struct fb* next;
};
```

pour les zones libres et

```
struct ab {
    size_t size;
};
```

pour les zones allouées.

mem_init (void* mem, size_t taille) :

Cette méthode initialise la mémoire utilisable à partir de l'adresse et de la taille donnés en paramètre. Cette initialisation se réduit à correctement définir first_fb, le pointeur indiquant la première zone libre de la mémoire. Ici, nous initialisons donc first_fb à l'adresse mem avec une taille identique à celle donnée en paramètre moins la place que prend l'entête first_fb. Au début, il n'y a qu'une zone libre donc la zone suivante de first_fb n'existe pas. La méthode mem_fit permet de définir, dans ce cas la fonction mem_fit_first comme fonction de recherche d'un bloc libre à allouer.

mem_show(void (*print)(void* zone, size_t, int free)) :

Cette méthode parcours tous les blocs, libre ou alloué, et les affiche avec la fonction print donnée en paramètre. On détermine tout d'abord les adresses minimale (mem_adr) et maximale (end_mem_adr). Puis, on initialise la variable adr avec la valeur mem_adr. On initialise également un pointeur fb à la valeur first_fb. L'idée est la suivante :

- si adr = fb, adr pointe sur une zone libre. On peut donc créer un pointeur de struct fb pointant sur le début du bloc. On peut ainsi afficher avoir accés au paramètre nécessaire à la fonction print. Après cela, on ajoute la taille de la zone à adr pour atterrir sur la zone suivante. De même, on avance fb vers le prochain bloc libre.
- si adr != fb, adr pointe sur une zone occupée, on a alors encore une fois accés aux champs nécessaire pour la fonction print. Après cela, on ajoute la taille de la zone à adr pour avancer jusqu'à la zone suivante.

mem_fit (mem_fit_function *f) :

Cette méthode permet de donner la bonne valeur à la variable mem_fit_fn. Cette variable prend en valeur l'adresse de la fonction que l'on souhaite utilisé lors de l'initialisation. Par la suite, en faisant appel à mem_fit_fn, nous appellerons la fonction qui se trouve à l'adresse indiquée.

mem_alloc (size_t taille) :

La première chose à faire est de déterminer à partir de la taille voulu par l'utilisateur (donnée en paramètre) la taille de bloc à rechercher. Pour cela, des restrictions systèmes favorisent grandement l'alignement des adresses mémoires fournies par mem_alloc comme étant multiple d'une certaine constante (dans notre cas défini à 16). Cette alignement se fait via la macro ALIGN défini au début du fichier qui applique le bon masque pour transformer la taille voulue en le multiple de 16 directement supérieur.

Nous appelons alors mem_fit_fn (qui lance donc dans notre cas mem_fit_first) pour rechercher un bloc mémoire de taille suffisante et correspondant au critère de recherche défini par la fonction de recherche choisie (ici le premier bloc de taille suffisante). Nous traitons d'abord le cas NULL, si fb (le bloc retourné par mem_fit_fn) est NULL, il n'y a pas de mémoire à allouer. Sinon, nous enregistrons la taille (dans taille_ini) et le bloc libre suivant (dans next_fb) au cas où l'emplacement mémoire où ces informations sont stockées est modifiés avant qu'elle ne nous soit plus utiles. On détermine alors le bloc mémoire précédent pour permettre de raccorder les blocs libres restant entre eux.

La zone à retourner est appelée new_ab, struct ab étant la structure définissant un bloc alloué. Cette zone se situera donc à la même adresse que fb. On distingue alors 2 cas :

- Si on a la place de séparer le bloc mémoire désiré en 2 bloc (un alloué et un libre), il faut alors créer la nouvelle zone libre ainsi créée. Cette dernière, intitulée new_fb est placée à l'adresse fb + sizeof(struct ab) + taille. En effet, cette zone doit démarrer directement après la zone que l'on vient d'allouer. De plus, sa taille est donc taille_ini – taille et le bloc libre suivant est next_fb.
- Si on n'a pas la place de séparer le bloc en 2, on ajoute l'éventuelle reste de mémoire disponible.

Dans les deux cas, on traite le cas où fb est first_fb, dans ce cas on met à jour first_fb.
Enfin, avec la taille correcte, on définit le champ size du bloc alloué et on retourne un pointeur vers le début de la mémoire utilisable (donc la mémoire juste après l'entête new_ab) en ajoutant à new_ab la taille de la structure ab.

mem_free (size_t taille) :

A cause d'un problème d'organisation, mem_free n'a pas pu être faite dans les temps.

mem_fit_first (struct fb* list, size_t size) :

Cette fonction détermine le premier bloc libre de taille supérieur à l'argument taille. Pour cela, on parcours la liste list et on compare à chaque fois le champ size de l'objet courant avec taille. Si il est suffisant (supérieur ou égal), on retourne le pointeur vers l'entête du bloc en question, sinon on passe au bloc suivant via le champ next.

mem_get_size (void* zone) :

Cette fonction doit retourner la taille d'une zone que l'utilisateur passe en paramètre. La particularité ici est de prendre en compte que l'adresse donnée n'est pas l'adresse de l'entête de la zone mais bien l'adresse du début de la mémoire utilisable. Comme cette zone est une zone alloué, il suffit de prendre l'adresse donnée et d'enlever la taille d'une struct ab pour retomber sur l'adresse de l'entête. On accède alors facilement au champ size que l'on retourne.

mem_fit_best (struct fb* list, size_t size) :

Cette fonction retourne la bloc libre de taille minimale mais toujours plus grande que size entré en paramètre. L'idée est de créer un pointeur de zone libre min que l'on initialise à nul. On crée également un entier init défini à 0 nous servant à déterminer si min a été initialisé ou non. Dans un premier temps, on cherche le premier bloc de mémoire de taille suffisante. Ce bloc sert d'initialisation à min. Dès lors, on fait passer la valeur de init à 1, signifiant que min a été correctement initialisé. Alors, on parcours les blocs libres restants en vérifiant si leur taille est suffisante et si leur taille est inférieur à celle de min, si oui, on met à jour min avec l'adresse du bloc en question. Sinon, on passe au bloc suivant. A la fin, on retourne min.

mem_fit_worst (struct fb* list, size_t size) :

Cette fonction retourne le bloc libre de taille maximale, pour peu qu'il soit de taille supérieur à size. Cette fonction est similaire à la précédente, cependant le pointeur mis à jour se

nomme max et n'est mis à jour que lorsque la taille du bloc libre courant est plus grande que celle de max.

Problèmes dans le code :

Un problème est récurrent dans le code. A savoir, un adressage complètement faux. Ainsi, la première allocation se situera à l'adresse 0 alors que la suivante, ne faisant que quelques octets par exemple, se retrouvera bien plus loin dans la mémoire.

La cause de ce problème semble être l'incrémentation d'adresse. En l'occurence :

struct fb* fb += sizeof(struct ab) ;

Nous souhaiterions que fb soit incrémenté de 8 octets (car sizeof(struct ab) = 8), pourtant fb se retrouve incrémentée de 8*16 octets. En cause, le fait que :

struct fb* fb += 1 ;

n'incrémente pas fb de 1, mais de sizeof(struct fb), soit 16 dans notre cas.

Après recherches, ce problème peut être réglé en castant le pointeur manipulés dans un type de taille 1 octets (à savoir void* par défaut ou char*). Cependant, cela entraîne d'autres problèmes que nous n'avons pas su résoudre. A savoir :

- *lvalue required as left operand of assignment*
- *segmentation fault*