TP4 : Allocation dynamique

Le but de ce TP est d'implémenter un gestionnaire de mémoire dynamique en suivant les principes vus en cours : *free-list*, recherche *first-fit*, etc.

1 Prise en main du code fourni

Exercice Téléchargez et décompressez l'archive correspondant à ce TP. Dans le répertoire obtenu, tapez `make` pour compiler le code puis `./main` pour l'exécuter. Vous devez obtenir approximativement l'affichage suivant :

```
heap_base = 0x101090000
  block at 0x101090000: header=0x00000050 size=80 flags=0 (free)
  block at 0x101090050: header=0x00000050 size=80 flags=0 (free)
  block at 0x1010900a0: header=0x00000050 size=80 flags=0 (free)
  block at 0x1010900f0: header=0x00000050 size=80 flags=0 (free)
  block at 0x101090140: header=0x00000050 size=80 flags=0 (free)
  block at 0x101090190: header=0x00000190 size=400 flags=0 (free)
heap_end = 0x101090320
```

Remarques

- Vous pouvez aussi prendre l'habitude de taper par exemple `make && ./main`, ce qui signifie «d'abord exécuter `make` puis, si ça s'est bien passé, exécuter `./main` ensuite».
- L'archive est composée de trois fichiers. L'allocateur proprement dit est implémenté dans `mem.c`, et son API utilisateur est décrite dans `mem.h`. Le fichier `main.c` représente une application.

Exercice Ouvrez le fichier `main.c` dans votre éditeur de texte préféré. Le rôle de ce programme est uniquement de simuler une application aux yeux du gestionnaire mémoire. Il ne fera donc rien d'intéressant à part de demander à allouer et à désallouer des blocs de mémoire. Durant le TP, vous aurez souvent à modifier ce programme pour provoquer, depuis l'extérieur, différents comportements de votre allocateur. Commencez donc par décommenter la fin du programme, puis recompilez et exécutez de nouveau.
Constatez que l'un des blocs est maintenant indiqué comme occupé.

Exercice Ouvrez et lisez le fichier `mem.h`, qui décrit l'interface entre l'application et le gestionnaire mémoire. Posez des questions sur ce que vous ne comprenez pas. Cette interface est constituée de plusieurs fonctions :

- `mem_init()` permet d'initialiser le gestionnaire mémoire. L'application doit appeler cette fonction une seule fois, au début de l'exécution.
- `mem_alloc(int64_t length)` permet de demander l'allocation d'une zone de mémoire. Cette fonction renvoie l'adresse de début de la région, ou `NULL` en cas d'erreur.
- `mem_release(void *ptr)` permet de libérer une zone de mémoire. Le pointeur doit correspondre à une zone préalablement allouée avec `mem_alloc()`.
- `mem_show_heap()` est là juste pour vous aider à débugger. Elle affiche le contenu du tas, en détaillant les informations disponibles sur chaque bloc.

Remarques

- Vous ne devez pas modifier le fichier `mem.h`.
- Toutes les tailles que vous manipulerez dans ce TP seront des variables de type `int64_t`. N'utilisez pas de variables `int` sous peine d'avoir des bugs difficiles à diagnostiquer.

Exercice Ouvrez le fichier `mem.c` qui contient l'implémentation de notre gestionnaire mémoire :

— la fonction d'initialisation `mem_init()` est déjà écrite. Elle commence par un appel `mmap()` pour demander de la mémoire au noyau. Dans ce TP, le tas fait 800 octets en tout. Ensuite (seulement pour des raisons pédagogiques) elle crée des blocs libres dans ce tas : cinq blocs de 80 octets, suivis d'un bloc de 400 octets. Le résultat est illustré ci-dessous, figure 1(a).

— la fonction `mem_release()` est vide : pour l'instant, notre gestionnaire mémoire ne sait pas désallouer. On règlera ce problème dans la partie 2 du TP, page suivante. En attendant, remarquez que «*ne rien faire*» est une implémentation valide, qui est certes insatisfaisante, mais qui respecte le contrat de notre interface.

— la fonction `mem_alloc()` est déjà implémentée, dans une version simple : elle parcourt le tas de bloc en bloc jusqu'à en trouver un qui soit à la fois libre et assez grand (autrement dit : allocation en *first-fit* sans découpage). Lisez le code, exécutez-le en rajoutant des `printf` pour visualiser ce qui se passe, et posez des questions sur ce que vous ne comprenez pas. Remarque : le format des blocs en mémoire est illustré ci-dessous, figure 1(b).

— la fonction `mem_show_heap()` est déjà implémentée également. Elle est très similaire à la précédente : on parcourt le tas de bloc en bloc, en affichant les méta-données de chaque bloc. Assurez-vous de comprendre comment sont manipulées ces méta-données :
  — `x & y` se lit «ET bit-à-bit entre x et y», par ex : `0b0110 & 0b1010 == 0b0010 == 2`
  — `x | y` se lit «OU bit-à-bit entre x et y», par ex : `0b0110 | 0b1010 == 0b1110 == 14`
  — `~x` se lit «NON bit-à-bit de x», par exemple sur 4 bits : `~0b0110 == 0b1001 == 9`

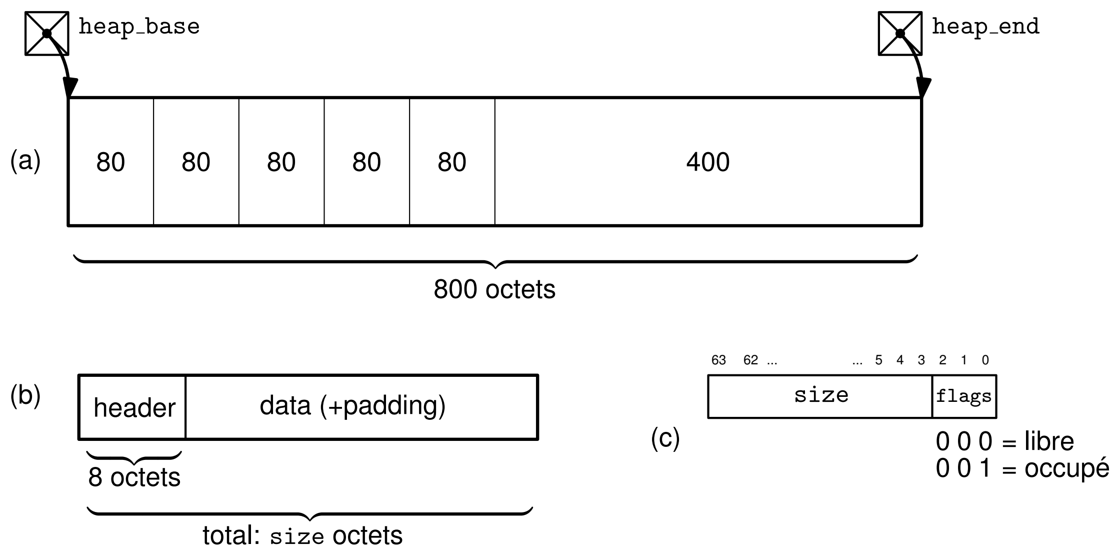

FIGURE 1 – (a) Contenu du tas après `mem_init()`. Attention, le pointeur `heap_end` indique la première adresse *en dehors* du tas. (b) Structure des blocs du tas. Dans ce TP, tous les blocs ont une taille (`size`) multiple de 8 octets. (c) Format de l'en-tête des blocs. Comme le champ `size` est toujours multiple de 8, ses trois derniers bits sont toujours zéro. Du coup on peut «détourner» ces trois bits pour y mettre des méta-données (`flags`) : 000 pour dire «bloc libre» et 001 pour dire «bloc occupé».

Exercice Écrivez dans votre `main.c` différents cas de test permettant d'explorer les cas «limite» de l'algorithme d'allocation. Par exemple :

— Vérifiez que vous pouvez allouer successivement six blocs de petite taille, par exemple 42 octets.

— Vérifiez que vous ne pouvez pas allouer un septième bloc : il n'y a plus de blocs libres et l'allocateur doit renvoyer NULL.

— Vérifiez que les tailles sont traitées correctement : une allocation de 200 doit aller dans le bloc libre de 400. Et une seconde allocation de 200 doit échouer.

Utilisez des `assert()` pour vérifier si les allocations réussissent ou échouent comme prévu. Pour pouvoir tester successivement différents scénarios dans une même exécution, invoquez la fonction `mem_init()` à chaque fois, pour remettre le tas dans son état initial. [1] Accumulez tous ces cas de test au fur et à mesure du TP pour vérifier que vous n'introduisez pas des bugs nouveaux dans votre code.

2 Recyclage des blocs désalloués

Pour l'instant, notre allocateur est vraiment simpliste. En particulier, il ignore froidement les requêtes de désallocation, et est donc incapable de réutiliser des vieux blocs pour les requêtes futures.

Exercice Implémentez la fonction `mem_release()`.

— À ce stade, on ne s'intéresse pas encore au découpage ou à la fusion de blocs. Votre tâche consiste donc uniquement à marquer le bloc comme libre.

— Attention, le paramètre `ptr` que vous passe l'application pointe sur l'espace utile, c'est à dire la zone `data` dans la figure 1(b), et non pas sur le début du bloc. Il faut soustraire 8 pour retomber sur l'en-tête.

Exercice Rajoutez des cas de tests pour vérifier que votre allocateur est maintenant capable de recycler les blocs libérés. Par exemple :

— Vérifiez que vous pouvez maintenant faire six allocations de 42 octets, puis une désallocation, puis une septième allocations de 42 avec succès.

— Vérifiez que vous pouvez allouer 200, puis les désallouer, puis redemander 300 avec succès.

3 Découpage des blocs trop grands

Exercice Dans la fonction `mem_alloc()`, ajoutez le nécessaire pour découper le bloc choisi s'il s'avère plus grand que la taille demandée par l'utilisateur.

— Il s'agit de créer de toutes pièces un nouveau bloc libre pour représenter l'espace libre restant.

Exercice Ajoutez des cas de tests pour mettre en évidence les améliorations, et notamment répondre aux questions suivantes :

— Vérifiez que vous pouvez maintenant faire sept allocations successives de taille 42, puisque le dernier bloc est assez gros pour contenir plusieurs allocations.

— En allant plus loin : combien d'allocations successives de 42 octets peut-on faire en partant de l'état initial du tas ?

— Supposons que l'application demande 60 octets alors que le tas contient uniquement des blocs libres de taille 80 : quel découpage choisissez-vous de faire ?

— Malheureusement, l'un des cas de tests implémentés dans les exercices précédents ne réussit plus. Lequel ? Comment s'appelle le problème que vous êtes en train d'observer ?

1. Ce qui n'aurait pas de sens dans une vraie application, mais en TP on peut tricher un peu pour se simplifier la vie

4 Boundary tags

Dans la suite du TP, on va s'intéresser à fusionner des blocs libres lorsqu'il sont voisins en mémoire. Pour déterminer efficacement quels sont les voisins d'un bloc, on va utiliser la technique des *boundary tags* vue en cours.
La nouvelle structure des blocs est illustrée à la figure 2 ci-dessous. Remarque : la taille minimale pour un bloc libre est maintenant de 16 octets, puisqu'on veut avoir la place de stocker au moins les deux *boundary tags*.

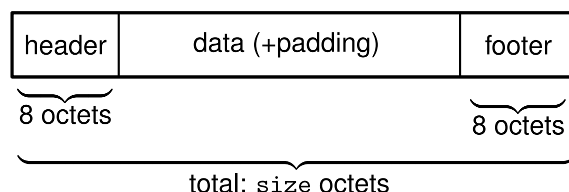

FIGURE 2 – Structure des blocs, version avec boundary tags. Remarque : le contenu du header et du footer sont identiques pour chaque bloc.

Exercice Implémentez les *boundary tags* dans votre allocateur :

- supprimez les blocs libres artificiellement créés dans `mem_init`, pour les remplacer par un seul gros bloc libre de 800 octets, avec *header* et *footer*.
- dans `mem_show_heap()`, on passe déjà par tous les blocs du tas un à un. Profitez-en pour afficher aussi le *footer*, et surtout vérifiez qu'il est bien égal au *header*, ça vous aidera pour détecter les bugs dans la suite.
- dans `mem_alloc()` et dans `mem_release()`, chaque fois que vous changez le *header* d'un bloc, vous devez maintenant changer aussi son *footer*.

5 Défragmentation du tas par fusion des blocs désalloués

Exercice Implémentez la fusion des blocs libérés :

- Pour simplifier, vous pouvez commencer par uniquement considérer le cas du bloc suivant.
- Dans le cas général, il y a quatre scénarios lorsqu'on désalloue un bloc B :
  - si les deux blocs voisins sont tous les deux occupés, alors suffit de marquer B comme libre
  - si le bloc suivant S est libre, mais pas le bloc précédent P, alors il faut fusionner B et S
  - si le bloc précédent P est libre, mais pas le bloc suivant S, alors il faut fusionner P et B
  - si les deux blocs voisins sont libres, alors il faut fusionner les trois blocs ensemble.

Remarque : faites des dessins représentant les quatre cas ci-dessus avant d'attaquer le code !