TP2 : Mémoire

Borne

1 Structure

Définition. On appelle *zone libre* un espace contigu d'adresses non utilisées.

Définition. On appelle *bloc alloué* un espace contigu d'adresses réservées.

Afin de pouvoir connaître à tout moment l'état de la mémoire, notre allocateur maintient une structure de liste doublement chaînée repérant le début des zones libres. L'ajout d'un lien vers la cellule précédente facilite le parcours de la liste et la fusion de cellules lors des opérations de libération.

On place au début de chaque zone libre une cellule de type `struct fb` définie comme suit :

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

En plus de la liste, au début de chaque bloc alloué, nous sauvegardons la taille de ce bloc. Enfin, nous réservons au tout début de la mémoire un emplacement "protégé" contenant un lien vers la première cellule de la liste.

Ces informations suffisent pour connaître à tout moment l'état de la mémoire et la parcourir.

2 Allocation

`mem_alloc` :

Spécification : `void* mem_alloc(size_t size)`

Sémantique : `mem_alloc(size)` alloue un bloc mémoire de taille t tel que :

$$t = max(\texttt{sizeof(struct fb)}, \texttt{size} + \texttt{sizeof(size\_t)})$$

Si aucune zone mémoire ne permet l'allocation, la fonction retourne NULL.

2.1 Taille minimale d'allocation

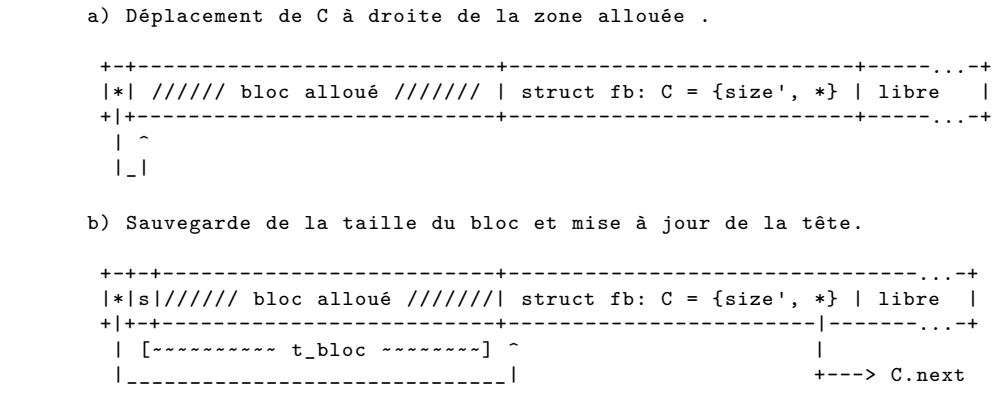
Lors d'une allocation, on ajoute au début du bloc mémoire alloué un élément de type `size_t` qui stocke la taille de la zone. Pour executer `mem_alloc(s)` il faut que `first_fit` cherche une zone libre de taille t avec

$$t >= s + \texttt{sizeof(size\_t)}$$

Supposons maintenant que la taille du bloc alloué b soit inférieure à `sizeof(struct fb)`. Lors d'une éventuelle libération de b par `mem_free(&b)`, on va se retrouver avec une zone libre que l'on va vouloir repérer par une nouvelle cellule de type `struct_fb`. Malheureusement il n'y aura pas assez de place dans la zone nouvellement libérée pour stocker une telle cellule. Pour remédier à ce problème on peut fixer la taille minimum d'un bloc alloué à `sizeof(struct fb)`. Ainsi, `mem_alloc(s)` alloue des blocs de taille t avec

$$t = \texttt{mem\_alloc}(max(s + \texttt{sizeof(size\_t)}, \texttt{sizeof(struct fb)}))$$

2.2 Illustrations



3 Libération

`mem_free`:

Spécification : `void mem_free(void* p)`

Pré-condition : `p` est l'adresse d'un bloc alloué retournée par `mem_alloc`.

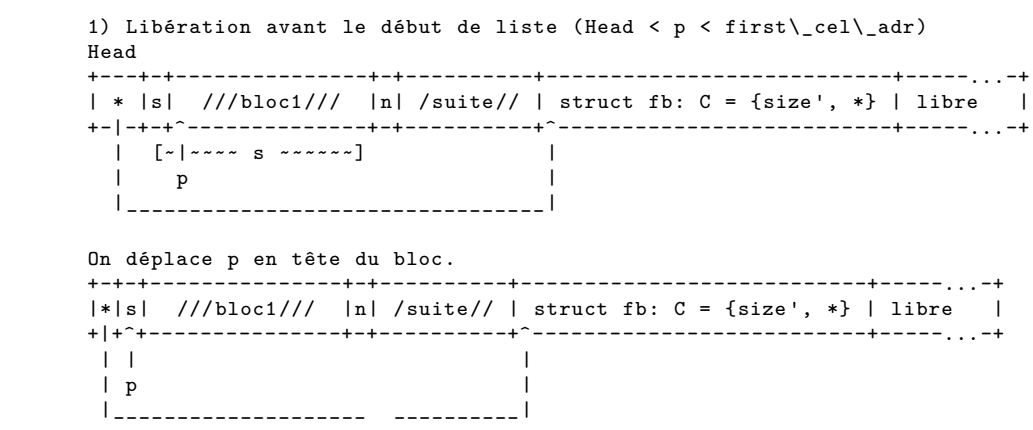
Sémantique : Libère le bloc alloué dont l'adresse `p` est fournie en paramètre. Crée une nouvelle cellule pour la zone libérée. Met à jour la liste des zones libres et Fusionne deux cellules de liste de zones libres si elles sont adjacentes.

La fusion de cellules est effectuée "en-place" après libération. On ne parcourt pas toute la liste des zones libres.

3.1 Illustrations

Libération

La méthode de libération d'un bloc varie selon que le bloc se trouve avant le début de la liste de zones vides ou non.



```
On place à l'adresse p une nouvelle cellule de liste de zone vide et on met
    à jour la liste.

+-+----------------+-+----------+--------------------------+-----...-+
|*| struct A {s,*} |n| /suite// | struct fb: C = {s_c  , *} | libre   |
+|+^---------------|--+-+----------+^-------------------------+-----...-+
 |_|            |_________________|

 *head <- A;
 A.size <- s -STRUCT_FB_SIZE; A.next <- C ; A.prev = head;
 C.prev <- A;
```

Si deux cellules A et C sont adjacentes après libération on les fusionne (voir section Fusion). Dans le cas de la libération avant le début de la liste seul la fusion en avant est à considérer. Le cas où le bloc à libérer se trouve après la première cellule de la liste de zones vides est similaire au cas 1 mais avec fusions de cellules avant et arrière à considérer.

Fusion

```
Fusion avant: Le bloc libéré est suivi par une cellule de zone libre.
  Exemple de fusion dans le cas 1):
  +-+--------------+-+-----------------+--------------------------+----..-+
  |*| // alloué // |s|   ///bloc1///    | struct fb: C = {s_c  , *} | libre |
  +-+---------------+-+^----------------+--------------------------+----..-+
                      |
                      p


  Après libération du bloc 1 on se retrouve dans la configuration suivante:

  +-+--------------+---------------------+-------------------------+----..-+
  |*| // alloué // | struct fb: A = {s,*} | struct fb: C = {s_c, *} | libre |
  +|+--------------+^-------------------|--+^----------------------|--+----..-+
   |_________________|                  |___|                      +--------->

   *head <- A;
   A.size <- s -STRUCT_FB_SIZE; A.next <- C ; A.prev = head;
   C.prev <- A;


  On fusionne les cellules A et C et on met à jour la liste;

  +-+--------------+---------------------+-----------------------------..-+
  |*| // alloué // | struct fb: A = {s,*} |            libre               |
  +|+--------------+^-------------------|--+-----------------------------..-+
   |_________________|                  +------------> Next

   *head <- A;
   A.size <- s + s_c + STRUCT_FB_SIZE
   A.next <- C.next
   Fusion arrière:
   Si le bloc libéré est précédé d'une zone libre on fusionne la Cellule
       précédente
   avec la cellule A nouvellement crée. Similaire au cas de fusion avant.
```

4 Tests

Dans nos test nous cherchons à remplir et vider la mémoire et retomber sur l'état initial. Chaque test affiche l'état de la mémoire après allocation de tous les blocs et à chaque libération.

Nous testons les configurations suivantes :

- Allocations de taille fixe suivies de libérations dans l'ordre inverse.
- Allocations de taille fixe suivies de libérations dans un ordre aléatoire.
- Allocations de tailles aléatoires suivies de libérations dans un ordre aléatoire.
- Enchevêtrement d'allocations aléatoires et libérations aléatoires.

5 Pistes d'améliorations

`mem_free` nécessite une factorisation de code. Certain invariants de la structure de liste ont étés exploités mais pas prouvés. Les programmes de tests fournis utilisent (honteusement) des sections de code partagées d'un fichier à un autre. Notre style de programmation peut être largement amélioré, il s'agit d'une première expérience avec le langage C. A titre d'exercice nous prévoyons de continuer de nettoyer et maintenir le code et les tests produits.

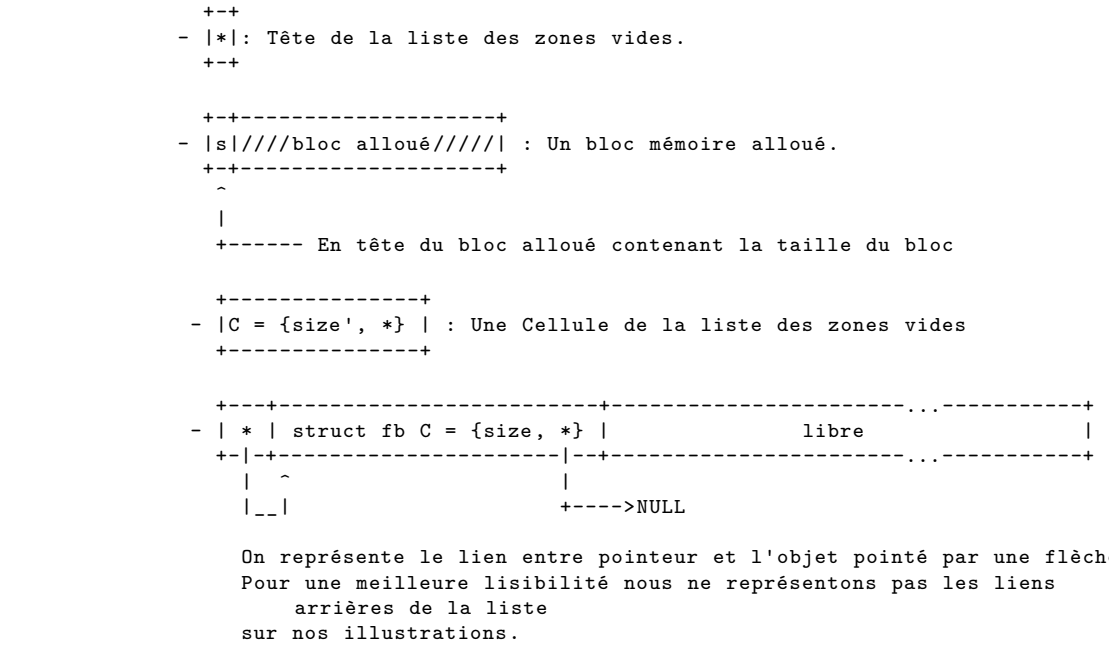
6 Conclusions

Ce travail nous a permis de nous familiariser avec le langage C et la manipulation de pointeurs. Étant donné la nature "bas niveau" des objets manipulés (pointeurs) il nous a semblé pertinent d'enrichir le code d'illustrations et de commentaires. Nous avons réalisé un allocateur mémoire simpliste (pas de multithread, firstfit, ...) mais fonctionnel. Nous avons constaté que le problème d'allocation est un problème bien documenté d'optimisation combinatoire proche du problème de sac de voyage et d'empaquetage en-ligne. Nous avons employé une heuristique firstfit naïve mais nous souhaiterions comparer les performances de différentes heuristiques (rapidité, fragmentation, ...). Nous avons apprécié travailler sur cette introduction à la gestion mémoire et allons continuer à parcourir la littérature sur le sujet.

7 Annexe

7.1 Charte ASCII

```
- (*): Un pointeur
```



On représente le lien entre pointeur et l'objet pointé par une flèche.
Pour une meilleure lisibilité nous ne représentons pas les liens arrières de la liste
sur nos illustrations.