

# TD n°8

## Gestion Mémoire

### 1 Objectif

Le but de cet exercice est de réaliser un allocateur dynamique de mémoire, c'est-à-dire un substitut aux fonctions `malloc` et `free` de la bibliothèque C.

Les allocateurs de mémoire généraux sont parmi les programmes système les plus délicats à réaliser et à tester, mais aussi ceux qui peuvent avoir une influence considérable sur les performances en temps et en mémoire. Nous ne prétendons pas réaliser ici un allocateur très sophistiqué, seulement donner une idée des problèmes.

Cet exercice est aussi l'occasion de manipuler à un niveau fin les pointeurs de C, en mettant vraiment les mains dans le cambouis, comme on a souvent à le faire en programmation système. L'exercice n'est pas facile, même si le code est court. Lisez bien les spécifications et les remarques qui suivent. Certains choix de conception ne deviendront clairs qu'après avoir complètement codé une solution.

Comme d'habitude, pour vous faire gagner du temps pour la mise en route, vous devrez récupérer l'archive sur la page du cours.

**Commencez par lire entièrement le sujet avant de vous lancer à programmer quoi que ce soit ! Vous avez 8 pages à lire et à comprendre avant de vous lancer. Posez des questions pour être sûr d'avoir compris l'intégralité avant de coder !**

### 2 Des fonctions de base : `malloc`, `free` ... et `sbrk`

#### 2.1 Les fonctions `malloc` et `free` d'ANSI C

En C, la fonction `malloc` permet au programmeur d'allouer dynamiquement de la mémoire, et la fonction `free` lui permet de rendre cette mémoire afin de la recycler pour l'utiliser dans un éventuel `malloc` suivant.

Voici un exemple d'utilisation :

```
struct Data { // Une structure de données
    char nom[100];
    int age;
};
...
// On alloue dynamiquement un objet de ce type
// La fonction malloc retourne un pointeur sur la zone allouée
struct Data *pdata = malloc(sizeof(struct Data));
// On peut maintenant utiliser librement cet objet
pdata->age = 12;
strcpy(pdata->nom, "Peter Pan");
...
// Et quand on n'en a plus besoin le libérer
free(pdata);
// Attention : ici le pointeur pdata n'est plus valide !
```

**Question 1:** Quelle sera a priori la taille en octets demandée par l'appel à la fonction `malloc` ?

La plupart des systèmes d'exploitation ne réalisent pas de manière primitive cette gestion du recyclage. Les fonctions `malloc` et `free` ne sont donc pas, en général, des appels systèmes mais des fonctions de bibliothèque (de la bibliothèque C en l'occurrence).

On peut d'ailleurs se demander pourquoi des fonctions aussi importantes ne sont pas directement réalisées par le noyau. La raison en est très simple : il est très difficile d'écrire un allocateur général de mémoire dynamique, qui doit

## TD n°8

# Gestion Mémoire

être à l'aise aussi bien pour allouer un grand nombre de petits objets qu'un grand nombre de très grands ou encore un mélange des deux. Des compromis de conception sont indispensables et les mauvais choix peuvent entraîner des pertes de performances parfois considérables. Donc il est préférable de ne pas figer les algorithmes de gestion mémoire dans le noyau. En les réalisant sous forme de fonctions de bibliothèque, on peut les changer et les remplacer facilement pour, par exemple, les adapter à un schéma d'utilisation mémoire particulier, pour lequel on peut imaginer des algorithmes plus efficaces que les compromis généraux.

### 2.2 La fonction UNIX `sbrk`

Si le système d'exploitation ne réalise pas lui-même la gestion du recyclage, il doit collaborer un peu pour permettre la réalisation de la fonction `malloc`. Le minimum qu'il ait à faire est de permettre d'augmenter l'espace mémoire d'un programme. Sous UNIX (et donc GNU/Linux), ceci est réalisable par l'appel-système `sbrk`<sup>1</sup> (`man sbrk`, donc...).

Cette primitive s'utilise très simplement : il suffit de faire :

```
void *pnew = sbrk(incr);
```

où `incr` est un entier non signé, pour que le segment de données du programme s'accroisse de (au moins) `incr` octets. La valeur de retour est un pointeur sur le début de la zone supplémentaire ainsi allouée. Notez bien que cette zone n'a absolument aucune structure ; elle est constituée d'octets à la suite les uns des autres, et c'est aux fonctions `malloc` et `free` qu'il appartiendra de la structurer.

Si le système ne peut plus allouer de mémoire supplémentaire, `sbrk` retourne `-1`, ce qui n'est pas une très bonne idée car `-1` n'est pas une valeur de pointeur (!) et cela rend le test un peu pénible :

```
if (pnew == ((void *) -1))
    fprintf(stderr, "Plus de memoire\n");
```

## 3 D'une solution naïve à une solution un peu plus optimisée

Nous allons donc réfléchir à la mise en place d'un algorithme de gestion de la mémoire lors des appels à `malloc` et `free`.

D'un point de vue du programme utilisateur, on fait donc des appels successifs à `malloc` et à `free` pour respectivement récupérer des zones mémoires ou les libérer. Vous noterez que d'un point de vue système, on ne connaît pas à l'avance le nombre de `malloc` et de `free` que fera un programme. La structure de données utilisée pour écrire le gestionnaire de mémoire doit donc répondre aux critères suivants :

- Permettre de désigner dans la(es) zone(s) mémoire(s) allouée(s) par `sbrk` celle(s) qui est(sont) utilisée(s) (i.e zones utiles au programme qui a fait `malloc`) ou qui est(sont) libre(s) (zones restées libres après un appel à `sbrk` suite à un `malloc` qui n'utilise pas toute la zone allouée, ou zones libérées par un appel à `free`) ;
- Permettre de connaître la taille des zones, qu'elles soient libres ou occupées ;
- Et bien sûr, être une structure de données dynamique, donc pas un tableau qui a une taille prédéfinie, mais une liste chaînée par exemple.

### 3.1 Une première solution naïve : données et métadonnées séparées

La première solution naïve consiste à penser que l'on va gérer de manière séparée :

- Les **données**, c'est-à-dire les zones mémoire libres ou occupées qui ont été attribuées avec les appels à `malloc/free` dans un programme.

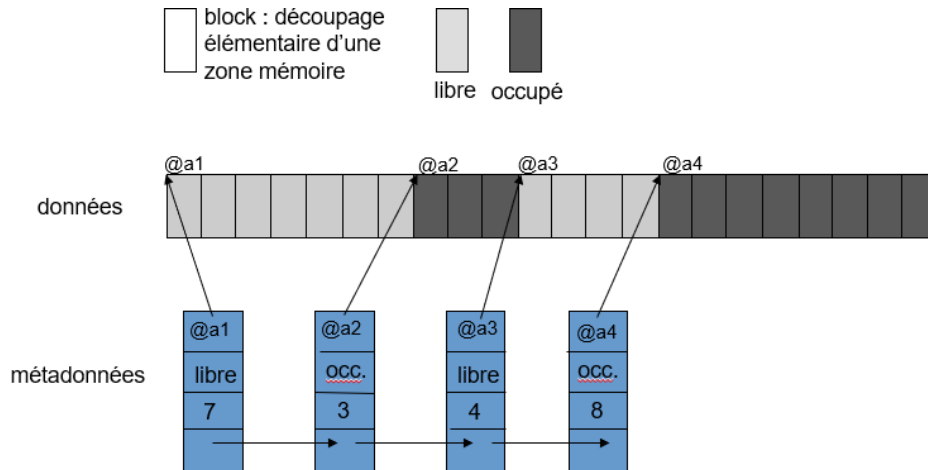
<sup>1</sup> Les fonctions `malloc` et `free` font partie de la norme ANSI C et donc de POSIX. Ce n'est pas le cas de `sbrk` qui est spécifique à UNIX : d'autres systèmes d'exploitation peuvent proposer un mécanisme fondamentalement différent pour obtenir de la mémoire du système.

## TD n°8

### Gestion Mémoire

- Les **métadonnées**, c'est-à-dire les informations sur les données : adresse (@) de la zone, si elle est libre ou occupée, et bien sûr la taille de la zone.

Voici un petit schéma pour tenter de comprendre la situation.



Les inconvénients de cette solution sont qu'elle n'est pas optimale d'une part et présente un paradoxe d'autre part.

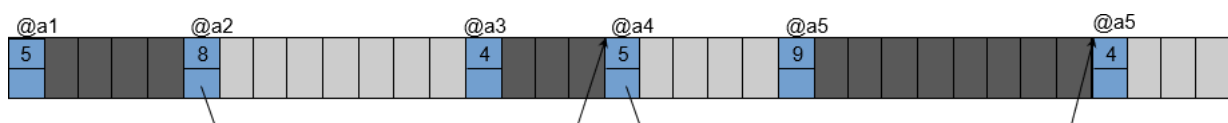
1. Cette solution est non optimale car on gère les zones libres et les zones occupées alors qu'il suffit de gérer les zones libres. En effet, la zone occupée est utile au programme qui a fait `malloc`, et c'est la responsabilité du programme de bien gérer la libération de la zone allouée. Donc pour le gestionnaire de mémoire, l'information indispensable est l'adresse de la zone occupée qu'il faut libérer avec `free`. Mais ceci est donné en paramètre à `free`, donc inutile de connaître toutes les adresses des zones occupées, elles seront fournies par le programme utilisateur quand il fait un appel à `free`.
2. Cette solution présente un paradoxe car les métadonnées sont dans une liste chaînée et il faut faire un `malloc` pour allouer de la mémoire pour cette liste. C'est en **CONTRADICTION** avec le fait qu'on est en train d'écrire `malloc` !

### 3.2 Une solution optimisée

Une solution possible est donc la suivante :

- Les données et les métadonnées sont dans la même zone mémoire (ce qui permet de résoudre le paradoxe précédent où l'on a besoin de la `malloc` pour créer la liste de gestion des métadonnées de `malloc`).
- Une métadonnée fait une taille de 1 bloc, on appelle ce bloc spécifique le « header » (ou entête de zone).
- Il suffit de chaîner les zones libres : le header contient alors l'adresse de la prochaine zone libre.
- Le header contient aussi la taille de la zone (libre ou occupée) que l'on compte en nombre de blocs, en tenant compte du bloc « header » lui-même. Regardez sur l'exemple les zones et leurs tailles.

Le schéma suivant permet de résumer la situation.

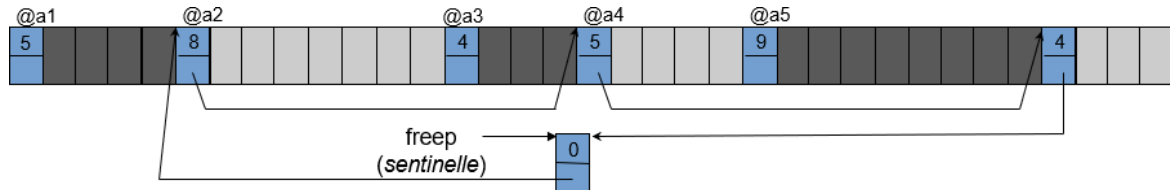


Si on va un peu plus loin dans les explications et la gestion de la liste :

- La zone mémoire est découpée en blocs qui font la taille d'un header.

## TD n°8 Gestion Mémoire

- Le header contient le nombre de blocs de la zone (y compris lui-même) et le pointeur vers le header de la prochaine zone **libre**).
- Les adresses mémoires sont par ordre croissant.
- On utilise une liste chaînée circulaire qui utilise une sentinelle ; ainsi la liste chaînée commence sur la sentinelle et reboucle sur la sentinelle que l'on appelle *freep* (pour pointeur « p ») sur la zone « free »).



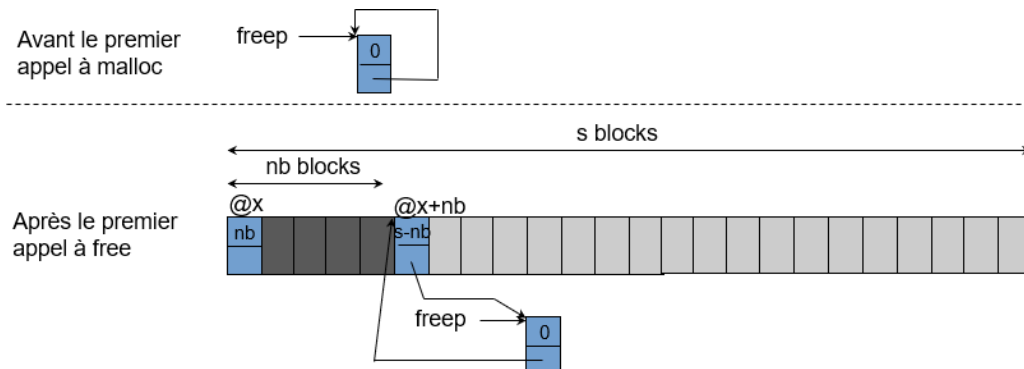
Le **parcours type** pour insérer ou supprimer un élément de cette structure de données (liste circulaire) est donné par l'algorithme suivant :

```
prevp = freep
P = freep
tant que l'on n'est pas à l'endroit voulu
    prevp = p
    p = next(p)
// après cette boucle, l'élément qui nous intéresse est entre prevp et p
// si p = freep, on est sur la sentinelle => on a fait un tour de liste, traitement
spécifique éventuel
```

### 3.3 malloc : cas initial

Au démarrage, on dispose de la sentinelle qui pointe sur elle-même (cas d'une liste circulaire vide). On va faire un premier appel à *malloc*, la liste étant vide.

- On doit tout d'abord convertir la quantité demandée en octets (paramètre de *malloc*) en nombre de blocs de notre structure de données, en prenant en compte le header qui contient les métadonnées.
- On appelle *sbrk* pour allouer une nouvelle zone de mémoire. Mais comme *sbrk* coûte cher en temps d'exécution (beaucoup de choses sont à faire par le système d'exploitation), on va allouer au minimum *s* blocs, même si on ne les utilise pas tous, pour ce *malloc*.
- Renvoie *@x+1*, le pointeur de début de zone utilisable (+1 pour passer le header, le programme ayant fait le *malloc* ne doit pas écraser ce header qui sert à la gestion de la structure de données).



### 3.4 malloc : cas générique

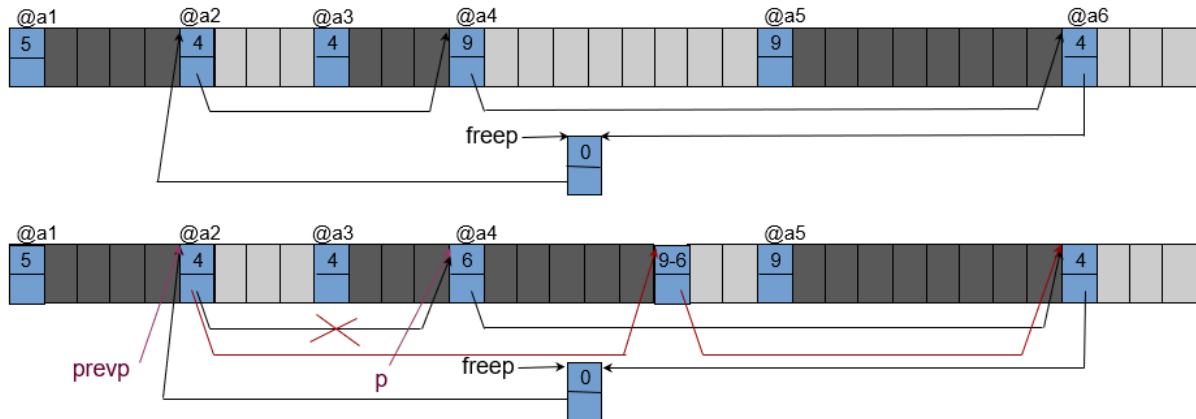
Voici les étapes pour le cas générique d'un *malloc* avec mise à jour de la liste des zones libres et des métadonnées :

## TD n°8

### Gestion Mémoire

- On avance dans la liste avec `prevp` et `p` jusqu'à trouver une zone telle que la taille de `p` est supérieur ou égale au nombre `nb` de blocs nécessaires pour satisfaire l'appel à `malloc`.
- On met alors à jour `next(prevp)`, `next(p)` et `size(p)`.
- Enfin, on renvoie `p+1` : le +1 est là pour se positionner juste après le header, le programme qui fait `malloc` ne doit pas écraser le header.

**Exemple : malloc qui nécessite 6 blocks**



Quand vous coderez votre `malloc`, vous veillerez bien sûr à traiter les deux cas (cas initial et générique) le plus proprement possible (en évitant les répétitions de code).

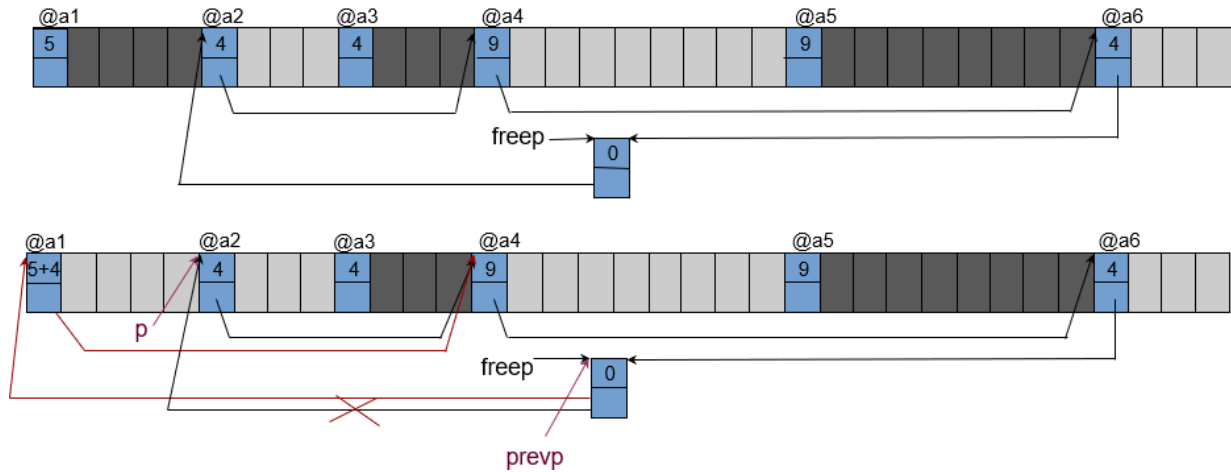
### 3.5 free : libérer une zone mémoire

Le programme utilisateur va faire appel à `free` avec une adresse `@x`. Le code de la fonction `free` devra alors :

- Considérer la mémoire à l'adresse `fp=@x-1` pour être sur le header de la zone qui a été allouée par le `malloc`.
- Quatre cas sont alors possibles (nota : ces 4 cas peuvent s'écrire simplement en regardant la zone adjacente à gauche puis à droite) ; soit la zone qu'on libère est adjacente :
  - à une zone libre à gauche,
  - à une zone libre à droite,
  - à une zone libre à gauche et à droite,
  - à une zone occupée à gauche et à droite.
- Principe :
  1. Se déplacer avec `prevp` et `p` en partant de `freep` jusqu'à ce que `p` soit plus grand que `fp`
  2. Selon le cas (coller à droite ou à gauche à, mettre à jour `size(fp)`, `next(fp)` ou `size(prevp)`, `next(prevp)` en utilisant les informations de `p`, `prevp` ou `fp`.

## TD n°8 Gestion Mémoire

Exemple : `free(@a1+1)`

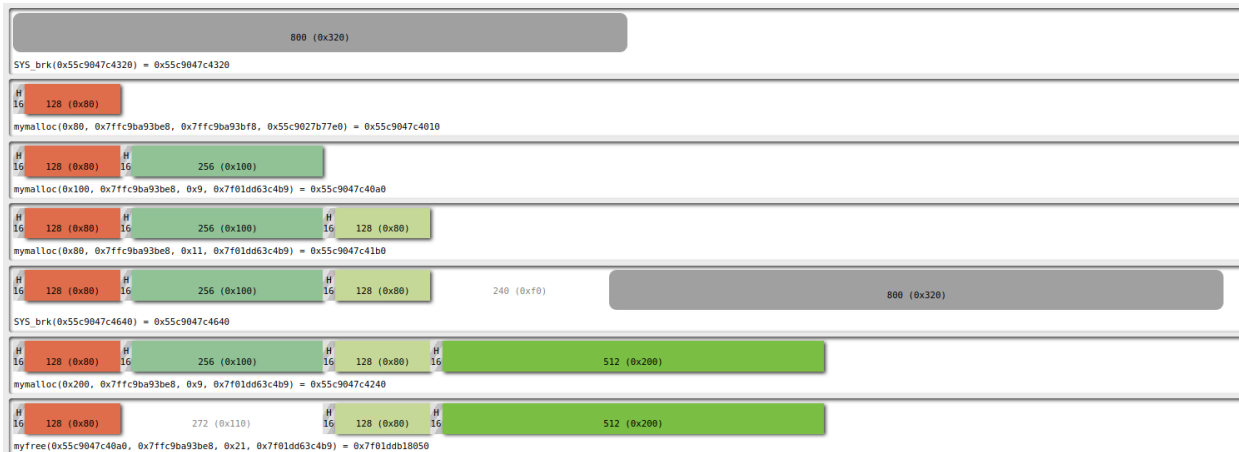


### 4 Expérimentations et visualisation

Même si nous avons bien détaillé l'ensemble des cas et des algorithmes de `malloc` et `free`, il n'en reste pas moins que vous allez devoir les coder... et les déboguer ! Alors encore un peu d'aide.

#### 4.1 Visualisation de l'état du tas (heap)

Il est toujours difficile de se représenter l'état de la mémoire et en particulier de la mémoire allouée dynamiquement. Afin de vous faciliter la tâche, nous vous proposons d'utiliser un petit script Python<sup>2</sup> permettant de générer un historique de l'état du tas (*heap* en anglais), après l'appel de chaque instruction allouant ou récupérant de la mémoire).



Ce script a été modifié pour ajouter la possibilité de visualiser les opérations de type `sbrk` et de prendre en compte vos fonctions `mymalloc` et `myfree` (plus quelques autres petites améliorations).

Pour générer ce type de graphique, il faudra récupérer les différents appels aux fonctions `mymalloc` et `myfree` ainsi que l'appel système `sbrk`. Pour simplifier au maximum l'instrumentation du code, nous vous fournissons des fonctions qui généreront les informations nécessaires à la réalisation de ces traces (ce que pourrait générer `ltrace` que nous avons vu lors du premier TD).

<sup>2</sup> <https://github.com/wapiflapi/villoc>

## TD n°8

# Gestion Mémoire

La trace d'exécution de votre programme de test sera envoyée en tant qu'entrée du script Python générant une page html avec la frise chronologique de l'état du tas. Pour simplifier cet appel, nous avons mis en place un petit script Shell qu'il suffit d'appeler de la manière suivante :

```
./villoc/to-html test-simple.exe
```

Celle aura pour résultat de générer un fichier `test-simple.html` pour visualiser le résultat de l'exécution de votre programme.

Pour vous faciliter cette visualisation, nous avons intégré la génération des fichiers html après avoir compilé les programmes de test (tout fichier `.c` qui est dans le dossier du `Makefile`). Donc il vous suffit de faire la commande suivante pour générer les exécutables et les fichiers html pour visualiser le résultat de votre implémentation de `malloc` et de `free` :

```
make visu
```

## 4.2 Visualisation et étude d'un cas simple

Pour vous aider à comprendre ce qui se passe en mémoire, nous allons commencer par utiliser un petit programme de test tout simple (`test-simple`) qui fait quelques `malloc` et `free` avec l'implémentation standard de la bibliothèque C et avec l'implémentation de l'algorithme décrit ci-dessus. Les fichiers html ont déjà été générés et vous sont fournis dans le dossier HTML (respectivement `malloc-test-simple.html` et `mymalloc-test-simple.html`).

### 4.2.1 Commençons par regarder le fichier `mymalloc-test-simple.html`

On voit sur ce graphique que lors du premier appel à `malloc`, on a un appel à `sbrk` qui étend la zone mémoire du processus pour lui permettre de faire de l'allocation dynamique. On alloue successivement 3 fois à l'aide de `malloc`. A la quatrième fois, la zone mémoire rendue par `sbrk` n'est plus assez grande, donc l'appel à `malloc` relance un appel à `sbrk` pour réétendre à nouveau la zone et ainsi disposer d'espace suffisant pour allouer la zone souhaitée par le 4<sup>ème</sup> `malloc`. L'appel à `free` libérera la zone mémoire correspondante.

La suite de la trace montre que l'on fait une allocation d'une zone avec un appel à `malloc` qui demande plus d'espace que ce qui est disponible dans la première zone libre. (`malloc(384)`). Les deux appels de `malloc` suivants vont utiliser la première zone mémoire disponible qui suffisamment grande.

**Question 2:** Si je fais un appel à `malloc` avec la valeur 80, quelle zone mémoire sera utilisée ? Cela déclenchera-t-il un nouvel appel à `sbrk` ?

**Question 3:** Quelle est la plus grande valeur que je peux demander à `malloc` pour utiliser le premier trou dans la zone mémoire ?

### 4.2.2 Puis, regardez le fichier `malloc-test-simple.html`

Vous pourrez constater que vous n'obtenez pas exactement le même résultat. En particulier, vous voyez que le `sbrk` qui est fait est bien plus conséquent que celui que nous vous proposons de faire en TD (dans le cas du TD, nous pouvons vérifier rapidement quand nous n'avons plus assez de mémoire pour une nouvelle allocation). Vous constaterez aussi que la réutilisation après une libération n'est pas identique à l'algorithme que nous avons décrit dans ce sujet. En effet, l'algorithme consistant à prendre la première place suffisamment grande est un peu trivial et ne présente pas de bonnes propriétés par rapport à d'autres algorithmes.

Maintenant que vous avez compris le principe, nous allons pouvoir passer à la mise en œuvre de votre propre allocateur de mémoire.



## TD n°8

# Gestion Mémoire

### 5 Un allocateur dynamique simple

Nous espérons que ces quelques explications et manipulations vous auront permis de mieux comprendre le fonctionnement que vous devrez mettre en place. Mais avant de vous lancer à coder une solution, voici quelques explications sur la structure de données à mettre en place et comment cela fonctionne (elle vous est fournie dans le code source `mymalloc.c`).

#### 5.1 Spécification de l'interface

Bien que nous ayons annoncé que c'était très difficile, nous allons réaliser une version simple de `malloc` et `free`. Évidemment notre version ne sera pas aussi évoluée ni aussi efficace que celles que l'on trouve dans les systèmes modernes. Mais elle sera complète et permettra de mettre en évidence les difficultés de la tâche. Pour exemple, dans la version 2.31 de la `glibc` de février 2020, le code source de `malloc.c` implémentant les fonction `malloc`, `free`, ..., le code source fait 179Ko, avec beaucoup de commentaires certes, mais tout de même).

Pour ne pas les confondre avec les versions standard, nous nommerons nos fonctions `mymalloc` et `myfree`. Leurs prototypes seront analogues à ceux du standard :

```
void *mymalloc(size_t size);
void myfree(void *p);
```

La fonction `mymalloc` retourne un pointeur sur une zone assez grande pour contenir un objet de taille `size` caractères (`size` est un entier long non signé). En cas d'échec, `mymalloc` retourne le pointeur `NULL`.

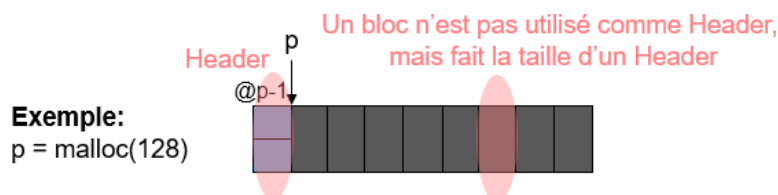
Quant à `myfree`, elle libère la zone pointée par `p` afin qu'elle soit réutilisable par un futur `mymalloc` dans le même programme ; après cet appel `p` est invalide (mais pas `NULL` ! en fait sa valeur n'est pas modifiée). Bien entendu, pour pouvoir appeler `myfree`, `p` doit avoir une valeur qui est le résultat d'un précédent `mymalloc`.

#### 5.2 Mise en œuvre

Comme nous l'avons vu précédemment, un élément important est la liste circulaire permettant de gérer la structure de données des zones libres. Nous présentons ici la définition d'une zone et son utilisation pour la liste chaînée.

##### 5.2.1 Entête de zone

Afin de gérer les zones libres, les fonctions `malloc` et `free` ont besoin, pour chaque zone, d'un pointeur de chaînage et d'un entier indiquant la taille utile de la zone. Par conséquent, chaque zone mémoire retournée à l'utilisateur sera précédée par un entête contenant ces informations :



Un entête de zone pourra être représenté par le type suivant :

```
typedef struct header { /* Header de zone */
    size_t size;         /* Taille du zone */
    struct header *ptr;   /* Zone libre suivante */
} Header;
```

La taille de cet entête sera notée `HEADER_SIZE` :

```
#define HEADER_SIZE sizeof(Header)
```



## TD n°8

# Gestion Mémoire

### 5.2.2 Problèmes d'alignement

Généralement, les processeurs imposent certaines contraintes sur les adresses auxquelles les données peuvent être stockées (par exemple, un double doit être rangé à une adresse multiple de 8). En C, la fonction `malloc`, doit s'occuper des problèmes d'alignement et rendre une adresse à laquelle on peut ranger des objets de type quelconque. Pour cela, on supposera que le type le plus contraignant est dénoté par la constante `MOST_RESTRICTING_TYPE` (sur un PC ce type pourra être défini à double). Par conséquent, le type `Header` sera redéfini de la façon suivante :

```
// Pour s'aligner sur des frontières multiples de la taille du type le plus
// contraignant
#define MOST_RESTRICTING_TYPE long double

typedef union header {
    struct {
        size_t size;
        union header *ptr;
    } info;
    MOST_RESTRICTING_TYPE dummy;
} Header;
```

A titre d'exemple, sur une machine 64 bits avec gcc 7.5, `size_t` est défini comme un `unsigned long` et fait 64 bits (8 octets), un pointeur fait 64 bits (8 octets), donc la structure `info` fait  $8+8 = 16$  octets. Le type `long double` qui fait 16 octets et on fait l'union de cette structure `info` avec le type `long double`. Dans le cas d'une union, la taille de l'union correspond à la taille du plus grand type stocké, donc dans notre cas 16 octets (les deux faisant la même taille de 16 octets). Donc `HEADER_SIZE` vaudra 16 octets dans notre exemple.

#### Exemple:

`p = malloc(128)`  
avec `sizeof(Header)`  
qui vaut 16 octets

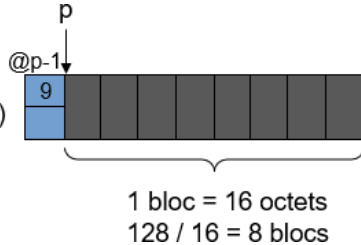


Illustration sur le modèle des exemples du TD

#### Exemple:

`p = malloc(128)`  
avec `sizeof(Header)`  
qui vaut 16 octets

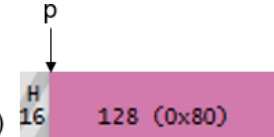


Illustration avec le fichier HTML généré par villoc

## 6 1, 2, 3... Maintenant vous pouvez coder !

Le travail demandé pour ce TD consiste donc à réécrire les fonctions `malloc` et `free` (puis dans un second temps `calloc` et `realloc`) de la bibliothèque C standard.

Dans le fichier `mymalloc.c` qui vous a été fourni, vous veillerez à ne pas modifier les fonctions externes (`mysbrk`, `mymalloc`, `myfree`, `myrealloc` et `mycalloc`) qui permettent de générer la trace attendue par le générateur de page HTML. Ces fonctions externes appellent les fonctions internes que vous allez réécrire : `allocate_core`, `internal_malloc`, et `internal_free`.

Pour écrire code code de `mymalloc`, vous aurez besoin de faire un appel à `allocate_core` et à `myfree`, or vous ne pourrez tester `myfree` qu'après avoir appelé `mymalloc`.

Vous constatez que dans le fichier `mymalloc.c` vous disposez déjà d'une implémentation pour chacune de ces fonctions, mais sous forme obfusquée<sup>3</sup>. Ne perdez surtout pas de temps à tenter de *désobfusquer* le code ! Celui-ci vous a été donnée afin d'assurer qu'à chaque instant vous disposez d'une solution fonctionnelle de ce qui est décrit dans

<sup>3</sup> [https://en.wikipedia.org/wiki/Obfuscation\\_\(software\)](https://en.wikipedia.org/wiki/Obfuscation_(software))

## TD n°8

# Gestion Mémoire

le sujet de TD. Sans cela, vous seriez obligé de faire le code des 3 fonctions avant de pouvoir tester. Avec cette méthode, vous pouvez refaire l'implémentation de chacune des fonctions les unes après les autres, en testant à chaque étape que votre code est correct. Pour cela, nous n'aurez qu'à commenter le code obfusqué et proposer votre implémentation.

### 6.1 Implémentation de `internal_malloc`

Commencer par fournir votre implémentation de `internal_malloc` qui suivra une stratégie « *first fit* » pour choisir la zone qui sera retenue dans la liste des zones libres. Cette stratégie consiste à choisir la première zone de taille suffisante dans la liste des zones libres de la couper en deux<sup>4</sup> et de laisser la partie inutilisée dans la liste des zones libres. Cette stratégie n'est pas optimale, puisqu'elle va morceler la mémoire, mais elle a l'avantage d'être simple à implémenter.

Vous vous référerez aux sections 3.2, 3.3, et 3.4, et en particulier aux illustrations et descriptions pour son implémentation.

Vous porterez une attention particulier au fait que votre `internal_malloc` doit travailler en taille multiple de la taille d'un `Header`. Or le paramètre qui lui est passé est le nombre d'octets (jusqu'à présent quand vous avez utilisé le `malloc` de la bibliothèque C, vous avez toujours exprimé le nombre d'octets à allouer et pas le nombre de blocs (taille d'un `Header`). Donc en début de fonction `malloc`, vous commencez par convertir le nombre d'octets en nombre de blocs.

Voici un algorithme en pseudo code simplifié pour vous faciliter l'implémentation de cette fonction :

```
je parcours la liste
  si je suis sur un zone de taille >= à la taille souhaitée
    si je suis sur un zone de taille = à la taille cherchée
      supprimer la zone de la liste
      retourner le pointeur comme résultat de la fonction
    sinon
      découper la zone et ajouter la zone restante libre à la liste
      retourner le pointeur comme résultat de la fonction

si j'ai réalisé un tour de liste /* donc sans trouver */
  faire un appel à allocate_core et ajouter ce quelle retourne à la liste
```

### 6.2 Implémentation de `allocate_core`

Vous allez maintenant fournir votre implémentation de `allocate_core`. Cette fonction est celle qui va faire appel à `sbrk` pour allouer une nouvelle grande zone mémoire (en fait étendre la zone mémoire adressable par le processus). Par définition, cette nouvelle zone mémoire allouée par `sbrk` est contigüe à la zone d'adressage du processus.

Le paramètre envoyé à la fonction est une taille en nombre de blocs. Mais pour être efficace, on allouera au moins `MIN_ALLOCATION` taille de `Header`, à convertir en octets car `sbrk` (et donc `mysbrk`), attendent un paramètre en nombre d'octets.

Pour vous aider, voici un pseudo code de ce que doit faire `allocate_core` :

```
s'assurer que l'on a un nombre de blocs minimum et le convertir en octets
allouer une nouvelle zone à la liste /* appel à sbrk */
ajouter la zone à la liste /* via un appel à free */
retourner le pointeur sur le début de la liste /* donc freep */
```

<sup>4</sup> sauf s'il fait juste la bonne taille bien sûr.

## TD n°8

# Gestion Mémoire

---

L'ajout de la zone à la liste en appelant `free` (donc `internal_free` dans notre cas particulier) peut vous paraître étrange. Mais quand vous implémenterez cette fonction, nous comprendrez l'intérêt.

### 6.3 Implémentation de `internal_free`

Pour faire l'implémentation de `internal_free`, vous vous reporterez aux explications données à la section 3.5.

Voici un algorithme en pseudo code simplifié pour vous faciliter l'implémentation de cette fonction :

```
parcourir la liste jusqu'à « la bonne place »  
fusionner avec la zone suivante si nécessaire sinon mettre à jour la liste  
fusionner avec la zone précédente si nécessaire sinon mettre à jour la liste
```

Et voilà, à ce stade vous devez disposer de votre propre implémentation de `malloc` et de `free`!

## TD n°8 Gestion Mémoire

### Pour aller plus loin

#### 7 Implémentation de `mycalloc` et `myrealloc`

Quant aux fonctions `mycalloc` et `myrealloc`, elles sont simples à écrire et s'expriment en fonction de `internal_malloc` et `internal_free`. C'est une extension possible et simple de votre travail pour le compléter (lors des révisions par exemple).

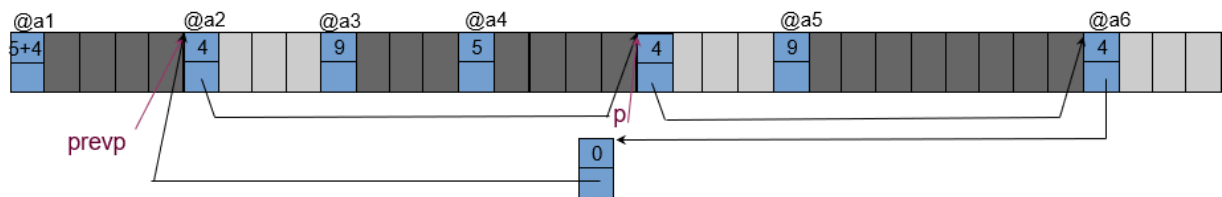
Vous avez un fichier de test particulier pour l'implémentation de `myrealloc` : `test-realloc.c`.

Dans l'algorithme à mettre en place pour `myrealloc`, vous veillerez à étendre une zone si la zone qui lui succède est bien libre et de taille suffisante (pensez aux cas où elle est de taille égale et où elle est de taille supérieure). Si la zone qui suit la zone à ré-allouer n'est pas assez grande ou pas libre, on veillera à réallouer un nouveau bloc de la bonne taille, à recopier les données nécessaires (celles que le programme utilisateur a pu y mettre) puis à libérer l'ancienne zone qui était utilisée.

Voici deux exemples pour illustrer deux cas de `realloc`.

##### Exemple : `realloc(@a3,16)`

La zone qui suit `@a3` est occupée, on doit ré-allouer et recopier. On sait que la zone est occupée en comparant l'adresse `p` et la dernière adresse de la zone `@a3` (calcul qui dépend de `@a3` et de la taille de `@a3`).



##### Exemple : `realloc(@a3,9)`

La zone qui suit `@a3` est libre, et assez grande pour le `realloc`, on change les informations de taille et le chaînage.

Nota : l'information dans le header de `@a4` n'est plus pertinente et sera écrasée par les données utilisateur

