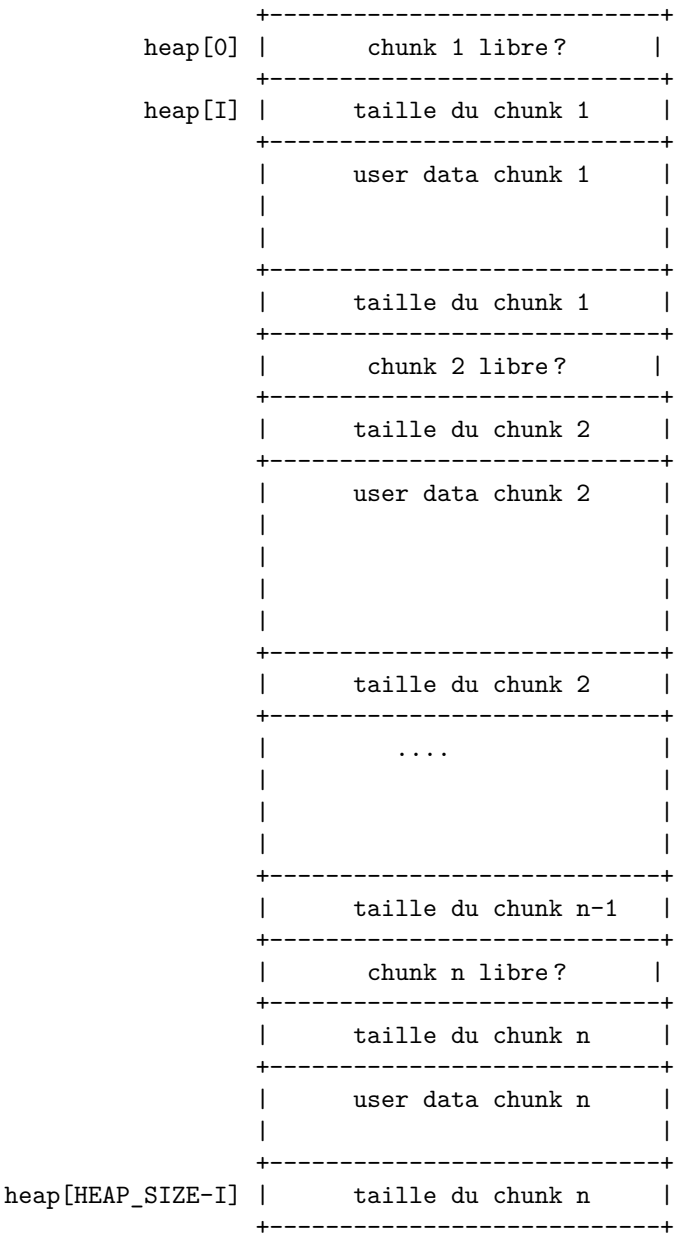
Vous **devez** utiliser l'outil `make` pour compiler vos programmes. Les options `-Wall -W -Werror` devront être utilisées.

Le but de l'exercice est de vous faire implémenter une version très simplifiée des fonctions `malloc` et `free` de la libc. Le principe est le suivant, on suppose qu'on dispose d'une variable globale :

**unsigned char** `heap[HEAP_SIZE];`

qui représente un tableau d'octets (de plusieurs kilo-octets). Le but d'un allocateur mémoire et de permettre de réserver des zones de tailles variables à l'intérieur de ce tableau et de les libérer quand on en a plus besoin.

Pour ce faire, nous allons organiser la mémoire contenue dans ce tableau sous forme de *chunks* ou blocs. Un bloc contient un entier nous précisant s'il est libre ou utilisé, sa taille et les données allouées. On retient également une deuxième fois la taille du bloc **après** les données. Ceci nous permet de connaître la taille du block précédent facilement. On peut représenter cette organisation sous la forme schématique suivante :




Si on connait l'adresse d'un bloc (c'est à dire un pointeur vers l'entier qui dit s'il est libre), alors la taille du bloc contient le nombre d'octet à ajouter à cette adresse pour obtenir l'adresse du bloc suivant (c'est à dire l'adresse de l'entier qui dit s'il est libre). Si on suppose qu'un entier occupe $I$ octets par exemple, la taille d'un chunk sera $I \times 3 + S$ où $S$ est la taille des données de l'utilisateur (le paramètre du malloc donc).

Afin de faciliter l'écriture, on utilisera la structure suivante :

```
typedef struct
{
    unsigned int free;
    unsigned int size;
    /* ces champs ne sont pas écrit dans la heap
       mais sont affectés par get_chunk */
    unsigned char *addr;
    unsigned char *next_chunk;
    unsigned char *previous_chunk;
} chunk;
```

**Exercice 1 : Implémentation de l'allocateur**

**Q 1.** Ecrire les fonctions :

```
unsigned int get_int(void *ptr)
void set_int(void *ptr, unsigned int val)
```

qui obtiennent et stockent respectivement un `unsigned int` à l'adresse pointée par `ptr`. Pensez à utiliser les casts.

**Q 2.** Ecrire la fonction :

```
void set_chunk(chunk *c, unsigned char *ptr)
```

qui stocke les informations contenues dans la structure pointée par `c` à partir de l'adresse `ptr`. `ptr` représente le début du chunk. Vous devrez donc :

1. Ecrire la valeur de `c->free` à l'adresse `ptr`
2. Ecrire la valeur de `c->size` à l'adresse `ptr + sizeof(unsigned int)`
3. Ecrire la valeur de `c->size` à l'adresse `ptr + c->size - sizeof(unsigned int)`

**Q 3.** Ecrire la fonction :

```
void get_chunk(chunk *c, unsigned char *ptr)
```

qui lit les informations situées à partir de l'adresse `ptr` et les stocke dans la structure pointée par `c`. En plus des champs `free` et `size` vous devrez mettre à jour les pointeurs `addr`, `next_chunk` et `previous_chunk` en fonction de `ptr` et de la taille du chunk lue. Attention, si `ptr` pointe sur le premier chunk du tableau `heap`, vous affecterez `NULL` au champ `previous_chunk`. De même, si `ptr` pointe sur le dernier chunk du tableau `heap`, vous affecterez `NULL` au champ `next_chunk`

**Q 4.** Ecrire la fonction :

```
void init_alloc()
```

qui initialise le tableau `heap` grâce à la fonction `set_chunk`.

**Q 5.** A l'aide des fonctions `get_chunk` et `set_chunk`, écrire la fonction :

```
void *my_malloc(unsigned int size);
```

qui alloue un bloc de `size` octets dans le tableau `heap`. Vous devez donc :

1. Rechercher le premier chunk libre de taille suffisante pour contenir `size` octets. Il devra donc être au moins de taille `size + 3*sizeof(unsigned int)` ;
2. Mettre à jour le chunk trouvé pour indiquer qu'il est occupé et indiquer sa nouvelle taille ;
3. Créer un nouveau chunk, à la fin de celui dont on vient de modifier la taille, qui sera libre et dont la taille sera telle qu'il occupera la place laissée libre après la mise à jour de la taille du chunk occupé.

Par exemple, si on considère des entiers de taille 1 octet (ce qui ne sera probablement pas le cas en réalité mais permet de faire des schéma plus simple), un tableau `heap` de taille 10 octets contenant 1 seul chunk libre sera stocké en mémoire comme suit :

```
+----+----+----------------------------+----+
| 1  | 10 |  zone de 7 octets          | 10 |
+----+----+----------------------------+----+
```

Après un appel à `malloc(3)`, le tableau `heap` devra contenir :

```
+----+----+------------+----+----+----+----+----+
| 0  | 6  |   3 octets | 6  | 1  | 4  |    | 4  |
+----+----+------------+----+----+----+----+----+
                                      ^--------- zone de 1 octet
```

**Q 6.** Toujours à l'aide des fonctions `get_chunk` et `set_chunk`, écrire la fonction :

**void** `my_free(`**void** `*ptr)`

qui libère l'espace pointé par `ptr`. Vous devez donc :

1. Trouver l'adresse de début du chunk correspondant à la zone mémoire `ptr`. Elle est simplement 2 entiers avant en mémoire ;
2. Marquer le chunk comme étant libre ;
3. Si le chunk précédent et/ou le chunk suivant sont également libres, fusionner ces 2 ou 3 chunks de façon à n'en faire plus qu'un dont la taille est la somme des tailles des chunks libres.

Par exemple, si le tableau `heap` est dans l'état suivant et qu'un entier est stocké sur 1 octet (encore une fois, j'insiste sur le fait que ce n'est qu'un exemple !) :

```
+----+----+--------------------+----+----+----+------------+----+----+----+----+----+
| 1  | 8  |    5 octets        | 8  | 0  | 6  |   3 octets | 6  | 1  | 4  |    | 4  |
+----+----+--------------------+----+----+----+------------+----+----+----+----+----+
```

Alors, appeler la fonction `free` en lui passant en paramètre le pointeur vers la zone de 3 octets placera le tableau `heap` dans l'état suivant :

```
+----+----+-------------------------------------------------------------------+----+
| 1  | 18 |                        15 octets                                  | 18 |
+----+----+-------------------------------------------------------------------+----+
```

**<u>Exercice 2</u> : Test approfondi de votre allocateur**

Pour pouvoir tester de façon approfondie votre allocteur, nous allons l'utiliser pour lancer des programmes usuels comme `ls, who, cat,...`. La véritable fonction `malloc` est utilisée par presque tous les programmes installés sur votre machine. Cette utilisation se fait au travers d'une librairie partagée.

Cette librairie est chargée automatiquement par le programme quand il fait appel à la fonction `malloc`. Le système linux permet de forcer le chargement d'une librairie **avant** la librairie standard et donc de « surcharger » des fonctions s'y trouvant. Il est donc possible de fabriquer vous même une librairie partagée qui contient les fonctions `malloc, free, calloc` et `realloc` et de forcer un programme à utiliser votre librairie plutôt que la librairie standard.

**Q 1.** Implémentez les fonctions `malloc`, `free`, `calloc` et `realloc` (vous devrez faire appel aux fonctions écrites dans l'exercice précédent). Vérifiez que vos fonctions répondent bien à la spécification des fonctions de la librairie standard. En particulier, il est stipulé dans la page de man de la fonction `free` : *If ptr is **NULL**, no operation is performed.*

**Q 2.** Modifiez votre `Makefile` pour construire une librairie partagée nommée `alloc.so`. Vous devez :

1. ajouter l'option `-fPIC` aux options de **compilation** (`CFLAGS`) ;
2. ajouter l'option `-shared` aux options de l'édition de liens.

**Q 3.** Exécuter la commande `ls` en utilisant la ligne de commande suivante :

```
user@host:~$ LD_PRELOAD=./alloc.so ls
```

L'affectation de la variable d'environnement aura pour effet de forcer l'utilisation des fonctions contenues dans votre librairie plutôt que celles de la librairie standard.

**Attention** Vous ne pouvez plus faire appeller explicitement `init_alloc` dans le `main` vu que celui ci ne sera pas appellé (il ne devrait d'ailleurs même plus être présent dans la librairie). Vous devez donc trouver un moyen de faire l'initialisation automatiquement au premier appel d'une de vos fonction si elle n'a pas déjà été faite. Pensez à l'utilisation de variable `static`.