TD XVIII

PROGRAMMATION D'UN ALLOCATEUR MÉMOIRE

Le but de ce TP et d'écrire un allocateur dynamique de mémoire utilisant un tableau alloué en début de programme. Afin d'éviter de manipuler différents types de pointeurs, notre allocateur ne renverra que des pointeurs de type `uint64_t*` (pointeurs vers des blocs d'entiers non signés de 64 bits). La signature de notre allocateur sera donc :

```
uint64_t *malloc_ui(uint64_t size);
void free_ui(uint64_t *p);
```

La fonction `malloc_ui` prend en paramètre une taille `size` et renvoie un pointeur `p` de type `uint64_t*` vers une portion mémoire pouvant accueillir `size` entiers `uint64_t`. Si le tas ne dispose plus de place, elle renvoie le pointeur `NULL`. La fonction `free_ui` est utilisée avec un pointeur non nul renvoyé par `malloc_ui`. Son rôle est de libérer la mémoire.

Afin de gérer notre tas, nous utiliserons une constant `HEAP_SIZE` et un tableau statique `heap` :

```
#define HEAP_SIZE 32
uint64_t heap[HEAP_SIZE];
```

Enfin, afin d'initialiser une zone mémoire obtenue *via* un appel à `malloc_ui`, le « client » pourra utiliser la fonction suivante :

```
void set_memory(uint64_t *p, uint64_t n, uint64_t value) {
    for (uint64_t i = 0; i < n; i++) {
        p[i] = value;
    }
}
```

Remarques générales

- Comme `heap` est une variable globale, elle est initialisée à zéro (toutes les cases sont nulles). Cependant, nous ferons comme si son contenu initial était imprévisible.
- Dans tout le sujet, quand on parle du « bloc d'indice `i` » ou de la « zone mémoire d'indice `i` » ou toute expression équivalente, on veut dire que la première case *accessible par l'utilisateur* porte l'indice `i` (ou précédemment accessible, si la zone a depuis été libérée). Dans la plus grande partie du sujet, il y aura des informations relatives à cette zone stockées dans la case $i-1$.
- On suppose que l'utilisateur ne fait que des appels licites à nos fonctions. Plus précisément :
  - il n'appelle `malloc_ui` qu'avec des tailles strictement positives ;
  - il n'appelle `free_ui` que sur des pointeurs issus directement de `malloc_ui` (et une seule fois sur un pointeur donné) ;
  - après avoir récupéré un pointeur `p` par `malloc_ui(size)`, il n'effectue que des accès licites (entre `p[0]` et `p[size - 1]`) ;
  - après un appel à `free`, il n'accède plus à la mémoire libérée.

Remarque

Vous trouverez un certain nombre de remarques dans l'énoncé. Elles expliquent pourquoi certaines stratégies étudiées peuvent être intéressantes, même si elles ne sont pas satisfaisantes pour implémenter le couple `malloc / free`. Typiquement, elles peuvent être utilisées soit *en plus* d'un allocateur général (allocateur linéaire), soit pour gérer rapidement certains types d'allocation dans le cadre d'un allocateur général (allocation de blocs de taille fixe). Ces remarques peuvent être sautées lors d'une première lecture[1].

1. *Remarque :* la remarque à laquelle cette remarque fait référence ne s'applique pas à elle-même.

1 Allocateur linéaire

Dans cette partie, nous organisons notre mémoire `heap` comme une suite contiguë de portions mémoire entre les indices `1` et `heap_size - 1`. La case `heap[0]` joue un rôle spécifique : elle contient un entier `end` tel que les portions réservées se trouvent parmi les cases d'indices $1, \dots,$ `end - 1`. Lors de la prochaine allocation, la nouvelle portion sera placée à partir de l'indice `end`. Dans cette stratégie naïve, la libération de mémoire n'a pas d'effet sur le tas. Comme on ne libère jamais, la fonction `free_ui` est triviale :

```
void free_ui(uint64_t *p) {}
```

Afin d'initialiser notre structure, c'est-à-dire la première case du tableau `heap`, on écrira une fonction de signature

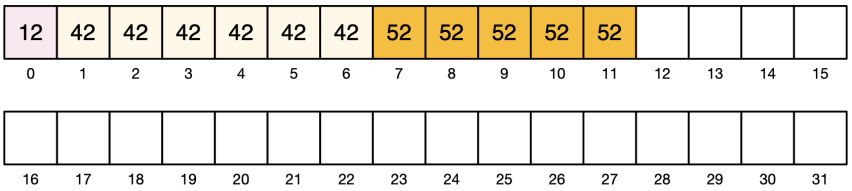
```
void init_heap(void);
```

à la question 1.
Notre code pourra donc être utilisé de la manière suivante. On crée ici 2 tableaux d'entiers de taille respective 6 et 5 que l'on initialise respectivement avec les valeurs 42 et 52. On libère enfin ces tableaux.

```
init_heap();
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(5);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 5, 52);
free_ui(p2);
free_ui(p1);
```

Après initialisation de le mémoire, le tableau `heap` est dans l'état suivant. Dans toute la suite de ce TP, une case vide signalera soit une case mémoire qui n'est pas initialisée, soit une case mémoire dont la valeur ne nous intéresse plus.



► **Question 1** Écrire la fonction `init_heap` qui initialise le tas pour que `heap[0]` contienne une valeur appropriée pour cette stratégie.

► **Question 2** Écrire la fonction `malloc_ui`, et déterminer sa complexité. Cette fonction renverra le pointeur nul (`NULL`) si la requête ne peut être satisfaite.

Remarque
On parle d'*allocateur linéaire* pour un allocateur qui utilise cette méthode. C'est bien sûr complètement inadapté dans le cas général (la mémoire n'étant jamais libérée...), mais cela a l'avantage d'être extrêmement rapide. En *complément* d'un `malloc` « général », cela peut être très intéressant. Un exemple typique est celui d'un ensemble d'objets temporaires qui vont tous cesser d'être utiles en même temps : dans ce cas, le bloc complet pourra être libéré en une seule fois par un seul appel au « vrai » `free`.

2 Réservations de blocs de taille fixe

Nous cherchons désormais à permettre la réutilisation de la mémoire libérée. Nous proposons pour cela une nouvelle stratégie d'implémentation. Nous fixons une variable globale `block_size` et nous allouerons systématiquement des bloc de taille `block_size` : si on nous demande un bloc plus petit, il y aura de la mémoire non utilisée, et si l'on nous demande un bloc plus grand, l'allocation échouera.

```
uint64_t block_size = 8;
```

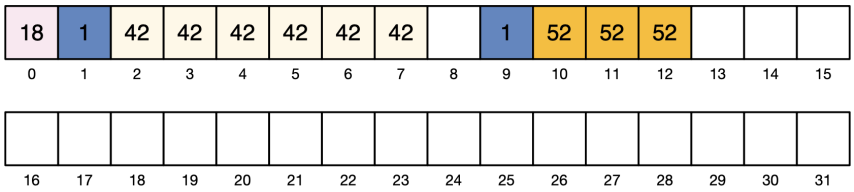
Une portion mémoire `i` réservée avec la taille n (où $n+1 \leqslant$ `block_size`) occupe $n+1$ cases d'un tel bloc. L'*en-tête* `heap[i - 1]` vaut 1 si la portion est encore réservée, ou 0 si elle a été libérée. Les cases suivantes sont utilisées pour stocker les données. Comme précédemment, la case `heap[0]` est réservée pour donner l'indice `i` de la prochaine portion libre pour créer un bloc lorsqu'aucun recyclage de bloc libéré n'est possible.

Une fois le tas initialisé et les opérations suivantes effectuées

```
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(3);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 3, 52);
```

l'état du tas est donné par le tableau ci-dessous. Le pointeur `p1` pointe vers la portion mémoire d'indice `i = 2` tandis que le pointeur `p2` pointe vers la portion mémoire d'indice `i = 10`.



► **Question 3** Écrire la fonction `init_heap` pour cette stratégie.

► **Question 4** Écrire une fonction `is_free` renvoyant `true` si la portion mémoire `i` est libre, et `false` sinon. Créez de même les fonctions `set_free` et `set_used` permettant de changer l'en-tête de la portion mémoire d'indice `i`.

```
bool is_free(uint64_t i);
void set_free(uint64_t i);
void set_used(uint64_t i);
```

Pour réserver une nouvelle portion, on cherche en priorité à réutiliser un bloc laissé libre par une précédente libération. La portion libre pointée par `heap[0]` n'est utilisée que si un tel bloc n'existe pas.

► **Question 5** Écrire la fonction `malloc_ui` pour cette stratégie d'implémentation, et déterminer sa complexité. On renverra `NULL` lorsque la taille `n` est trop grande vis-à-vis de `block_size`.

► **Question 6** Écrire la fonction `free_ui` pour cette stratégie d'implémentation, et déterminer sa complexité. Si possible (c'est-à-dire si la zone libérée est là « plus à droite »), on mettra à jour `heap[0]`, sinon on marquera simplement la zone comme étant libre.

Remarques

- Pour un « vrai » couple `malloc`/`free`, la taille du tas varie au cours de l'exécution du programme : quand il n'y a pas de zone libre suffisamment grande pour satisfaire la requête, `malloc` réclame de la mémoire au système d'exploitation, ce qui se traduit par un déplacement de la limite droite du tas. Quand de la mémoire est libérée par `free`, elle est à nouveau disponible pour l'application (évidemement), mais elle n'est en général « rendue » au système d'exploitation, *sauf* s'il s'agit de la zone la plus à droite. Essentiellement, les mises à jour de `heap[0]` correspondent ici à ces interactions avec le système d'exploitation :
  - — incrémenter cette valeur revient à réclamer davantage de mémoire à l'OS ;
  - — décrémenter cette valeur revint à rendre de la mémoire à l'OS.
- Notre allocateur est très limité puisqu'il ne gère qu'une seule taille d'allocation. Une telle stratégie ne serait bien sûr jamais utilisée seule, mais elle peut être utile pour gérer efficacement les petites allocations (on utiliserait quand même une technique permettant d'accélérer la recherche d'une zone libre).

3 Zones mémoire avec en-tête et pied de page

Nous abordons maintenant une autre stratégie d'implémentation qui permettra de ne pas limiter autant la taille de chaque portion mémoire. Nous munissons pour cela chaque portion d'une en-tête mais aussi d'un *pied de page*. Pour chaque portion, ces deux cases additionnelles contiennent la même valeur : un entier encodant deux informations sur la portion courante. La première information indique si la portion est réservée ou pas. La deuxième indique la taille réservée à cette portion. Nous imposons que cette taille soit toujours un entier pair. Si un programmeur demande à réserver une zone de taille n impaire, nous attribuerons une taille $n+1$ à cette zone, mais le programmeur n'est pas autorisé à consulter cet espace supplémentaire. Si la taille de la portion est $2k$ et la portion est utilisée, l'en-tête et le pied de page contiendront la valeur $2k+1$. Si la portion est libre, ils contiendront la valeur $2k$. Pour une zone d'indice i et de taille $2k$, l'en-tête sera donc située en $i-1$ et le pied de page en $i+2k$.

Nous utiliserons deux portions spéciales, une portion *prologue* et une portion *épilogue*. Ces deux portions spéciales sont de taille nulle et toujours marquées réservées. Nous les plaçons en début et en fin de la zone de réservation. La portion prologue se situe à une position fixée donnée par la variable globale suivante.

```
uint64_t prologue = 2;
```

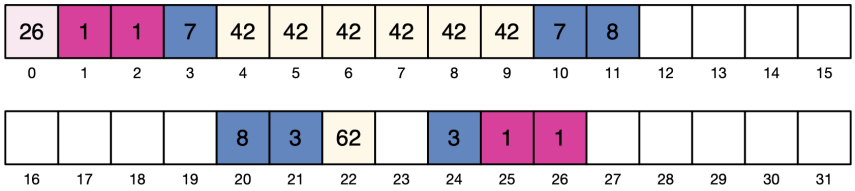
La case `heap[0]` indique l'indice de l'épilogue. L'épilogue est contigu au prologue au démarrage, puis est déplacé vers des indices plus élevés quand la zone des portions réservées doit être agrandie, ou vers des indices plus faibles lorsque cette zone rétrécit.

Après initialisation du tas et appels suivants par le programmeur

```
uint64_t *p1 = malloc_ui(6);
uint64_t *p2 = malloc_ui(7);
uint64_t *p3 = malloc_ui(1);
```

```
set_memory(p1, 6, 42);
set_memory(p2, 7, 52);
set_memory(p3, 1, 62);
free_ui(p2);
```

la figure suivante présente le contenu de `heap`.



► **Question 7** Écrire les fonctions `is_free` et `read_size` permettant de savoir si une portion mémoire `i` est libre où non et de connaître sa taille.

► **Question 8** Écrire les fonctions `set_free` et `set_used` permettant de définir l'en-tête et le pied de page d'une portion de mémoire `i` et de fixer sa taille à `size`.

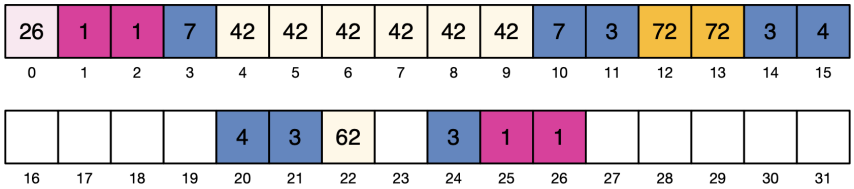
► **Question 9** Écrire les fonctions `next` et `previous` permettant respectivement d'obtenir l'indice de la portion mémoire suivante et de la portion mémoire précédente. Ces fonctions renverront respectivement les indices du prologue et de l'épilogue lorsque la portion mémoire `i` est la première ou la dernière zone de notre tas.

```
bool is_free(uint64_t i);
uint64_t read_size(uint64_t i);
void set_free(uint64_t i, uint64_t size);
void set_used(uint64_t i, uint64_t size);
uint64_t next(uint64_t i);
uint64_t previous(uint64_t i);
```

▶ **Question 10** Écrire la fonction `init_heap` pour cette stratégie.

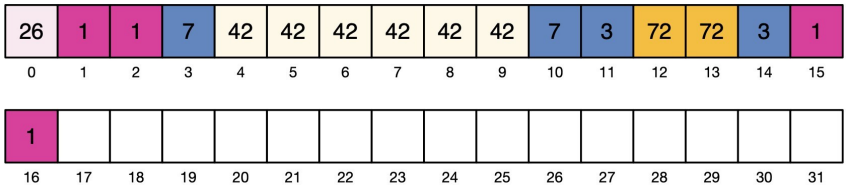
À part les deux portions spéciales épilogue et prologue, toutes les portions ont une taille strictement positive. Lors de la réservation d'une nouvelle portion, on réserve en priorité dans la zone mémoire comprise entre le prologue et l'épilogue, et en dernier recours, on déplace l'épilogue. Si une portion libre est suffisamment grande, une réservation dans cette zone la sépare en une portion réservée et une portion libre. La figure suivante illustre ce mécanisme en présentant le contenu de la mémoire de la figure précédente après l'appel suivant :

```
uint64_t *p4 = malloc_ui(2);
set_memory(p4, 2, 72);
```



▶ **Question 11** Écrire la fonction `malloc_ui` pour cette stratégie, et déterminer sa complexité.

Lors de la libération d'une portion, on étudie les portions adjacentes libres et on réalise si possible une fusion afin qu'il n'y ait jamais deux portions adjacentes libres après un appel à la fonction `free_ui`. De plus, si possible, on déplace l'épilogue vers la gauche. La figure ci-dessous illustre ce mécanisme en présentant le contenu de `heap` après l'appel de `free_ui(p3)`.



▶ **Question 12** Écrire la fonction `free_ui` pour cette stratégie et déterminer sa complexité. Expliquer l'intérêt des portions prologues et épilogues.

▶ **Question 13** Pourquoi est-il important de fusionner les zones libres adjacentes ? Cela suffit-il à régler entièrement le problème ?

▶ **Question 14** La stratégie d'allocation utilisée ici est dite *first fit* : on utilise la première zone libre suffisamment grande pour satisfaire notre allocation. Une autre stratégie courante est dite *best fit* : on utilise la *plus petite* zone libre suffisamment grande pour satisfaire notre allocation. Quels sont les avantages et inconvénients d'une telle stratégie ?

4 Chaînage explicite des portions libres

Nous souhaitons maintenant améliorer l'implémentation de la partie précédente pour accélérer la recherche de portions libérées. Nous allons pour cela maintenir une *chaîne des portions libres*. Il s'agit d'une séquence de portions libres organisée de la manière suivante.

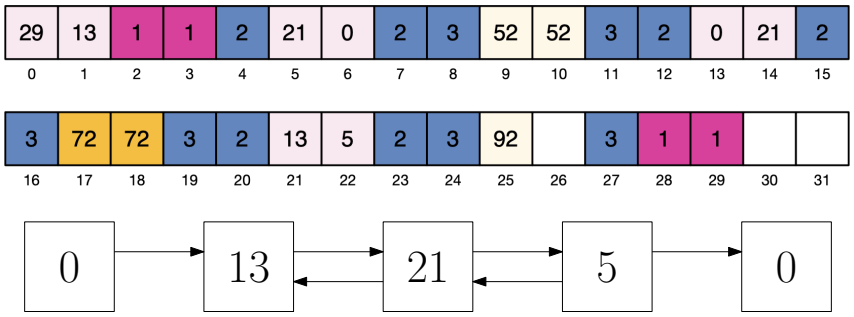
- Dans chaque portion libre `i` dans la chaîne, on stocke une information dans les cases `heap[i]` et `heap[i + 1]`.
  - La case `heap[i]` contient l'indice de la portion libre prédécesseur dans la chaîne.
  - La case `heap[i + 1]` contient l'indice de la portion libre successeur dans la chaîne.
- L'indice de l'entrée de la chaîne est stockée dans la case `heap[1]`. Par convention, elle vaut 0 si et seulement si la chaîne est vide. Si la chaîne n'est pas vide, son premier élément est une portion dont le prédécesseur vaut 0. De même, elle contient un dernier élément, éventuellement égal au premier, dont le successeur vaut 0. Toutes les autres portions de la chaîne ont des prédécesseurs et successeurs non nuls.

Après initialisation du tas et appels suivants par le programmeur

```
uint64_t *p1 = malloc_ui(2);
uint64_t *p2 = malloc_ui(2);
uint64_t *p3 = malloc_ui(2);
uint64_t *p4 = malloc_ui(2);
uint64_t *p5 = malloc_ui(1);
uint64_t *p6 = malloc_ui(1);
```

```
set_memory(p1, 2, 42);
set_memory(p2, 2, 52);
set_memory(p3, 2, 62);
set_memory(p4, 2, 72);
set_memory(p5, 1, 82);
set_memory(p6, 1, 92);
free_ui(p1);
free_ui(p5);
free_ui(p3);
```

la figure suivante présente le contenu de `heap`, ainsi que la chaîne des zones libres :



► **Question 15** Écrire la fonction `add_begin_chain` qui ajoute la portion libre `i` en tête de la chaîne et en fait son premier élément, la portion `i` n'appartenant pas à la chaîne au moment de l'appel.

```
void add_begin_chain(uint64_t i);
```

► **Question 16** Écrire la fonction `remove_from_chain` qui supprime la portion libre `i` de la chaîne.

```
void remove_from_chain(uint64_t i);
```

► **Question 17** Écrire la fonction `init_heap` pour cette stratégie.

► **Question 18** Écrire la fonction `malloc_ui` pour cette stratégie et déterminer sa complexité.

Pour libérer une portion, on réalise comme dans la partie précédente une fusion avec les éventuelles portions libres adjacentes en mémoire, mais en les supprimant au préalable de la chaîne, puis en ajoutant en entrée de la chaîne la portion libre créée par la fusion. À nouveau, on déplace l'épilogue vers la gauche lorsque c'est possible.

► **Question 19** Écrire la fonction `free_ui` pour cette stratégie et déterminer sa complexité.

► **Question 20** Commenter les avantages de cette stratégie par rapport à la stratégie précédente.

Cet énoncé est une adaptation du sujet d'informatique de tronc commun 2021 du concours XENS (il était donc prévu pour être traité en Python).