La version en ligne de ce PDF est disponible ici

Allocateur de mémoire en C

Programmation 1 (2026 - 2027)

Vincent Lafeychine
vincent.lafeychine@ens-paris-saclay.fr

Lorsque la quantité de mémoire nécessaire à l'exécution d'un programme ne peut être déterminée lors de la compilation, notamment parce qu'elle dépend des entrées de l'utilisateur, le programme peut requérir au *tas* pour allouer dynamiquement de la mémoire.

Le tas est une zone de mémoire réservée aux données non statiques et non locales aux fonctions. Le tas est représenté par une pile (donc contigüe) de *pages de mémoire*, et n'autorise donc que deux opérations :

- **acquérir de la mémoire**, correspondant à obtenir une page à la suite des pages déjà obtenues, et;
- **libérer de la mémoire**, correspondant à redonner la dernière page obtenue.

Ainsi, tant que la dernière page acquise est utilisée, il est impossible de libérer de la mémoire.

Un allocateur mémoire est une bibliothèque, généralement incluse dans la bibliothèque standard de C (appelée *libc*) qui permet d'utiliser efficacement le tas en maximisant l'utilisation des pages déjà acquises. En d'autres termes, l'allocateur mémoire sert d'intermédiaire entre le programme utilisateur et le système.

Un allocateur mémoire offre à l'utilisateur la possibilité de (voir `man 3 malloc`) :

- **réserver un bloc** de mémoire d'une taille `size`, avec `void *malloc(size_t size)`,
- **libérer un bloc** de mémoire `ptr` préalablement réservé, avec `void free(void *ptr)`, et
- **redimensionner un bloc** de mémoire déjà réservé, avec `void *realloc(void *ptr, size_t size)`.

La réservation et le redimensionnement renvoient, soit `NULL` si l'opération n'a pas pu être effectuée, soit l'adresse, dans le tas, du début d'une zone de mémoire suffisamment grande.

Les blocs de mémoire alloués à l'utilisateur ne peuvent pas être déplacés tant qu'ils ne sont pas libérés. Différents algorithmes d'allocation de mémoire ont été décrits pour optimiser l'utilisation des pages tout en réduisant le temps de fonctionnement, l'objectif de ce projet est de programmer le *Doug Lea's malloc*.

Ce projet est divisé en 3 parties, qui ajouteront progressivement des fonctionnalités à l'allocateur mémoire. Chaque partie devra être rendue pour 12h00 le lundi suivant, laissant ainsi une semaine pour chaque partie. Vous devrez ensuite modifier le même projet en réutilisant le code déjà écrit pour réaliser les parties suivantes.

1. Contenu du projet

Le projet se compile à l'aide de la commande `make`, qui ne compilera que les fichiers modifiés. Pour ce projet, vous n'avez pas besoin de modifier le fichier `GNUmakefile`.

Trois binaires sont générés : `playground` (`make`), `run_tests` (`make test`) et `libmalloc.so` (`make bonus`).

- `playground` est un binaire exécutable (`./playground`) pour essayer votre allocateur en appelant la fonction `main` dans `src/main.c`.
- `run_tests` est un binaire exécutable (`./run_tests`) pour tester votre allocateur en utilisant les fichiers présents dans le dossier `tests`.

Si vous constatez une erreur lors de la compilation, ou lors de l'exécution de ces deux commandes, prévenez-nous immédiatement.

- Bonus: `libmalloc.so` est une bibliothèque partagée qui remplace l'allocateur de la *libc* par le vôtre, à essayer en fin de projet. N'oubliez pas que certaines fonctions, comme `printf`, requiert `malloc`.
  - Sur Linux/WSL : `LD_PRELOAD=./libmalloc.so <programme>`.

2. Déroulement du projet

Travail à fournir Il est attendu que vous changiez les fichiers dans le répertoire `src/` (sauf `playground.c`) ainsi que les fichiers de tests `.cc`. Ainsi, il est interdit de modifier le dossier `include/` : Tout changement sera sanctionné.

Modalité de rendu Il est attendu de régulièrement envoyer vos modifications sur l'instance GitLab afin que nous puissions évaluer votre avancement. Le jour du rendu, à midi, votre projet sur GitLab sera récupéré et l'état du projet servira de soumission. Vous devrez également ajouter un fichier de *quelques* lignes, nommé `README.md`, qui contiendra vos retours d'expérience, notamment si vous n'avez pas compris certaines notions.

Évaluation Nous évaluerons la qualité du code, mais surtout sa correction avec l'allocateur présenté dans ce sujet. Il est donc recommandé de ne pas apporter d'améliorations à votre projet. Par ailleurs, les erreurs de segmentation (*segfault*) doivent absolument être évitées et seront sévèrement sanctionnées. Nous évaluerons également la qualité des tests que vous écrirez (voir Section 6).

Périmètre du projet Chaque partie vous impose des fonctions auxiliaires pour vous permettre de réaliser le projet. Nous évaluerons ces fonctions avec uniquement des arguments valides. Par exemple, nous n'évaluerons pas votre fonction avec `NULL` si le contexte est inapproprié.

Nous vous conseillons d'ajouter des assertions dans votre projet afin de détecter plus facilement les erreurs. Lorsque l'expression d'une assertion n'est pas équivalente à la valeur `true`, l'assertion affiche un message d'erreur, comme dans l'exemple ci-dessous, puis interrompt l'exécution du programme :

```
assert(expr);  // playground: <file>:<line>: <function>: Assertion `<expr>` failed.
```

3. Préambule: Arithmétique de pointeurs

L'arithmétique des pointeurs décrit les opérations arithmétiques lorsqu'au moins un opérateur est un pointeur. Nous allons uniquement rencontrer le cas de l'addition (ou la soustraction) d'un pointeur avec un entier et la soustraction de deux pointeurs.

Lorsqu'un pointeur est additionné (ou soustrait) avec un entier, alors l'opération arithmétique effectuée prends en compte la taille de l'objet pointé en mémoire.

Par exemple, si `p` est un pointeur de type `T *` (donc pointe sur des éléments de taille `sizeof(T)`), alors l'opération `p + n` ajoutera `n` fois `sizeof(T)` à `p`.

Nous vous proposons d'exécuter le programme suivant à écrire dans le fichier `main.c` dans le dossier `src` :

```
int main() {
    int  *ptr   = malloc(sizeof(int) * 4);
    int  *ptr_1 = ptr + 1;
    char *ptr_2 = (char *)ptr + 1;

    printf("ptr = %p\n", ptr);
    printf("%p - %p\n", ptr_1, ptr_2);
}
```

Pour exécuter ce programme, il vous suffit de lancer la compilation avec la commande `make` puis de lancer l'exécutable avec la commande `./playground`. Observez et analysez la sortie du programme, puis faites de même avec la soustraction de deux pointeurs.

4. Allocation de pages mémoires (`playground.c`)

Comme évoqué en introduction, l'allocation de mémoire auprès du système fonctionne selon un principe similaire à celui d'une pile. Dans le langage commun, on parle de *program break*, un « curseur » représentant la fin du tas dans la mémoire virtuelle.

En poussant le « curseur » vers la droite, la taille du tas augmente, ce qui a pour effet de demander de la mémoire au système. De même, en poussant le « curseur » vers la gauche, la taille du tas diminue, indiquant ainsi au système que la mémoire n'est plus utilisée.

Ce « curseur » peut être déplacé grâce aux fonctions `brk` / `sbrk` (voir `man 2 brk`).

Pour vous permettre de développer ce projet dans les meilleures conditions, nous vous proposons un environnement de développement qui simule l'allocation de pages de mémoire avec la fonction suivante:

```
void *move_program_break(intptr_t size);
```

Il n'est pas nécessaire de modifier ou de comprendre le code présent dans le fichier `playground.c`.

Bonus: Vous pouvez à tout moment quitter l'environnement de développement en utilisant la bibliothèque partagée `libmalloc.so` (voir Section 1). Au lieu d'utiliser la fonction `move_program_break`, votre allocateur utilisera automatiquement la fonction `sbrk`.

5. Alignement (`alignment.c`)

L'alignement est une contrainte imposée aux adresses mémoire, limitées à certains multiples de valeurs.

Comme vous le verrez en cours d'« Architecture et système », les interactions avec la mémoire nécessitent de respecter un alignement à des fins d'optimisations (1). De plus, la correction de certains programmes, notamment `ls`, requiert que l'alignement soit respecté pour fonctionner (2).

Pour ce projet, la taille des pages (1) et l'alignement des pointeurs (2) seront à prendre en compte.

Pour effectuer l'alignement de vos valeurs, vous allez écrire deux fonctions dans le fichier `alignment.c`:

```
size_t align_floor(size_t value, size_t alignment);
size_t align_ceil(size_t value, size_t alignment);
```

Ici, `size_t` est un entier non signé, ayant la capacité (principalement 64 bits sur les ordinateurs actuels) de stocker n'importe quelle taille en mémoire (voir `man 3 strlen`).

La fonction `align_floor` arrondit la valeur `value` *vers le bas* à l'alignement `alignment` qui doit être une puissance de 2. Similairement, La fonction `align_ceil` arrondit la valeur `value` *vers le haut* à l'alignement `alignment` qui doit être une puissance de 2. Nous ne considérerons pas le cas du dépassement d'entier.

Par exemple, `align_floor(20, 8)` et `align_floor(16, 8)` devra renvoyer la valeur `16`. Similairement, `align_ceil(20, 8)` et `align_ceil(24, 8)` devra renvoyer la valeur `24`.

Pour écrire ces fonctions, nous vous conseillons de regarder les opérateurs bit à bit `&`, `|` ainsi que `~`.

6. Écriture de tests (`tests/`)

Pour vous assurer de la conformité de vos fonctions et éviter toute régression logicielle, nous vous proposons d'utiliser la bibliothèque Google Test afin d'écrire vos différents tests.

Vous allez écrire des tests pour les deux fonctions que vous venez d'écrire concernant l'alignement. Ainsi, vous allez modifier le fichier `tests/alignment.cc`.

Pour définir un nouveau test, Google Test met à disposition la macro `TEST` qui prend en argument le nom de la suite de tests (dans ce projet, on utilisera le nom du fichier, soit `alignment` dans ce cas) et le nom du test (par exemple, `align_floor` pour les tests correspondant à `align_floor`).

Dans le corps du test, Google Test met à disposition différentes macros, dont la macro `ASSERT_EQ` qui prend deux arguments et vérifie leur égalité. Si les arguments sont des pointeurs, la macro compare la valeur des pointeurs (donc, compare les adresses) et non le contenu des valeurs pointées.

Par ailleurs, même si vous implémenter un allocateur, vous avez la possibilité d'utiliser `malloc` pour vos tests.

Il est conseillé de définir un nouveau test pour chaque fonctionnalité de votre fonction, plutôt qu'un seul test. Ainsi, il sera plus facile de repérer le test qui échoue.

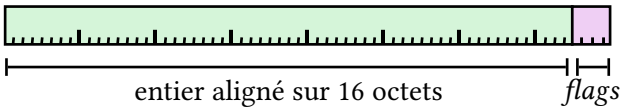
Pensez à envoyer régulièrement vos modifications sur GitLab. Faites-le maintenant si ce n'est pas déjà fait.

7. Stockage d'information dû à l'alignement (`word.c`)

Les pointeurs renvoyés par l'allocateur devront respecter une contrainte d'alignement de `PTR_ALIGNMENT`, soit 16 octets. Cela signifie que la différence entre deux pointeurs renvoyés par l'allocateur (donc, la taille d'un bloc de mémoire de l'allocateur) sera toujours un multiple de 16.

Ainsi, si l'on souhaite stocker la taille des blocs de mémoire (ce qui sera fait en Section 8), les 4 derniers bits ne sont pas nécessaires, car ils seront forcément à 0. Il s'ensuit qu'il est possible de stocker d'autres informations dans cet espace restreint de 4 bits.

Pour utiliser cette propriété, un type d'entier 64 bits sera utilisé, nommé `word_t`. Ce type stockera à la fois un entier en vert, aligné sur 16 octets, ainsi que des informations en violet, appelées *flags*, sur les 4 derniers bits. Un *flag* est un type d'entier, nommé `flag_t`, qui est une puissance de 2, activant uniquement l'un de ses bits.



Pour manipuler ce type de stockage, vous allez écrire cinq fonctions dans le fichier `word.c` :

```
bool word_is_flag(word_t word, flag_t flag);
word_t word_set_flag(word_t word, flag_t flag);
word_t word_clear_flag(word_t word, flag_t flag);

size_t word_get_value(word_t word);
word_t word_set_value(word_t word, size_t value);
```

Un premier *flag* est présent dans le projet: `FLAG_CHUNK_FREE` (voir Section 10), qui équivaut à la valeur `0b1`.

Afin de manipuler l'entier d'un `word_t`, la fonction `word_get_value` permet d'extraire l'entier, et la fonction `word_set_value` permet d'écrire un entier, tout en conservant les *flags* présents.

Afin de manipuler les *flags* d'un `word_t`, la fonction `word_is_flag` vérifie si le *flag* est activé et les fonctions `word_set_flag` / `word_clear_flag` permettent d'activer/désactiver un *flag*, tout en conservant l'entier.

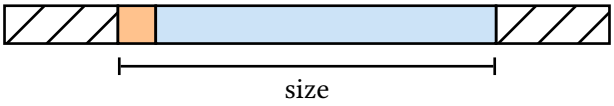
Nous ne considérons pas les cas où l'entier n'est pas aligné, et où le *flag* est plus grand que `PTR_ALIGNMENT`.

Par exemple:

- `word_is_flag(64, FLAG_CHUNK_FREE)` renvoie `false`.
- `word_get_value(word_set_flag(64, FLAG_CHUNK_FREE))` renvoie `64`.
- `word_is_flag(word_set_value(word_set_flag(64, FLAG_CHUNK_FREE), 32), FLAG_CHUNK_FREE)` et `word_is_flag(word_set_flag(64, FLAG_CHUNK_FREE), FLAG_CHUNK_FREE)` renvoient `true`.

8. Bloc de mémoire (`chunk.c`)

Le bloc de mémoire, appelé « *chunks* », représente une partie de la mémoire de l'allocateur. Il est composé d'une en-tête en orange (`chunk_t`, de taille `chunk_metadata_size`) contenant toutes les métadonnées du *chunk*, puis de la charge utile en bleu, appelé *payload*, contenant les données de l'utilisateur.



Pour cette partie, la seule métadonnée est le champ `word` qui contiendra la taille du *chunk* (regroupant l'en-tête et le *payload*, appelée « size » dans la figure ci-dessus) ainsi que des *flags* pour décrire l'état du *chunk*.

Pour manipuler les blocs de mémoire, vous allez écrire neuf fonctions dans le fichier `chunk.c` :

```
bool chunk_is_flag(chunk_t *chunk, flag_t flag);
void chunk_set_flag(chunk_t *chunk, flag_t flag);
void chunk_clear_flag(chunk_t *chunk, flag_t flag);

size_t chunk_get_size(chunk_t *chunk);
void chunk_set_size(chunk_t *chunk, size_t size);

void *chunk_get_payload(chunk_t *chunk);
chunk_t *chunk_from_payload(void *payload);

chunk_t *chunk_get_next(chunk_t *chunk);

chunk_t *chunk_split(chunk_t *chunk, size_t size);
```

Les cinq premières fonctions sont des fonctions tierces qui manipulent le champ `word` d'un *chunk*, de manière similaire à ce qui a été fait dans la Section 7. Il est conseillé d'utiliser les fonctions que vous avez écrites.

Lorsqu'un utilisateur appellera l'allocateur, seul un pointeur d'un *payload* suffisamment grand sera renvoyé. Ainsi, les deux fonctions `chunk_get_payload` / `chunk_set_payload` permettent d'obtenir un pointeur vers le *payload* en ayant un pointeur vers l'en-tête, et inversement. Il peut être utile de relire la partie sur l'arithmétique des pointeurs si nécessaire.

Les *chunks* se suivent en mémoire, et l'en-tête (donnant la taille du chunk) permet de trouver le *chunk* suivant. La fonction `chunk_get_next` permet d'obtenir le prochain *chunk* en lisant la taille du *chunk* actuel. Par ailleurs, afin de conserver l'alignement faite en Section 7, si un utilisateur demande une taille de mémoire non-alignée, il se verra attribuer un bloc de mémoire (*payload*) plus grand (en Section 10), ce qui n'aura aucune incidence.

Pour terminer, la fonction `chunk_split` sépare un *chunk*, dit « source », en deux *chunks* contigus, sans tenir compte du *payload*. D'une part, la taille du *chunk* source sera réduite à la taille spécifiée en argument, et ses *flags* seront conservés. D'autre part, l'espace désormais vacant sera utilisé pour un nouveau *chunk*, dont la taille est donc connue, dont aucun *flag* ne devra être actif, et qui sera le *chunk* renvoyé par la fonction. Nous considérons que la taille du *chunk* source est suffisante pour faire la séparation.

Pour écrire des tests, n'hésitez pas à utiliser `malloc` comme suit:

```
TEST(chunk, ...) {
    size_t size = PTR_ALIGNMENT * 4;            // On souhaite réserver un bloc aligné.
    chunk_t *chunk = (chunk_t *)malloc(size);   // On alloue la mémoire nécessaire:
    chunk_set_size(chunk, size);                //  - on initialise la taille du chunk.
    chunk_<set/clear>_flag(chunk, 0b1);         //  - on initialise un drapeau.
```

Encore une fois, pensez à envoyer régulièrement vos modifications sur GitLab.

9. Arène (`arena.c`)

L'arène est une structure en mémoire qui est présente dès l'initialisation du programme et qui permet de conserver toutes les données nécessaires à l'allocateur. Lors de l'exécution du programme, une liste contigüe de *chunks* se formera, l'allocateur devra alors au moins conserver dans l'arène le premier *chunk*, appelé *first chunk*, ainsi que le dernier *chunk*, appelé *top chunk*.

Pour l'arène, vous allez écrire le contenu de sa structure ainsi que cinq fonctions dans le fichier `arena.c` :

```
struct arena_s { ... };

chunk_t *arena_get_first_chunk();
chunk_t *arena_get_top_chunk();
void arena_set_top_chunk(chunk_t *chunk);

bool arena_initialise(size_t size);
bool arena_increase_top_chunk(size_t size);
```

Pour cette partie, l'arène devra uniquement stocker un pointeur sur le *first chunk* et un pointeur sur le *top chunk*. Comme la structure de l'arène n'est connue que de ce fichier, les fonctions `arena_get_first_chunk` et `arena_get_top_chunk` permettent respectivement de récupérer le *first chunk* et le *top chunk*.

La fonction `arena_set_top_chunk` permet d'assigner un nouveau *top chunk*. Cette fonction intervient lorsque le *top chunk* est séparé avec `chunk_split`, créant ainsi un nouveau *chunk* après le *top chunk*, qui devient alors le nouveau *top chunk*.

Les fonctions `arena_initialise` et `arena_increase_top_chunk` déplacent le *program break*, discuté en Section 4, afin de récupérer des *pages de mémoire* du tas. À des fins d'optimisations, il faut respecter une contrainte d'alignement sur les *pages de mémoire* de `PAGESIZE` : seules des augmentations d'un multiple de `PAGESIZE` devront être effectuées.

À l'initialisation du programme, le tas est vide et l'arène ne contient que des valeurs initialisées à 0 (ce qui signifie que les pointeurs sont `NULL`). La fonction `arena_initialise` ne sera appelée qu'une seule fois et déplace pour la première fois le *program break* d'une valeur, celle reçue en paramètre, arrondie *vers le haut* à l'alignement `PAGESIZE`, puis assigne les valeurs nécessaires pour le *chunk* et pour l'arène.

La fonction `arena_increase_top_chunk` permet d'augmenter le `top_chunk` en déplaçant le *program break* d'une valeur, celle reçue en paramètre, arrondie *vers le haut* à l'alignement `PAGESIZE`, puis assigne les valeurs nécessaires pour le *chunk* uniquement. En effet, la position du `top_chunk` reste inchangée.

Malgré la ressemblance de ces deux dernières fonctions, il est conseillé de dupliquer le code.

10. Allocateur (`dl.c`)

Avec toutes les parties décrites précédemment, un allocateur primitif peut maintenant être écrit. Nous décrivons ci-dessous les différents algorithmes souhaités de manière concise. Pensez donc bien à conserver l'intégrité des données.

Pour définir l'allocateur, vous allez écrire quatre fonctions dans le fichier `dl.c` :

```
void *dl_malloc(size_t size);
void dl_free(void *ptr);
void *dl_realloc(void *ptr, size_t size);

void dl_dump();
```

Algorithme d'allocation `void *dl_malloc(size_t size)`

1. Après avoir vérifié que la taille demandée n'est pas `0` (dans ce cas, renvoyer `NULL`), la taille demandée devra être modifiée pour que l'alignement des prochains blocs reste valide. Ci-après, chaque référence à la « taille demandée » fait référence à la taille après la modification de la taille par cette étape.
2. Si l'arène n'a pas encore été initialisée, il faut l'initialiser avec une taille d'au moins celle demandée.
3. Afin d'éviter de nouvelles allocations, il faut itérer sur chaque *chunk* de l'arène pour chercher un *chunk* libre (voir l'algorithme de libération) de *même* taille que demandé et le renvoyer le premier s'il en existe.
4. Il est maintenant nécessaire de séparer le *top chunk* pour trouver un nouveau bloc. Au préalable, il faut s'assurer que le *top chunk* est suffisamment grand par rapport à la taille demandée, et l'augmenter de la taille demandée si nécessaire.
5. Il suffit maintenant de séparer le *top chunk* en deux, et de renvoyer le premier *chunk*.

Algorithme de libération `void dl_free(void *ptr)`

Après avoir vérifié que le pointeur n'est pas `NULL` (dans ce cas, ne rien faire), il suffit de marquer le bloc comme étant libre en utilisant le *flag* `FLAG_CHUNK_FREE`.

Cet algorithme se complexifiera lors des prochaines parties. N'oubliez pas : nous attendons que votre implémentation soit correct vis-à-vis des algorithmes décrits, n'ajoutez donc aucune optimisation.

Algorithme de réallocation `void *dl_realloc(void *ptr, size_t size)`

1. Après avoir vérifié que la taille demandée n'est pas `0` (auquel cas, il suffit de libérer le bloc avec `dl_free`).
2. Il faut récupérer un bloc de mémoire de taille `size` en appelant `dl_malloc`.
3. Si le pointeur `ptr` n'est pas `NULL`, il faut copier le contenu du bloc de mémoire source (`ptr`) vers le nouveau bloc de mémoire alloué, puis de libérer le bloc de mémoire source. Il peut être utile de regarder `man 3 memmove`.
4. Il suffit maintenant de renvoyer le bloc de mémoire alloué.

De même, cet algorithme n'est pas optimisé : si l'utilisateur demande de réduire la taille d'un bloc, l'algorithme ne devrait pas allouer un autre *chunk* mais simplement réduire le *chunk* actuel.

Affichage des données `dl_dump`

Cette fonction permet d'afficher l'état courant de votre allocateur. Il peut être utile de regarder `man 3 printf`.

Nous évaluerons vos implémentations en comparant la sortie de cette fonction à ce qui est attendu. Nous effectuerons des tests avec des centaines d'allocations et de libérations.

Vous afficherez la taille total prise par votre allocateur, la taille actuelle du *top chunk* ainsi que l'ensemble des *chunks* et leur état. Il est attendu sur cette exemple d'obtenir l'affichage ci-après:

```c
char *ptr_1 = dl_malloc(64);
char *ptr_2 = dl_malloc(64);
dl_free(ptr_1);
dl_dump();
dl_free(ptr_2);
```

```
Total allocated memory: 4096
Top chunk: size: 3936
List of chunks:
 * size: 80 - is_freed: 1
 * size: 80 - is_freed: 0
```