Allocateur de mémoire en C

Programmation 1 (2026 - 2027)

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

La deuxième version de l'allocateur réduit au mieux son empreinte mémoire, notamment lors de la réallocation. Cependant, un dernier défaut subsiste: l'allocation n'est pas efficace. En effet, il est nécessaire de parcourir l'ensemble des *chunks* pour trouver un *chunk* libre à donner, quitte à parcourir l'ensemble des *chunks* avant d'arriver au *top chunk*.

Dans cette partie, nous améliorerons l'algorithme d'allocation grâce à une multitude de listes chaînées, ce qui permettra d'accéder au plus vite à un *chunk* libre.

Un algorithme d'allocation effectuera toujours des décisions, qui peuvent donc être contestées. Généralement, un allocateur essaie d'être performant sur la plupart des cas d'utilisation, que ce soit par rapport aux programmes (très minimaliste comme `ls`, ou consommateur et dans la durée comme un navigateur web) ou par rapport à la machine (un ordinateur personnel ou un système embarqué).

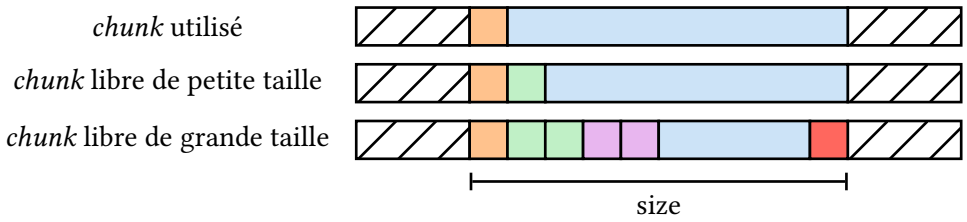
Dans le cas de l'implémentation dans la *GNU LibC*, il est possible de modifier le comportement de l'allocateur : vous pouvez consultez la page dédiée `man 3 mallopt`. Des gains de performances sont possibles lorsque l'environnement d'exécution est connu d'avance.

Pour ce projet, nous considérerons qu'il faut écrire un allocateur assez versatile, qui s'attend à recevoir principalement de petites allocations (inférieures ou égales à 256 octets) et dont l'empreinte doit être minime (il faut déplacer le *program break* « vers la droite » le moins possible).

1. Blocs de petite et grande taille

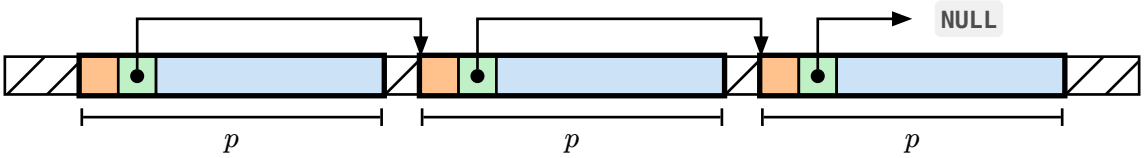
Dans cette partie, nous séparerons les *chunks* en deux catégories : ceux de petite taille (moins ou égal à `SMALL_CHUNK_SIZE_THRESHOLD` octets, soit 256 octets) et ceux de grande taille (donc, de plus de 256 octets).

Il faudra considérer le cas d'un *chunk* utilisé, et les nouveaux cas d'un *chunk* libre de petite et grande taille :

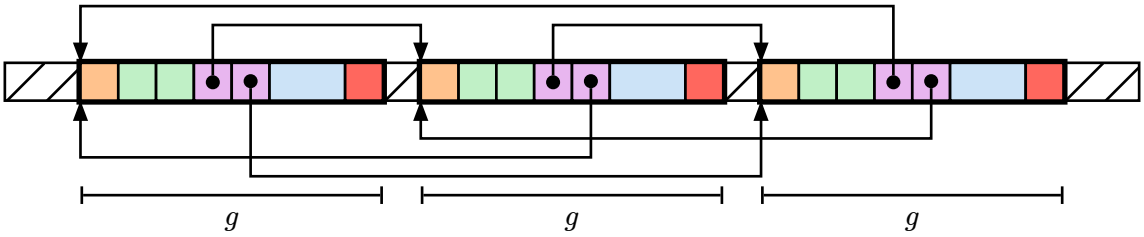


La structure des *chunks* utilisés, quelle que soit leur taille, reste inchangée dans cette partie.

Les blocs libérés de *même* petite taille p, appelés « *cluster* », seront reliés entre eux par une liste simplement chaînée en vert, terminant par `NULL`. Contrairement à la précédente partie, les blocs libérés de petite taille ne possèdent plus la taille du *chunk* en fin de *chunk*.

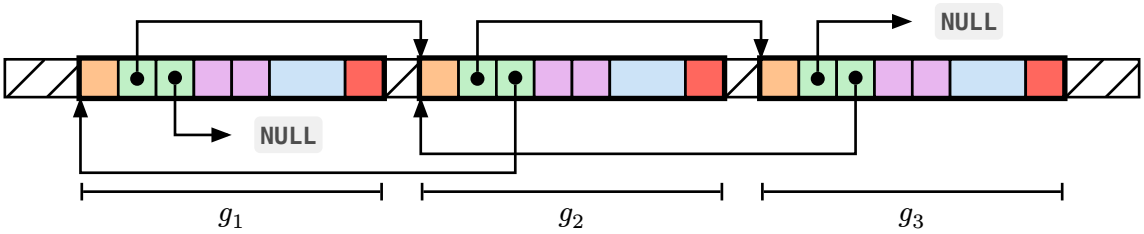


Les blocs libérés de *même* grande taille g, également appelés « *cluster* », seront reliés entre eux par une liste doublement chaînée et circulaire en violet. Pour les blocs libérés de grande taille, la taille reste stockée à la fin du *chunk* en rouge.



Contrairement aux blocs libérés de petite taille, les différents *clusters* de blocs libérés de grande taille sont reliés entre eux par une liste doublement chaînée non circulaire. Chaque *cluster* possède un *chunk* « référent », qui est le seul *chunk* du *cluster* à être relié aux autres *clusters*, alors que les autres *chunks* du *cluster* auront la valeur `NULL`.

Les « référents de *cluster* » de blocs libérés de grande taille g_1, g_2 et g_3, où $g_1 < g_2 < g_3$, seront reliés entre eux par une liste doublement chaînée non circulaire en vert, où les extrémités pointent vers `NULL` :



Ainsi, comme un bloc libéré n'écrit pas forcément sa taille en fin de *chunk*, deux *flags* sont disponibles:

- `FLAG_SMALL_CHUNK_FREE`, qui indique que le *chunk* actuel est petit et libre.
- `FLAG_PREV_LARGE_CHUNK_FREE`, qui indique que le *chunk* précédent est grand et libre, où sa taille est donc disponible en fin de *chunk* (similaire à `FLAG_PREV_CHUNK_FREE` dans la partie précédente).

Pour faire les différentes listes chaînées, vous allez implémenter quatre fonctions dans `chunk.c` :

```
void chunk_link_to_cluster(chunk_t *chunk, chunk_t *chunk_to_link_to);
chunk_t *chunk_unlink_from_cluster(chunk_t *chunk);
chunk_t *chunk_fetch_from_cluster(chunk_t *chunk, size_t requested_size);
void chunk_dump(chunk_t *chunk);
```

La première fonction `chunk_link_to_cluster` prend deux paramètres: le *chunk* que l'on souhaite lier, et un *chunk* du *cluster* ou `NULL` si aucun *cluster* n'existe. Ainsi, de nombreux cas sont à considérer.

Premier cas : Si `chunk` est de petite taille, nous souhaitons l'ajouter à son *cluster*. Si `chunk_to_link_to` est `NULL`, alors `chunk` est le premier membre du *cluster*. Sinon, `chunk_to_link_to` est un *chunk* du *cluster* pour lequel on insère `chunk` après `chunk_to_link_to` dans la liste simplement chaînée.

Second cas : Si `chunk` est de grande taille, nous souhaitons l'ajouter à son *cluster* :

- si `chunk_to_link_to` est `NULL`, alors `chunk` est le premier membre du *cluster* et devient un référent.
- si `chunk_to_link_to` est un *chunk* de même taille que `chunk`, alors il faut ajouter `chunk` dans son *cluster* en l'insérant dans la liste doublement chaînée circulaire (en violet), juste après `chunk_to_link_to`.
- si `chunk_to_link_to` est un *chunk* de taille différente que `chunk`, alors il faut trouver le *chunk* référant dans la liste doublement chaînée (en vert), s'il existe. Si tel est le cas, il suffit d'effectuer le point précédant pour insérer le `chunk`. Si tel n'est pas le cas, il faut insérer le `chunk` dans la liste doublement chaînée (en vert) en tant que référent, tout en conservant la propriété de croissance des référents de la liste chaînée.

Pour les *chunks* de petite taille, il est considéré comme étant une erreur logique si `chunk` et `chunk_to_link_to` ont des tailles différentes. Pour les *chunks* de grande taille, il est considéré comme étant une erreur logique si `chunk_to_link_to` n'est pas un référent. Finalement, pour n'importe quel cas, il est considéré comme étant une erreur logique si `chunk` est déjà lié.

La deuxième fonction `chunk_unlink_from_cluster` prend en paramètre le *chunk* à délier du *cluster*. Dans le cas d'un *chunk* de petite taille, cette fonction délie `chunk` et renvoie l'ancien prochain *chunk* de `chunk`. Dans le cas d'un *chunk* de grande taille, cette fonction délie `chunk` et renvoie l'ancien *chunk* suivant dans son *cluster* qui n'est pas lui-même s'il existe, sinon son ancien *cluster* précedent s'il existe, sinon son ancien *cluster* suivant s'il existe, sinon `NULL`.

La troisième fonction, `chunk_fetch_from_cluster` prend en paramètres un *chunk* et une taille demandée puis recherche dans le *cluster* de `chunk` un *chunk* de la taille souhaitée ou supérieure. Si tel est le cas, le *chunk* est renvoyé tout en restant dans le *cluster*, sinon `NULL` est renvoyé.

La dernière fonction `chunk_dump` permet d'afficher l'état d'un *chunk*, comme suit :

```c
chunk_link_to_cluster(new_small_chunk, small_chunk);
chunk_dump(small_chunk);
chunk_dump(new_small_chunk);

assert(chunk_get_size(large_chunk) == 512);
assert(chunk_get_size(large_chunk_2) == 512);
assert(chunk_get_size(larger_chunk) == 896);

chunk_link_to_cluster(large_chunk_2, large_chunk);
chunk_dump(large_chunk);
chunk_dump(large_chunk_2);
chunk_dump(larger_chunk);

chunk_link_to_cluster(larger_chunk, large_chunk);
chunk_dump(large_chunk);
chunk_dump(large_chunk_2);
chunk_dump(larger_chunk);
```

```
Number of following chunks : 1
```

```
Number of following chunks : 0
```

```
Inside a cluster of size : 2
```

```
Inside a cluster of size : 2
```

```
Inside a cluster of size : 1
```

```
Referent of a cluster of size : 2\n
With neighbours : 0 and 896\n
```

```
Inside a cluster of size : 2
```

```
Referent of a cluster of size : 1\n
With neighbours : 512 and 0\n
```

2. Gestion des corbeilles dans l'arène

Une « corbeille », également appelée « *bin* », recense les *chunks* libres d'une taille fixe pour les *chunks* de petite taille, et les *chunks* libres d'une plage de tailles pour les *chunks* de grande taille.

Il existe 16 tailles différentes pour les *chunks* de petite taille : `PTR_ALIGNMENT` $* k$, pour k allant de 1 à 16. Nous allons ainsi créer 16 corbeilles (`NUMBER_SMALL_BINS`) pour chacun des *clusters* de *chunks* de petite taille.

Les *chunks* de grande taille seront répartis en 15 plages différentes de 2^k à $2^{k+1} - 1$, pour k allant de 8 à 22. Il y aura une dernière plage allant de 2^{23} à `SIZE_MAX` (la valeur maximale d'un `size_t`). Nous allons ainsi créer 16 corbeilles (`NUMBER_LARGE_BINS`) pour chacune des plages de *clusters* de *chunks* de grande taille.

Plus précisément, une corbeille est un pointeur vers un référent du *cluster* de la taille correspondante. Les corbeilles seront accessibles depuis l'arène : il est ainsi nécessaire de modifier la structure `arena_t`.

Pour la gestion des corbeilles, vous allez implémenter quatres fonctions dans `arena.c` :

```c
chunk_t **arena_get_bin(size_t size);
void arena_insert_chunk_to_bin(chunk_t *chunk);
void arena_remove_chunk_from_bin(chunk_t *chunk);
chunk_t *arena_fetch_chunk_from_bin(size_t requested_size);
```

La fonction `arena_get_bin` renvoie l'*adresse* de la case mémoire de la corbeille correspondante à la taille `size` dans l'arène. En recevant l'adresse, l'appelant est capable de modifier le contenu de la case mémoire :

```
chunk_t *chunk = ...;
chunk_t **bin = arena_get_bin(...);
(*bin) = chunk;
```

Pour récupérer l'adresse d'une case mémoire, l'opérateur `&` est disponible dans le langage C. Cet opérateur est l'inverse de l'opérateur `*` qui permet de déréférencer une adresse. Nous considérons que le paramètre `size` est aligné sur `PTR_ALIGNMENT`.

Les trois fonctions suivantes (`insert` / `remove` / `fetch`) sont équivalentes aux fonctions des *clusters* des *chunks* (`link` / `unlink` / `fetch`) décrites dans la partie précédente. Ces trois nouvelles fonctions ont, en plus, la responsabilité de vérifier si le pointeur de la corbeille a été modifié. Ainsi, ces trois nouvelles fonctions utiliseront `arena_get_bin`, suivi de la fonction équivalente pour manipuler les *clusters*.

Concernant `arena_remove_chunk_from_bin`, des dépilements successifs seront nécessaires pour les *chunks* de petite tailles. Ainsi, une corbeille temporaire doit être maintenue.

3. Impacts concernant la nouvelle représentation des blocs

Cette nouvelle représentation des *chunks* libres n'impactent pas les fonctions dans `alignment.c` et `word.c`.

Pour `chunk.c`, en plus d'implémenter les fonctions présentées dans les sections précédentes, il faudra :

- modifier la structure `chunk_t` pour intégrer les champs ajoutés par la section précédente.
- limiter l'utilisation de la fonction `chunk_set_footer_size` aux *chunks* de grande taille.
- limiter l'utilisation de la fonction `chunk_get_prev` lorsque le *chunk* précédent est libre et de grande taille.
- considérer que la fonction `chunk_try_merge_backward` n'effectue une action uniquement lorsque le *chunk* précédent est libre et de grande taille, tout en supprimant le *chunk* précédent de son *cluster*.
- considérer que la fonction `chunk_try_merge_forward` n'effectue une action que si le *chunk* suivant est libre et de n'importe quelle taille, tout en supprimant le *chunk* suivant de son *cluster*.

Pour les fonctions `chunk_try_merge_backward` / `chunk_try_merge_forward`, il est recommandé d'utiliser `arena_remove_chunk_from_bin` pour conserver l'intégrité des corbeilles.

Pour `arena.c`, aucune modification n'est à apporter, outre celles décrites dans la section précédente.
Pour `dl.c`, les algorithmes d'allocation et de libération diffèrent et sont présentés dans la dernière section.

4. Doug-Lea malloc (version simplifiée)

L'allocateur devenant complexe, il est conseillé de créer des fonctions intermédiaires dans `dl.c`. Comme ces fonctions n'ont pas vocation à être accessible ailleurs, vous pouvez ajouter le qualificatif `static` comme suit :

```
static void *malloc_copy_free(size_t size) { ... }
```

Algorithme de libération

Dès qu'un *chunk* est libéré, une tentative de fusion avant, et arrière, ne sera effectuée que si le *chunk* actuel est de grande taille et si le *chunk* avant, et arrière, est également de grande taille. En effet, les *chunks* de petite taille restent fragmentées le plus longtemps possible (voir l'algorithme d'allocation).

Si le *chunk* obtenu est le *top chunk*, une tentative de diminution du *top chunk* est effectuée. Sinon, le *chunk* obtenu est placée dans sa corbeille correspondante.

Algorithme d'allocation

Lorsqu'un *chunk* est demandé, il suffit de regarder dans la corbeille correspondante pour voir si un *chunk* a une taille suffisante, et d'effectuer un *split* si nécessaire. Si aucun *chunk* n'est trouvé, il faut alors chercher dans chacune des corbeilles successives. Pour un *chunk* de petite taille, les « corbeilles successives » correspondent d'abord aux corbeilles de petite taille, puis à chacune des corbeilles de grande taille. Concernant l'itération, il est conseillé dans un premier temps d'écrire les valeurs successives (16, 32, …, 256, 512, …) dans un tableau.

Si aucun *chunk* n'est trouvé, il ne faut pas immédiatement déplacer le *program break*. En effet, la mémoire est désormais fragmentée, il peut y avoir des *chunks* adjacents libres de petite taille, entrecoupés de *chunks* libres de grande taille. Il faut entamer une procédure de « nettoyage ».

Ainsi, itérativement, chaque *chunk* de chaque corbeille (en commençant par les corbeilles de petite taille, puis par les corbeilles de grande taille) effectuera une tentative de fusion comme suit :

0. Dès lors qu'un *chunk* d'une taille suffisante aura été trouvé après une fusion, le nettoyage s'arrête.
1. Pour les *chunks* de petite taille, seule une tentative de fusion en avant sera effectuée.
2. Pour les *chunks* de grande taille, une tentative de fusion en arrière et en avant sera effectuée.
    1. Après une fusion réussie, le *chunk* obtenu sera replacé directement dans la corbeille correspondante, afin qu'il puisse être traité par la procédure actuelle de nettoyage.
    2. Cependant, si le *chunk* n'a pas été agrandi, il ne peut pas être replacé directement dans sa corbeille, ce qui impliquerait que la procédure ne se termine jamais : Prenez le cas des *chunks* de petite taille par exemple. Ainsi, une corbeille temporaire doit être maintenue par la fonction d'allocation pour traiter les *chunks* de la corbeille courante, non agrandis après une fusion.

Si aucun *chunk* n'est trouvé après la procédure de nettoyage, il est nécessaire de déplacer le *program break*.

Algorithme de réallocation

Avec les changements précédents, l'algorithme de réallocation ne diffère que dans le cas d'une réallocation où le *chunk* suivant est libre, car la mémoire est maintenant fragmentée (voir l'algorithme d'allocation).

Il est nécessaire d'itérer sur autant de blocs libres que nécessaire pour effectuer une fusion. Si la fusion potentielle n'est pas assez grande, aucune fusion n'est effectuée et un « *malloc-copy-free* » s'impose.

Par exemple, considérons un *chunk* c de taille 1000 suivi par des *chunks* libres de taille 100, 1000 et enfin 200. Si une réallocation de c demande une taille 2000, il faudra fusionner le prochain *chunk* de taille 100 ainsi que le *chunk* de taille 1000, tout en laissant le dernier *chunk* de taille 200 intact.