

Projet de Programmation Système : Implémentation d'un Allocateur Mémoire

Penda THIAO

Université de Versailles Saint Quentin en Yvelines

13/12/2024



- ① Implémentation Basique
- ② Analyse des Performances
- ③ Optimisations
- ④ Conclusion

- ① Implémentation Basique
- ② Analyse des Performances
- ③ Optimisations
- ④ Conclusion

Implémentation Basique - Partie 1

my_malloc :

- Alloue de la mémoire avec `mmap` pour une taille donnée.
- Si la taille est valide, `mmap` est utilisé avec les options `MAP_PRIVATE` | `MAP_ANONYMOUS`.
- Retourne un pointeur vers la mémoire allouée ou `NULL` en cas d'échec.

my_free :

- Libère la mémoire allouée avec `munmap`.
- Vérifie que le pointeur est valide et que la taille est non nulle avant d'appeler `munmap`.
- Retourne 0 en cas de succès, -1 en cas d'échec.

Implémentation Basique - Partie 2

Tests Unitaires :

- **Allocation simple** : Test d' allocation d' un bloc de 1024 octets.
- **Écriture et lecture** : Vérification de la lecture/écriture dans la mémoire allouée.
- **Libération** : Test de la libération correcte de la mémoire avec `my_free`.
- **Allocation nulle** : Vérification que `my_malloc(0)` retourne `NULL`.
- **Libération sur NULL** : Vérification que libérer un pointeur `NULL` avec `my_free` retourne une erreur.
- **Libération après allocation** : Vérification de l'échec de la double libération d' un même bloc de mémoire.

Implémentation Basique - Partie 3

Conclusion : Les tests unitaires ont confirmé le bon fonctionnement de l'implémentation dans les cas spécifiés.

```
penda@penda-Latitude-7480:~/Documents/AISE_25_Projet$ make
gcc -Wall -g -c src/tests.c -o build/tests.o
gcc -Wall -g build/my_allocator.o build/tests.o -o build/my_allocator_tests
penda@penda-Latitude-7480:~/Documents/AISE_25_Projet$ ./build/my_allocator_tests
Running tests...
Test 1 passed: Allocation simple
Test 2 passed: Écriture et lecture
Test 3 passed: Libération
Test 4 passed: Allocation nulle
Test 5 passed: Libération sur NULL
All tests passed!
penda@penda-Latitude-7480:~/Documents/AISE_25_Projet$
```

- ① Implémentation Basique
- ② Analyse des Performances
- ③ Optimisations
- ④ Conclusion

Analyse des Performances

Nous avons comparé les performances de notre allocateur personnalisé (utilisant `mmap/munmap`) et de `malloc/free` en mesurant le temps d'allocations et de désallocations pour un bloc de taille 1024 octets et un total de 100 000 allocations. Les résultats montrent que `malloc/free` est nettement plus rapide, en raison de l'optimisation de la gestion de la mémoire dans la heap par rapport à `mmap`, qui est plus coûteux en termes de gestion de la mémoire.

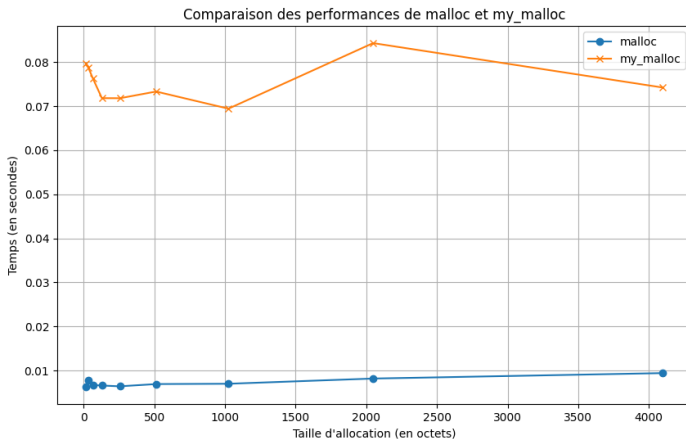
Comparaison des Performances

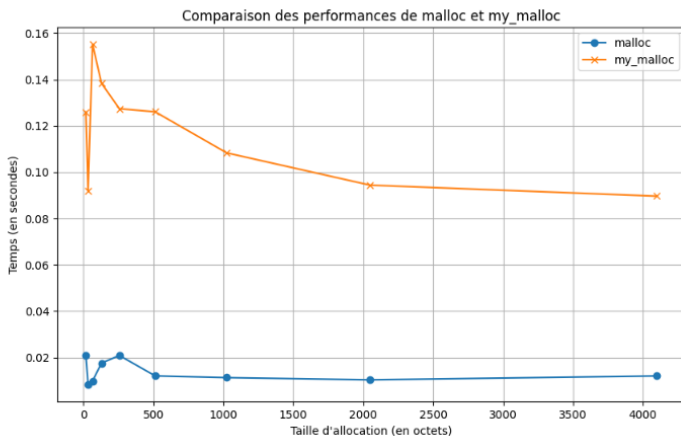
Méthode	Taille du bloc	Nombre d'allocations	Temps écoulé (secondes)
my_malloc/my_free (mmap/munmap)	1024 octets	100 000	0.941169
malloc/free	1024 octets	100 000	0.003448

表 1: Comparaison des performances avec malloc/free

```
penda@penda-Latitude-7480:~/Documents/AISE_25_Projet$ ./build/performance_test
Test de my_malloc et my_free (utilisant mmap/munmap)...
Temps écoulé (my_malloc/my_free) : 0.941169 secondes
Test de malloc et free...
Temps écoulé (malloc/free) : 0.003448 secondes

Comparaison des performances :
Temps avec my_malloc/my_free : 0.941169 secondes
Temps avec malloc/free : 0.003448 secondes
penda@penda-Latitude-7480:~/Documents/AISE_25_Projet$
```





Identification des Goulots d'Étranglement

Plusieurs facteurs peuvent ralentir le processus d'allocation et de désallocation de mémoire dans notre allocateur personnalisé par rapport à `malloc/free` :

- **Gestion de la mémoire avec `mmap/munmap`** : L'utilisation de `mmap` pour allouer de la mémoire est plus coûteuse en termes de temps d'exécution par rapport à `malloc`, qui utilise la heap du système.
- **Fragmentation de la mémoire** : La gestion des blocs de mémoire et la recherche de blocs libres peuvent entraîner une fragmentation, augmentant les coûts des allocations futures.
- **Fréquence des appels système** : Chaque appel à `mmap` et `munmap` implique un appel système, ce qui peut être lent comparé à l'allocateur de heap de `malloc/free`.

Ces goulots d'étranglement expliquent la différence de performance observée entre notre allocateur personnalisé et `malloc/free`.

- ① Implémentation Basique
- ② Analyse des Performances
- ③ Optimisations**
- ④ Conclusion

Optimisations

Dans cette section, nous abordons les différentes optimisations mises en place pour améliorer l'efficacité de l'allocation et de la gestion de la mémoire.

Gestion des blocs de mémoire et des classes de tailles

Nous utilisons une méthode de gestion de mémoire par *listes libres* pour chaque classe de tailles. La mémoire est allouée à des tailles spécifiques qui sont des puissances de 2. Pour améliorer la performance, nous avons divisé la mémoire en *classes* selon la taille du bloc de mémoire. Chaque classe contient une liste libre associée, ce qui permet de réutiliser des blocs déjà alloués.

- **Classes de tailles** : Nous définissons un tableau de tailles maximales pour chaque classe, allant de 1 octet à 1024 octets. Cela permet de regrouper les allocations en fonction de leur taille, réduisant ainsi la fragmentation.
- **Alignement** : Les tailles sont alignées à la puissance de 2 la plus proche pour optimiser l'allocation.

Classe	Taille maximale (octets)
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512


Algorithme de recherche et d'allocation

`my_malloc` est la fonction principale d'allocation de mémoire.

Lorsqu'une allocation est demandée, la fonction :

- **Aligne la taille** : La taille demandée est arrondie à la puissance de 2 la plus proche pour optimiser l'allocation.
- **Recherche un bloc libre** : La fonction vérifie si un bloc de la taille demandée est disponible dans la liste libre correspondante.
- **Alloue de la mémoire via `mmap`** : Si aucun bloc n'est disponible, de la mémoire est allouée avec `mmap`, garantissant que la mémoire soit isolée et privée.

Étape	Action
Alignement	Taille ajustée à la puissance de 2 supérieure
Recherche	Bloc libre vérifié dans la liste correspondante
Allocation	Utilisation de <code>mmap</code> si nécessaire

表 3: Étapes principales de l'allocation. 

Libération et recyclage de la mémoire

Lorsque la mémoire est libérée via `my_free`, elle est remise dans la liste libre correspondante. Un compteur des pointeurs actifs est également mis à jour.

Un mécanisme de *fusion* des blocs adjacents est utilisé pour réduire la fragmentation. Cela se fait par la fonction `coalesce_free_blocks`, qui parcourt la liste des blocs libres et fusionne les blocs voisins afin de créer des blocs de taille plus grande.

Bloc A	Bloc B
Libre	Libre
Taille x	Taille y
Fusionné : Taille $x + y$	

表 4: Fusion des blocs adjacents.

Affichage et suivi des pointeurs actifs

Pour le débogage, nous avons inclus une fonctionnalité permettant d'afficher le nombre de pointeurs actifs en mémoire. Cela est fait à l'aide de la fonction `print_active_pointers` qui imprime le nombre actuel de pointeurs actifs.

Conclusion des optimisations

L'utilisation des classes de tailles, de l'alignement des tailles à la puissance de 2, et de la fusion des blocs adjacents permet d'améliorer les performances de l'allocation mémoire, en réduisant la fragmentation et en réutilisant les blocs de mémoire de manière efficace. Ces optimisations garantissent une gestion de la mémoire plus rapide et plus stable.

Optimisation	Avantage
Classes de tailles	Réduction de la fragmentation
Alignement	Allocation plus rapide
Fusion	Réutilisation efficace des blocs

表 5: Résumé des optimisations.

Évaluation des performances

Afin de valider les optimisations mises en place pour l'allocateur de mémoire, nous avons réalisé une série de tests comparatifs entre notre implémentation (`my_malloc` et `my_free`) et l'allocateur système standard (`malloc` et `free`). Trois scénarios de test ont été définis:

- **Test 1:** Taille des blocs jusqu'à 1024 octets, incréments de 8 octets, 100000 itérations.
- **Test 2:** Taille des blocs jusqu'à 2048 octets, incréments de 16 octets, 50000 itérations.
- **Test 3:** Taille des blocs jusqu'à 4096 octets, incréments de 32 octets, 20000 itérations.

Évaluation des performances

Compilation sans optimisation system

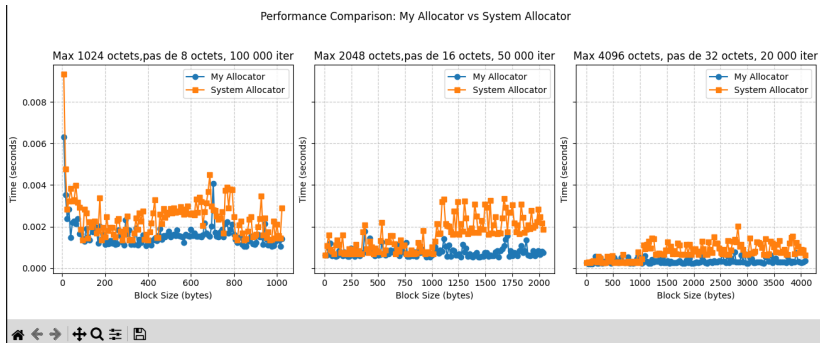


图 1: Performance d'allocation pour différents scénarios de test

Évaluation des performances

Compilation avec O2

Performance Comparison: My Allocator vs System Allocator

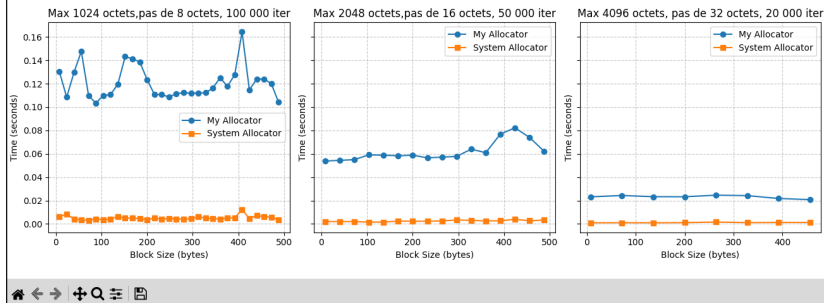


图 2: Performance d'allocation pour différents scénarios de test

Évaluation des performances

Compilation avec O3

Performance Comparison: My Allocator vs System Allocator

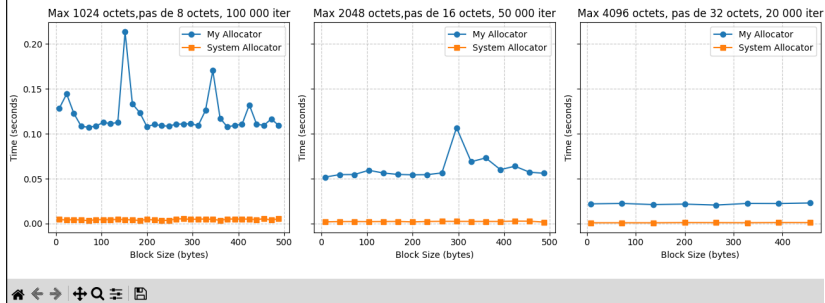


图 3: Performance d'allocation pour différents scénarios de test

Évaluation des performances

Ces résultats valident l'efficacité des optimisations mises en place et mettent en lumière l'efficacité de notre allocateur pour les blocs de grande taille.

- ① Implémentation Basique
- ② Analyse des Performances
- ③ Optimisations
- ④ Conclusion

Conclusion

Le développement s'est déroulé en plusieurs étapes :
implémentation initiale des fonctions `my_malloc` et `my_free`,
validation par tests unitaires, et comparaison des performances
avec l'allocateur système. Par la suite, diverses optimisations ont
été explorées, notamment l'utilisation de listes libres segmentées, le
recyclage des blocs libérés, et la fusion des blocs adjacents, afin de
réduire la fragmentation et d'améliorer l'efficacité. Ce travail a
permis de maîtriser des concepts fondamentaux de gestion mémoire
bas niveau tout en développant des compétences en optimisation
des performances et analyse des goulots d'étranglement.