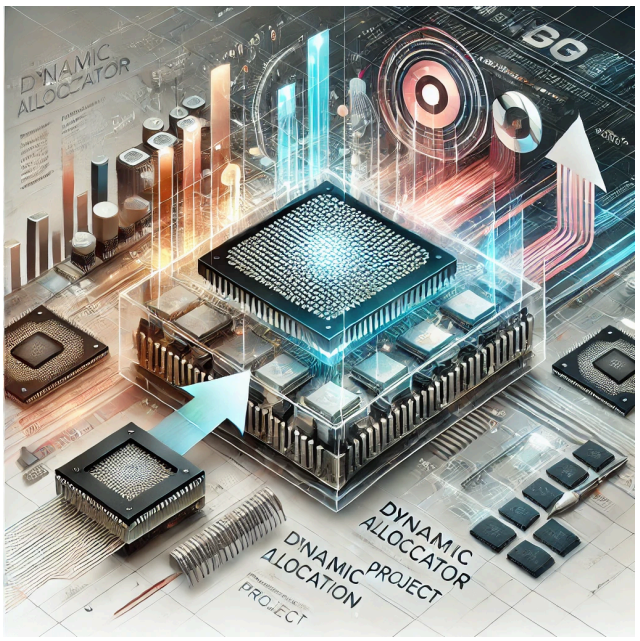
UVSQ
université PARIS-SACLAY

Rapport -Projet de Programmation Système

Implémentation d'un Allocateur Mémoire



Yifan LIyifan.li@ens.uvsq.fr
Bowen LIU bowen.liu@ens.uvsq.fr

Enseignant : Jean-Baptiste Besnard
Email de l'enseignant : jbbesnard@paratools.com

Sommaire

1. Introduction

1.1 L'introduction du projet

Un allocateur de mémoire est un composant clé d'un système informatique. Il gère la distribution et la récupération de la mémoire. Le but de ce projet est de créer un allocateur de mémoire à partir de zéro et de l'optimiser pour améliorer les performances globales du programme.

Les programmes doivent souvent gérer la mémoire de manière dynamique lors de l'exécution, mais l'allocation dynamique de la mémoire pose certains défis :

- La nature dynamique des allocations et libérations rend difficile de déterminer à l'avance la taille exacte de mémoire nécessaire.
- La fragmentation de la mémoire peut réduire l'efficacité de son utilisation.
- Dans les programmes multi-threads, l'allocateur doit garantir la sécurité des threads et une gestion efficace de la concurrence lorsque plusieurs threads demandent de la mémoire en même temps.

Dans les sections suivantes, nous allons nous concentrer sur l'allocation dynamique de mémoire et introduire une série d'optimisations pour améliorer le fonctionnement de l'allocateur.

1.2 Les objectifs du projet

Ce projet vise à améliorer la performance de programmes en optimisant un allocateur de mémoire. Les principaux objectifs sont suivants:

- Implémenter une version initiale d'un allocateur de mémoire en utilisant les appels système mmap et munmap.
- Réduire la fréquence des appels système par l'optimisation afin d'améliorer l'efficacité de la gestion de la mémoire.
- Comparer les performances entre l'allocateur personnalisé et celui fourni par le système, identifier les goulets d'étranglement, et formuler des propositions d'amélioration.

2. La méthodologie et la mise en œuvre

Dans ce projet, pour l'implémentation des opérations de base, nous avons utilisé les appels système *mmap* et *munmap* pour gérer l'allocation et la libération de la mémoire, conformément aux exigences. Chaque fois que *my_malloc* est appelé, l'application demande une nouvelle zone de mémoire directement au système d'exploitation, et *my_free* est chargé de libérer cette zone. Voici les détails précis de cette implémentation :

2.1 Conception de la structure de données

Afin de connaître la taille de chaque bloc de mémoire lors de sa libération, nous avons utilisé une structure de métadonnées, *block_header_t*, qui sert à stocker les informations relatives à la taille de chaque bloc de mémoire.
Cette structure de mémoire de métadonnées nous permet de savoir rapidement où se

trouve chaque zone de mémoire, puis la zone de données allouée. Bien entendu, cette partie introduit d'autres indicateurs dans l'optimisation.

2.2 Implémentation de my_malloc

La fonction *my_malloc* est responsable de l'allocation de blocs de mémoire de taille spécifiée. Voici le processus détaillé :

1. **Calcul de la taille à allouer** : Cela inclut la taille demandée par l'utilisateur et la taille des métadonnées.
2. **Obtention de la taille de la page système** : En utilisant sysconf*(_SC_PAGESIZE)*, nous obtenons la taille de la page pour effectuer un alignement correct de la mémoire.
3. **Alignement de la taille d'allocation** : La taille totale est arrondie à un multiple entier de la taille de la page pour satisfaire les exigences d'alignement du système.
4. **Appel à mmap pour allouer la mémoire** : Nous utilisons l'appel système mmap pour demander de la mémoire, avec les flags définis pour une carte anonyme et des permissions en lecture-écriture.
5. **Stockage des métadonnées** : Avant de retourner la zone mémoire à l'utilisateur, nous stockons les métadonnées (taille du bloc).
6. **Retour du pointeur à l'utilisateur** : Nous déplaçons le pointeur après les métadonnées et le retournons à l'utilisateur.

L'alignement dans cette partie est effectué pour garantir que la taille de la mémoire allouée est conforme à la taille de la page, ce qui améliore les performances et assure la compatibilité.

2.3 Implémentation de *my_free*

La fonction *my_free* sert à libérer un bloc de mémoire alloué par *my_malloc*. Les étapes sont les suivantes :

1. **Vérification de la validité du pointeur** : Si le pointeur est *NULL*, la fonction retourne immédiatement.
2. **Localisation des métadonnées** : À partir du pointeur utilisateur, calculer l'adresse de début des métadonnées (en-tête du bloc).
3. **Lecture de la taille totale allouée** : Extraire la taille totale du bloc à partir des métadonnées.
4. **Libération avec munmap :** Appeler la fonction système *munmap* pour libérer tout le bloc, y compris les métadonnées et les données utilisateur.

2.4 Détection des tests

Voici les résultats de comparaison des performances entre l'allocateur personnalisé et l'allocateur système (10 000 allocations, plage de taille 16B ~ 2048B) :

Nom du test	Allocateur personnalisé (my_malloc/my_free	Allocateur système (malloc/free)	Amélioration des performances

	)		
Test de performance mono-thread	0,001297 secondes	0,001356 secondes	4,5 %
Test de performance multi-thread (4 threads)	0,003618 secondes	0,004921 secondes	36%

2.5 Analyse de l'implémentation de base

À ce stade, l'implémentation de base est terminée, avec les fonctionnalités fondamentales d'allocation et de libération de mémoire. Cependant, cette version présente plusieurs limites. Chaque opération d'allocation ou de libération nécessite un appel à mmap ou munmap, ce qui engendre un coût élevé lié aux appels système. De plus, l'absence de mécanisme de mise en cache empêche une gestion efficace des allocations répétées et la réutilisation de la mémoire libérée. L'allocateur ne peut pas exploiter les fragments de mémoire plus petits que la taille demandée, ce qui accroît la fragmentation. Enfin, les blocs libres adjacents ne sont pas fusionnés, aggravant encore la fragmentation et réduisant l'efficacité de la gestion de la mémoire. Ces problèmes montrent la nécessité d'optimisations pour améliorer les performances globales.

3. Optimisation et analyse des performances

Pour résoudre les insuffisances identifiées dans la version précédente de l'allocateur, nous avons mis en œuvre plusieurs optimisations visant à améliorer les performances et l'utilisation de la mémoire.

3.1 Liste des optimisations

1. **Segmentation en classes de tailles**
   Utiliser des listes de blocs pour différentes tailles (par exemple, des puissances de 2) afin d'optimiser la gestion des allocations.
2. **Recyclage des blocs libérés**
   Implémenter un cache pour les blocs précédemment alloués afin de réduire les appels coûteux à mmap.
3. **Coalescence des blocs libres**
   Fusionner les blocs adjacents libérés afin de réduire la fragmentation mémoire et améliorer l'utilisation globale.
4. **Allocation "best fit"**
   Rechercher le bloc libre dont la taille est la plus proche de celle demandée pour minimiser le gaspillage de mémoire.
5. **Gestion multi-thread avec locks**
   Ajouter un support pour les environnements multi-thread en utilisant des mutex pour protéger les opérations de l'allocateur.

6. **Cache par thread**
   Fournir un cache mémoire dédié à chaque thread pour limiter les accès aux locks globaux et améliorer les performances.
7. **Alignement des blocs mémoire**
   Garantir l'alignement des blocs pour des performances optimales, notamment sur certaines architectures spécifique**s.**
8. **Détection des fuites mémoire**
   Ajouter un suivi des allocations et des désallocations afin d'identifier les éventuelles fuites de mémoire.
9. **Allocation par arène**
   Diviser la gestion de la mémoire en plusieurs arènes distinctes pour limiter la contention dans les environnements multi-thread.
10. **Optimisation des métadonnées**
    Réduire la taille des en-têtes des blocs pour économiser de la mémoire et améliorer l'efficacité globale.

3.2 Détails des optimisations

Dans cette section, nous analyserons en détail la mise en œuvre de ces optimisations.

3.2.1 Gestion des blocs de mémoire par classification de la taille

Dans cette partie de l'implémentation, nous classons les blocs de mémoire par taille afin de minimiser le temps de recherche lors de l'allocation et de la libération de la mémoire.

Les méthodes de l'optimisation

- **Classification préalable des blocs** : Une série de tailles de blocs prédéfinies (par exemple, 8B, 16B, 32B, etc.) est utilisée pour gérer les demandes de mémoire de tailles différentes.
- **Listes libres** : Pour chaque catégorie de taille de bloc, une liste libre indépendante est créée pour stocker les blocs de mémoire inutilisés correspondants.
- **Stratégie d'allocation** : Lors d'une demande de mémoire, la recherche est effectuée dans la liste libre correspondante pour trouver le plus petit bloc satisfaisant la demande.
- **Stratégie de libération** : Lors de la libération de mémoire, le bloc est ajouté à la liste libre de sa catégorie de taille pour une réutilisation future.

Analyse de l'optimisation

Cette méthode réduit considérablement le temps consacré aux opérations d'allocation et de libération, augmentant ainsi l'efficacité. Cependant, l'utilisation de tailles de blocs prédéfinies peut entraîner un gaspillage de mémoire et une fragmentation interne. Pour les demandes dépassant la taille maximale des blocs définis, une nouvelle catégorie est créée, et dans les cas critiques, l'appel système mmap est utilisé pour allouer la mémoire directement, garantissant ainsi une certaine flexibilité.

3.2.2 Réutilisation des blocs de mémoire libérés

Après chaque libération, les blocs de mémoire inutilisés ne sont pas réutilisables dans les allocations futures. Cette optimisation vise à recycler ces blocs libérés pour les rendre disponibles lors de nouvelles allocations, en améliorant la stratégie décrite dans l'optimisation 3.2.1.

Les méthodes de l'optimisation

- **Gestion des listes libres** : Comme dans l'optimisation 3.2.1, les blocs libérés sont marqués comme disponibles et ajoutés à la liste libre correspondante.
- **Priorité à la réutilisation** : Lors d'une nouvelle allocation, les blocs disponibles dans les listes libres sont utilisés en priorité avant toute nouvelle allocation.
- **Cache local par thread** : Chaque thread dispose d'une liste libre locale pour réduire la contention sur les verrous et limiter les appels système.

3.2.3 Fusion des blocs libres adjacents

Lors de chaque libération, des fragments de mémoire inutilisables sont créés. Si une demande dépasse la taille de ces fragments, davantage de mémoire est allouée, augmentant la fragmentation. Cette optimisation introduit un mécanisme pour fusionner les blocs libres adjacents lors des libérations, réduisant ainsi la fragmentation.

Les méthodes de l'optimisation

- **Structure des blocs** : Chaque bloc de mémoire est modifié pour inclure des pointeurs vers les blocs précédents et suivants, facilitant le parcours et la gestion des blocs en mémoire.
- **Logique de fusion** : Lorsqu'un bloc est libéré, on vérifie si les blocs adjacents sont également libres. Si c'est le cas, ils sont fusionnés en un bloc plus grand.
- **Mise à jour des listes libres :** Après la fusion, la liste libre est mise à jour pour inclure le nouveau bloc fusionné.

Analyse de l'optimisation

Bien que cette optimisation ajoute une certaine complexité au processus de libération, elle améliore considérablement l'efficacité lors des allocations répétées en réduisant la fragmentation. De plus, elle augmente les chances de réussir les allocations de grande taille en créant de plus grands blocs de mémoire disponibles grâce à la fusion.

3.2.4 Allocation par meilleur ajustement

Pour gérer les demandes d'allocation, différentes stratégies sont disponibles, comme *sequential fit* et *indexed fit*. Nous avons choisi la stratégie *best-fit* pour optimiser l'utilisation de la mémoire. Cette méthode sélectionne le bloc libre le plus proche de la taille demandée, ce qui réduit la fragmentation.

Les méthodes de l'optimisation

- **Recherche dans la liste libre** : Identifier le bloc le plus adapté dans les listes libres classées par taille.
- **Extension de la recherche** : Si aucune correspondance n'est trouvée, chercher dans des catégories de taille supérieure.
- **Division des blocs** : Si le bloc trouvé est beaucoup plus grand que la demande, le diviser en deux parties : une pour satisfaire la demande, l'autre restant dans la liste libre.

Analyse de l'optimisation
Bien que la stratégie *best-fit* augmente le temps de recherche par rapport à *first-fit*, elle réduit la fragmentation, en particulier pour les petits blocs, ce qui améliore l'efficacité globale.

3.2.5 Support multi-threads

Pour les environnements multi-threads, l'allocateur doit permettre des opérations simultanées sans erreurs. Les solutions actuelles utilisent un verrou global pour protéger les listes libres, mais cela entraîne une contention importante, affectant les performances en cas de forte concurrence.

Les méthodes de l'optimisation

- **Verrou global** : Un mutex global protège les opérations sur les listes libres.
- **Verrou local** : Chaque accès ou modification à une liste libre est sécurisé par un verrou local (*pthread_mutex_lock* et *pthread_mutex_unlock*).

Analyse de l'optimisation
Cette approche garantit un fonctionnement sûr des allocations et libérations dans un environnement multithread, mais elle augmente le coût des performances, surtout en cas de haute concurrence.

3.2.6 Cache local par thread

Pour limiter la dépendance au verrou global et améliorer l'efficacité dans les environnements multi-threads, chaque thread dispose de son propre cache de petits blocs mémoire, réduisant les accès aux listes globales.

Les méthodes de l'optimisation

- **Cache local par thread** : Chaque thread possède un cache local, contenant des listes de blocs libres classés par taille.
- **Allocation mémoire** : Lors d'une demande, on tente d'abord d'allouer depuis le cache local. Si aucun bloc n'est disponible, on consulte les listes globales ou on utilise *mmap.*

- **Libération mémoire** : Les blocs libérés sont ajoutés en priorité au cache local. Si le cache est plein, ils sont transférés aux listes globales.
- **Gestion du cache** : Une limite de taille est imposée au cache local. Les blocs dépassant cette limite sont retournés aux listes globales.

Analyse de l'optimisation
Le cache local réduit la contention sur les verrous globaux et améliore la vitesse d'allocation dans les scénarios impliquant des petits blocs. Cependant, il introduit une certaine perte de mémoire due aux caches locaux.

3.2.7 Alignement des blocs mémoire

L'alignement des blocs mémoire respecte les contraintes matérielles, évitant les erreurs liées à des accès non alignés et améliorant les performances de la mémoire.

Les méthodes de l'optimisation
Les détails de cette optimisation sont expliqués dans la section 2 de l'implémentation de base.

Analyse de l'optimisation
Cette approche améliore la compatibilité et la stabilité du programme tout en augmentant les performances générales de l'allocateur.

3.2.8 Détection des fuites de mémoire

Les fuites de mémoire sont un problème courant pour les allocateurs de mémoire. Elles se produisent lorsque la mémoire allouée n'est pas correctement libérée, entraînant un gaspillage des ressources et une baisse de la stabilité du système.

Les méthodes de l'optimisation

Il est possible d'ajouter une structure pour enregistrer les allocations. Cette structure permettra de suivre chaque bloc de mémoire alloué en enregistrant son adresse, sa taille, et son état (libéré ou non). Les détails sont les suivants :

1. **Enregistrement des allocations** : Après un appel réussi à *my_malloc*, créer une entrée dans la liste des allocations.
2. **Suppression des enregistrements** : Après un appel réussi à *my_free*, retirer l'entrée correspondante de la liste.
3. **Détection des fuites** : À la fin du programme ou lors de la fermeture de l'allocateur, parcourir la liste des allocations pour identifier les blocs non libérés. Ces informations sont affichées via des journaux ou directement dans la console, indiquant la taille et l'emplacement des fuites.
4. **Modification des fonctions** : Mettre à jour *my_malloc et my_free* pour inclure des appels aux fonctions *add_allocated_block et remove_allocated_block* pour gérer les enregistrements.

Analyse de l'optimisation
Cette approche permet de détecter et de corriger les fuites de mémoire après chaque allocation ou libération, réduisant les risques de fuites au fil du temps. Bien qu'elle augmente légèrement le coût en performances, elle facilite grandement la maintenance et l'identification des problèmes.

3.2.9 Allocation basée sur les arenas

Malgré l'ajout de verrous globaux et du cache local par thread, la contention sur les verrous peut encore entraîner une baisse des performances. Pour résoudre ce problème, nous introduisons le concept d'**arenas**. Un arena est une zone de gestion de mémoire indépendante, permettant à chaque thread d'utiliser en priorité sa propre zone, réduisant ainsi les conflits.

Les méthodes de l'optimisation

1. **Comprendre la structure des arenas** : Un arena est une zone indépendante pour gérer la mémoire.Il est responsable de l'allocation et de la libération des blocs de mémoire qui lui appartiennent. Chaque arena maintient une liste de blocs libres, classés par taille.
2. **Création et initialisation des arenas**
    - Utiliser `mmap` pour allouer une région mémoire dédiée à chaque arena.
    - Initialiser les verrous et les listes de blocs libres de chaque arena.
3. **Associer un arena à chaque thread**
    - Chaque thread possède son propre arena.
    - Utiliser le stockage local de thread (TLS) pour enregistrer et gérer l'arena associé au thread.
4. **Logique d'allocation et de libération**
    - Lors d'une allocation, le thread essaie en priorité d'obtenir de la mémoire depuis son propre arena.
    - Si l'arena ne peut pas satisfaire la demande, un nouvel arena peut être créé dynamiquement ou les ressources globales peuvent être utilisées.

Analyse des performances
Cette optimisation partage des similitudes avec celle du cache local par thread (section 3.2.6). Les deux visent à réduire la contention des verrous et à améliorer les performances dans les environnements multithreads. Ainsi, l'implémentation peut être choisie selon les besoins, avec une préférence pour l'optimisation 3.2.6 dans ce cas.

4.Conclusion

Sur la base de l'implémentation d'un allocateur de mémoire fondamental, nous avons tenté d'introduire différentes stratégies d'optimisation. Bien que nous ayons exploré toutes les mécanismes d'optimisation requis par le projet, nous avons constaté l'existence de nombreux conflits entre eux. Par conséquent, nous avons choisi des mécanismes d'optimisation relativement adaptés. En implémentant des mécanismes tels que la gestion de la mémoire et

le cache par thread, nous avons progressivement amélioré les performances et l'utilisation des ressources de l'allocateur de mémoire. Les comparaisons ont révélé que l'allocateur de mémoire de base repose sur une logique de gestion de la mémoire solide. Cependant, de nombreuses voies restent à explorer, notamment l'ajustement dynamique des catégories de tailles, la détection automatique des goulets d'étranglement ou l'intégration de nouvelles stratégies d'allocation.

References

1. Nbsun. (2021, March 15). 多线程环境下内存分配器. Retrieved from https://www.cnblogs.com/nbsun/p/13254565.html
2. Yingnanxuezi. (2020, December 1). 线程缓存（*tcache*）:提高多线程性能的秘密武器. Retrieved from https://blog.51cto.com/yingnanxuezi/12337759
3. Mandagod. (2022, July 7). 高性能内存分配器*jemalloc*基本原理. Retrieved from https://blog.csdn.net/mandagod/article/details/123680210
4. Rong_Toa. (2021, November 24). 内存分配器*ptmalloc*、*jemalloc*、*tcmalloc*调研与对比. Retrieved from https://blog.csdn.net/Rong_Toa/article/details/110689404
5. Weixin_45605341. (2023, February 10). *glibc malloc*内存分配优化策略解读. Retrieved from https://blog.csdn.net/weixin_45605341/article/details/140566108
6. Zhuanlan.zhihu.com. (2023, April 15). 性能优化——内存篇. Retrieved from https://zhuanlan.zhihu.com/p/563180198