UCLouvain

École polytechnique de Louvain

LINFO1252 - Système Informatiques

Projet 0:
Allocation dynamique de mémoire

Groupe 4

***Auteurs**:*
Delsart Mathis
Guerrero Gurriaran Anthony

***Enseignant**:*
Rivière Etienne

***Assistants**:*
Buchet Aurélien
Piraux Maxime
Rousseaux Tom
de Keersmacker François

2022-2023

1 Approche

1.1 Allignement

Notre heap est alignée par blocs de deux octets, ce qui permet une réduction de complexité du code, car nous utilisons des métadonnées de deux octets pour gérer les blocs. Cependant, la complexité temporelle peut être plus élevée comparée à des alignements supérieur à 2 (de 16 octets, par exemple), en raison de l'augmentation du nombre de blocs. Cette augmentation de complexité est compensée par l'efficacité dans la gestion de petites allocations et par la réduction de la fragmentation, améliorant l'utilisation globale de la mémoire.

1.2 Stratégie de parcours : Nextfit

La stratégie de parcours Nextfit est utilisée en raison de sa simplicité et de son efficacité. L'algorithme recherche le premier espace libre depuis le dernier bloc alloué. Cette méthode présente de bonnes propriétés de localité, car les données allouées proche dans le temps sont souvent créées dans un espace continu (= proche dans l'espace). Un autre avantage de Nextfit est sa rapidité d'exécution par rapport à d'autres stratégies de parcours telles que "Bestfit". Cependant, un inconvénient majeur de cette stratégie est la fragmentation, que nous essayons de minimiser via la fusion multiblocs (section 1.3).

1.3 Gestion de la fragmentation : fusion multiblocs

La fragmentation peut entraîner un gaspillage de mémoire et des temps d'accès moins efficaces. Pour minimiser cet effet, nous fusionnons un maximum de blocs libres ensemble, une technique que nous appelons la fusion multiblocs. Le principe est de fuionner chaque bloc libre avec autant de blocs que possible qui le suivent jusqu'à trouver un bloc alloué.
Cependant, réduire la fragmentation a un coût en temps d'exécution (et donc en complexité temporelle) mais cela s'avère nécessaire.

2 Métadonnées

2.1 L'offset Nextfit

Les deux premiers octets de la heap sont utilisés pour stocker la position dans la heap du dernier bloc alloué par un appel à `my_malloc()`.

2.2 Headers

Les headers contiennent la taille du bloc sur deux octets. Les 15 premiers bits sont utilisés pour la taille (qui est paire, donc le dernier bit est inutilisé), et le dernier bit (le 16ème) sert à indiquer si le bloc est alloué ou non (principe de flag).

3 Structure de données

Nous utilisons une liste simplement chaînée avec les différents headers comme nœuds et les tailles des blocs comme valeurs. Cela nous permet d'itérer sur les headers plutôt que sur chaque case de la heap. On parcours la heap entière en O(n), n étant le nombre de blocs (libres et alloués). La complexité temporelle est cependant moins bonne en pratique en raison du coût de la fusion multiblocs.
Nous avons fait le choix d'implémenter notre algorithme avec une liste implicite, car l'avantage de rapidité du parcours de la heap pour une liste explicite aurait été compensé par le coût (en temps d'exécution) de la mise à jour des offsets (= "pointeurs"). De plus, nous avons préféré avoir plus de mémoire et utilisé des métadonnées les plus petites possible car notre heap est fort limité (64000 octets).

4 Fonctionnement du code illustré par les schémas

Veuillez vous référer à la section 5 pour les schémas et la légende associée.

4.1 Étape 1 : Initialisation de la HEAP (appel à `init()`)

Lors de l'appel de la fonction `init()`, deux initialisations sont réalisées :

1. L'offset Nextfit est initialisé à 1 (1 et non 2, car nous itérons sur la heap castée en `uint16_t` en raison de l'alignement pair).
2. Le premier header est initialisé avec la taille du prochain bloc libre, c'est-à-dire 63996.

4.2 Étape 2 : Allocation de mémoire (appel à `my_malloc()`)

À chaque appel de la fonction `my_malloc()` :

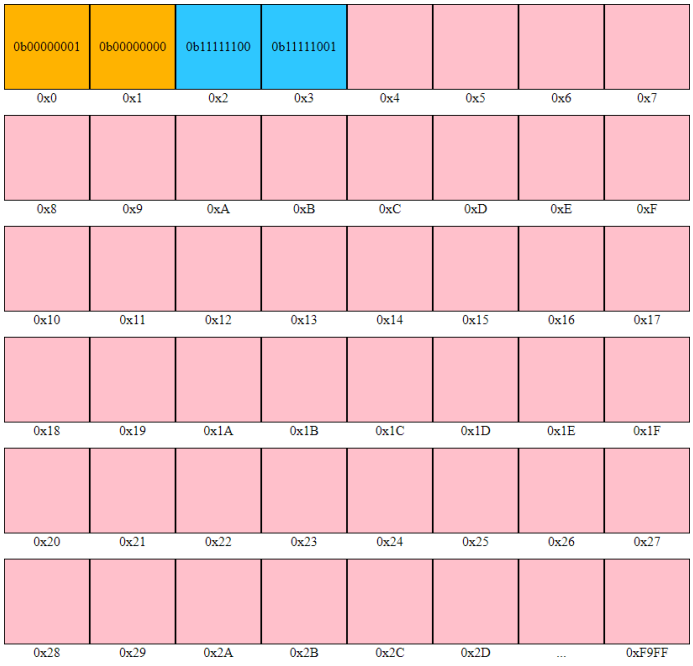
1. Deux headers sont créés, sauf dans le cas spécial en fin de heap où un seul header est créé. Ces headers contiennent respectivement la taille du bloc alloué (augmentée de 1 si impaire) par l'appel et la taille du prochain bloc libre.
2. L'offset Nextfit est mis à jour pour contenir l'index de la heap où se trouve le second header créé (sauf dans le cas spécial, où il est réinitialisé à 1).
3. Un pointeur vers l'adresse de la case qui suit le premier header (= première case alloué par l'utilisateur) est renvoyé à l'utilisateur.

Dans le cas général, plus complexe que cet exemple, il se peut que certains blocs libres ne soient pas suffisamment grands pour la taille demandée. Dans ce cas, la fusion multiblocs est automatiquement appliquée.
Lorsqu'un bloc de taille suffisante est trouvé, s'il est trop grand, il est divisé en deux pour éviter de la fragmentation et une perte de stockage mémoire.

4.3 Étape 3 : Libération de mémoire (appel à `my_free()`)

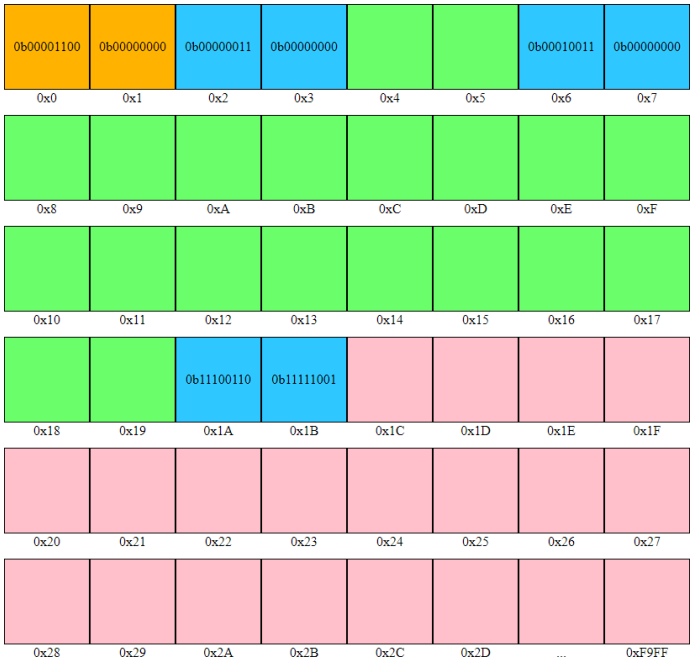
Dans la fonction `my_free()`, nous marquons le bloc alloué comme libre en utilisant le flag dans le header, puis nous effectuons la fusion multiblocs. Cette approche permet de répartir le temps d'exécution entre `my_free()` et `my_malloc()`.
C'est pourquoi, à la fin de la fonction, il ne reste que 3 blocs dans la heap, au lieu de 4.

5 Schémas et légende



Étape 1 : La heap après l'initialisation

Étape 2 : La heap après l'allocation de trois blocs (de taille 1, 17 et 13)



Étape 3 : La heap après la libération du dernier bloc précédemment alloué (le bloc de taille 13)

Légende des différents schémas:

1. **Numéros en-dessous des cases :** Les adresses de la heap (commencent à 0x0 ici pour simplification)
2. **Points rouges :** Pointeur vers l'adresse de la heap, renvoyé par `my_malloc()`
3. **Couleurs:**
   - **Orange:** Métadonnée (Offset Nextfit)
   - **Bleu:** Métadonnées (Headers)
   - **Rose:** Blocs libres
   - **Vert:** Blocs alloués