Laffont Emile, Baladi Ayoub

Compte Rendu TP Mémoire

1. Résumé

Ce qui a été implémenté:

mem_init, mem_show,mem_alloc,mem_free, les 3 stratégies,la prise en compte de l'alignement, mem_realloc et plusieurs tests complémentaire

Nous avons testé l'ensemble avec les tests fournis plus ceux que l'on a implémentés.

Toutes les fonctions cités fonctionnent.

Organisation: Pour les premières fonctions: mem_init, mem_show, alloc, free et les 3 stratégies, on travaillait à deux dessus. Puis pour les tests,les performances et le realloc on s'est réparti le travail pour avancer plus vite.

2. Principes d'implémentation

Nous avons utilisé deux structures chaînées, une pour les blocs libres et une pour les blocs occupés. Avec la taille et l'adresse du maillon suivant.

Pour l'entête on a en premier l'adresse de la fonction fit, puis l'adresse du premier maillon des blocs libres puis celle des occupés.

Cas limite: lorsque l'on a une zone libre et que l'on veut allouer, si il ne reste pas assez en taille : si la structure de contrôle ne rentre pas ou passe tout juste, dans la zone restante alors on l'occupe en entier. On ne prend pas en compte alloc(0), il renvoie un pointeur NULL.

Pour l'alignement chaque structure de contrôle sont alignées ainsi que donc que la zone des données. Pour respecter cet alignement on arrondit la taille des structures de contrôle ainsi que du bloc alloué au multiple supérieur de l'alignement.

3. Structure du code

Les fonctions d'allocation, libération, réallocation et les 3 stratégies ont été implémentées dans mem.c.

Les structures de contrôle ont été définies dans le fichier mem_os.h. Le fichier memshell.c a été modifié pour pouvoir lancer des réallocations.

Pour la fonction mem_free, on utilise une fonction fusion, et pour le realloc de multiples fonctions pour le découper son code. toutes les fonctions se trouvent dans mem.c

4. Tests effectués

Nous avons testé nos fonctions avec les tests fournis, et avons implémenté des tests pour les compléter.

Tests complémentaires:

test_structures_de_controle: qui permet de tester si nos structures de contrôle sont correctement utilisées.

test_alignement: qui vérifie si l'alignement est correctement respecté.

test_free, test_allocation: test de cas de base, des cas limites et de grande série d'instructions.

test_realloc : qui permet de tester le realloc. Tester les différents cas du realloc. realloc(adr, 0) (free(adr)), raccourcir la taille de la zone mémoire, l'agrandir. Dans le cas de déplacement de la zone mémoire on teste aussi si la copie du bloc est correct.

La plupart du temps nos tests ont permis de dévoiler quelque petit bug "tout bête", oubli d'un cast, ou d'une condition...

Tous les tests ont été effectués sous l'environnement des machines de l'im2ag, sous mandelbrot.

5. Compilation et exécution

Se mettre dans le répertoire src du tp.
Pour compiler le code: make all
Pour exécuter le code: ./memshell
Pour exécuter tous les test: make test
Pour exécuter un seul test: ./test_<nom du test> ex : ./test_allocation

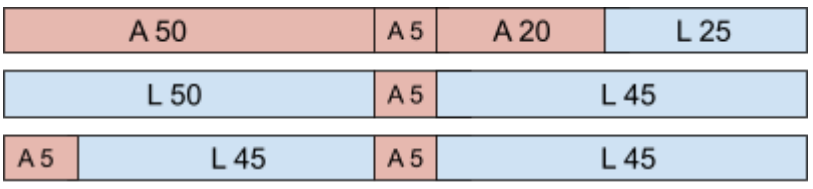
Il est possible de changer la taille de la mémoire et l'alignement dans le Makefile. Pour les tests la mémoire doit être suffisamment grande par exemple 500 000.

6. Performance

Comparaison entre politiques:

Pour donner un exemple de fragmentation forte, on prend une mémoire de taille 100.

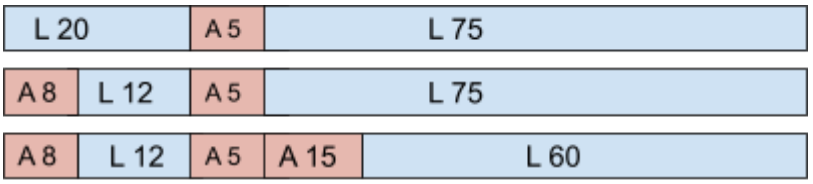
-first fit: serie instructions: a 50,a 5, a 20. puis f bloc 1 et bloc 3
puis a 5. puis maintenant on ne peut plus faire a 50.



First fit

-best fit: instructions: a 20, a 5, f block 1.

Puis a 8 qui va être alloué dans le bloc libre (de taille suffisante) le plus petit.
de même a 15 qui va être alloué dans le bloc libre de taille 75.
Et maintenant on ne pourrait plus allouer une zone de taille 65 par exemple.



Best fit

-worst fit: instructions: a 20, a 5, f block 1.

Puis a 20 qui va être alloué dans le bloc libre le plus grand soit L 75.
Et maintenant on ne pourrait plus allouer une zone de taille 65 par exemple.

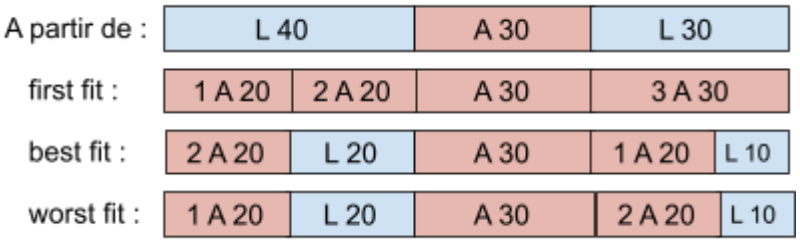
Worst fit

<u>Exemple pour lesquels les stratégies sont meilleurs que les autres:</u>

De manière global le numéro qui précède le A qui représente l'allocation est la n-ième allocation.

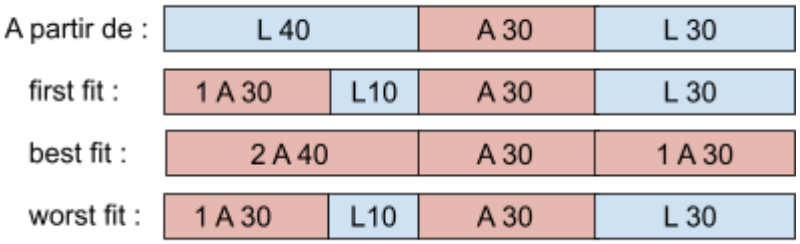
-first fit: 1er allocation de taille 20
2ème allocation de taille 20
3ème allocation de taille 30

pour cet exemple le first fit a pu faire les 3 allocations mais pas les autres stratégies.



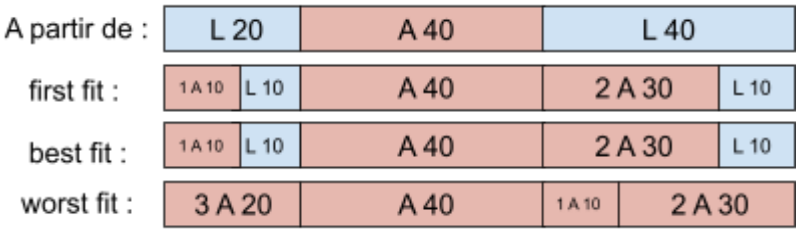
-best fit: 1er allocation de taille 30
2ème allocation de taille 40

Le best fit est plus efficace pour cet exemple



-worst fit:1er allocation de taille 10
2ème allocation de taille 30
3ème allocation de taille 20

Le worst fit est plus efficace pour cet exemple



Coup mémoire des structures de gestion :

On peut remarquer que cela dépend beaucoup de la taille des blocs en mémoire. Plus on a de gros blocs moins le ratio structures/ taille mémoire est élevé. Mais plus on a de petit bloc alors plus on a besoin de structures de contrôle et alors on obtient un ratio plus élevé. Car plus on a de petit bloc alloué plus notre nombre de structures de contrôle est élevé et alors ils occupent beaucoup de place.
Notre entête prend 24 octet et les maillons 16. Sans compter l'arrondissement selon l'alignement. Si on alloue que des blocs de taille inférieure ou égale à 10. alors on obtient tout de même que les ⅔ de l'espace occupé de la mémoire est occupé par des structures de contrôle.
En revanche, si les blocs sont plus grands cela diminue très vite. Par exemple des blocs de taille moyenne de 100. Les structures de contrôle occupent environ 16%. Mais quelque soit la taille des grands blocs, les structures de contrôle occupent toujours de l'espace.

Remarques et observations:
Nous avons testé les performances de chaque stratégie avec des blocs de tailles entre 1 et 64 octets d'un premier coup ensuite avec des blocs de tailles entre 65 et 512, afin d'observer le comportement de chaque stratégie concernant la fragmentation. Avec test_frag.

Globalement on remarque que plus la taille des blocs à alloués est grande, plus on a un taux de fragmentation élevé .

Le plus on a de blocs occupés de base, le moins efficace deviennent les stratégies (first et best fit) par contre worst fit tend vers l'efficacité des deux stratégies

Le moins de blocs occupés de base, le plus efficace devient best fit (un peu moins first fit) puis le moins efficace devient worst fit.

Best fit: très efficace quand on travaille avec des petits blocs, taux faible de fragmentation en moyenne,

Worst fit: Elle devient intéressante quand on travaille avec des blocs avec des grandes tailles -> taux de fragmentation légèrement faible en moyenne.

First fit: Cette stratégie se place au milieu en terme d'efficacité entre les best et worst fit.