M1 Info, Conception des systèmes d'exploitation et programmation concurrente

-

Compte rendu T.P. 2 - Allocateur mémoire

Calvin Massonnet, Mathis Perrier

1. Résumé

Notre allocateur fonctionne sous les trois différentes stratégies (*first fit*, *best fit* et *worst fit*) avec les tests fournis, ainsi que quelques tests supplémentaires effectuant des tests critiques.

2. Principe d'implémentation

Nous avons trois types de zones différents implémentés sous forme de structure qui sert d'en-tête à ces mêmes zones :

1. Un en-tête '*entete*' qui contient un pointeur vers la tête de la liste chaînée des zones libres et un pointeur vers la stratégie d'allocation mémoire utilisée ;
2. Une zone libre '*fb*' qui contient sa taille (taille structure + taille mémoire) et un pointeur vers la prochaine zone libre ;
3. Une zone occupée '*bb*' qui ne contient que sa taille (taille structure + taille mémoire).

Pour l'allocation mémoire, nous avons implémenté trois types de stratégie différentes telles que vues en cours et attendues à l'issue de ce T.P. : '*mem_first_fit()*', '*mem_best_fit()*' et '*mem_worst_fit()*'. Pour la fusion lors de la libération de mémoire allouée, nous sommes restés au plus simple et avons opté pour une boucle qui parcourt la liste chaînée des zones libres et effectue une fusion à droite dès lors que deux zones libres se retrouvent contiguës l'une de l'autre.

Lorsque la taille de l'allocation demandée est trop petite - inférieure à la taille de la structure d'une zone libre - un problème se présente à la libération de cet espace mémoire. Nous avons donc fait le choix d'allouer au minimum une taille d'espace mémoire égale à la taille de la structure des zones libres. Et inversement, lorsque la taille d'allocation demandée est trop grande - supérieure à la taille du tableau, en-tête exclue - nous renvoyons *NULL* à la place d'une adresse.

Lors de l'allocation d'une zone, l'allocateur vérifie si la zone restante de la zone libre ait une taille suffisante pour contenir au minimum la taille de la structure des zones libres. Si non, l'allocateur attribue cette zone à la zone allouée.

3. Structure de code

L'entièreté du code écrit durant ce T.P. peut être trouvée dans le fichier '*mem.c*', mise à part les tests élaborés qui se trouvent dans '*test_critique.c*', ainsi que quelques lignes dans '*mem.h*' et '*memshell.c*' permettant d'ajouter une extension au *memshell* afin d'avoir une visualisation différente des zones libres (commande *j*).
Les portions de code effectuant une tâche précise ont été écrites dans des fonctions et procédures identifiables par leur nom. De ce fait, une méthode '*mem_free_fusion()*' a été créée.

4. Tests effectués

Les tests fournis avec le T.P. terminent tous les quatre sur un résultat positif, sans erreur, peu importe la stratégie d'allocation utilisée.
De plus, nous fournissons un test supplémentaire, *test_critique*, permettant de vérifier certaines allocations et libérations que l'on pourrait trouver d'absurde telles que l'allocation et la libération de zone hors du tableau d'allocation (comptant l'en-tête).

Le code fourni avec ce rapport n'a été exécuté que sur des machines bénéficiant d'un système d'exploitation UNIX 64 bits.

5. Compilation et exécution

Pour compiler le code rendu sous une plateforme Linux, il suffit d'ouvrir une console, de se placer dans le répertoire du code source dans lequel le *Makefile* est situé et de lancer la commande :

make

Une fois le code compilé, il sera possible de lancer le *memshell*, ainsi que différents tests parmi lesquels les tests donnés ainsi que le *test_critique* pourront être trouvés en prétextant l'exécutable voulu d'un point suivi d'un slash avant :

./memshell
./test_init
./test_base
./test_cheese
./test_fusion
./test_critique