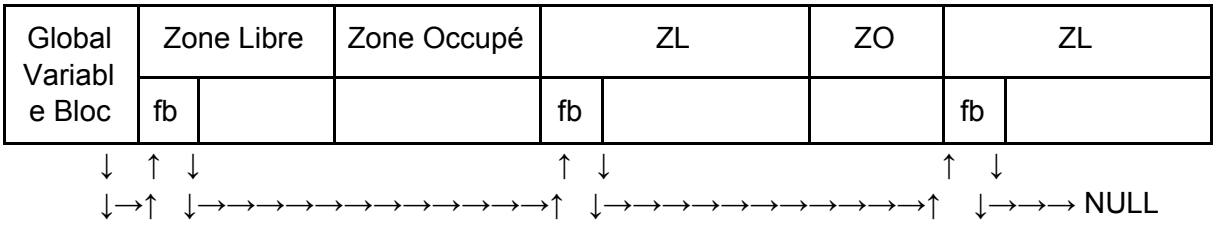
Blanquet Antoine - Portier Barnabé

Compte Rendu API - CPS

TP4 : Compression avec algorithme de Huffman

Interface de l'allocateur

La mémoire peut être représenté comme ci-dessous : une combinaison de zones libres et zones occupés.



Un bloc (ou "une zone") est représenté par sa taille, et les blocs libres sont associés entre eux avec un lien de chaînage vers la zone libre suivante. Nous avons rajouté un **header** (**G**lobal **V**ariable **B**loc) qui contient la première zone libre de la mémoire ainsi que sa taille globale.

Initialisation (*mem_init*)

Pour l'initialisation, nous mettons à jour notre **header** avec la taille de mémoire disponible et un accesseur zone mémoire libre. Zone libre qui a pour taille toute la mémoire disponible et qui n'a pas pas de zone libre suivante : la valeur "next" du bloc est égale à NULL.
On initialise également la politique appliqué dans le code pour la recherche de la zone libre à allouer. Dans notre cas, nous avons implémenter la politique "*first fit*" (première zone libre de taille suffisante disponible)

Affichage (*mem_show*)

Afin de pouvoir tester notre programme au fur et à mesures, nous devons pouvoir afficher la mémoire à un instant t avec les zones occupés et libres. Cette fonction réalise un parcours de la mémoire, et pas uniquement sur les zones libres ! Chaque bloc contenant sa taille, il plutôt simple de parcourir chaque bloc.
Cependant, pour faire la différence entre une zone libre ou occupé, il nous faut également notre **header** qui contient la liste des zones libres. On vérifie comme cela si l'adresse du bloc est celle du bloc libre courant et l'on peut distinguer un bloc libre d'un bloc occupé.

Fonctions auxiliaires utiles pour l'allocation (*align, alignZoneLibre*)

align : Accéder à une donnée mal alignée peut faire ralentir un programme considérablement, ou causer des bugs. De ce fait la fonction align prend en compte la donnée demandé par l'utilisateur. Si celle est n'est pas un multiple de 8 (L'alignement étant égal à 8 dans notre code), alors sa nouvelle taille est égale à 8 * (quotient(taille,8)) + 1.
Exemple : 17 ---> quotient **q** de 17/8 = 2. La fonction align renverra : 8 * **q** + 1 = 24.

alignZoneLibre : Si la mémoire “libre” restante après une allocation n’est pas suffisante pour y placer une zone libre supplémentaire, on considère alors cette espace comme occupé afin de mieux pouvoir libérer cette mémoire par la suite.

Allocation de mémoire (*mem_alloc*)

Cette fonction permet de stocker une certaine taille demandé par l’utilisateur dans une des zones libres de la mémoire si celles ci peuvent contenir la taille.
La taille en mémoire comprends 8 octets supplémentaire pour pouvoir inscrire la taille du bloc occupé, et subit également un ajustement d’alignement grâce à la méthode ***align***.

Première idée :

Sur un schéma classique, nous avons déterminé 2 cas possibles :

- La zone libre va complètement devenir occupé
- une partie de la zone libre va être occupé et le reste va rester libre.

Dans le second cas, nous avons pensé à placer la zone occupé au début de la zone vide et déplacer la zone libre à la suite de la zone occupé. Cela était complexe à mettre en place et nous avions jamais réussi à implémenter cette solution.

De ce postulat nous nous sommes rendu comptes de plusieurs facteurs d’échecs :

- L’importance d’avoir un code commenté : en particulier pour ce projet
- Le choix des noms de variables
- L’importance des schémas/dessins.

Deuxième idée (implémentée) :

Avec l’aide de Théo Teyssier, nous avons opté sur une manière différente de voir l’allocation mémoire dans le cas où la zone libre n’était pas complètement alloué. En effet, il était plus facile d'enregistrer la zone occupée à la fin de la zone libre, et ainsi ne pas modifier l’entête de la zone libre contenant sa taille et son lien de chaînage vers une autre zone libre (en mettant bien évidemment la taille du bloc libre à jour).

Dans le cas de la première zone libre qui devient complètement occupé : il faut mettre à jour le champ “next” du **header**. Également, sii il n’y pas assez d'espace libre pour une zone libre supplémentaire, on fait appel à ***alignZoneLibre***.

Libération de mémoire (*mem_free*) - ***non fonctionnel à 100%***

Pour libérer la mémoire nous avons suivi les étapes suivantes :

- 1er cas : on libère une zone libre qui est situé avant la première zone libre actuellement connu (si il y en a une).

Si c’est le cas, on change le champ “list” du header (gvb), le nouveau bloc libéré aura pour champ “next” la prochaine zone libre (NULL si non existant).

- 2ème cas : on libère un bloc, et la zone qui le précède est également une zone libre.

Dans ce cas : on change juste la taille de la zone libre qui le précède (cela évite de créer une struct fb en plus en mémoire)

- Autres cas : on libère le bloc occupé. On lui associe un struct fb en le chaînant à une zone libre suivante.

Enfin pour finir, quelques soit le cas, à chaque création de bloc, si celui ci est suivi d'une zone libre, on fusionne les blocs.

Résultats :

Nous avons pas réussi à faire en sorte que le free fonctionne correctement. Le seul cas fonctionnant pour le moment le numéro 2, celui où on agrandit la taille de la zone précédente. Les autres cas donnent lieux à une Erreur de segmentation.
PS : pour libérer la mémoire, il faut bien taper “**M**” et utiliser l’adresse inscrite sur la console. Il ne faut pas prêter attention au message “Adresse libéré en ..” car celui ci est faux (tout du moins pour l’utilisateur).

Extensions

Nous avons pas eu le temps d’implémenter les extensions.