Compte-rendu : Allocateur mémoire

Approche de la problématique:

L'allocateur mémoire est un outils dont le rôle est de pourvoir plusieurs fonctions permettant de gérer la mémoire. L'enjeu de le manipulation de la mémoire est avant tout de faire en sorte qu'aucun conflit ne survienne lors de l'utilisation des cases mémoire et de conserver les informations sur les adresses utilisées. Ainsi nous avons entâmé la structuration de la mémoire sous forme de blocs tels qu'un bloc définisse un ensemble continu de cases mémoire et qu'il connaisse l'adresse du bloc suivant. Un bloc connaît également sa propre taille. L'espace mémoire de travail est précédé d'un bloc de tête invariable qui servira de base à la chaîne de blocs. Ce dernier connaît la taille de l'espace de travail.

Implémentation et choix:

Structure:

Afin de représenter cette architecture de blocs en C nous avons utilisé des structures, une pour les blocs et une pour le bloc de tête. Les blocs possèdent leur taille sous forme de *size_t* (un *long unsigned bit*) dans les premières cases mémoire qu'ils gèrent. Un bloc est avant tout libre, ce qui signifie que lorsqu'il devient occupé il disparaît en tant que structure, c'est à dire qu'il est enlevé de la chaîne partant de la tête. En revanche la valeur contenue dans ses premières cases est conservée et n'est pas modifiée ou modifiable par l'utilisateur selon une utilisation normale. De la sorte cette valeur peut servir, notamment lorsqu'il s'agit de libérer une zone mémoire désignée par un simple pointeur *void**.

Allocation:

L'allocation va dans un premier temps avoir le rôle de corriger la taille de la zone mémoire demandée, selon différents critères. Tout d'abord nous avons décidé que toute taille serait un multiple de 8, pour conserver la cohérence de l'octal. Par conséquent tout espace de mémoire demandé sera arrondi à la taille multiple de 8 supérieure. Ensuite, pour ne pas perdre d'espace entre les blocs nous avons choisit d'absorber les plages de mémoire séparant la fin de celle demandée et le début de la suivante occupée, dont la taille était inférieure à 8. L'écriture de la taille d'un bloc (*size_t*) se faisant sur un octet, nous n'aurions jamais pu écrire sur ce morceau de mémoire (nous rappellons que même les parties occupées possèdent leur taille à écrire dans la mémoire).

Ensuite nous appellons la méthode de séléction de mémoire qui nous renvoit une zone utilisable. A partir de celle-ci nous découpons deux sous-parties telles que la première propose la taille que l'utilisateur a demandé arrondie et la deuxième étant le reste du bloc libre initial, qui reste libre. Si la deuxième partie est nulle, c'est à dire si tout le bloc est réquisitionné, on ne créé pas ce deuxième bloc.

Enfin nous décidons de reformer la chaîne des blocs dans l'ordre de la mémoire afin d'éviter de possibles problèmes dans la libération. Pour cela, si il existe un reste de bloc libre nous ammenons le pointeur du bloc précédent sur lui puis lui donnons le pointeur du bloc libre initial. Si il n'existe pas, le bloc précédent prend directement le bloc suivant du bloc initial. Pour trouver le bloc précédent il nous faut parcourir la chaîne depuis la tête. En fin de programme nous renvoyons un pointeur vers la zone mémoire

située après la taille du bloc. Celle-ci n'est donc pas le bloc mais bien une zone utilisable destinée à l'utilisateur. Les valeurs affichées sont toujours celles que l'utilisateur doit voir et non celles du développeur qui manipule les blocs.

Libération

Dans un premier temps nous récupérons les données laissées dans la taille du bloc, c'est à dire l'octet qui précéde la zone mémoire donnée. A partir de là nous allons chercher à reconstruire la chaîne dans l'ordre de la mémoire encore une fois. A partir des données du bloc précédent et suivant nous pouvons vérifier si le bloc précédent et/ou le bloc suivant est/sont adjacents de sorte à les fusionner.

Choix de présentation

Quand un espace mémoire de travail est initialisé, il reçoit avant tout le bloc de tête dont la structure prend trois octets. (Ici nous aurions pu afficher les adresses décalées de 24 pour l'utilisateur, et donc commencer à 0) Ensuitel le bloc libre devant exister, un espace d'un octet est toujours réservé à la taille, et donc la première adresse vue par l'utilisateur est 32, pour les quatres octets déjà pris. A chaque allocation un octet est "perdu de vue" par l'utilisateur pour cette raison, sauf dans le cas où il demande l'espace d'un bloc entier. Par ailleurs si l'utilisateur fait la somme d'une adresse (début de la plage) et de sa taille, il constatera qu'un octet manque pour arriver à l'adresse suivante. Les tailles affichées correspondent bien à l'espace effectivement utilisable et non à la taille d'un bloc.

Problèmes connus:

Une probable erreur de cast dans l'allocation créé un décalage de 120 bits sur les adresses des pointeurs reçus pour la libération.

Le programme ne supporte pas de recevoir des pointeurs sur des emplacements mémoire invalides. S'il reçoit un pointeur qui n'est pas un de ceux fournis par l'allocation alors cela produit inévitablement des erreurs de segmentations. Nous n'avons trouvé aucune manière de vérifier une adresse avant de la traiter et créer l'erreur. Nous pensons que le seul moyen serait de conserver en mémoire tous les pointeurs en cours d'utilisation et de vérifier si celui qui est fourni est parmi eux, mais cette solution aurait été lourde pour notre exercice.