Valisp séance 1 : Allocateur mémoire

L2 Informatique – Université Côte d'Azur

On suppose que vous êtes venu au CM.

Rappel : la mémoire sera vue comme une succession de blocs mémoires. Un bloc correspond à une liste de cases en mémoire qui peuvent être libres ou occupées. Chaque bloc commence par un mot de 32 bits de type `bloc_description` contenant quatre informations :

1. La marque pour le ramasse-miette (sur 1 bit)
2. l'indice du bloc précedent (sur 15 bits)
3. La disponibilité du bloc (sur 1 bit)
4. l'indice du bloc suivant (sur 15 bits)

Par conventions, le bloc précédant du premier bloc est le bloc précédant lui-même et le bloc suivant du dernier bloc est là aussi lui-même (c'est-dire le dernier bloc). Par convention le dernier bloc (qui sera toujours vide) sera toujours marqué comme non disponible.

Exercice 1 — Préambule

1. Créez un répertoire `valisp` et créer y les fichiers `main.c` et `Makefile`. Écrivez un *Hello world* puis vérifier que la compilation fonctionne.
2. Créez maintenant les deux fichiers utiles pour ce TP `allocateur.c`, `allocateur.h`. Dans le fichier *header* ajouter la déclaration des 6 fonctions marquées par le cadre `.h` (on fera attention à bien mettre les directives `#ifndef ALLOCATEUR_H`).
3. Dans le fichier `allocateur.c` commencez par créer un type `bloc` correspondant au type `uint32_t`.
4. On fera appelle aux tests dans la fonction `main`. Libre à vous d'ajouter de nouvelles fonctions et de les exporter via le *header* pour les utiliser dans le fichier principal `main.c`.

Exercice 2 — Mise en place de la mémoire

Au commencement, la mémoire ne contient que deux blocs situés dans la première et dernière case de notre mémoire. Toutes les autres cases représentent les cases libres de la mémoire que l'on pourra allouer plus tard.

Les adresses étant codés sur 15 bits, notre tableau contient $N = 2^{15}$ cases.

0	0	premier	0	dernier
1	…	…	…	…
…	…			
N-1	…	…	…	…
N	0	premier	1	dernier

⇔

0	000000000000000	0	111111111111111
…	…	…	…
…	…		
…	…	…	…
0	000000000000000	1	111111111111111

1. Créer une macro pour définir la constante `TAILLE_MEMOIRE_DYNAMIQUE` à 2^{15}
2. Creéz de manière globale le tableau `MEMOIRE_DYNAMIQUE` contenant `TAILLE_MEMOIRE_DYNAMIQUE` cases.
3. `.h` Écrivez une procédure `void initialiser_memoire_dynamique()` qui initialise les première et dernière cases de la mémoire avec les bonnes valeurs.

Exercice 3 — Création des blocs

On rappelle que chaque bloc est sur 32 bits de la forme | b0 | precedant | b1 | suivant |

1. `bloc cons_bloc(int rm, int precedant, int libre, int suivant)`. Cette fonction construit un entier de 32 bits en se basant sur quatre entiers de tailles respectives 1 bit, 15 bits, 1 bit et 15 bits.
2. Écrivez maintenant les quatres fonctions inverses permettant de décomposer un bloc. En paramètre, les fonctions utilisent l'indice d'un bloc dans la mémoire dynamique créée globalement dans l'exercice précédant.
   (a) `int bloc_suivant(int i)`
   (b) `int bloc_precedant(int i)`
   (c) `int usage_bloc(int i)`
   (d) `int rm_bloc(int i)`
3. Écrivez une fonction `int taille_bloc(int i)` qui calcule l'espace disponible dont le bloc d'indice `i` est responsable. L'unité sera le nombre de cases dans le tableau.
4. Nous allons pouvoir commencer à tester votre code. Télécharger le fichier `etudiant.c`. Copier la première partie dans votre fichier `allocateur.c` et vérifier que tout compile. Copier maintenant les testes dans votre fichier `main.c` au fur et mesure que les fonctions seront écrites.

Exercice 4 — La fonction `malloc`

Il est temps maintenant d'implémenter les deux fonctions d'allocation et de libération de la mémoire. On remarquera que nos deux fonctions ont bien la même signature que les fonctions d'origine. Dans cet exercice, commençons par `malloc`.

1. Le `malloc` utilise un paramètre `size` en octets. Or notre tableau est organiser en case. Écrivez une fonction `int octets_vers_blocs(size_t size)` qui renvoie le nombre de cases minimale pour allouer `size` octets.
2. Écrivez la fonction `int rechercher_bloc_libre(int taille)` qui parcourt la liste doublement chaînée de la mémoire et trouve un espace libre de taille suffisante pour contenir un élement de nécéssitant `taille` cases. On renverra -1 si la mémoire ne contient aucun espace suffisamment grand et l'indice du bloc disponible sinon.
3. `.h` En déduire une fonction `void *allocateur_malloc(size_t size)` qui renvoie un pointeur (et non un indice) vers la zone de mémoire trouvée par la fonction précédente et `NULL` s'il n'y a pas assez d'espace disponible.

Félicitation, vous avez atteint l'objectif de ce TP. l'implémentation de `free` est facultative. En effet, la mémoire sera libérée automatiquement sans qu'il soit nécessaire d'y faire appel.

Exercice 5 — La fonction `free`

1. Écrivez une fonction `void allocateur_free_bloc (int i)` qui libère le bloc associé. On fera attention de fusionner si nécessaire le bloc avec les blocs libres adjacents.
2. `.h` Écrivez maintenant `void allocateur_free (void *ptr)` qui libère le bloc correspondant au pointeur fourni en paramètre.