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 case 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.

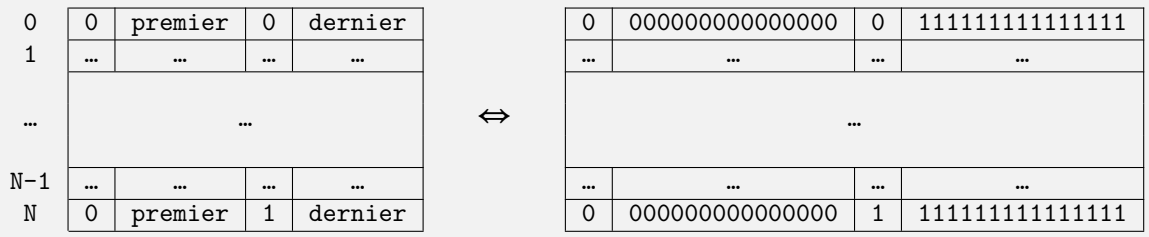
Exercice 1 — Préambule

1. Créer un répertoire `valisp` et créer y les fichiers `main.c` et `Makefile`. Écrire un *Hello world* puis vérifier que la compilation fonctionne.
2. Créer 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.



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` Écrire 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 contruit un entier de 32 bits en se basant sur quatre entiers de tailles respectives 1 bit, 15 bits, 1 bit et 15 bits.
2. Écrire 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. Écrire 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. Écrire une première fonction `int rechercher_bloc_libre(size_t size)` qui parcourt la liste doublement chaînée de la mémoire et trouve un espace libre de taille suffisante pour contenir un élement de taille `size`. On renverra -1 si la mémoire ne contient aucun espace suffisamment grand et l'indice du bloc disponible sinon. On fera attention au fait que la fonction `taille_bloc` renvoie un nombre de case et que notre fonction prend en paramètre une taille en octets.
2. `.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. Les deux parties suivantes sont soit facultatives, soit sera nécessaire plus tard dans le projet (mais il est bon de prendre de l'avance).

Exercice 5 — Une API pour le ramasse-miette

1. Pour utiliser notre ramasse-miette, nous aurons à indiquer à notre allocateur quels sont les blocs qu'il faudra préserver. Pour cela, écrire les deux fonctions qui permettent la lecture et l'écriture du bit du ramasse-miette. Attention, les paramètres sont des pointeurs et non les indices du bloc.
   (a) `.h` `int ramasse_miette_lire_marque(void * ptr)`
   (b) `.h` `int ramasse_miette_poser_marque(void * ptr)`
2. Écrire une fonction `int bloc_libre(i)` qui renvoie un booléen indiquant si le bloc d'indice `i` doit être supprimée. Attention, le dernier bloc (celui qui est son propre successeur) ne doit jamais être supprimé.
3. `.h` Écrire enfin la fonction `void ramasse_miette_liberer()` qui parcours les blocs mémoires et libère tout les blocs supprimables. On fera attention de fusionner les blocs libres adjacents (cf. CM).

Exercice 6 — La fonction `free`

Comme indiqué au dessus, l'implémentation de `free` est facultative. En effet, la mémoire sera libérée automatiquement sans qu'il soit nécessaire d'y faire appel.

1. Écrire 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` Écrire maintenant `void allocateur_free (void * ptr)` qui libère le bloc correspondant au pointeur fourni en paramètre.