Système d'Exploitation - L3, S6
TP 1 - Ramasse Miettes

Téléchargez et désarchivez le fichier **TP1-memoire.zip** dans l'onglet TP du cours en ligne.

Exercice 1.1 *Échauffement*

1. Lisez le module **memoire.c** et expliquez en quoi il permet de pister les fuites mémoire.
2. Permet-t-il de réparer les fuites mémoire ?

Exercice 1.2 *Ramasse-miettes GC*
Le ramasse-miettes GC permet de détecter les fuites de mémoire et aussi de les collecter pour les libérer.

1. En vous documentant sur ce site (ou en regardant le fichier README), expliquez comment utiliser le ramasse-miettes GC pour détecter les fuites mémoire, mais aussi comment l'utiliser comme ramasse-miettes.
2. En utilisant GC, modifiez le code source du module **list** pour détecter les fuites mémoire dans le fichier **test-list.c**.
3. Corrigez les fuites mémoire détectées d'une part sans utiliser le ramasse-miettes et d'autre part en l'utilisant. Comparez les deux solutions. Laquelle des deux solutions (avec ou sans ramasse-miettes) est plus robuste ?
4. Comparez les deux implémentations incrémentale et générationnelle de ramasse-miettes proposées dans GC

Exercice 1.3 *Implémentation ramasse-miettes*
Dans cet exercice l'objectif c'est d'implémenter des ramasses-miettes. On va supposer que la mémoire est divisée en mots de même taille t (en octets) et qu'il y en a b (numéroté de 1 à b). Allouer de la mémoire de taille l, c'est fournir $\lceil \frac{l}{t} \rceil$ mots consécutifs. Pour gérer la mémoire, on va la diviser en blocs, chaque bloc étant représenté par une adresse de début (un mot de taille t), une taille (en mots) et un drapeau indiquant si le bloc est utilisé ou pas. Pour la gestion, les blocs sont chaînés dans une liste.

1. Ecrivez la fonction `allocationMemoire` qui prend en paramètre la liste de blocs et une taille (en octets) à allouer, qui alloue de la mémoire de taille demandée et qui retourne l'adresse de début du bloc alloué. Il faudra gérer les cas où il n'y a pas de plage mémoire consécutive de taille demandée.
2. Ecrivez la fonction `liberationMemoire` qui prend en paramètre la liste de blocs et une adresse de début d'un bloc alloué et qui libère la mémoire allouée. Il faudra gérer les cas où l'adresse donnée n'est pas celle d'un bloc alloué. Pensez à regrouper les blocs libres.

3. Ecrivez une fonction `associeBlocs` qui prend en paramètre la liste des blocs, une adresse source (représentant un bloc de taille 1) et une adresse destination (représentant celle d'un bloc alloué) et qui permet de signifier que l'adresse source référence le bloc représenté par l'adresse destination. Vous aurez peut-être besoin de modifier la structure bloc.
4. L'objectif maintenant c'est d'implémenter des ramasse-miettes. Le module sera nommé ramasse-miettes et dans le header on aura deux fonctions, l'une pour la technique du compteur de référence et l'autre pour un algorithme traversant.
   (a) Implémentez un compteur de références des zones de mémoire allouées, qui ne réalise la suppression que lorsque toutes les références à un bloc sont supprimées. Vous aurez peut-être besoin de modifier la structure bloc.
   (b) Testez votre implémentation. Vous pouvez par exemple écrire une fonction `initialiseMemoireAleatoirement` dans un fichier **test-gc.c** qui initialise de la mémoire aléatoirement, en appelant 10 fois la fonction `allocationMemoire` sur des tailles aléatoires, et en faisant 10 appels à `associeBlocs` pour des adresses destinations aléatoirement choisies parmi les 10 créées (vous ferez attention de bien choisir les adresses source).
   (c) Implémentez l'algorithme *mark and sweep* de Dikjstra. Refaites le test de 4(b) avec 100 et exécuter à des endroits différents du code de test ce ramasse-miettes. Que remarquez-vous ?