SORBONNE UNIVERSITÉ

TD : Garbage Collectors — semaine 4

1 mars 2018

Objectif(s)

✯ Étude de différents algorithmes de *garbage collection*.

Exercice 1 – Simulation d'algorithmes de GC

Considérons le programme suivant :

pseudo C	pseudo ML
	`type ('a,'b) cell=`
`x=make2(NULL,NULL);`	`{mutable tete : 'a; mutable queue 'b};;`
`y=make2(x,NULL);`	`let x={tete=None;queue=None};;`
`z=make2(x,NULL);`	`let y={tete=Some x;queue=None};;`
`x=make2(y,z);`	`let z={tete=Some x;queue=None};;`
`y[1]=x;`	`let x={tete=Some x;queue=Some z};;`
`y=NULL;`	`y.queue<-Some x;;`
`x=z;`	`let y=None;;`
`z=make2(x,NULL);`	`let x=z;;`
	`let z={tete=some x; queue=None};;`

On suppose que l'ensemble des racines est composé de toutes les variables globales du programme.

1. Indiquer l'état de la mémoire à la fin de son exécution.
2. Simuler un algorithme de compteur par références sur cette mémoire.
3. Simuler les trois algorithmes explorateurs suivants, en déclenchant le GC à la fin de l'execution du programme plus haut :
   — Mark & Sweep
   — Mark & Compact
   — Stop & Copy

Exercice 2 – Crible d'Eratosthène

Dans cet exercice, on cherche à évaluer l'impact des `malloc` et `free` sur les performances d'un programme. On écrira ensuite une gestion de mémoire dédiée pour l'application.
On travaillera avec le type `liste_entier` suivant :

```
struct cons {int car; struct cons *cdr;};
typedef struct cons *liste_entier;
```

1. Écrire une fonction `cons` qui ajoute un entier à une liste et retourne la nouvelle liste, on utilisera `malloc` ici pour faire cela.
2. Écrire une fonction intervalle qui prend deux entiers a et b comme paramètres et construit la liste de tous les entiers entre a et b.
3. Écrire une fonction qui calcule la liste des nombres premiers inférieures à un entier n donné. On utilisera pour cela le crible d'Eratosthène. Écrire une version avec récupération mémoire (avec `free`) en utilisant les fonctions précédentes et dans un style de programmation fonctionnelle.
4. Que dire de l'occupation mémoire après l'appel à `eratosthene(1000)` avec la récupération mémoire et sans ?

Exercice 3 – Allocation optimisée

1. Écrire une fonction `init_alloc` qui alloue en une fois une zone mémoire pouvant contenir n cons. Que faut t'il faire sur cette zone pour pouvoir ensuite l'utiliser par un malloc utilisateur.
2. Écrire une fonction `mon_malloc` qui utilise la zone déclarée par init_alloc.
3. Écrire une fonction `mon_free` qui libère une cellule de cette zone.

Exercice 4 – Récupération automatique de `liste_entier`

1. Écrire une fonction `init_GC` version modifiée de `init_alloc` qui conserve les informations de début et de fin du tas, ainsi que les informations sur la zone de la pile. On utilisera, pour déterminer la zone statique, les symboles globaux `end` et `edata`. On calculera l'adresse de début de pile et son sens en faisant un calcul sur les adresses des variables locales.
2. Écrire une fonction qui détermine si une valeur (un pointeur) est un pointeur sur une liste d'entiers.
3. Écrire une fonction `mark` qui prend un pointeur valide en entrée et le marque dans le tas ($|1$) sur le champs `cdr` et marque ses descendants s'il y en a.
4. Écrire une fonction `sweep` qui explore le tas et reconstruit la liste des cellules libres en enlevant les marques des cellules à conserver.
5. Écrire une fonction `gc` qui récupère automatiquement l'espace disponible dans le tas.
6. Utiliser les fonctions précédentes pour modifier la fonction `mon_malloc` qui déclenche le GC s'il n'y a plus de place disponible.
7. Que se passe t'il s'il n'y a toujours pas de place libre ?
8. Que se passe t'il si un entier correspondant à une valeur de pointeur valide se trouve dans la pile ?
9. Que se passe t'il si une liste d'entiers apparait comme champ d'une valeur struct ?