

Chapitre 10

Gestion de la mémoire

Sommaire

10.1 Allocation et désallocation	1
10.2 GC : Structure du tas, racines	2
10.3 GC : algorithmes	2
10.3.1 Comptage de références	3
10.3.2 Mark and sweep	3
10.3.3 Stop and copy	3
10.3.4 Variantes	4
10.4 Implémentations	4
10.4.1 Java	4
10.4.2 OCaml	5
10.4.3 Et CTigre?	5

Ce cours est en version préliminaire. Merci à François Pottier et Emmanuel Chailloux pour leur aide.

10.1 Allocation et désallocation

`malloc` et `free`

L'appel système `sbrk` permet d'augmenter l'espace alloué sur le tas.

Cependant un `malloc` qui fait `sbrk` à chaque fois n'est pas viable car l'espace ne fait que croître.

Il faut donc :

- désallouer les structures de données qui ne sont plus utilisées (fonction `free`);
- et réutiliser l'espace libéré.

`malloc` et `free` doivent donc maintenir une table de l'espace mémoire utilisé. L'algorithme d'allocation doit chercher à optimiser l'occupation mémoire pour éviter la fragmentation.

Allocation

En C, l'appel à `malloc` est explicite.

En Java, CTigre ou OCaml, l'appel à la fonction `malloc` est inséré par le compilateur lors de la création d'un bloc sur le tas.

Désallocation manuelle

En C, l'appel à `free` est explicite.

Mais ce mécanisme n'est pas sûr et en plus fastidieux.

- Il faut implémenter soi-même un parcours de la structure de données pour désallouer récursivement tous les fils (par exemple un arbre ou un graphe) ;
- le risque d'oublier de désallouer est grand, ce qui cause des *fuites de mémoire* (*memory leaks*) ;
- il existe aussi un risque de désallouer un bloc encore vivant ;
- et un risque de désallouer deux fois le même bloc...

Ces erreurs sont très difficiles à détecter et corriger.

Désallocation automatique

Java ou OCaml ont un mécanisme de désallocation implicite (automatique) appelé *Garbage Collector* (GC, ou ramasse-miettes, ou glaneur de cellules...).

10.2 GC : Structure du tas, racines

Principe

On considère le tas comme un graphe, dont les sommets sont les blocs (objets, records, tableaux...) et les arêtes sont les pointeurs.

Les blocs auxquels le programme peut accéder directement (parce qu'il possède leur adresse) sont les *racines* du graphe.

Tout objet qui n'est pas accessible depuis une racine en parcourant le graphe est mort (garbage) et peut être désalloué.

De temps en temps (par exemple lors d'une allocation, quand le tas est plein), on examine le graphe pour déterminer quels objets sont accessibles. On désalloue les autres.

Approximation

Attention : certains blocs accessibles peuvent tout de même être inutiles... Ils ne seront pas désalloués. La présence d'un GC ne garantit donc pas l'absence de fuites de mémoire. Il faut quand même être vigilant lors de l'écriture de programme qui doivent tourner longtemps (serveurs notamment).

10.3 GC : algorithmes

Plusieurs algorithmes

Nous allons voir plusieurs algorithmes de GC :

- Comptage de références ;
- Mark and sweep ;

- Stop and copy

10.3.1 Comptage de références

Comptage de références

Une première idée (1960) (mauvaise) consiste à stocker dans chaque bloc un compteur du nombre de pointeurs vers ce bloc. Lorsque ce compteur devient nul, le bloc est désalloué.

Problèmes :

- Des structures de données cycliques peuvent être inaccessibles sans avoir de compteur de référence nul ;
- la mise à jour des compteur a un coût prohibitif.

10.3.2 Mark and sweep

2 étapes :

1. On parcourt le graphe depuis les racines en marquant les sommets accessibles ;
2. puis on *balaie* le tas linéairement en désallouant tous les blocs qui n'ont pas été marqués et enlève la marque pour les autres.

Pour faire le parcours du graphe (par exemple en profondeur) on a envie d'utiliser une pile (donc occupation supplémentaire de mémoire).

L'algorithme de Deutsch, Schorr et Waite permet de faire le parcours sans utiliser de pile. Idée : inverser les arêtes sur le chemin parcouru pour pouvoir retourner sur ses pas.

10.3.3 Stop and copy

Principe

Recopier la partie du graphe accessible dans une autre partie du tas.

Après recopie, les blocs seront contigus, donc aucune fragmentation.

Le coût est proportionnel à la taille du graphe accessible.

Le tas est donc divisé en deux zones. L'une des deux n'est pas utilisée jusqu'à la copie (perte de place). Ensuite, on intervertit le rôle des deux zones.

« Forwarding »

Problème : lorsqu'un bloc est copié, son adresse change. Il faut donc modifier tous les pointeurs vers lui. Il faut donc mémoriser la correspondance entre l'ancienne et la nouvelle adresse. Mais où ?

Idée : une fois que le bloc est recopié, on écrit la nouvelle adresse dans le premier mot de l'ancien emplacement. Ce champ est appelé « forwarding pointer ».

10.3.4 Variantes

GC à génération

Les blocs créés récemment ont une espérance de vie plus courte. On va donc se concentrer sur les objets jeunes.

On divise le tas en plusieurs zones (générations) qui correspondent à l'âge des blocs. Par exemple 2 générations (vieux et jeunes).

On ne travaille que sur la zone la plus jeune par défaut (mark and sweep ou stop and copy). Les blocs qui survivent changent de génération.

Si la zone plus ancienne est pleine, on lui applique une passe de GC.

Lors du GC sur la jeune génération, il faut tenir compte des racines de cette zone qui sont dans les générations plus anciennes.

Dans un langage purement fonctionnel, il n'y en a jamais.

En pratique ces pointeurs sont rares. On rajoute un test à chaque fois que l'on *modifie* un champ d'une génération ancienne. (voir Appel pour plus de détails).

GC incrémental ou concurrent

Pour certaines applications, il n'est pas tolérable d'interrompre le programme pendant un ramassage complet.

Solutions :

- Ramassage incrémental : comme précédemment mais par tranches incomplètes ;
- GC concurrent, qui travaille en tâche de fond.

10.4 Implémentations

10.4.1 Java

Le GC de Java

Java (au moins dans la version 1.3 du compilateur de Sun) comme .net utilisent une variante de mark & sweep appelée mark & compact. Lors de la phase de balayage, le tas est défragmenté vers le bas en recopiant les objets de manière à ne pas laisser de trou.

Il utilise 2 générations.

```
System.GC.Collect()
```

```
public class ObjetFinalise{
    public ObjetFinalise(){
        // Constructeur, invoqué par l'utilisateur par l'opérateur "new"
    }
    protected finalize() throws Throwable{
        // Méthode de finalisation, invoquée par le Garbage Collector
    }
}
```

Un ensemble de racines est maintenu à jour (quelque part dans le tas).

On peut connaître le type de chaque objet à l'exécution. L'information sur le contenu de chaque classe est gardé en mémoire pendant l'exécution.

10.4.2 OCaml

Le GC de OCaml

à 2 générations (ancienne et nouvelle).

- Stop & Copy sur la nouvelle (GC mineur)
- Mark & Sweep incrémental sur l'ancienne génération (GC majeur)

Caractéristiques :

- Un objet jeune qui survit à 1 GC change de zone.
- Conservation des pointeurs de zone ancienne vers zone jeune
- Si le Mark & Sweep échoue, alors un Stop & Copy est déclenché pour compacter le tas!

Le polymorphisme paramétrique impose une représentation uniforme des données. Chaque valeur est stockée sur exactement un mot, qui est soit une *valeur immédiate*, soit un pointeur sur le tas.

Plus aucune information de type à l'exécution.

La distinction entre valeur immédiate et pointeur se fait à l'aide du bit de point faible de chaque mot : les pointeurs sont toujours pairs (sur des architectures au moins 16 bits). Les entiers sont codés sur 31 ou 63 bits.

Il faut donc modifier un peu le code des opérations arithmétiques, mais en contrepartie, on gagne beaucoup d'efficacité et d'espace mémoire pour le GC.

Le module Gc

10.4.3 Et CTigre ?

- On peut utiliser l'astuce de OCaml pour repérer les racines (entier sur 31 bits). La représentation des données est uniforme, ce qui laisse l'espoir de faire du polymorphisme paramétrique.
- On peut aussi garder à l'exécution l'information sur le contenu de chaque bloc d'activation (racines ou valeurs immédiates) et de chaque structure de donnée. Par exemple un pointeur au début du bloc vers cette information.