Gestion dynamique de la mémoire

Martin Odersky

version 1.2

Plan du cours

Mémoire dynamique

La plupart des langages allouent sur le tas les données qui :

- ne sont pas copiées lorsqu'on les réassigne ou passe en paramètre,
- vivent plus longtemps que la fonction qui les a créées.

Exemple

En Vier, une liste peut vivre plus longtemps que la méthode qui l'a créée : on la place donc sur le tas.

Le système d'exécution (*runtime system*) :

- alloue des données sur le tas,
- régénère la mémoire des données qui ne sont plus utilisées.

C'est un ensemble de méthodes (parfois parallèles) qui sont intégrées à chaque programme exécuté.

Allocation et dé-allocation d'objets

L'idée de base est la suivante :

- Le système garde une liste de blocs mémoire inutilisés (*free list*).
- Les données sont alloués dans un bloc suffisamment grand que l'on trouve et supprime dans la liste :
  - par ordre d'adresse, *FIFO*, *LIFO* ou encore aléatoirement,
  - de taille suffisante, minimale, standard minimale, etc.
- Si aucun bloc n'est trouvé, on annule le programme ou on augmente la taille du tas.
- Si l'entier du bloc n'est pas utilisé, on retourne le sous-bloc restant dans la liste.
- La mémoire de données dé-alloué retourne dans la liste.
- On rattache les blocs contigus pour en créer de plus gros.

Problèmes d'allocation et de dé-allocation

Chercher dans la liste est coûteux (en terme de temps).

Gérer l'état de la liste (entrer et sortir les blocs) est coûteux.

La mémoire se fragmente, c'est-à-dire que la taille d'un bloc de données que l'on peut allouer devient inférieure à la taille de la mémoire réellement disponible.

On parle de fragmentation :

interne lorsque le gaspillage est dû à l'utilisation de blocs plus volumineux que les données qu'ils contiennent.

externe lorsque le gaspillage est dû à une organisation spatiale déficiente des blocs occupés ne laissant plus que des petits blocs libres entre eux.

On mesure la fragmentation comme : $\frac{\text{mémoire requise sur le tas}}{\text{taille des données allouées}}$

Le meilleur moyen d'optimiser ce chiffre dépend beaucoup de la répartition des tailles des objets :

- En 1968, D. Knuth, se basant sur un modèle synthétique des besoins d'allocation, avance l'idée que prendre le meilleur bloc n'est pas favorable.
- Des mesures récentes sur de vraies applications montrent que :
  - prendre le meilleur bloc est clairement supérieur.
  - prendre le meilleur bloc en ordre d'adresse ou FIFO fragmente le moins.
  - c'est aussi la technique de recherche dans la liste la plus lente.

Un tas fragmenté peut être compacté en déplaçant des blocs, ce qui est coûteux.

Défauts de la dé-allocation manuelle

L'allocation de mémoire traditionnelle (manuelle) est difficile :

- Si un objet est dé-alloué trop tôt, les références pointeront vers des données aléatoires (*dangling pointers*).
- Si un objet est dé-alloué trop tard ou pas du tout, le tas devient trop volumineux (*space leaks*).
- Environ la moitié du temps passé à déverminer un programme C++ est imputable à des erreurs de gestion de la mémoire. *Purify* ou *CenterLine* aident à détecter ces problèmes.

La dé-allocation manuelle restreint les librairies :

- Il n'est pas possible d'allouer des valeurs générales sur le tas car il faut toujours garder un pointeur pour la dé-allocation.
- Comparez les chaînes de caractères en C/C++ et en Scala.

La dé-allocation manuelle n'est pas sûre.

Glanage de cellules

Le glanage de cellules ou GC (*garbage collection*) permet de régénérer automatiquement l'espace mémoire inutilisé.

- Un espace est considéré inutilisé s'il n'est pas atteignable.
- Un objet est atteignable si un chemin y mène.
- Un chemin est une séquence de références débutant par une référence globale ou une référence sur la pile et suivie ou non par des sélections de champ ou d'élément.

Il existe trois familles principales de GC :

1. Par comptage de références
2. Par marquage–balayage
3. Par copie

GC par comptage de références

Le comptage de références (*reference counting*) fonctionne de la manière suivante.

- Chaque bloc sur le tas contient un champ `refCount` qui indique le nombre de références vers cet objet.
- A la création du bloc, `refCount` vaut zéro.
- Pour chaque affectation `x = y`, on exécute :
```
if (x != null) x.refCount = x.refCount - 1;
if (y != null) y.refCount = y.refCount + 1;
```
- Chaque fois qu'un pointeur non nul est dé-alloué de la pile, on réduit aussi son `refCount`.
- Lorsque `refCount` vaut zéro pour un bloc, on dé-alloue ce bloc et régénère la mémoire utilisée.
- Pour toutes les références d'un bloc dé-alloué, `refCount` est réduit ; ceci peut amener à des dé-allocations supplémentaires.

Le GC par comptage de références :

- est facile à implanter ;
- peut être utilisé sans support externe ou grâce à des classes spéciales («*smart pointers*» de C++) ;
- régénère immédiatement la mémoire ;
- Ce qui implique une meilleure localité des données du tas.

Mais aussi :

- est cher dans le cas des opérations simples sur les pointeurs.
- requiert des données additionnelles dans chaque bloc.
- Les références cycliques ne sont pas dé-allouées (ce qui est le plus grave).

GC par marquage–balayage

Le marquage–balayage (*mark & sweep*) se fait en deux phases :

1. Le marquage : en partant des variables globales et de la pile (les racines), on suit tous les chemins et on marque tous les objets comme visités (en posant un bit dans le descripteur).
2. Le balayage : on traverse linéairement la mémoire et on régénère tous les objets qui n'ont pas été marqués.

Problème : Lors de la visite des chemins, on doit se souvenir où aller ensuite (par récursion ou en maintenant une pile).

- Si un chemin est long, ces données peuvent être conséquentes.

Solution : On garde l'adresse de retour directement dans les objets visités, par exemple en utilisant la rotation de pointeur.

Le GC par marquage–balayage :

- peut dé-allouer les références cycliques ;
- est relativement simple à implanter ;
- consomme peu d'espace additionnel.

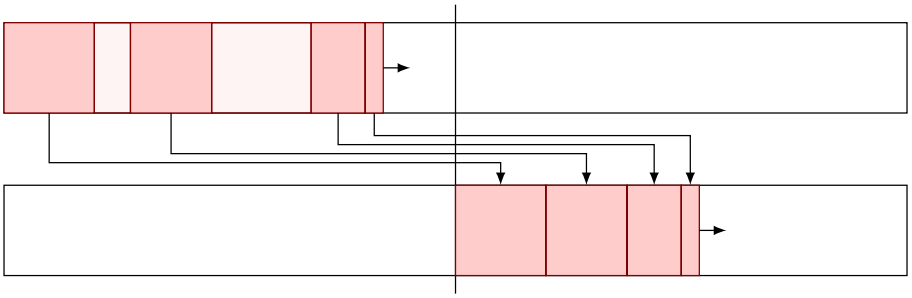
Mais aussi :

- La fragmentation devient vite problématique.
- L'allocation depuis une liste peut être coûteuse.

GC par copie

La copie fonctionne de la manière suivante.

- Le tas est séparé en deux espaces.
- Les objets sont alloués linéairement dans un espace.
- Quand un espace est plein, on copie tous les objets atteignables dans l'autre espace.



Algorithme : GC par copie

```
scan = free =  Start of To-Space
for R in Roots: R = copy(R)
while scan < free:
  for f in fields(object at scan):
    scan.f = copy(scan.f)
  scan = scan + size(scan)

copy(P) =
  if P.forward != null:
    return P.forward
  else
    copy size(P) bytes from P to free
    P.forward = free
    free = free + size(P)
    return P.forward
```

Le GC par copie :

- peut dé-allouer les références cycliques ;
- est très simple à implanter ;
- permet des allocations extrêmement rapides :
  Il suffit d'accroître le pointeur de tas et de tester la limite. On peut même le faire en-ligne.
- La compactage est automatique.

Mais aussi :

- Requiert deux fois plus de mémoire car deux espaces sont requis.
- La localité des données est catastrophique si l'espace est plus grand que l'antémémoire.

Glanage par génération

Problème : Glaner l'entier du tas est long.

On observe que la plupart des objets meurent jeunes. C'est-à-dire que plus un objet vit longtemps, plus il est probable qu'il survivra au prochain glanage.

Glaner les objets jeunes est donc le plus profitable.

Solution : Garder les objets jeunes dans un petit espace dédié qui sera glané plus souvent :

- Le glanage est plus rapide et
- il libère proportionnellement plus de place.

Pour implanter le GC par génération, on remplace les deux espaces du glanage par copie par n espaces, où n est le nombre de générations.

1. On collecte la génération la plus jeune de l'espace 0 vers l'espace 1.
2. Si cela réussit, le GC est terminé : nous avons obtenu de l'espace.
3. Sinon, si l'espace 1 se remplit lors de la collecte, on copie ses données dans l'espace 2.
4. Etc.

Attention : Pour éviter de devoir chercher les références dans l'entier du tas, on stocke les références de génération plus ancienne vers génération plus jeune.

Glaneurs de cellules : aujourd'hui

Pour les ordinateurs personnels ou les serveurs, les glaneurs de cellules les plus performants utilisent la copie multi-générationnelle.

- La taille de la génération la plus jeune est suffisamment réduite pour tenir dans l'antémémoire.
- Les générations les plus anciennes (et les plus grandes) sont souvent subdivisées afin d'en améliorer la localité et réduire les temps de pause (algorithme évolutif)

Pour les systèmes embarqués qui ont des contraintes de mémoire plus fortes, on utilise en général un glaneur par marquage–balayage.

Glaneurs de cellules : demain

Les techniques de glanage de cellules sont un domaine de recherche encore actif aujourd'hui. On étudie les glaneurs :

Concurrents ou comment procéder au glanage simultanément à l'exécution d'un programme qui alloue et modifie de la mémoire dans le tas ?
C'est tout particulièrement important avec les processeurs multi-coeurs modernes.

Temps-réel ou comment réduire et prédire les temps d'arrêt causés par le glanage pour le rendre compatible avec des systèmes temps-réel ?

Distribués ou comment passer des références entre systèmes avec des performances acceptables ?