Projet METAL

GC

Auteur : Sylvain Huet

Création : 18/12/02
Dernière mise-à-jour : 24/01/03

1 Introduction

1.1 Aperçu sur le document

Description d'un mécanisme de GC incrémental.

2 Principe

2.1 Objectifs

Il s'agit de développer un système de gestion de mémoire permettant un mécanisme de GC incrémental, c'est-à-dire qui ne nécessite pas la suspension du calcul en cours (pour éviter les petits blocages constatés dans Scol).
D'une manière générale, il s'agit de pouvoir découper le mécanisme de GC en opérations en temps constants (que l'on appellera des 'pas'), et de les entrelacer avec les traitements du programme Scol.

Il semble qu'une telle contrainte oblige à avoir des allocations mémoires immobiles : un bloc alloué ne peut se déplacer. En effet, outre le temps de copie du bloc, il se poserait la question de la mise à jour des liens vers ce bloc. On voit mal cette opération pouvoir être en temps constant.

On rappelle qu'il y a deux types de blocs de mémoire : les blocs binaires et les blocs tableau. Seuls les blocs tableau peuvent référencer d'autres blocs.
Le système reconnaît des « racines » qui sont les blocs immédiatement utiles. Par parcours du graphe le long des blocs tableau, le système est alors capable de calculer l'ensemble des blocs encore utiles.

Le fait de ne pouvoir déplacer les blocs alloués n'aboutit pas à l'allocation optimale de la mémoire, mais à un découpage en gruyère de la mémoire.

2.2 Algorithme

La mémoire est découpée en blocs.

Un bloc est constitué d'un header et d'une zone de contenu.

A tout moment un bloc peut être dans deux états (et cette information est contenue dans le header) :
- bloc potentiellement encore utile
- bloc disponible à l'allocation

Le système garantit qu'il n'y a pas deux blocs disponibles adjacents (ils seront automatiquement fusionnés).

Les blocs disponibles à l'allocation sont référencés dans une table comme on le verra un peu plus loin.

Le GC procède par deux étapes successives :
- détermination des blocs encore utiles
- suppression des blocs inutiles

On gère une liste des blocs « utiles ». Cette liste est initialisée une seule fois au démarrage (liste vide).

Chaque bloc potentiellement encore utile contient un flag 'marqué'.
Au début de la première étape, tous ces flags sont nuls.
L'algorithme de la première étape est le suivant :
- au début tous ces flags sont nuls.
- on ajoute la liste des racines à la liste des blocs « utiles »
- à chaque pas, on dépile un élément de la liste, on le marque, et s'il s'agit d'un bloc tableau, on empile dans la liste les éléments non marqués référencés par le bloc

(on objectera que chaque pas ne se fait pas en temps constant mais linéaire sur la taille d'un bloc tableau ; il serait aisé de découper ce pas en pas constants, mais l'intérêt sera probablement faible).

A la fin de la première étape, tous les blocs utiles ont un flag mis à 1. Il y a éventuellement quelques blocs désormais inutiles qui ont un flag mis à 1.

La seconde étape parcourt linéairement les blocs de la mémoire.
L'algorithme est le suivant :
- à chaque pas on examine le prochain bloc
- s'il s'agit d'un bloc potentiellement utile, mais non marqué :
    - si le bloc précédent était disponible à l'allocation, alors on fusionne les deux blocs, et on les place comme disponibles à l'allocation
    - sinon, on définit le bloc comme disponible à l'allocation
- s'il s'agit d'un bloc potentiellement utile et marqué, on efface le flag
- s'il s'agit d'un bloc disponible à l'allocation et si le bloc précédent était disponible à l'allocation, alors on fusionne les deux blocs, et on les place comme disponibles à l'allocation

Un bloc nouvellement alloué voit sont flag mis à 1.
La création d'un lien vers un bloc dont le flag est à 0 provoque l'ajout de ce bloc dans la liste des blocs « utiles ».

La création d'un lien peut se faire de deux manières :

- ajout d'un lien dans un bloc tableau
- ajout d'un lien dans une racine (pile par exemple : stackpush ou stackset)

A la fin de la seconde étape, tous les flags qui étaient à 1 à la fin de la première étape sont bien remis à 0, et il n'y a pas deux blocs adjacents disponibles à l'allocation.

Le stockage des blocs disponibles à l'allocation doit se faire dans un souci d'optimisation des processus d'allocation et de désallocation.
Rappelons que le calcul de l'allocation consiste à trouver le bloc de taille supérieure à la taille requise, mais si possible le plus proche de la taille requise. Une fois le bloc trouvé, il faut le supprimer de la structure de stockage des blocs disponibles. Si le bloc est trop gros, on le découpe en deux et on stocke la partie excédentaire (il faut que la taille de l'excédent soit au moins le header plus un).
Le calcul de désallocation consiste à stocker un bloc dans la structure des blocs disponibles.
Le calcul de fusion de deux blocs peut être vu comme un mécanisme d'allocation/désallocation.

Une manière efficace est de stocker les blocs disponibles dans des listes doublement chaînées. Chaque liste contient des blocs de taille comparable.

Le calcul d'allocation consiste a priori à chercher un bloc dans la liste correspondant à la taille recherchée. Si cette liste est vide, on passe à la liste suivante et ainsi de suite.
Si le nombre de listes est élevé, cette recherche peut s'avérer très longue. Surtout qu'au début (un seul bloc pour toute la mémoire), la recherche se fera sur toutes les listes, jusqu'à la liste des plus grands blocs.
Une solution consiste à regrouper les listes en paquets. Pour chaque paquet, on indique le prochain paquet non vide. A l'intérieur de chaque paquet, et pour chaque liste, on indique la prochaine liste non vide (éventuellement aucune, dans ce cas on cherchera le prochain paquet non vide).
En prenant comme taille de paquet (et comme nombre de paquets) la racine carrée du nombre de listes, on obtient :

- un algorithme de recherche en temps constant
- un algorithme de stockage en au plus 2*sqrt(n)
- un algorithme de déstockage en au plus 2*sqrt(n)

La structure d'un bloc alloué doit donc contenir les informations suivantes :

- taille
- type (binaire/tableau)
- flag (marqué)
- pile (pour la première étape)

La structure d'un bloc disponible à l'allocation doit contenir les informations suivantes :

- taille
- 2 pointeurs suivant/précédent

On a alors besoin d'un header de taille 2 pour les blocs alloués et 3 pour les blocs non alloués.