Garbage collecting
(Ramasse-miettes, glanage de cellules)

B. Dupouy

Plan

- Introduction
  - Rôle d'un *Garbage Collector (GC)*
  - Rappels sur les objets
- Algorithmes
  - Comptage de références
  - Mark & Sweep, Mark & Compact
  - Algorithmes copiants
  - Treadmill
- GC incrémentaux
- GC générationnels
- Exemple : le GC de la JVM

© ENST INFRES

Introduction

- Garbage collector se traduit par : ramasse-miettes, glanage de cellules, que nous abrégerons en GC,
- La gestion de la mémoire (allocation et libération) peut être laissée à la charge du programmeur:
  - C : *malloc*, C++ : *new,*
  - Les problèmes à résoudre sont alors :
    - Quand faut-il libérer la mémoire (*free*) ?
    - Eviter les incohérences :
      - *dangling pointers* (pointeurs vers « rien » : mémoire « trop rapidement» libérée),
      - *memory leaks* (accumulation de blocs inutilisés : mémoire « pas assez rapidement» libérée),
- La gestion de la mémoire peut aussi incluse dans le langage (Java, Lisp, Small Talk)

© ENST INFRES

Un GC Est-il nécessaire?

- « *La mémoire RAM est de plus en plus étendue, alors à quoi bon un GC ?* »
  - La gestion de la mémoire n'a généralement rien à voir avec la sémantique du programme et peut la dévoyer :
    - Elle dépend de la configuration matérielle,
    - Elle est difficile à traiter car les bogues apparaissent de façon non répétitive, ou seulement si l'application est utilisée assez longtemps,
  - Les applications embarquées peuvent être contraintes par la taille de la mémoire (*small memory footprint*),

- Prise en charge de cette gestion par le GC :
  - Augmente la productivité, simplifie les API,
  - Etre actif (live, vivant) est une propriété globale, indépendante du module où se trouvent les objets :
    - le GC adopte donc une stratégie globale : détecter périodiquement les objets inutilisés (morts, deads) et recupérer l'espace mémoire qui leur été alloué,
  - coopération possible entre compilateur et GC

© ENST INFRES

Objets et mémoire

- Un objet est un ensemble structuré, dont le type est facilement identifiable, il peut contenir des **références** (pointeur + informations de typage) vers d'autres objets,
- Son espace d'adressage est alloué sur le tas (*heap*),
- Qu'est ce qu'un objet inutilisé, mort ou *garbage* ?
  - définition dynamique, point de vue du processeur et du compilateur :
    - un objet qui ne sert plus lors de l'exécution du programme,
  - définition statique, point de vue de la mémoire et du GC :
    - un objet qui ne peut pas être atteint (transitivement) à partir d'une racine,
- La racine (*root set)* est l'espace à partir duquel sont atteints tous les objets actifs :
  - pile de la *run time*, variables statiques , registres…

© ENST INFRES

Rôle du GC

- **Trouver** les objets ou les blocs mémoire inutilisés,
- **Récupérer** (rendre accessible) l'espace qu'ils occupent,
- Les contraintes sur la mémoire sont les suivantes :
  - Eviter la **fragmentation**,
  - Maintenir de bonnes propriétés de **localité,**
- Durée de vie d'un objet :
  - un objet, un bloc mémoire est considéré comme inutilisé, libre, mort, *garbage*, si il n'est pas accessible via un jeu de références,
  - un objet qui devient *garbage* le **reste,**
  - si il n'existe pas de chemin de références vers un objet : il est garbage, mais un objet référencé est-il forcément vivant ?
  - mesure de la durée de vie : notion de *conservatisme* du GC

© ENST INFRES

Fonctionnement

.• La gestion de la mémoire par un GC se fait en deux temps. Ces deux phases peuvent être séparées ou imbriquées :

- **phase 1 :** trouver les objets inutilisés. Il existe diverses techniques : marquage (*mark*), comptage de références (*reference counting*), …
- **phase 2** : récupérer l'espace libéré, par construction d'une liste (*free list, bitmap) ,* ou par copie (les phases 1 et 2 peuvent être entrelacées),

.• Les algorithmes de la phase 1 déterminent ceux de la phase 2

• Si le GC travaille parallèlement aux applications : notion de *mutator* et *collector* (Dijkstra) :

- le *collector* (le GC) cherche des objets inutilisées pour libérer de l'espace mémoire,
- les *mutators* (les applications) s'allouent de l'espace mémoire,

© ENST INFRES

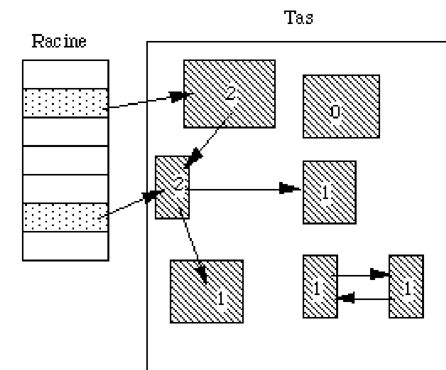
Comptage de références

• Un compteur est associé à chaque objet :

- il est incrémenté lors de chaque nouvelle référence à l'objet,
- il est décrémenté à chaque suppression de référence sur cet objet (déréférencement),

• Phase 2 : plutôt que d'attendre que la mémoire soit saturée, on libère les objets dès que plus aucun pointeur ne les référence,

• Problème : cette méthode ne détecte pas les cycles, exemple :



© ENST INFRES

Comptage de références
Exemple

• Si une reference est changée pour pointer d'un objet vers un autre, il faut mettre à jour deux compteurs et faire un test de nullité,

• Exemple :

- ici new renvoie une réference sur objet et initialise son champ `Nombre_de_ref` à 1;
- si Ptr2 est une référence vers l'objet Obj2,
- alors l'affectation : Ptr1 = PtR2 ;

Aura l'effet suivant :

```
/* Incrementer compteur de ref sur Obj2 */
Ptr2. Nombre_de_ref ++;
/* Decrementer compteur de ref sur Obj1 */
Ptr1. Nombre_de_ref --;
if (Ptr1.Nbre_ref == 0)
            { Marquer Objet Obj2 inutilisé }
Ptr1 = PtR2;
```


© ENST INFRES

Comptage de références
Avantages/Inconvénients

• Avantages :

- facile à implémenter,
- incrémental par nature, il n'interrompt pas trop longtemps les applications (sauf si on alloue/libère de grands espaces), il réparti dans le temps les effets du GC,
- il peut être rendu temps réel par seuillage des libérations/allocations,
- bonne gestion des objets à courte vie,

• Inconvénients :

- **les listes circulaires ne sont pas détectées,** (le comptage de references est une approximation ***conservative*** de la durée de vie. Si un objet n'est pas référencé par un autre, il est garbage. Le contraire n'est pas forcement vrai : un objet garbage peut en référencer un autre.
- un compteur par objet,
- Coût proportionnel à l'activité du programme,
- Copies répétitives des objets à longue durée de vie,

© ENST INFRES

Comptage de références

• Optimisation :

- ne pas gérer systématiquement les variables locales et ne faire que périodiquement les mises à jour des compteurs des objets sur la pile,

• Implantation :

- En arrière plan,
- Le comptage peut être fait avec un compteur sur 2 bits, si ce compteur vaut 3 l'objet est considéré comme vivant et le reste,
- Peut être géré au niveau du thread (Ada, C++)

© ENST INFRES

Mark and sweep

• Etape 1 (*mark*) :

- Construire un graphe des objets utilisés en partant de la racine (plusieurs techniques possibles : profondeur d'abord, …), les objets non atteints seront considérés comme *garbage*

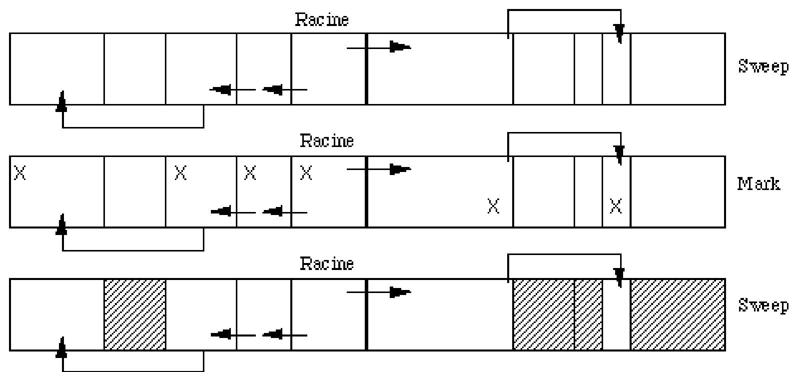
• Etape 2, le *collector* (*sweep, de-allocation)* :

- parcourir les objets libres, c'est-à-dire non marqués, et les repérer grâce à une technique ad hoc (*free list, bit map*),
- on parle de libération en place, *in place de-allocation,*
- le *lazy sweep* est celui qui se fait au moment de la recherche de blocs libres,

© ENST INFRES

Mark and sweep

• Exemple :



© ENST INFRES

Mark and sweep Avantages/Inconvénients

• Caractéristiques:

- Sa durée est proportionnelle à la taille du tas,
- On scrute à la fois les objets vivants et les objets inutilisés.

• Inconvénients :

- on arrête tout pour l'exécuter (*stop and go*, *disruptive*),
- **fragmentation**, surtout si les blocs sont de taille très variable,
- non respect de la localité, Les objets ne bougent pas et des objets d'âges différents vont devenir voisins (problème en mémoire paginée),

© ENST INFRES

Mark and compact

• Remédie à la fragmentation de Mark & Sweep : pas de libération en place, mais compactage des blocs libérés,

• Etape 1 (*mark*) : comme Mark & Sweep

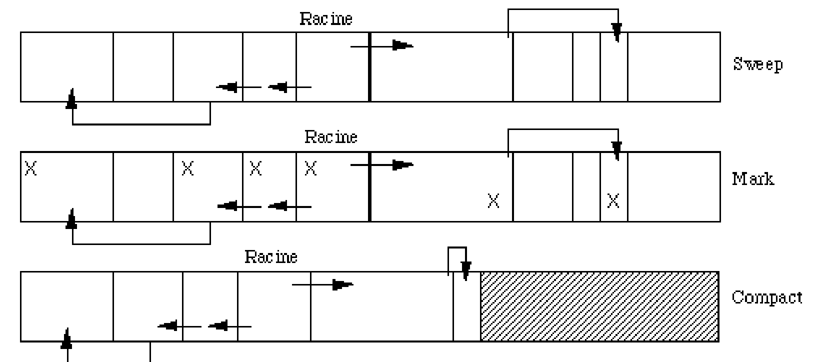
• Etape 2, amélioration de Mark & Sweep :

- Compactage des blocs libres et des blocs utilisés,
- **Mais** :
  - il faut modifier les références vers les objets (adresses à recalculer),
  - copie successives des objets vivants à chaque compactage,

© ENST INFRES

Mark and compact

• On reprend l'exemple précédent :



© ENST INFRES

Mark and compact Avantages/Inconvénients

• Avantages :

- Allocation de mémoire par simple incrément du pointeur sur la zone libre,
- comme pour Mark & Sweep, tous les threads sont suspendus pendant le GC, mais la durée de cette suspension est proportionnelle au nombre d'objets sur le tas, pas à sa taille,

• Inconvénients :

- durée du compactage (trois passes : calcul des nouvelles adresses, mise à jour des pointeurs, déplacement des objets),
- un objet à longue durée de vie va être copié plusieurs fois,

© ENST INFRES

Algorithmes copiants

• Pour remédier au problème de fragmentation, on peut utiliser des algorithmes dits copiants (*copying GC*) :

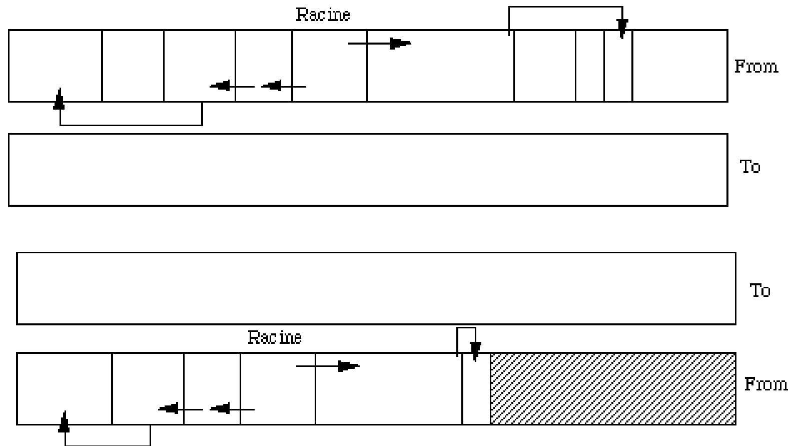
- Le tas est divisé **en deux demi-espaces** *From* et *To*
  - **(1)** Le(s) mutator(s) alloue(nt) dans *From*
- **(2)** Le GC est lancé quand *From* est plein :
  - **(3)** les objets vivants sont compactés et copiés dans *To*,
  - **(4)** puis *To* et *From* sont permutés
  - **(5)** on considére que tout l'espace *From* est libre et on recommence en 1.

• Schéma fonctionnel , il existe de nombreuses implémentations

© ENST INFRES

GC copiants

• Exemple :



© ENST INFRES

Algorithmes copiants

• Caractéristiques :

- Pas de fragmentation,
- On parle de s*cavenging* (nettoyage) : les objets à garder sont sélectionnés, tout le reste est *garbage,*
- utilisation de *forwarding pointers* : quand on accède à un objet O, et si cet objet O a bougé, on le remplace par un *forwarding pointer* pour éviter les copies multiples pendant la même passe,
- le GC en arrière plan copie les objets vivants dans To, quand ils ont tous été copiés, on permute les espaces,
- Plusieurs implémentations, un classique : Cheney (parcours en largeur d'abord),

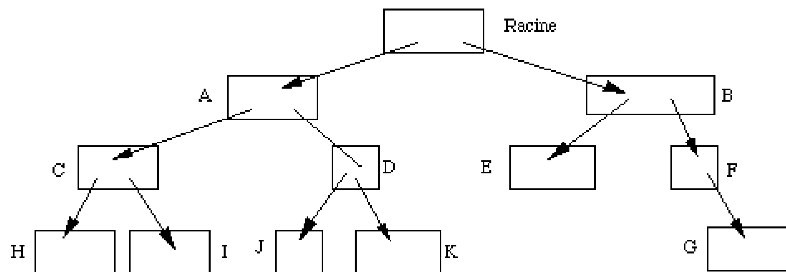
© ENST INFRES

Algorithmes copiants : Cheney

• Implantation **itérative** d'un algorithme copiant, elle entrelace les phases de scrutation (*scan*) et de copie,

- pointeurs nécessaires :
  - un sur l'espace à parcourir (*scan*),
  - un sur la zone libre (*free*),
  - les *forwarding pointers*…
- Attention : la scrutation en « largeur d'abord « va à l'encontre du principe de localité

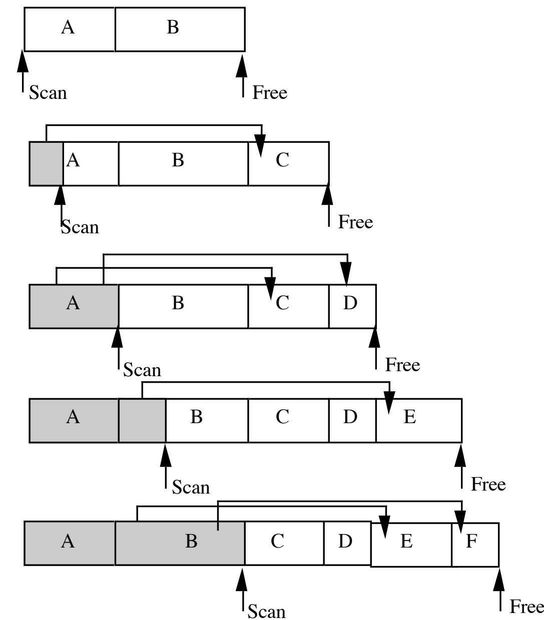
• Exemple, soit la configuration mémoire suivante (zone *From*) :



© ENST INFRES

Algorithmes copiants Cheney (exemple)

• Rangement des blocs dans la zone *To* au fur et à mesure du *scanning* :



© ENST INFRES

Algorithmes copiants

- Déclenchés quand la mémoire est saturée,
- On peut aussi les déclencher à une date arbitraire, leur durée est fonction du nombre d'objets vivants à cet instant,
- Ils travaillent en une seule passe : scrutation et déplacement se font en même temps, le temps d'exécution est proportionnel au nombre d'objets vivants,
- Le *collecting* est implicite : les objets qui n'ont pas été transférés dans la zone active sont considérés comme inactifs,
- le mutator accède à From,

© ENST INFRES

Algorithmes Copiants Avantages/Inconvénients

- Avantages :
  - Leur durée est proportionnelle au nombre d'objets vivants, ils sont efficaces lorsqu'il y a beaucoup d'objets temporaires,
  - Allocation facile : incrément d'un pointeur,
- Inconvénients :
  - Il faut des "forwarding pointers" pour déplacer les données,
  - Copie répétitives des objets à longue durée de vie,
  - Comment différencier pointeur et entier ?
  - Gourmands en espace mémoire, ils sont inutilisables dès que la taille du tas est importante. C'est-à-dire souvent…

© ENST INFRES

Algorithmes copiants : efficacité

• La durée du travail proportionnelle à la quantité de mémoire active :

- si on suppose constante cette quantité, diminuer la fréquence du GC réduira sa durée. En effet l'âge des objets va augmenter et ils seront sans doute plus nombreux à être inutilisés au moment du GC et n'auront donc pas à être copiés,
- diminuer la fréquence du GC en augmentant la taille du tas, exemple :
  - le processus P utilise 20 Mo dont 1Mo sont actifs en permanence. On dispose de 2 demi-ensembles de 3 Mo.
    - Donc le GC se déclenche environ 10 fois (il reste 2Mo) en moyenne, et il copie environ la moitié de ce qui est alloué.
  - Si on double la taille des demi ensembles, alors 5 Mo seront libres, le GC se déclenchera environ 4 fois,

© ENST INFRES

Algorithme De Baker

• C'est une implémentation (qui ne copie pas !) d'algorithme copiant :

- Au lieu de déplacer les objets, on les caractérise pour indiquer à quel espace ils appartiennent,
- trois champs sont associés à chaque objet : une couleur, Blanc ou Noir (qui indique From ou To), et deux pointeurs (les listes sont doublement chaînées),
- quand le mutator accède à From:
  - l'objet est copié dans le To (Noir)
  - si on le rend incrémental, il faut ajouter une couleur

© ENST INFRES

Algorithme De Baker

• Comparaison avec les algorithmes copiants :

- incrémental,
- rapide : simple déplacement de pointeurs,
- inconvénients : plus d'espace de gestion,
- fragmentation (idem),
- le parcours des gros objets plus rapide (comme dans Mark & Sweep), les objets ne bougent pas,
- moins de contraintes pour les compilateurs : pas de pointeurs à changer,

© ENST INFRES

Marquage des Trois couleurs

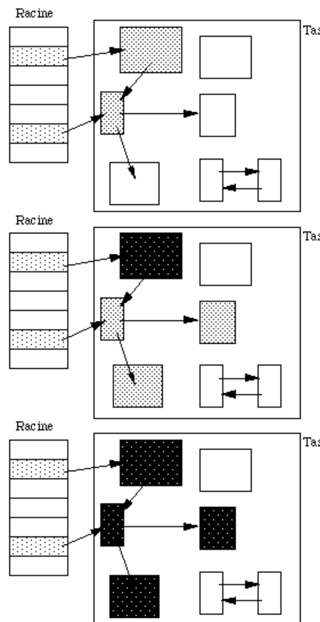
• Algorithme des trois couleurs :

- les couleurs :
  - noir : actifs déjà visités,
  - gris : pas encore visités,
  - blanc : inutilisés,
- un invariant : pas de lien d'un noir vers un blanc,
- quand il n'y a plus d'actifs à noircir, le parcours est fini
  - début : tous blancs, racine grise
  - tant qu'il y a des gris , pour chaque gris : marquer ses fils gris, si tous ses fils sont marqués, il passe noir
  - fin : ceux qui sont restés blancs sont déclarés inutilisés, les noirs sont des objets actifs, il ne reste plus de gris

© ENST INFRES

Marquage. des Trois couleurs

• On reprend l'exemple donné pour le comptage de références :



© ENST INFRES

Marquage. des Trois couleurs

- On introduit le gris à cause de l'activité **simultanée** du GC (*collector*) et des tâches utilisateurs (*mutators)* : elles peuvent faire accès à un objet Blanc depuis un objet Noir, introduisant ainsi une incohérence,

- Ceci est possible dans plusieurs cas :
  - lecture, par un objet Noir, d'une référence qui pointe vers un objet Blanc,
  - mise à jour, par un objet Noir, d'une référence qui va maintenant pointer vers un objet Blanc,

- Remarque : les GC font du coloriage, le garbage est Blanc le reste est Noir :
  - M&S est une implémentation directe,
  - dans les copiants, ce qui reste dans From = Blanc, To = Noir,
  - le gris est introduit à cause de l'activité **simultanée** du GC (*collector*) et des tâches utilisateurs (*mutators)* : elles peuvent faire accès à un objet Blanc depuis un objet Noir, introduisant ainsi une incohérence, (cf. notion de *grey wavefront*)

© ENST INFRES

Synchronisation Mutator/collector

• *read barrier* :

- elle contrôle où se fait l'accès mémoire lors de la lecture d'une référence (c.a.d quelle espace mémoire est désigné par le pointeur),
- si on tente une lecture vers un objet Blanc depuis un objet Noir, marquer le Blanc en Gris ou tous les deux Gris,

• *write barrier* :

- elle contrôle les mises à jour des références, c'est-à-dire les écritures dans les pointeurs,
- si on créé un pointeur vers un objet Blanc dans un objet Noir, alors cet objet devient Gris

• Remarque :

- On essaie d'utiliser un seul type de barrière, généralement en lecture (utilisation en lecture des pointeurs plus fréquente que leur modification)

© ENST INFRES

Synchronisation Mutator/collector

• Création d'un pointeur d'un objet Noir vers un objet Blanc :

- Noir devient Gris
- Ou : Blanc devient Gris

• Remarque, il n'y a un problème que si :

- **(1)** dans un objet Noir : mise à jour d'une référence pour la faire pointer vers un objet Blanc,
- **(2)** ET destruction du pointeur original avant que le collector ne le voie

• En effet :

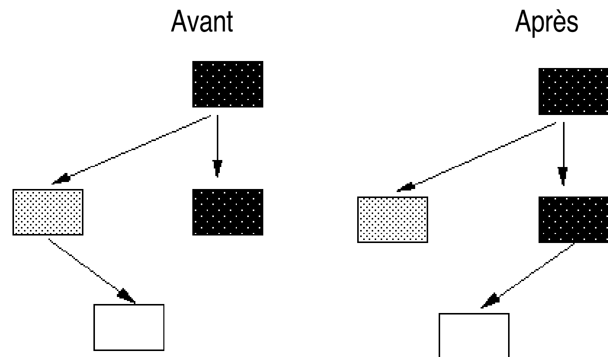
- si **(1)** n'est pas vraie, soit il y a un pointeur vers ce Blanc dans un Gris, il sera retenu, sinon ce Blanc est bien du garbage
- si **(2)** n'est pas vraie, l'objet sera atteint via le pointeur original, et retenu,

© ENST INFRES

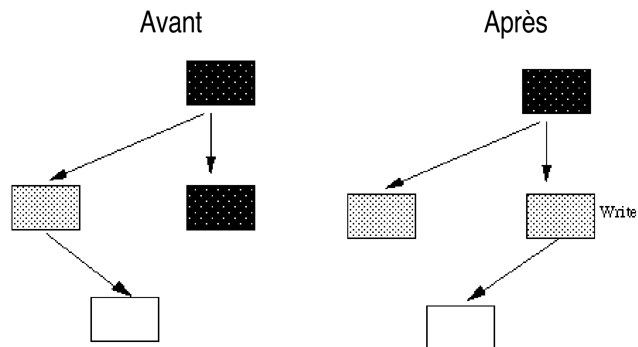
Synchronisation Mutator/collector

• Exemple :

- l'objet Gris libère un objet ,
- dans l'objet Noir, on initialise un pointeur vers cet objet :



• Solution :



© ENST INFRES

Synchronisation Mutator/collector

• Quand on fait accès à un objet Blanc depuis un Noir en modifiant un pointeur, cet objet Noir devient Gris :

• Quand on fait accès à un objet Blanc depuis un Noir en lisant un pointeur, un ou les deux deviennent Gris

© ENST INFRES

GC incrémentaux

• Le traçage incrémental est plus intéressant que le *collecting* incrémental :

- on rend donc les traceurs incrémentaux
- pb : le programme peut changer le graphe (*mutate*), pendant le travail du GC
  - attention à ne pas introduire d'incohérences. Il faut maintenir l'invariant, un Noir ne doit pas pointer vers un Blanc (notion de *wavefront*)
- définir le « conservatisme » (cf. *floating garbage*)

• Synchronisation *collector/mutator*, au choix :

- *write barrier* :
  - détecter les créations de pointeurs vers des objets marqués Blancdans un Noir. Le Noir devient Gris, ou le Blanc devient Gris (on avance la *wavefront).*
- *read barrier* :
  - détecter les lectures par un Noir vers un objet Blanc. Le Blanc devient Gris.

© ENST INFRES

Techniques Incrémentales Et trois couleurs

• Mark & Sweep et Mark & Compact ne vont plus être « atomiques » :

- on va les exécuter de façon incrémentale,
- synchroniser l'accès au graphe qui va être partagé entre le GC et les applications (*mutators*),
- Mark & Sweep : un écrivain, N lecteurs (seul le mutator change le graphe),
- Les copiants : N écrivains, N lecteurs (le *mutator* et le *collector* modifient les pointeurs),

• Algorithme des trois couleurs :

- *Mark & Sweep* : on utilise un bit pour la couleur, les gris sont sur la pile,
- Cheney : gris, ceux qui sont entre les pointeurs,
- Copiants : noirs et gris dans *To*, blancs dans *From*,

© ENST INFRES

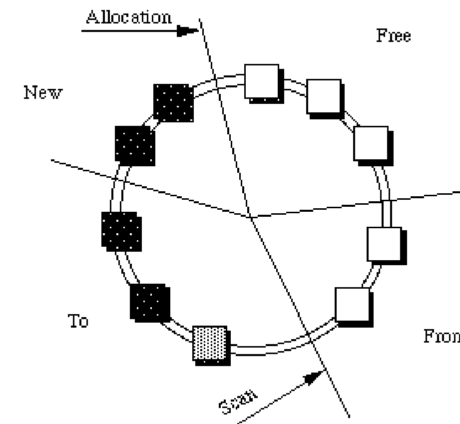
Techniques Incrémentales : Treadmill

• Baker donne une variante temps réel de son GC incrémental :

- on ne gère pas des espaces séparés, mais des listes doublement chaînées,
- elles sont organisées en un tampon circulaire divisé en 4 zones :
  - *New* pour l'allocation des nouveaux objets par les *mutators* (Noir) pendant une passe du GC,
  - *From* où se trouvent les objets qui doivent être traités par le GC,
  - le GC fait passer des objets de *From* à *To* (de Blanc à Gris ou Noir),
  - à la fin du passage du GC :
    - l'espace récupéré dans *From* (les objets Blancs) est accolé à celui de *Free* par simple jeu de pointeurs,
    - *To* et *New* sont réunis dans *To* qui sera traitée par le cycle suivant du GC

© ENST INFRES

Treadmill Fonctionnement



• Au début :

- *From* = *New* + *To* du cycle précédent,
- *New* et *To* sont vides,

• Les applications s'allouent de la mémoire en prenant des blocs dans *Free* pour les mettre dans *New* (où ils sont Noirs),

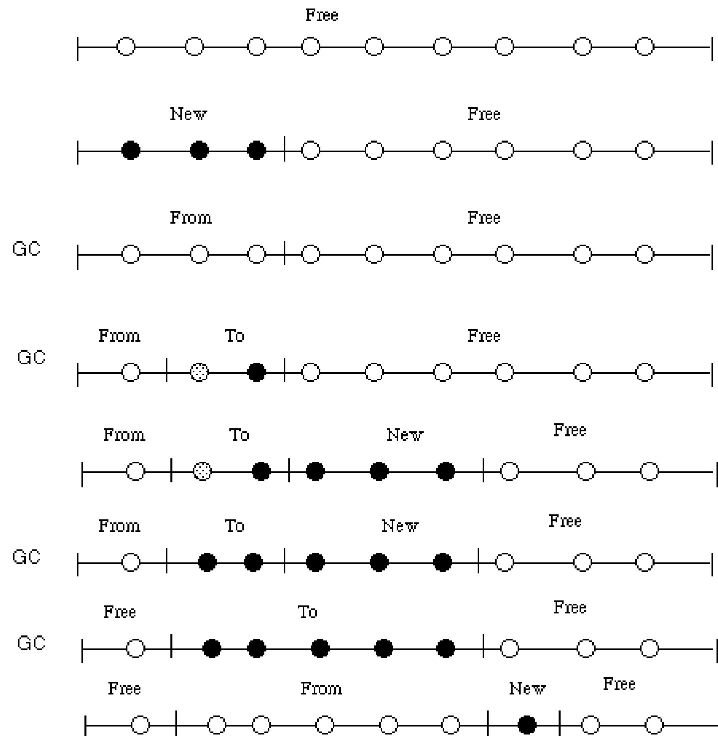
• Le GC déplace (*scan*) des objets de *From* (où ils sont Blancs) vers *To* (où ils sont Gris ou Noirs),

• A la fin du cycle du GC :

- *From* est réuni avec *Free* pour former un nouveau *Free*
- *To* (tous Noirs) est réuni avec *New* et forme un nouveau *To*

© ENST INFRES

Treadmill : Exemple



© ENST INFRES

GC Générationnels :

- S'appuient sur les constations suivantes :
  - beaucoup d'objets deviennent rapidement inaccessibles : la plupart des objets ont une vie courte,
  - seuls, quelques objets restent utilisés très longtemps,
- Conséquences :
  - Le GC doit se concentrer sur les objets jeunes,
  - Le tas est divisé en plusieurs zones correspondant à des âges différents,
  - *Minor collection* sur la jeune, rapide et fréquente, (GC copiant)
  - *Major Collection* sur la vieille, lente, moins souvent, (Mark & Compact)

© ENST INFRES

GC Générationnels

- Les objets sont regroupés en générations en fonction du temps écoulé depuis leur création :
  - les jeunes générations, parcourues plus souvent parce qu'on y trouve des objet récents inutilisés (utilisation d'un algorithme copiant, plus rapide si les objets à récupérer sont nombreux, mémoire requise proportionnelle à la taille de ces générations)
  - les vieilles générations, parcourues moins souvent (utilisation d'un algorithme de marquage et balayage, l'espace mémoire à parcourir est proche de la taille de ces générations, il demande peu de mémoire supplémentaire)

• Promotion de la jeune à la vieille,

• Hypothèse : il y a peu de références depuis des objets vieux vers des objets jeunes

• Améliore la localité

© ENST INFRES

Récapitulatif

• Principe : quand un objet devient garbage, il le reste,

• Trois étapes pour gérer l'espace mémoire:
- Identifier, détecter les objets actifs (marquage, comptage)
  - *Type aware* ou conservatifs (on ne connaît pas le type des données)
- Rendre, libérer, collecter l'espace inutilisé (collector)
- Ré-allouer cet espace (gérer une free list, une bitmap)

• Différents types de GC :
- Types de base (Comptage, Copie, Marquage)
  - Mark & Compact et copie : les blocs bougent
- Incrémentaux (non disruptive)
- Générationnels
  - regroupement des blocs en fonction de l'âge des objets
  - algorithmes différents pour chaque espace

© ENST INFRES

Récapitulatif

- disruptive / nondisruptive
  - disruptive : on arrête les applications et on traite toute la mémoire
  - nondisruptive : incrémental (adaptés au temps réel) ou concurrent (pb de synchro. avec les applications)
- famille « stop and collect » :
  - mark and sweep, pb : fragmentation, parcours de toute le mémoire,
  - mark and compact, pb : mise à jour des compteurs, copies répétitives,
- famille « copying » (*scavenging*),
  - pb : gros consommateurs de mémoire, si il y a des finalizers, il faut quand même parcourir les objets morts

© ENST INFRES

Récapitulatif

- Deux aspects dans la gestion de la mémoire :
  - allocation,
  - GC, trois composantes :
    - Initialisation : scruter la racine
    - Travail par unité d'allocation : fonction du volume (M & S) ou du nombre d'objets (actifs, cas des copiants, ou tous)
    - Travail durant la scrutation (GC proprement dit)
  - Exemple :
    - 10 % des données sont actives, 90 jamais scrutées
    - si le coût de scrutation (sweep) est 10% de la copie, M& S a le même coût qu'un copiant

- Réalisation :
  - pour de meilleures performances : GC non copiants pour les gros objets,
  - argument en faveur des Mark & Sweep ou *treadmill* : ils peuvent être conservatifs, tandis que les copiants doivent savoir si une valeur est un pointeur ou non (pour le mettre à jour après déplacement de l'objet)

© ENST INFRES

GC et hiérarchie mémoire

- Interaction GC et caches :
  - 1. Mettre les objets les plus fréquemment utilisés dans le cache L2 . Pourquoi? Aide apportée par un GC générationnel ?
  - 2. Faire de l'allocation séquentielle en mémoire. Pourquoi? Aide apportée par un GC copiant ?
  - 3. Limiter le nombre de conflits sur les objets fréquemment utilisés dans le cache. Pourquoi? Aide apportée par un GC générationnel ?
  - 4. Préallouer de la mémoire pour les allocations. Pourquoi? Aide apportée par un GC copiant ?
  - 5. Favoriser une bonne localité. Pourquoi? Aide apportée par un GC copiant ?

© ENST INFRES

Java : GC générationnel

- Pas de spécification d'implémentation du GC dans la JVM
- Le tas (JDK 1.2 et suivantes) est divisé en **deux** parties :
  - **jeune** génération (New Object Region) : petite, objets à durée de vie courte, trois sous parties :
    - Eden, géré en pile, l'allocation d'un objet se fait facilement en incrémentant un pointeur,
    - deux espaces de survivants (survivors) : From et To. Quand l'Eden est plein, le GC vérifie l' accessibilité des objets de l'Eden et copie les vivants dans la région To, puis To devient From et From devient To
  - **vieille** génération, grande objets à durée de vie longue,

- Algorithmes utilisés :
  - copy sur la JG,
  - marquage (trouver les objets inutilisés) et balayage (restitution de leur espace d'adressage) sur la VG,
  - les objets de la JG passent en VG si ils sont assez vieux

© ENST INFRES

New object, Old object regions

- New object region :
  - Objets copiés vers To depuis From et Eden
  - Un seuil (*Tenuring Threshold*) détermine le nombre de fois qu'un objet est copié d'un espace surivants à l'autre avant de la passer en VG (Old Object region)
  - Effet de bord : un des deux espaces de survivants est toujours vide

- New object region :
  - réservée aux objets à durée de vie longue,
  - on suppose que la plus grande partie du garbage est provoquée par des objets à vie courte, et que la VG ne sera pas souvent parcourue

© ENST INFRES

Java

- Syntaxe :

  objets vivants avant GC -> objets vivants après GC (taille du tas), durée du GC

- Option de java : -verbosegc :

```
[GC 1667K->1295K(1984K),     0.0101756 secs]
[GC 1807K->1434K(1984K),     0.0223998 secs]
[GC 1946K->1574K(2112K),     0.0116185 secs]
[Full GC 1574K->1574K(2112K),   0.0830561 secs]
[GC 3454K->2081K(4672K),     0.0495951 secs]
[GC 4001K->2599K(4672K),     0.0274256 secs]
[GC 4519K->3101K(5056K),     0.0308995 secs]
[Full GC 3101K->3101K(5056K),   0.1452472 secs]
[GC 7039K->4131K(9452K),     0.0777414 secs]
[GC 8227K->5174K(9452K),     0.0627538 secs]
[GC 9270K->6209K(10348K),    0.1125570 secs]
```


© ENST INFRES

Incremental

• Syntaxe :

```
objets vivants avant GC -> objets vivants  après GC
(taille du tas), durée du GC
```

• Option de java : -Xincgc

- l'"incremental collector" divise la VG en plusieurs sous-ensemble qu'il traite un par un
- les pauses sont plus courtes, mais le rendement global est plus faible

```
[Inc GC 3566K->3950K(5120K), 0.0309922 secs]
[GC 4078K->3594K(5184K), 0.0264542 secs]
[Inc GC 3594K->3978K(5120K), 0.0272683 secs]
[GC 4106K->3627K(5120K), 0.0272381 secs]
[Inc GC 3627K->4011K(5056K), 0.0285464 secs]
[GC 4139K->3666K(5184K), 0.0281388
```


© ENST INFRES

concurrent GC

• Syntaxe :

```
objets vivants avant GC -> objets vivants  après GC
(taille du tas), durée du GC
```

• Option de java : -Xconcgc

- l'"concurrent collector" permet à d'autre threads de travailler en meme temps que le GC,
- JDK 1.4

```
[GC 1463K->1093K(2560K), 0.0089573 secs]
[GC 1093K(2560K), 0.0053470 secs]
[GC 1094K(2560K), 0.0092867 secs]
[GC 1604K->1228K(2560K), 0.0104823 secs]
[GC 1228K(2560K), 0.0062662 secs]
[GC 1234K(2560K), 0.0097820 secs]
[GC 1740K->1373K(2560K), 0.0115875 secs]
```


© ENST INFRES