Université de Montréal

Gestion mémoire

Comme les boîtes à vitesse:

- Automatique: la désallocation est prise en charge par le langage
- Manuelle: la désallocation est à la charge du programmeur
- Semi-automatique: le langage vérifie les désallocations

Granularité:

- Par objet: chaque objet est alloué/désalloué individuellement
- Par région: chaque objet est alloué dans une région, la désallocation opère sur tous les objets d'une région

Règle générale: facile pour le programmeur $\Rightarrow$ dur pour le compilateur

Université de Montréal

Gestion mémoire manuelle

En C, le compilateur ne peut rien faire

Sinon, il peut optimiser:

- le code de l'allocation
- le lieu d'allocation (tas, pile, global)

Université de Montréal

Gestion mémoire automatique

- Comptage de références: à chaque objet est associé un compteur qui indique combien de pointeurs existent. Lorsque le compteur passe à 0, on peut désallouer l'objet.
- GC (Glanage de Cellules ou plutôt Garbage Collection): à partir des variables globales et de la pile, traverser tous les objets atteignables en passant par tous les pointeurs: les objets non-visités peuvent être désalloués.
- Régions: une analyse du code détermine dans quelle région allouer chaque objet, et à quel moment désallouer chaque région.

Université de Montréal

Garbage Collection

Périodiquement (e.g., besoin de mémoire) chercher les déchets:

En partant des *racines*, marquer transitivement les objets accessibles

Puis, récupérer les objets restés inaccessibles

Deux algorithmes de base: mark&sweep et stop&copy

Le *collecteur* et le *mutateur* doivent se synchroniser

Une phase de collection peut coûter très cher

Université de Montréal

Mark (&Sweep)

```
mark (ptr) {
    if (!ptr->marked) {
        ptr->marked = True;
        for i = 0 to ptr->size
            mark (ptr[i]);
}   }

mark_all () {
    for varptr in roots
        mark (*varptr);
}
```

Université de Montréal

(Mark&) Sweep

```
sweep_all () {
    ptr = heap_start;
    do {
        if (ptr->marked)
            ptr->marked = False;
        else
            free_object (ptr);
    } while (ptr = next_object (ptr))
}
```

Université de Montréal

(Stop&)Copy

```
copy (ptr) {
    if (ptr->forward = NULL) {
        for i = 0 to ptr->size
            *(alloc_ptr + i) = ptr[i];
        ptr->forward = alloc_ptr;
        alloc_ptr += ptr->size;
    }
    return ptr->forward;
}
```

Université de Montréal

Stop&Copy

```
stop&copy () {
   alloc_ptr = scan_ptr = alloc_new_heap ();
   for varptr in roots
      *var = copy (*var);
   while (scan_ptr < alloc_ptr) {
      for i = 0 to scan_ptr->size
         ptr[i] = copy (ptr[i]);
      scan_ptr += scan_ptr->size;
   }
   free_old_heap ();
}
```

Université de Montréal

Comptage de références

Convention d'appel: incrément soit par l'appelant soit par l'appelé

Avant de libérer un objet, il faut décrémenter tous les pointeurs contenus

Synchronisation en cas de concurrence

affectation: incrémenter *avant* de décrémenter

Cher en taille de code et temps d'exécution

Taille des compteurs

Taille d'un compteur: un mot, quelques bits, 1 bit

1 bit suffit pour tous les objets qui n'ont qu'un seul pointeur

En cas de dépassement:

- Compteurs à saturation: lorsqu'on atteint le maximum, on y reste
  Ces objets ne peuvent plus être récupérés, à moins d'un GC
- réserve auxiliaire de grands compteurs

Université de Montréal

Réduire les mises à jour de compteurs

Ne pas compter les pointeurs de la pile [Deutsch&Bobrow, 1976]:

- Augmente le nombre d'objets où 1bit suffit
- Un compteur à 0 met l'objet sur une liste de candidats
- Scan occasionnel de la pile pour trouver quels candidats libérer

Mise à jour occasionnelle des compteurs [Levanoni&Petrank, 2001]:

- au lieu de *inc*(*newX*); *dec*(*oldX*), sauve *oldX* ainsi que $\&X$.
- La liste des $\&X$ peut être compactée
- Plus besoin de synchronisation dans le mutateur
- Ça s'appelle une *barrière en écriture* (*write barrier*)

Collecter les cycles

Utiliser un GC de temps à autres:

- Peut aussi servir à mettre à jour les compteurs saturés
- Ou à compacter le tas pour éviter la fragmentation

Détecter les cycles morts [Martínez&Wachenchauzer&Lins, 1990]:

- Décrément à une valeur autre que 0 $\Rightarrow$ candidat
- Prendre les candidats comme des “racines négatives”;
  Décrémenter transitivement les compteurs des objets pointés;
  Récupérables: les objets qui finissent avec un compteur à zéro;
  Ré-incrémenter transitivement les compteurs depuis les survivants

Université de Montréal

Finalization, pointeurs faibles

Les systèmes à base de GC offrent souvent la possibilité de détecter quand un objet est désalloué:

- *Finaliseur*: fonction à exécuter avant de désallouer un objet
- *Pointeur faible*: pointeur qui n'empêche pas le GC de désallouer l'objet pointé. À chaque usage, il faut vérifier s'il est NULL

D'abord, marquer les objets sans suivre les pointeurs faibles;

ensuite, sauver la liste des objets finalisables non-marqués
et mettre à NULL les pointeurs faibles sur des objets non marqués;

puis, marquer à nouveau, depuis les objets à finaliser;

finalement, appeler chacun des finaliseurs (dans quel ordre?)

GC++

GC *incrémental*: la phase de collection est saucissonnée

GC *concurrent*: collecteur et mutateur peuvent rouler simultanément

GC *parallèle*: le collecteur est divisé en sous-processus parallèles

GC *partitionné*: le tas est partitionné de sorte qu'une collection puisse ne récupèrer les déchets que dans une partition à la fois

GC *générationnel*: GC partitionné selon l'âge des objets

GC *distribué*: GC où le tas est distribué sur plusieurs machines

GC *temps-réel*: être mythique aux caractéristiques peu comprises, qui semblent inclure des bornes sur l'usage CPU du GC et sur le temps pendant lequel le mutateur peut être bloqué par le GC

Besoins d'un GC

Quand exécuter le GC: GC-check

Introspection (aussi utile pour runtime-typecheck, et débogage):

- Racines: liste des variables globales, et celles de la pile
- Trouver la taille des objets, distinguer les pointeurs
  - tagbits, descripteurs de type
- Idem pour les blocks d'activations de la pile:
  - safe points, stack maps

Un bit par objet (mark ou forward), voire plus pour concurrence

Barrière en écriture ou en lecture pour algorithmes plus sophistiqués

GC-check

Parfois basé sur le temps: nécessite un tas illimité

Vérification régulière de l'espace restant

- À chaque allocation
- Entrée des fonctions et à chaque tour de boucle (si allocation)
  - Déterminer allocations maximales par boucle/fonction
  - Toujours assurer e.g. 4KB lors du test
    - Ajouter un test si l'allocation maximale est plus grande

Polling régulier, aussi utile pour d'autres fins

- Stack overflow, réaction aux signaux externes (e.g. barrier)

Université de Montréal

Tags et descripteurs

Déterminer la taille des objets

Déterminer où sont les pointeurs

- E.g. un seul bit de tag “pointeur/autre”

Peut être fourni au GC séparément

Encodages variés

- Fonction/méthode de marquage
- Vecteur de descripteurs
- Bitmap

Impact sur la vitesse du GC

Université de Montréal

Description de la pile et des registres

Convention de description

- E.g. Pointeurs dans registres r0-r8
- Frames d'activation avec header, comme des objets
- Impossible d'obéir 100% tout le temps
  - Besoin de safe-points

Stack map

- Table indexée par le PC
- Décrit l'état des registres et de la pile
- Peut couvrir 100% des cas (et même éliminer les tags)

GC conservateur

Présumer que ce qui ressemble à un pointeur l'est

- Pas besoin de distinguer pointeurs des autres valeurs
- Déterminer si une valeur est un pointeur valide
- Besoin de découvrir la taille des objets

Peut-être utilisé seulement pour certaines parties de la mémoire

- Typiquement utilisé pour la pile

Empêche de déplacer les objets (stop&copy ou compaction)

Université de Montréal

GC incrémental ou concurrent

Coloration d'objet: blanc = non-vu, gris = vu, noir = scanné

Déplacer une référence d'un objet non-*noir* à un objet *noir*?

- La référence pourrait ne pas être vue par le collecteur!

Deux approches:

- *read barrier*: noircir les objets avant de les lire
  - généralement coûteux sans soutient matériel
- *write barrier*: éclaircir les objets après modification
  - moins d'écritures que de lectures
  - asynchrone et se prête à diverses optimisations

Université de Montréal

GC incrémental ou concurrent (exemple)

Mutateur: `x->f := y` devient `x->f := y; flag (x);`

Collecteur:

```
stop_the_world ();
read_roots ();
unstop_the_world ();
mark_everything ();
while (! empty (flagged))
  for (p in flagged)
    unflag (p); mark (p);
sweep ();
```

Université de Montréal

GC partitionné

Tas divisé en partitions plus ou moins indépendantes

Faire le GC d'un sous-ensemble P des partitions

- Autant que possible ne scanner que les objets de P
- Besoin de connaître les références *entrantes*
  - Gestion de *remembered set*
- Cycles irrécupérables si pas entièrement dans P

Différents algorithmes de GC pour des partitions différentes

Write barrier pour maintenir à jour le *remembered set*

GC générationnel

Haute mortalité infantile

$\Rightarrow$ GC efficace d'une partition *nursery*

Généralisation à plusieurs *générations*

Remembered set petit: peu de références de vieux à jeune

tenure policy: quand déplacer les objets à une autre génération

`stop&copy` populaire pour la *nursery*

- Allocation très rapide
- GC efficace si la proportion de déchets est élevée

Taille de la *nursery* souvent choisie sur la base d'un cache!

Write barriers

Opération d'ajout dans un ensemble; rapidité cruciale

Pas nécessaire pour toutes les mutations

- Choix de filtrer avant d'ajouter à l'ensemble ou après

Précision de l'ensemble

- Adresses exactes: coûteux si l'ensemble est grand
- Card marking: un booléen par *carte* de 2^n bytes
  - `flag(x)` devient `cards[x >> n] = true;`
  - `char` volontiers utilisé comme booléen (atomique et rapide)

Découplage par un *log*: `flag(x)` devient `log[h++] = x`

Université de Montréal

Compile-time GC

Déterminer quels objets on peut allouer sur la pile

Idée de base: référence qui est *escaping*

- Stockée dans un objet *escaping* ou global
- Passée à une fonction
- Renvoyée par la fonction

Rafinement: arguments de fonction qui ne sont pas *escaping*