Compilatoare

Garbage Collection

CPL
IR crunch

Heap Management

- Memoria libera – de obicei o lista inlantuita de blocuri marcate libere.
- Alocarea – alegerea unui bloc liber (first fit, best fit), si alocarea unei parti din el.
  - Blocurile alocate sunt si ele adaugate intr-o lista
- Eliberarea – marcarea blocului ca liber.
  - De exemplu trecerea din lista de blocuri alocate in lista de blocuri libere.
  - Blocurile libere consecutive trebuie unite
  - Explicita (ceruta de programator) sau implicita
- Cat dureaza operatiile?
- Problema: fragmentarea – daca obiectele au dimensiuni diferite.

CPL IR crunch

Garbage collection

- Termenul de **colectare a memoriei disponibile** (**gc**) se referă la algoritmii de eliberare implicită a memoriei dinamice (sau altfel spus de colectare a zonelor de memorie devenite inaccesibile).
- Zonele care pot să fie eliberate (**garbage**) sunt zone de memorie la care nu se mai poate ajunge prin intermediul unui pointer sau eventual a unei succesiuni de pointeri accesibili. Despre aceste zone se spune că sunt **inaccesibile** spre deosebire de zonele care sunt accesibile şi despre care se spune ca sunt **în viaţă.**
- Iniţial aceste tehnici au apărut în legătură cu limbajele de tip **Lisp** pentru care alocarea memoriei se face implicit. În prezent se încearcă utilizarea acestor tehnici şi pentru limbajele care utilizează alocarea explicită a memoriei dinamice (**C**, **C++**). Limbaje mai noi cum este **Java** au fost proiectate pentru a putea să utilizeze această tehnică.

Avantaje / Dezavantaje

- Reducerea efortului de programare:
  - Prevenire memory leaks
  - Prevenirea dealocarii premature a memoriei (mai ales cand mai multe module folosesc un obiect)
- Timp de rulare suplimentar
  - Problematic in special in real-time

- O solutie alternativa sunt ‘memory pools’.

CPL IR crunch

Functionare GC

- GC se executa de regula cand nu mai este memorie
- Trebuie să rezolve două probleme:
  - identificarea garbage
    - Conservativ! Daca nu suntem siguri, e “in viata”
  - eliberarea spaţiului ocupat de garbage
- Identificarea zonelor de memorie "în viaţă" se face pornind de la variabilele accesibile atunci când se execută **colectarea memoriei**.
  - Var. globale, var. locale din stiva curenta, registre -> mulţime rădăcină (**root**).
  - Pornind de la mulţimea rădăcină şi parcurgând obiectele accesibile prin intermediul unor pointeri se pot identifica toate obiectele accesibile. Tot ce nu este accesibil în acest fel reprezintă zona inaccesibilă (**garbage)**.

CPL IR crunch

Functionare GC(2)

- În vederea identificării zonelor accesibile trebuie să existe o strategie pentru a răspunde la două întrebări:
  - dându-se un obiect, acesta conţine pointeri ?
  - dându-se un pointer unde este începutul şi sfârşitul obiectului spre care indică pointer-ul ?
- In Lisp – simplu (RTTI, pointeri doar implicit); in C, foarte complicat
  - Exemplu - aritmetica pe pointeri
  - Conservative collector (Boehm-Demers-Weiser)

Tipuri de algoritmi GC

- Secventiali
  - Un singur thread, ‘stop-the-world’.
- Paraleli
  - Ruleaza simultan pe mai multe procesoare.
  - GC nu devine o problema pentru scalabilitate
- Incrementali
  - GC in paralel cu executia programului.
  - Limiteaza timpul petrecut intr-un pas de GC.
- Cu compactare / copiere
  - Reduc fragmentarea memoriei
  - Cresc gradul de localitate a datelor
  - Alocare rapida (incrementare de pointer)

CPL IR crunch

Algoritmi secventiali de GC

- **Reference Counting** – numara referintele la un obiect
  - Nu poate elibera structuri ciclice.
- **Mark and Sweep** - parcurge toata zona de memorie dinamică (**heap**) identificând zonele care nu mai sunt în viaţă după care le eliberează.
  - Nu realizează compactarea spaţiului disponibil
  - CMS – Compacting M&S – compactarea e un pas separat.
  - Pt. compactare rapida: "Handles" – exista un singur pointer catre un obiect. Referintele sunt indecsi in array-ul "Handles".
- **Copying Collection** - copiază zonele ramase în viaţă într-un spaţiu nou.
  - Implicit realizează şi compactarea spaţiului disponibil.

Algoritmi Mark&Sweep

- Algoritmii din aceasta clasă presupun parcurgerea tuturor lanţurilor posibile de pointeri accesibili şi marcarea zonelor de memorie indicate de acestea (**Mark**). Este ca şi cum s-ar turna vopsea prin pointeri şi zonele de memorie utilizate (accesibile) devin colorate.
- După ce se realizează această operaţie se parcurge întreaga zonă heap şi se realizează înlănţuirea zonelor de memorie nemarcate care vor forma spaţiul disponibil (**Sweep**).
- Acest tip de algoritmi este "vechi". Primele implementări care utilizau această clasa de algoritmi au apărut la începutul anilor '60.

Algoritmul

```
new(A){
    if (freeList este goala){
        mark&sweep()
        if (freeList este goala)
           return (“out of memory”)
     }
     pointer = allocate(A)
     return pointer
}

mark&sweep(){
   for p in root
       mark(p)
   sweep()
}
```

```
mark(Obiect){
    if (marc(Obiect) == nemarcat){
         marcheaza Obiect
         for d in descendentii (Obiect)
             mark(d)
    }
}

sweep(){
    p = bazaHeap
    while (p < topHeap){
        if (marc(p) == nemarcat)
           free(p)
        else{
           sterge marcaj p
           p = p + size(obiect p)
        }
    }
}
```

CPL IR crunch

CPL IR crunch

Algoritmul, iterativ

- Problema cu stiva utilizata dinamic – deja memoria e la limita
- Putem folosi un algoritm iterativ unde stiva e deja prealocata

```
Mark&sweep(){
   stivaMark = null
   for obiect referentiat din root{
        marcheaza obiect
        push(obiect, stivaMark)
   }
    marcheazaHeap()
    sweep()
}
```

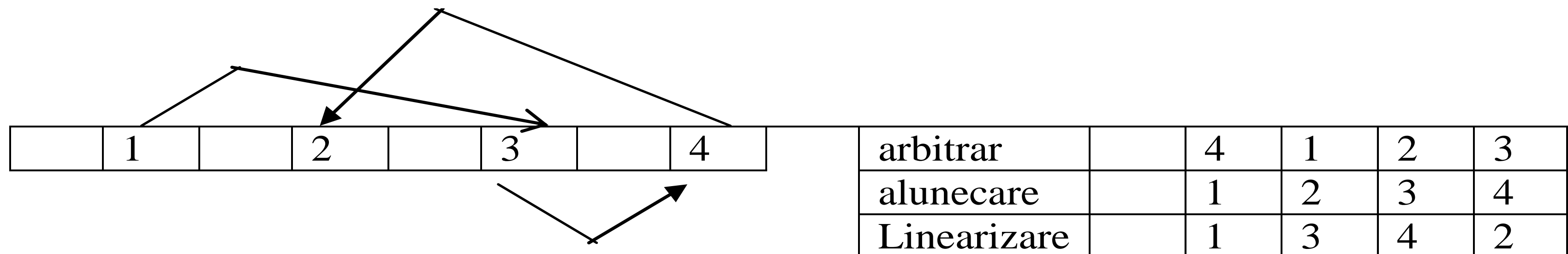
```
marcheazaHeap(){
   while (stivaMark != empty){
      obj = pop(stivaMark)
      for d in descendenti(obj){
        if(marc(d) == nemarcat){
          marcheaza d
          push (d, stivaMark)
        }
      }
   }
}
```

Probleme Mark&Sweep

- Fragmentarea memoriei dinamice - dacă obiectele alocate sunt de dimensiuni foarte diferite şi alocarea se face într-o secvenţa nefavorabilă se poate ajunge în situaţia ca deşi spaţiul total disponibil este suficient pentru o cerere de alocare, totuşi aceasta să nu poată să fie satisfăcută;
- '**mark**' presupune numai parcurgerea zonelor de memorie în viaţă, durata depinde de numărul obiectelor alocate "în viaţă"; in schimb **sweep** presupune parcurgerea întregii zone heap. Rezultă că durata execuţiei acesteia depinde de dimensiunea zonei de memorie dinamice care poate să fie mult mai mare decât partea utilă . Acest aspect poate să limiteze semnificativ performanţele algoritmilor de tip **Mark and Sweep**.
- Obiectele alocate în memoria dinamica nu se mută -> obiecte create la începutul execuţiei programului ajung "vecine" cu obiecte create mult mai târziu. În acest mod localitatea referinţelor este distrusă (apar probleme de perf. intr-un sistem cu memorie virtuala)

Variatiuni Mark&Sweep

- Fragmentarea se poate rezolva
  - Mai multe liste de ‘spatiu liber’, dupa dimensiuni; se aloca ‘best fit’
  - Compactarea zonelor disponibile vecine
  - Se aloca pagini suficient de mari ca sa tina orice obiect (dar se iroseste multa memorie…)
- Mark-Compact - compactarea spaţiului disponibil prin "alunecarea" zonelor "în viaţă" peste zonele inaccesibile(rezolva fragmentare, localitate dar mareste timpul de executie). Variante:
  - arbitrar – nu există nici o garanţie a ordinii obiectelor
  - alunecare - se face alunecarea pentru a păstra ordinea iniţială de alocare
  - liniarizare – obiectele sunt mutate conform modului în care se referă unul la altul



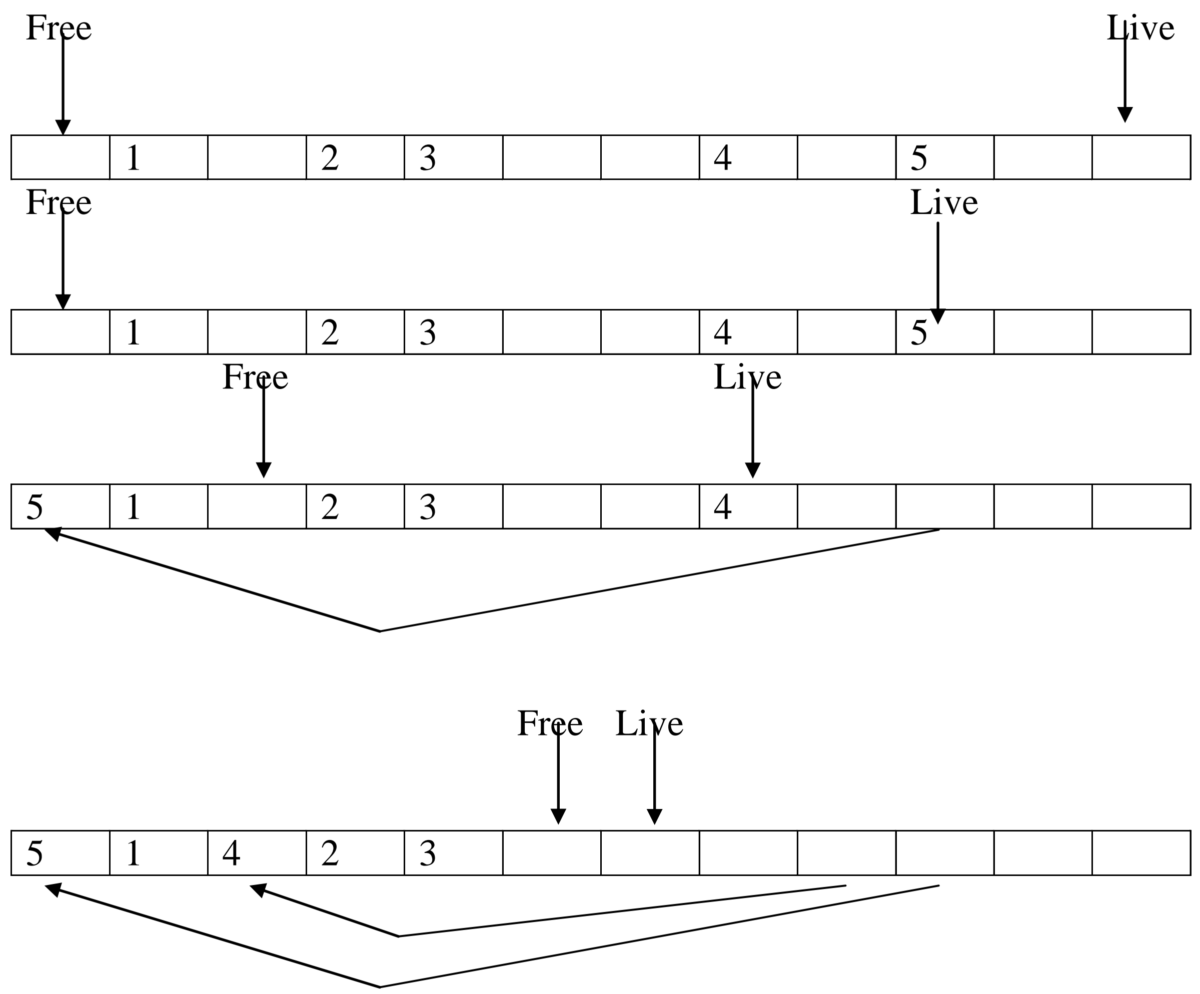
arbitrar		4	1	2	3
alunecare		1	2	3	4
Linearizare		1	3	4	2

CPL IR crunch

Algoritmul ‘two fingers’

- Varianta de mark&sweep, se poate utiliza daca toate obiectele au aceaşi dimensiune.
- Algoritmul are doi pasi - in primul pas se face compactarea in al doilea pas se face actualizarea pointerilor.
- Se utilizează doi pointeri; free care parcurge heap-ul de la limita de pornire cautând poziţii libere, live parcurge heap-ul de la capăt spre început căutând obiecte în viaţă. Când free găseşte o poziţie liberă şi live a găsit şi el un obiect în viaţă se face deplasarea obiectului. După ce se face mutarea o referinţă la noua poziţie este memorată în vechea locaţie.
- În pasul al doilea se parcurg obiectele live, dacă ele indică spre zona liberă se face corecţia corespunzătoare.
- Avantaj: simplu, nu necesita spatiu suplimentar
- Dezavantaj: ordinea arbitrara, distruge localitatea; o singura dimensiune de obiecte (se pot utiliza mai multe zone de heap pt. dimensiuni diferite)

Exemplu – ‘two fingers’



CPL IR crunch

Reference counting

- Se păstrează contoare de utilizare pentru fiecare obiect.
  - De fiecare dată când un obiect este referit de un pointer contorul este incrementat
  - De fiecare dată când un pointer este distrus contorul obiectului spre care acesta indică este decrementat.
- Dacă un contor a ajuns la zero înseamnă că obiectul respectiv nu mai este accesibil.
- Obiectul poate să fie trecut imediat în lista spaţiului disponibil sau se poate face o fază de măturare în care se caută obiecte cu contor zero.
- Pt. accelerare **Mark-and-Sweep** sau separat.

Reference counting (2)

- Probleme:
  - structurile ciclice nu ajung la zero;
  - menţinerea contoarelor măreşte timpul de execuţie. Dacă de exemplu se execută o instrucţiune de forma **p = q** (unde **p** şi **q** sunt pointeri) atunci trebuie să se mărească contorul pentru obiectul indicat de **q** şi să se decrementeze contorul pentru obiectul indicat anterior de **p**.
- Se poate combina cu execuţia periodică a unui algoritm **Mark-and-Sweep** - se poate limita valoarea contoarelor. Dacă se ajunge la limita maximă atunci contorul nu mai este nici incrementat şi nici decrementat, în acest mod pentru obiectele des referite se limitează numărul de operaţii suplimentare. Prin execuţia ulterioară a algoritmului **Mark-and-Sweep** se va parcurge toată memoria şi se vor identifica atât structurile ciclice cât şi obiectele cu contor blocat.

Algoritmi Copy Collection

- Problema: algoritmii de tip **Mark-and-Sweep** ating şi zonele inaccesibile
- CC - memoria dinamică este împărţită în două zone. Se face alocarea de memorie într-o singură zonă (numită **from-space**) până când aceasta se umple. În acest moment se declanşează execuţia algoritmului. Acesta va copia toate zonele de memorie accesibile în a doua zonă (numită **to-space**), care nu va mai contine si 'garbage'-ul. În continuare cele două zone îşi schimba rolurile.
  - Copierea zonelor alocate din zona **from** în zona **to** se poate face în orice ordine.
- Tot vechi – prima implementare Misky (pt lisp) in anii '60 (folosea un fisier, nu era foarte eficient)

CPL IR crunch

Algoritmul Cheney pt. CC

- Spre zona **to-space** indică doi pointeri numiţi **scan** şi respectiv **next**.
- Fiecare obiect accesibil poate să fie referit de către mai mulţi pointeri din obiecte diferite – trebuie actualizati pointerii; se memoreaza noua adresa (din 'to-space') la vechea adresa (in 'from-space'). Această adresă se numeşte **forwarding** pointer.
- Se foloseste o fct. 'forward' care intoarce tot timpul valoarea din to-space pt. un pointer
- Algoritmul are două faze.
  - În prima fază obiectele accesibile direct din **root** sunt mutate în zona **to-space**. Cu această ocazie în copiile vechi ale obiectelor se memorează adresele în zona **to-space**. La rândul lor obiectele mutate în zona **to-space** pot să conţină pointeri către alte obiecte din zona **from**.
  - A doua fază parcurge obiectele care sunt conţinute între adresele indicate de către pointerii **scan** şi **next** şi tratează pointerii conţinuţi în aceste obiecte. În urma acestei parcurgeri se vor copia noi obiecte în **to-space** sau se vor actualiza pointerii care indică spre obiecte conţinute deja în **to-space**.

CPL IR crunch

Algoritmul (pseudocod)

```
forward(p) {
  if p indică spre from-space
     if p.f1 indica spre to-space
        return p.f1
     else{
        *next = *p // copiere de obiect
        p.f1 = next
        next = next + dim(*p)
        return p.f1
     }
  else
     return p
}
```

```
### MAIN:
scan = next = începutul zonei to-space
for each registru r din root {
  r = forward(r)
}
while scan < next {
  for fiecare camp fi al obiectului *scan
    scan.fi = forward(scan.fi)
  scan = scan + dim(*scan)
}
```

CPL IR crunch

Probleme cu alg. Cheney

- Formularea originala face o trecere BFS – distruge localitatea datelor
- Se poate face o trecere DFS – dar avem nevoie din nou de stiva
- Se poate copia doar obiectul + descendentii imediati

Non-Copying Implicit Collection(Baker)

CPL IR crunch

- In loc să se facă o mutare fizica a obiectelor dintr-o zona în alta se mută pointerii la obiecte între două liste.

- Fiecare obiect are trei câmpuri suplimentare invizibile pentru programul care se execută. Două dintre ele sunt utilizate pentru ca obiectul să fie legat într-o listă dublu înlănţuită. Al treilea câmp indică lista la care este conectat obiectul.

- trei liste: o listă a spaţiului disponibil, o listă **from** şi o listă **to**

- Alocarea de memorie -> elemente din lista spaţiului disponibil se mută în lista **from.**Cand epuizam prima lista, se declanseaza algoritmul de GC

Non-Copying Implicit Collection(Baker)

CPL IR crunch

- Algoritmul de colectare a memoriei -> mută obiectele în viaţă din lista **from** în lista **to**. Când toate obiectele accesibile au fost mutate, lista **from** conţine numai pointeri spre obiecte care nu mai sunt în viaţă. Lista **from** devine o listă a spaţiului disponibil. Ca şi în algoritmul clasic cele doua liste îşi schimbă acum rolurile. Vechea listă **to** care conţine acum numai obiecte accesibile devine lista **from** la care se adaugă tot ce se alocă nou.

- Execuţia copierii pointerilor se face într-o manieră similară cu cea a algoritmului Cheney.
- Similaritati Mark & Sweep ?

- Dezavantaj - e posibil să apară fragmentarea
- Avantaj – viteza (nu se fac copieri, valorile pointerilor 'vizibili' nu se schimba) ceea ce simplifică rolul compilatorului.

CPL IR crunch

Eficienta M&S vs CC

- Pentru orice algoritm de colectare a memoriei, timpul suplimentar consumat se datorează următoarelor acţiuni:
  - identificarea mulţimii **root**
  - costul alocărilor
  - costul detectării memoriei devenite disponibile

- Prima componentă nu depinde de metoda de colectare utilizată. Depinde numai de caracteristicile programului care se execută.

- Al doilea cost depinde de numărul de obiecte alocate.

- Ultimul cost este proporţional cu numărul total al obiectelor din memorie (**Mark-and-Sweep**), sau cu numarul obiectelor în viaţă (**Copy Collection**).

Eficienta M&S vs CC

- Un algoritm pentru care timpul este proporţional cu memoria alocată (**Mark-and-Sweep**) poate să fie competitiv cu cele la care timpul este proporţional cu numarul de obiecte in viata (**Copy Collection**), deoarece timpul necesar pentru atingerea obiectelor care nu mai sunt în viaţă poate să fie compensat de timpul de copiere al obiectelor.

- Eficienţa celor două clase de algoritmi depinde de comportarea reală a programelor:
  - raportul între numărul de obiecte în viaţă şi cele inaccesibile în momentul în care se declanşează execuţia algoritmului;
  - costul execuţiei marcării (**mark)** unui obiect relativ la costul copierii unui obiect (**c2** faţă de **c3**).

- Algoritmii bazaţi pe copiere sunt de preferat pentru cazul în care există obiecte mici care trăiesc puţin. Se poate utiliza o soluţie în care obiectele mari sunt puse într-o zonă în care se face Mark and Sweep.

CPL IR crunch

Alte criterii de selecţie

- Utilizarea algoritmului de copiere evită fragmentarea.
- Algoritmii care nu presupun copierea pot fie utilizaţi pentru limbaje care lucrează explicit cu pointeri şi pentru care pot sa apară situaţii în care în faza de execuţie nu se poate decide în mod univoc dacă o valoare reprezintă sau nu un pointer.

CPL IR crunch

Algoritmi incrementali de GC

- Intreruperile necesare GC sunt inacceptabile intr-un sistem de timp real - se face colectarea incremental
- Doua procese – “mutator” (programul) si “colector” (GC) -> problema de consistenta a datelor
  - M&S – cititori-scriitor (doar mutatorul modifica pointerii)
  - CC – mai multi scriitori
- Consistenta relaxata – aproximatie conservativa a grafului de referiri (obiectele care devin inaccesibile dupa ce au fost ‘vazute’ de colector)

Marcajul tricolor

- Notatie folosita pentru sincronizare.
- Obiectele pot să fie colorate cu o culoare din trei posibile: alb, gri, negru.
- **Alb** = marcajul la inceputul ciclului de colectare.
  - La sfârşitul ciclului de colectare toate obiectele inaccesibile rămân albe.
- **Negru** = obiectul este în viaţă la sfârşitul ciclului de colectare
- **Gri** = obiectul a fost identificat ca fiind în viaţă, dar obiectele la care se poate ajunge utilizând câmpurile obiectului respectiv nu au fost încă parcurse. La CC – gri sunt obiectele dintre **scan** şi **next**.

- Indiferent de tipul de tipul de algoritm utilizat **colectorul** trebuie să respecte condiţia - nici un câmp dintr-un obiect negru nu conţine un pointer către un obiect alb.

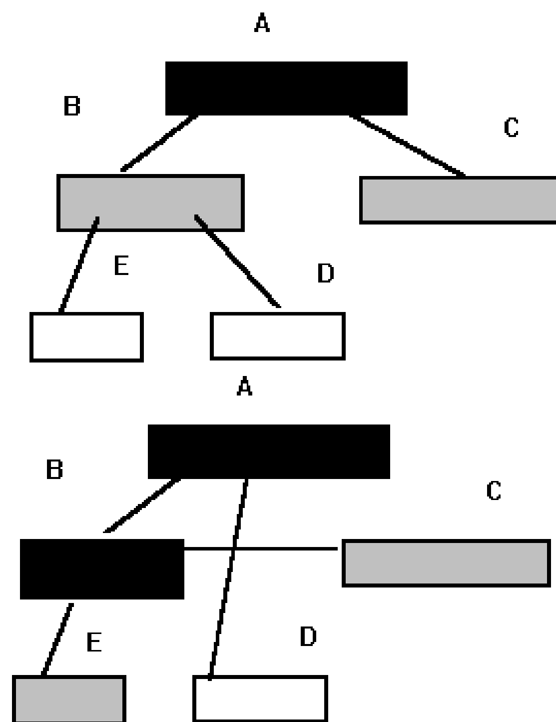
CPL IR crunch

Marcajul tricolor

- **Colectorul** realizează traversarea grafului de obiecte în viaţă şi le schimbă culoarea.
- **Mutatorul** poate să modifice obiectele care au fost deja tratate.
  - Nu poate să facă dintr-un obiect inaccesibil un obiect accesibil şi nici să modifice câmpuri din interiorul unui astfel de obiect
  - Un obiect care a fost deja marcat în viaţă poate să devină inaccesibil
  - Un câmp (pointer) dintr-un obiect care a fost deja tratat poate să fie modificat.
- Prima situaţie poate să fie ignorată, considerarea unui obiect inaccesibil ca fiind în viaţă este conservativă şi obiectul respectiv va fi identificat ca inaccesibil la următoarea trecere a algoritmului.
- Probleme - modificarea câmpurilor în obiectele care au fost deja tratate.

Coordonarea mutator-colector

- Coordonarea între **mutator** şi **colector** presupune existenţa unui mecanism prin care
  - **mutatorul** să fie împiedicat să acceseze un obiect alb sau
  - să fie împiedicat să scrie valoarea unui pointer către un obiect alb într-un obiect negru.
- În primul caz se utilizează o **barieră la citire**,(detecteaza daca mutatorul incearca sa utilizeze un pointer la un obiect alb) .Acesta poate fi vopsit în gri pentru că acum "se ştie" că obiectul este accesibil, dar nu se ştie cum sunt descendenţii acestuia.
- În al doilea caz se utilizează o **barieră la scriere**.(inregistreaza scrierile de pointeri in obiecte)



CPL IR crunch

Bariere la scriere

- În cazul algoritmilor care nu realizează copierea se utilizează barierele la scriere (nu se pune problema ca **mutatorul** să citească un pointer incorect).

- Exista 2 tipuri de bariere la scriere
  - Snapshot-at-beginning
    - Se salveaza o copie a tuturor pointerilor inlocuiti la atribuiri. Valorile salvate se adauga la **root**.
    - Toate obiectele accesibile la inceputul ciclului vor fi negre.
    - Obiectele nou create in timpul unui ciclu sunt negre. De ce?
  - Incremental update
    - Se detecteaza cand pointerii sunt scrisi in obiecte negre; daca pointerul indica spre un obiect alb, se coloreaza in gri obiectul negru.
    - Posibila implementare: se reparcurg obiectele negre din paginile de memorie marcate "dirty"

Bariera la scriere tip 'incremental update' (Dijkstra)

CPL IR crunch

- Algoritmul nu se uita la pointerii modificati din obiectele care nu au fost analizate inca, ci doar marcheaza ca 'gri' obiectele 'negre' in care au fost modificati pointerii, si le reanalizeaza la sfarsitul ciclului de colectare.
- Obiectele nou create sunt 'albe'
  - Avantaj: pot 'muri' inainte de a apuca sa fie analizate de GC fiindca majoritatea obiectelor au 'viata' destul de scurta.
  - Dezavantaj: daca rămân totusi accesibile, trebuie să fie traversate –ceea ce presupune operaţii în plus faţă de **snapshot-at-beginning** (unde obiectele sunt create negre, direct).
- Dacă într-un obiect negru este memorat un pointer la un obiect alb
  - obiectul negru este colorat în gri (va fi reparcurs in final, dar poate obiectul alb 'dispare' pana atunci), <u>sau</u>
  - obiectul alb este făcut gri (solutie mai rapida - se parcurg mai puţine obiecte).

Algoritmii Baker cu bariera la citire

CPL IR crunch

- Copiere incrementala (bazat pe Cheney)
  - Op. atomica - se invalideaza obiectele din 'from-space', se copiaza 'root-set'-ul in 'to-space' (practic 'from-space'-ul devine alb, root setul devine gri). Apoi, se colecteaza incremental("background scavenging")
  - Daca mutatorul citeste un pointer catre from-space, obiectul respectiv este copiat din from-space in to-space (marcat 'gri'), si pointerul actualizat.
  - Obiectele nou alocate sunt alocate in to-space (sunt 'negre')
- Implementarea barierei
  - Software - compilatorul trebuie să genereze pentru fiecare referinţă la un pointer un cod corespunzător
  - Hardware - pentru maşini dedicate, de ex. folosind memorie virtuala. Cum?

Performanta alg. Baker

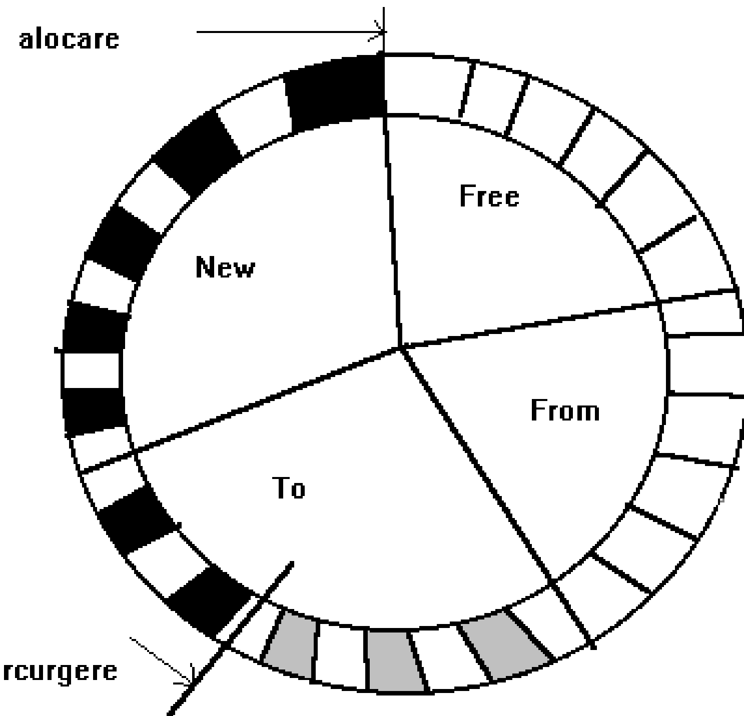
- Utilizarea unei bariere la citire este în general destul de ineficientă deoarece presupune că pentru fiecare referire de pointer se face un test referitor la zona în care este conţinut obiectul respectiv. Dacă obiectul este într-o zonă de tip **from** se va declanşa operaţia de copiere a obiectului respectiv în zona **to**.

- Se poate utiliza şi un sprijin din partea compilatorului care poate să identifice accese care se referă la câmpuri din acelaşi obiect şi să optimizeze pe această bază codul generat.

- Alg. Baker e conservativ – obiectele noi sunt negre, deci chiar daca mor imediat, nu vor fi sterse decat la urmatorul ciclu de colectare

Treadmill

- In loc sa se copieze fizic, se foloseste o lista ciclica (4 liste dublu inlantuite)
- Zona **new** conţine obiectele nou alocate.
- Zona **from** conţine pointerii la obiectele care au fost alocate înainte de începerea ciclului de colectare
- Zona **to** e zona in care se face colectarea
- Zona **free** e zona spatiului liber/a obiectelor dealocate



Treadmill

- La începutul ciclului de colectare lista **new** este goală şi alocarea se face prin mutarea limitei dintre zona **free** şi **new**. Obiectele noi se consideră că sunt negre (deci accesibile) şi fac parte din zona **new**.
- Zona **from** conţine pointerii la obiectele care au fost alocate înainte de începerea ciclului de colectare şi asupra cărora se execută de fapt algoritmul de colectare. Pe măsură ce se execută ciclul de colectare obiecte din lista **from** sunt mutate în lista **to**.
- La începutul ciclului lista **to** este goală. Această listă creşte prin adăugarea de elemente preluate din lista **from**. În zona **to** există atât obiecte negre (pentru care au fost trataţi şi descendenţii) şi obiecte gri care sunt accesibile (fiind mutate în zona **to**) dar pentru care urmează să se cerceteze şi descendenţii.
- Pentru sincronizare se folosesc bariere la scriere (cum?)

CPL IR crunch

Treadmill

- Similar algoritmului Cheney va exista un pointer care delimitează obiectele care au fost parcurse împreună cu descendenţii lor (negre) de cele pentru care urmează abia să se cerceteze descendenţii.

- Când toate obiectele accesibile au fost mutate în zona **to** şi au devenit negre înseamnă că ciclul de colectare a fost parcurs. Ceea ce a rămas în lista **from** reprezintă pointeri spre obiecte inaccesibile - deci se poate face eliberarea memoriei corespunzătoare acestor obiecte. Această eliberare se face prin concatenarea listelor **from** şi **free.**

- Listele **to** şi **new** conţin pointeri spre obiecte accesibile şi deci se pot combina formând noua lista **from**.

CPL IR crunch

Replication copy collection

- **mutatorul** "vede" numai obiecte din zona **from.** În timp ce **mutatorul** lucrează pe aceste obiecte, **colectorul** construieşte copii în zona **to**. Când toate obiectele au fost mutate **mutatorul** va vedea numai zona **to** care devine **from**.
- Deoarece **mutatorul** lucrează pe obiecte nemodificate nu este necesară execuţia nici unui test asupra pointerilor prelucraţi (ei nu pot să indice decât în zona **from**). Pe de altă parte este necesară utilizarea unei bariere la scriere astfel încât copiile din zona **to** să reflecte schimbările din zona **from**.
- Spre deosebire de algoritmul Baker, pentru care modificările obiectelor se fac numai în zona **to**, aici este posibil să se modifice un obiect din zona **from** care are deja o copie neagră în zona **to**. Rezultă că este necesară memorarea tuturor modificărilor, urmând ca atunci când se face comutarea zonelor să se facă şi actualizările corespunzătoare.
- Această variantă de algoritm poate să fie foarte costisitoare daca se produc modificări frecvente ale pointerilor şi deci bariera la scriere va fi utilizată des. Pentru limbaje cum este **ML** pentru care nu apar multe efecte laterale algoritmul poate să fie foarte interesant.
- Oricum, in general traversarile de pointeri sunt mai frecvente decat scrierile de pointeri…

CPL IR crunch

Utilizarea generatiilor

- Generational GC incearca sa beneficieze de o proprietate observata empiric a obiectelor alocate:
  - Majoritatea obiectelor traiesc foarte putin, si doar o mica parte traiesc perioade mai lungi
  - (sau reformulat) Daca un obiect supravietuieste la una/cateva colectari, exista mari sanse sa supravietuiasca vreme indelungata.
- Obiectele 'cu viata lunga' incetinesc in mod nenecesar GC – atat cele M&S cat may ales cele pe baza de 'copy collection'.
- Tehnicile pe baza de generatii impart heap-ul in mai multe sub-heap-uri si separa obiectele pe sub-heap-uri in functie de 'generatia' fiecarui obiect. Obiectele noi sunt alocate intr-un subheap dedicat. Cand nu mai exista memorie, se scaneaza doar primul subheap – iar majoritatea obiectelor vor fi probabil dealocate. Subheap-urile cu generatii mai mare sunt scanate mai putin frecvent.
- De vreme ce se scaneaza fragmente mici de heap si se recupereaza (proportional) mai mult spatiu, eficienta e imbunatatita.

CPL IR crunch

Cand se face mutarea

- Câte cicluri de colectare trebuie să fie supravieţuite de către un obiect pentru a fi mutat într-o generaţie mai veche ?
  - Daca se pastreaza un singur ciclu, se poate folosi CC si se pot copia direct in generatia ‘mai batrana’ obiectele care supravietuiesc
  - Dar un obiect cu viata scurta care a fost creat chiar inainte de colectare, va avansa degeaba la o generatie mai mare

CPL IR crunch

Cand se face mutarea

- Ungar: Prima generatie poate avea heap-ul impartit in trei zone: una pentru obiectele nou create, celelalte fiind zonele 'from' si 'to'
  - Se face astfel diferenta intre obiectele foarte noi si cele mai vechi din generatia curenta
  - O varianta a alg. Ungar tine doar 2 zone: "de memorare" si "de alocare". Obiectele care sunt colectate din zona de memorare se copiaza intr-o generatie veche, iar cele din zona de alocare se copiaza in zona de memorare.

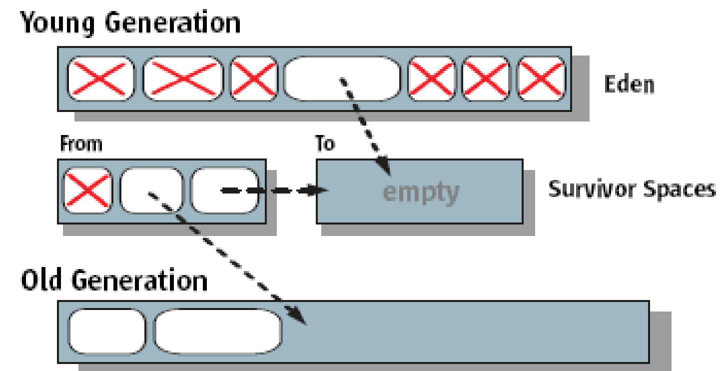


Figure 3. Serial young generation collection

Sursa: java.com whitepaper:
Memory Management in the Java HotSpot™ Virtual Machine

CPL IR crunch

Pointeri intre generatii

- Problema: pointer dintr-o generatie veche catre obiect dintr-o generatie noua
- Solutie: bariere la scriere similare celor utilizate în cazul algoritmilor de alocare incrementali.
  - Pentru orice operaţie de modificare a unui câmp de tip pointer trebuie să se facă o verificare pentru a stabili dacă nu cumva este vorba de un pointer de la un obiect dintr-o generaţie mai veche la un obiect dintr-o generaţie mai nouă.
  - Pointerul respectiv va trebui să fie utilizat în mulţimea rădăcină pentru generaţia nouă.
  - Abordarea este conservativă (obiectul dintr-o generaţie mai veche datorită căruia se păstrează un obiect dintr-o generaţie mai nouă poate să nu mai fie accesibil).

Pointeri intre generatii

- Solutie: folosirea memoriei virtuale (LISP: Symbolics)
  - În loc să se înregistreze obiectele care conţin pointeri între generaţii se înregistrează paginile din memoria virtuală care conţin astfel de pointeri.
  - Granularitatea utilizată este la nivel de pagină.
  - Timpul pentru parcurgerea setului înregistrat va depinde de numărul de pagini şi de lungimea paginilor şi nu de numărul de obiecte în care s-au scris pointeri.

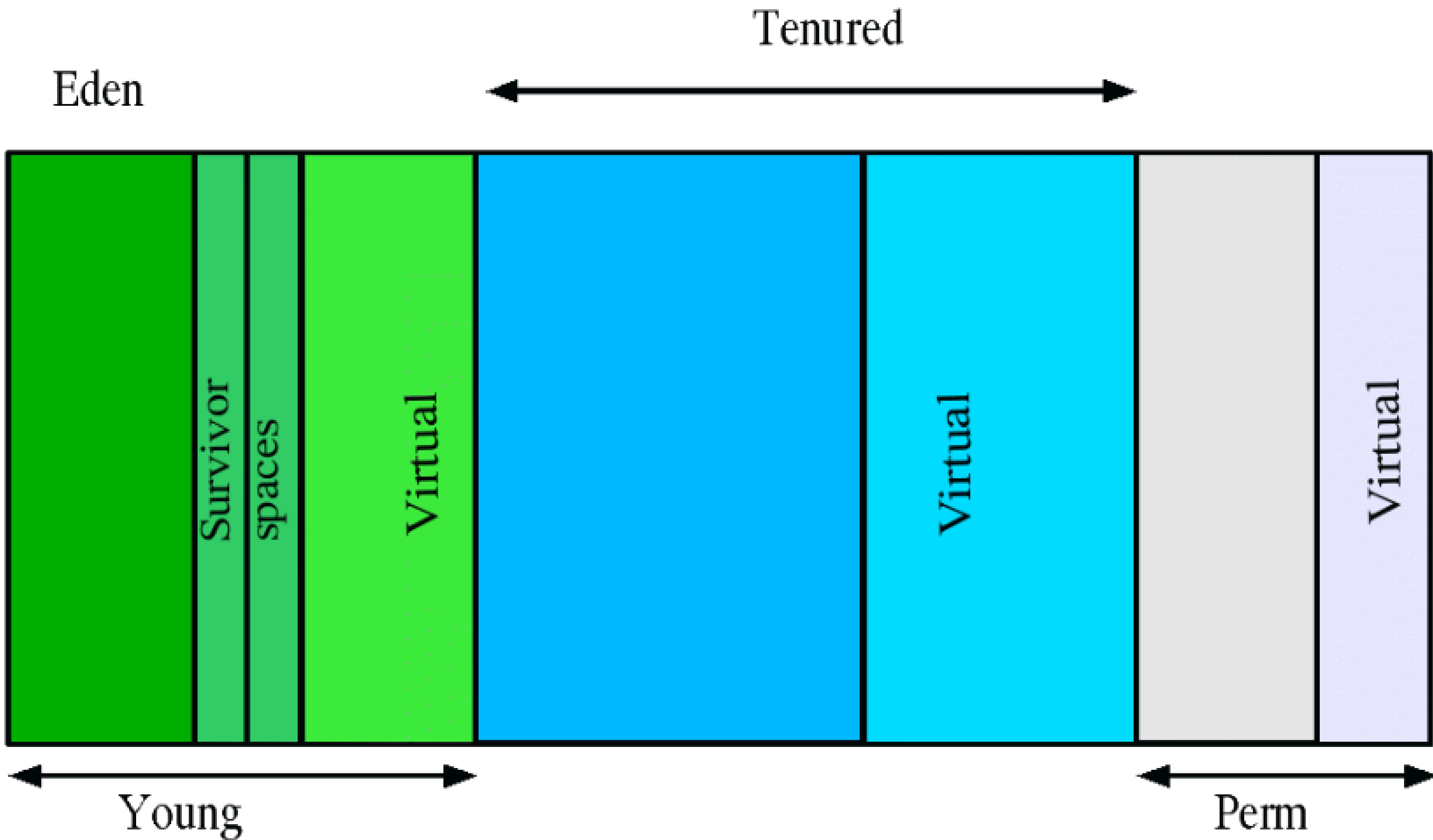
- Solutie: înregistrarea pointerilor intr-o lista înlănţuita de adrese. (Standard ML)
  - Dezavantajul: valori duplicate in lista.
  - Timpul de colectare este proporţional cu numărul de memorări de pointeri şi nu cu numărul de obiecte în care se face memorarea

Ce foloseste Java ~~Sun~~ Oracle JDK?

CPL IR crunch

- Nu foloseste 'reference counting'. In rest... mai toate.
- Primele JDKuri foloseau un mark-sweep sau mark-compact single-threaded.
- Experienta a aratat ca mark-compact merge bine, fiindca obiectele 'vechi' tind sa se acumuleze la baza heap-ului si sa nu mai fie copiate ulterior
- JDK 1.2 a introdus o abordare hibrida – colectarea pe generatii, folosind algoritmi diferiti.

Memoria in Java



CPL IR crunch

Ce foloseste Java?

- 2 Generatii (3, respectiv 2 zone).
  - GC minor (1 generatie), major (toata memoria)
  - Fiecare thread aloca in alta parte din zona de alocare (Local Allocation Buffers - TLAB)
- Selectie de algoritmi diversi
- Dimensionarea generatiilor si algoritmul optimizeaza...
  - **Timpul** (% CPU ocupat de GC)
  - **Pauzele** (oprirea programului pt GC)
  - **Promptitudinea** (timpul pana cand memoria unui obiect inaccesibil e colectata)
  - **Dimensiunea** setului de lucru
- JDK 1.4 – algoritmi paraleli de GC.
  - Amdahl: 1% din timp in GC pe 1 procesor inseamna doar 24x speed-up pentru 32 procesoare.
- JDK 1.5 – alegerea automata a algoritmului in functie de masina pe care se ruleaza.

Algoritmi folositi – Sun JDK 1.5

- Secvential (CC pt. gen 1 / mark & compact pt. gen 2)
- Paralel CC (gen 1) + serial mark & compact
- Paralel CC + Paralel mark & compact
  - Optimizare: compacteaza doar regiunile fragmentate.
- Incremental (CMS: concurrent mark & sweep)
  - Gen 1 – CC secvential
  - Gen 2 - Mark & Sweep (fara compactare)
    - Concurent, cu exceptia pasului de reprocesare a pointerilor din root sau "dirty"
    - Incremental update
    - Bariera la scriere pt. referinte intre generatii implementata sub forma de bit "dirty" pe regiuni de heap.

Algoritmi folositi – JDK 1.6

- Paralel CC + Paralel mark & compact – selectat implicit
- Incremental (CMS)
  - Implementare atat incrementala, cat si paralela
  - Dimensiunile generatiilor sunt variabile.
  - Copierile – ordonate pentru o localitate mai buna (cache friendly)
- Algoritmul ales la rulare in functie de hardware (memorie disponibila, nr. procesoare) si tipul aplicatiei.

Algoritmi folositi – JDK 1.7

- Algoritm incremental nou, cu compactare (G1)
- Heap impartit in regiuni de 1MB, alocate dinamic catre gen 1 sau gen 2.
- Colectarea gen 1 : CC paralel, dar nu concurent
  - Unele regiuni gen 1 devin gen 2
- Gen 2:
  - Marcare concurenta (snapshot-at-the-beginning)
  - Compactare 'stop-the-world'
  - Se compacteaza doar regiunile fragmentate sau goale.
  - Bariera la scriere pt. referinte intre regiuni
- 'Soft real time' – aplica euristici pentru a calcula momentul pauzelor.

See paper: **Garbage First Garbage Collection,** David Detlefs, Christine Flood, Steve Heller, Tony Printezis, Sun Microsystems, Inc