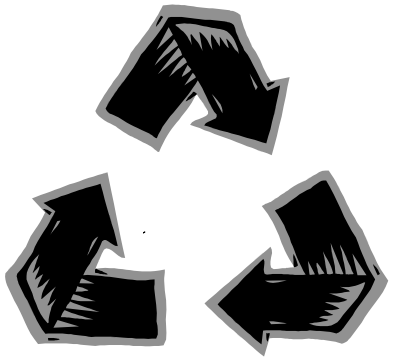
Roskankeruu
Garbage Collection



16.3.2000

Ville Salmensuu
<Ville.Salmensuu@iki.fi>

Sisältö

- Mitä on roskankeruu ja missä sitä tarvitaan?
- Perusalgoritmit
  - viittausten laskenta (Reference Counting)
  - Mark-and-Sweep
  - kopioiva kerääjä, Stop-and-Copy
- Parannuksia
  - iteratiiviset versiot
  - sukupolvittaiset kerääjät
- Konservatiivinen roskankeruu
  - roskankeruun toteutus C- ja C++-ohjelmiin
- Roskankeruun käyttökelpoisuudesta
- Cessun esitys

Muistinvaraus

- Staattinen
  - yksinkertainen ohjelmoijalle ja kääntäjälle (esim. C:n globaalit muuttujat)
  - ei muuttuvankokoisia rakenteita, ei rekursiivisia aliohjelmia, ei dynaamista muistinvarausta
- Pino (stack)
  - rekursiiviset aliohjelmat mahdollisia, aliohjelman sisäiset muuttujat voivat olla vaihtuvan kokoisia
  - pinoon tallennetut arvot eivät säily seuraavaan aktivaatioon, aktivaatiotietue ei säily aliohjelmasta palaamisen jälkeen (edellytys mm. lambda-funktioille)
  - aliohjelma voi palauttaa vain ennalta määrätyn mittaisen paluuarvon
- Keko (heap)
  - muistia voidaan varata ja vapauttaa missä järjestyksessä tahansa
  - mahdollistaa helposti rekursiiviset tietorakenteet (listat, puut, …)
  - muuttuvan kokoiset tietorakenteet
  - aliohjelman aktivaatiotietue voidaan tallentaa
- MITEN HOITAA KEON MUISTINVARAUS?

Eksplisiittinen muistinhallinta (C,C++)

```
char *text;
text = malloc(strlen(”morjens”)+1);
strcpy(text,”morjens”);
free(text);
…
char msg[] = ”terve”;
text = malloc(strlen(”morjens”)+1);
strcpy(text,”morjens”);
text = msg; /* muistivuoto, vanha arvo muuttui roskaksi */
…
text = malloc(strlen(”morjens”)+1);
free(text);      /* text on roikkuva osoitin */
strcpy(text,”morjens”);  /* virhe */
```

Kuka vastaa jaettujen muistilohkojen vapauttamisesta?

Mesa:n kehitysajasta 40% meni muistinhallinnan kanssa

Roskankeruu

- Tekniikka, joka hoitaa keon automaattisen muistinhallinnan (=vapautuksen)
- Helpottaa ohjelmoijan työtä, erityisesti monimutkaisten tietorakenteiden kanssa
  - funktionaaliset ja oliokielet
- Ei sovellu kaikkiin tilanteisiin
  - lisää useimmiten ohjelman ajoaikaa ja/tai muistivaatimuksia
  - ei yleistä algoritmia
- Ensimmäiset algoritmit julkaistu 1960
  - McCarthy, Collins (molemmat Lispiä varten)
  - huono suorituskyky, ”jumiutumiset” => roskankeruu sai huonon maineen
- Roskankeruu on taas muodissa
  - Java (muita: Lisp, Scheme, Smalltalk, Eiffel, Modula-3, ….)
  - trendi: siirrytään parhaasta mahdollisesta suorituskyvystä kohti luotettavia sovelluksia ja nopeaa sovelluskehitystä
  - tehokkaita algoritmeja reaaliaikaisiin, rinnakkaisiin ja hajautettuihin järjestelmiin

Viittausten laskenta (Reference Counting)

- Ensimmäiset SmallTalkit, InterLisp, Modula-2+, MS Component Object Model
- Pidetään yllä tietoa, kuinka monta viittausta on kuhunkin varattuun muistilohkoon (”solu”). Kun soluun ei enää viitata, se voidaan kierrättää
- Free_listiin tallennetaan vapaat solut, niiden RC on nolla
- Kun muutetaan pointteria, täytyy sen aiemmin osoittaman solun RC:tä vähentää yhdellä ja uuden solun RC:tä kasvattaa yhdellä. Jos RC menee nollaan, solu täytyy vapauttaa ja jatkaa rekursiolla
- Ensimmäinen viittausten laskenta-algoritmi: Collins 1960
- [algoritmi ja esimerkki]

Viittausten laskenta (2)

```
allocate() =
    newcell = free_list
    free_list = next(free_list)
    return new_cell
New() =
    if free_list == nil
        abort ”Memory exhausted”
    newcell = allocate()
    RC(newcell) = 1
    return newcell
free(N) =
    next(N) = free_list
    free_list = N
delete(T) =
    RC(T) = RC(T) – 1
    if RC(T) == 0
        for U in Children(T)
            delete (*U)
        free(T)
Update(R, S) =
    RC(S) = RC(S) + 1
    delete(*R)
    *R = S
```

Viittausten laskenta (3)

- Etuja
  - työ jakautuu tasaisesti ohjelman suorituksen ajalle (vrt. tracing collectors), sopii interaktiivisiin ja reaaliaikaisiin ohjelmiin (rekursiivinen versio ei aivan täysin)
  - hyvä lokaalisuus l. kerääjän ei tarvitse käydä läpi kuin vapautettavat solut (cache)
  - roskat tulee kerättyä välittömästi (Singapore-malli)
    - paikalla päivitykset (in-place updates), Glasgow Haskell
    - finalisoijan/destruktorin välitön kutsuminen (C++, Java)
- Haittoja
  - vie paljon prosessoriaikaa (ilman rautatukea)
  - vaatii tiukkaa sitomista ajettavaan ohjelmaan tai kääntäjään
  - vaatii ylimääräistä tilaa joka solua kohti
  - ei kykene vapauttamaan syklisiä rakenteita
    - usein käytetään apuna jäljittävää (tracing) kerääjää

Mark-and-Sweep

- McCarthy, 1960. Ensimmäinen roskankeruualgoritmi, jäljittävä
- käyttäjän ohjelma (mutator) keskeytetään muistin loppuessa
- jäljittää juurisolmuista lähtien kaikki käytössä olevat solut ja merkitsee ne
- lopuksi käy läpi koko keon ja vapauttaa merkitsemättömät solut
-
```
mark_sweep() =
    for R in Roots
        mark(R)
    sweep()
    if free_pool is empty
        abort ”Memory exhausted”
mark(N) =
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for M in Children(N)
            mark(*M)
sweep() =
    N = Heap_bottom
    while N < Heap_top
        if mark_bit(N) == unmarked
            free(N)
        else mark_bit(N) = unmarked
        N = N + size(N)
```

Mark - Sweep (2)

- Miranda, konservatiiviset roskankerääjät (C:tä varten)
- Etuja
  - syklisten rakenteiden vapautus onnistuu luonnostaan
  - pointterimanipulaatioilla ei lisäkustannusta
  - pienempi CPU-kustannus kuin viittausten laskennalla
- Haittoja
  - käyttäjän ohjelma pysähtyy roskankeruun ajaksi
    - 1980-luvun alussa suuret Lisp-ohjelmat keräsivät roskia 25-40% ajastaan ja käyttäjät joutuivat odottelemaan 4.5 sekuntia joka 79:s sekunti
    - ei sovellu sellaisenaan aikakriittisiin järjestelmiin
  - asymptoottinen kompleksisuus O(M), jossa M on koko keon koko
  - fragmentoi muistia => cache-ongelmat, muistinvaraus hidastuu
  - elävän muistiaineksen pitää olla pienempi kuin keon koko, muuten roskia joudutaan keräämään jatkuvasti

Stop-and-Copy

- Kopioivat keräilijät jakavat keon kahteen saman kokoiseen puolitilaan (semi-space)
- Toisessa on elävä data, toisessa vanhentunut data
- Keräys alkaa vaihtamalla puolitilojen roolia
- Jäljitetään kaikki *Fromspace*:n elävät solut, ja kopioidaan ne *Tospace*:en
- Solu kopioidaan vain yhden kerran, samalla vanhaan paikkaan jätetään ”osoitteenmuutosilmoitus” (forwarding address, broken heart)
- Kun kopioitava solu viittaa soluun, joka on jo kopioitu, kirjoitetaan uuteen kopioon viitatun solun forwarding address
- Ainoastaan elävät solut *Fromspace*:ssa käydään läpi (vrt. Mark-Sweep)

Stop - Copy (2)

```
• init() =
      Tospace = Heap_bottom
      space_size = Heap_size/2
      top_of_space = Tospace + space_size
      Fromspace = top_of_space + 1
      free = Tospace
  New(n) =
      if free + n > top_of_space    flip()
      if free + n > top_of_space    abort ”Memory exhausted”
      newcell = free
      free = free + n
      return newcell
  flip() =
      Fromspace, Tospace = Tospace, Fromspace
      top_of_space = Tospace + space_size
      free = Tospace
      for R in Roots                R = copy(R)
  copy(P) =                          -- P osoittaa sanaan, ei soluun
      if atomic(P) or P == nil      return P
      if not forwarded(P)
              n = size(P)
              P’ = free
              free = free + n        -- varaa tilaa Tospacesta
              temp = P[0]            -- tähän tulee forw.address
              forwarding_address(P) = P’
              P’[0] = copy(temp)
              for i = 1 to n-1      P’[i] = copy(P[i])
      return forwarding_address(P)
```

Stop - Copy (3)

- Etuja
  - muistinvaraus äärimmäisen halpaa: ylivuototarkistus ja pointterin kasvatus
  - ei fragmentointia
  - helpompia toteuttaa kuin Mark-Sweep-kerääjät
- Haittoja
  - ei-kopioiviin kerääjiin verrattuna kaksinkertainen muistin tarve
  - jos molemmat puolitilat eivät mahdu keskusmuistiin, tulee enemmän sivunkorvauksia (page faults)
  - toisaalta fragmentoituminen lisäisi muistin tarvetta
- Suoritusaika O(R), jossa R on elävien solujen määrä
- Jos vain pieni osa soluista selviää elävinä roskankeruuseen asti, Stop-and-Copy on periaatteessa tehokkaampi kuin Mark-and-Sweep
- Roskankerääjän toteutus ja ajettavan ohjelman muistikäyttäytyminen vaikuttavat tehokkuuteen usein enemmän kuin algoritmin periaate ja kompleksisuuden raja-arvo
  - keon lineaarinen skannaus usein nopeampaa kuin solujen kopiointi

Ehtoja roskankeräykselle

- Kopioivan roskankerääjän täytyy saada ohjelman juuripointterit, jotta se voi löytää kaikki elävät solut
- Sen täytyy tietää tarkasti missä kohtaa solussa on osoittimia
  - jos luulee sanaa virheellisesti osoittimeksi, kirjoittaa siihen virheellistä dataa
  - jos ei huomaa osoitinta, sen osoittama solu jää tutkimatta ja osoitin päivittämättä
- Solujen pituus on myös tiedettävä
- Mark-Sweep-tyyppiset roskankerääjät eivät vaadi yhtä täsmällisiä tietoja, koska soluja ei siirretä
- Konservatiiviset roskankerääjät toimivat ilman kääntäjän tukea
  - yrittävät päätellä bittimössöstä pointtereita
  - voi käyttää C- tai C++-ohjelmassa kirjastona
  - käyttää jotain mark-sweep-algoritmia

Kehittynyt viittausten laskenta

- Iteratiivinen toteutus
  - laiska vapautus: solun lasten vapautus siirretty New-aliohjelmaan
  - vaati vakiomäärän muistia
  - keräyksen kustannus jakautuu vieläkin tasaisemmin
- Viivästetty viittausten laskenta
  - normaalisti ohjelman käydessä listaa läpi, jokainen iteraattoripointterin muuttaminen maksaa
  - manuaalisessa viittausten laskennassa aliohjelmakutsujen sisällä voidaan jättää laskenta tekemättä, jos tiedetään, että summat eivät muutu => vähemmän CPU-aikaa, enemmän debuggausaikaa
  - kääntäjä voi myös optimoida turhaa laskentaa pois
  - Deutsch-Bobrow-algoritmi: jätetään paikallisten osoittimien ja kääntäjän apumuuttujien aiheuttamat laskurimuutokset tekemättä. Luku tarkoittaa nyt vain viittauksia keon muista soluista. Nollaan osoituskertaan putoavat solut lisätään taulukkoon (ZCT = Zero Count Table), jonka sisältö tarkistetaan ajoittain katsomalla, pääseekö pinon osoittimista niihin. Jos ei pääse, ZCT:n solu voidaan poistaa

Kehittynyt viittausten laskenta (2)

- viittauslaskurin kokoa voidaan pienentää (jopa yhteen bittiin), koska useimpiin soluihin ei viitata kuin yhdestä toisesta solusta
-
```
incrementRC(N) =
    if RC(N) < sticky
        RC(N) = RC(N) + 1
decrementRC(N) =
    if RC(N) < sticky
        RC(N) = RC(N) – 1
```
- laskurit saadaan taas ennalleen käyttämällä apuna jäljittävää kerääjää
- syklisten rakenteiden vapauttamiseen voidaan käyttää myös jäljittävää kerääjää, tai erityisesti tätä varten suunniteltuja algoritmeja
  - Christopher, Lins

Kehittynyt Mark-Sweep

- Iteratiivisia versioita
  - Mark:in havaitsemat uudet merkitsemättömät solut tallennetaan pinoon
  - Käännetään solujen väliset pointterit ympäri => ylimääräistä tilaa ei tarvita
- Pienten solujen mark-bitit voidaan tallentaa yhteen isoon bittikarttaan
  - kerääjän muistikäyttäytyminen lokalisoituu
- Laiska mark-sweep
  - jos vapaa muisti loppuu, tehdään mark
  - sweepataan sitä mukaa kuin muistia varataan
- Mark-Compact
  - poistetaan fragmentointi kompaktoimalla keko
  - solujen järjestys muuttuu a) satunnaiseksi b) viittausjärjestykseen c) ei ollenkaan (säilyy luontijärjestykssä)
  - c-tapa säilyttää lokaalisuuden parhaiten
  - kahden sormen algoritmi => solut satunnaiseen järjestykseen

Kehittynyt kopioiva kerääjä

- Cheneyn algoritmi
  - iteratiivinen, tarvitaan vain kaksi osoitinta
  - kokonaan käsitellyt solut ovat mustia, käsittelemättömiä osoittimia sisältävät solut harmaita, ja käsittelemättömät solut valkoisia
  - `flip() =`

```
 Fromspace, Tospace = Tospace, Fromspace
 scan = free = Tospace
 for R in Roots
        R = copy(R)
 while scan < free
        for P in Children(scan)
               *P = copy(*P)
        scan = scan + size(scan)
copy(P) =
 if forwarded(P)
        return forwarding_address(P)
 else
        addr = free
        move(P,free)
        free = free + size(P)
        forwarding_address(P) = addr
        return addr
```

Kehittynyt kopioiva kerääjä (2)

- On hyödyllistä käsitellä suuret ja pitkäikäiset solut poikkeuksellisesti esim. jakamalla keruualue palasiin
- Jos tiedetään, että solussa ei ole yhtään osoitinta, se kannattaa jättää kokonaan skannaamatta (esim. isot bittikartat)
- Toisaalta isot oliot voidaan käyttöjärjestelmän muistinhallinnan avulla sijoittaa omille sivuilleen ja mäpätä nopeasti Tospaceen
- Kun uusi solu varataan levylle swapatusta Tospacen sivusta, siellä on roskaa, joka turhaan ladataan muistiin
- Tämä saattaa johtaa sellaisen Fromspacen sivun swappaukseen levylle, jossa myöskin on roskaa
- Käyttöjärjestelmälle olisi siis syytä kertoa roskankerääjän toimista
- Välimuistin tehokkuuteen vaikuttaa mihin muotoon solut tallennetaan
  - depth first, breadth first, hierarchical

Sukupolvittainen keruu

- Nykyisin yleisesti käytetty tekniikka, toteutetaan useimmiten kopioivana
- On todettu, että vastikään varatut solut hylätään todennäköisemmin kuin vanhat solut (Weak generational hypothesis)
- Jaetaan siis kerättävä muistialue sukupolviin
- Tehdään roskankeruuta etupäässä nuorimman eli ensimmäisen sukupolven soluille (pieni keruu, minor collection), ja ”ylennetään” niitä ajan kuluessa vanhempiin sukupolviin
- Vanhempia sukupolvia kerätään harvemmin
  - suuri keräys, major collection, joka tehdään järjestyksessä nuorimmasta vanhimpaan, useampia sukupolvia kerrallaan
- Ongelmana isot juurijoukot sekä sukupolvien väliset osoittimet, joista täytyy pitää kirjaa (remembered sets)
  - tarvitaan write barrier mutatoriin, jotta kerääjä havaitsee vanhempiin sukupolviin tehtävät mutaatiot
- Koska keräyksiä tehdään sukupolvi tai jokunen kerrallaan, voi vanhempiin sukupolviin jäädä pitkäksikin aikaa hylättyjä soluja (tenured garbage)

Konservatiivinen keruu

- Myös C- ja C++-ohjelmissa voi käyttää roskankeruuta
  - erikoistapauksena esim. Lispin kääntäminen C-kielelle käyttäen apuna roskankeruukirjastoa
- malloc- ja free-kutsut korvataan kutsuilla roskankeruukirjastoon
- Kirjasto ei voi luottaa kääntäjän apuun keruussa, joten sen täytyy tehdä päätökset vapautettavista soluista konservatiivisesti
- Käytetään mark-sweep-algoritmia, koska soluja ei uskalla siirtää
- Pinoa ja rekisterien sisältöä tutkimalla selvitetään elävät solmut
  - päätellään, onko kokonaisluku osoitin
    - osoittaako kekoon, jo tunnettuun soluun tai tunnetun tyypin koon kerrannaiseen
- Muistivuoto on mahdollista
- C++:ssa voidaan hyödyntää osoitinoperaatioiden ylilatausta (smart pointers)
- Roskankeruu toimii varsin tehokkaasti myös muokkaamattomasta C-koodista [Zorn 1992, Detlefs, 1994] (luokkaa 20% hitaampi keskimäärin, suuri vaihtelu)

Inkrementaalinen keruu

- Rinnakkaisessa tai reaaliaikaisessa järjestelmässä roskankeruusykliä ei voi tehdä alusta loppuun yhdellä kertaa
  - tarkoituksena pienentää pysähtymisaikoja
- Rinnakkain ajettavan mutatorin toimintaa voidaan kontrolloida luku- tai kirjoitusesteillä (barrier)
  - havaitaan osoitinmuutokset ja estetään mutatoria saamasta vääriä osoitteita
  - hidastaa mutatorin toimintaa
- Kova reaaliaikajärjestelmä vaatii (pientä) ylärajaa roskankeruun viemälle ajalle
- Sukupolvittainen keruu on inkrementaalista, mutta ylärajaa ei aina voi taata
  - jos suuret keräyksiä joudutaan tekemään liian usein
- Nilsen kehittänyt muistimoduleita, jotka keräävät roskia rautapohjaisesti

Roskankeruun käyttökelpoisuudesta

- Roskankerääjien suorituskyky on parantunut alkuajoista huomattavasti
  - sukupolvittaisilla roskankerääjillä päästään useimmiten lähelle manuaalisen muistinhallinnan nopeutta
- Roskankeruualgoritmi on valittava ongelman mukaan, jotta saadaan paras tehokkuus
- Tietorakenteiden monimutkaistuessa roskankeruu tulee yhä houkuttelevammaksi
  - suoraviivaisissa ongelmissa roskankeruu on turhaa
  - roskankeruu vähentää rajusti muistinhallinnan virhealttiutta
- Roskankeruu toimii sitä tehokkaammin, mitä enemmän on muistia käytössä

Lisätietoa

- Jones Richard, Lins Rafael: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. 1996
- Röjemo Niklas: Garbage collection, and memory efficiency, in lazy functional languages. 1995, väitöskirja
- Oksanen, Kenneth: Real-time Garbage Collection of a Functional Persistent Heap. 1999, lisensiaattityö
- Wilson Paul: Uniprocessor Garbage Collection Techniques. 1994, survey
- `http://reality.sgi.com/boehm/gc.html`
  Roskankerääjä C- ja C++-ohjelmiin