Automaattinen muistinhallinta

Timo Tapanainen (ttapanai@cs.helsinki.fi)

Helsinki 12. huhtikuuta 2004

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Dynaamisen muistin (keon) hallinta on monissa systeemi tason ohjelmointikielissä jätetty ohjelmoijan vastuulle. Ohjelmoijan täytyy eksplisiittisesti vapauttaa käyttämätön varattu muisti. Tällaista muistinhallintaa, jossa ohjelmoija on vastuussa muistinhallinnasta, kutsutaan mm. eksplisiittiseksi muistinhallinnaksi. Eksplisiittinen muistinhallinta vaatii ohjelmoijalta paljon ja siinä epäonnistuminen johtaa vaikeasti löydettäviin virheisiin.

Varatun muistin vapauttamatta jättäminen johtaa muistivuotoon. Muistivuoto tarkoittaa sitä että muistin allokoija ei pysty tarjoamaan ohjelmalle vuotanutta muistia uudelleen, jolloin vaarana on muistin loppuminen. Muistin liian aikainen vapauttaminen johtaa taas *roikkuvaan viitteeseen* (dangling reference), jossa ohjelmalla on viite jo vapautettuun muistialueeseen. Roikkuvan viitteen käyttö johtaa odottamattomiin muutoksiin toisaalla ohjelmassa jos tuo vapautettu muistialue on otettu ohjelmassa uudelleen käyttöön. Muistivuotojen ja roikkuvien viitteiden korjaaminen on todella hankalaa niiden epäsäännöllisyyden vuoksi. Ne voivat ilmaantua esim. vasta kun ohjelma on ollut käynnissä kauan tai kun ohjelma on siirretään toiseen ympäristöön.

Vaikka ohjelmoija muistaisikin vapauttaa varatun muistin, on tehtävä vielä päätös milloin ja missä tuo vapauttaminen tehdään. Koska ohjelmat ovat nykyään modulaarisia ei voida välttyä siltä että moduulien välillä välitetään olioviitteitä. Jotta muisti voitaisiin vapauttaa on tiedettävä varmasti että millään moduuleista ei enää ole viitettä vapautettavaan muistialueeseen. Näin muistin vapauttajan on tunnettava muiden moduulien toiminta jotta vapauttaminen olisi turvallista. Tämä lisää moduulien välistä riippuvuutta ja johtaa joskus myös hyvän ohjelmointitavan rikkomuksiin. Olioiden kopioiminen moduulien välillä on näistä yksi.

2 Automaattinen muistinhallinta

Automaattisessa muistinhallinnassa ohjelmoija on vapautettu dynaamisen muistin hallinnasta. Ohjelmoijan ei tarvitse vapauttaa varaamaansa muistia, vaan tämä hoidetaan automaattisesti ajoaikaisen ympäristön toimesta. Muistin vapauttamisesta vastaa niin sanottu *roskienkerääjä* (garbage collector). Roskienkerääjän toiminta

perustuu keon objektien saavutettavuuteen eli siihen onko ohjelman nykytilasta vielä mahdollista päästä käsiksi keon objekteihin. Kaikki saavuttamattomissa olevat objektit ovat roskia ja ne voidaan poistaa roskienkerääjän toimesta. Roskienkerääjän toiminta voidaan jakaa kahteen loogiseen vaiheeseen. Ensimmäisessä vaiheessa roskienkerääjä erottelee saavutettavat objektit saavuttamattomista ja toisessa vaiheessa se poistaa saavuttamattomissa olevat objektit.

Seuraavissa luvuissa tarkastellaan erilaisia roskienkeruu -algoritmeja. Luvussa kolme esitellään algoritmi jossa objektien saavutettavuus perustuu viitelaskureihin. Luvussa neljä tarkastellaan algoritmeja jotka päättelevät saavutettavuuden jäljittämällä. Luvussa viisi ja kuusi tarkastellaan mahdollisia laajennuksia roskienkeruu algoritmeihin.

3 Roskienkeruu viitelaskurilla

Viitelaskuri tekniikassa objektien saavutettavuus päätellään objektiin liitetyn viitelaskurin perusteella. Viitelaskuri kertoo kuinka monta viitettä ohjelmasta on kyseiseen objektiin. Kun objektiin osoittava viite luodaan tai kopioidaan, kasvatetaan viitelaskuria ja vastaavasti viitteen poistaminen vähentää laskuria. Kun laskuri saavuttaa nollan, voidaan objekti poistaa. Ennen objektin poistamista on kuitenkin mukautettava sen viittaamien objektien viitelaskureita. Poistaminen voi siis johtaa lumivyöryn omaiseen objektien poistamiseen.

Viitelaskureihin perustuvilla roskienkerääjillä on monia hyviä ominaisuuksia. Niiden toiminta on tyyliltään inkrementaalista, missä roskien kerääminen suoritetaan pienissä erissä. Tällöin roskien keruu on nopeaa ja se aiheuttaa yleensä vain pienen viiveen ohjelman suoritukseen. Tämä on toivottu ominaisuus etenkin reaaliaika sovelluksissa, missä pyritään suorituksen ennustettavuuteen.

Viitelaskureiden käytössä on myös omat ongelmansa. Viitelaskuri tekniikalla ei pystytä vapauttamaan syklisiä rakenteita ja sitä vaivaa tehottomuus lyhyissä metodeissa. Välitettäessä kekoon osoittavia parametreja, on viitelaskureiden arvoja kasvatettava. Kun metodi päättyy, on näiden arvoja taas vähennettävä. Tämä voi aiheuttaa suhteettoman suuren kustannuksen muuten nopeissa metodeissa. Tähän on ehdotettu ratkaisuksi viivästettyjä viitelaskureita.

4 Jäljittävä roskienkerääjät

Jäljittävässä roskienkeruussa objektien saavutettavuus päätellään ohjelman muistialueista tutkimalla. Muistialueista etsitään kekoon osoittavia viitteitä objektien saavutettavuuden selvittämiseksi. Etsintä aloitetaan niin sanotusta *juuri joukosta* (root set), joka käsittää staattiset muistialueet, pinon ja rekisterit. Kun näistä muistialueista löydetään kekoon osoittava viite, merkitään viitattu objekti saavutetuksi ja jatketaan muiden viitteiden etsintää juuri joukosta tai saavutetuista objekteista. Kun kaikki viittaukset on käyty läpi, voidaan kaikki saavuttamattomat objektit poistaa.

Jäljittävillä roskienkerääjillä saavutettavien objektien löytäminen perustuu yllä mainittuun tapaan. Erot löytyvät lähinnä siitä miten saavuttamattomien objektien poistaminen tapahtuu. Seuraavissa aliluvuissa tarkastellaan jäljittäviä roskienkerääjiä, joilla on erilainen tapa roskien poistamiseen [Wil94].

4.1 Merkitse ja lakaise

Merkitse ja lakaise (mark-sweep) roskienkerääjä on yksinkertaisin jäljittävistä roskienkerääjistä. Merkintä vaiheessa kaikki saavutetut objektit merkitään lakaisu vaihetta varten. Lakaisu vaiheessa koko keko käydään läpi ja kaikkien merkitsemättömien objektien tila vapautetaan. Vapauttaminen tarkoittaa käytännössä sitä että vapautettava muistialue on jotenkin merkittämä allokoijan vapaiden muistialueiden listaan.

Merkitse ja lakaise algoritmin suurimmat vaikeudet liittyvät muistin allokoijan toimintaan. Allokoijan on tasapainoiltava nopeuden ja muistin pirstoutumisen suhteen.

4.2 Merkkaa ja tiivistä

Merkitse ja tiivistä (mark-compact) algoritmi ratkaisee merkkaa ja lakaise -algoritmin allokoijaan liittyvät ongelmat. Kun saavutettavat objektit on merkitty, tiivistetään keko kopioimalla merkityt objektit yhtenäiseksi muistialueeksi ja päivitetään muistialueiden viitteet osoittamaan objektien uusiin paikkoihin. Tiivistetty keko ei vaadi yhtä monimutkaista allokoijaa kuin merkkaa ja lakaise -algoritmissa, sillä nyt muistia voidaan varata lineaarisesti tiivistetyn keon päältä. Allokointi on siis erittäin nopeaa ja muisti ei pirstaloidu.

Merkkaa ja tiivistä algoritmin heikkoutena on mahdollinen hitaus. Algoritmi vaatii suoriutuakseen ainakin kolme keon läpikäyntiä. Merkkaus vaiheen jälkeen määritellään objektien uudet osoitteet ja kolmannella kerralla tehdään varsinainen kopiointi ja viitteiden päivitys määritettyjen osoitteiden mukaan.

4.3 Kopioiva kerääjä

Kopioiva kerääjä (copy collector) tiivistää keon kuten merkkaa ja tiivistä -algoritmi, mutta siinä ei käytetä erillistä merkkaus vaihetta. Kopioivaa kerääjää käytettäessä keko on jaettu kahteen yhtenäiseen *puoliavaruuteen* (semispace): *lähdeavaruuteen* (fromspace) ja *kohdeavaruuteen* (tospace). Muistin varaukset tehdään vain lähdeavaruudesta. Kun lähdeavaruudesta ei kyetä enää varaamaan muistia, suoritetaan kopioiva kerääjä. Kopioiva kerääjä kopioi kaikki lähdealueen saavutettavat objektit kohdeavaruuteen. Kopioinnin jälkeen kohdeavaruus asetetaan lähdeavaruudeksi jonka jälkeen voidaan ohjelman suoritusta jatkaa. Kopioiva kerääjä siis käytännössä tiivistää keon puolikkaan keon toiseen puolikkaaseen. Kuten merkkaa ja tiivistä -algoritmissäkin, on ohjelman viitteet päivitettävä osoittamaan keon objektien uusiin paikkoihin.

Yksinkertaisimpia kopioivan kerääjän toteutukset perustuvat Cheneyn algoritmiin [Che70]. Algoritmin ensimmäisessä vaiheessa käydään läpi juuri joukko, josta etsitään viitteitä kekoon. Kun viite löydetään, kopioidaan viitteen osoittama objekti lähdealueelta kohdealueelle, merkitään tuon objektin lähdealueelle *edelleenvälitysosoitin* (forwarding pointer) ja päivitetään viite osoittamaan objektin uuteen osoitteeseen. Edelleenvälitysosoitin sisältää merkinnän objektin siirrosta ja uuden objektin osoitteen. Kun algoritmissa kohdataan edelleenvälitysosoitin, riittää kun käsiteltävä viite päivitetään edelleenvälitysosoittimen sisältämällä osoitteella. Kun ensimmäinen vaihe on suoritettu, on kohdealueelle kopioitu kaikki juuri joukon suoraan osoittamat objektit. Seuraavassa vaiheessa viitteiden etsintää jatketaan kohdealueelta. Kohdealueen alusta edetään muistipaikka kerralla kohti kohdealueen loppua ja kun viite kohdataan, suoritetaan samat toimenpiteet kuin juuri joukon viitteen käsittelyssä. Kun kohdealueen viimeisen objektin viimeinen muistipaikka on käsitelty, on kaikki saavutettavat objektit kohdealueella ja kohdealueen vaihto lähdealueeksi voidaan tehdä.

5 Inkrementaalinen roskienkeruu

Jäljittävät roskienkerääjät suorittavat roskienkeruun kokonaisuudessaan yhdessä vaiheessa. Kun allokoija ei pysty suorittamaan muistin varausta, käynnistetään roskienkerääjä. Roskienkeruun jälkeen allokoijalla on taas riittävästi vapaata muistia varauksen suorittamiseen, jonka jälkeen ohjelman suoritus jatkuu. Roskienkeruun aiheuttama viive on erityisen ongelmallinen reaaliaika sovelluksille. Tämän ongelman ratkaisemiseksi on kehitetty inkrementaalisia roskienkerääjiä, jotka suorittavat roskienkeruun pienissä osissa. Tämä aiheuttaa ohjelmalle useampia pysähdyksiä, mutta nämä ovat lyhyitä.

Inkrementaalisissa roskienkerääjissä ohjelman ja roskienkerääjän toiminta on limittäistä. Tällöin on huolehdittava että ohjelman tekemät muutokset objekteihin huomioidaan roskienkeruussa. Jos esimerkiksi roskienkerääjä on käsitellyt objektin A, jonka jälkeen se siirtyy käsittelemään A:n osoittamia objekteja B ja C. Jossain vaiheessa palataan suorittamaan ohjelmaa, joka muuttaa A:n viittauksen C:hen viittaukseksi muualta saavuttamattomaan objektiin D. Jos roskienkerääjä ei havaitsisi A:n muuttumista, se poistaisi objektin D, jolloin ohjelmalla olisi roikkuva viite. C:n poistamatta jättäminen ei haittaisi sillä se poistettaisiin seuraavalla roskienkeruu kerralla. Roskienkerääjän ja ohjelman toiminnan koordinoimiseen käytetään luku ja kirjoitus esteitä. Käytännössä näitä tekniikoita käyttämällä roskienkerääjä havaitsee ohjelman tekemät muutokset muistialueille.

6 Sukupolvellinen roskienkeruu

Sukupolvelliset roskienkerääjät jakavat keon eri sukupolviin. Uudet objektit allokoidaan nuorimmasta sukupolvesta ja kun tuo sukupolvi on täynnä, suoritetaan sille roskienkeruu. Kun nuorimman sukupolven objektit ovat selvinneet tietystä määrästä roskienkeruuta, siirretään ne seuraavaan sukupolveen. Sukupolvien täyttyminen siis laukaisee roskienkeruun, jolloin on mahdollista että objektit siirtyvät vanhempiin sukupolviin. Keon jakaminen sukupolviin tehostaa roskienkeruuta, koska tarkasteltavana on kulloinkin vain yksi sukupolvi. Lisäksi etuna on lyhyt ikäisten objektien tehokas roskienkeruu, koska nuorin sukupolvi tarkistetaan useimmin. Sukupolvien käyttö on yleistä kopioivissa roskienkerääjissä, koska ne vähentävät kopioinnin määrää.

7 Yhteenveto

Tässä dokumentissa esiteltiin roskienkeruussa käytetyt perusalgoritmit. Viitelaskureihin perustuvat roskienkerääjät päättelevät objektien saavutettavuuden viitelaskureita käyttämällä. Viitelaskureita käyttävien roskienkerääjien toiminta on inkrementaalista mikä on erityisesti reaaliaika sovelluksissa haluttua. Viitelaskureiden suurin heikkous on kyvyttömyys poistaa syklisiä rakenteita. Jäljittävät roskienkerääjät päättelevät objektien saavutettavuuden juuri joukon viitteitä seuraamalla. Näistä merkitse ja lakaise -algoritmi aiheuttaa keon pirstaloitumista ja hidastaa sekä monimutkaistaa allokoijan toimintaa. Kopioivat roskienkerääjät eivät kärsi edellä mainituista ongelmista, mutta niiden toiminta on hitaampaa objektien kopioimisen ja viitteiden päivittämisen vuoksi. Kopioivien roskienkerääjien toimintaa voidaan kuitenkin tehostaa jakamalla keko sukupolviin.

Viitteet

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. Communications of the ACM, 13(11):677–678, 1970.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.