Mikko Nieminen

Muistinsiivous reaaliaikajärjestelmissä

Tietotekniikan
pro gradu -tutkielma
1. syyskuuta 2004

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Mikko Nieminen
Yhteystiedot: nieppa@cc.jyu.fi
Työn nimi: Muistinsiivous reaaliaikajärjestelmissä
Title in English: Garbage collection in real-time systems
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 129
Tiivistelmä: Tutkielmassa esitetään automaattisen muistinhallinnan, muistinsiivouksen, ja reaaliaikajärjestelmien perusteet. Pääpaino on kuitenkin sillä, kuinka näitä menetelmiä tarvitsee muuttaa, jotta muistinsiivousta voidaan käyttää reaaliaikajärjestelmissä. Lisäksi esitetään mitä ongelmia muistinsiivouksen reaaliaikaistaminen tuottaa ja kuinka muistinsiivous liitetään osaksi reaaliaikajärjestelmiä. Tutkielmassa esitetään kuinka näitä teorioita hyödyntämällä muistinsiivous saadaan liitettyä reaaliaikajärjestelmiin siten, että järjestelmän reaaliaikavasteet saadaan tyydytettyä.
English abstract: This thesis presents basic techniques of automatic memory management – garbage collection. Thesis also includes introduction to real-time systems. It is also presented what are required changes to basic techniques so that these techniques could be used in real-time systems and what problems can be expected when garbage collection is used in real-time systems. It is also shown how garbage collection can be scheduled as a real-time task using theories presented in the thesis, so that real-time demands of the system can still be satisfied.
Avainsanat: Muistinsiivous, reaaliaikajärjestelmät, aikataulutus, vähittäinen muistinsiivous, samanaikainen muistinsiivous.
Keywords: Garbage collection, real-time systems, scheduling, incremental garbage collection, concurrent garbage collection.

Sisältö

Kuvat

1 Johdanto

Kysymys, voidaanko muistinsiivousta käyttää reaaliaikajärjestelmissä, on askarruttanut muistinhallinnan tutkijoita lähes yhtä kauan kuin muistinsiivousta on tutkittu. *Muistinsiivous (engl. garbage collection)*, roskien keruu tai automaattinen muistinhallinta, mitä nimeä sitten käytetäänkään, tunnetaan varmasti parhaiten Java-ohjelmointikielen ominaisuutena, mutta tekniikan kehityksen juuret ulottuvat "kaukaiseen menneisyyteen", alukuperäisen LISP-ohjelmointikielen aikoihin 50- ja 60-lukujen taitteeseen. Tämä LISPiä varten kehitettu ensimmäinen muistinsiivoin on John McCarthyn [36] aikaansannos. Nykyään kaikki funktiokielet ja lähes kaikki oliokielet hallinnoivat dyynaamista muistia automaattisesti. Lisäksi muistinsiivoimia on kehitetty myös perinteisille ohjelmointikielille, kuten Pascalille, C:lle ja C++:lle.

Perinteisesti dynaamista muistia käyttävissä ohjelmissa ohjelmoijat ovat olleet vastuussa sekä muistin varaamisesta että vapauttamisesta. Tällaista tapaa hallita muistia kutsutaan manuaaliseksi muistinhallinnaksi. Manuaalinen muistinhallinta on erittäin virhealtista. Lisäksi manuaalinen muistinhallinta joko luo ohjelmamoduulien välille ylimääräisiä kytköksiä tai pakottaa ohjelmoijan luomaan datasta ylimääräisiä kopioita. Muistinsiivous on tekniikka, jossa dynaamisesti varattu muisti vapautetaan järjestelmän toimesta silloin, kun varattu muistialue voidaan turvallisesti vapauttaa. Muistinsiivouksen käyttäminen poistaa roikkuvat osoittimet ja muistivuodot, jotka ovat yleisimpiä virheitä manuaalista muistinhallintaa käytettäessä.

Suomenkielinen termi "(muistin)siivous" ei ole yleisessä käytössä. Tässä tutkielmasssa termiä käytetään historiallisista syistä. Sairanen [48] toteutti vuonna 1984 Jyväskylän yliopiston Tietojenkäsittelyopin laitokselle pro gradu -tutkielman, jossa hän kyseistä termiä käytti. Myös Kaijanaho [27] käytti termiä muistinsiivous Jyväskylän yliopiston Tietotekniikan laitokselle toteuttamassa LuK-tutkielmassaan. Siivous myös mainitaan ATK-sanakirjassa kyseessä olevan asian suomenkielisenä nimenä.

Alkuaikojen muistinsiivoimet siivosivat muistin atomisena operaationa ja siivotessaan ne pysäyttivät järjestelmän varsinaisen toiminnan pitkiksi ajoiksi. Pahimmillaan ohjelmat saattoivat kuluttaa lähes puolet suoritusajastaan muistia siivoten. Vuosien saatossa muistinsiivousmenetelmät ovat kehittyneet ja nopeutuneet. Monet interaktiiviset järjestelmät käyttävät muistinsiivousta, ja tällöin siivouksesta aiheutuvan pysäh-

dyksen on oltava niin lyhyt, ettei järjestelmän käyttäjä sitä huomaa. Interaktiivissa järjestelmissä voidaan käyttää muistinsiivoustekniikkaa, jota kutsutaan vähittäiseksi siivoukseksi. Vähittäisessä siivouksessa muistia siivotaan pienissä inkrementeissä aina, kun ohjelmat varaavat muistia. Tällöin muistinsiivoimen aiheuttama pysähdys jakautuu tasaisemmin järjestelmän varsinaisen suorituksen lomaan. Muistinsiivous on myös varteenotettava vaihtoehto reaaliaikajärjestelmissä, mutta tällöin siivouksen vasteajan on oltava vieläkin lyhyempi ja siivoimen toiminnan ennustettavissa.

Reaaliaikajärjestelmillä tarkoitetaan sellaisia tietojärjestelmiä, joiden oikeellisuutta ei mitata pelkästään järjestelmän laskennan tuloksella, vaan oikeellisuuteen vaikuttaa myös aika, jolloin tämä tulos saadaan. Reaaliaikajärjestelmien on toimittava aina oikein ja tuotettava oikea tulos oikeaan aikaan. Reaaliaikajärjestelmissä järjestelmät mallinnetaan tehtävillä, joille on määrätty keskinäinen prioriteetti ja jokaisella tehtävällä on vasteaikavaatimus. Tehtävät on suoritettava siten, että reaaliaikavasteet saadaan tyydytetyksi. Jotta tehtävät voidaan aikatauluttaa, tehtävien suorituksen on oltava ennustettavissa. Ennustettavuuteen vaaditaan, että tehtävien kaikkien alkeistoimintojen suoritusajat ovat lyhyitä ja rajoitettuja. Suoritusvuoro annetaan sellaiselle tehtävälle, jonka prioriteetti on korkein.

Tämän tutkielman tarkoituksena on selvittää millaisia tekniikoita muistinsiivouksen reaaliaikaistamisessa voidaan käyttää; kuinka muistinsiivoimen toiminnasta saadaan riittävän ennustettava, jotta siivous voidaan irroitaa omaksi reaaliaikavasteet omaavaksi tehtäväkseen, miten tämä muistinsiivoustehtävä saadaan aikataulutettua järjestelmän muiden tehtävien ohella ja millaisia rajoitteita tämä järjestelmän toiminnalle asettaa.

Tutkielma rakentuu kolmesta osasta. Näistä ensimmäisessä esitellään taustatietoja reaaliaikajärjestelmistä, muistinhallinnasta, muistinhallinnan ongelmista sekä näiden ongelmien ratkaisemisesta. Ensimmäinen osa sisältää luvut 2, 3 ja 4. Luvussa 2 käsitellään reaaliaikajärjestelmien perusteita, luvussa 3 tutustutaan muistinhallinan perusteisiin ja luvussa 4 kerrotaan muistinsiivouksen perusalgoritmeistä ja käsitteistä.

Tutkielman toinen osa syventyy muistinsiivouksen reaaliaikaistamiseen, ja se sisältää luvun 5, jossa käsitellään muistinsiivouksen reaaliaikaistamisen liittyviä perusteita, sekä luvun 6, jossa katsotaan sitä kuinka muistinsiivous voidaan suorittaa omana reaaliaikatehtävänään.

Kolmannessa osassa esitellään tutkielman ohessa toteutettu muistinsiivoin sekä analysoidaan tämän siivoimen toimivuutta reaaliaikajärjestelmissä. Tämä osa koostuu luvusta 7. Luku 8 on tutkielman yhteenveto. Liitteessä A on tutkielman yleisimmät ter-

mit ja lyhenteet. Liitteeseen B on koottu tutkielman kaavoissa käytetyt merkinnät. Liite C sisältää toteutetun siivoimen otsikkotiedoston ja liite D toteutetun siivoimen lähdekoodin.

2 Reaaliaikajärjestelmät

Reaaliaikajärjestelmillä tarkoitetaan sellaisia tietojärjestelmiä, joissa suoritettavien toimintojen on valmistuttava ennalta määrätttynä ajanhetkenä. Reaaliaikajärjestelmien toiminnan on oltava ennustettavissa sekä reaaliaikatehtävien on toimittava "nätisti", jottei järjestelmän toiminta vaarannu. Tällaisia järjestelmiä toteutettaessa kiinnitetään huomio siihen kuinka järjestelmän tehtävät aikataulutetaan siten, että jokainen tehtävä saa riittävästi suoritusaikaa. Tässä luvussa tutustutaan reaaliaikajärjestelmien perusteisiin [17].

2.1 Tietojärjestelmien vasteaikavaatimukset

Erilaiset tietojärjestelmät voidaan jakaa kolmeen ryhmään, sen mukaan millaisia vasteaikoja niiltä vaaditaan. Ensimmäisessä ryhmässä ovat *eräajojärjestelmät (batch systems)*. Eräajojärjestelmät toimivat siten, että käynnistyessään järjestelmä ottaa syötteen, suorituksen aikana järjestelmä prosessoi syötettä, ja lopuksi järjestelmä tulostaa saadun tuloksen [18]. Tällaisien järjestelmien suoritusnopeudella ei usein ole muita vaatimuksia kuin, että tulokset on saatava kohtuullisessa ajassa. Tämä vasteaika voi olla, tapauksesta riippuen, jopa useita viikkoja. Eräajo ei myöskään vaadi käyttäjiltä toimenpiteitä suorituksen aikana. Esimerkkinä eräajojärjestelmistä toimii mm. ohjelmointikielten kääntäjät ja palkanlaskennan ohjelmistot.

Toinen ryhmä on *interaktiiviset järjestelmät (interactive systems)*. Tällaisten järjestelmien toiminta on usein seuraavanlainen: Käyttäjä antaa komennon, jonka järjestelmä suorittaa ja ilmoittaa tuloksen käyttäjälle, käyttäjä antaa uuden käskyn, järjestelmä suorittaa sen ja niin edelleen. Tällaisissa järjestelmissä vasteajan on oltava samaa luokkaa kuin järjestelmän käyttäjän, ihmisen, vasteaika on [18]. Usein tällainen vasteaika tarkoittaa sekunnin kymmenesosaa tai vastaavaa aikaa. Mikäli vasteaika kasvaa liian suureksi, on järjestelmän käyttäminen ärsyttävää, mutta suurempaa vahinkoa ei kuitenkaan synny. Usein interaktiivisen järjestelmän esimerkkinä käytetään tekstinkäsittelyohjelmia.

Kolmas ryhmä, ja samalla se mihin tässä tutkielmassa keskitytään, on *reaaliaikajärjestelmät (real-time systems)*. Reaaliaikajärjestelmän määritelmästä on olemassa useita

hieman erilaisia versioita, mutta yhteistä kaikille määritelmille on se, että reaaliaikajärjestelmän oikeellisuus ei riipu pelkästään laskennan tuloksesta vaan myös laskentaan käytetystä ajasta [51]. Asia kiteytetään usein niin, että järjestelmän pitää toimia oikein aina ja tuottaa tämä oikea tulos ennalta määrätyssä ajassa. Reaaliaikajärjestelmissä järjestelmän vasteaika on siis ratkaiseva tekijä. Eri järjestelmien välillä vasteajat voivat vaihdella huomattavasti. Joissakin järjestelmissä vasteajat ovat muutamia millisekunteja, kun taas toisissa järjestelmissä vaadittu vasteaika on useita sekunteja. Esimerkkinä prosessinohjausjärjestelmät, joissa vasteajat ovat tyypillisesti 10 millisekunnin luokkaa. Usein on vaikea vetää raja siihen, milloin järjestelmää voidaan todella pitää reaaliaikaisena. Tämän tutkielman kannalta relevantteina voidaan pitää sellaisia reaaliaikajärjestelmiä, joiden vasteajat ovat korkeintaan muutaman millisekunnin luokkaa.

2.2 Reaaliaikajärjestelmien ominaisuudet ja jako

Kuten edellä mainittiin, reaaliaikajärjestelmien ominaisuuksiin kuuluu se, että niiden toiminnan oikeellisuus riippuu myös ajasta jolloin laskennan tulos on valmiina. Toisin sanoen, reaaliaikajärjestelmien on oltava oikeassa sekä loogisesti (ohjelman oikeellisuuden suhteen) että temporaalisesti (ajan suhteen). Reaaliaikaisessa tiedonkäsittelyssä on kyse siitä, että reaaliaikavasteet tyydetetään riittävän hyvin ja riittävän ennustettavasti [23]. Se mikä on riittävän hyvin ja riittävän ennustettava on sidoksissa kyseessä olevaan sovellukseen. Reaaliaikaisuus ei tarkoita sitä, että järjestelmä toimii erittäin nopeasti.

Perinteisesti reaaliaikavaatimukset jaetaan kahteen luokkaan: *kovaan (hard real-time)* ja *pehmeään (soft real-time)* [51]. Järjestelmä on kova, jos myöhästynyt vaste on automaattisesti vakava virhe. Pehmeissä järjestelmissä myöhästynyt vaste ei ole automaattisesti virhe, vaikkakin vasteen myöhästyminen saattaa olla kiusallinen. Parhaimpana esimerkkinä järjestelmästä, jonka reaaliaikavaatimus on kova, on ydinvoimalan ohjausjärjestelmä. Pehmeä vastevaatimus voisi olla multimediajärjestelmän dekooderissa, missä pieni viivästyminen aiheuttaa kiusallisen pätkähdyksen, mutta mitään suurempaa haittaa ei kuitenkaan pääse syntymään. Käytännössä jako reaaliaikajärjestelmien välillä on hiuksenhieno. Vaikeinta on määrittää milloin kaikkia vasteaikavaatimuksia ei tarvitse saavuttaa — siis milloin järjestelmä on pehmeä [45].

Usein reaaliaikajärjestelmiin liitetetään myös käsitteet kestävyys ja vikasietoisuus. Kestävyydellä tarkoitetaan sitä, että järjestelmä toimii oikein jopa odottamattomissa

olosuhteissa. Vikasietoisuus tarkoittaa sitä, että järjestelmä tunnistaa virhetilanteet ja osaa toipua niistä. Reaaliaikajärjestelmät voivat olla myös *sulautettuja järjestelmiä (embedded systems)*, jolloin laitteistorajoitteet asettavat lisävaatimuksia järjestelmien toiminnalle.

2.3 Vasteaikavaatimusten toteuttaminen

Reaaliaikajärjestelmissä vasteaikavaatimukset ovat tärkeässä asemassa. Luvun loppuosa käsittelee sitä kuinka reaaliaikavasteet saadaan tyydytettyä.

2.3.1 Tehtävät reaaliaikajärjestelmissä

Usein reaaliaikajärjestelmiltä vaaditaan kykyä suorittaa useita toimintoja samanaikaisesti. Järjestelmät mallinnetaan *tehtävien (task)* avulla [35]. Yksittäinen tehtävä voi olla prosessi, säie tai vuorottaisrutiini (co-routine). Järjestelmän koostuessa useammista tehtävistä on eri tehtävillä usein erilaiset vasteaikavaatimukset, ja järjestelmät koostuvat usein sekä kovista että pehmeistä tehtävistä. Kovat tehtävät ovat sellaisia tehtäviä, joiden reaaliaikavasteet ovat kovia. Vastaavasti pehmeiden tehtävien vasteaikavaatimukset ovat pehmeät. Tällöin järjestelmän kriittiset osat ovat kovia ja toissijaiset pehmeitä. Järjestelmissä voi lisäksi olla toimintoja, joiden suoritukselle ei ole määriteltynä minkäänlaisia vasteaikavaatimuksia ja niitä suoritetaan vain jos resursseja on jäljellä. Tälläisissa järjestelmissä tehtäviä, joilla on kova vasteaikavaatimus, kutsutaan *korkean prioriteetin prosesseiksi (high priority process)*. Vasteajaltaan pehmeät prosessit ovat *alhaisen prioriteetin prosesseja (low priority process)*.

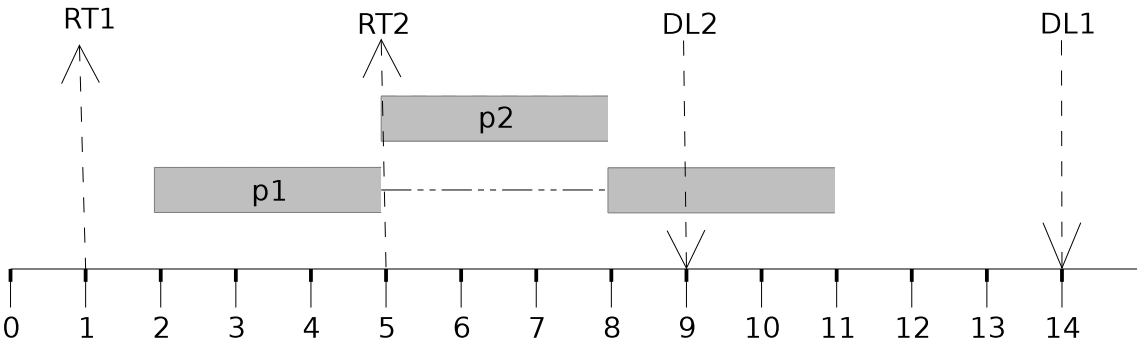
Tehtäviä suoritetaan yhden prosessorin tietokoneissa *samanaikaisesti (concurrently)*. Suorituksessa on vain yksi prosessi kerrallaan, mutta suoritettavaa prosessia vaihdetaan niin, että näyttäisi siltä, että useita prosesseja suoritetaan samanaikaisesti. Prosessien suorituksen tulee olla *keskeytettävissä (pre-emptable)*, jotta samanaikaisesti suorittaminen onnistuu [35]. Tehtävän suorittamista ei välttämättä pystytä keskeyttämään välittömästi, sillä tehtävän suoritus saattaa olla kriittisessä osiossa, jonka pitää suorittaa loppuun ennen tehtävän keskeyttämistä. Tästä järjestelmään aiheutuu *latenssia (latency)*. Latenssi on se aika, joka kuluu tehtävän laukaisevasta ärsykkeestä siihen hetkeen, kun tehtävän varsinainen suoritus aloitetaan [18]. Latenssia voidaan aiheuttaa myös tahallisesti. Esimerkiksi sellaiset tehtävät, jotka näytteistävät järjestelmään syötettävää signaalia eivät välttämättä saa aloittaa suoritustaan välittömästi, kun pyyntö

saapuu, vaan ne pakotetaan odottamaan parasta mahdollista näytteistysaikaa. Lisäksi suoritettavan tehtävän vaihtamisesta (context switch) aiheutuu hidastusta. Samanaikainen suoritus kannattaa erottaa todellisesta rinnakkaisuudesta. Edellä esitettiin tehtävien samanaikainen suorittaminen, mutta todellinen *rinnakkaissuoritus (parallel)* tapahtuu kuitenkin tietokoneissa, joissa on useampi kuin yksi prosessori [55, luku 2]. Tällöin jokaisessa prosessorissa ajetaan omaa tehtävää, jotka todellakin suoritetaan rinnakkain.

Korkeaprioriteettisen prosessin suorittaminen *estää (block)* kaikkien alhaisemman prioriteetin prosessien suorittamisen. On myös mahdollista, että alemman prioriteetin prosessi estää korkeaprioriteettisen prosessin suorituksen. Tällainen tilanne syntyy silloin, kun alhaisen prioriteetin prosessi varaa käyttöönsä resurssin, jota myös korkeaprioriteettinen tehtävä tarvitsee. Mikäli korkeaprioriteettinen tehtävä saa suoritusvuoron, ei tehtävää kuitenkaan pysytä suorittamaan, sillä resurssi ei ole käytettävissä. Yleensä korkean prioriteetin prosessin suoritukselle aiheutuva latenssi on pieni, sillä lukoilla suojataan vain lyhyitä kriittisiä osioita. Varsinainen ongelma kuitenkin syntyy, jos sillä välillä, kun korkeaprioriteettinen tehtävä on estettynä alhaisen prioriteetin tehtävän toimesta, jokin kolmas "keskiprioriteettinen" prosessi estää alhaisen prioriteetin prosessin suorituksen. Tämän ongelman tunnisti Lampson ja Redell [32]. Nykyisin kyseistä tilannetta kutsutaan *prioriteetti-inversioksi (priority inversion)*, ja se saattaa aiheuttaa korkeaprioriteettiselle prosessille mielivaltaisen pitkän pysähdyksen. Kuuluisa käytännön esimerkki prioriteetti-inversiosta on vuodelta 1997 NASA:n Mars Pathfinder -projektista, jossa Mars-planeetalle laskeutunut kulkija alkoi onnistuneen alun jälkeen käynnistymään uudelleen. Uudelleenkäynnistymisten syyksi todettiin prioriteetti-inversio, joka esti liian kauan erään korkean prioriteetin prosessin suorittamisen [44].

On myös mahdollista, että järjestelmä *lukkiutuu (deadlock)* [55, luku 2]. Tällä tarkoitetaan sellaista tilaa, jossa kaksi tai useampi prosessia estää toistensa suorittamisen, ja järjestelmä ikuiseen odotustilaan. Tällaista tilannetta ei luonnollisesti saa tapahtua, sillä se on yksi pahimmista virhetilanteista, johin järjestelmä voi joutua.

Kuvassa 2.1 on esitettynä aikajanalla kaksi prosessia: p1 ja p2. Prosessin p1 oletetaan olevan alhaisempaa prioriteettia kuin prosessi p2. *Vapautushetki (release time)* on se hetki, jolloin tehtävän laukaiseva ärsyke saadaan. Kuvassa prosessin p1 vapautushetkeä on merkitty RT1:llä ja prosessin p2 vapautushetkeä RT2:lla. Esimerkissä prosessilla p1 tämä tarkoittaa ajanhetkeä 1 ja prosessilla p2 taas hetkeä 5. Prosessien *absoluuttinen takaraja (absolute deadline)*, on se hetki jolloin prosessin suorituksen tu-



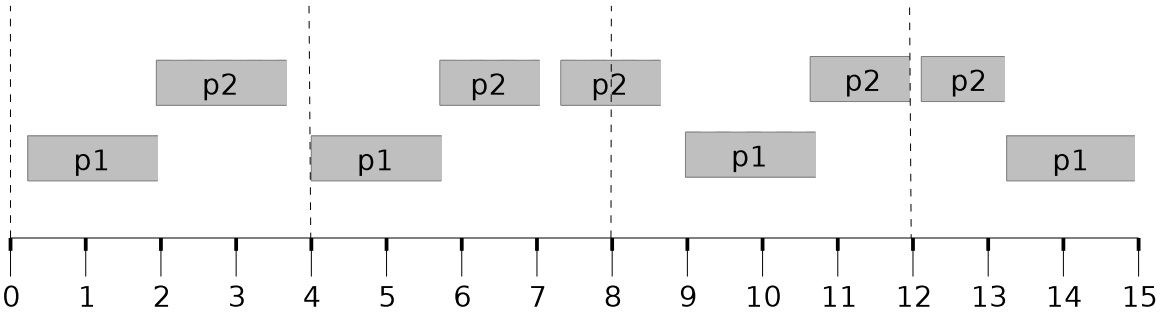
Kuva 2.1: Esimerkki prosesseista

lee olla valmis. Kuvassa prosessin p1 absoluuttinen takaraja on merkittynä DL1:llä ja prosessin p2 takaraja DL2:lla. Prosessilla p1 tämä on 14 ja prosessilla p2 vuorostaan 9. Prosessien *suhteellinen takaraja (relative deadline)* on aika vapautushetken ja absoluuttisen takarajan välillä; prosessilla p1 13 ja prosessilla p2 4. *Vasteaika (response time)* tarkoittaa aikaa, joka kuluu vapautushetkestä tehtävän suorituksen päättymiseen, prosessilla p1 vasteaika on 10 ja prosessilla p2 3. Lisäksi prosessi p1 on estettynä prosessin p2 toimesta 3 aikayksikköä, ja prosessilla p1 on 1 aikayksikkö latenssia. Lisäksi sanotaan, että tapahtuu *ylivuoto (overflow)*, jos prosessin vasteaikavaatimusta ei saada tyydytetyksi.

Reaaliaikaprosessit voivat olla periodillisia (periodic), peridiottomia (aperiodic) tai sporadisia (sporadic) [51]. Periodillisten prosessien pyynnöt tapahtuvat aina ja vain tietyn ajan kuluttua siitä, kun prosessin edellinen pyyntö tapahtui. Prosessin periodi on aina sama, mutta eri prosesseilla voi luonnollisesti olla eripituiset periodit. Periodittomien eli asynkronisten prosessien pyynnöt voivat tapahtua milloin vain, eikä pyynnöillä ole mitään rajoitteita. Sporadiset prosessit toimivat muuten kuin periodittomat prosessit, mutta niiden vapautushetkien välillä on jokin vähimmäispituus. Kuvassa 2.2 on havainnollistettuna sekä periodillinen että perioditon prosessi. Prosesseista p1 on periodillinen ja sen periodin pituus on neljä aikayksikköä, siis sen vapautushetket ovat aina neljän aikayksikön välein. Prosessi p2 on perioditon ja sen vapautushetki voi olla milloin vain.

2.3.2 Ennustettavuus

Reaaliaikajärjestelmien ja perinteisten tietojärjestelmien välillä on suuri ero. Usein perinteisiä järjestelmiä toteutettaessa ollaan kiinnostuneita järjestelmän keskiarvoisesta



Kuva 2.2: Periodillinen ja periodioton prosessi

suoritusnopeudesta, sillä harvoin tapahtuvat, hetkelliset suorituskyvyn heikkenemiset ovat vain hieman kiusallisia. Reaaliaikajärjestelmissä tilanne on kuitenkin toinen. Niissä huomio kiinnitetään vain ja ainoastaan pahimpaan mahdolliseen tilanteeseen, sillä järjestelmän toiminta on taattava myös siinä epätodennäköisessä tilanteessa, että pahin tapahtuu. Reaaliaikajärjestelmiä toteutettaessa onkin usein suosittava ennustettavuutta tehokkuuden kustannuksella. On mahdollista, että keskiarvoisessa tilanteessa reaaliaikajärjestelmä käyttää vain pienen osan mahdollisista resursseista, mutta pahimman mahdollisen tilanteen sattuessa käyttämättömät resurssit tarvitaan suorituksen takaamiseen.

Tärkein asia reaaliaikajärjestelmiä suunniteltaessa ja toteuttaessa on *ennustettavuus (predictability)* [23]. Jotta järjestelmä pystyy toteuttamaan kaikki sille asetetut vasteaikavaatimukset, tulee sen tilan ja suorituksen olla ennustettavissa. Tämä tarkoittaa sitä, että järjestelmän jokaisen alkeistoiminnon suoritusajan on oltava ylhäältä rajoitettu. Tämä rajoitus koskee myös rekursiivisia funktiota ja tietorakenteita, joiden syvyydellä on oltava yläraja [41]. Näiden suoritusaikojen avulla järjestelmän tehtäville voidaan suorittaa suoritusaika-analyysi, jonka tuloksena saadaan suurin aika, joka tehtävän suoritukseen kuluu, lyhyesti *WCET (Worst Case Execution Time)* [11].

Ennustettavuus ja suoritusaikarajoitteet eivät koske pelkästään prosessien käyttämää prosessoriaikaa vaan myös muita tehtävien käyttämiä resursseja. Yksittäiset tehtävät eivät saa varata käyttämiään resursseja pitkiksi ajoiksi, vaan varatut resurssit pitää vapauttaa riitävän nopeasti, etteivät prosessit estä toistensa suorittamista liiaksi.

2.3.3 Tehtävien samanaikainen suorittaminen

Reaaliaikajärjestelmien koostuessa useista tehtävistä ongelmaksi muodostuu usein se kuinka tehtävien suoritus *aikataulutetaan (scheduling)* siten, että jokainen prosessi saa

riittävästi suoritusaikaa. Aikataulutus on mahdollista suorittaa käyttämällä hyväksi suoritusaika-analysin tuloksia. Prosessien WCET-arvojen avulla voidaan suorittaa analyysi siitä, että ovatko prosessit aikataulutettavissa pahimmassa mahdollisessa tilanteessa [11]. Aikataulutusalgoritmit ovat yksinkertaisesti joukko sääntöjä, joiden avulla päätetään mitä tehtävää suoritetaan.

Suoritettaessa tehtäviä samanaikaisesti kiinnitetään huomio siihen, mitä prosessia kulloinkin suoritetaan. Suoritusvuoro on annettava sellaiselle prosessille, jonka on kiireellisin. Jotta tämä voidaan tehdä, prosesseille on määrätävä keskinäinen prioriteettijärjestys sen mukaan, kuinka tärkeä prosessin suoritus on muihin tehtäviin verrattuna. Kiireellisemmän prosessin tullessa tilaan, jossa se on suoritettavissa, pitää suorituksessa oleva alhaisemman prioriteetin prosessi keskeyttää ja suoritusvuoro antaa korkeamman prioriteetin prosessille. Jotta tähän pystytään, tulee jokaisen tehtävän olla keskeytettävissä [35].

Noudattaen Liun ja Laylandin [35] jakoa, aikataulutusalgoritmit jaetaan kolmeen luokkaan: *stattisen (static)*, *dynaamisen (dynamic)* ja *yhdistetyn (mixed)* prioriteetin algoritmeihin. Staattisen prioriteetin algoritmeissa, toiselta nimeltään *kiinteän prioriteetin (fixed priority)* algoritmit, jokaisella prosessilla on ennalta määrätty prioriteetti, eikä tämä prioriteetti voi muuttua. Dynaamisen prioriteetin algoritmeissa prosessin prioriteetti voi muuttua suorituksen aikana. Yhdistetyn prioriteetin algoritmeissa osalle prosesseista annetaan kiinteä prioriteetti ja jäljelle jääneiden prosessien prioriteetti määritellään dynaamisesti.

2.3.4 Kiinteän prioriteetin aikataulutus

Parhaiten tunnettu staattisen prioriteetin aikataulutusalgoritmi on *RMS (Rate Monotonic Scheduling Algorithm)*, joka perustuu *RMA:han (Rate Monotonic Analysis)*, ja sen matemaattinen teoria on peräisin Liulta ja Laylandilta [35] vuodelta 1973. Menetelmä on optimaalinen siinä mielessä, että mikäli RMA:n perusteella annetulla prioriteettijärjestyksellä ei tehtäviä voida aikatauluttaa, ei niitä voida aikatauluttaa millään muullakaan kiinteän prioriteetin menetelmällä. Aikataulutettaessa tehtäviä RMA:n mukaisesti järjestelmän tehtävien suoritusta analysoidaan tehtävien periodin perusteella ja niiden prioriteetit asetetaan siten, että korkeimman prioriteetin saa se tehtävä, jonka periodi on lyhin.

RMA:n idea on, että mikäli aikataulutettavat tehtävät saadaan aikataulutettua pahimassa mahdollisessa tilanteessa, saadaan ne aikataulutettua myös kaikissa muissa tilanteissa [35]. Tämä aikataulutuksen pahin mahdollinen tilanne tapahtuu silloin, kun

kaikkien prosessien vapautushetki on samaan aikaan.

RMA:n perusmuotoa voidaan käyttää, jos järjestelmän tehtävät täyttävät seuraavat ehdot:

1. Kaikki tehtävät ovat keskeytettävissä.
2. Tehtävät ovat toisistaan riippumattomia ja periodisia.
3. Tehtävät voidaan järjestää periodien mukaiseen kasvavaan järjestykseen.
4. Tehtävien takaraja ja periodi ovat samat.

Lisäksi tehtävät pystytään aikatauluttamaan ja niiden prioriteetit määräämään RMA:n mukaisesti, jos tehtävien yhteenlasketulle prosessorin hyötykuormalle pätee

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U(n) = n(2^{1/n} - 1). \tag{2.1}$$

Kaavassa n on tehtävien lukumäärä, C_i on tehtävän i suoritukseen kuluva maksimiaika (prosessin WCET), T_i on tehtävän i periodin pituus ja $U(n)$ on prosessorin hyötykuorma. RMA:sta kannattaa huomioida, että prosessorin hyötykuormafunktio on laskeva ja

$$\lim_{n \to \infty} n(2^{1/n} - 1) \approx 0,69 \tag{2.2}$$

Tämä tarkoittaa, että prosessien lukumäärän kasvessa, prosessoria voidaan hyödyntään maksimissaan vain 69 prosenttisesti, jotta tehtävien vaatimat vasteajat voidaan toteuttaa.

Perusmuodossaan RMA on rajoittunut, mutta sillä on myös hyvät puolensa. Ensinnäkin RMA ottaa kantaa vain koviin reaaliaikavasteisiin, eikä sen perusmuotoa voida käyttää pehmeiden tehtävien analysoimiseen. Lisäksi tehtäville asetetut rajoitteet ovat melko vaativia, sillä useinkaan kaikki tehtävät eivät ole periodillisia. On myös mahdollista, varsinkin jos tehtävät käyttävät jaettuja resursseja, että alemman prioriteetin tehtävä estää korkeamman prioriteetin tehtävän suorituksen varaamalla jonkin resurssin, jota myös korkeamman prioriteetin tehtävä käyttää. RMA ei myöskään huomioi järjestelmän latenssia, vaan siinä prosessien oletetaan olevan keskeytettävissä välittömästi. On myös mahdollista, että RMA:n perusteella saadaan tulos ettei tehtäväjoukkoa saada aikataulutettua, vaikka aikatauluttaminen todellisuudessa onnistuisikin. RMA:sta onkin kehitetty myös paranneltuja versioita, joissa näihin ongelmiin otetaan kantaa, kyseisiin menetelmiin ei tässä tutkielmassa paneuduta tarkemmin.

RMA:n ja samalla myös kaikkien muiden kiinteän prioriteetin aikataulutusalgoritmien eduksi voidaan mainita, että mikäli analyysin tuloksena saadaan varmuus siitä, että tehtävät voidaan aikatauluttaa, ei tämä tulos myöskään muutu, mikäli järjestelmään ei tehdä muutoksia. Voidaan siis olla varma siitä, että järjestelmä todellakin pystyy toteuttamaan reaaliaikavasteet.

2.3.5 Periodittomat tehtävät kiinteän prioriteetin aikataulutuksessa

Kiinteän prioriteetin aikataulutusmentelmien avulla ei ole mahdollista aikatauluttaa periodittomia tehtäviä [35]. Yksinkertaisimmillaan ongelma voidaan ratkaista pollaamalla periodittomien tehtävien laukaisevia ärsykkeitä periodillisesti [30, luku 5]. Pollaaminen saattaa kuitenkin aiheuttaa periodittomien tehtävien suoritukselle suurta hidastusta, sillä mikäli periodittoman tehtävän laukaiseva ärsyke tulee välittömästi pollaamisen jälkeen joudutaan odottamaan koko pollaussykli ennen kuin kyseistä tehtävää voidaan suorittaa.

Paremmat ratkaisut tähän ongelmaan perustuvat yleensä *periodillissiin palvelimiin (asynchronous servers)* [30]. Näiden palvelinmenetelmien ideana on se, että periodittomien tehtävien suorittamista varten luodaan periodillinen tehtävä, jonka suoritusaika varataan asynkronisten pyyntojen toteuttamiseen [7]. Periodillisista palvelimista tämän tutkielman kannalta olennaisin on *sporadinen palvelin (sporadic server)*, jonka toiminta esitetään seuraavaksi.

Sporadinen palvelin on korkeaprioriteettinen periodillinen tehtävä, jonka suoritusaika varataan asynkronisten tehtävien suorittamiseen taaten samalla, että periodilliset tehtävät saadaan aikataulutetuksi [30, luku 5]. Sporadinen palvelin odottaa asynkronisen tehtävän laukaisevaa ärsykettä, ja pyynnön saapuessa palvelin aloittaa tämän tehtävän suorittamisen. Tehtävää suoritetaan kunnes se on suoritettu loppuun tai kunnes sporadisen palvelimen suorituskapasiteetti loppuu. Mikäli tehtävää ei saada suoritettua loppuun palvelimen suorituskapasiteettijakson aikana, lopetetaan tehtävän suorittaminen hetkeksi, ja suoritusta jatketaan, kun palvelimen suorituskapasiteetti täydennetään palvelimen seuraavan periodin alussa. Sporadisen palvelimen avulla periodittomien tehtävien analysointi voidaan tehdä periodillisilla menetelmillä, ja periodittomat tehtävät voidaan aikatauluttaa RMA:n avulla.

Sporadisen palvelimen toiminta kiteytyy kolmeen parametriin: palvelimen prioriteettiin, palvelimen suorituskapasiteettiin (execution capacity) ja palvelimen täydennysperiodiin (replenishment period) [30, luku 5]. Palvelimen prioriteetti kertoo sen kuinka korkealla prioriteetilla periodittomia pyyntöjä palvellaan. Suoritusaikakapasi-

P_i	T_i	C_i
P_1	10	5
P_2	20	10
P_{SS}	10	?

Taulukko 2.1: Tehtävien parametrit

teetti on aikaviipale, joka annetaan palvelimen käyttöön yhden periodin aikana; siis sporadisen palvelimen maksimisuoritusaika. Täydennysperiodi on se aika, jonka on kuluttava, jotta palvelimen suoritusaikakapasitetti palautetaan, eli yksikertaisemmin palvelintehtävän periodi.

Palvelimelle voidaan määrätä prioriteetti, jolla asynkronisia tehtäviä suoritetaan. Usein asynkroniset tehtävät suoritetaan korkeimmalla mahdollisella prioriteetilla, joka kiinteän prioriteetin aikataulutuksessa tarkoittaa, että palvelimen periodi on lyhin. Mikäli näin on, voidaan periodin (T_{SS}) avulla laskea palvelimen suoritusaikakapasiteetille turvallinen arvo, jota käytettäessä pystytään takaamaan kaikkien tehtävien aikataulutus siten, että tehtävät tyydyttävät vasteaikavaatimuksensa [28]. Suoritusaikakapasiteetti (SS_{size}) saadaan kaavalla

$$SS_{size} = \min \left\{ x \left| \left\lceil \frac{T_i}{T_{SS}} \right\rceil \cdot x + \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil \cdot C_j \leq D_i \wedge i \in \{1, ..., n\} \right. \right\} \tag{2.3}$$

Kaavassa käytetään seuraavia merkintöjä: n prosessien lukumäärä, T_i prosessin i periodi, T_{SS} sporadisen palvelimen periodi, C_i prosessin i suurin mahdollinen suoritusaika ja D_i prosessin i takaraja. Lisäksi oletetaan, että $T_i \leq T_j$ kun $i \leq j$.

Idea sporadisen palvelimen suoritusaikakapasiteetin laskemisen taustalla on seuraava: Prosessin i periodin aikana on otettava huomioon kaikkien prosessia i korkeampiprioriteettisten tehtävien suoritukset. Tehtävän itsensä ja kaikkien korkeaprioriteettisten tehtävien instanssien, jotka ehditään suorittamaan prosessin i periodina aikana, yhteenlaskettun suoritusajan on oltava pienempi kuin tehtävän i takaraja. Tämä lasketaan kaavan 2.3 summatermillä. Mikäli aikaa jää jäljelle, voidaan se käyttää sporadisen palvelimen suoritusaikana. Palvelimen periodin laskenta perustuu Lehoczkyn, Shan ja Dingin [33] analyysiin RMS-algorimista.

Tarkastellaan sporadisen palvelimen suoritusaikakapasiteetin laskemista vielä esimerkin avulla: Talukossa 2.1 on esimerkissä käytettävien tehtävien parametrit. Sarakkeessa P_i on identifioituna eri tehtävät, sarakkeessa T_i on tehtävien periodit ja sarakkeessa C_i tehtävien WCET-arvot. Prosessien takaraja on sama kuin kyseisen periodin

i	Kaavan 2.3 soveltaminen	x
1	$1 \cdot x + 1 \cdot 4 \leq 10$	6
2	$2 \cdot x + 2 \cdot 4 + 1 \cdot 10 \leq 20$	1

Taulukko 2.2: Palvelimen suoritusaikakapasiteetin laskeminen

pituus. Esimerkissä on siis kaksi tehtävää ja sporadinen palvelin, jonka periodi on 10.

Taulukossa 2.2 on esitettynä palvelimen suoritusaikakapasiteettin määrääminen kaavan 2.3 mukaisesti. Ensimmäisessä sarakkeessa on iteraatiokierrosta merkitsevä i:n arvo, toisessa sarakkeessa varsinainen lasku ja kolmannessa sarakkeessa kyseiselle iteraatiolle saatu suoritusaikakapasiteetin arvo. Ensimmäisellä iteraatiolla tarvitsee ottaa huomioon prosessin P_1 suoritus, jolloin luppoaikaa, jota voidaan käyttää palvelimen suoritukseen, jää 6 aikayksikköä. Toisella iteraatiokierroksella tarvitsee huomioida sekä tehtävän P_2 suoritus että tehtävän P_1 suoritukset, jotka tapahtuvat tehtävän P_2 periodin aikana. Näitä tehtävän P_1 suorituksia on 2 kappaletta, jotka kukin vievät suoritusaikaa 4 aikayksikköä. Lisäksi tehtävän P_2 suoritus kuluttaa 10 aikayksikköä, jolloin luppoaikaa jää 2 aikayksikköä. Koska palvelimen periodi on 10 aikayksikköä, ehditään sitä suorittamaan 2 kertaa tehtävän P_2 periodin aikana. Näin ollen palvelimen suoritusaikakapasiteetiksi kunkin palvelimen periodin aikana tulee 1 aikayksikkö. Siis SS_{size} saa arvokseen 1.

2.3.6 Dynaaminen aikataulutus

Dynaamiset aikataulutusmenetelmät ovat kiinteisiin menetelmiin verrattuna paljon joustavampia. Käytettäessä dynaamisia menetelmiä järjestelmä voi reagoida tilanteisiin siten, että prosessoriaikaa annetaan enemmän prosesseille, jotka suoritusaikaa todella tarvitsevat. Eräs esimerkki dynaamisesta aikataulutuksesta on *EDF (Earliest Deadline First)*-aikataulutus [35]. EDF aikataulutuksessa korkeimman prioriteetin saa se prosessi, jonka takaraja on lähimpänä. Millä tahansa hetkellä suoritusvuoro voidaan siirtää sellaiselle tehtävälle, joka on suoritettavissa ja jonka prioriteetti on korkein.

EDF on optimaalinen aikataulutusalgoritmi. Tällä tarkoitetaan sitä, että sen avulla aikataulutetut prosessit pystyvät tyydyttämään vasteaikavaatimuksensa, mikäli se vain on mahdollista. EDF-aikataulutuksella pystytään myös saavuttamaan sataprosenttinen prosessorin hyötykäyttö [35]. Tämän lisäksi, mikäli sattuisi niin ikävästi, että vasteaikoja ei pystytä tyydytetyksi, EDF:n ominaisuuksiin kuuluu se, että se minimoi ajan, jolla vasteajat ylitetään [23]. Lisäksi EDF aikataulutusta käytettäessä tehtävien ei tarvitse

olla periodillisia, vaan ne voivat olla myös asynkronisia. EDF:n käyttö on tosin vaikeampaa kuin RMA:n, sillä ei ole helppo todistaa, että joukko tehtäviä on mahdollista aikatauluttaa EDF:n avulla. Toisaalta tiedetään, että mikäli prosessijoukko saadaan aikataulutettua RMA:n avulla, saadaan se aikataulutettua myös EDF:n avulla.

3 Muisti ja sen hallinta

Toimiakseen tietokoneohjelmat tarvitsevat muistia. Tässä luvussa tutustutaan muistinhallinnan perusteisiin sekä muistinhallinnan ongelmiin.

3.1 Muistinkäytön perusteet

Ohjelmat voivat varata *varata (allocate)* muistia kolmella eri tavalla: *stattisesti (static)*, *pinosta (stack)* tai *keosta (heap)* [22, sivu 2].

Staattinen muistinvaraus toimii siten, että kaikki käytettävä muisti määritellään jo ohjelman käännöksen aikana, eikä varattun muistin kokoa tai paikkaa voida enää muuttaa [22, sivu 3]. Staattinen muistinvaraus aiheuttaa seuraavat rajoitteiteet: käytettävien tietorakenteiden koko on tiedettävä käännöksen aikana, eivätkä aliohjelmat voi olla rekursiivisia. Toisaalta stattista muistinvarausta käyttävien ohjelmien suoritus on usein erittäin nopeaa, sillä tietorakenteita ei tarvitse luoda ohjelman suorituksen aikana. Lisäksi staattista muistinvarausta käyttävät ohjelmat eivät voi kaatua muistin loppumiseen, sillä ohjelman muistinkäytön tarve on ennalta tiedossa.

Varattaessa muistia pinosta voidaan staattisen muistin ongelmat osittain kiertää, sillä pinosta muistia varaavissa ohjelmissa on mahdollista sekä toteuttaa rekursiivia aliohjelmia että varata erikokoisia lokaaleja tietorakenteita aliohjelman eri suorituskerroilla [22, sivu 3]. Koska pinosta voidaan poistaa vain se alkio, joka siihen lisättiin viimeisenä, ei pinosta varattu muisti voi elää enää sen jälkeen, kun lohkon varanneen aliohjelman suoritus päättyy. Lisäksi pinon avulla voidaan aliohjelmista palauttaa vain sellaisia muistilohkoja, joiden koko tiedetään jo käännösaikaisesti.

Sekä staattinen muistinkäyttö että pinosta varattava muisti ovat kovin rajoittavia ohjelman suorituksen kannalta. Varattaessa muistia keosta ei edellä mainittuja rajoitteita ole, sillä tätä dynaamista muistia voidaan varata ja vapauttaa missä järjestyksessä tahansa. Dynaamisella muistilla ei ole myöskään rajoitteita tietorakenteiden koon tai aliohjelmasta palautettavan muistialueen koon suhteen [22, sivu 3].

Dynaamisella muistilla on myös muutama ongelma: se saattaa vaikuttaa heikentävästi ohjelman suorituskykyyn sekä muisti saattaa pirstoutua. Seuraavissa luvuissa kerrotaan näistä ongelmista hieman tarkemmin: luvussa 3.1.3 kerrotaan dynaamisen

muistin vaikutuksesta ohjelman suoritukseen ja luvussa 3.2 vuorostaan pirstoutumisesta. Lisäksi dynaamista muistia käytettäessä varattu muisti on myös vapautettava. Tätä muistin vapauttamista kutsutaan tämän tutkielman yhteydessä muistinhallinnaksi, ja se voidaan toteuttaa kahdella tavalla: manuaalisesti (katso luku 3.3) tai automaattisesti (luku 3.4). Näistä jälkimmäistä, automaattista muistinhallintaa, kutsutaan *muistinsiivoukseksi (garbage collection*).

3.1.1 Virtuaalimuisti

Tietokonejärjestelmissä muistin määrä on rajallinen, joten nykyaikaiset käyttöjärjestelmät käyttävät *virtuaalimuistia (virtual memory)* [55, luku 4]. Käytettäessä virtuaalimuistia sellainen muistin osa, jota ei sillä hetkellä tarvita, kopioidaan johonkin toissijaiseen tietovarastoon, kuten esimerkiksi kovalevylle, ja kun kyseistä muistia jälleen tarvitaan, kopioidaan se takaisin varsinaiseen muistiin. Virtuaalimuistin avulla järjestelmän muistin määrää saadaan näennäisesti kasvatettua, mutta samalla myös järjestelmän toiminta hidastuu, sillä muistin kopiointi muistista tietovarastoon ja tietovarastosta muistiin on huomattavasti hitaampaa kuin pelkkä muistin lukeminen tai kirjoittaminen.

Useimmissa virtuaalimuistijärjestelmissä muistinkäsittelyyn kuluva aika on rajoittamaton, sillä muistin kopiointiin muistista levylle tai levyltä muistiin ei sovelleta minkäänlaisia prioriteettejä [30, luku 9]. Vaikka virtuaalimuistin toteuttamiseen käytettäisiinkin prioriteettejä on ero keskiarvoisen ja pahimman mahdollisen muistinkäsittelyyn kuluvan ajan välillä niin suuri, että virtuaalimuisti on käyttökelvoton aikakriittisiin järjestelmiin. Virtuaalimuistin käytöstä aiheutuvat viiveet saattavat vaikuttaa kriittisten tehtävien suoritukseen siten, että tehtävät eivät pysty toteuttamaan niille asetettuja vasteaikavaatimuksia. Tästä syystä reaaliaikajärjestelmissä on vain harvoin virtuaalimuistia, eli kaikki järjestelmän muisti on todellista [40].

3.1.2 Muistin rakenne

Tässä luvussa tarkennetaan tutkielman kannalta tärkeimpiä muistinhallintaan liittyviä termejä. Nämä termit ja käsitteet kuuluvat alan yleissivistykseen, joten ne esitellään vain pintapuolisesti.

Kuten aikaisemmin jo kerrottiin, dynaamisessa muistinvarauksessa käytettävä muisti varataan keosta. Keon koko on rajattu, ja keko jaetaan kahteen osaan: varattuun ja vapaaseen. Varattu muisti, siis se, joka sovelluksilla on käytössä, koostuu puolestaan

muuttujista (variable). Muuttujat voivat olla joko *osoittimia (pointer)*, *atomaarisia* tai *tietueita*. Atomaariset muuttujat sisältävät hyötydataa ja osoittimet ovat sellaisia alkioita, jotka viittaavat toiseen muuttujaan. Osoittimet voivat myös olla *tyhjiä (null)*.

Muuttuja on *tietue (record)*, jos kyseinen muuttuja on vektori, joka sisältää useita erityyppisiä alkioita. Tässä tutkielmassa tietueesta käytetään myös oliopohjaisista ohjelmointikielistä tuttua termiä olio, ja tutkielman kannalta kyseiset termit voidaan olettaa yhteneväisiksi, pois lukien ne muutamat kohdat, joissa käsitellään pelkästään olio-ohjelmointia. Tästä ei kuitenkaan pitäisi seurata epäselvyyttä, sillä tällöin termin olio käyttö tarkennetaan liittyväksi olio-ohjelmointiin.

3.1.3 Dynaamisen muistin vaikutus ohjelman suorituskykyyn

Dynaamisen muistin käytöllä on myös negatiivisia vaikutuksia. Dynaamisen muistin varaaminen vie aikaa, eikä muistin vapauttaminenkaan ole ilmaista. Yleiskäyttöiset muistiallokaattorit toimivat keskiarvoisesti hyvin, mutta niiden vaihtelevat vasteajat muistia varattaessa ja vapauttaessa ovat reaaliaikajärjestelmien kannalta sietämättömiä [43]. Muistia varattaessa allokaattorin on etsittävä tietorakenteidensa kätköistä alue, jolla muistinvarauspyyntö saadaan tyydytetyksi. Tämän alueen etsimiseen kuluva aika on riippuvainen useasta asiasta. Siihen vaikuttaa mm. varattavan alueen koko, keon nykyinen tila ja tietenkin allokaattorin toteutus. Samanlaisia ongelmia on myös muistia vapautettaessa, sillä käytössä ollut muistialue on palauttettava allokaattorin tietorakenteiden syövereihin.

Reaaliaikajärjestelmissä tällainen allokaattorin toiminta ei ole sallittua, sillä muisti on kyettävä sekä varaamaan että vapauttamaan rajoitetussa ajassa, jotta tehtävien WCET-arvot pystytään määrittämään.

3.2 Muistin pirstoutuminen

Dynaamisen muistin kanssa ongelmaksi saattaa muodostua muistin *pirstoutuminen (fragmentation)*. Pirstoutuneella muistilla tarkoitetaan vapaata muistia, jota ei voida hyödyntää muistia varattaessa [60]. Muisti voi olla pirstoutunut kahdella eri tavalla: joko *ulkoisesti (external)* tai *sisäisesti (internal)*.

Ulkoisesti pirstoutunut muisti on vapaata muistia joka koostuu pienistä epäjatkuvista muistialueista [60]. Tällaista vapaata mutta pirstoutunutta muistia voi olla yhteenlaskettuna tarpeeksi, jotta muistinvarauspyyntö saataisiin tyydytetyksi, mutta

Kuva 3.1: Keko a) ulkoisesti pirstoutuneena ja b) ideaalitilanteessa

todellisuudessa tämä ei kuitenkaan onnistu, koska vapaasta muistista ei löydy tarpeeksi suurta yhtenäistä aluetta. Pahimmassa tapauksessa yksi keon keskellä oleva varattu tavu estää varaamasta muistia tietueelle, joka tarvitsisi tilaa hieman päälle puolet keon koosta. Muisti pirstoutuu ulkoisesti, kun muistialueita varataan erikokoisina paloina, ja käytetyt muistialueet vapautetaan eri järjestyksessä kuin missä ne varattiin. Kuvassa 3.1 on esiteltynä kaksi kekoa. Kuvassa valkoinen alue kuvaa vapaata muistia ja musta varattua. Kuvan keko a) esittää tilannetta, jossa muisti on ulkoisesti pirstoutunut. Vaikka keossa on vapaata muistia jäljellä lähes puolet, rajoittaa suurin yhtenäinen vapaa muistialue varattavan muistin maksimikokoa . Kuvan keko b) esittää samaa muistinkäyttöä kuin keko a), mutta ideaalitilanteessa, jossa kaikki käytetty muisti on pakattuna keon alkuun ja vapaa tila on yhtenäisenä keon lopussa. Keossa b) ei siis ole ulkoisesti pirstoutunutta muistia.

Sisäisesti pirstoutuneella muistilla tarkoitetaan sellaista muistialuetta, joka on merkitty varatuksi mutta jota ei voida käyttää mihinkään [60]. Usein sisäisesti pirstoutunut muisti johtuu siitä, että muistiallokaattori-implementaatiot saattavat merkitä muistia varatuksi hieman enemmän kuin olisi tarve. Tämä johtuu siitä, että jotkin allokaattorit jakavat vapaan muistin määrätyn kokoisiin alueisiin, eivätkä salli muun kokoisten alueiden varaamista. Allokaattorin toteutuksesta riippuen se voi käyttää vain yhden kokoisia alueita tai mahdollisesti useamman kokoisia alueita sekaisin.

Esimerkkinä tällaisesta allokaattorista on Buddy System, joka toimii siten, että muistia varataan vain kahden potenssien kokoisina paloina [31]. Varattaessa Buddy System -allokaattorilta muistialuetta, jonka koko on s, varataan todellisuudessa alue, jonka koko on 2^k, missä $k = \lceil \log_2 s \rceil$. Näitä ylimääräisiä pyöristyksistä aiheutuvia varattuja tavuja ei ohjelmassa pystytä käyttämään, joten ne ovat hukkaan heitettyjä. Esimerkiksi varattaessa viisi tavua muistia varaa allokaattori oikeasti alueen, jonka koko on kahdeksan tavua, kun taas varattaessa muistialuetta, jonka koko on 15 tavua, varattaisiin todellisuudessa 16 tavua.

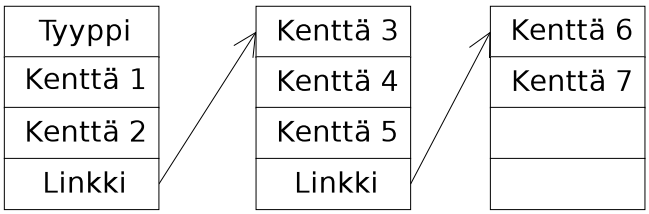
Johstone ja Wilson [25] esittävät, että hyvin tunnettuja muistiallokaattoreita käytettäessä muistin pirstoutuminen on hyvin pientä suurimmassa osassa ohjelmia, eikä pirstoutumisesta näin ollen tarvitsisi välittää. Tämä analyysi kertoo kuitenkin vain keskiarvoisen tilanteen eikä pahinta mahdollista tilannetta, jonka tapahtuessa pirstoutuminen saattaisi olla suurta. Lisäksi kuten Bacon, Cheng ja Rajan [3] toteavat, analyysi toteutettiin ohjelmilla, joita suoritettiin vain varsin lyhyt aika, eikä analyysissä käytetty pitkään ajossa olevia ohjelmia, joita reaaliaikajärjestelmät usein ovat. Näin ollen Johonstonen ja Wilsonin tuloksiin ei reaaliaikajärjestelmissä voida tukeutua.

Reaaliaikajärjestelmissä korkean prioriteetin prosessit eivät saa epäonnistua. Käytettäessä dynaamista muistia reaaliaikajärjestelmissä on pidettävä huoli siitä, että muistinvarauspyyntö saadaan tyydytetyksi. Prosessien on käytettävä muistia rajoitetusti ja pirstoutuminen on estettävä, tai ainakin pirstoutuneen muistin määrän on oltava rajattu [40]. Sisäinen pirstoutuminen ei ole ongelma, mikäli se otetaan huomioon järjestelmää suunniteltaessa, sillä sisäisesti pirstoutuneen muistin määrä on aina rajallinen [46]. Ulkoista pirstoutumista ei pitkällä aikavälillä voida sallia lainkaan, sillä pirstoutumisen pahentuessa saavutetaan välttämättömästi tilanne, jolloin muistin varaaminen ei enää onnistu. Seuraavaksi tutustutaan siihen kuinka muistin pirstoutuminen voidaan estää.

3.2.1 Pirstoutumisen estäminen

Kuten luvussa 3.2 esitettiin, on muistin pirstoutuminen vakava ongelma reaaliaikajärjestelmissä, joissa käytetään dynaamista muistia. Pirstoutuminen on siis pakko estää. Pirstoutumisen estämiseksi on kaksi tapaa: muistia varataan vain kiinteän kokoisissa alueissa tai käytetään kopioivaa muistinsiivointa (kts. luku 4.3) tai kopioivan siivoimen johdannaista [50]. Näistä ensimmäinen poistaa vain ulkoisen pirstoutumisen, jälkimmäisellä tavalla on mahdollista poistaa sekä ulkoinen että sisäinen pirstoutuminen. Reaaliaikajärjestelmissä ulkoinen pirstoutuminen on eliminoitava täysin, mutta sisäistä pirstoutumista voidaan sietää mikäli sen määrä pysyy ennalta määrätyissä rajoissa [40]. Näin ollen molemmat tavat ovat hyväksyttäviä, vaikka on toivottavaa, että myös sisäinen pirstoutuminen eliminoidaan kokonaisuudessaan. Koska kopioivien siivointen toimintaperiaate on esitettynä luvussa 4.3, käydään tässä luvussa läpi vain kiinteän kokoisten alueiden käyttäminen.

Ulkoinen pirstoutuminen saadaan eliminoitua käyttämällä muistin varaamisessa kiinteänkokoisia alueita. Kiinteänkokoisten alueiden käyttämisellä tarkoitetaan, että järjestelmässä muisti varataan aina vain yhden tietyn ennalta määritellyn kokoisis-



Kuva 3.2: Paloiteltu tietue [50]

sa palasissa. Tässä strategiassa alueen kokoiset ja sitä pienemmät tietueet varataan suoraan, kun taas aluetta suuremmat tietueet varataan linkittämällä ne tarvittavasta määrästä palasia [50]. Tarpeen mukaan linkittäminen voidaan tehdä joko lineaarisena listana tai vaikka esimerkiksi puurakenteena. Lisäksi tällaisen koostetun tietueen ensimmäinen alue voi sisältää myös esimerkiksi tietueen tyyppi-informaatiota

Kuvassa 3.2 on esitettynä Siebertin [50] esimerkki tietuesta, joka on koostettu paloista. Tietue koostuu kolmesta lohkosta, joissa jokaisessa on neljä kenttää. Tietueessa itsessään on seitsemän kenttää hyötyinformaatiota. Ensimmäisen lohkon ensimmäisessä kentässä on tallennettuna tietueen tyyppi-informaatio. Ensimmäisen ja toisen lohkon viimeiset kentät ovat osoittimia tietueen seuraavaan lohkoon. Tällaisella pilkkomisella kolmanteen lohkoon jää kaksi kenttää sisäisesti pirstoutunutta muistia. Yhteensä kolme kenttää kuluu tietueen tyyppi-informaation ja rakenteen ylläpitämiseen. Jos kentän koko on neljä tavua, saadaan tietueen kooksi 48 tavua, joista vain 28 tavua on hyötykäytössä.

Tekniikalla on myös haittapuolensa. Ensinnäkin mikäli varatut tietueet eivät ole täsmälleen käytettävän alueen koon moninkertoja, ilmenee järjestelmässä sisäistä pirstoutumista. Sisäisen pirstoutumisen määrä on kuitenkin rajattu kaikilla varattavilla tietueilla, sillä sisäisesti pirstoutuneen muistin määrä yhtä tietuetta kohden on selvästi pienempi kuin käytettävä alueen koko [50]. Toiseksi kaikkiin lukuihin ja kirjoituksiin, joka koskevat tietueiden sellaisia kenttiä, jotka eivät ole ensimmäisen alueen sisällä, tulee ylimääräistä viivettä, sillä tällaisiin kenttiin ei päästä käsiksi suoraan, vaan käsittelyyn kuluva aika riippuu alueesta, jossa kenttä sijaitsee. Aika on lineaarinen kentän paikan suhteen. Kentän käsittelemiseen kuluva aika on $O(p)$, missä p on kentän paikka tietueessa. Tämä aika voidaan kuitenkin määrittää staattisella analyysillä, jolloin menetelmä on kelvollinen myös reaaliaikajärjestelmiin.

Taulukoiden esittäminen tuottaa ongelmia paloittelua käyttävissä järjestelmissä. Taulukoiden kokoaminen lineaarisena listana on erittäin tehotonta, ja vaikka taulukon jokaiselle alkiolle voidaankin määrätä kiinteä ja rajoitettu saantiaika, on se mitä toden-

näköisimmin liian suuri isojen taulukoiden kohdalla. Taulukot kannattaakin muodostaa puurakenteina, jolloin niiden alkioihin voidaan viitata logaritmisessa ajassa, toisin kuin listamaisessa rakenteessa, jossa tarvittava aika on lineaarinen. Sibert [50] esittää tällaisen rakenteen, jossa taulukot tallennetaan puuna, joissa lisäksi ensimmäiset palaset sisältävät myös informaatiota taulukoiden koosta. Tällaisen taulukon alkioon viittaaminen onnistuu ajassa $O(\ln(p))$, missä p on taulukon koko. Hän myös esittää algoritmin tällaisen taulukon alkioiden osoitteiden löytämiseen. Tämän algoritmin ohjelmakoodi koostuu vain kymmenestä ARM-prosessorin konekielikäskystä, jolloin algoritmia voidaan käyttää myös sisäisaliohjelmana, ilman että käännetyn ohjelman koko kasvaa liiaksi, olettaen, että ohjelmassa käytetään taulukoita kohtuudella.

Käytettäessä muistin varaamiseen vain kiinteän kokoisia palasia ongelmaksi muodostuu sisäinen pirstoutuminen, sillä kiinteiden alueiden kanssa sisäistä pirstoutumista esiintyy aina, kun muistia varattaessa muistin todellinen tarve on erisuuri kuin käytettävä alueen koko [60]. Sisäinen pirstoutuminen on kiusallista, mutta ei välttämättä suuri haitta, mikäli sen määrä saadaan minimoiduksi. Sisäisen pirstoutumisen määrään vaikuttaa ennen kaikkea muistin varaamiseen käytettävän alueen koko. Mikäli valittu lohkokoko on liian suuri, tulee kaikkiin varattuihin tietueisiin paljon sisäistä pirstoutumista. Liian pienet alueet taas tarkoittavat sitä, että alueita tarvitaan enemmän. Tällöin kasvaa alueiden linkittämiseen käytettävien osoittimien määrä, mahdollisten puolityhjien viimeisten palasten määrä sekä paloittelusta aiheutuva tietueen kenttien käsittelyviive. Siebert [50] toteaa, että parhaimmasta mahdollisesta lohkokoosta on lähes mahdotonta antaa arvioita, sillä lohkokoon valintaan vaikuttaa ohjelmien käyttämien tietueiden koko, sekä ohjelman muistinvarauskäyttäytyminen. Eri ohjelmat vaativat eri kokoisten kiinteiden alueiden käyttämistä. Suorituskyvyn kannalta parhaimpaan tulokseen päästään lohkoilla, joiden koko on kahden potenssin moninkerta.

3.2.2 Pirstoutumisen eliminoinnin hyödyt

Sen lisäksi, että kiinteänkokoisten alueiden käyttäminen eliminoi ulkoisen pirstoutumisen, se helpottaa myös muistinvaraamista. Kiinteänkokoisia alueita käytettäessä muistiallokaattorin ei tarvitse pitää listaa kuin yhden kokoisista alueista. Tämä voidaan tehdä esimerkiksi listalla, johon kaikki vapaat alueet on talletettuna. Muistia varattaessa otetaan listan alusta riittävä määrä alueita, linkitetään alueet ja palautetaan osoitin varattavan tietueen ensimmäiseen alkioon. Sama koskee luonnollisesti myös muistin vapauttamista. Vapautettavat lohkot vain liitetään takaisin listaan, joka sisältää vapaat muistialueet. Sekä muistin varaaminen että vapauttaminen saadaan tällöin tehtyä va-

kioajassa. Tämä aika vaihtelee tarvittavien lohkojen määrän mukaan.

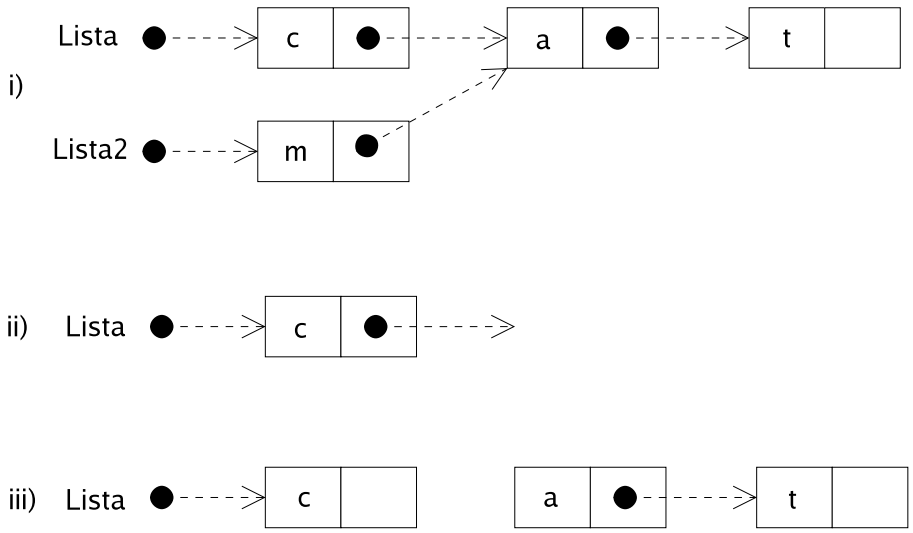
Kiinteiden alueiden käyttäminen myös helpottaa muistinsiivointen toteuttamista, koska siivoimen ei tarvitse tietää tietueiden varsinaista rakennetta, sillä riittää, että siivoin osaa käsitellä lohkoja [50]. Lohkojen alkuun voitaisiin sijoittaa bittitaulu, joka kertoo mitkä kyseisen lohkon kentistä ovat osoittimia. Tästä asiasta puhutaan lisää luvussa 4.5.2.

3.3 Muistin manuaalinen hallinta

Perinteisesti ohjelman käyttäessä dynaamisesti varattavaa muistia on ohjelmoija ollut vastuussa sekä muistin varaamisesta että vapauttamisesta. Muistin varaamisessa ei ongelmia ole, mutta vapauttaminen on virhealtista. Usein saattaa olla lähes mahdotonta tietää, milloin varattu muisti todellakin voidaan vapauttaa. Varmasti jokainen hieman enemmän ohjelmoinut on viettänyt tuskaisia hetkiä metsästäessään syitä siihen, miksi ohjelma kaatuu, koska jokin dynaamisesti luotu olio on vapautettu liian aikaisin, tai metsästäessään paikkaa, missä varattu muisti jää vapauttamatta.

Tarkastellaan tilannetta, joka on esitetty kuvan 3.3 kohdassa i). Kuvassa kahdella listalla on yhteinen häntä. Tuhottaessa jompaa kumpaa listojen ensimmäisistä solmuista pitäisi luonnollisesti myös koko listan loppuosa vapauttaa, mutta tämä aiheuttaisi toiseen listaan nk. *roikkuvan osoittimen (dangling pointer)*. Roikkuva osoitin on viite, joka osoittaa muistin sellaiseen osaan, joka on jo vapautettu [22, sivu 6]. Tätä on havainnollistettu kuvan 3.3 kohdassa ii). Roikkuvat osoittimet ovat sovelluksen toiminnan kannalta erittäin vakava ongelma. Parhaimmassa tapauksessa seurattaessa roikkuvaa osoitinta sovellus kaatuu ja virhe huomataan. Huonoimmassa tapauksessa sovellus saattaa roikkuvan osoittimen kautta tuhota tai muuttaa tietoa, joko omissa tietorakenteissaan tai muiden sovellusten tietorakenteissa.

Varatun muistialueen vapauttamatta jättäminen tarkoittaa *muistivuotoja (memory leak)* [58]. Muistivuodot eivät välttämättä ole paha asia, jos kyseessä on työpöytäohjelma, joka sammutetaan riittävän usein, mutta palvelinohjelmistoissa, jotka saattavat olla päällä jatkuvasti, muistivuodot todennäköisesti johtavat muistin loppumiseen ja siten virhetilanteeseen. Muistivuotoa kutsutaan *roskaksi (garbage)*, mistä johtunee muistinsiivouksesta yleisemmin käytetty termi roskien keruu. Kuvan 3.3 kohdassa iii) on havainnollistettu muistivuotoa, siinä viite listan ensimmäisestä solmusta c listan toiseen solmuun a on poistettu, mutta listasta poistettuja solmuja ei ole vapautettu, siis solmut a ja t ovat roskia.



Kuva 3.3: Manuaalisen muistinhallinan ongelmat [22, luku 1]

3.4 Automaattinen muistinhallinta – muistinsiivous

Automaattinen muistinhallinta on ratkaisu edellä esitettyihin manuaalisen muistinhallinnan ongelmiin. Automaattisessa muistinhallinnassa ohjelmoijan on vain varattava muisti, ja järjestelmä hoitaa muistin vapauttamisen silloin, kun se on turvallista [58]. Varattu muisti voidaan vapauttaa, kun siihen ei ole yhtään viittausta ohjelman käytössä olevasta datasta. Ohjelman käytössä olevan varatun muistin sanotaan olevan *elossa (live)*. Tietue on elossa, jos se kuuluu niin kutsuttuun *juurijoukkoon (root set)*, tai siihen on viite elossa olevasta tietueesta [22, sivu 4]. Varattu data, joka ei ole elossa, on roskaa.

Juurijoukolla tarkoitetaan niitä tietueita, jotka ovat sovelluksen välittömässä käytössä [22, sivu 4]. Tämä tarkoittaa prosessorin rekistereissä sekä ohjelman pinossa ja globaaleissa muuttujissa olevia viitteitä ja olioita. Elossa oleva data kuuluu juurijoukon transitiiviseen sulkeumaan. Se, kuinka muisti siivotaan, riippuu käytössä olevasta muistinsiivousalgoritmista. Algoritmien ja menetelmien välillä on eroja siinä, kuinka nopeasti datan kuoleman jälkeen sen varaama tila vapautetaan.

Muistinsiivouksen yhteydessä varsinaista sovellusta kutsutaan *muuntimeksi (mutator)* [13]. Muutin sisältää varsinaisen hyötyohjelman. Nimi muunnin johtaa juonensa siitä, että muistinhallinnan kannalta muutin vain muuttaa tietorakenteita, joita yritetään hallita.

Muistinsiivoimelta vaaditaan oikeellisuutta, sillä sen tehtävä on vapauttaa vain sel-

lainen muisti, jota ei enää käytetä, ja väärin toimiessaan siivoin saattaa aiheuttaa pahempia ongelmia kuin manuaalinen muistinhallinta. Lisäksi muistinsiivoimen tulee pystyä vapauttamaan kaikki roskaksi muuttunut muisti. Eri siivointen välillä on eroja tässäkin asiassa. Siinä, missä toiset menetelmät ovat yksinkertaisia ja tehokkaita tiettyjen tietorakenteiden kanssa, saattaa niissä olla ongelmia toisenlaisten rakenteiden kanssa. Luvussa 4 esitellään muistinsiivouksen perusalgoritmit, sekä kerrotaan millaisia ongelmia ja etuja kullakin menetelmällä on.

3.5 Muistinhallinnan uudet tuulet

Eräs muistinhallinnan uusista suuntauksista on *aluellinen muistinhallinta (region-based memory management)*, jossa yhdistyy pinosta varattavan muistin ennustettavuus ja keosta varattavan muistin automaattisen hallinnan turvallisuus [56]. Tässä muistinhallintatekniikassa käytetty muisti organisoidaan alueisiin, jotka voivat kasvaa suorituksen aikana. Näitä alueita hallitaan kekomaisesti. Toften ja Talpinin [56] alkuperäisessä ML-kielisessä toteutuksessa alueiden varaaminen ja vapauttaminen lisätään sovellusohjelmaan käännösaikaisesti ohjelmakoodin staattisen analyysin perusteella. Cyclone-kielessä on myös mahdollista hallinnoida alueita dynaamisesti [14]. Lisäksi ainakin Hallenberg, Elsman ja Tofte [16] ovat kehittäneen muistinsiivoimen, joka tukee alueellista muistinhallintaa. Alueelliseen muistinhallintaan ei tutkielmassa tarkemmin tutustuta.

Seuraavassa luvussa kerrotaan tarkemmin muistinsiivouksen toteuttamisesta.

4 Muistinsiivouksen toteutustavat ja ominaisuudet

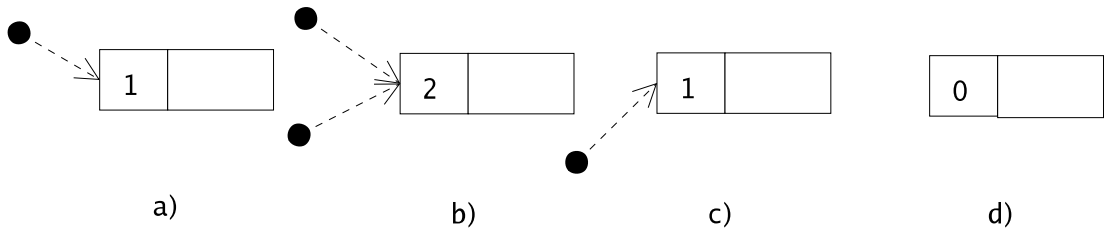
Muistinsiivousalgoritmit voidaan jakaa kahteen katgoriaan: *välittömiin (direct)* ja *jäljittäviin (tracing)* algoritmeihin [22, sivu 5]. Välittömät algoritmit ovat poikkeuksetta viitelaskuritekniikkaan perustuvia, ja niissä kuolevat tietueet tunnistetaan välittömästi tietueen kuollessa. Jäljittäviä muistinsiivousalgoritmeja ovat käytännössä kaikki loput tekniikat, ja niissä elossa oleva data tunnistetaan seuraamalla rekursiivisesti kaikki osoittimet, joihin juurijoukosta päästään.

Tässä luvussa käydään läpi muistinsiivouksen perusalgoritmit, joihin kaikki käytössä olevat muistinsiivousmenetelmät perustuvat. Aluksi tutustutaan viitelaskuritekniikkaan, jonka jälkeen siirrytään tarkastelemaan jäljittäviä tekniikoita. Perusalgoritmien jälkeen esitellään lyhyesti muita muistinsiivouksen menetelmiä, kerrotaan muistinsiivoukseen liittyvistä käsitteistä sekä muistinsiivouksen tuomista ongelmista. Tutkielmassa algoritmeja esitellään vain yleisesti, eikä minkäänlaista luonnoskoodia niiden toiminnasta anneta. Tarkemman kuvauksen algoritmien toiminnasta antaa mm. Jonesin ja Linsin kirja [22], jossa alla esitettävät algoritmit kuvataan myös pseudokoodina. Kyseisessä kirjassa algoritmeista esitetään myös useita eri variaatioita.

4.1 Viitelaskuri

Yksinkertaisin muistinsiivousmenetelmä on *viitelaskuri (reference counting)*. Viitelaskurissa jokaiseen olioon lisätään sovellusohjelmalle näkymätön kenttä, jossa ylläpidetään lukumäärää niistä osoittimista, jotka kyseiseen olioon viittaavat [58]. Laskuria kasvatetaan aina, kun olioon osoitetaan uusi viite. Laskurin arvoa vähennetään, kun viite olioon poistetaan. Laskurin saatua arvokseen nolla tiedetään, että kyseiseen olioon ei enää ole viitteitä, ja olio voidaan turvallisesti vapauttaa.

Kuvassa 4.1 on neliosaisella esimerkillä selvitetty viitelaskurin toimintaa. Kohdassa a) tietueeseen osoittaa yksi osoitin, jolloin viitelaskurin arvo on yksi. Kohdassa b) tietueeseen osoitetaan toinen osoitin, jolloin viitelaskuri kasvatetaan arvoon kaksi. Kohdassa c) ensimmäinen osoitin, joka tietueeseen osoitti poistetaan käytöstä, ja viitelaskurin arvoa vähennetään yhdellä. Laskuri saa tällöin arvon yksi. Kohdassa d) poistetaan viimeinen tietueeseen osoittava osoitin, ja laskuri saa arvokseen nolla, jolloin tiedetään,



Kuva 4.1: Esimerkki viitelaskurin toiminnasta

että tietue on roskaa ja sen varaama muistialue vapautetaan.

Viitelaskurin eduksi voidaan mainita sen kyky havaita olion kuoleminen välittömästi, kun viimeinen viite olioon poistuu [22, sivu 23]. Lisäksi viitelaskurin vaatima suoritusaika sulautuu yhteen varsinaisen sovellusohjelman suorituksen kanssa, eikä sovellusta erikseen pysäytetä muistin siivoamisen ajaksi [22, sivu 21]. Näistä jälkimmäinen voi tosin aiheuttaa myös ongelmia, varsinkin siinä tapauksessa, että viimeinen viite jonkin tietorakenteen juureen, kuten esimerkiksi linkitetyn listan ensimmäiseen alkioon, poistetaan. Tässä tapauksessa on koko tietorakenne vapautettava, mikäli listan alkiohin ei ole listan ulkopuolisia viitteitä. Vapauttamiseen kuluva aika on lineaarinen tietorakenteen koon suhteen, ja rakenteen ollessa riittävän suuri operaatio kestää kauan, mikä voi osaltaan häiritä sovelluksen suorittamista.

Suurin ongelma viitelaskuritekniikassa on kuitenkin se, että sen avulla ei kyetä vapauttamaan syklisiä tietorakenteita [58]. Vaikka sykliset tietorakenteet saattavat kuulostaa harvinaisilta, kannattaa huomioida, että tällaisia tietorakenteita ovat esimerkiksi kahteen suuntaan linkitetty lista ja puu, jonka solmuissa on viite takaisin solmun vanhempaan [22, sivu 24]. Tällaisessa tapauksessa, vaikka ulkoiset viitteet kahteen suuntaan linkitettyyn listaan poistetaankin, jää listan päässä sijaitsevan alkion laskurin arvoksi yksi, eikä listaa näin ollen vapauteta.

Viitelaskuri voidaan yhdistää johonkin myöhemmin esiteltävään tekniikkaan, jolloin viitelaskuria käytetään normaaliin tapaan, ja tätä toista tekniikkaa käytetään vapauttamaan se osuus varatusta muistista, jonka vapauttamiseen viitelaskuri ei pysty. Käytettäessä viitelaskurin apuna jotain toista tekniikkaa vähenee viitelaskurista saatavat hyödyt. Lisäksi viitelaskurista on kehitelty paranneltuja muunnoksia, joissa edellä mainittuja ongelmia on korjattu. Näihin muunnelmiin ei tässä tutkielmassa tutustuta.

4.2 Merkkaa ja lakaise

Merkkaa ja lakaise -algoritmi (mark-sweep -algorithm) on muistinsiivousmenetelmistä vanhin, ja sen kehitti John McCarthy alkuperäistä LISP-kieltä varten [22, sivu 25]. Tässä menetelmässä siivous tapahtuu kahdessa osassa [58]. Aluksi merkitään kaikki elossa oleva muisti. Tämä tapahtuu seuraamalla rekursiivisesti kaikkia osoittimia, joihin päästään aloittaen juurijoukon muuttujista, ja asettamalla kaikkiin saavutettuihin olioihin merkki siitä, että kyseinen olio on elossa [22, sivu 25]. Kun kaikki juurijoukon transitiiviseen sulkeumaan sisältyvät tietueet on merkitty, tiedetään, että merkkaamaton muisti on roskaa, jolloin se voidaan huoletta vapauttaa. Toisessa vaiheessa keko käydään läpi, ja merkkaamattomat muistialueet vapautetaan. Algoritmin nimen mukaisesti ensimmäistä osaa, elossa olevien tietueiden merkkausta, kutsutaan merkkaukseksi ja jälkimmäistä osaa, roskien vapauttamista, lakaisuksi.

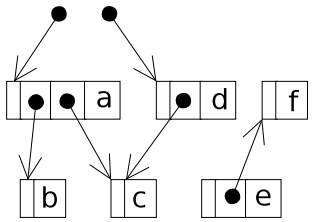
Tutustutaan merkkaa ja lakaise -algoritmin toimintaan esimerkin avulla. Kuvassa 4.2 on tilanne, josta esimerkki aloitetaan. Esimerkissämme on kuusi tietuetta ja juurijoukko, joka sisältää kaksi osoitinta. Tietueet merkitään "värjäämällä" merkkibitti, joka sijaitsee kunkin tietueen alussa. Merkattua tietuetta kuvaa mustaksi värjätty merkki, ja merkkaamatonta tietuetta valkoinen merkki. Aloitustilanteessa kaikki tietueet ovat merkkaamattomia.

Merkkausvaihe etenee sillä tavalla, että vasemmanpuoleisesta juurijoukon muuttujasta edetään *syvyyshaulla (depth first search)*, ja merkitään kaikki tietueet, joihin päästään. Aluksi siis merkitään tietue a. Tämän merkkauksen jälkeinen tilanne on kuvassa 4.3. Tästä jatketaan merkkaamalla tietue b, jonka jälkeen merkataan tietue c. Näiden merkkausten jälkeinen tilanne on esitettynä kuvassa 4.4. Merkattuaan tietueen c, siivoin siirtyy seuraavaan juurijoukon muuttujaan, sillä ensimmäisen juurijoukon muuttujan avulla ei uusia tietueita enää saavuteta.

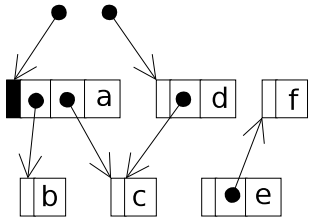
Oikeanpuoleisen ja samalla viimeisen juurijoukon muuttujan kautta saadaan merkattua tietue d. Kuvassa 4.5 on tilanne, johon päädytään, kun kyseinen tietue on merkattu. Tämä on samalla myös merkkausvaiheen loppu, sillä kaikki elossa olevat tietueet on merkattu. Merkkausvaihe lopetetaan ja muisti lakaistaan. Esimerkissä roskia ovat tietueet e ja f, joiden varaama tila vapautetaan lakaisun yhteydessä.

Esimerkissä elävä muisti käytiin läpi syvyyshaulla, mutta yhtälailla oltaisiin voitu käyttää *leveyshakua (breadth first search)*, kunhan vain kaikki elossa oleva muisti olisi saatu merkattua.

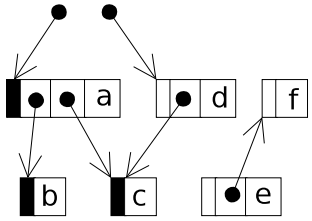
Perusmuodossaan merkkaa ja lakaise -algoritmi toimii siten, että muistivuotojen annetaan kasvaa, kunnes varatun muistin määrä ylittää tietyn rajan, ja rajan ylittyes-



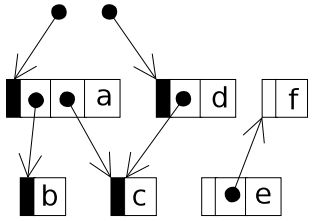
Kuva 4.2: Merkkaa ja lakaise 1



Kuva 4.3: Merkkaa ja lakaise 2



Kuva 4.4: Merkkaa ja lakaise 3



Kuva 4.5: Merkkaa ja lakaise 4

sä sovellusohjelman suoritus pysäytetään ja muisti siivotaan [22, sivu 25]. Aika, joka muistin siivoamiseen kuluu, on suhteessa keon kokoon [58]. Aluksi elossa oleva data on tunnistettava ja tämän jälkeen kuolleet tietueet vapautettava. Tämä vaatii keon läpikäymisen vähintään kaksi kertaa. Huonoimmillaan siivoamiseen kuluva pysähdys voi olla erittäin suuri, mistä johtuen menetelmä perusmuodossaan ei sovellu sovelluksiin, joissa kaivataan lyhyttä vasteaikaa [22, sivu 28].

Merkkaa ja lakaise -tekniikalla voidaan siivota myös sykliset tietorakenteet ilman mitään erillisiä tekniikoita, mutta elävien tietueiden merkkaamisen rekursiivinen luonne tarkoittaa sitä, että menetelmä itsessään vaatii muistia, johon talletetaan merkkaamisen vaatima väliaikainen tietorakenne [22, sivu 28].

4.3 Kopioivat siivoimet

Kopioivat muistinsiivoimet (copying garbage collection) perustuvat periaatteessa samaan ideaan kuin merkkaa ja lakaise -algoritmikin, mutta sekä toteutus että lopputulos eroavat toisistaan täydellisesti. Kopiovat menetelmät eivät varsinaisesti vapauta varattua muistia, vaan ne siirtävät elossa olevat tietueet yhtenäiseksi alueeksi, jonka jälkeen loppuosan tiedetään olevan uudelleenkäytettävissä, sillä se sisältää vain roskaa [58]. Yhteistä kopioivilla siivoimilla ja merkkaa ja lakaise -tekniikalla on se, että molemmat käyvät oliota läpi juurijoukosta aloittaen, mutta kaikki muut siivoimien toiminnot ovat täysin erilaisia.

Seuraavaksi Cheneyn kopioivan siivoimen toiminta esitetään Jonesin ja Linsin mukaan [22, sivu 28]. Kopioivissa tekniikoissa muisti jaetaan kahteen yhtä suureen osaan: *lähdeavaruuteen (fromspace)* ja *kohdeavaruuteen (tospace)*. Muistiavaruuksista muutin näkee vain kohdeavaruuden. Näistä jälkimmäinen sisältää kaiken käytössä olevan datan sekä vapaan muistialueen. Varsinainen muistinsiivous aloitetaan vaihtamalla kohde- ja lähdeavaruus keskenään. Tämän jälkeen kaikki juurijoukon osoittamat tietueet kopioidaan lähdeavaruudesta kohdeavaruuden alkuun. Nyt aloitetaan käydä kohdeavaruuteen kopioituja tietueita järjestyksessä läpi. Tietueen kaikki välittömät jälkeläiset kopioidaan aiemmin kopioitujen tietueiden perään, ja siirrytään käsittelemään seuraavaa tietuetta. Kun kaikki kohdeavaruudessa olevat tietueet on käyty läpi, on muistinsiivous suoritettu, ja lähdeavaruus sisältää vain roskaa.

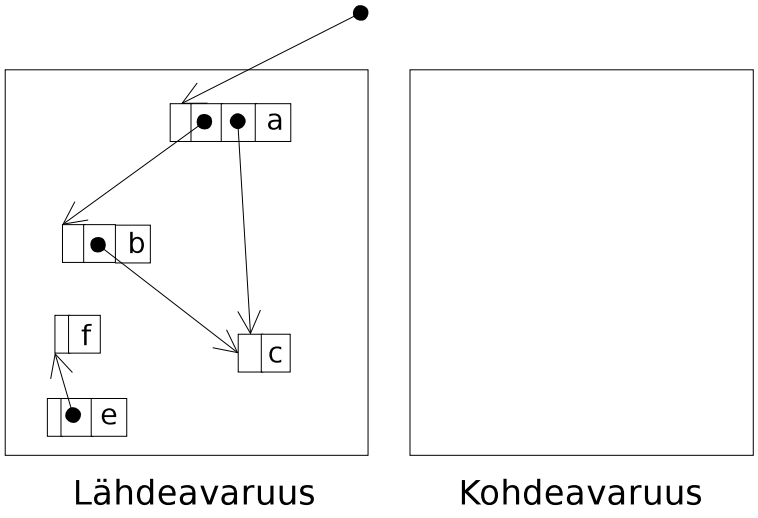
Menetelmän kannalta kaikkein tärkein operaatio on kopiointi. Jottei datasta pääsisi syntymään ylimääräisiä kopioita, ei kopiointia voida suorittaa ilman varotoimenpiteitä. Kun tietuetta kopioidaan, sen alkuperäiselle paikalle lähdeavaruuteen jätetään osoitin,

joka osoittaa tietueen uuteen kohdeavaruuden osoitteeseen [58]. Tätä osoitinta kutsutaan *kopio-osoittimeksi (forwarding pointer)*. Mikäli tietue on jo kopioitu, ei kopiointia suoriteta uudelleen, vaan päivitetään se osoitin, josta tietueeseen päädyttiin, osoittamaan tietueen uuteen lähdeavaruuden osoitteeseen. Tämä osoite saadaan tietueen kopio-osoittimesta.

Edellä esitetyssä muodossaan kopioiva siivoin voi käyttää korkeintaan puolet muistista [22, sivu 32]. Ylimääräisen muistin tarve on siis melkoinen. Kopioivat siivoimet pakkaavat käytössä olevan muistin niin, ettei siihen jää aukkoja. Samalla tekniikka jättää myös kaiken vapaan muistin yhtenäiseksi. Näin ollen muistin varaaminen on nopeaa ja helppoa, sillä vapaasta alueesta voidaan yksinkertaisesti lohkaista tarvittava alue [22, sivu 32]. Muistin tiivistämisen ansiosta kopioivat siivoimet eliminoivat sekä ulkoisen että sisäisen pirstoutumisen. Lisäksi kopioivan siivoimen muistinsiivoukseen käyttämä työ on verrannollinen elävän datan määrään [58]. Tällä on huomattava ero verrattuna esimerkiksi merkkaa ja lakaise -tekniikalla toimiviin siivoimiin. Merkkaa ja lakaise -tekniikalla toimivia ja kopioivia siivoimia on vaikea verrata keskenään, sillä kopiointi on hitaampaa kuin pelkkä tietueen merkkaaminen, mutta kopioivien siivoimien muut edut puolustavat niiden käyttöä [22, sivu 35].

Kopioivissa siivoimissa on, pirstoutumisen eliminoinnin lisäksi, yksi suuri etu verrattuna perinteiseen merkkaa ja lakaise -siivoimeen, sillä kopioivissa siivoimissa muistia ei erikseen vapauteta. Tämä on mahdollista, koska elossa oleva data pakataan yhtenäiseksi jolloin tiedetään, että loppuosa on roskaa ja sen yli voidaan huoletta kirjoittaa. Merkkaa ja lakaise -tekniikalla toimivissa siivoimissa on merkkausvaiheen jälkeen käytävä keko läpi, ja vapautettava kaikki merkkaamattomat alueet. Kopioivat siivoimet siis vapauttavat muistin implisiittisesti, kun merkkaa ja lakaise -tekniikassa muisti vapautetaan siivoimen toimesta eksplisiittisesti. Kopioivissa siivoimissa ei näin ollen tarvitse käyttää suoritusaikaa muistin vapauttamiseen. Jones ja Lins [22, sivu 36] toteavat, että kopioivat siivoimet ovat olleet käytetympiä, mutta viittaavat useisiin lähteisiin, joiden mittauksissa on todettu, että muistinsiivoimen valintaan vaikuttaa huomattavasti muistinsiivoimen toteutus ja muuntimen käyttäytyminen.

Seuraavaksi esitetään esimerkin avulla kopioivan siivoimen toiminta. Esimerkin alkutilanne on kuvassa 4.6 ja se sisältää viisi tietuetta sekä juurijoukon, johon kuuluu vain yksi muuttuja. Lisäksi esimerkissä käytetään seuraavanlaisia merkintöjä: tietueiden sisältämät osoittimet merkitään nuolella, jonka viiva on kiinteä, ja siivoimen tekemät tietueen kopion uutta osoitetta ilmaisevat kopio-osoittimet ovat kuvissa nuolia, joissa on käytetty katkoviivoja.

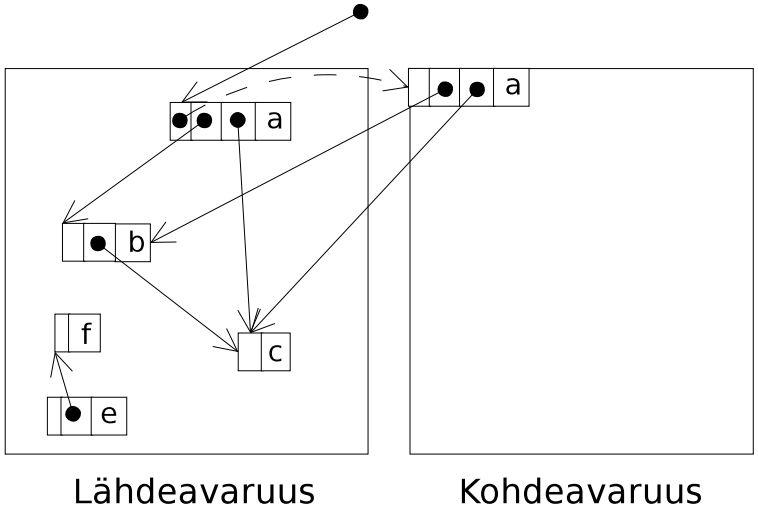


Kuva 4.6: Kopioiva siivoin 1

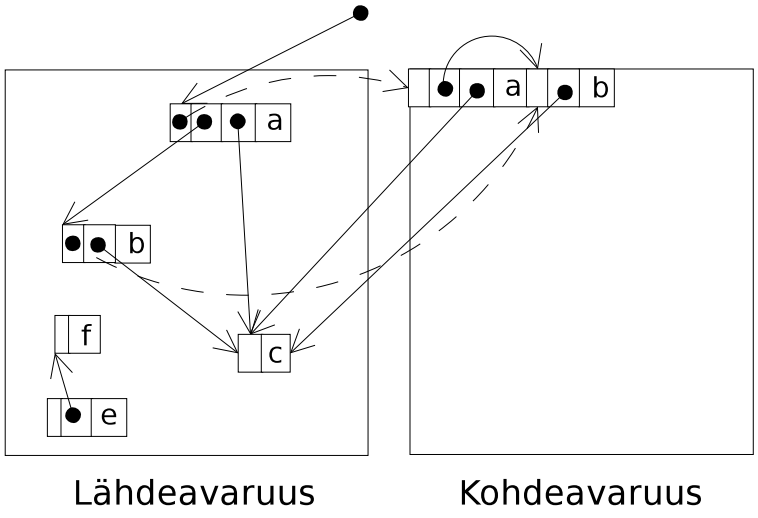
Siivous etenee seuraavasti: Aluksi kopioidaan juurijoukon jälkeläiset kohdeavaruuteen, esimerkissämme tämä tarkoittaa tietuetta a. Tietue a siis kopioidaan kohdeavaruuteen ja lähdeavaruuteen luodaan kopio-osoitin osoittamaan kopion paikkaa kohdeavaruudessa (kuva 4.7). Tämän jälkeen aletaan käydä kohdeavaruuteen kopioituja tietueita läpi ja aina, kun vastaan tulee osoitin, kopioidaan osoittimen päästä löytyvä tietue kohdeavaruuteen jo kopioitujen tietueiden perään ja päivitetään osoitin, josta kyseiseen tietueeseen päästiin.

Toisessa vaiheessa kohdeavaruuteen kopioidaan tietue b, ja lähdeavaruuden tietueeseen b lisätään kopio-osoitin, joka osoittaa kohdeavaruuden kopioon. Samalla myös päivitetään kohdeavaruuden tietueessa a olevaa osoitinta, jonka kautta tietueeseen b päästiin. Tämä tilanne on havainnollistettuna kuvassa 4.8. Seuraavaksi siivoimemme löytää osoittimen, joka osoittaa lähdeavaruuden tietuetta c. Tämä tietue c kopioidaan kohdeavaruuteen, lähdeavaruuteen jätetään osoitin osoittamaan kopion osoitetta ja kohdeavaruuden tietueen a osoittimen arvo päivitetään osoittamaan juuri kopioituun tietueeseen. Tämä tilanne on esitettynä kuvassa 4.9.

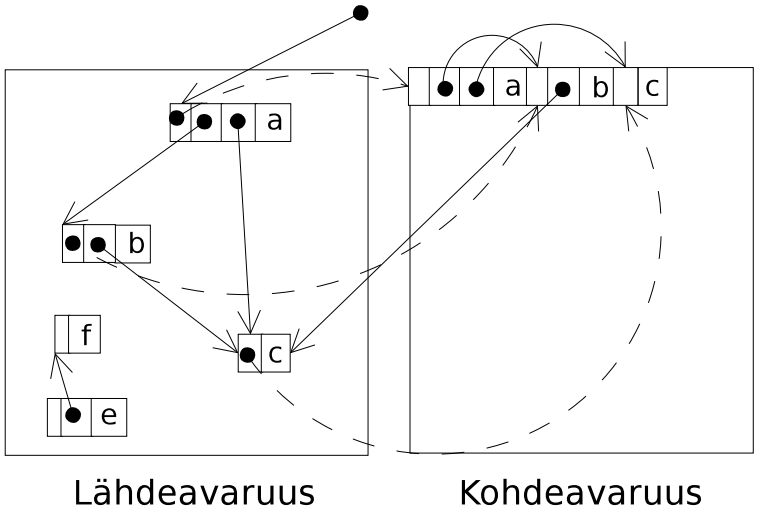
Nyt kohdeavaruudessa sijaitsevan tietueen a jälkeläiset on saatu kokonaisuudessaan kopioitua, ja siivoin siirtyy käsittelemään kohdeavaruuden tietuetta b. Tietueesta b löytyy osoitin, joka osoittaa lähdeavaruuden tietueeseen c. Koska tämä tietue c on jo



Kuva 4.7: Kopioiva siivoin 2



Kuva 4.8: Kopioiva siivoin 3

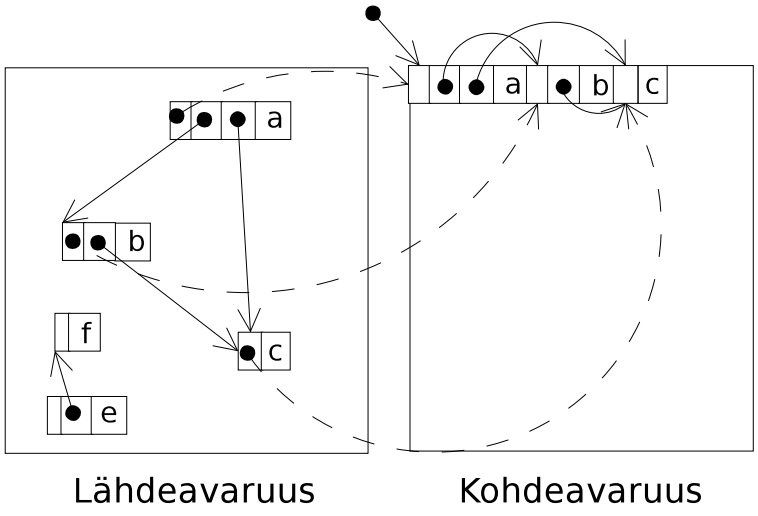


Kuva 4.9: Kopioiva siivoin 4

kopioitu, ei kopiointia suoriteta uudelleen, vaan tietueesta b löytyvä osoitin päivitetään osoittamaan samaan paikkaan johon lähdeavaruudessa olevan tietueen c kopio-osoitin osoittaa. Tämän toiminnon jälkeinen tilanne on havainnollistettuna kuvassa 4.10. Tilanne myös päättää käynnissä olevan muistinsiivoussyklin, sillä kohdeavaruudesta ei enää löydy osoittimia, joita pitkin päästään lähdeavaruuteen. Kaikki käytetty muisti on pakattuna kohdeavaruuden alkuun ja loppuosa kohdeavaruudesta on yhtenäistä vapaata aluetta, josta voidaan helposti lohkaista pala, kun muistia varataan. Lähdeavaruus sen sijaan sisältää vain roskaa ja sellaisia tietueita, jotka on kopioitu kohdeavaruuteen, joten lähdeavaruudessa olevat tietueet voidaan siis huoletta ylikirjoittaa seuraavan muistinsiivoussyklin aikana.

4.4 Muut siivoustekniikat

Seuraavaksi esitetään pintapuolisesti kaksi muistinsiivoustekniikkaa, jotka poikkeavat perusalgoritmien toiminnasta. Ensimmäisenä näistä tekniikoista on merkkaa ja tiivistä -tekniikka (luku 4.4.1) ja jälkimmäisenä ikäperustainen siivous (luku 4.4.2). Esitettävien tekniikoiden lisäksi on olemassa myös nk. *vähittäinen siivous (incremental garbage*



Kuva 4.10: Kopioiva siivoin 5

collection). Vähittäisen siivouksen toimintaan paneudutaan luvussa 5.1.

4.4.1 Merkkaa ja tiivistä

Merkkaa ja tiivistä -algoritmit (mark-compact collection) perustuvat samaan ideaan kuin merkkaa ja lakaiset -tekniikkakin; tuloksena saadaan keko, jonka varatut muistialueet sijaitsevat yhtenäisenä alueena keon alussa [58]. Merkkaa ja tiivistä -tekniika toimii siten, että aluksi merkataan elossa oleva data, jonka jälkeen elävät tietueet tiivistetään yhtenäiseksi alueeksi. Yksinkertaisimmillaan tiivistys tapahtuu siten, että tietueita liutetaan kohti keon alkua, kunnes ne ovat joko edellisen varatun alueen vieressä tai keon alussa.

Merkkaa ja tiivistä -algoritmit eliminoivat ulkoisen pirstoutumisen. Lisäksi etuna verrattaessa kopioiviin siivoimiin on se, että merkkaa ja tiivistä -algoritmit mahdollistavat keon käyttämisen kokonaisuudessaan, sillä tiivistys tehdään keon alkuun, eikä ylimääräistä puoliavaruutta vaadita. Toisaalta tiivistäminen on huomattavasti kopiointia vaativampaa, ja merkkaa ja tiivistä -algoritmien on käytävä keko useita kertoja läpi, jotta operaatio saadaan vietyä loppuun [22]. Merkkaa ja tiivistä -algoritmeja ei tässä tutkielmassa tämän enempää käsitellä.

4.4.2 Ikäperustainen siivous

Ikäperustainen siivous (generational garbage collection) perustuu tiedolle siitä, että useimmat tietueet elävät vain lyhyen ajan. Kuten Wilson raportissaan useista eri lähteistä kokoaa, yleensä 80–98 prosenttia uusista tietuesta kuolee muutaman miljoonan käskyn sisällä tai ennen kuin muistia on varattu megatavu lisää, ja useimmat näistä uusista tietueista kuolevat vieläkin nopeammin [58].

Ikäperustaisissa siivousmenetelmissä varattu muisti jaetaan kahteen tai useampaan osaan sen perusteella, kuinka kauan niissä asustavat tietueet ovat olleet elossa [22]. Näissä *sukupolvissa (generation)* siivousta suoritetaan eri tahdilla. Sellaisissa sukupolvissa, jotka sisältävät nuoria tietueita, siivousta suoritetaan useammin, ja sukupolvissa, joissa on vanhempia tietueita, siivotaan harvemmin. Nuoren tietueen säilyttyä hengissä muutaman muistinsiivoussyklin ajan oletetaan, että kyseinen tietue ei tule lähiaikoina kuolemaan, ja se ylennetään vanhempaan sukupolveen.

Vain nuoria tietueita siivoavat syklit ovat usein lyhyitä, eivätkä yleensä aiheuta riittävän pitkään pysähdystä, jotta käyttäjä sen huomaisi [58]. Toisaalta aina silloin tällöin, kun vanha sukupolvi on täyttymässä, on ikäperustaisissa muistinsiivoimissa tehtävä suurempi, koko keon kattava siivous, jonka suoritusaika ei enää ole lyhyt. Ikäperustaiset siivoimet ovatkin tehokas tapa parantaa muistinsiivoimen keskiarvoista vasteaikaa, mutta reaaliaikajärjestelmiin ne eivät sovellu [22, sivu 184].

4.5 Siivouksen ongelmista

Muistinsiivouksen käytöstä ei seuraa pelkästään hyvää, sillä siitä aiheutuu myös haittoja. Seuraavissa luvuissa tutustutaan siivouksen negatiivisiin vaikutuksiin ja siivouksen toteuttamiseen vaikuttaviin asioihin.

4.5.1 Muistinsiivouksen vaikutus ohjelman suorituskykyyn

Muistinsiivouksen maine ei ole kovin mairitteleva, sillä yleisesti oletetaan, että siivous vaikuttaa negatiivisesti ohjelman suorituskykyyn [22, sivu 12]. Tämän oletuksen taustalla on muistinsiivouksen ja LISP-kielen alkutaipaleet, jolloin joidenkin tutkimusten mukaan ohjelmat saattoivat käyttää jopa 40 prosenttia suoritusajastaan muistia siivoten. Nykyaikana muistinsiivoustekniikat ovat kuitenkin kehittyneet niin paljon, että nämä luvut eivät pidä paikkaansa. Wilson [58] arvioi muistinsiivouksen aiheuttavan noin 10 prosentin hidastuksen verrattuna manuaaliseen muistinhallintaan. Appel [1]

esittää, että joissain tapauksissa, kun järjestelmässä on tarpeeksi muistia, on muistinsiivous jopa tehokkaampaa kuin muistin manuaalinen vapauttaminen.

Muistinsiivouksen varsinaista tehokkuutta on kuitenkin vaikea mitata, sillä muistinsiivouksen käyttäminen vaikuttaa useaan eri asiaan. Ohjelman suoritusnopeuteen vaikuttaa aika, joka kuluu tietueiden varaamiseen. Muistinsiivoimissa, jotka tiivistävät käytetyn muistin, on muistin varaaminen yleensä huomattavasti nopeampaa kuin perinteisissä muistiallokaattoreissa [22, sivu 13]. On myös otettava huomioon sovelluskehitykseen kuluva aika. Manuaalista muistinhallintaa käyttävien ohjelmien kehittäminen on usein työläämpää kuin vastaavan ohjelman toteuttaminen automaattisen muistinhallinnan kanssa [58].

Myös eri tekniikoilla toimivien muistinsiivoimien vertailu on vaikeaa [22, sivu 13]. Vaikka algoritmien kompleksisuuden vertailu on yleensä mahdollista, ei teoria kuitenkaan vastaa käytäntöä. Usein algoritmien kompleksisuusanalyysissä esiintyvillä vakiolla on suuri merkitys siihen, kuinka algoritmit toimivat käytännössä. Lisäksi muistinsiivoustekniikat saattavat aiheuttaa muuntimen toiminnalle muita hidasteita. Esimerkiksi viitelaskurissa kaikki osoittimien muutokset tulee huomioida ja laskureiden arvoja muuttaa. Muistinsiivouksen tehokkuutta voidaan usein myös parantaa kasvattamalla muistin määrää.

4.5.2 Osoittimien tunnistaminen

Erittäin kriittinen asia muistinsiivouksen kannalta on se, kuinka muuttujien tyyppi ja osoittimet tunnistetaan. Muistinsiivousta käsittelevässä kirjallisuudessa tehdään usein oletus, että muuttujien tyyppi on aina tiedossa. Wilsonin raportti [58] on hyvä esimerkki tällaisesta kirjoituksesta. Usein todellisuus ei kuitenkaan vastaa oletusta, sillä esimerkiksi yleiset C-kielen toteutukset eivät vakiona tarjoa minkäänlaisia keinoja osoittimien tunnistamiseen.

Muistinsiivoimen tehokkuuteen tunnistaa osoittimet vaikuttaa siivoimen *tyyppitietoisuus (type accuracy)*, joka kuvaa siivoimen kykyä tunnistaa osoittimet eroon atomaarisista alkioista [19]. Ääritapauksissa siivoin tietää kaikkien muuttujien tyypin, jolloin puhutaan täysin tietoisista siivoimista. Toisessa ääripäässä on konservatiiviset siivoimet, joilla ei ole minkäänlaista tietoa muuttujien tyypistä.

Erityistä merkitystä osoittimien tunnistamisella on siivoimissa jotka liikuttavat muistissa olevaa tietoa [22, sivu 37]. Tietoa liikuttavilla siivoimilla tarkoitetaan tässä yhteydessä sekä kopioivia että tiivistäviä siivoimia, sekä näiden johdannaisia. Näissä siivoimissa juurijoukko ja kaikki osoittimet on tunnistettava täydellisesti. Jos osoitinta

ei tunnisteta voi järjestelmään syntyä leijuvia osoittimia tai pahimmassa tapauksessa tämä voi johtaa myös jonkin olion ennenaikaiseen vapauttamiseen. Mikäli atomia, joka ei ole osoitin, luullaan osoittimeksi, saatetaan tätä atomia siirtämisen yhteydessä päivittää, jolloin järjestelmän toiminta vaarantuu.

Tyyppitietoisten muistinsiivoimien toteuttamisen lähtökohtana on usein yhteistyö ohjelmointikielen kääntäjän kanssa [22, sivu 228]. Tällöin käännöksen yhteydessä kerätään siivoimen tarvitsemien tiedot. Toinen vaihtoehto on se, että ohjelmoijien edellytetään keräävän nämä tiedot käsityönä. Sanomattakin on selvää, että jälkimmäinen tapa on virhealtista ja aikaa vievää, mutta kaikissa tapauksissa ei kääntäjältä tarvittavaa tietoa saada.

Osoittimien tunnistaminen voidaan tehdä muutamalla eri tavalla [15]. Osoittimien laputuksen perusideana on se, että alkiot merkitään merkkibitillä, joka kertoo onko kyseessä atomi vai osoitin. Muuttujankuvaimissa jokaiseen muuttujaan lisätään osoitin, joka osoittaa tietorakenteeseen, joka kertoo muuttujan sisällön. Tyypinkuvaimissa tyyppi-informaatio asetetaan ohjelman staattisen tiedon joukkoon, josta siivoin sen lukee. Räätälöidyissä siivoimissa kääntäjä generoi jokaiselle erilaiselle tietuetyypille oman merkkausaliohjelman. Eräs esimerkki räätälöidystä merkkaa ja lakaise -tekniikalla toimivasta muistinsiivoimesta on Colnetin, Coudaudun ja Zendran Eiffel-kieltä varten toteuttama siivoin, jossa käännösvaiheessa jokaiselle oliotyypille luodaan oma merkkausaliohjelma [10].

Tapauksissa, joissa tyyppi-infomaatiota ei ole saatavilla, ratkaisu löytyy *konservatiivisista muistinsiivoimista (conservative garbage collection)*. Nämä siivoimet pystyvät toimimaan ilman minkäänlaista ajonaikaista tukea. Kyseisissä siivoimissa kaikkia mahdollisia muistista löytyviä arvoja tutkitaan ja niistä koitetaan päätellä onko kyseessä osoitin vai ei. Mikäli asiasta on pienikin epäselvyys, tulkitaan arvo osoittimeksi ja oletetun osoittimen osoittama muistialue säästetään. Konservatismi aiheuttaa järjestelmään nk. leijuvaa roskaa, jonka vaikutuksiin palaamme myöhemmin vähittäisen siivouksen yhteydessä luvussa 5.1.3. Konservatiiviset siivoimet ovat tämän tutkielman aihepiirin ulkopuolella.

4.5.3 Finalisaatio

Oliopohjaisissa ohjelmointikielissä on usein tarve suorittaa ylimääräisiä siivoustoimenpiteitä, kun olio vapautetaan. Muistinsiivouksen yhteydessä näitä automaattisesti tapahtuvia toimenpiteitä kutsutaan *finalisaatioksi (finalization)* [8]. Perinteisissä, manuaalisesta muistinhallintaa käyttävissä, ohjelmointikielissä puhutaan yleensä olioiden

tuhoamisesta (destruction) [21]. Yhteistä näille molemmille on se, että niitä käytetään, kun oliot varaavat käyttöönsä resursseja, jotka on vapautettava ennen kuin resursseja voidaan käyttää uudelleen. Olioiden varaamien resurssien hallinnointi on tällä tavoin helpompaa. Usein resurssit varataan olion konstruktorissa ja vapautetaan olion destruktorissa tai finalisaattorissa. C++:n yhteydessä tällaista tapaa hallita resursseja kutsutaan RAII:ksi (resource acquisition is initialization) [54, sivu 366]. Esimerkiksi olion käyttämä tiedosto avataan konstruktorissa ja suljetaan destruktorissa. Tuhoamisen ja finalisaation erona on se, että tuhoaminen tapahtuu synkronisesti ohjelman suorituksen kanssa, kun taas finalisointi on asynkronista [8].

Finalisaation asynkroninen luonne johtuu siitä, että ei voida täysin varmasti tietää sitä hetkeä, jolloin olion muisti vapautetaan ja finalisaatio suoritetaan [8]. Pahimmassa tapauksessa finalisaatio viivästyy siihen asti, kun sovellus sammutetaan. Asynkronisuuteen on kuitenkin poikkeus, sillä viitelaskuritekniikka mahdollistaa finalisaation suorittamisen synkronisesti, sillä finalisointi voidaan suorittaa välittömästi, kun olion viitelaskuri saa arvokseen nolla. Finalisaation asynkroninen luonne aiheuttaa ongelmia varsinkin reaaliaikajärjestelmissä, sillä finalisaatio voi tapahtua periaatteessa milloin vain ja aiheuttaa järjestelmään viiveitä, joita on mahdotonta ennustaa. Myös olioiden finalisoinnin järjestyksellä on suuri merkitys, sillä finalisointi tulisi tehdä topologisessa järjestyksessä [22, 21]. Tällä tarkoitetaan, että jos oliosta A on viittaus olioon B, tulisi olio A finalisoida ennen oliota B, koska on mahdollista, että olion A finalisointi käyttää oliota B hyväkseen.

Toinen ongelma finalisaation kanssa on se, että sen avulla voidaan herättää kuolleita olioita henkiin [21]. Tämä on mahdollista siksi, että itse finalisoitava olio saattaa sisältää viittauksen elossa olevaan olioon. Mikäli finalisaatiotoimenpiteet käyttävät elossa olevan olion palveluita ja antavat käyttämänsä metodikutsun parametriksi osoittimen johonkin kuolleeseen olioon, pahimmassa tapauksessa finalisoitavaan olioon itseensä, on katastrofi valmis. Tämä elossa oleva olio voi tallentaa viitteen kuolleeseen olioon attribuuttiinsa, jolloin finalisoitava olio on jälleen ohjelman käytettävissä. Sinänsä toimenpide on sallittu, mutta se vaikeuttaa finalisointia entisestään. Esiin nousee nyt kysymys: “mitä tehdään, kun olio kuolee uudelleen?”

Finalisaation ennustamattomuudesta johtuen voidaan sen käyttäminen reaaliaikajärjestelmissä unohtaa.

4.6 Hopealuoti?

Muistinsiivous, hyödyllisyydestään huolimatta, ei ole ratkaisu kaikkiin muistinhallinnan ongelmiin [22, sivu 11]. Kyseessä ei todellakaan ole hopealuoti, jota käyttämällä kaikki muistinhallinnan ongelmat ratkeavat. On totta, että siivousta käyttämällä vältytään monilta ongelmilta, joita manuaalinen muistinhallinta tuottaa, mutta siivous tuottaa myös omia ongelmiaan. Osa näistä ongelmista esitettiin yllä ja osa ongelmista liittyy käytännön toteutukseen. Esimerkiksi siivointen debuggaus voi olla erittäin ongelmallista, ja virheellisesti toimiva siivoin saattaa aiheuttaa pahempia ongelmia kuin mitä manuaalisesti voidaan saada aikaan.

Interaktiivisissa ohjelmissa muistinsiivouksen käyttäminen alkaa olla jo yleistä, kiitos tästä kuuluu mm. Javalle ja yhä enenevässä määrin C#:lle, funktionaalisia ohjelmointikieliä unohtamatta. Muistinsiivouksen käyttäminen reaaliaikajärjestelmissä ei kuitenkaan ole yleistä, sillä yleiskäyttöistä siivointa, joka pystyisi takaamaan reaaliaikavasteet, ei ole tiedossa.

Seuraavassa luvussa käsitellään tekniikkaa, jolla muistinsiivouksen vasteaikaa saadaan parannettua siten, että muistinsiivouksen käyttäminen on mahdollista myös reaaliaikajärjestelmissä.

5 Kohti reaaliaikaista muistinsiivousta

Tässä luvussa tutustutaan muistinsiivouksen reaaliaikaistamisen mahdollistaviin menetelmiin, näiden menetelmien käyttämisestä aiheutuviin ongelmiin sekä tapoihin, joilla nämä ongelmat voidaan ratkaista. Lisäksi esitellään muutama muistinsiivoin, joiden avulla muistia on mahdollista siivota reaaliaikaisesti.

5.1 Vähittäinen siivous

Perinteisissä muistinsiivousmenetelmissä siivous käynnistetään, kun vapaan muistin määrä alittaa jonkin tietyn rajan. Tällöin muuntimen toiminta keskeytetään ja muisti siivotaan. Siivouksen valmistuttua muuntimen toimintaa jälleen jatketaan. Reaaliaikajärjestelmien kannalta tällainen toiminta ei kuitenkaan ole sallittua, sillä siivoukseen kuluva aika on todennäköisesti liian suuri, eikä vasteaikavaatimuksia saada täytetyksi. Perinteisiin menetelmiin onkin kehitetty parannuksia, joilla siivouksesta aiheutuvaa katkosta saadaan lyhennetyksi. Eräs tällaisista menetelmistä on luvussa 4.4.2 esitetty ikäperustainen siivous. Ikäpeurustaisella siivouksella saadaan parannettua muistinsiivouksen keskimääräistä vasteaikaa, mutta pahimmassa tapauksessa muistinsiivoukseen kuluva aika voi olla erittäin pitkä ja näin ollen ikäperustainen siivous ei ole riittävän tehokasta, jotta sitä voitaisiin käyttää reaaliaikajärjestelmissä.

Kuten Wilson [58] toteaa, reaaliaikajärjestelmissä muistinsiivousta ei voida suorittaa yhtenä atomisena operaationa, vaan siivoimen toiminta on jaettava pieniin inkrementteihin, joita suoritetaan silloin tällöin. Yleensä tämä tarkoittaa, että muistia siivotaan hieman aina, kun muutin varaa muistia. Tällaista tekniikkaa kutsutaan *vähittäiseksi muistinsiivoukseksi (incremental garbage collection)* ja muistinvarauksen yhteydessä suoritettua siivousta kutsutaan *inkrementiksi*.

Viitelaskuri on valmiiksi vähittäinen, sillä sen vaatimia toimia suoritetaan lomitetusti aina, kun muutin muuttaa osoittimien arvoja [46]. Viitelaskuritekniikan tunnetuista ongelmista johtuen se ei kuitenkaan ole soveltuva yleiskäyttöiseksi siivoimeksi reaaliaikajärjestelmiin; siis on käytettävä jäljittäviä muistinsiivousmeneltemiä. Jäljittävissä siivoimissa vähittäinen siivous tarkoittaa sitä, että siivoin suorittaa inkrementtinsä aikana muutaman toimenpiteen. Merkkaa ja lakaise -tekniikalla toimiva siivoin

merkkaa muutaman olion ja kopioiva siivoin kopioi. Inkrementit voivat olla atomisia, mikäli ne ovat riittävän lyhyitä. Käytettäessä jäljittäviä siivoimia vähittäisesti ongelmaksi muodostuu se, että muutin voi muuttaa osoittimia sellaisesta oliosta, jonka siivoin on jo käynyt läpi. Siis muutin voi muistinsiivoussyklin aikana muuttaa juurijoukon transitiivista sulkeumaa.

Raportissaan [58] Wilson toteaa, että ongelmaa kannattaa tarkastella koherenssiongelmana, joissa useat prosessit jakavat muuttuvan tiedon, mutta tarvitsevat yhtenäisen näkemyksen asiasta. Vähittäinen merkkaa ja lakaise -tekniikka vastaa ongelmana yhden kirjoittajan ja usean lukijan ongelmaa (single writer, multiple readers). Siinä siivoimen pitää reagoida elävissä tietueissa tapahtuviin muutoksiin, mutta vain muutin voi muuttaa tietueita. Vähittäin kopioivat siivoimet vastaavat vaikeampaa usean kirjoittajan ja usean lukijan (multiple reader, multiple writers) ongelmaa. Kopioivissa siivoimissa sekä muutin että siivoin muuttavat juurijoukon transitiivistä sulkeumaa ja molemmat osapuolista on suojattava muutoksilta.

Muistinsiivoimen näkemä osoittimien muodostama graafi ei välttämättä ole täydellinen kopio siitä graafista, jonka muutin näkee, kunhan tietyt ehdot säilyvät. Luonnollisesti siivoimen on nähtävä kaikki elossa olevat oliot, jottei se vapauta oliota virheellisesti. Mikäli siivoin hetkellisesti näkee sellaisia olioita, jotka ovat juurijoukon transitiivisen sulkeuman ulkopuolella, ja merkkaa kyseiset oliot eläviksi, ei suuria ongelmia aiheudu. Tällöin järjestelmään saadaan leijuvaa roskaa. Leijuvasta roskasta ja sen aiheuttamista ongelmista on kerrottu luvussa 5.1.3.

Koherenssiongelman ymmärtämiseksi muistinsiivousta kannattaa tarkastella kolmivärimerkkausabstraktion avulla, johon tutustutaan seuraavaksi.

5.1.1 Kolmivärimerkkausabstraktio

Vähittäistä siivousta kuvattaessa käytetään usein Djikstran *kolmivärimerkkausabstraktiota (tricolour marking)* [13, 22, 58]. Vaikka abstraktio alunperin kehitettiin kuvaamaan vähittäistä siivousta, voidaan sillä kuvata myös tavallisia jäljittäviä muistinsiivoimia. Hyvä esimerkki tästä on Jonesin ja Linsin kirjassa sivulla 119 alkavassa esimerkissä, jossa he kuvaavat Cheneyn kopioivan siivoimen toiminnan kolmivärimerkkauksen avulla.

Nimensä mukaisesti kolmivärimerkkauksessa tietueet "värjätään" kolmella värillä siten, että solmu on väriltään:

musta jos kyseinen tietue ja kaikki sen jälkeläiset on merkattu löydetyiksi. Siivoimen

ei enää tarvitse palata mustaksi värjättyyn olioon.

harmaa jos siivoimen on vielä palattava olioon. Harmaa tietue voi olla joko sellainen, että itse olio on merkattu löydetyksi, mutta sen kaikkia jälkeläisiä ei, tai harmaan olion osoittimia on muutettu ja siivoimen on tutkittava sen jälkeläiset uudelleen.

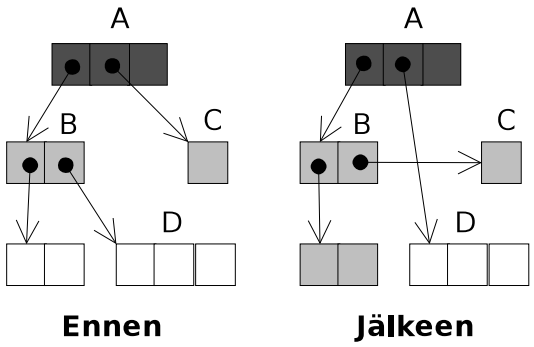
valkoinen jos siivoin ei ole kyseisessä oliossa vieraillut. Muistinsiivoussyklin lopussa valkoiset oliot ovat roskia.

Muistinsiivous aloitetaan siten, että juurijoukko väritetään harmaaksi. Tämän jälkeen aletaan käydä harmaita olioita läpi ja aina, kun olion kaikki jälkeläiset on merkattu joko harmaaksi tai mustaksi, merkataan itse olio mustaksi. Muistinsiivoussykli on saatu päätökseen, kun harmaita oliota ei enää ole. Tällöin kaikki valkoiset oliot ovat roskaa ja ne voidaan vapauttaa.

Siivoussyklin aikana muutin saattaa muuttaa osoittimien muodostamaa graafia. Tämä on ongelmallista, jos muutin tekee muutoksensa siivoimelta piilossa. *Kolmiväri-invariantti (tricolour invariant)* sanoo, että mustaksi väritetystä oliosta saa olla osoitin vain harmaaseen tai mustaan olioon [42]. Toisin sanoen mustan ja valkoisen alueen eroittaa aina harmaa alue.

Tilanteessa, jossa muutin muuttaa osoittimia, on tärkeää tarkastella oliota, jonka osoittimia muutetaan [42]. Ongelmia ei synny, jos muutin muuttaa valkoisen tai harmaan olion osoittimia. Valkoiseen olioon tehtyjä muutoksia ei huomioida, sillä siivoin ei vielä ole ehtinyt kyseistä oliota tarkastelemaan, ja kyseinen olio joko merkataan myöhemmin tai se muuttuu roskaksi ennen kuin siivoin sen löytää. Harmaan olion tapauksessa ei myöskään ole hätää, sillä siivoimen on joka tapauksessa myöhemmin palattava kyseiseen olioon.

Muuntimen muuntaessa jo mustaksi väritetyn olion osoittimia on mahdollista, että siivoimen näkemä graafi joutuu sellaiseen tilaan, että siivoin virheellisesti vapauttaa elossa olevaa dataa [42]. Muutettaessa mustan olion osoittimia, on kiinnitettävä huomio muutettavan osoittimen kohdeolioon. Kuten aikaisemmin mainittiin, kolmiväri-invariantin mukaan mustista oliosta saa olla osoitin vain harmaisiin tai mustiin oliohin. Virhetilanne syntyy siis silloin, kun muutin muuttaa mustaksi värjätyn olion osoittimen osoittamaan valkoiseen olioon. Muuttimen tehdessä jotain, joka rikkoisi kolmiväri-invariantin, on sen korjattava asia. Muuntimen ja siivoimen toiminnan on oltava synkroitua. Synkronointi hoidetaan *muureilla (barriers)*, josta kerrotaan enemmän luvussa 5.2. Toisaalta tilanne, jossa muutin vaihtaa mustan olion osoittimen osoittamaan harmaaseen olioon on täysin sallittua, eikä tilanteeseen reagoida millään tavalla.



Kuva 5.1: Kolmiväri-invariantin rikkoutuminen [58]

Tarkastellaan muuntimen ja siivoimen välisen koordinoinnin tarvetta esimerkin avulla. Esimerkki on peräisin Wilsonilta [58]. Kuvan 5.1 vasemmassa puoliskossa on tilanne, joka on voimassa siivoimen suoritettua inkrementtinsä. Tässä tilanteessa olio A on käsitelty kokonaan, eli sen kaikki jälkeläiset on löydetty. Nyt oletetaan, että muutin muuttaa tietueita siten, että osoitin oliosta A olioon C vaihdetaan osoittimen oliosta B olioon D kanssa. Tämä tilanne on esitettynä kuvan oikean puoleisessa osassa. Mikäli muuntimen ja siivoimen välillä ei ole minkäänlaista koordinointia, jatkuisi siivoimen toiminta siten, että se värjäisi olion B mustaksi sekä päätyisi olioon C uudelleen B:n kautta. Koska olio A on jo värjätty mustaksi, ei siihen enää palata, eikä siivoin täten saavuta oliota D, jolloin tämä olio D vapautettaisiin virheellisesti.

5.1.2 Riittävän siivouksen takaaminen

Vähittäistä siivousta käytettäessä on pidettävä huoli siitä, että siivoin siivoaa riittävästi, jotta vapaa muisti ei lopu kesken [22, sivu 184]. Yleinen tapa riittävän siivouksen varmistamiseksi on se, että käytetään nk. *allokaatiokelloa (allocation clock)* [58]. Tällöin siivoin tekee joka allokaatiolla ennalta määrätyn verran työn aina, kun muutin varaa muistia. Työn määrä on suhteutettava varattavan alueen kokoon. Tällä tavalla voidaan varmistaa se, että vaikka muutin varaisi muistia kuinka nopeasti tahansa siivotaan silti riittävästi.

Jones ja Lins [22] esittävät laskelman, jolla riittävä siivous saadaan taattua. Oletetaan, että siivoin jäljittää k tietuetta joka kerta, kun muutin varaa muistia. Lisäksi keossa on muistinsiivoussyklin alussa L tavua elossa olevia tietueita ja oletetaan, että kaikki siivousyklin aikana varatut uudet tietueet väritetään mustaksi, eikä niitä käydä läpi kyseisen syklin aikana. Keon maksimikokoa merkitään M:llä, jolloin pätee $L < M$.

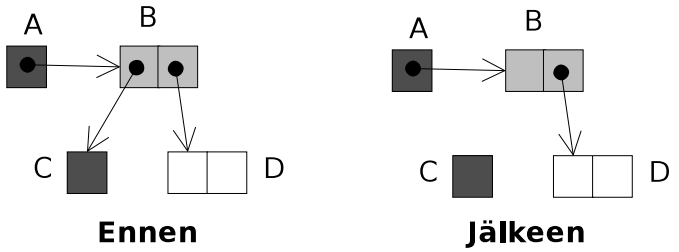
Nyt kaikki elävä data on merkattu L/k allokaation jälkeen, ja merkkausvaiheen jälkeen keossa on korkeintaan $L + L/k$ elossa olevaa tietuetta. Riittää, että siivoin jäljittää $L(1 + 1/k)$ tietuetta jokaisella kerralla, kun muutin varaa muistia. Tämä arvo tuplataan, jos käytetään kopioivaa siivointa, joka jakaa keon kahteen puoliavaruuteen. Lisäksi k:n arvon on oltava suurempi kuin $L(M - L)$, jotta siivoimen toiminta edistyy riittävän nopeasti. On huomioitava, että edellä esitetyllä tavalla toimivassa vähittäisessä siivoimessa uuden tietueen varaamiseen kuluva aika kasvaa lineaarisesti k:n suurentuessa, joten k:n arvo ei saa olla myöskään liian suuri. Mitä enemmän työtä kullakin allokaatiolla tehdään, sitä pidemmän aikaa allokointi kestää.

5.1.3 Konservatisimi muistinsiivouksessa

Kun muistinsiivousta ei suoriteta yhtenä atomisena operaationa on huomioitava muutokset, joita muunnin tekee siivoimelta piilossa. Aikaisemmin kolmivärimerkkauksen yhteydessä, luvussa 5.1.1, kerrottiin ongelmasta, joka voi syntyä, kun muutin näitä muutoksiaan tekee. Aivan kaikkia muuntimen tekemiä muutoksia ei kuitenkaan huomioda, sillä muutokset eivät välttämättä aiheuta siivoimen virheellistä toimintaa. Muistinsiivouksen yhteydessä *konservatismilla (conservatism)* tarkoitetaan siivoimen suhdetta muuntimen tekemien muutosten määrään [58].

Siivoimen näkemä osoittimien muodostama graafi onkin yleensä vain konservatiivinen aproksimaatio todellisesta juurijoukon transitiivisesta sulkeumasta [58]. Näillä eroilla ei ole merkitystä, jos ne eivät vaikuta siivoimen toimintaan. Siivoin voi hetkellisesti pitää kuolleita olioita elävinä, kunhan se ei luule eläviä olioita kuolleiksi. Vähittäisessä siivouksessa käy yleensä niin, että joukko oliota kuolee sen jälkeen, kun siivoin on merkinnyt ne eläviksi. Tällöin puhutaan *leijuvista roskista (floating garbage)*. Leijuvat roskat säilyvät hengissä meneillään olevasta muistinsiivoussyklistä, mutta niiden varaama tila vapautetaan seuraavan siivoussyklin aikana. Leijuvat roskat eivät ole pelkästään vähittäisen siivouksen aikaansaannoksia, sillä ikäperustaisissa siivoimissa toimitaan periaattessa samalla tavalla; koska vanhemmissa sukupolvissa ei siivousta suoriteta usein, kertyy niihin leijuvaa roskaa [22, sivu 186].

Tarkastellaan tilannetta, joka esiintyy kuvassa 5.2. Kuvassa vasemman puoleinen kaavio kuvaa tilannetta, johon merkkaa ja lakaise -tekniikalla toimiva muistinsiivoin inkrementtinsä lopuksi päätyy. Siivoin on värjännyt sekä tietueen A että tietueen C mustaksi ja tietueen B harmaaksi. Näiden merkkausten jälkeen suoritusvuoro palaa muuntimelle, joka muuntaa tietueita siten, että tietueesta B tietueeseen C ei enää ole yhteyttä. Tämä tilanne on kuvattuna oikean puoleisessa kaaviossa. Koska siivoimemme



Kuva 5.2: Leijuvat roskat

on konservatiivinen, ei se tätä muutosta huomioi. Muuntimen tekemien muutosten jälkeen tietue C on edelleen värjättynä mustaksi, vaikka se todellisuudessa onkin kuollut. Siivoin siis jättää tietueen B vapauttamatta tämän siivoussyklin aikana, mutta varmuudella tiedetään, että tietue B vapautetaan seuraavan muistinsiivoussyklin lopuksi.

Leijuvat roskat eivät ole vakavia järjestelmän toiminan kannalta, mikäli muistia vain on riittävä määrä. Järjestelmissä, joissa muistin määrä on vähäinen, saattaa leijuvien roskien esiintyminen pakottaa siivoamaan muistia useammin. Mitä konservatiivisempi siivoimen näkymä on, sitä enemmän leijuvaa roskaa esiintyy [22, sivu 186]. Mikäli siivoimen ja muuntimen näkymät halutaan pitää yhtenäisinä, pakottaa tämä siivoimen huomioimaan pienetkin muuntimen tekemät muutokset. Tämä puolestaan aiheuttaa sen, että muuntimen ja siivoimen toiminnat synkronoivien muurien on oltava kompleksisia, jolloin järjestelmän toiminta saattaa hidastua.

Konservatismiin vaikuttaa olennaisesti siivoimessa käytettävät muurit. Muureista sekä niiden vaikutuksesta konservatismiin kerrotaan tarkemmin seuraavaksi.

5.2 Muurit – siivoimen ja muuntimen synkronointi

Vähittäisen siivouksen yhteydessä, luvussa 5.1, kerrottiin ongelmasta, joka syntyy, kun siivousta ei suoriteta yhtenä atomisena operaationa. Muuntimen ja siivoimen toiminta on tällöin synkronoitava. Kolmiväri-invariantin mukaan mustasta oliosta ei saa olla viitteitä valkoisiin olioihin (kts luku 5.1.1). Kolmiväri-invariantin säilyttämiseksi käytetään *muureja (barriers)*, joita on kahta perustyyppiä: *luku- (read)* ja *kirjoitusmuuri (write)* [58]. Nimet johtavat juonensa siitä, että lukumuureja käytetään osoittimia luettaessa ja kirjoitusmuureja osoittimien arvoja muutettaessa.

Muurit ovat synkronisointioperaatiota, joiden avulla muunnin aktivoi siivoimen aina ennen kuin se suorittaa osoittimien luku- tai kirjoitusoperaation [58]. Muurit ovat

muutaman käskyn mittaisia ja kirjoitusmuurissa ne lisätään osoittimien arvon muuttamisen yhteyteen ja lukumuurissa puolestaan jokaiseen lukuoperaatioon.

Toimiakseen kaikki vähittäin siivoavat siivoimet vaativat joko luku- tai kirjoitusmuurin, ja joissain tapauksissa on välttämätöntä käyttää molemman tyyppisiä muureja [58]. Merkkaa ja lakaise -tekniikalla toimivissa siivoimissa siivoin on suojattava muuntimen tekemiltä muutoksilta [22, sivu 188]. Tällöin riittää, että muuntimessa käytetään vain kirjoitusmuuria. Käytettäessä siivoimia, jotka siirtävät keossa olevia tietueita, on myös muunnin suojattava muutoksilta. Tällöin tarvitaan joko lukumuuria tai sekä luku- että kirjoitusmuuria, riippuen käytettävän lukumuurin kompleksisuudesta. Kirjoitusmuurilla suojataan siivoin muuntimen tekemiltä muutoksilta ja lukumuurilla estetään muutinta näkemästä kopioitujen olioiden vanhoja versioita, jotka sijaitsevat lähdeavaruudessa. Myös viitelaskuri toteutetaan kirjoitusmuuria käyttämällä; jokainen osoittimen muutos suorittaa kirjoitusmuurin, joka päivittää laskurin arvon oikeaksi.

Muurien toteutus voidaan hoitaa usealla eri tavalla, muurien kompleksisuudesta riippuen [58]. Mikäli muurin ohjelmakoodi on tarpeeksi lyhyt, voidaan muuri toteuttaa sisäisohjelmana, joka on paras vaihtoehto nopeuden kannalta, mutta samalla tämä kasvattaa ohjelman kokoa. Vaihtoehtoisesti muuri voidaan toteuttaa tavallisena aliohjelmakutsuna. Mikäli käännösympäristö tukee muistinsiivousta, on mahdollista, että kääntäjä sijoittaa muurit paikalleen. Muussa tapauksessa voi ohjelmoija joutua hoitamaan työn käsin.

Vähittäisten siivoimien aiheuttama ohjelman suorituksen hidastuminen ja muistinkäytön kasvaminen on riippuvainen käytetystä muurista, ja etenkin muurin konservatismista, eli siitä kuinka muuri vaikuttaa siivoimen näkemään osoitingraafin täsmällisyyteen [22, sivu 188]. Mitä enemmän työtä muurissa tehdään, sitä hitaammaksi ohjelman suoritus käy, mutta samalla leijuvan roskan määrä pienenee. Lisäksi muurin implementaatiolla on suuri vaikutus; mikäli muuria suoritetaan ehdollisesti, saadaan todennäköisesti suoritukseen kuluvaa aikaa keskiarvoisesti pienennettyä. Tällä ei ole merkitystä reaaliaikajärjestelmissä, sillä reaaliaikajärjestelmiä toteutettaessa on oletettava, että jokainen muurin aktivointi kuluttaa maksimiajan.

Lukumuurien aiheuttamaa hidastusta pidetään liian suurena, jotta niiden käyttäminen olisi järkevää [22, sivu 188]. Lisäksi Zorn [61] sai mittauksissaan tulokseksi, että osoittimien lukuoperaatiot saattavat viedä 13 - 15 prosenttia ohjelmakoodista. Tällöin jo seitsemän käskyn mittainen sisäisohjelmana toteutettu lukumuuri tuplaa ohjelman koon. Ohjelman paisumisen lisäksi tämä varmasti aiheuttaa myös suorituksen hidastumisen. Lukumuurien hidastavan vaikutuksen vuoksi niitä ei juurikaan käytetä muualla

kuin tietoa liikuttavissa siivoimissa, joissa niiden käyttäminen on pakollista järjestelmän oikeellisuuden takaamiseksi [22, sivu 188]. Lukumuurien käyttämistä puolustaa Johnson [24], joka totetaa, että laitteistolla toteutetun lukumuurin käyttäminen ei vaikuta huomattavasti muistinsiivoimen vasteaikaan verrattuna kirjoitusmuuriin. Lisäksi lukumuuria käytettäessä saadaan parannettua järjestelmän muistikäyttäytymistä, jolloin virtuaalimuistin vasteajat saattavat parantua.

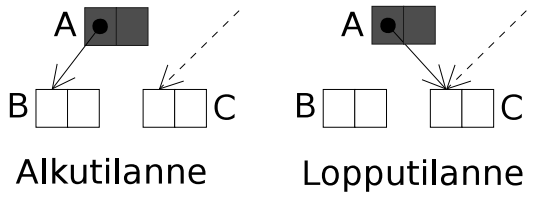
Kirjallisuudessa muureihin viitataan yleensä joko algoritmeina, kuten Bakerin algoritmi, tai muureina, kuten Bakerin lukumuuri. Tämä johtuu siitä, että muuritekniikat on alunperin kehitetty tiettyä algorimia varten, josta ne on myöhemmin irroitettu yleisempään käyttöön. Seuraavaksi tarkastellaan muureja hieman tarkemmin kirjoitusmuureista aloittaen.

5.3 Kirjoitusmuurit

Kirjoitusmuurin tehtävä on estää siivoimen sekoaminen muutimen tehdessä muutoksia osoitingraafiin [22, sivu 189]. Muutin suorittaa kirjoitusmuurin aina osoittimien arvoja muutettaessaan. Muurilla estetään muutinta kirjoittamasta osoitinta mustasta oliosta valkoiseen.

Wilson [58] jakaa kirjoitusmuurit kahteen ryhmään sen perusteella, kuinka muurit toimivat: yrittävätkö ne säilyttää alkuperäiset viitteet, vai huomioivatko ne muutokset osoitingraafissa. Nämä ryhmät ovat: *vedosmuurit (snapshot-at-beginning)* ja *vähittäisen päivityksen muurit (incremental update)*.

Vedosmuureissa estetään olioiden kuoleminen siivoussyklin aikana [58]. Muistinsiivouksen aikana muuntimen vaihtaessa osoittimen arvoa alkuperäinen osoittimen arvo talletetaan ja tarkastetaan myöhemmin. Vedosmuurit ovat erittäin konservatiivisia ja niitä käytettäessä järjestelmään syntyy paljon leijuvaa roskaa [22, sivu 190]. Vedosmuurien toiminta voidaan ajatella niin, että siivoussyklin alussa vedosmuurit tekevät osoitingraafista siivoimelle kopion, eikä tätä kopioita voida enää muistinsiivoussyklin aikana muuttaa. Vedosmuurien käyttämisessä tulee ottaa huomioon myös siivouksen aikana luotujen tietueiden väri. Koska halutaan, että yksikään olio ei kuole siivoussyklin aikana väritetään uudet oliot mustiksi. Näin ollen kaikki muistinsiivoussyklin alussa elossa olleet oliot sekä siivoussyklin aikan luodut oliot vapautetaan vasta seuraavan siivoussyklin aikana, vaikka ne kuolisivatkin käynnissä olevan syklin kuluessa. Vedosmuureja käytettäessä siivoimen näkymä osoitingraafista on yhdistelmä uusista olioista ja osoitingraafista, joka siivouksen alkaessa oli [58].



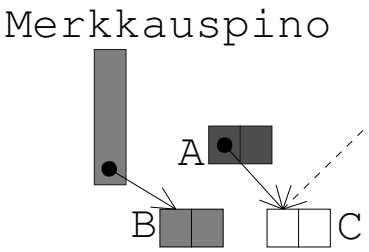
Kuva 5.3: Muuntimen toiminta kirjoitusmuuriesimerkeissä [22, sivu 189]

Toisin kuin vedosmuureissa, vähittäisen päivityksen muureissa olioiden annetaan kuolla siivoussyklin aikana [58]. Sen sijaan, että alkuperäiset osoittimet säilytetään, vähittäisen päivityksen muureissa reagoidaan muuntimen tekemiin muutoksiin. Muuntimen kirjoittaessa mustaan olioon osoittimen, joka viittaa valkoiseen olioon värjätään jompikumpi olioista harmaaksi. Vedosmuureihin verrattaessa vähittäisen päivityksen muurit ovat vähemmän konservatiivisia; vähittäin päivittäviä muureja käytettäessä leijuvan roskan määrä on pienempi kuin vedosmuureissa [22, sivu 191].

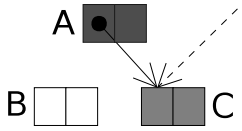
Kuvassa 5.3 on aloitustilanne, jonka valossa kaikki alla esitettävät kirjoitusmuuri-implemetaatiot esitetään. Muuntimen oletetaan muuttavan oliossa A olevan osoittimen olioon B. Alkutilanne on havainnolistettuna kuvan vasemman puoleisessa osassa, ja lopputilanne muuntimen muutosten jälkeen kuvan oikean puoleisessa osassa. Kirjoitusmuuri aktivoitaisiin tämän osoittimen muutoksen aikana. Nämä esimerkkikuvat ovat peräisin Jonesilta ja Linsiltä [22].

5.3.1 Yuasan kirjoitusmuuri

Ehkäpä parhaiten tunnetu ja yksinkertaisin vedosmuureista on Yuasan kirjoitusmuuri [58]. Yuasan muurissa toimitaan siten, että muuntimen muuttaessa osoittimen arvoa, alkuperäinen osoitin talletetaan merkkauspinoon, josta se myöhemmin jäljitetään. Tällä tavalla taataan, että yksikään olio ei kuole siivouksen aikana. Yuasan muurissa uudet oliot värjätään mustiksi. Tähän on kuitenkin pieni poikkeus, sillä siivoimen lakaisuvaiheen aikana uusia olioita ei värjätä, mikäli ne sijaitsevat keossa sen kohdan alapuolella, johon asti ollaan jo lakaistu [22, sivu 191].



Kuva 5.4: Yuasan kirjoitusmuuri [22, sivu 189]

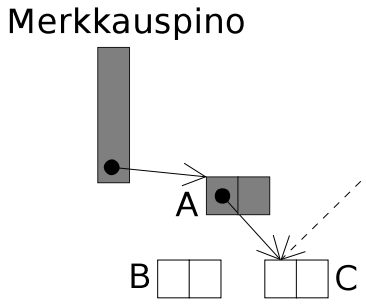


Kuva 5.5: Dijkstran ym. kirjoitusmuuri [22, sivu 192]

Kuvassa 5.4 on esitetty Yuasan kirjoitusmuurin toiminta. Muuntimen toimiessa kuvassa 5.3 esitetyllä tavalla Yuasan muurissa toimitaan siten, että alkuperäinen osoitin oliosta A lisätään merkkauspinoon ja olio B värjätään harmaaksi. Myöhemmin merkkauspinon sisältämät osoittimet jäljitetään, ja olio B sekä kaikki sen jälkeläiset merkataan ja täten säilytetään hengissä.

5.3.2 Dijkstran kirjoitusmuuri

Dijkstran ym. [13] kirjoitusmuuri on konservatiivisin vähittäin päivittävistä muureista [22, sivu 192]. Muurissa toimitaan siten, että valkoiset oliot, joihin luodaan osoitin mustista oliosta, värjätään harmaaksi. Dijkstran muurissa uudet oliot luodaan valkoisina ja ne saattavat kuolla ennen siivouksen lopettamista. Wilson [58] toteaa, että Dijkstran muurin toiminta on helppo todistaa oikeaksi, sillä sekä muutin että siivoin kuljettavat harmaata rintamaa eteen päin.



Kuva 5.6: Steelen kirjoitusmuuri [22, sivu 193]

Dijkstran muurin toiminta on esitettynä kuvassa 5.5. Oliosta A päivitettävä osoitin olioon C aiheuttaa sen, että olio C värjätään harmaaksi.

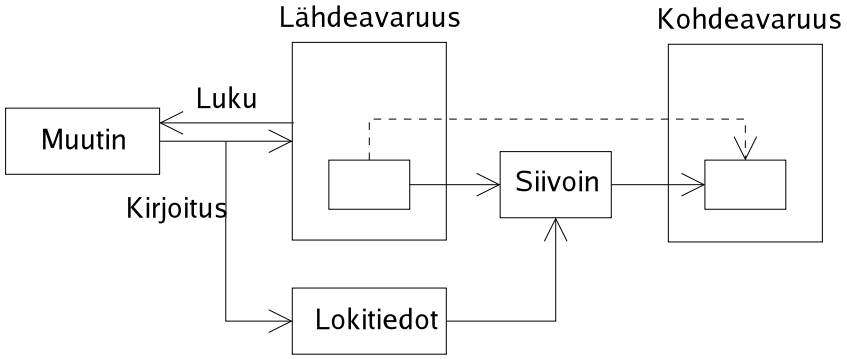
5.3.3 Steelen kirjoitusmuuri

Steelen [52] vähittäin päivittävässä kirjoitusmuurissa, toisin kuin Dijkstran muurissa, harmaata rintamaa siirretään takaisinpäin. Tämä tehdään siten, että muuntimen kirjoittaessa osoittimen arvon yli, merkataan olio, jossa tämä osoitin sijaitsi, harmaaksi ja olio painetaan merkkauspinoon. Muuri saattaa vaatia siivointa tarkastamaan uudelleen alkuperäisen olion, jossa muutettu osoitin sijaitsee. Harmaan rintaman takaisinpäin siirtämisen ansiosta Steelen muuria käyttävissä järjestelmässä ilmenee vähemmän leijuvaa roskaa Dijkstran muuriin verrattuna [22, sivu193].

Jatkaaksemme esimerkkiä, kuvassa 5.6 on havainnollistettuna Steelen kirjoitusmuurin toiminta. Sen jälkeen, kun oliosta A lähtevää osoitinta on muutettu, värjätään olio A harmaaksi ja se painetaan merkkauspinoon, joka käydään läpi myöhemmin.

5.3.4 Nettlesin ja O'Toolen kirjoitusmuuri

Nettlesin ja O'Toolen replikoiva siivoin käyttää heidän kehittämäänsä kirjoitusmuuria, jonka ansiosta vähittäin kopioivissa siivoimissa ei tarvitse käyttää lukumuuria [39]. Heidän tekniikassaan siivoimen tehtävänä on tehdä täydelliset replikat lähdeavaruudessa sijaitsevista tietueista kohdeavaruuteen. Siivoin kopioi tietueet sellaisinaan ja muuntimen annetaan tämän jälkeenkin käyttää lähdeavaruudessa sijaitsevia kopioita. Muutin siis näkee pelkästään lähdeavaruudessa sijaitsevat oliot. On mahdollista, että muutin muuttaa lähdeavaruuden tietueita sen jälkeen, kun tietue on replikoitu kohdeavaruu-



Kuva 5.7: Netlesin ja O'Toolen kirjoitusmuuri [39]

teen. Muuntimen annetaan tehdä nämä muutokset, mutta samalla nämä muutokset tallennetaan lokikirjaan.

Ennen muistinsiivoussyklin loppumista siivoimen on käytävä läpi kaikki tietueet, jotka muutoslokikirjassa ovat. Siivoin käsittelee muutoslokissa olevan tietueen tarkistamalla, että kaikki tietueen jälkeläiset on jo kopioituna kohdeavaruuteen, ja mikäli näin ei ole, kopioidaan puuttuvat tietueet sinne. Tämän jälkeen siivoin hylkää kyseisen muutoslokin tiedot ja siirtyy käsittelemään seuraavaa lokitietoa. Tätä jatketaan kunnes muutosloki on tyhjä, jolloin siivoussykli lopetetaan. Nettlesin ja O'Toolen replikoivan siivoimen rakennetta on kuvattu kuvassa 5.7.

Nettles ja O'Toole [39] sanovat, että heidän replikoiva siivoimensa voidaan teoriassa keskeyttää milloin vain, joten siivointa on mahdollista käyttää reaaliaikajärjestelmissä, mutta heidän alkuperäinen implementaationsa ei kuitenkaan pystynyt takaamaan rajattuja suoritusaikoja. Negatiiviseksi asiaksi heidän muuristaan voidaan mainita se, että muutoslokin ylläpitäminen vaatii ylimääräistä muistia, sekä kirjoitusmuurin tehokkuus riippuu siitä kuinka tarkasti muutokset lokiin tallennetaan. Mikäli tallennetaan vain tietue, jota muutettiin, on lokitiedostoon tallennettava vain osoitin tähän tietueeseen. Toisaalta tällöin myös siivoimen on tehtävä enemmän töitä muutoslokia läpikäydessään. Toinen ääripää on kirjata lokiin kaikki mahdollinen tieto muutoksista. Tällöin kirjoitusmuurin suoritus on hitaampaa, mutta siivoimen toimintaa voidaan tehostaa lokia läpikäytäessä.

5.4 Lukumuurit

Lukumuureja käytetään tietoa liikuttavien siivoimien kanssa. Tällöin estetään muutinta näkemästä valkoisia olioita [22, sivu 187]. Kopioivissa siivoimissa tämä tarkoittaa lähdeavaruudessa sijaitsevia olioita, jotka on jo kopioitu kohdeavaruuteen.

Myös merkkaa ja lakaise -tekniikalla toimivia vähittäisiä siivoimia voidaan toteuttaa käyttämällä lukumuureja, mutta tämä on turhaa, sillä merkkaa ja lakaise -tekniikkaan perustuvat siivoimiet voidaan toteuttaa käyttämällä pelkästään kirjoitusmuureja, ja kuten aikaisemmin todettiin, kirjoitusmuurit ovat lukumuureja tehokkaampia.

5.4.1 Bakerin lukumuuri

Bakerin vähittäin kopioiva siivoin käyttää lukumuuria, joka estää muutinta näkemästä valkoisia, lähdeavaruudessa sijaitsevia olioita [4]. Muuntimen yrittäessä osoittaa lähdeavaruudessa sijaitsevaan olioon, lukumuuri laukaistaan ja kyseinen olio kopioidaan välittömästi kohdeavaruuteen, ja muuntimelle palautetaan osoite kopioituun olioon. Mikäli olio, johon muutin yrittää päästä käsiksi, on jo kopioitu, palautetaan osoitin kopioon suorittamatta mitään muita toimenpiteitä. Tällä tavoin muutin näkee vain kohdeavaruudessa sijaitsevat oliot, ja harmaiden olioden rintamaa kuljetetaan muuntimen edellä [22, sivu 203].

Baker kuvaa siivointansa reaaliaikaiseksi sillä perusteella, että kaikkien muistinsiivoimen operaatioiden suoritusajat ovat ylhäältä rajattuja [4]. Bakerin siivointa käyttävissä järjestelmissä on kuitenkin mahdollista, että muutin suorittaa useita perättäisiä osoittimien lukuoperaatioita, jotka vaativat olioiden kopioimisen kohdeavaruuteen. Koska olion kopiointiin kuluva aika on verrannollinen olion kokoon, saattaa tästä aiheutua odottamattomia pysähdyksiä.

Baker antaa ratkaisun myös tähän ongelmaan [4]. Bakerin ratkaisussa isoista olioista kohdeavaruuteen kopioidaan vain osa, ja kohdeavaruudessa sijaitsevaan olioon asetataan osoitin takaisin lähdeavaruuden alkuperäiseen kopioon. Nyt aina, kun olion tietoja muutetaan, tarkistetaan onko muutettava olion osa jo kopioituna. Mikäli muutettava osa oliosta on jo kohdeavaruudessa, muutetaan kohdeavaruuden oliota suoraan. Jos olion muutettavaa osaa ei ole vielä kopioitu, muutetaan lähdeavaruudessa sijaitsevaa olioita. Jones ja Lins toteavat, että Bakerin ratkaisussa ongelmaksi tulee se, että myös kaikki olioihin tapahtuvat kirjoitukset vaativat tarkistuksen, joka selvittää muutetaanko oliota lähde- vai kohdeavaruudessa, sekä olion tietojen lukemisessa on tarkistettava, että kummasta oliosta tieto luetaan [22, sivu 205].

Bakerin vähittäin kopioivassa siivoimessa uudet oliot värjätään mustiksi [22, sivu 203]. Tällä tavoin siivoimen ei näitä uusia olioita tarvitse tutkia, seurauksena leijuvan roskan määrä kasvaa.

Kuten jo aikaisemmin todettiin, lukumuurien käyttäminen on huomattavasti tehottomampaa kirjoitusmuureihin nähden. Bakerin muurissa muuntimelle aiheutuu viivästystä ei pelkästään olioiden merkkaamisesta, mutta myös olioiden kopioimisesta kohdeavaruuteen. Olion käsittelemiseen kuluva aika vaihtelee sen mukaan, että sijaitseeko olio lähde- vai kohdeavaruudessa. Jones ja Lins toteavatkin, että Bakerin siivoin epäonnistuu reaaliaikavasteiden toteuttamisessa [22, sivu205].

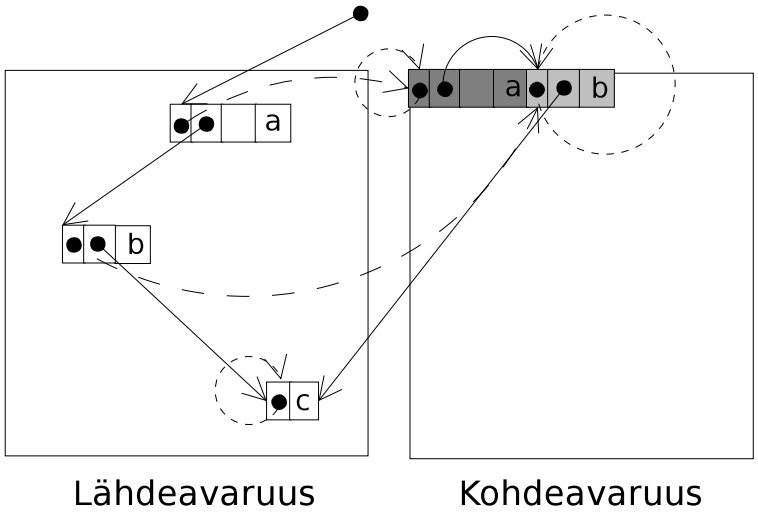
5.4.2 Brooksin lukumuuri

Brooksin [9] lukumuuri on parannettu versio Bakerin muurista. Sen sijaan, että oliohin viitattaessa ne kopiotaisiin kohdeavaruuteen, Brooksin muurissa käytetään niin sanottua *välitysosoitinta (indirection pointer)*, jota seurataan aina olioihin viitatessa. Välitysosoitin osoittaa olioon itseensä, mikäli oliota ei vielä ole kopioitu. Mikäli olio on kopioitu kohdeavaruuteen, on välitysosoittimen arvona tämän kopion osoite.

Kuvassa 5.8 on esitettynä Brooksin lukumuurin toiminta. Kuvan tilanteessa siivoin on ehtinyt kopioimaan oliot a ja b kohdeavaruuteen. Näiden olioiden lähdeavaruuden alkuperäisiin versioihin lisätään välitysosoitin kohdeavaruuteen. Tämä vastaa kopioivan siivoimen kopio-osoitinta. Kohdeavaruuteen jo kopioituihin olioihin, sekä lähdeavaruudessa vielä sijaitsevaan olioon c lisätään välitysosoittimet olioihin itseensä. Aina, kun muunnin käyttää olioita, se seuraa välitysosoitinta. Yrittäessään käyttää lähdeavaruudessa sijaitsevaa oliota a, käyttäisi muutin todellisuudessa olion a oikeaa kohdeavaruuteen kopioitua versiota. Toisaalta taas käyttäessään oliota c tai kohdeavaruuteen kopioitua oliota b, muutin päätyy välitysosoittimen avulla takaisin siihen olioon, jota se alunperin yritti käyttää.

5.5 Vähittäisiä muistinsiivoimia

Tässä luvussa esitetään muutamia toteutettuja muistinsiivoimia, joiden avulla reaaliaikavasteet voidaan toteuttaa.

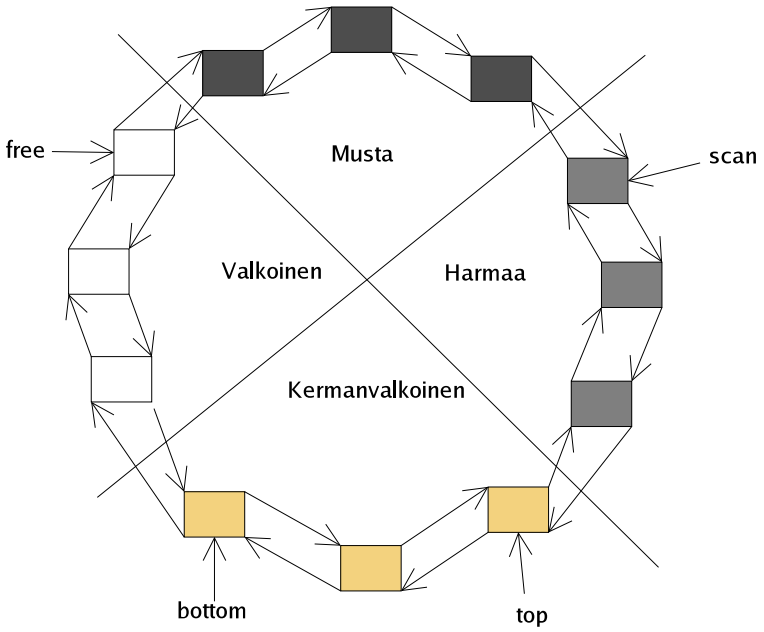


Kuva 5.8: Brooksin lukumuuri

5.5.1 Bakerin Polkumylly ja sen variaatiot

Bakerin *Polkumylly (treadmill)* [5] on vähittäinen reaaliaikainen merkkaa ja lakaise -tekniikalla toimiva siivoin. Polkumyllyssä muisti jaetaan palasiin, ja palaset tallennetaan sykliseen kahteen suuntaan linkitettyyn listaan. Listassa eri väriset tietueet pidetään yhtenäisinä jonoina. Polkumyllyssä oliot väritetään neljällä eri värillä: mustalla, harmaalla, valkoisella ja kermanvalkoisella. Kuvassa 5.9 on havainnollistettu tätä tilannetta. Eri väriset segmentit tunnistetaan neljän osoittimen avulla: bottom, top, free ja scan-osoittimilla. Osoittimet ja värisegmentit ovat järjestyksessä: bottom-osoitin, kermanvalkoinen alue, top-osoitin, harmaa alue, scan-osoitin, musta alue, free-osoitin, valkoinen alue ja bottom-osoitin. Muistin varaaminen tapahtuu siten, että free-osoitinta siirretään eteenpäin siten, että valkoisesta tietueesta tulee välittömästi musta. Muuntimelle palautetaan osoitin siihen lohkoon, joka varattaessa värjättiin mustaksi. Olioiden merkkaaminen tapahtuu tutkimalla scan-osoittimen osoittaman olion jälkeläiset. Sen jälkeen, kun scan-osoittimen alla olevan harmaan tietueen jälkeläiset on siirretty harmaan alueen loppuun, merkitään tietue mustaksi siirtämällä scan-osoitinta takaisinpäin ja näin merkataan tietue mustaksi. Harmaan tietueen jälkeläisiä tarkastastettaessa sellaiset tietueet, jotka ovat mustia tai harmaita jätetään paikalleen ja valkoiset oliot leikataan pois paikaltaan ja liitetään harmaaseen alueeseen, rintamahaussa top-osoittimen ja syvyyshaussa scan-osoittimen kohdalle.

Muistinsiivoussykli päättyy, kun harmaat tietueet loppuvat, siis silloin, kun scan ja



Kuva 5.9: Polkumylly

top-osoittimet kohtaavat [5]. Siinä vaiheessa, kun free ja bottom-osoittimet kohtaavat, ei jäljellä ole kuin kahta väriä, mustaa ja kermanvalkoista. Tällöin on suoritettava värien vahtaminen. Kermanvalkoisesta alueesta tehdään valkoinen ja mustasta alueesta kermanvalkoinen, lisäksi bottom ja top-osoittimet vaihdetaan keskenään. Tässä vaiheessa juurijoukon osoittamat tietueet merkitään harmaaksi siirtämällä ne kermanvalkoisesta osiosta harmaaseen osioon scan ja top-osoittimien väliin.

Polkumylly on isomorfinen Bakerin vähittäin kopioivan siivoimen kanssa; värien vaihtaminen vastaa lähde- ja kohdeavaruuksien vaihtamista [5]. Polkumyllyä käytettäessä saavutetaan joitain kopioivista siivoimista tuttuja etuja. Pääsiassa tämä tarkoittaa sitä, että tietueet vapautetaan implisiittisesti, eikä niitä tarvitse erikseen vapauttaa [34]. Perinteiseen merkkaa ja lakaise -tekniikkaan etuna on se, että erillistä merkkauspinoa ei tarvita. Toisaalta jokaisessa tietuessa on oltava kaksi ylimääräistä osoitinta, joilla Polkumyllyn rakennetta ylläpidetään [22, sivu 219].

Bakerin alkuperäinen Polkumylly käytti apunaan lukumuuria, joka merkkasi oliot aina, kun niitä yritettiin lukea [5]. Lukumuurin ajatuksena oli suojata muutinta siivoimen tekemiltä muutoksilta. Jones ja Lins kuitenkin toteavat, että koska Polkumylly ei tietoa liikuta, aiheuttaa lukumuurin käyttäminen vain turhaa hidastusta, ja tilalla voidaankin käyttää kirjoitusmuuria [22, sivu 220]. Jones ja Lins osaavat kertoa, et-

tä ainakin Wilson ja Johnstone ovat tehneet Polkumyllystä version, jossa käytetään kirjoitusmuuria.

Lim, Pardyak ja Bershad [34] suunnittelivat Polkumyllyyn perustuvan siivoimen, jossa muistin hyötykäyttöä tehostettaan alkuperäiseen Polkumyllyyn verrattuna. Heidän siivoimessaan olioita varatataan erikokoisissa alueissa, ja kutakin kokoluokkaa hallinnoidaan omalla polkumyllyllään. Lisäksi Limin, Pardyakin ja Bershadin siivoin käyttää kirjoitusmuuria muuntimen ja siivoimen yhteistyön takaamiseen.

Yhdistämällä polkumylly ja muistin varaaminen kiinteän kokoisissa alueissa (luku 3.2.1) voitaisiin algoritmin toimintaa tehostaa entisestään, sillä tällöin kaikki varatut muistialueet voidaan hallinnoida yhdellä ainoalla polkumyllyllä. Polkumyllyn operaatiot, joilla tietueiden paikkaa linkitetyssä listassa siirretään voidaan helposti toteuttaa siten, että kunkin operaation suorituksella on yläraja [22, sivu 218]. Tämän ansiosta algoritmilla voidaan todellakin mahdollistaa reaaliaikavasteet.

5.5.2 Reaaliaikainen viitelaskuri

Ritzau esittää viitelaskurin, joka kykenee tyydyttämään reaaliaikavasteet [46]. Ritzaun viitelaskurissa olioiden rekursiivinen vapauttaminen estetään käyttämällä *piilevää viitelaskuria (deferred reference counting)*. Piilevässä viitelaskuritekniikassa olioita ei vapauteta suoraan, vaan tietueet, joiden laskurin arvo putoaa nollaksi lisätään *vapautuslistaan (to-be-freed list)*. Varattaessa uutta muistia, vapautuslistasta otetaan ensimmäinen alkio, ja vasta tässä vaiheessa tietueessa olevat viitteet nollataan. Mikäli jokin nollattavien viitteiden osoittaman olion laskuri saa arvon nolla, lisätään tämä olio vapautuslistan alkuun.

Ritzau estää muistin ulkoisen pirstoutumisen käyttämällä muistinvarauksessa kiinteän kokoisia alueita, joista isommat tietueet koostetaan aivan kuten luvussa 3.2.1 kerrottiin. Kiinteän kokoisten alueiden käyttämisellä on myös toinen myönteinen vaikutus, sillä Ritzaun tekniikassa kiinteän kokoisten alueiden ansiosta voidaan määrätä yläraja, joka kuluu olion varaamiseen, siis lohkossa sijaitsevien viitteiden nollaamiseen ja lohkojen poistamiseen vapautuslistasta. Kiinteän kokoisia alueita käytettäessä on jokaisen alueen viitteiden lukumäärällä yläraja, ja täten myös niiden nollaamiselle saadaan laskettua yläraja [46]. Näin ollen uuden tietueen varaamiseen kuluva aika on riippuvainen siitä kuinka monta lohkoa kyseinen tietue vie. Ritzaun viitelaskurissa sykliset tietorakenteet siivotaan käyttäen merkkaa ja lakaise -tekniikkaa, joka omalta osaltaan lisää muistinsiivouksesta aiheutuvaa hidastusta. Vaikka Ritzaun viitelaskuri onkin varteenotettava vaihtoehto sovelluksissa, joissa ei syklisiä tietorakenteita esiinny,

ei menetelmästä kuitenkaan ole yleiseksi reaaliaikaiseksi muistinsiivoimeksi.

5.6 Ongelmia vähittäisessä siivouksessa

Vähittäisessä siivouksessa on kaksi suurta ongelmaa, jotka vaikeuttavat sen käyttämiseen reaaliaikajärjestelmissä [47]. Ensiksi vaikka yksittäisessä muistinvarauksessa siivoukseen kuluva aika on pieni ja rajoitettu, aiheuttavat useat peräkkäiset muistinvaraukset kokonaisuudessaan pitkän pysähdyksen [3]. Tämä ei tietenkään ole sallittua reaaliaikajärjestelmissä, joissa vaaditaan lyhyttä vasteaikaa. Tämän lisäksi, jotta pystytään takaamaan pieni ja rajoitettu aika, joka yhteen siivousinkrementtiin kuluu, tarvitaan monimutkainen metriikka, jolla muistinsiivoimen tekemää työtä voidaan arvioida. Liian yksinkertaista metriikkaa käytettäessä saatetaan inkrementeissä suorittaa liian paljon työtä siihen nähden, mitä todellisuudessa tarvitaan. Koska muistia siivotaan aina, kun muutin varaa muistia, saadaan helposti todistettua, että suoritetaan riittävä määrä siivousta.

Muistinvarauksen jälkeen vähittäin siivoavat siivoimet saattavat saada suoritusvuoron ennen kuin muutin ehtii alustaa varaamansa olion [28]. Mikäli varattavaa muistia ei alusteta, on vaarana se, että siivoin kerkeää analysoimaan juuri varatun tietueen ja tällöin siivoin saattaa luulla tietuessa olevia vanhentuneita osoittimia käytössä oleviksi. Tästä saattaa aiheutua ylimääräistä leijuvaa roskaa. Järjestelmän toiminnan takaamiseksi muistinsiivoimen on alustettava varattava muisti. Tämä voidaan tehdä joko inkrementaalisesti, jokaisen muistinvarauksen yhteydessä, tai yhtenä atomisena operaationa muistinsiivoussyklin lopussa. Atomisena operaationa suoritettavan muistinalustuksen on luonnollisesti oltava tarpeeksi nopeaa, jotta järjestelmän reaaliaikavasteet saadaan tyydytettyä. Lisäksi kopioivissa siivoimissa muistin alustamattomuus voi aiheutua kohtalokkaaksi, sillä siivoin saattaa virheellisesti kopioida tietueita ja päivittää osoittimia.

Vähittäinen siivous ei ole riittävän tehokasta, jotta tekniikalla saataisiin taattua se, että kaikki reaaliaikatehtävät pystyvät toteuttamaan vasteaikavaatimuksensa. Vähittäinen siivous on kuitenkin hyvä alku muistinsiivouksen reaaliaikaistamiselle. Seuraavassa luvussa kerrotaan, kuinka vähittäistä siivousta voidaan parantaa siten, että saadaan toteutettua muistinsiivoin, jolla reaaliaikavasteiden tyydyttäminen ei vaarannu.

6 Muistinsiivouksen reaaliaikaistaminen

Tässä luvussa esitellään tekniikoita, joilla muistinsiivous saadaan reaaliaikaistettua. Tämä tapahtuu siirtämällä vähittäinen muistinsiivous omaksi reaaliaikavasteet omaavaksi tehtäväksi siten, että kaikki järjestelmän tehtävät saadaan aikataulutettua.

6.1 Reaaliaikaisen muistinsiivouksen vaatimukset

Kasataanpa aluksi edellisten lukujen pohjalta vaatimuksia, jotka reaaliaikaisen muistinsiivoimen tulee toteuttaa. Näiden vaatimusten lisäksi siivoimen tulee luonnollisesti toteuttaa myös kaikki tavalliselle muistinsiivoimelle asetetut vaatimukset. Reaaliaikaisen muistinsiivoimen lisävaatimukset ovat:

Vähittäisyys Muistinsiivous tulee toteuttaa vähittäisesti.

Pirstoutumattomuus Reaaliaikajärjestelmissä muisti ei saa pirstoutua.

Muistin pirstoutuminen voidaan estää käyttämällä joko kopioivaa siivointa tai varaamalla muistia vain yhden ennalta määrätyn kokoisissa alueissa. Kopioivan siivoimen käyttäminen on yleisempää, sillä niiden käyttäminen sekä eliminoi muistin pirstoutumisen että pienentää muistivaraukseen kuluvaa aikaa. Lisäksi tietueiden koostaminen alueista tuottaa suorituskykyongelmia varsinkin silloin, kun ohjelmat käyttävät taulukoita.

Muistinsiivouksen vähittäisyys on välttämätön, jotta siivouksen aiheuttama latenssi pysyy reaaliaikajärjestelmän kannalta siedettävissä rajoissa. Vähittäinen siivous ei yksinään ole riittävää reaaliaikajärjestelmissä, vaikkakin jokaisen muistinsiivoimen alkeistoiminnon on oltava suoritusajaltaan rajoitettu. Muistinsiivouksen vasteaikojen parantamiseksi voidaan siirtää siivous omaksi tehtäväkseen.

6.2 Samanaikainen siivous

Vähittäisessä siivouksessa muistia siivotaan aina, kun muunnin varaa muistia. Tämä hidastaa muistin varaamista ja aiheuttaa muuntimelle ylimääräistä kuormaa. Nämä on-

gelmat saadaan poistettua, kun muistinsiivous siirretään omaksi tehtäväkseen. *Samanaikaisella siivouksella (concurrent garbage collection)* tarkoitetaan sitä, että siivointa ajetaan omana tehtävänä järjestelmän muiden tehtävien rinnalla [47]. Samanaikainen siivous perustuu samaan ideaan kuin vähittäinen siivous; siivousta suoritetaan pienissä inkrementeissä, joiden suoritusaika on lyhyt ja rajattu. Tästä johtuen samanaikaisessa siivouksessa on myös samat ongelmat kuin vähittäisessä siivouksessa muuntimen ja siivoimen yhteistyön toteuttamisessa. Siivoimen on oltava tietoinen muuntimen tekemistä muutoksista ja lisäksi käytettäessä dataa liikuttavia siivoimia, on myös muutin suojattava siivoimen tekemiltä muutoksilta.

Eräs esimerkki samanaikaisesta siivoimesta on nk. Appel-Ellis-Li siivoin [2]. Heidän siivoimensa on kopioiva siivoin, joka toimintoja suoritetaan erillisessä säikeessä. Appel, Ellis ja Li kutsuvat siivointaan reaaliaikaiseksi, mutta väitteelle he eivät anna mitään muita perusteita kuin sen, että siivousta suoritetaan pienissä inkrementeissä. Robertz ja Henriksson toteavatkin, että tyypillisesti näiden niin kutsuttujen reaaliaikaisten muistinsiivointen yhteydessä ei anneta mitään takeita sille, että siivoin pystyy siivoamaan muistia tarpeeksi nopeasti [47].

Henriksson [17] esittää, että samanaikainen siivous on mahdollista toteuttaa siten, että järjestelmän reaaliaikavasteet saadaan tyydytettyä. Tällöin siivous joudutaan aikatauluttamaan muiden tehtävien ohella. Aikataulutuksen vaatimuksena on, että siivoin saa tarpeeksi suoritusaikaa, jotta järjestelmän muisti ei lopu kesken [18]. Muistinsiivouksen aikatauluttamiseksi tehtävänä, jolla on reaaliaikavasteet, on tiedettävä muistinsiivoimen WCET-arvo [29]. Muistinsiivoimen WCET-arvoon vaikuttaa pääasiassa elossa olevan datan määrä, sillä siivoimen vasteaikaan vaikuttaa se kuinka paljon työtä on tehtävä. Merkkaa ja lakaise -tekniikassa elossa oleva data vaikuttaa WCET-arvoon, koska siivoimen on merkattava kaikki elossa olevat tietueet. Tämän lisäksi merkkaa ja lakaise -tekniikassa siivoimen vasteaikaan vaikuttaa myös keon koko, sillä merkkaamisen jälkeen on muisti vielä lakaistava. Kopioivissa siivoimissa WCET-arvon määrää elossa olevan datan määrä, tarkemmin ottaen elossa olevan datan kopiointiin kuluva aika. Lisäksi on tiedettävä tehtävien muistinallokointinopeus, eli se kuinka nopeasti muistinsiivous on saatava valmiiksi, jottei muisti pääse loppumaan. Aivan kuten reaaliaikajärjestelmän muutkin tehtävät, samanaikainen siivous voidaan suorittaa säikeenä, prosessina tai vuorottaisrutiinina.

6.3 Muistinkäytön analysointi

Tehtävien muistinkäytön analysointi voidaan jättää joko ohjelmoijan harteille, tai analyysi voidaan automatisoida. Seuraavissa luvuissa esitetään muutama tapa, jolla elävän datan määrää voidaan arvioida. Arvioinnissa on tärkeää, että saatu arvio on vähintään yhtä suuri kuin mitä todellinen elävän muistin määrä. Liian pienellä arviolla muistinsiivoin ei välttämättä saa tarpeeksi aikaa, jotta siivous saadaan toteutettua. Toisaalta myös liian suuret arviot vaikuttavat järjestelmän toimintaan, sillä tällöin siivoin saattaa saada paljon enemmän suoritusaikaa kuin mitä todellisuudessa olisi tarpeen. Tämä saattaa vaikuttaa järjestelmän tehokkuuteen ja vaikeuttaa muiden tehtävien suorittamista.

6.3.1 Yksinkertainen analyysi

Yksinkertainen metriikka, jolla elävästä datasta saadaan ylimalkainen estimaatti, on käyttää oletusta, että kaikki osoittimet viittaavat uniikkiin olioon [41]. Tällä metriikalla saadaan mahdollisimman pessimistinen arvio elävän datan määrästä, sillä suurempaa arvoa ei todellakaan voida saavuttaa.

Yksinkertaisella metriikalla on kolme ongelmaa, jotka vaikeuttavat sen käyttöä ja vääristävät tuloksia. Nämä ongelmat ovat: rekursiiviset tietorakenteet, olioiden aliasointi, sekä oliopohjaisissa kielissä lisäksi perintä ja virtuaaliset metodit [41].

Rekursiivisissa tietorakenteissa ongelmaksi muodostuu se, että ilman lisäinformaatiota on mahdotonta määrittää kuinka paljon dataa rakenteessa todellakin on [41]. Reaaliaikajärjestelmissä tämä ei kuitenkaan ole ongelma, sillä rekursiivisten rakenteiden syvyys on oltava rajoitettu, jotta rakenteen käsittelemiseen kuluva aika saadaan määritetyksi.

Olioiden aliasoinnilla tarkoitetaan sitä, että kaikki osoittimet eivät osoita uniikkiin olioon, vaan useat osoittimet voivat osoittaa samaan olioon [41]. Tällöin todellinen muistinkäyttö on huomattavasti vähäisempää kuin varovainen arvio siitä, että jokaisen osoittimen päässä on uniikki olio.

Oliopohjaisissa kielissä on huomioitava se, että yliluokan tyyppinen osoitin voi todellisuudessa osoittaa aliluokan instanssiin, jolloin osoitin saattaa osoittaa suurempaan tietomäärään kuin mitä sen tyyppi antaisi ymmärtää [41]. Lisäksi virtuaalimetodeista ei voida varmasti tietää mitä metodia kulloinkin kutsutaan, joten on varauduttava siihen, että virtuaalimetodikutsuissa analyysi tehdään sen metodin mukaan, jonka muistinkäyttö on suurin. Oliokielissä, joissa sallitaan moniperintä, ongelma on vieläkin

```
int *p, *q;
int i = 5;
p = &i;
q = p;
```

Kuva 6.1: Elävän muistin analysointi

vaikeampi, sillä tällöin on periaattessa mahdollista, että osoitin sisältää viitteen oman tyyppiseen olioon tai johonkin aliluokkansa instanssiin ja lisäksi vielä tämä instanssi voi moniperinnän vaikutuksesta sisältää myös muita luokkia, jotka eivät ole relaatiossa osoittimen tyyppiin.

Tarkastellaan lyhyttä C-kielistä esimerkkiä, joka on kuvassa 6.1. Kyseisessä lähdekoodissa on esiteltynä kaksi int-tyyppistä osoitinta p ja q, sekä yksi int-tyyppinen muuttuja i. Käyttämällä edellä esitettyä konservatiivista metodia saadaan elävän muistin määräksi $3 \cdot$sizeof(int), lisäksi on tietysti huomioitava osoittimien vievä tila. Koodista kuitenkin selvästi nähdään, että todellinen elävän muistin määrä on sizeof(int). Käytetyllä metriikalla saadaan estimaatti, joka on selvästi liian suuri. Estimaatin suuruudella on vaikutusta muistinsiivoimelle annettavaan prosessoriaikaan, joten liian suuren estimaatin käyttäminen pakottaa varaamaan muistinsiivoimelle enemmän aikaa kuin mitä se todellisuudessa tarvitsee, ja näin prosessorin hyötykuorma laskee. Tarvitaan siis parempia metriikoita elävän datan määrän määräämiseksi.

6.3.2 Rekursio ja redundantit osoittimet

Persson esittää tekniikan, jolla rekursiosta ja osoittimien aliasoinnista aiheutuvat ongelmat saadaan korjattua [41]. Persson jakaa osoittimet neljään kategoriaan: *juuri (entry)*, *linkki (link)*, *redundantti (redundant)* ja *yksinkertainen (simple)*. Juuri-osoittimet ovat Perssonin jaossa sellaisia osoittimia, jotka ovat tietorakenteiden aloituspisteitä, kuten esimerkiksi linkitetyn listan ensimmäinen alkio. Linkki-osoittimet ovat sellaisia, jotka tuottavat tietorakenteisiin rekursiota, esimerkiksi linkitetyn listan seuraavaa solmua osoittavat osoittimet. Redundantit osoittimet ovat viitteitä sellaisiin olioihin, joihin on viite osoittimesta, joka ei ole redundantti. Tällainen osoitin on esimerkiksi kahteen suuntaan linkitetyn listan solmun edeltäjään osoittava osoitin. Redundantit osoittimet eivät vaikuta elävän muistin määrään, ja ne voidaan analyysissä sivuuttaa, siis ne ovat edellä mainittuja osoittimien aliaksia. Yksinkertaisilla osoittimilla Person tarkoittaa osoittimia, jotka viittaavat tietueisiin, jotka eivät ole rekursiivisia.

```
struct element {
  int data;
  element *next;
  element *prev;
}
```

Kuva 6.2: Rekursiivinen rakenne

Osa osoitintyypeistä voidaan konservatiivisella arvioinnilla tunnistaa automaattisesti [41]. Osoittimet rekursiivisiin olioihin ovat joko juuria tai linkkejä ja osoittimet olioihin, jotka eivät ole rekursiivia ovat yksinkertaisia. Redundanttien osoittimien tunnistaminen ja rekursiivisten rakenteiden maksimisyvyyden määräämistä ei kuitenkaan voida automaattisesti tehdä.

Tarkastellaan ongelmaa esimerkin valossa. Kuvasssa 6.2 on Perssonin esimerkkiä mukaillen toteutettu rekursiivinen tietue, jolla voidaan toteuttaa esimerkiksi kahteen suuntaan linkitetty lista tai binääripuu. Tämän rekursiivisen rakenteen maksimisyvyyttä ei voida C-kielisestä tietuemäärittelystä arvata, vaan on varauduttava päättymättömään rekursioon. Tämä ei kuitenkaan reaaliaikajärjestelmissä ole sallittua, sillä rakenteen käsittelyyn kuluva aika ei tällöin ole rajattu [41]. Jos tiedetään, että kuvassa 6.2 esitetyllä tietueella toteutetaan kahteen suuntaan linkitetty lista, jossa `next`-osoittimella viitataan listan seuraavaan alkioon ja `prev`-osoittimella listan edelliseen alkioon. Tällöin myös tiedetään, että `prev`-osoitin on redundantti. Ilman lisäinformaatiota `prev`-osoittimen redundanttisuudesta lista on mahdotonta erottaa binääripuusta.

Persson [41] esittää, että analyysin helpottamiseksi ohjelmoija voi tuottaa lisäinformaatiota selittämällä lähdekoodistaan sellaisia osioita, joita ei suoralla analyysilla voida tunnistaa. Nämä selitteet lisätään ohjelman kommentteihin ennalta määrättyä syntaksia käyttäen. Kuvassa 6.3 on esitettynä rekursiivinen tietue Perssonin selitteillä siten, että tietueella toteutetaan kahteen suuntaan linkitetty lista. `path-bound`-selitteellä kerrotaan rekursiivisen rakenteen maksimisyvyys, joka tässä tapauksessa on 50 tietuetta. Selitteellä `redundant` ilmaistaan, että kyseinen osoitin on redundantti. Poistamalla redundantit osoittimet analyysista saadaan elävän datan estimaattia tarkennettua huomattavasti. Pelkästään kahteen suuntaan linkitettyä listaa käytettäessä putoaa estimaatti puoleen siitä mitä se olisi luvussa 6.3.1 esiteltyä pessimististä estimaattia käytettäessä.

Näiden selitteiden avulla Persson [41] esittää tekniikan, jolla elävän muistin määrä voidaan rekursiivisesti arvioida. Hänen tekniikassaan lasketaan elävän datan määrä si-

```
struct element /*$ path-bound 50 */ {
  int data;
  element *next;
  element *prev; /*$ redundant */
}
```

Kuva 6.3: Selitykset Perssonin tekniikalla

ten, että kaikissa rekursiivisissa rakenteissa oletetaan olevan maksimimäärä tietueita. Lisäksi hänen artikkelissaan esitetään tapa, jolla myös perinnästä aiheutuvat ongelmat saadaan hoidetuksi, mutta tähän ongelmaan ei tässä tutkielmassa tarkemmin tutustuta.

6.3.3 Allokaatioon perustuva analyysi

Kim, Chang ja Shin [29] esittävät toisenlaisen metriikan, jolla elossa olevan datan määrää voidaan estimoida. Heidän metriikkansa perustuu periodillisten tehtävien yhden periodin aikana varaaman muistin määrän arvioimiseen, sekä tehtävien keskinäisen toiminnan analysointiin. Kim ym. jakavat keosta varattavat tietueet kahteen ryhmään: globaaleiksi ja lokaaleiksi tietueiksi. Lokaalit tietueet ovat sellaisia, joita voi käsitellä vain se tehtävä, joka tietueen varasikin. Globaalit tietueet ovat kaikkien tehtävien käytössä. Heidän menetelmänsä perustuu aikaisemmille tutkimuksille siitä, että globaalien tietueiden määrä pysyy kutakuinkin vakiona ja lokaalien tietueiden määrä vaihtelee kunnes muistinsiivous suoritetaan. Heidän menetelmässään keskitytään tutkimaan lokaalien tietueiden määrää. Tarkemmin sanottuna lokaalin datan maksimimäärän arviointiin.

Kim ym. [29] suorittavat analyysinsä perustuen sille, että tehtävät voivat olla joko aktiivisia tai inaktiivisia. Aktiiviset tehtävät ovat sellaisia, jotka ovat joko suorituksessa tai niiden suoritus on estettynä. Muuten tehtävä on heidän analyysissään inaktiivinen. Kaikkien tehtävien oletetaan olevan periodillisia. Elävän muistin määrä vaihtelee tehtävän periodin aikana, mutta stabiloituu periodin päättyessä. Analyysissään Kim ym. määrittelevät aktiivisen tehtävän elävän muistin määräksi tehtävän periodinsa aikana varaaman muistin määrän suurimman mahdollisen arvon. Tämä tarkoittaa tehtävän pahimman mahdollisen muistinvarauksen määrää, jota he merkitsevät A_i:llä. Inaktiivisilla tehtävillä elävän muistin määräksi asetetaan A_i kerrottuna vakiolla γ_i, joka saa arvon väliltä 0 ja 1. Vakiolla γ_i tarkoitetaan elossa olevaa osaa A_i:n varaamasta muis-

tista, kun tehtävä on inaktiivinen. Sen kuinka arvot A_i ja γ_i löydetään, Kim et al. sivuuttavat.

Kimin, Changin ja Shinin menetelmässä lokaalin datan maksimiarvoksi (L_{max}) saadaan

$$L_{\max} = \max(\sum_{M_i \text{ aktiivinen}} A_i + \sum_{M_j \text{ inaktiivinen}} \gamma_j \cdot A_j), \tag{6.1}$$

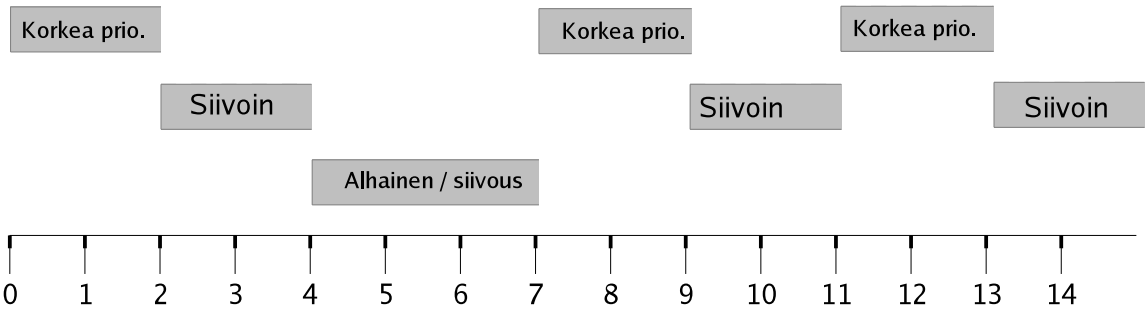
jossa joukolla M_i aktiivinen tarkoitetaan aktiivisia tehtäviä ja joukolla M_j inaktiivinen inaktiivisia tehtäviä. Kaavan triviaaliseksi ylärajaksi saadaan

$$L_{\max} = \sum_{i=1}^{n} A_i. \tag{6.2}$$

Tässä n:llä tarkoitetaan tehtävien lukumäärää. Tämä triviaali maksimi tarkoittaa sitä, että kaikki tehtävät ovat aktiivisia, jolloin korkeimman prioriteetin omaamaa tehtävää suoritetaan ja kaikkien muiden tehtävien suoritus on estettynä. Kim ym. [29] toteavat, että tämä triviaali yläraja on liian pessimistinen ja esittävät kolmekohtaisen menetelmän, jolla arvoa saadaan pienennettyä. Tämä menetelmä kuuluu pääpiirteissään seuraavasti: Aluksi kaikille tehtäville lasketaan *aktiivisuusajat (active windows)*. Aktiivisuusajoilla tarkoitetaan niitä aikoja, jolloin prosessi on aktiivinen. Tämän jälkeen prosesseille etsitään *transitiiviset keskeytysikkunat (transitive preemption window)*. Tämä tarkoittaa sitä, että etsitään kaikki ne kombinaatiot, joilla tehtävät estävät toistensa suorituksen, ja näin saadaan tietoon kaikki kombinaatiot aktiivisista ja inaktiivisista tehtävistä. Tämän jälkeen kullekin keskeytysikkunalle lasketaan sivulla 65 olevan kaavan 6.1 mukainen lokaalin datan maksimiarvo. Transitiivisista keskeytysikkunoista valitaan käytettäväksi se, jolla saadaan suurin arvo $L_{\max}$:lle.

6.4 Muistinsiivouksen aikataulutus

Kun tehtävien muistinkäyttö on saatu analysoiduksi, voidaan muistinsiivousta yrittää aikatauluttaa. Seuraavaksi esitetään menetelmiä, joita aikatauluttamiseen on käytetty. Aikataulutus voidaan tehdä joko työn tai ajan perusteella. Aikataulutettaessa siivousta työn perusteella reagoidaan muuntimen muistinvarauspyyntöihin periaatteessa samalla tavalla kuin vähittäisessä siivouksessa. Ajan perusteella aikataulutettaessa muistinsiivous aikataulutetaan siten, että muistinsiivoussykli päättyy tiettynä ajanhetkenä.



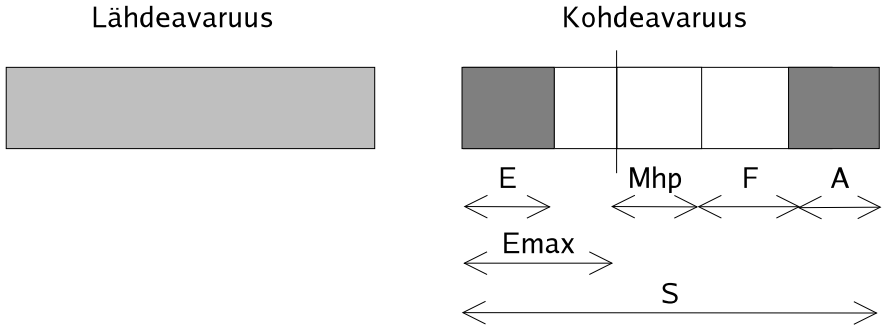
Kuva 6.4: Henrikssonin menetelmä

6.4.1 Henrikssonin osittain samanaikainen siivous

Henriksson [17] esittää aikataulutusanalyysin, joka perustuu varattavan muistin määrään aivan kuin vähittäisen siivouksen tekniikat. Muistinsiivoimen reagoidessa muuntimen tekemiin muistinvarauksiin ongelmaksi muodostuu se, että muuntimen muistinvaraukset voivat tapahtua purskeisesti, jolloin kasaantuvat muistinsiivouksen toiminnot saattavat hidastaa muutinta niin paljon, että reaaliaikavasteita ei saada tyydytetyksi. Henriksson toteaa, että muistinsiivoustyötä ei saisi tehdä laisinkaan silloin, kun korkean prioriteetit prosessit ovat suorituksessa [18]. Muistia on kuitenkin siivottava riittävällä nopeudella, jotta järjestelmän muisti ei pääse loppumaan. Henrikssonin tekniikassa muistinsiivoimen annetaan jäädä jälkeen korkean prioriteetin prosessien suorituksen aikana, mutta siivoin ottaa aikataulunsa kiinni välittömästi korkean prioriteetin prosessien suorituksen jälkeen.

Henrikssonin [17] tekniikassa järjestelmän tehtävät jaetaan kolmeen prioriteettiluokkaan: korkean prioriteetin prosesseiksi, muistinsiivousprosessiksi ja alhaisen prioriteetin prosesseiksi. Siis tehtävä, joka siivoaa muistia, on prioriteetiltään korkean prioriteetin prosessien ja alhaisen prioriteetin prosessien välissä. Prioriteettiluokkien sisällä tehtävillä voi olla lisäksi keskinäinen prioriteettijärjestys. Kuvassa 6.4 on havainnollistettu Henrikssonin mentelmän prosessijakoa. Korkean prioriteetin prosessit suoritetaan ilman häiriötä siivoimelta ja välittömästi näiden prosessien suorituksen jälkeen siivotaan muistia sen verran kuin korkean prioriteetin prosessien muistinvaraukset vaativat. Jäljelle jäävä aika annetaan alhaisen prioriteetin prosessien käyttöön, jotka suorittavat muistinsiivousta vähittäisesti jokaisen muistinvarauksensa yhteydessä. Siivoimen toiminnasta johtuen Henriksson kutsuu menetelmäänsä osittain samanaikaiseksi (semi-concurrent).

Henriksson [17] toteuttaa kopioivan siivoimen, jossa käytetään Brooksin lukumuu-



Kuva 6.5: Keko Henrikssonin menetelmässä

ria (kts luku 5.4.2), mutta toteaa, että analyysia voitaisiin käyttää myös merkkaa ja lakaise -tekniikalla toimivien siivoimien kanssa. Kopioivaa siivointa käytettäessä on mahdollista, että järjestelmä lukkiutuu, jos elossa olevia tietueita ei kopioida kohdeavaruuteen riittävällä nopeudella verrattuna muuntimien allokointinopeuteen [18]. Hänen tekniikassaan alhaisen prioriteetin prosessit ovat velvollisia suorittamaan riittävän siivouksen itse, mutta korkean prioriteetin prosessit varaavat muistia ennen kuin siivous suoritetaan. Ongelmaksi osoittautuu se, että mikäli korkean prioriteetin prosessi saa suoritusvuoron muistinsiivoussyklin lopussa, ei välttämättä ole tarpeeksi vapaata muistia, jotta korkean prioriteetin tehtävän muistinvaraustarve saadaan tyydytettyä. Henriksson ratkaisee ongelman sillä, että kohdeavaruudesta varataan tietty osa pelkästään korkean prioriteetin prosessien käyttöön. Tätä korkean prioriteetin prosessien käyttöön varattavaa muistialuetta Henriksson merkitsee M_{hp}:llä.

Kuvassa 6.5 on kuvattuna keko, joka Henrikssonin tekniikassa on käytössä. Koska käytössä on kopioiva siivoin, siivoamiseen kuluva aika on verrannollinen elävän datan määrään. Tarkemmin sanottuna elävän datan maksimi määrään. Tätä määrää merkitään L_{max}:lla. Jos kohdeavaruuden koko on S, tällöin vapaan muistin määrä muistinsiivoussyklin alussa on $F = S - L_{max} - M_{hp}$. Siivoimen on saatava tarpeeksi suoritusaikaa, jotta siivoussykli saadaan päätökseen ennen kuin vapaa tila F on kulutettu loppuun. Kuvassa E:llä merkitään kopioitujen tietueiden määrää ja A:lla muistinsiivoussyklin aikana varatun muistin määrää.

Varsinainen muistinsiivouksen aikataulutus on Henrikssonin [18] tekniikassa kaksivaiheinen. Ensimmäisessä vaiheessa analysoidaan, voidaanko korkean prioriteetin prosessit aikatauluttaa. Tämä aikataulutusanalyysi tehdään RMA:n perusteella (kts luku 2.3.4). Jos korkean prioriteetin tehtäviä ei saada aikataulutettua, ei järjestelmä ole käyttökelpoinen, eikä tällöin muistinsiivouksen aikataulutuksesta tarvitse huolehtia. Muis-

tinsiivouksella on hieman vaikutusta korkean prioriteetin prosesseihin, sillä siivouksen käytöstä aiheutuu se, että siivointehtävän ja korkean prioriteetin prosessien synkronisointiin käytettävä muuri kasvattaa muutintehtävien maksimisuoritusaikaa. Lisäksi on mahdollista, että järjestelmässä ilmenee hieman enemmän latenssia. Jos korkean prioriteetin tehtävät saadaan aikataulutettua, voidaan alkaa tutkimaan saadaanko myös muistinsiivous aikataulutettua.

Henrikssonin [17] analyysi perustuu sille, että joka kerta, kun korkean prioriteetin prosessi, jonka periodi on T_i, suoritetaan, oletetaan tehtävän suoritukseen kuluvan maksimiajan C_i. Tällä suorituksella prosessi suorittaa muistitoimintoja, jotka vaativat sen, että siivoustyötä on pahimmassa tapauksessa suoritettava G_i yksikköä. Henrikssonin tekniikassa muistinsiivoimen vasteaika (R_{GC}) lasketaan rekursiivisella kaavalla

$$R_{gc}^{n+1} = \sum_{i=1}^{N} \left(\left\lceil \frac{R_{gc}^{n}}{T_i} \right\rceil \cdot (C_i + G_i) \right). \tag{6.3}$$

Kaavassa N:llä tarkoitetaan prosessien lukumäärää. Kaavasta 6.3 saadaan R_{gc} ratkaistua sijoittamalla rekursioon

$$R_{gc}^{0} = \sum_{i=1}^{N} C_i. \tag{6.4}$$

Jos muistinsiivous on mahdollista aikatauluttaa, kaava 6.3 konvergoi [17]. Tämä on helppo havaita, sillä rekursion kaksi peräkkäistä arvoa ovat tällöin yhtä suuret. Henriksson todistaa, että muistinsiivoimen suoritusajan maksimin tulee olla pienempi tai yhtä suuri kuin korkeaprioriteettisten prosessien periodien pienin yhteinen jaettava (least common multiple) $pyj(T_1, ..., T_n)$. Tämä todistus perustuu samaan ideaan kuin RMA, sillä aikataulutuksessa pahin mahdollinen tilanne tapahtuu silloin, kun kaikki tehtävät vapautetaan saman aikaisesti. Tämä tilanne toistuu välttämättä, kun aikaa on kulunut $pyj(T_1, ..., T_n)$ yksikköä. Mikäli muistinsiivoussykliä ei tämän ajan kuluessa saada valmiiksi, ei myöskään seuraavassa tällaisessa periodissa ole riittävästi aikaa muistinsiivouksen suorittamiseen. Näin ollen tekemätön muistinsiivoustyö kumuloituu. Täten rekursion konvergoimattomuus havaitaan, jos yksikin rekursion 6.3 arvo saa suuremman arvon kuin $pyj(T_1, ..., T_n)$.

Sen jälkeen, kun muistinsiivoimen maksimisuoritusaika on saatu laskettua, voidaan laskea tila, joka korkean prioriteetin prosessien muistinvaraukselle on varattava. Varattavaa tilaa Henriksson merkitsee M_{hp}:llä. Tämän muistialueen koko saadaan kertomalla prosessin maksimi muistinvaraustarve (A_i) kerrottuna mahdollisten prosessin

suoritusten, jotka voivat tapahtua muistinsiivousyklin aikana, määrällä [18]

$$M_{hp} = \sum_{i=1}^{N} \left\lceil \frac{R_{gc}}{T_i} \right\rceil \cdot A_i. \tag{6.5}$$

Henrikssonin menetelmällä voidaan taata, että korkeaprioriteettiset tehtävät pystyvät toteuttamaan reaaliaikavasteensa [47]. Lisäksi tämän analyysin avulla tehtävät on mahdollista aikatauluttaa käyttäen kiinteän prioriteetin aikataulutusmenetelmiä. Menetelmässä on myös haittansa, sillä sen avulla ei pystytä takaamaan, että alhaisen prioriteetin prosessit saavat suoritusaikaa, eikä menetelmä suoranaisesti sovellu EDF:llä aikataulutettaviin järjestelmiin. Menetelmän heikkoudeksi voidaan sanoa se, että menetelmän muistinsiivousnopeus on riippuvainen varattavan muistin määrästä, eli siivous aikataulutetaan allokaation perusteella ja siivoussykli päätetään tietyn allokaatiomäärän jälkeen.

6.4.2 Robertzin ja Henrikssonin ajastettu muistinsiivous

Vaikka Henrikssonin menetelmässä voidaankin taata korkeaprioriteettisten prosessien vasteajat, perustuu menetelmä siihen, että siivoimen annetaan reagoida muuntimien tekemiin muistinvarauksiin. Lisäksi muistinsiivoustehtävä aikataulutetaan siten, että siivousta suoritetaan välittömästi korkeaprioriteettisten prosessien suorituksen jälkeen. Robertz ja Henriksson [47] esittävät menetelmän, jolla muistinsiivous voidaan todellakin aikatauluttaa yhdessä muiden järjestelmän tehtävien kanssa. Menetelmää kutsutaan *ajastetuksi (time-triggered)* muistinsiivoukseksi. Myös Robertzin ja Henrikssonin menetelmä toimii osittain samanaikaisesti; prosessit on jaettu kolmeen prioriteettiluokaan: korkeaprioriteettisiin prosesseihin, muistinsiivousprosessiin ja alhaisen prioriteetin prosesseihin. Alhaisen prioriteetin prosessit suorittavat muistinsiivousta vähittäisesti.

Ajastetussa muistinsiivouksessa muistinsiivousprosessi aikataulutetaan siten, että muistinsiivoussykli päättyy tiettynä ajanhetkenä [47]. Tällöin tarvitaan arvio muistinsiivoussyklin pituudelle, sillä sykli on saatava päätökseen ennen kuin järjestelmän muisti loppuu. Tämän arvion on oltava joko täysin oikea tai hieman konservatiivinen. Robertz ja Henriksson todistavat, että muistinsiivoussyklille saadaan laskettua yläraja, jolla voidaan taata, että järjestelmän muisti ei lopu kesken. Heidän ylärajansa on

$$T_{GC} \leq \frac{\frac{(M - L_{\max})}{2} - \sum_{j \in P} A_j}{\sum_{j \in P} f_j \cdot A_j}. \tag{6.6}$$

Kaavassa käytetään seuraavanlaisia merkintöjä: M on keon koko, $L_{\max}$ elävän datan maksimimäärä, P kaikkien prosessien joukko, f_j prosessin j taajuus sekä A_j prosessin j periodin aikana varaaman muistin määrä. Tämä analyysi ottaa huomioon leijuvan roskan tuomat ongelmat.

Kun siivoussyklille ollaan laskettu yläraja, voidaan siivouksen aikataulutusta alkaa miettimään. Kiinteän prioriteetin aikataulutuksessa, kuten RMA:ssa (luku 2.3.4), korkeaprioriteettiset prosessit saavat suoritusvuoron ennen alhaisen prioriteetin prosesseja. Tällöin on pidettävä huoli siitä, että muistinsiivousprosessi ei kuluta kaikkea prosessoriaikaa, jota voitaisiin antaa myös alhaisen prioriteetin prosessien käyttöön. Tästä syystä muistinsiivoimen työ on jaettava tasaisesti koko muistinsiivoussyklin ajalle ja siivoimen lopetettava inkrementtinsä, kun inkrementin aikana muistia on siivottu riittävästi [47]. Tällöin on oltava metriikka sille kuinka paljon muistia on siivottu. Luonnollisesti on taattava, että muistinsiivoussykli saadaan lopetettua ennen kuin T_{GC} on kulunut. Robertz ja Henriksson toteavat, että siivoimen on pidettävä huoli siitä, että muistinsiivousta varten tehdylle työlle pätee

$$\sum w \geq W_{\max} \cdot \frac{t - t_{cs}}{T_{GC}}. \tag{6.7}$$

Tässä kaavassa w:llä tarkoitetaan yksittäisten siivousinkrementtien aikana tehtyä työtä, $W_{\max}$:lla siivousyklin kokonaistyömäärää, t:llä nykyistä ajanhetkeä ja t_{cs}:llä muistinsiivousyklin alkamisaikaa.

EDF-aikataulutuksessa (kts. luku 2.3.6) muistinsiivous voidaan aikatauluttaa suoraan siivousyklin aikarajan perusteella, mutta alhaisen prioriteetin prosessit tuottavat EDF-aikataulutukseen ongelmia, sillä EDF-menetelmällä ei voida aikatauluttaa sellaisia tehtäviän, joille ei ole määrättynä takarajaa [47]. Robertz ja Henriksson kuitenkin esittävät, että tämä ongelma voitaisiin ratkaista käyttämällä tasaisen kaistanleveyden palvelimia (Constant Bandwidth Servers), joille annetaan prioriteetit. Koska asia ei varsinaisesti tutkielman aihepiiriin liity, sivuutetaan sen käsittely.

6.4.3 Kimin, Changin ja Shinin aikataulutusmenetelmä

Robertzin ja Henrikssonin [47] menetelmässä siivousprosessia ajetaan periodisena prosessina, ja siivoin saa suoritusaikaa ennaltamäärätysti, vaikka ei tätä aikaa käyttäisikään. Kim, Chang, Kim ja Shin [28] esittävät mallin, jossa muistinsiivous ajetaan asynkronisena prosessina, jolloin siivous käynnistetään vasta, kun sitä todella tarvitaan.

Kimin ym. [28] menetelmä käyttää modifioitua Brooksin lukumuuriin (kts. luku 5.4.2) perustuvaa kopioivaa siivointa. Heidän kirjoitusmuurissaan osoittimien muutokset aiheuttavat toimenpiteitä, mikäli muutos koskee kohdeavaruuteen kopioidun tietueen osoitinta. Tällöin muutoksesta tehdään nk. *päivitysmerkintä (update entry)*, mikäli osoittimen uusi arvo osoittaa lähdeavaruuteen. Päivitysmerkintä on tietue, jossa säilytetään tieto siitä, mikä kohdeavaruuden tietue on muuttunut ja mikä tietueen kenttä oli muutoksen kohteena. Muistinsiivoussyklin lopussa siivoin käy nämä päivitysmerkinnät läpi ja tutkii muutetut kentät uudelleen. Heidän kirjoitusmuurinsa on periaatteessa muunnelma Nettlesin ja O'Toolen kirjoitusmuurista (kts. luku 5.3.4).

Kim ym. [28] toteavat, että heidän menetelmässään siivoimen suurin mahdollinen suoritusaika riippuu neljän tekijän summasta: juurijoukon maksimikoosta (RS), elävän datan maksimimäärästä ($L_{\max}$), päivitysmerkintöjen maksimimäärästä (ε) ja järjestelmän muistintarpeesta (M). Lisäksi jokaisella näistä tekijöistä on kerroin, jonka arvo riippuu järjestelmän ympäristöstä, kuten laitteistosta, käyttöjärjestelmästä sekä käytetystä kääntäjästä. Näiden kertoimien arvojen löytämiseen he eivät anna mitään tarkkaa analysointitekniikkaa. Esimerkissään Kim ym. saavat nämä kertoimet analysoimalla muistinsiivoimen suoritusta kohdeympäristössä. Kimin ym. muistinsiivoimen suoritusaikaestimaatiksi saadaan

$$C_{GC} = c_1 RS + c_2 L_{\max} + c_3 \varepsilon + c_4 M. \tag{6.8}$$

Kimin ym. [28] menetelmässä asynkroninen muistinsiivoustehtävä aikataulutetaan käyttämällä sporadista palvelinta (kts. luku 2.3.5). Heidän menetelmässään sporadiselle palvelimelle asetetaan suurin mahdollinen prioriteetti. Kiinteän prioriteetin aikataulutuksessa tämä tarkoittaa sitä, että palvelintehtävän periodi on lyhin. Tällöin palvelimen suoritusaikakapasiteetti voidaan laskea kaavan 2.3 (sivu 13) avulla.

Kun muistinsiivoimen suoritukseen kuluva aika sekä sporadisen palvelimen periodi ja suoritusaikakapasiteetti ovat tiedossa, voidaan laskea muistinsiivouksen vasteaika. Sporadisen palvelimen suoritusaikakapasiteetin avulla voidaan laskea kuinka monta palvelimen periodia siivoussyklin päättämiseen tarvitaan. Lisäksi pahin mahdollinen tilanne on silloin, kun muistinsiivouspyyntö saapuu sellaisena ajanhetkenä, että palvelin on juuri käyttänyt kaiken suoritusaikansa, ja palvelimen on odotettava periodinsa loppuun, jotta se saa lisää suoritusaikaa. Kim ym. [28] toteavat, että muistinsiivouksen vasteaika (R_{GC}) saadaan kaavalla

$$R_{GC} = (T_{SS} - SS_{size}) + \left(\left\lceil \frac{C_{GC}}{SS_{size}} \right\rceil - 1 \right) (T_{SS} - SS_{size}) + C_{GC}. \tag{6.9}$$

Tässä kaavassa T_{SS}:llä merkitään sporadisen palvelimen periodia, SS_{size}:llä palvelimen suorituskapasiteettia ja C_{GC}:llä sivun 71 kaavan 6.8 mukaista muistinsiivoimen suoritusaika-arviota.

Kimin, Changin, Kimin ja Shinin [28] menetelmässä muistinsiivous käynnistetään, kun vapaan muistin määrä alittaa tietyn rajan. Tällöin muistinsiivousprosessi vapautetaan suoritukseen. Tätä vapautusaikaa ei tiedetä etukäteen. Näin ollen järjestelmässä tulee olla riittävästi vapaata muistia, jottei muisti pääse siivouksen aikana loppumaan. Tarvittavan vapaan muistin määrä vaihtelee muistinsiivousyklin pituuden mukaan. Mitä kauemmin sykli kestää, sitä enemmän aikaa myös muuntimet saavat, ja sitä enemmän muistia ne syklin aikana kerkeävät varaamaan. Muistinsiivoimen vasteajan avulla voidaan laskea kuinka paljon muistia tehtävät muistinsiivouksen aikana kerkeävät varaamaan eli, kuinka paljon vapaata muistia on oltava muistinsiivoussyklin alussa. Mikä heidän menetelmässään tarkoittaa kopioivan siivoimen kohdeavaruuden kokoa. Kim ym. toteavat, että heidän menetelmänsä kokonaismuistivaatimus (M), lähde- ja kohdeavaruuksien yhteenlaskettu koko, saadaan kaavalla

$$M = 2\left(\sum_{i=1}^{n} \pi_i A_i + L_{\max}\right). \tag{6.10}$$

Tässä A_i:llä tarkoitetaan tehtävän i periodinsa aikana varaaman muistin maksimimäärää, $L_{\max}$:lla elävän muistin maksimimäärää ja π_i:llä prosessin i instanssien maksimimäärää muistinsiivousyklin aikana. Tästä kaavasta puoliavaruuden koko saadaan jakamalla muistin kokonaistarve kahdella. Prosessi-instanssien lukumäärä voidaan puolestaan laskea, koska muistinsiivoimen maksimivasteaika tiedetään. Kimin ym. [28] muistinsiivouksen aikana muutinprosessin i aktiivisten instanssien lukumäärä saadaan kaavalla

$$\pi_i = \left\lceil \frac{R_{GC}}{T_i} \right\rceil + f(i), \text{ jossa } f(i) = \begin{cases} 1, \text{jos } R_i \geq T_i - \left(R_{GC} - (\left\lfloor \frac{R_{GC}}{T_i} \right\rfloor T_i + 1)\right) \\ 0, \text{muulloin} \end{cases}. \tag{6.11}$$

Kaavassa R_{GC}:llä merkataan muistinsiivoimen suurinta mahdollista vasteaikaa, R_i:llä tehtävän i suurinta mahdollista vasteaikaa ja T_i:llä tehtävän i periodin pituutta. Kaavalla lasketaan muistinsiivousyklin aikana suoritettavien prosessien lukumäärä. Tämä arvo ei kuitenkaan kerro kaikkea, sillä on mahdollista, että prosessin suoritus on alkanut ennen muistinsiivoussyklin alkua ja suoritus päätetään muistinsiivousyklin aikana. Tässä tapauksessa muistinsiivoussyklin aikana suoritettavien tehtävien määrää kasvatetaan yhdellä. Tätä lisäystä yllä olevassa kaavassa kuvaa funktion $f(i)$ arvo.

Konservatiivinen arvio saadaan olettamalla, että $f(i)$ saa arvokseen yksi kaikissa tapauksissa. Kimin ym. muistintarpeen analysointimenetelmä on lähestulkoon sama kuin Henrikssonin esittämä analyysi korkean prioriteetin prosessien tarpeeseen varattavan muistialueen koosta, josta kerrottiin luvussa 6.4.1.

Kimin, Changin, Kimin ja Shinin [28] menetelmä asettaa järjestelmälle hieman rajoitteita, sillä heidän menetelmänsä ei salli periodittomia muutintehtäviä. Hyvänä puolena on taasen se, että muistinsiivoimelle annetaan suoritusaikaa vain sen verran kuin se todellisuudessa tarvitsee.

6.5 Aikataulutusmenetelmien vertailua

Aikataulutettaessa muistinsiivousta allokaatioon perustuen on pidettävä huoli, että kuinka paljon työtä kussakin muistinsiivoimen inkrementissä tehdään [47]. Mikäli tästä ei pidetä huolta, saattaa muistinsiivoimen työ jakaantua epätasaisesti muistinsiivoussyklin aikana. Tämän seurauksena muistinsiivousprosessi saattaa syödä suoritusaikaa alhaisen prioriteetin prosesseilta. Allokaatioon perustuva aikataulutus ei myöskään sovellu dynaamisen prioriteetin aikataulutukseen.

Ajastetussa muistinsiivouksessa ei näitä ongelmia ole, ja muistinsiivouksen aikatauluttaminen voidaan jättää järjestelmän yleisen aikataulutusalgoritmin harteille, perustuipa käytetty aikataulutusalgoritmi sitten kiinteisiin tai dynaamisiin prioriteetteihin [47].

Siivouksen aikataulutusmenetelmät vaativat sen, että järjestelmän muistinkäyttö saadaan analysoitua. Reaaliaikajärjestelmissä tämän ei kuitenkaan pitäisi olla ongelma, sillä tehtävien on muutenkin käyttäydyttävä nätisti, jotta niiden suoritus saadaan analysoitua.

7 Kokeelliset tulokset

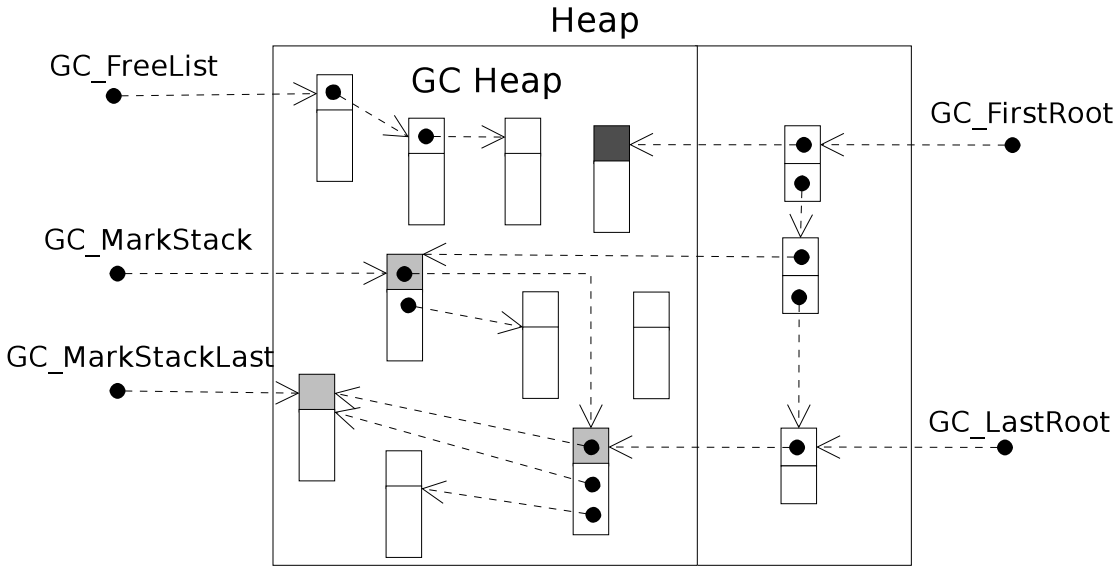
Tässä luvussa esitetään tutkielman ohessa toteutetun vähittäisen merkkaa ja lakaise -tekniikalla toimivan muistinsiivoimen toimintaa. Toteutettu siivoin toimii vähittäisesti, mutta ei samanaikaisesti, joten siitä ei ole reaaliaikaiseksi muistinsiivoimeksi. Luvussa 7.1 tutustutaan siivoimen rakenteeseen ja toimintaan, luvussa 7.2 siivoimen toimintaa analysoidaan. Edellä mainituissa luvuissa on viittauksia siivoimen lähdekoodiin. Lähdekoodi on kokonaisuudessaan liitteessä D ja tarvittava otsikkotiedosto liitteessä C. Vaikka siivoin ei ole reaaliaikainen, luvussa 7.3 analysoidaan, kuinka siivoin toimisi kuvitteellisessa reaaliaikajärjestelmässä. Lopuksi luvussa 7.4 annetaan muutamia kehitysideoita, joilla siivoimen käyttömahdollisuuksia voidaan parantaa.

7.1 Siivoimen rakenne ja toiminta

Kuvassa 7.1 on esitettynä keko, jollaiseen päädytään toteutettua muistinsiivointa käyttämällä. Siivoin varaa keosta itselleen muistialueen, jota se hallinnoi. Kuvassa kekoa kuvaa suuri suorakaide, ja siivoimen hallinnoimaa keon osaa kuvataan suorakaiteella, joka on otsikoitu "GC Heap". Tämä siivoimen hallinnoima alue jaetaan tasakokoisiin lohkoihin, joita käyttämällä toteutetaan kaikki muistinvaraukset, jotka siivoimen hallinnoimaan keon osaan tehdään. Tällä toteutuksella mahdollistetaan se, että muuntimet voivat halutessaan varata muistia joko siivoimen hallinnoimasta keon osasta tai keosta yleisesti, jolloin muisti tarvitsee manuaalisesti vapauttaa. Lohkojen toteutuksesta kerrotaan enemmän luvussa 7.1.1.

Ennen kuin siivointa voidaan käyttää, tulee se alustaa. Tämä tapahtuu kutsumalla funktiota `GC_initialize`, jolle annetaan parametreiksi siivoimen hallintaan annettavan alueen koko tavuina, juurijoukon maksimikoko ja elävien lohkojen maksimimäärä. Siivoin varaa keosta halutun kokoisen alueen käyttöönsä ja alustaa sen jakamalla alueen lohkoiksi, sekä tallentaa nämä lohkot listaan. Kun siivointa ei enää käytetä, on siivoimen tietorakenteet vapautettava. Tämä tapahtuu kutsumalla funktiota `GC_destroy`.

Siivoin käyttää viittä osoitinta keon hallinnointiin. Osoitin `GC_FreeList` osoittaa muistinsiivoimen keossa vapaana olevien lohkojen listan ensimmäiseen alkioon. Vapaista lohkoista kerrotaan enemmän luvussa 7.1.2. Merkkauslistaa ylläpidetään kah-



Kuva 7.1: Muistinsiivoimen keko

della osoittimella: `GC_MarkStack` ja `GC_MarkStackLast`. Näistä ensimmäinen osoittaa merkkauslistan ensimmäiseen alkioon ja jälkimmäinen listan viimeiseen alkioon. Merkkauslistan toteutuksesta kerrotaan tarkemmin luvussa 7.1.3. Loput kaksi osoitinta ovat juurijoukon ylläpitämiseen ja nämä osoittimet ovat: `GC_FirstRoot` ja `GC_LastRoot`. Juurijoukon toimintaa tarkennetaan luvussa 7.1.4. Merkkauslista ja vapaiden lohkojen lista sijaitsee muistinsiivoimen hallinnoimassa keon alueessa, mutta juurijoukko sijaitsee sen ulkopuolella.

Siivoimella on sisäinen tila `GC_State`, jolla kuvataan sitä siivoussyklin vaihetta, jossa siivoin kullakin hetkellä on. Mahdolliset arvot tälle tilalle ovat: IDLING, MARKING ja SWEEPING. Siivoimen tila IDLING kuvaa tilannetta, jossa siivoussykliä ei ole käynnissä. Tila MARKING kertoo tilanteesta, jossa siivoin on merkkausvaiheessa ja SWEEPING ilmaisee, että siivoin lakaisee kekoa. Siivoimen tilan perusteella päätetään se, mitä kunkin inkrementin aikana tehdään. Luonnollisesti siivoimen alustamisen jälkeen on siivoin IDLING-tilassa.

7.1.1 Lohkot ja otsakkeet

Kuten jo aikaisemmin mainittiin, siivoimen hallinnoima keon osa on jaettu ennaltamäärätyn kokoisiin alueisiin pirstoutumisen eliminoimiseksi (luku 3.2.1). Toteutetussa siivoimessa siivoimen hallinnoima keon alue jaetaan samankokoisiin lohkoihin, ja jo-

```
typedef struct obj_header {
        unsigned int          _ptrTable;  /* bitmap of pointers in real "data-area" */
        enum mark_colors      _color;     /* Color of the object */
        void*                 _markNext;  /* Pointer to next block in "mark stack" */
} OBJ_HEADER;
```

Kuva 7.2: Lohkojen otsake

kainen lohko sisältää *otsakkeen (header)*, jossa on tietoa lohkosta. Otsake sisältyy lohkoon, joten valittua lohkokokoa ei voida kokonaisuudessaan käyttää varsinaisen datan esittämiseen, vaan otsakkeen tarvitsema tila voidaan laskea sisäisesti pirstoutuneeksi muistiksi. Pirstoutumista on analysoitu luvussa 7.2.1. Kuvassa 7.2 on tietue, jolla lohkon otsaketta kuvataan. Lohkojen koolla ei toiminnan kannalta ole merkitystä, ja kokoa voidaankin vaihtaa käännösaikaisesti. Siivoimen käyttämä lohkokoko määritetään vakiolla `GC_BLOCKSIZE`.

Otsakkeen kenttä `_ptrTable` on tyypinkuvain, jolla esitetään lohkon varsinaisen data-alueen osoittimet. Tyypinkuvain on toteutettu bittikarttana, jossa bitin arvo `1` merkitsee osoitinta ja arvo `0` atomia. Bittikartta koostetaan siten, että vähiten merkitsevällä bitillä tarkoitetaan lohkon data-alueen ensimmäisestä tavusta alkavaa prosessorin sanaleveyden mittaista muistialuetta, toiseksi vähiten merkitsevä bitti data-alueen toista prosessorin sanaleveyden kokoista muistialuetta ja niin edelleen. Kokonaislukuna esitettävä bittikartta rajoittaa lohkossa olevien osoittimien määrää. Periaatteessa lohko voi olla kooltaan mielivaltainen, mutta siinä olevien osoittimien on mahduttava bittikartan osoittamalle alueelle. Siivoimen toteutus ei kuitenkaan salli sellaisia lohkokokoja, joiden data-alueen kaikkia konesanoja ei voida esittää bittikartassa. Ohjelmoija on vastuussa bittikartan muodostamisesta, ja kartta tarvitsee antaa parametriksi varattaessa muistia siivoimen hallinnoimasta keon osasta.

Kenttää `_color` käytetään lohkon värin määrittämiseen. Tällä kentällä on neljä eri vaihtoehtoa: NONALLOCATED, WHITE, GREY ja BLACK. Viimeiset kolme arvoa ovat oleellisia muistinsiivoimen toiminnan kannalta ja niiden avulla toteutetaan kolmivärimerkkaus. Arvoista ensimmäinen kertoo, että kyseinen lohko on vapaalistassa.

`_markNext`-kenttää käytetään sekä merkkauslistan (kts. luku 7.1.3) että vapaiden lohkojen listan (kts. luku 7.1.2) ylläpitämiseen.

7.1.2 Muistin varaaminen ja siivoimen toiminta

Muistinsiivoimen hallinnoiman keonosan käyttämättömät lohkot säilytetään linkitetyssä listassa. Vapaiden lohkojen listaa ylläpidetään lohkon otsakkeessa sijaitsevalla `_markNext`-osoittimella. Mikäli lohko on vapaana osoittaa tämä kenttä seuraavaan vapaalistan alkioon ja lohkon väri on NONALLOCATED. Vapaalistan viimeisessä alkiossa `_markNext`-osoitin on tyhjä.

Muistia varattaessa tästä listasta otetaan ensimmäinen alkio ja siivoimen vapauttaessa lohkon lisätään vapautettu lohko listan ensimmäiseksi alkioksi. Muistin varaaminen tapahtuu kutsumalla funktiota `GC_malloc`, jolle annetaan parametreiksi tarvittavan muistialueen koko ja bittikartta, jolla kuvataan lohkossa olevien osoittimien paikat. Varatun lohkon väriksi asetetaan vakion NEWOBJECTCOLOR arvo, joka toteutuksessa on musta.

Muistinvarauksen yhteydessä suoritetaan yksi siivoimen inkrementti, jonka toiminnot vaihtelevat siivoimen tilan mukaan. IDLING-tilassa oleva siivoin aloittaa muistinsiivoussyklin merkkaamalla kaikki juuret yhtenä atomisena operaationa. MARKING-tilassa siivoin merkkaa muutaman lohkon. Merkattavien lohkojen määrä määrätään juurijoukon merkkaamisen jälkeen, ja se vaihtelee sen mukaan kuinka paljon vapaata muistia siivoussykliä aloitettaessa on jäljellä. SWEEPING-tilassa oleva siivoin lakaisee hallinnoimansa keon osan yhtenä atomisena operaationa. Lakaisun yhteydessä kaikki ne lohkot, joiden väri on valkoinen, palautetaan takaisin vapaalistaan ja niiden sisältämä data-alue tyhjennetään. Mustat lohkot säilytetään käytössä, mutta niiden väri muutetaan valkoiseksi, jolloin ne saavat mahdollisuuden kuolla seuraavan muistinsiivoussyklin aikana. Siivoimen inkrementti on toteutettuna funktiossa `GC_DoIncrement`. Lakaisu suoritetaan funktiolla `GC_sweep`, ja juurijoukon merkintä funktiolla `GC_markRoots`.

7.1.3 Merkkaus ja kirjoitusmuuri

Siivoimen merkkauspinoa, tai pikemminkin merkkauslistaa, ylläpidetään samoin kuin vapaalistaa. Merkkauslistan seuraavaa alkiota osoitetaan lohkon otsakkeen osoittimella `_markNext`. Merkkauslistaa siivoin ylläpitää kahden osoittimen avulla. Osoittimista ensimmäinen, `GC_MarkStack`, osoittaa merkkauslistan ensimmäiseen alkioon ja osoittimista jälkimmäinen, `GC_MarkStackLast`, listan viimeiseen alkioon. Toteutetussa siivoimessa käytetään rintamahakua, jolloin merkattavat lohkot lisätään merkkauslistan loppuun. Yhtä hyvin voitaisiin käyttää myös syvyyshakua, jolloin merkattavat lohkot lisättäisiin listan alkuun. Lohko merkataan ja lisätään listaan funktion `GC_shadeObject`

avulla.

Siivoin käsittelee merkkauslistaa siten, että se ottaa käsittelyyn listan ensimmäisen lohkon. Tämän lohkon kaikki jälkeläiset tunnistetaan lohkon otsakkeen osoitinbittikartan avulla ja nämä jälkeläiset merkataan. Seuraavaksi lohko poistetaan merkkauslistasta ja vaihdetaan väriltään mustaksi. Tämä toiminto on toteutettuna funktiossa `GC_blackenObject`.

Koska siivoin toimii vähittäisesti tarvitaan siivoimen ja muuntimen välistä synkronointia. Siivoin toteutettiin käyttämällä Dijkstran kirjoitusmuuria (kts. luku 5.3.2). Kirjoitusmuurin toiminta on toteutettuna funktiossa `GC_WriteBarrier` ja ohjelmoija on velvollinen käyttämään tätä funktiota sen sijaan, että hän muuttaisi osoittimien arvoja suoraan.

7.1.4 Juurijoukko

Juurijoukkoa ylläpidetään keossa linkitettynä listana. Kustakin listan alkiosta on osoitin siihen muistinsiivoimen keossa sijaitsevaan lohkoon, joka on juuri. Siivoimella on juurijoukko-listaan kaksi osoitinta: `GC_FirstRoot` ja `GC_LastRoot`. Ensimmäinen osoitin osoittaa listan ensimmäiseen alkioon ja toinen listan viimeiseen alkioon. Uusi juuri lisätään listan loppuun.

Siivoin ei itse tutki mahdollisia juuria, vaan ohjelmoija on velvollinen rekisteröimään juuriksi ne tietueet, jotka kulloinkin ovat juuria. Tämä tapahtuu kutsumalla muistinsiivoimen funktiota `GC_AddRoot`, joka lisää parametrina annetun osoittimen osoittaman lohkon juurijoukkoon. Ohjelmoija on myös velvollinen poistamaan juurijoukosta sellaiset juuret, jotka eivät enää ole juuria. Tämä tehdään kutsumalla muistinsiivoimen funktiota `GC_RemoveRoot`. Kuten aikaisemmin mainittiin, juurijoukolle määrätään maksimikoko siinä vaiheessa, kun ohjelmoija alustaa siivoimen. Uutta juurta ei pystytä rekisteröimään, mikäli juurijoukko on maksimikokoinen. Juurijoukon koolle tarvitsee määrätä yläraja, jotta juurten merkkaamiseen kuluvalle ajalle voidaan määrätä yläraja.

7.1.5 Taulukot siivointa käytettäessä

Käytettäessä muistinsiivointa, joka jakaa kekonsa tasakokoisiin lohkoihin, on taulukoiden käyttäminen ongelmallista. Luvussa 3.2.1 esitettiin kaksi erilaista tapaa, joilla lohkoista voidaan koostaa taulukoita. Siivointa toteutettaessa näistä tekniikoista valittiin yksinkertaisempi ja hitaampi; taulukot luodaan lohkoista lineaarisena listana. Tauluk-

ko koostuu taulukon otsikkolohkosta ja varsinaisista datalohkoista. Otsikkolohkossa, jota esitetään tietueella `arrayHeader`, on tietoa taulukon koosta ja siihen talletettavan datan tyypistä. Varsinaiset datalohkot sisältävät taulukon datan siten, että taulukon data-alkiot sijaitsevat prosessorin sananleveyden päässä toisistaan. Lisäksi kaikki muut lohkot, paitsi viimeinen taulukon lohkoista, sisältää osoittimen taulukkolistan seuraavaan solmuun. Luonnollisesti jokainen taulukon lohko sisältää lohkon otsakkeen ja taulukon käytössä on vain lohkon data-alue, jossa taulukon osake ja data-alkiot sijaitsevat.

Koska taulukon luonti on muuntimesta katsottuna atominen operaatio, on taulukkoa luotaessa suoritettava varotoimenpiteitä, sillä taulukon luomisen aikana suoritetaan inkrementtejä jokaisen taulukon lohkon varaamisen yhteydessä. Taulukon otsikkolohkon luonnin jälkeen on tämä lohko rekisteröitävä juureksi, sillä mikäli näin ei tehtäisi, on mahdollista, että taulukon datalohkoja luotaessa saadaan kaksi siivoussykliä päätökseen, jolloin taulukkoa varten varatut lohkot vapautetaan virheellisesti.

Taulukko luodaan `CreateLinearArray`-funktiolla, jolle annetaan parametriksi taulukon koko ja taulukon tietotyyppi. Funktio palauttaa osoittimen taulukon otsakelohkoon. Taulukkoa voidaan käsitellä funktioilla `GetLinearArray` ja `SetLinearArray`, joille annetaan parametriksi taulukon otsakelohko sekä indeksi taulukon alkiosta, jota halutaan käsitellä. Jälkimmäinen funktio vaatii lisäksi parametrikseen osoittimen tietoon, joka taulukon käsiteltävään alkioon halutaan sijoittaa. Ensimmäinen funktioista palauttaa osoittimen haluttuun taulukon alkioon. Jälkimmäinen funktio ei arvoja palauta, mutta muuttaa taulukon alkion sisältöä.

7.1.6 Muita siivoimen toimintoja

Edellä esitetyn vähittäisen toiminnallisuuden lisäksi siivoimeen toteutettiin toiminto, jonka avulla koko muistinsiivoimen keko voidaan siivota yhtenä atomisena operaationa. Tämä toiminnallisuus on funktiossa `GC_markSweep`. Ohjelmoija voi pakottaa siivoimen siivoamaan muistin välittömästi tätä funktiota kutsumalla.

Kyseistä funktiota käytetään myös varotoimenpiteenä muistinvarauksen yhteydessä suoritettavan vähittäisen siivouksen kanssa. Mikäli vähittäisen siivouksen jälkeen ei muistia ole saatavilla, suoritetaan koko keon kattava atominen siivous ja yritetään muistinvarausta uudelleen. Tällaisesta toiminnallisuudesta on kuitenkin huomautettava, että siihen turvautuminen voi olla erittäin vaarallista reaaliaikajärjestelmissä sillä siivottaessa keko atomisena operaationa ei muutintehtävien vasteaikoja välttämättä saada tyydytettyä.

7.2 Siivoimen toiminnan analyysi

Tässä luvussa analysoidaan siivoimen toteutusta ja pohditaan siivoimen mahdollista reaaliaikakäyttöä. Koska siivoin on vähittäinen eikä samanaikainen, voidaan suoralta kädeltä sanoa, että siivoin ei sovellu yleiskäyttöiseksi reaaliaikaiseksi muistinsiivoimeksi. Aluksi luvussa 7.2.1 analysoidaan lohkojen käytöstä aiheutuvaa sisäisesti pirstoutuneen muistin määrää. Luvussa 7.2.2 tutkitaan siivoimen suurinta mahdollista suoritusaikaa. Luvussa 7.2.3 tutustutaan muistivaatimuksiin, luvussa 7.2.4 tutkitaan mitä ohjelmoijan tarvitsee ottaa huomioon siivointa käyttäessään.

7.2.1 Pirstoutunut muisti

Siivoin, joka käyttää muistinvaraamiseen kiinteän kokoisia lohkoja, eliminoi ulkoisen pirstoutumisen, mutta sisäistä pirstoutumista saattaa esiintyä, mikäli muistinvaraukset eivät ole täsmälleen lohkokoon moninkertoja. Toteutetussa siivoimessa sisäistä pirstoutumista esiintyy aina, sillä lohkojen otsakkeiden viemä tila voidaan mieltää sisäisesti pirstoutuneeksi muistiksi. Lisäksi useinkaan muistinvaraukset eivät ole täsmälleen lohkokoon moninkertoja. Kaikesta huolimatta sisäinen pirstoutuminen on rajoitettua, sillä sisäisen pirstoutuneen muistin maksimimäärä on yhtä tietuetta kohden korkeintaan yhtäsuuri kuin käytetty lohkokoko.

Otsakkeen aiheuttama sisäisesti pirstoutunut muisti vaihtelee eri tietokonejärjestelmien välillä, sillä otsakkeen koko vaihtelee sen mukaan kuinka pitkä tietokoneen sananleveys on, sekä siitä kuinka kääntäjä tietotyypit ilmaisee. Lisäksi otsakkeen kokoon vaikuttaa se, että pakkaako kääntäjä otsakkeessa olevat kentät vai sijaitseeko kukin otsakkeen kenttä omassa konesanassaan. Olettaen, että jokainen otsakkeen kenttä on yhden konesanan mittainen, saadaan otsakkeen koolle yläraja, joka on $3 * \text{sizeof(void*)}$. Tämä yläraja on usein kuitenkin liian suuri. Tarkempi arvo otsakkeen koolle saadaan empiirisellä analyysilla.

Taulukossa 7.1 on esitettynä muutamia mitattuja arvoja erilaisilla laitteisto- ja kääntäjäkokoonpanoilla. Taulukon ensimmäisessä sarakkeessa ilmaistaan käytetyn testilaitteiston konesanan pituus bitteinä. Taulukon toisessa sarakkeessa on käytetty lohkokoko tavuina. Kolmannessa sarakkeessa on kääntäjän ilmoittama otsakkeen koko tavuina. Tässä sarakkeessa on myös ilmaistuna, mikäli otsakkeen lohko on pakattuna. Taulukon viimeisessä sarakkeessa on otsakkeen suhteellinen koko lohkokokoon verrattuna. Testeissä käytettiin GCC-kääntäjän versiota 3.3.2 ja otsakkeet pakattiin kääntäjän optioilla `-fshort-enums` ja `-fpack-struct`.

Konesana	Lohkokoko	Koko	Suhteellinen koko
32	32	12	37,5%
32	32	9 (pakattu)	28,1%
64	32	16	50,0%
64	32	13 (pakattu)	40,6%
32	64	12	18,8%
32	64	9 (pakattu)	14,1%
64	64	16	25,0%
64	64	13 (pakattu)	20,3%

Taulukko 7.1: Lohkon otsakkeen viemä tila

Taulukosta nähdään, että otsakkeen osuus lohkosta on melkoisen suuri, mikäli lohkokoko on pieni, varsinkin sellaisissa käännöksissä, joissa kääntäjä ei pakkaa lohkon otsaketta millään tavalla. Pienillä lohkoilla myös pakatut otsakkeet kuluttavat suhteellisen paljon muistia. Parhaimmillaankin lohkoilla, jotka ovat kooltaan 64 tavua, kuluu lohkojen otsakkeisiin hieman päälle 14 prosenttia kaikesta siivoimen hallinnoimasta muistista. Lohkokokoa kasvattamalla otsakkeen suhteellista kokoa saadaan pienenettyä, mutta samalla kasvaa myös varsinaisen data-alueen sisältämä sisäisesti pirstoutunut muisti. Lohkojen pakkaamista käytettäessä on huomioitava se seikka, että pakattujen otsakkeiden käsitteleminen saattaa olla hitaampaa kuin pakkaamattomien otsakkeiden.

Lisäksi toteutuksessa käytetty taulukoiden esittämistapa pakottaa taulukoiden alkioiden asemoinnin siten, että alkiot sijaitsevat prosessorin sananleveyden päässä toisistaan. Tämä toteutus aiheuttaa sen, että sellaiset lohkot, joilla esitetään taulukoita, sisältävät paljon sisäisesti pirstoutunutta muistia mikäli taulukon alkioiden tietotyypit ovat konesanaa lyhyempiä.

Siivoimen toteutus ei näin ollen sovellu sellaisiin järjestelmiin, joissa muistia on vähän. Tällaisissa järjestelmissä pelkästään lohkojen otsakkeiden viemä tila on kohtuuttoman suuri. Kuten taulukossa 7.1 esitettyjen mittausten perusteella saatiin selville, otsakkeiden viemää tilaa voidaan pienentää pakkaamalla otsakkeen kentät, mutta ihmeitä ei tälläkään tekniikalla saada aikaan. Taulukoiden toteutusta voitaisiin muuttaa siten, että useita sellaisia alkioita, jotka ovat konesanaa lyhyempiä pakattaisiin yhden konesanan sisälle.

7.2.2 Siivoimen suoritusaika

Vaikka siivoin ei reaaliaikavasteita pysty takaamaan, voidaan sen toimintaa silti analysoida reaaliaikaisuus mielessä. Siivoussyklin pituuteen vaikuttaa kolme tekijää: juurijoukon merkkaamiseen kuluva aika, elävän datan merkkaamiseen kuluva aika ja keon lakaisemiseen kuluva aika. Siivoimen vasteaika on mahdollista laskea, sillä kaikkien näiden operaatioiden suoritukselle voidaan määrätä yläraja. Juurijoukon maksimikoon avulla on mahdollista määrittää yläraja juurijoukon merkkaamiseen kuluvalle ajalle. Elävän datan maksimiarvon avulla saadaan määritettyä merkkaamiseen kuluvan ajan maksimiarvo ja koska siivoimen keon koko on tiedossa saadaan lakaisun kestolle määritettyä yläraja.

Juurijoukon merkkaamiseen kuluva maksimiaika on olennaisesti yhden lohkon merkkaamiseen kuluva aika kerrottuna juurijoukon maksimikoolla. Jos juurijoukon maksimikokoa merkataan RS:llä ja yhden lohkon merkkaamiseen kuluva maksimiaika on $\mathrm{C_{markblock}}$, saadaan juurijoukon merkkaamiseen kuluva maksimiaika kaavalla

$$\mathrm{C_{Rootset}} = (\mathrm{C_{markblock}} + \mathrm{c_1}) \cdot \mathrm{RS}, \tag{7.1}$$

jossa c_1:llä tarkoitetaan pientä hidastusta, joka aiheutuu juurijoukon iteroinnista.

Kaiken elävän datan merkkaamiseen kuluvaan maksimiaikaan vaikuttaa lohkon tummentamiseen kuluva aika. Lohkon tummentamisella tarkoitetaan tässä yhteydessä sitä, että harmaaksi värjätty lohko otetaan merkkauslistan alusta, lohkon jälkeläiset merkataan värjäämällä ne harmaaksi ja lisäämällä jälkeläiset merkkauslistaan. Tämän jälkeen lohko värjätään mustaksi. Lohkon tummentamiseen kuluva aika vaihtelee oleellisesti sen mukaan kuinka monta jälkeläistä kullakin lohkolla on. Koska siivoimessa käytetään kiinteän kokoisia lohkoja on jälkeläisten määrä rajoitettu. Jälkeläisten määrän yläraja ($\mathrm{ChildCount_{max}}$) saadaan laskettua kaavalla

$$\mathrm{ChildCount_{max}} = \left\lfloor \frac{\mathrm{GC\_BLOCKSIZE} - \mathrm{sizeof(OBJ\_HEADER)}}{\mathrm{sizeof(void*)}} \right\rfloor. \tag{7.2}$$

Kaavan merkinnät ovat seuraavat: GC_BLOCKSIZE tarkoittaa käytettävää kohkokokoa tavuina, sizeof(OBJ_HEADER) lohkon otsakkeen kokoa tavuina ja sizeof(void*) konesanan pituutta tavuina.

Sen jälkeen, kun lohkon jälkeläisten määrän yläraja on tiedossa, voidaan lohkon tummentamiseen kuluvaa aikaa analysoida. Yläraja ajalle, joka yhden lohkon tummentamiseen kuluu, saadaan kaavalla

$$\mathrm{C_{blacken}} = (\mathrm{C_{markblock}} + \mathrm{c_2}) \cdot \mathrm{ChildCount_{max}} + \mathrm{c_3}. \tag{7.3}$$

Kaavassa c_2 tarkoittaa maksimiaikaa, joka kuluu lohkon jälkeläisten löytämiseen ja c_3 aikaa joka kuluu lohkon värjäämiseen mustaksi ja lohkon poistamiseen merkkauslistasta. Lohkon tummentamiseen kuluvan ajan ylärajan avulla saadaan laskettua elävän datan merkkaamiseen kuluvan ajan yläraja. Tämä yläraja saadaan luonnollisesti kertomalla yhden lohkon tummentamiseen kuluva aika elävien lohkojen maksimimäärällä

$$C_{\text{blackenLiveMax}} = C_{\text{blacken}} \cdot L_{\text{max}}. \tag{7.4}$$

Kaavassa L_{max} tarkoittaa elävän datan maksimimäärää lohkoina. Kolmas asia, joka muistinsiivoimen vasteaikaan vaikuttaa on keon lakaisuun kuluva aika. Lakaisuun kuluva aika on suoraan verrannollinen keon kokoon, sillä kaikki keossa olevat lohkot on käytävä läpi lakaisun aikana. Toteutuksesta johtuen lakaisuvaiheessa yhden lohkon lakaisemiseen kuluu suurin aika silloin, kun lohko tarvitsee palauttaa vapaalistaan. Näin ollen keon lakaisuun kuluvan ajan yläraja (C_{sweep}) saadaan kaavalla

$$C_{\text{sweep}} = (C_{\text{sweepblock}} + c_4) \cdot \text{NOB}. \tag{7.5}$$

Tässä kaavassa $C_{\text{sweepblock}}$ tarkoittaa yhden lohkon vapautuslistaan lisäämiseen kuluvaa maksimiaikaa, NOB:lla lohkojen maksimimäärää ja c_4 pientä viivettä, joka kuluu siihen, että keossa siirrytään seuraavaan lohkoon.

Näiden siivoussyklin osioiden yhteenlaskettujen maksimisuoritusaikojen avulla saadaan määrättyä siivoimen suurin mahdollinen suoritusaika. Tämä suoritusajan yläraja saadaan summaamalla osioiden suurimmat mahdolliset suoritusajat yhteen. Siivoimen suurin mahdollinen suoritusaika (C_{GC}) on

$$C_{\text{GC}} = C_{\text{Rootset}} + C_{\text{blackenLiveMax}} + C_{\text{sweep}}. \tag{7.6}$$

7.2.3 Leijuvat roskat ja muistivaatimukset

Koska siivoin toimii vähittäisesti, eikä siivoa muistia yhtenä atomisena operaationa, esiintyy järjestelmässä leijuvaa roskaa. Leijuvan roskan tarkkaa määrää on vaikea arvioida, sillä leijuvan roskan määrään vaikuttaa muutinohjelmien käyttäytyminen. Toteutettu siivoin käyttää Dijkstran kirjoitusmuuria, joka on tunnettu konservatiivisuudestaan. Näin ollen voidaan olettaa, että leijuvan roskan määrä on suuri. Dijkstran muurissa uudet tietueet luodaan mustina, jolloin ne säilyvät hengissä vähintään parhaillaan menossa olevan siivoussyklin loppuun ja niiden varaama tila vapautetaan aikaisintaan seuraavan siivoussyklin lopussa.

Kirjoitusmuurin toiminnasta johtuen pahimmassa tapauksessa juuri muistinsiivoussyklin päätyttyä keosta on varattuna muistia elävän datan maksimimäärän ja muutintehtävien siivoussyklin aikana varaaman muistin maksimimäärän summa. Näin ollen, jotta järjestelmä olisi käyttökelpoinen on jokaisen muistinsiivoussyklin alussa oltava vapaata muistia vähintään sen verran, kuin muutintehtävät siivoussyklin aikana maksimissaan varaavat.

Siivoussyklin alussa varatun muistin maksimimäärä ($A_{\max}$) noudattaa kaavaa

$$A_{\max} = \sum_{i=1}^{n} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i + L_{\max}. \tag{7.7}$$

Kaavassa R_{GC} ilmaisee siivoimen vasteaikaa, T_i tehtävän i periodia, A_i tehtävän i periodinsa aikana varaaman muistin maksimimäärää, $L_{\max}$ elävän datan maksimimäärää sekä n järjestelmän muutintehtävien lukumäärää.

Vastaavasti vaadittavan muistin minimimäärä ($F_{\min}$), joka siivoussyklin alkaessa tarvitsee olla vapaana saadaan kaavalla

$$F_{\min} = \sum_{i=1}^{n} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i. \tag{7.8}$$

7.2.4 Siivoimen käyttämisen vaikutukset

Koska ohjelmoijan tarvitsee itse rekisteröidä juuret, aiheutuu siivoimen käyttämisestä erittäin suuri taakka ohjelmoijalle. Periaatteessa jokaista aliohjelmakutsua ennen kaikki aliohjelmalle annettavat parametrit on rekisteröitävä juuriksi ja aliohjelmakutsun jälkeen nämä parametrit on poistettava juurijoukosta. Käytännössä voi olla mahdollista, että ohjelman suoritus tiedetään niin hyvin, että aivan kaikkia aliohjelmakutsuja ei tarvitse huomioida juurijoukkoa varten. Olipa tilanne sitten kummin vain, juurijoukon rekisteröinti käsin on virhealtista, eikä esitetty menetelmä siksi ole suositeltava.

Toinen asia, josta ohjelmoija joutuu huolehtimaan käsin on lohkojen osoitinbittikarttojen koostaminen. Useimmissa tapauksissa lohkojen sisällön tietotyypit pysyvät samoina, joten bittikarttojen koostaminen on helppoa. Koostaminen on kuitenkin turhaa työtä, sillä edistyneemmillä menetelmillä tämä voitaisiin tehdä automaattisesti. Eräs mahdollisuus olisi se, että käännösvaiheessa kääntäjä kasaisi osoitinbittikartan automaattisesti.

Tehtävä	T_i[ms]	C_i[ms]	A_i[lohkoa]	L_i[lohkoa]	$R_{size}[lohkoa]$
1	10	1	2	1	1
2	40	5	6	3	2
3	75	20	10	5	2
4	200	40	20	20	13
$\sum$	325	66	38	29	18

Taulukko 7.2: Kuvitteellisen järjestelmän muutintehtävät

7.3 Reaaliakakäyttäytyminen

Seuraavaksi analysoidaan muistinsiivoimen käyttäytymistä kuvitteellisessa reaaliaikajärjestelmässä. Kuvitteellinen järjestelmä koostuu neljästä periodillisesta muutintehtävästä. Muutintehtävien parametrit on lueteltuna taulukon 7.2 riveillä. Taulukon viimeiselle riville on summattu tehtävien parametrien summat. Taulukon ensimmäisessä sarakkeessa on annettu kullekin tehtävälle juokseva indeksi, ja tehtävät on järjestetty prioriteetin mukaiseen kasvavaan järjestykseen. Toisessa sarakkeessa on tehtävän periodin pituus millisekunteina ja kolmannessa sarakkeessa tehtävän maksimisuoritusaika millisekunteina. Sarakkeessa neljä on tehtävän periodinsa aikana varaaman muistin maksimimäärä lohkoina. Viidennessä sarakkeessa on tehtävän elävän muistin maksimimäärä lohkoina ja viimeisessä sarakkeessa on kyseisen tehtävän juurijoukon maksimikoko. Oletetaan lisäksi, että tehtävien takaraja on sama kuin tehtävän periodin pituus.

Muutintehtävien lisäksi järjestelmässä on muistinsiivoustehtävä. Siivoustehtävän parametrit ja järjestelmän yleiset arvot on listattuna taulukossa 7.3. Taulukon ensimmäisessä sarakkeessa on nimettynä parametri, jonka arvo on taulukon toisessa sarakkeessa. Parametrilla sizeof(void$*$) tarkoitetaan konesanan pituutta. GC_BLOCKSIZE-parametri puolestaan tarkoittaa muistinsiivoimessa käytettävää lohkokokoa ja parametrilla sizeof(OBJ_HEADER) kerrotaan lohkon otsakkeen koko tavuina. Parametri $ChildCount_{\max}$ ilmaisee lohkon jälkeläisten maksimimäärän. Parametri $C_{markblock}$ kertoo kuinka kauan yhden lohkon lisäämisen merkkauslistaan maksimissaan kestää ja parametri $C_{sweepblock}$ kertoo maksimiarvon yhden lohkon lakaisemiseen kuluvalle ajalle. Parametrilla NOB ilmaistaan keossa olevien lohkojen maksimimäärä. Lisäksi laskujen yksinkertaistamisen vuoksi oletetaan, että siivoimen parametrit c_1, c_2, c_3 ja c_4 ovat niin pieniä ettei niitä tarvitse huomioida analyysissä. Lisäksi analyysissa oletetaan, että suoritettavan tehtävän vaihtamisesta aiheutuva viivästys on niin pieni, ettei

Parametri	Arvo
sizeof(void*)	4 tavua (32 bittiä)
GC_BLOCKSIZE	64 tavua
sizeof(OBJ_HEADER)	12 tavua
$ChildCount_{\max}$	13
$C_{markblock}$	0,01 ms
$C_{sweepblock}$	0,1 ms
NOB	200 lohkoa

Taulukko 7.3: Siivoimen kuvittelliset parametrit

sitä tarvitse huomioida. Näistä oletuksista kannattaa kuitenkin huomioida se, että oikeita reaaliaikajärjestelmiä toteutettaessa tällaiset oletukset ovat erittäin vaarallisia ja saattavat johtaa siihen, että järjestelmä ei pysty toteuttamaan sille asetettuja vasteaikavaatimuksia.

Aloitetaan analyysi tarkastamalla, että järjestelmän muutintehtävät saadaan aikataulutettua RMA:n mukaisesti.

7.3.1 Muutintehtävien aikatauluttaminen

Muutintehtävien aikatauluttamista voidaan analysoida RMA:n avulla. Tämä tehdään soveltamalla kaavaa 2.1. Tällöin tulokseksi saadaan

$$\sum_{i=1}^{n} \frac{C_i}{T_i} = 0,6916$$

ja

$$U(n) = 0,7240.$$

Saadut arvot ovat sellaiset, että kaava 2.1 pätee, mikä tarkoittaa, että muutintehtävät voidaan aikatauluttaa RMA:n avulla. Analyysin ensimmäinen vaihe on onnellisesti takana ja voidaan ryhtyä varsinaisesti analysoimaan siivoimen suoritusta.

7.3.2 Siivoimen suoritusaika

Seuraavaksi lasketaan siivoimen suurin mahdollinen suoritusaika. Aluksi kaavan 7.1 perusteella lasketaan juurijoukon merkkaamiseen kuluva aika. Tällöin juurijoukon merkkaamiseen kuluva aika on

$$\mathrm{C_{Rootset}} = (\mathrm{C_{markblock}} + \mathrm{c_1}) \cdot \mathrm{RS} = (0,1\mathrm{ms} + 0) \cdot 18 = 1,8\mathrm{ms}.$$

Seuraavaksi lasketaan elävän datan merkkaamiseen kuluva maksimiaika kaavan 7.3 perusteella. Yhden lohkon käsittelyyn kuva aika on

$$\mathrm{C_{blacken}} = (\mathrm{C_{markblock}} + \mathrm{c_2}) \cdot \mathrm{ChildCount_{max}} + \mathrm{c_3} = (0,01\mathrm{ms} + 0) \cdot 13 = 0,13\mathrm{ms}.$$

Kaiken elävän datan tummentamiseen kuluva aika saadaan kertomalla saatu arvo elävien lohkojen maksimimäärällä

$$\mathrm{C_{blackenLiveMax}} = \mathrm{C_{blacken}} \cdot L_{\max} = 0,13\mathrm{ms} * 29 = 3,77\mathrm{ms}.$$

Tässä kaavassa $L_{\max}$:lla tarkoitetaan tehtävien elävien lohkojen yhteenlaskettua maksimimäärää. Kolmantena vaiheena siivoussyklin suoritusajan laskemisessa on lakaisuun kuluvan ajan määrittäminen. Tämä aika saadaan kaavan 7.5 avulla

$$\mathrm{C_{sweep}} = (\mathrm{C_{sweepblock}} + \mathrm{c_4}) \cdot \mathrm{NOB} = (0,1\mathrm{ms} + 0) \cdot 200 = 20\mathrm{ms}.$$

Näiden kolmen arvon perusteella saadaan kokonaisaika, joka siivoussyklin läpiviemiseen kuluu. Tämä aika saadaan kaavan 7.6 perusteella, ja arvoksi saadaan tällöin

$$C_{GC} = \mathrm{C_{Rootset}} + \mathrm{C_{blackenLiveMax}} + \mathrm{C_{sweep}} = 1,8\mathrm{ms} + 3,77\mathrm{ms} + 20\mathrm{ms} = 25,57\mathrm{ms}.$$

Kun siivoimelle on laskettu suurin mahdollinen suoritusaika voidaan analysoida onko siivous mahdollista suorittaa reaaliaikatehtävänä. Tähän paneudutaan seuraavaksi.

7.3.3 Siivoimen suorittaminen reaaliaikatehtävänä

Käytetään muistinsiivouksen varsinaiseen reaaliaika-analyysiin Kimin, Changin ja Shinin aikataulutusmenetelmää, joka on esitettynä luvussa 6.4.3. Menetelmässä siivoustehtävää ajetaan asynkronisena prosessina sporadisena palvelimena, jonka prioriteetti on pienin. Tämä tarkoittaa sitä, että sporadisen palvelimen periodi on pienempi tai yhtä suuri kuin yksikään muutintehtävien periodeista. Asetetaan sporadisen palvelimen periodiksi sama kuin mikä tehtävän 1 periodi on. Näin ollen T_{SS} saa arvokseen 10 millisekuntia. Nyt kaavan 2.3 avulla voidaan sporadiselle palvelimelle laskea suoritusaikakapasiteetti, joka mahdollistaa myös muutintehtävien aikatauluttamisen.

Kaavaa 2.3 soveltamalla päästään seuraavaan lopputulokseen:

$$\begin{aligned}
i = 1: &\quad 1x + 1 \cdot 1 \leq 10 &\Leftrightarrow\quad & x \leq 9 \\
i = 2: &\quad 4x + 4 \cdot 1 + 1 \cdot 5 \leq 40 &\Leftrightarrow\quad & x \leq 7,75 \\
i = 3: &\quad 8x + 8 \cdot 1 + 2 \cdot 5 + 1 \cdot 20 \leq 75 &\Leftrightarrow\quad & x \leq 4,625 \\
i = 4: &\quad 20x + 20 \cdot 1 + 5 \cdot 5 + 3 \cdot 20 + 1 \cdot 40 \leq 200 &\Leftrightarrow\quad & x \leq 2,75
\end{aligned}$$

Saaduista x:n arvoista sporadisen palvelimen suoritusaikakapasiteetiksi (SS_{size}) valitaan pienin. Näin ollen sporadisen palvelimen suoritusaika on 2,75 millisekuntia palvelimen periodin, jonka pituus on 10 millisekuntia, aikana. Palvelimen suoritusaikakapasiteetin, palvelimen periodin ja muistinsiivoimen suurimman mahdollisen suoritusajan avulla voidaan laskea kaavan 6.9 mukainen arvo siivoimen vasteajalle

$$\begin{aligned} R_{GC} &= (T_{SS} - SS_{size}) + (\left\lceil \frac{C_{GC}}{SS_{size}} \right\rceil - 1)(T_{SS} - SS_{size}) + C_{GC} \\ &= (10\text{ms} - 2,75\text{ms}) + \left\lceil \frac{25,57\text{ms}}{2,75\text{ms}} \right\rceil - 1) \cdot (10\text{ms} - 2,75\text{ms}) + 25,57\text{ms} \\ &= 83,57\text{ms}. \end{aligned}$$

Kun muistinsiivoimen vasteaika on selvillä voidaan laskea onko muistinsiivoimen keko tarpeeksi suuri. Tarkastellaan seuraavaksi vaatimuksia siivoimen keon koolle ja tätä analysoidaan seuraavaksi.

7.3.4 Siivoimen keon koko

Analyysin aluksi lyötiin lukkoon siivoimen keon koko. Tämä on välttämätöntä, sillä siivoimen suoritusaika vaihtelee keon koon mukaan. Suoritusaika-analyysin jälkeen on tutkittava onko siivoimelle varattu keon osa riittävän suuri siivouksen reaaliaikaistamiseen. Kaavan 7.7 avulla saadaan laskettua varattujen lohkojen maksimimäärä

$$A_{\max} = \sum_{i=1}^{n} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i + L_{\max} = \cdots = 105.$$

Lisäksi kaavalla 7.8 saadaan laskettua tarvittavien vapaiden lohkojen maksimimäärä

$$F_{\min} = \sum_{i=1}^{n} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i = \cdots = 76.$$

Näiden tulosten summasta saadaan minimikoko, joka siivoimen hallinnoimalla keon osalla tarvitsee olla, jotta muutintehtävien muistinvarauspyynnöt saadaan tyydytettyä. Siivoimen keon tarvitsee siis olla vähintään 181:n lohkon kokoinen.

Analyysia aloitettaessa annettiin alkuarvot, joilla analyysi suoritettiin. Näissä arvoissa siivoimen keon kooksi asetettiin 200 lohkoa. Analyysin perusteella saatiin siivoimen keon koolle vaatimukseksi vähintään 181 lohkoa, joka on pienempi kuin oletettu keon koko. Näin ollen järjestelmään voitaisiin lisätä muistinsiivoustehtävä, joka siivoaisi muistia siten, että muutintehtävien reaaliaikavasteet eivät vaarannu eikä järjestelmän muisti pääse loppumaan.

7.3.5 Huomioita analyysista

Mikäli siivouksen reaaliaikaistaminen ei olisi onnistunut, voitaisiin silti yrittää kahta eri vaihtoehtoa: siivoimelle järjestetään lisää suoritusaikaa tai siivoimelle varattua keon osaa kasvatetaan. Siivoimen suoritusajan kasvattaminen tarkoittaa sitä, että suoritusaikaa on otettava pois muutintehtäviltä, mikä ei ole niin helppo tehtävä. Usein muutintehtävien suoritusaikavaatimuksia on mahdotonta muuttaa. Mikäli siivoimen hallinnoimaa kekoa pystytään kasvattamaan tarpeeksi on mahdollista, että siivointa voitaisiin käyttää reaaliaikajärjestelmässä, vaikka siivoin ei lisää suoritusaikaa saisikaan. Keon kasvattamisessa myös siivoussykliin kuluva suoritusajan maksimi kasvaa, jolloin analyysi tarvitsee tehdä uudelleen. Analyysissä käytettyjen arvojen seurauksena sporadisen palvelimen suoritusaikakapasiteetti jää niin pieneksi, että tämä keino on todennäköisesti tuhoon tuomittu.

Analyysi voitaisiin tehdä myös toisella tapaa. Koska keon maksimikoko ja muutintehtävien ominaisuudet tiedetään, voitaisiin analyysi aloittaa siitä, että näiden arvojen ja kaavojen 7.7 ja 7.8 avulla lasketaan muistinsiivoimen vasteajalle suurin mahdollinen arvo, jota käyttämällä muutintehtävät eivät kerkeä varaamaan muistia enempää kuin mitä keossa on tilaa. Tämän arvon ja sporadisen palvelimen periodin avulla voidaan laskea sporadisen palvelimen suoritusaikakapasiteetille arvo, joka on riittävän suuri siihen ettei muisti lopu kesken. Tämäkään analyysi ei takaa, että siivousta pystytään suorittamaan reaaliaikatehtävänä, mutta analyysin tuloksena saatavien tietojen avulla voi olla mahdollista edelleen analysoida sitä, kuinka muutintehtäviä on muutettava, jotta muistinsiivous saadaan reaaliaikaistettua.

7.4 Jatkokehitysideoita

Vaikka siivoimesta ei sellaisena kuin se toteutettiin ole reaaliaikaiseksi muistinsiivoimeksi, eikä välttämättä edes yleiskäyttöiseksi siivoimeksi, voitaisiin muutamilla pienillä parannuksilla siivoimen käyttömahdollisuuksia parantaa.

Siivoimen toteutus ei mahdollista siivoimen käyttämistä monisäikeisissä ohjelmissa. Ensimmäinen ja tärkein jatkokehityksen kohde olisi se, että siivoin muutettaisiin säieturvalliseksi. Tämä vaatii sen, että sellaiset siivoimen funktiot, joita muutintehtävät käyttävät, suojataan lukoilla. Lukoilla olisi suojattava vähintään juurijoukon käsittelemiseen, lohkojen merkkaamiseen ja vapaalistan käsittelyyn tarkoitetut funktiot.

Siivoimesta saataisiin pienellä vaivalla myös yleiskäyttöinen mielivaltaisen kokoi-

set muistinvaraukset salliva siivoin, mikäli siivoimen hallinnoivan keon osaa ei jaettaisi kiinteän kokoisiin lohkoihin, vaan tästä siivoimen keosta muistia varattaisiin jollain yleisesti tunnetulla muistinvarausalgoritmilla. Eräs tällainen ratkaisu olisi se, että lohkorakenteen sijaan käytettäisiin Buddy Sytem-allokaattoria. Tällöin otsakkeisiin tarvitsisi kuitenkin tehdä muutoksia, sillä yleensä muistinvarausalgoritmit vaativat sen, että kunkin varatun muistialueen koko on tiedossa. Tällaista tekniikkaa käytettäessä menetetään kuitenkin kiinteän kokoisten lohkojen käytöstä saatava ulkoisen pirstoutumisen eliminoituminen. Lisäksi otsakkeiden osoitinbittikarttojen implementaatiota tulisi muuttaa siten, että niiden avulla voidaan esittää mielivaltaisen kokoisten muistialueiden osoittimet.

8 Yhteenveto

Tutkielmassa tutkittiin muistinsiivouksen soveltuvuutta reaaliaikajärjestelmiin, joissa järjestelmän oikeellisuus ja vasteajat ovat kriittisiä tekijöitä. Muistinsiivous on tekniikkana jo melkoisen vanha, ja parin viime vuosikymmen aikana tekniikat ja menetelmät ovat kehittyneet melkoisesti. Reaaliaikajärjestelmissä tehtävät eivät saa epäonnistua muistinkäsittelystä johtuvien virheiden tai muistinvarauspyynnön epäonnistumisen seurauksena. Käyttämällä muistinsiivousta nämä ongelmat saadaan ratkaistua.

Muistinsiivouksen perusalgoritmit joko sisältävät perustavanlaatuisia ongelmia, joiden takia algoritmit eivät ole soveltuvia yleiskäyttöisiksi siivoimiksi, tai pysäyttävät muutintehtävät liian pitkiksi ajoiksi, jotta tehtävien reaaliaikavasteita ei saada tyydytetyksi.

Perusalgoritmit voidaan muuttaa vähittäin toimiviksi, jolloin muistinsiivousta suoritetaan pienissä inkrementeissä. Näiden muistinsiivousinkrementtien WCET-arvot voidaan määrittää, jolloin menetelmiä voidaan soveltaa reaaliaikajärjestelmiin. Tällöin muistinsiivoimen ja muuntimen toimintojen tarvitsee kuitenkin olla synkronoitua. Vähittäinen siivous toteutetaan siten, että muistia siivotaan aina, kun muutin varaa muistia. Toteutustavasta johtuen muistinsiivous voi hidastaa järjestelmää liiaksi, mikäli muunnin varaa muistia purskeisesti. Näin ollen vähittäinen muistinsiivous ei ole riittävä muistinsiivouksen reaaliaikaistamiseen ja sitä voidaan käyttää vain erikoistilanteissa.

Vähittäin toimiva muistinsiivoin voidaan irrottaa omaksi tehtäväksi, jolloin siivouksen aiheuttamat pysähdykset eivät kumuloidu muistinvarausten yhteyteen. Näin muistinsiivous on mahdollista suorittaa reaaliaikatehtävänä, jolloin muistinsiivousta voidaan todellakin käyttää reaaliaikajärjestelmissä. Muistinsiivoustehtävän reaaliaikaistaminen vaatii kuitenkin, että järjestelmän muiden tehtävien muistinkäyttöominaisuudet tiedetään. Lisäksi muistinsiivoustehtävä on otettava huomioon järjestelmän aikataulutusanalyysissa. Tutkielmassa esiteltiin tapoja, joilla tämä analyysi on mahdollista suorittaa.

Tutkielman empiirisessä osuudessa esitettiin tutkielman ohessa toteutetun vähittäisen merkkaa ja lakaise -tekniikalla toimivan siivoimen toimintaa. Koska siivoin toimii vähittäisesti ei sitä voida pitää yleiskäyttöisenä reaaliaikaisena muistinsiivoimena. Toteutetun siivoimen toimintaa myös analysoitiin kuvitteellisessa reaaliaikajärjestelmässä käyttäen tutkielmassa esitettyjä menetelmiä. Analyysin mahdollistamiseksi oletettiin,

että siivoin toimisi reaaliaikaisesti.

Analyysin tuloksena saatiin tieto siitä, että kuvitteelliseen reaaliaikajärjestelmään voitaisiin lisätä muistinsiivoustehtävä, jonka avulla järjestelmän muisti voitaisiin siivota siten, että muutintehtävien reaaliaikavasteet saadaan tyydytettyä. Näin teorioiden toimivuus todennettiin esimerkin avulla.

Tutkielmassa tutustuttiin muistinsiivoukseen vain yksiprosessorisissa tietokoneissa. Näiden menetelmien lisäksi on kehitetty *rinnakkaisia muistinsiivoimia (parallel garbage collection)*, jotka soveltuvat jaetun muistin moniprosessorisiin tietokoneisiin, sekä *hajautettuja muistinsiivoimia (distributed garbage collection)*, joilla myös hajautettujen järjestelmien muistia voidaan siivota.

9 Kirjallisuutta

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] Andrew W. Appel, John R. Ellis, ja Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] David F. Bacon, Perry Cheng, ja V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. Ks. Morrisett ja Aiken [37], ss. 285–298.

[4] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[5] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, maaliskuu 1992.

[6] A. Michael Berman, toim. *ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, GA, toukokuu 1999. ACM Press.

[7] Guillem Bernat ja Alan Burns. New Results on Fixed Priority Aperiodic Servers. Kirjassa *Proceedings 20th IEEE Real-Time Systems Symposium*, ss. 68–78. IEEE Computer Society Press, joulukuu 1999.

[8] Hans-J. Boehm. Destructors, Finalizers, and Synchronization. Ks. Morrisett ja Aiken [37], ss. 262–272.

[9] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. Ks. Steele [53], ss. 256–262.

[10] Dominique Colnet, Philippe Coucaud, ja Olivier Zendra. Compiler support to customize the mark and sweep algorithm. Ks. Jones [26], ss. 154–165.

[11] M. Corti, R. Brega, ja T. Gross. Approximation of worst-case execution time for preemptive multitasking systems. Kirjassa *Proceedings of the ACM SIGPLAN*

2000 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2000), Vancouver, Canada, kesäkuu 2000.

[12] Peter Dickman ja Paul R. Wilson, toim. *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, lokakuu 1997.

[13] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, ja E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, marraskuu 1978.

[14] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yangling Wang, ja James Cheney. Region-Based Memory Management in Cyclone. Kirjassa *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ss. 282–293, Berlin, kesäkuu 2002. ACM Press.

[15] David Gudeman. Representing type information in dynamically-typed languages. Tekninen raportti TR93-27, University of Arizona, Department of Computer Science, Tucson, Arizona, 1993.

[16] Niels Hallenberg, Martin Elsman, ja Mads Tofte. Combining region inference and garbage collection. Kirjassa *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ss. 141–152. ACM Press, 2002.

[17] Roger Henriksson. Predictable automatic memory management for embedded systems. Ks. Dickman ja Wilson [12].

[18] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems.* Väitöskirja, Lund Institute of Technology, heinäkuu 1998.

[19] Martin Hirzel ja Amer Diwan. On the type accuracy of garbage collection. Ks. Hosking [20].

[20] Tony Hosking, toim. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, sarjan *ACM SIGPLAN Notices* osa 36(1), Minneapolis, MN, lokakuu 2000. ACM Press.

[21] Richard L. Hudson. Finalization in a garbage collected world. Ks. Wilson ja Hayes [59].

[22] Richard E. Jones ja Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, heinäkuu 1996.

[23] E. Douglas Jensen. Real-time for the real world: It's not your father's real-time. *NASA JPL Center for Space Mission Information Systems and Software IT Seminar*, 2003. URL: http://www.real-time.org/.

[24] Douglas Johnson. The case for a read barrier. *ACM SIGPLAN Notices*, 26(4):279–287, huhtikuu 1991.

[25] Mark S. Johnstone ja Paul R. Wilson. The memory fragmentation problem: Solved? Ks. Dickman ja Wilson [12].

[26] Richard Jones, toim. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, sarjan *ACM SIGPLAN Notices* osa 34(3), Vancouver, lokakuu 1998. ACM Press.

[27] Antti-Juhani Kaijanaho. Muistinhallinta siivousmenetelmien avulla. LuK-tutkielma, Jyväskylän yliopisto: Tietotekniikan laitos, 2001.

[28] Taehyoun Kim, Naehyuck Chang, Namyun Kim, ja Heonshik Shin. Scheduling garbage collector for embedded real-time systems. Ks. Berman [6], ss. 55–64.

[29] Taehyoun Kim, Naehyuck Chang, ja Heonshik Shin. Bounding worst case garbage collection time for embedded real-time systems. Kirjassa *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, 2000.

[30] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, ja Michael Gonzalez Harbour. *A Practioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Massachusettes, USA, 2003.

[31] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, kolmas laitos, 1997.

[32] B. W. Lampson ja D. D. Redell. Experience with processes and monitors in Mesa. Kirjassa *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, ss. 43–44, 1979.

[33] John Lehoczky, Lui Sha, ja Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior. Kirjassa *Proceedings of the Real-Time Systems Symposium*, ss. 166–171, joulukuu 1989.

[34] Tian F. Lim, Przemyslaw Pardyak, ja Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. Ks. Jones [26], ss. 118–129.

[35] C. L. Liu ja James W. Layland. Scheduling algorithms for multiprogramming in hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[36] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[37] Greg Morrisett ja Alex Aiken, toim. *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, LA, tammikuu 2003. ACM Press.

[38] Frank Mueller ja Uli Kremer, toim. *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, kesäkuu 2003. ACM Press.

[39] Scott M. Nettles ja James W. O'Toole. Real-time replication-based garbage collection. Ks. Wexelblat [57].

[40] Kelvin Nilsen ja H. Gao. The real-time behaviour of dynamic memory management in C++. Kirjassa *IEEE Real-Time Technologies and Applications Symposium*, ss. 142–153, Chicago, toukokuu 1995. IEEE Press.

[41] Patrik Persson. Live memory analysis for garbage collection in embedded systems. Ks. Berman [6], ss. 45–54.

[42] Pekka P. Pirinen. Barrier techniques for incremental tracing. Ks. Jones [26], ss. 20–25.

[43] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. Kirjassa *14th Euromicro conference on Real-Time Systems*, ss. 41–49, kesäkuu 2002.

[44] Glenn Reeves. Re: What really happened on Mars? *Risks-Forum Digest*, 19(58), tammikuu 1998. URL: http://catless.ncl.ac.uk/Risks/19.54.html#subj6.

[45] Tobias Ritzau. *Memory Efficient Hard Real-Time Garbage Collection*. Väitöskirja, Linköping University, toukokuu 2003.

[46] Tobias Ritzau ja Peter Fritzson. Decreasing memory overhead in hard real-time garbage collection. Ks. Sangiovanni-Vincentelli ja Sifakis [49], ss. 213–226.

[47] Sven Gestegøard Robertz ja Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. Ks. Mueller ja Kremer [38].

[48] Erkki Sairanen. *Muistinsiivous*. Pro gradu, Jyväskylän yliopisto: Tietojenkäsittelyopin laitos, Jyväskylä, 1984.

[49] A. Sangiovanni-Vincentelli ja J. Sifakis, toim. *Second International Workshop on Embedded Software (EMSOFT '02)*, sarjan *Lecture Notes in Computer Science* osa 2491, Grenoble, 2002. Springer.

[50] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. Kirjassa *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, marraskuu 2000.

[51] John A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys*, 28(1), 1996.

[52] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, syyskuu 1975.

[53] Guy L. Steele, toim. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, elokuu 1984. ACM Press.

[54] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusettes, kolmas laitos, 1999.

[55] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, toinen laitos, 2001.

[56] Mads Tofte ja Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[57] Richard Wexelblat, toim. *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, sarjan *ACM SIGPLAN Notices* osa 28(6), Albuquerque, NM, kesäkuu 1993. ACM Press.

[58] Paul R. Wilson. Uniprocessor garbage collection techniques. Tekninen raportti, University of Texas, tammikuu 1994.

[59] Paul R. Wilson ja Barry Hayes, toim. *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, lokakuu 1991.

[60] Paul R. Wilson, Mark S. Johnstone, Michael Neely, ja David Boles. Dynamic storage allocation: A survey and critical review. Kirjassa *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[61] Benjamin Zorn. Barrier methods for garbage collection. Tekninen raportti CU-CS-494-90, University of Colorado, 1990. url:ftp://ftp.cs.colorado.edu/pub/cs/techreports/zorn/CU-CS-494-90.ps.Z.

A Termit ja lyhenteet

EDF (engl. Earliest Deadline First) Aikataulutusmenetelmä, jossa korkeimman prioriteetin saa se tehtävä, jonka takaraja on lähimpänä.

Elävä data (engl. live data) Muisti, joka on laillisesti ohjelman käytössä. Elävä data joko kuuluu juurijoukkoon tai siihen on viite elossa olevasta datasta; juurijoukon transitiivinen sulkeuma.

Juurijoukko (engl. root set) Välittömästi ohjelman käytössä olevat tietueet: prosessorin rekistereissä, pinossa ja globaaleissa muuttujissa sijaitsevat tietueet ja osoittimet.

Latenssi (engl. latency) Aika, joka kuluu vapautushetkestä siihen hetkeen, kun tehtävää aletaan suorittaa.

Leijuva roska (engl. floating garbage) Roska, jonka siivoin on ehtinyt merkitä eläväksi ja joka vapautetaan seuraavan siivoussyklin päätyttyä.

Muistivuoto (engl. memory leak) Vapauttamatta jäänyt muistialue, johon ei enää ole osoittimia. Yleinen virhe manuaalista muistinhallintaa käytettäessä.

Muutin (engl. mutator) Järjestelmän varsinainen hyötyohjelma.

Periodillinen tehtävä (engl. periodical task) Tehtävä, jonka pyynnöt saapuvat aina ennalta määritellyin väliajoin.

Perioditon tehtävä (engl. asyncronic task) Tehtävä, jonka pyynnöt voivat saapua milloin vain.

Prioriteetti (engl. priority) Järjestelmän tehtävien keskinäinen tärkeysjärjestys. Suoritusvuoron saa se tehtävä, jonka prioriteetti on korkein.

RMA (engl. Rate Monotonic Analysis) Analyysi, jolla voidaan tukia, onko periodillisten tehtävien joukko aikataulutettavissa. Mikäli RMA:n perusteella tehtävät saadaan aikataulutettua, korkeimman prioriteetin saa se tehtävä, jonka periodi on lyhyin.

Roikkuva osoitin (engl. dangling pointer) Osoitin, joka osoittaa sellaiseen muistialueeseen, jonka on vapautettu. Yleinen virhe manuaalista muistinhallintaa käytettäessä.

Roska (engl. garbage) kts. muistivuoto.

Siivoin (engl. garbage collector) Tehtävä, joka hoitaa muistinsiivouksen.

Sisäisesti pirstoutunut muisti (engl. internal fragmentation) Varatun muistialueen sisällä sijaitseva muistialue, jota ei voida käyttää mihinkään. Johtuu usein pyöristyksistä, joita tehdään muistia varattaessa.

Sporadinen palvelin (engl. sporadic server) Periodilinen tehtävä, jonka suoritusaika varataan periodittomien tehtävien suorittamiseen.

Takaraja (engl. deadline) Aika, jolloin tehtävän suorituksen tulee olla valmiina

Tehtävä (engl. task) Prosessi, säie tai vuorottaisrutiini, joka hoitaa yhden järjestelmän toiminnon.

Tyyppitietoisuus (engl. type accuracy) Siivoimen kyky tunnistaa osoittimet. Siivoin voi olla täysin tyyppitietoinen, jolloin se tunnistaa kaikkien alkioiden tyypin, konservatiivinen, jolloin siivoimella ei ole mitään tietoa alkioiden tyypeistä tai jotain näiden kahden väliltä.

Ulkoisesti pirstoutunut muisti (engl. external fragmentation) Vapaa muistialue, jota ei kuitenkaan voida käyttää, koska alue on liian pieni, jotta muistinvarauspyyntö saadaan tyydytettyä.

Vapautushetki (engl. release time) Ajanhetki, jolloin tehtävän laukaiseva ärsyke saadaan.

Vasteaika (engl. response time) Aika, joka kuluu vapautushetkestä tehtävän suorituksen päättymiseen.

WCET (engl. Worst Case Execution Time) Maksimiaika, joka tehtävän suoritukseen yhden periodin aikana kuluu.

B Käytetyt merkinnät

A_i	Prosessin i periodin aikana varaaman muistin maksimimäärä.
C_i	Prosessin i suurin mahdollinen suoritusaika (WCET, Worst Case Excecution Time).
C_{GC}	Muistinsiivoustehtävän suurin mahdollinen suoritusaika (WCET, Worst Case Execution time).
D_i	Prosessin i takaraja (deadline).
f_i	Prosessin i taajuus.
F	Vapaan muistin määrä.
G_i	Prosessin i periodin aikana tekemistä muistinvarauksista aiheutuva siivoustyön maksimimäärä.
L	Elävän datan määrä
$L_{\max}$	Elävän datan maksimimäärä.
M	Keon koko.
M_{hp}	Korkeaprioriteettisten prosessien muistinvaraustarve siivoussyklin aikana.
R_i	Prosessin i suurin mahdollinen vasteaika.
R_{GC}	Muistinsiivoimen suurin mahdollinen vasteaika.
RS	Juurijoukon maksimikoko.
S	Kopioivan siivoimen puoliavaruuden koko ($= \frac{M}{2}$).
SS_{size}	Sporadisen palvelimen suoritusaikakapasiteetti.
T_i	Prosessin i periodi.
T_{SS}	Sporadisen palvelimen periodi.
$\lceil x \rceil$	Pienin kokonaisluku y, joka toteuttaa ehdon $y \geq x$. Nk. kattofunktio (ceil function).
$\lfloor x \rfloor$	Suurin kokonaisluku y, joka toteuttaa ehdon $y \leq x$ Nk. lattiafunktio (floor function).

C Siivoimen otsikkotiedosto

```
/*
        gctest.h
*/

#ifndef _GC_TEST_H
#define _GC_TEST_H

#include <stdlib.h>

/* ------------------------------------------- */
/* enumeration of Garbage Collector mark colors */
enum mark_colors {
        NONALLOCATED, /* free block (== belongs to free list )*/
        WHITE,        /* unmarked object */
        GREY,         /* grey / marked, but all childs are not yet found by collector */
        BLACK         /* black / marked, and all childs are marked (either gray or black) too */
};

/* ------------------------------------------- */
/* Enumeration of Garbage Collector states */
enum GC_states {
        IDLING,       /* Garbage collector cycle not running */
        MARKING,      /* Collector in marking phase */
        SWEEPING      /* Collector in sweeing pahse */
};

/* ------------------------------------------- */
/* Enumeration of possible array elements */
enum ArrayType {
        CHAR,         /* Char table */
        INT,          /* Integer table */
        DOUBLE,       /* Double array */
        POINTER       /* Pointer array */
};
```

```
/* ------------------------------------------- */
/* Header for each block of the garbage collector heap */
typedef struct obj_header {
        unsigned int        _ptrTable;  /* bitmap of pointers in real "data-area" */
        enum mark_colors    _color;     /* Color of the object */
        void*               _markNext;  /* Pointer to next block in "mark stack" */
}OBJ_HEADER;

/* ------------------------------------------- */
/* Structure to hold garbage collector roots */
typedef struct gc_root {
        struct obj_header *_root;
        struct gc_root  *_nextRoot;
}GC_ROOT;

/* ------------------------------------------- */
/* struct which is used as header to array */
typedef struct arrayHeader {
        int     size;           /* size of array */
        int     blocks;         /* number of blocks from which the array is constructed */
        enum ArrayType type; /* type of array elements */
        void    *arrayPtr;      /* Pointer to first block of array */
} ARRAYHEADER;

/* ------------------------------------------- */
/* Functions to initialize and free Garbage Collector */
void GC_initialize(int HeapSize, int MaxRootSetSize, int MaxLiveBlocks);
void GC_destroy(void);

/* ------------------------------------------- */
/* Functions to reserve memory and to manipulate pointers */
void * GC_malloc(size_t size, unsigned int ptrTable);
void GC_WriteBarrier(void *object,void **ptr, void *target);

/* ------------------------------------------- */
/* Functions to print information of the Garbage collector heap */
void GC_printHeap();
void GC_printHeapStatus();

/* ------------------------------------------- */
/* Functions which force garbage collection */
void GC_markSweep(void);
```

```
/* ------------------------------------------- */
/* functions to handle Garbage collector rootset */
void GC_RemoveRoot(void *object);
void GC_AddRoot(void *object);

/* ------------------------------------------- */
/* functions to manipulate arrays which are created as linear list of blocks */
ARRAYHEADER * CreateLinearArray(int arrsize, enum ArrayType arrtype);
int SetLinearArray(ARRAYHEADER *arrayHeader, int position, void *value);
void * GetLinearArray(ARRAYHEADER *arrayHeader, int position);

#endif
```

D Siivoimen lähdekoodi

```
/*
  gctest.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gctest.h"

/* Private Garbage Collector functions */                                    10
static inline void * GC_getObjectFromHeader(OBJ_HEADER *header);
static inline OBJ_HEADER * GC_getObjHeader(void* object);
static void GC_DoIncrement(void);
static void GC_InitializeFreeList(void);
static void GC_sweep(void);
static void GC_ReturnToFreeList(OBJ_HEADER *header);
static void * GC_GetObjectFromFreeList(void);

/* constants for GC implementation */
const int GC_BLOCKSIZE          = 64;                                    20
const enum mark_colors NEWOBJECTCOLOR = BLACK;

/* RootSet */
GC_ROOT *GC_FirstRoot = NULL;
GC_ROOT *GC_LastRoot = NULL;

/* MarkStack */
OBJ_HEADER *GC_MarkStack = NULL;
OBJ_HEADER *GC_MarkStackLast = NULL;
                                                                         30
/* Pointers to handle Garbage Collector heap */
void *GC_HeapBottom; /* start of GC heap */
void *GC_HeapTop;  /* End of GC heap */
void *GC_FreeList;   /* Pointer to first free block in GC heap */

/* GC Variables */
```

```
int GC_MaxLiveBlocks     = 0;       /* maximum live objects */
int GC_MaxLiveData       = 0;       /* maximun live data in bytes */
int GC_HEAPSIZE          = 4096;    /* default heapsize */
int GC_MaxRootSetSize    = 4;       /* default maximum rootsetsize */                    40
int GC_RootSetSize       = 0;       /* count for current rootsetsize */
int GC_AllocatedBlocks   = 0;       /* count for allocated bytes */
int GC_CurrentCycleIncrementSize = 0; /* GC increment size */
enum GC_states GC_State = IDLING; /* GC state */
int GC_maxChildCount     = 0;       /* Maximum number of pointers in data-area */

/* ------------------------------------ */
/* Creates linearlist of blocks which can be treated as array of given size and type */
ARRAYHEADER *
CreateLinearArray(int arrsize, enum ArrayType arrtype)                                50
{
  int i;
  int arrPtrMask = 0x1;
  int nextPtrOffset = (GC_maxChildCount - 1) * sizeof(void *);
  void *prevArrayBlock;
  void *nextArrayBlock;
  void **nextPointer;

  /* allocate array header */
  ARRAYHEADER *array = (ARRAYHEADER *)GC_malloc(sizeof(ARRAYHEADER), 0x4);             60

  /* Header must be registered as root while we are creating the array.
     Otherwise there could be hazardous situation, where array could be
     freed before it has been fully constructed */
  GC_AddRoot(array);

  /* fill header with required values */
  array->size = arrsize;
  array->type = arrtype;
  array->blocks = (arrsize / (GC_maxChildCount - 1)) + 1;                             70
  if(arrsize % (GC_maxChildCount - 1) != 0)
    array->blocks += 1;

  /* Create pointerbitmap for blocks */
  arrPtrMask = arrPtrMask << (GC_maxChildCount - 1);

  /* Allocate blocks of array and link them */
  for(i = 0; i < array->blocks - 1; i++)
```

```
    {
        nextArrayBlock = GC_malloc( GC_BLOCKSIZE - sizeof(OBJ_HEADER),arrPtrMask);
        if(i == 0)
          {
            array->arrayPtr = nextArrayBlock;
          }
        else
          {
            nextPointer = (prevArrayBlock + nextPtrOffset);
            *nextPointer = nextArrayBlock;
          }
        prevArrayBlock = nextArrayBlock;
    }

  /* Array construction ready, we can remove array from rootset */
  GC_RemoveRoot(array);

  return array;
}



/* ------------------------------------ */
/* Returns pointer to value in given arrayposition */
void *
GetLinearArray(ARRAYHEADER *arrayHeader, int position)
{
  if(position >= arrayHeader->size)
    {
      return 0;
    }

  /* Offset of the pointer which points to next block of array */
  int nextPtrOffset = (GC_maxChildCount - 1) ;

  /* get first real block of the array */
  void *nextArrayBlock = arrayHeader->arrayPtr;
  void **tmpPointer;

  /* how many blocks we have to traverse */
  int blocksToAdvance = position / (GC_maxChildCount - 1);
```

```
  /* offset of machineword, which we want to change, in block */
  int positionInCorrectBlock = position % (GC_maxChildCount - 1);

  int i = 0;

  /* traverse needed block */
  while(i < blocksToAdvance)
    {
      tmpPointer = nextArrayBlock + nextPtrOffset * sizeof(void *);
      nextArrayBlock = *tmpPointer;                                    130
      i++;
    }

  /* count the offset of */
  nextArrayBlock += positionInCorrectBlock * sizeof(void *);


  return nextArrayBlock;
}
                                                                       140
/* ------------------------------------- */
/* Adds pointer to RootSet */
void
GC_AddRoot(void *object)
{
  OBJ_HEADER *header = GC_getObjHeader(object);

  /* Increment root set size, and if we get more roots than specified => error*/
  if(GC_RootSetSize > GC_MaxRootSetSize)
    {                                                                  150
      printf("Max RootSetSize exceeded!!!!!!");
      abort();
    }
  GC_RootSetSize++;

  /* Create new root entry */
  GC_ROOT *newRoot = (GC_ROOT *)malloc(sizeof(GC_ROOT));
  newRoot->_root = header;
  newRoot->_nextRoot = NULL;
                                                                       160
  /* Add root entry to root list */
  if(GC_LastRoot == NULL)
```

```
    {
      GC_FirstRoot = newRoot;
      GC_LastRoot = GC_FirstRoot;
    }
  else
    {
      GC_LastRoot->_nextRoot = newRoot;
      GC_LastRoot = newRoot;                                          170
    }
}

/* ------------------------------------ */
/* Checks if given block is member of RootSet */
int
GC_ObjectInRootSet(void *header)
{
  GC_ROOT *tmp = GC_FirstRoot;
  /* Loop thru roots and return true if given block is member of roots */   180
  while(tmp != NULL)
    {
      if(tmp->_root == header)
        return 1;
      tmp = tmp->_nextRoot;
    }
  /* Block is not part of the rootset */
  return 0;
}
                                                                      190
/* ------------------------------------ */
/* Marks given object and adds it to mark stack */
void
GC_shadeObject(OBJ_HEADER *header)
{
  if(header == NULL) return;

  if(header->_color == WHITE)
    {
      /* Mark block */                                                200
      header->_color = GREY;
      /* And add it to the end of the marklist */
      if(GC_MarkStack == NULL)
        {
```

```
        GC_MarkStack = header;
        GC_MarkStackLast = GC_MarkStack;
      }
    else
      {
        GC_MarkStackLast->_markNext = header;                          210
        GC_MarkStackLast = header;
      }
  }
}

/* ------------------------------------------- */
/* Blackens given object and adds it childs to markstack */
OBJ_HEADER *
GC_blackenObject(OBJ_HEADER *header)
{                                                                      220
  void      *object      = NULL;
  void      **child      = NULL;
  void      *childHeader = NULL;
  OBJ_HEADER *rChildHeader = NULL;
  int       ptrMask      = 0x1;

  int       i;

  if(header == NULL) return NULL;
                                                                       230



  if(header->_color == GREY)
    {
      object = GC_getObjectFromHeader(header);

      /* Loop though ptrTable, and search for child objects */
      for(i = 0; i < GC_maxChildCount; i++)
        {
          /* If child object found, get pointer to it, get it's header and mark it */   240
          if((header->_ptrTable & ptrMask) != 0)
            {
              child = ((void *)object) + i * sizeof(void *);
              if(*child != NULL) childHeader = GC_getObjHeader(*child);
              GC_shadeObject(childHeader);
            }
```

```
        ptrMask = ptrMask << 1;
      }

                                                                    250
    /* Blacken the object and return pointer to next object in markstack */
      OBJ_HEADER *tmp = header->_markNext;
      header->_color = BLACK;
      header->_markNext = NULL;
      return tmp;
    }
}

/* ------------------------------------------- */
/* Removes given object from RootSet */                            260
void
GC_RemoveRoot(void *object)
{
  OBJ_HEADER *header = GC_getObjHeader(object);
  GC_ROOT *tmp = GC_FirstRoot;
  GC_ROOT *prev = NULL;

  /* Loop through rootset */
  while(tmp != NULL)
    {                                                               270
      /* if we found the object from rootset, remove it */
      if(tmp->_root == header)
        {
          if(tmp == GC_FirstRoot)
            {
              GC_FirstRoot = tmp->_nextRoot;
            }
          else
            {
              prev->_nextRoot = tmp->_nextRoot;                     280
            }

          if(tmp == GC_LastRoot)
            GC_LastRoot = prev;

          free(tmp);          /* Free rootset-entry */

          GC_RootSetSize--; /* decrement RootSetSize */
```

```
          break;
        }
      prev = tmp;
      tmp = tmp->_nextRoot;
    }
}

/* ------------------------------------------- */
/* Clears mark of the object */
inline void
GC_ClearMark(OBJ_HEADER *header)
{
  if(header->_color == BLACK)
    {
      header->_color = WHITE;
      header->_markNext = NULL;
    }
}

/* ------------------------------------------- */
/* Returns header of object */
inline OBJ_HEADER *
GC_getObjHeader(void* object)
{
  int blockId = (object - GC_HeapBottom) / GC_BLOCKSIZE;
  void *headerAddr = GC_HeapBottom + blockId * GC_BLOCKSIZE;
  return (OBJ_HEADER*)(headerAddr);
}

/* ------------------------------------------- */
/* Returns pointer to object */
inline void *
GC_getObjectFromHeader(OBJ_HEADER *header)
{
  void *objectAddr = (void *)header;
  objectAddr += sizeof(OBJ_HEADER);
  return objectAddr;
}

/* ------------------------------------------- */
/* Returns 1 if block is colored white, 0 otherwise */
```

```
inline int
GC_ObjectIsWhite(OBJ_HEADER *header)
{
  return (header->_color == WHITE ? 1 : 0);
}

/* ------------------------------------------- */
/* Returns 1 if block is colored black, 0 otherwise */
inline int
GC_ObjectIsBlack(OBJ_HEADER *header)                                    340
{
  return (header->_color == BLACK ? 1 : 0);
}

/* ------------------------------------------- */
/* Returns 1 if block is colored black, 0 otherwise */
inline int
GC_ObjectIsGrey(OBJ_HEADER *header)
{
  return (header->_color == GREY ? 1 : 0);                              350
}

/* ------------------------------------------- */
/* Creates fixed size blocks from heap, and stores them in linear list.
   Basically this function initializes garbage collector heap. */
void
GC_InitializeFreeList(void)
{
  void *tmpPointer = GC_HeapBottom;
  OBJ_HEADER* header;                                                   360

  GC_FreeList = GC_HeapBottom;

  /* Advance thru heap with GC_BLOCKSIZEd increments, treat each block as
     header and add those headers to freelist */
  do
    {
      header = (OBJ_HEADER*)tmpPointer;
      header->_color = NONALLOCATED;
      header->_markNext = tmpPointer + GC_BLOCKSIZE;                    370

      tmpPointer += GC_BLOCKSIZE;
```

```
    }
  while((tmpPointer + GC_BLOCKSIZE) <= GC_HeapTop);

  header->_markNext = NULL;
}

/* ------------------------------------------- */
/* Add given object to first postion of GC_FreeList.                    380
   Clear memory ocupied by object with zeros. */
void
GC_ReturnToFreeList(OBJ_HEADER *header)
{
  header->_markNext = GC_FreeList;
  header->_color = NONALLOCATED;

  /* Fill data-area of block with zeros */
  void *object = GC_getObjectFromHeader(header);
  memset(object, 0, GC_BLOCKSIZE-sizeof(OBJ_HEADER));                    390

  /* Add block to freelist */
  GC_FreeList = (void *)header;
  GC_AllocatedBlocks -= 1;
}

/* ------------------------------------------- */
/* Pops first block of freelist and returns pointer to it */
void *
GC_GetObjectFromFreeList(void)                                           400
{
  if(GC_FreeList == NULL) return NULL;

  void *tmpPointer = GC_FreeList;
  OBJ_HEADER *header = (OBJ_HEADER *)tmpPointer;
  GC_FreeList = header->_markNext;
  header->_markNext = 0;

  return tmpPointer;
}                                                                        410

/* ------------------------------------------- */
/* Initializes Garbage Collector */
void
```

```
GC_initialize(int HeapSize, int MaxRootSetSize, int MaxLiveBlocks)
{
  GC_maxChildCount = (GC_BLOCKSIZE - sizeof(OBJ_HEADER)) / sizeof(void*);

  GC_MaxRootSetSize = MaxRootSetSize;
  GC_RootSetSize = 0;                                                    420

  GC_HEAPSIZE = HeapSize;

  GC_HeapBottom = malloc(GC_HEAPSIZE);

  GC_HeapTop = GC_HeapBottom + GC_HEAPSIZE;
  GC_InitializeFreeList();
  //  GC_MaxLiveData = MaxLiveDataBytes;
  GC_MaxLiveBlocks = MaxLiveBlocks;
                                                                         430
}

/* ------------------------------------- */
/* Free memory ocupied by GC */
void
GC_destroy(void)
{
        free(GC_HeapBottom);
}
                                                                         440
/* ------------------------------------- */
/* Allocates new block from GC's Freelist */
void *
GC_malloc(size_t size, unsigned int ptrTable)
{
  OBJ_HEADER *header;

  /* Current implementation allows to allocate memory areas which are
     blocksized or smaller */
  if( size + sizeof(OBJ_HEADER) > GC_BLOCKSIZE)                          450
    {
      printf("Allocation request too lager!\n");
      abort();
    }

  /* Get next block from FreeList */
```

```
  void *newPointer = GC_GetObjectFromFreeList();

  /* If we have more memory allocated than livedata maximum is, we have to do GC increment */
  if(GC_AllocatedBlocks >= GC_MaxLiveBlocks) GC_DoIncrement();                    460

  if(newPointer == 0)
    {
      /* If we can't allocate memory, Incremental collector has not collected enough,
         therefore collect memory with stop-and-copy aproach => possible miss of deadlines
         in real-time systems. */
      GC_markSweep();

      newPointer = GC_GetObjectFromFreeList();
      if(newPointer == NULL)                                                      470
        {
          printf("Out of memory error!\n");
          abort();
        }
    }

  /* Increase amount of allocated blocks */
  GC_AllocatedBlocks += 1;

  /* fill header with needed values */                                            480
  header = (OBJ_HEADER*)(newPointer);
  header->_ptrTable = ptrTable;
  header->_color = NEWOBJECTCOLOR;

  /* calculate real address of object and return it */
  newPointer += sizeof(OBJ_HEADER);
  return newPointer;
}

/* ------------------------------------- */                                       490
/* Sweeps the heap */
void
GC_sweep(void)
{
  void *tmp, *object;
  OBJ_HEADER *header;

  int freeCount = 0;
```

```
  /* Loop thru all blocks in heap */                                    500
  tmp = GC_HeapBottom;
  while(tmp + GC_BLOCKSIZE <= GC_HeapTop)
    {
      header = (OBJ_HEADER *)tmp;

      /*
        During sweeping,
        free white objects or
        clear mark of black objects.
        If we encounter GreyObjects, there has been error => abort*/   510
      if(GC_ObjectIsWhite(header))
        {
          freeCount += 1;
          GC_ReturnToFreeList(header);
        }
      else if(GC_ObjectIsBlack(header))
        {
          GC_ClearMark(header);
        }
      else if( GC_ObjectIsGrey(header))                                 520
        {
          printf("GC_PANIC: Grey object encountered during sweep!!!!\n");
          abort();
        }
      tmp += GC_BLOCKSIZE;
    }
}

/* ------------------------------------ */
/* Blackens all objects in markstack, and clears stack afterwards */    530
void
GC_mark(void)
{
  while( GC_MarkStack != NULL )
    {
      GC_MarkStack = GC_blackenObject( GC_MarkStack );
    }

  GC_MarkStack = NULL;
  GC_MarkStackLast = NULL;                                              540
```

```
}

/* ------------------------------------------- */
/* Blackens given amount of blocks from markstack */
void
GC_markIncrement(int numBlocks)
{
  int i = 0;
  while(i < numBlocks)
    {                                                          550
      /* Blacken first object of markstack, that is, add children of
         block to markstack and change block color to black */
      GC_MarkStack = GC_blackenObject( GC_MarkStack );

      /* If markstack get empty, stop marking and start sweeping in
         next GC increment */
      if(GC_MarkStack == NULL)
        {
          GC_State = SWEEPING;
          break;                                               560
        }
      i++;
    }
}

/* ------------------------------------------- */
/* Shades all roots GREY and adds them to markstack.
   Operation is done in one atomic operation */
void
GC_markRoots(void)                                             570
{
  GC_ROOT *tmp=GC_FirstRoot;
  while(tmp != NULL)
    {
      GC_shadeObject(tmp->_root);
      tmp = tmp->_nextRoot;
    }
}

/* ------------------------------------------- */              580
/* Stop and collect implementation of mark-sweep algorithm */
void
```

```
GC_markSweep(void)
{
  GC_markRoots();
  GC_mark();
  GC_sweep();
}

/* ------------------------------------- */                                    590
/* Calculates increment size for current GC cycle */
int
GC_CalculateIncrementSize(void)
{
  int FreeMemory = (GC_HEAPSIZE / GC_BLOCKSIZE) - GC_AllocatedBlocks;
  int nof = FreeMemory;
  int noa = GC_AllocatedBlocks;
  if(nof/2 - 2 == 0) nof = 1;
  else
       nof = nof/2 - 2;                                                       600
  GC_CurrentCycleIncrementSize = noa / (nof);
}

/* ------------------------------------- */
/* Does single Garbage Collector increment */
void
GC_DoIncrement(void)
{
  switch(GC_State)
    {                                                                         610
      case IDLING:   /* GC Cycle is started by marking all the roots with atomic operation */
        GC_markRoots();
        GC_State = MARKING;
        GC_CalculateIncrementSize();
        break;

      case MARKING: /* During marking phase, mark objects acording to increment size */
        if( GC_CurrentCycleIncrementSize == 0) GC_CalculateIncrementSize();
        GC_markIncrement(GC_CurrentCycleIncrementSize);
        break;                                                                620

      case SWEEPING:  /* Cycle ends by sweeping the heap with atomic operation */
        GC_sweep();
        GC_State = IDLING;
```

```
    }
}

/* ------------------------------------------- */
/* Sets given element of LinearArray.
   Returns: 0  -  everything ok.                                    630
            -1 -  Error                        */
int
SetLinearArray(ARRAYHEADER *arrayHeader, int position, void *value)
{

  /* Check that we don't have table over or underflow */
  if((position >= arrayHeader->size) || (position < 0) )
    {
      return -1;
    }                                                                640

  /* Offset of the pointer which points to next block of array */
  int nextPtrOffset = (GC_maxChildCount - 1) ;

  /* get first real block of the array */
  void *nextArrayBlock = arrayHeader->arrayPtr;
  void **tmpPointer;

  /* how many blocks we have to traverse */
  int blocksToAdvance = position / (GC_maxChildCount - 1);           650

  /* offset of machineword, which we want to change, in block */
  int positionInCorrectBlock = position % (GC_maxChildCount - 1);

  int i = 0;

  /* traverse needed block */
  while(i < blocksToAdvance)
    {
      tmpPointer = nextArrayBlock + nextPtrOffset * sizeof(void *);  660
      nextArrayBlock = *tmpPointer;
      i++;
    }

  /* count the offset of machineword that we are changing */
  nextArrayBlock += positionInCorrectBlock * sizeof(void *);
```

```
  /* Set the value */
  switch( arrayHeader->type )
    {
      case CHAR:
        *(char *)nextArrayBlock = *(char *)value;
        break;
      case INT:
        *(int *)nextArrayBlock = *(int *)value;
        break;
      case DOUBLE:
        *(double *)nextArrayBlock = *(double *)value;
        break;
      case POINTER:
        nextArrayBlock = value;
      default:
        return -1; /* Error */
    }

  return 0;

}

/* ------------------------------------------- */
/* Implements writebarrier used in collector */
void
GC_WriteBarrier(void *object,void **ptr, void *target)
{

  /* actually change the pointer value */
  *ptr = target;

  if(target == NULL) return;

  OBJ_HEADER *objectHeader = GC_getObjHeader(object);

  /* if object whose pointer we are changin is BLACK actions must be taken.
     Dijktra's write barrier implementation:
       Shade target object, don't modify original */
  if(objectHeader->_color == BLACK)
    {
      OBJ_HEADER *targetHeader = GC_getObjHeader(target);
```

```
        GC_shadeObject(targetHeader);
    }                                                                    710
}
```