Paavo Parkkinen

Markkinapohjaiset mallit käyttöjärjestelmän muistinhallinnassa

Tietotekniikan
Pro Gradu tutkielma
19. elokuuta 2007

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Paavo Parkkinen
Yhteystiedot: pparkkin@jyu.fi
Työn nimi: Markkinapohjaiset mallit käyttöjärjestelmän muistinhallinnassa
Title in English: Market-based models for operating system memory management
Työ: Tietotekniikan Pro Gradu tutkielma
Sivumäärä: 66
Tiivistelmä: Käyttöjärjestelmiä käyttävien sovellusten ja käyttäjien joukon kasvaessa ja monipuolistuessa, muistinhallintajärjestelmät joutuvat entistä hankalampien haasteiden eteen. Sovellusten muistitarpeiden muuttuessa entistä ennalta arvaamattomammiksi ja monipuolisemmiksi perinteiset keskitetyt muistinhallintamallit eivät pysty enää jakamaan muistia tietokonejärjestelmissä sovellusten tarpeiden mukaisesti. Eräs tapa ratkaista muistinhallinnan ongelmat tulee ihmistalouksien toiminnasta. Käyttöjärjestelmä voidaan asettaa myymään tietokonejärjestelmän keskusmuistia sovelluksille, jotka voivat näin itse paremmin toteuttaa itselleen sopivinta muistin jaon toimintalinjaa niille osoitettujen varojen puitteissa.
English abstract: As the number of users and applications on a single operating system grows, the memory management subsystems of the operating systems are faced with increasingly difficult challenges. The memory needs of applications are becoming more unpredictable and diverse, and traditional centralized models for memory management are less able to meet those needs. A way to handle this increasing complexity comes from the way human economies work and distribute resources. The operating system can be seen as the owner of a certain amount of memory resources, that it tries to sell to the users and applications of the system. The users and applications buy the memory they need to carry out their own memory usage policy using funds allocated to them.
Avainsanat: muistinhallinta, käyttöjärjestelmät
Keywords: memory management, operating systems

Copyright © 2007 Paavo Parkkinen

All rights reserved.

Sisältö

1 Johdanto **1**

2 Käyttöjärjestelmistä **4**

2.1 Resurssienhallintatehtävä 4

3 Muistinhallinta **6**

3.1 Virtuaalimuisti 8

3.2 Sivunkorvaus 9

3.3 Mach 11

3.4 UVM 15

4 Kansantaloustiede **22**

4.1 Hinnanmuodostus 24

4.2 Resurssien jakautuminen 27

5 Markkinapohjaiset mallit muistinhallinnassa **29**

5.1 Hinnanmääritys 32

5.2 Agentit 34

5.3 Hartyn ja Cheritonin muistinhallintamalli 35

5.4 Utahin yliopiston TENEX järjestelmä 38

6 Oma toteutukseni markkinapohjaisesta muistinhallintajärjestelmästä **41**

6.1 Muistinhallinta Intel arkkitehtuurissa 41

6.2 Movitz 43

6.3 Toteutus 44

6.3.1 Kellokäsittelijä 44

6.3.2 Fyysinen muistiavaruus 44

6.3.3 Sivuvälimuistihoitaja 44

6.3.4 Toteutuksen testaus 45

7 Kysymyksiä ja jatkopohdintaa **48**

8 Viitteet **51**

Liitteet

A Lähdekoodi **53**

1 Johdanto

Tietokoneiden ja verkkoteknologioiden kehittyessä lukuisia tietokonejärjestelmiä voidaan kytkeä toisiinsa lisäresurssien tarjoamiseksi käyttäjille ja sovelluksille. Kasvavia järjestelmiä puolestaan käyttää kasvava ja lisääntyvässä määrin epäyhtenäinen joukko käyttäjiä. Kasvavat ja monipuolistuvat resurssit ja käyttäjäjoukot lisäävät jatkuvasti järjestelmien monimutkaisuutta.

Monimutkaisuutta lisäävät myös muut seikat. Useissa järjestelmissä eri resurssit saattavat olla eri organisaatioiden omistuksessa. Järjestelmän suorituskyky myös määräytyy useiden resurssien samanaikaisesta osoituksesta, ja joidenkin sovellusten suorituskykyyn voidaan vaikuttaa vaihtamalla resursseja toisiin. Suuressa järjestelmässä myös käyttäjät ja suoritettavat sovellukset vaihtuvat jatkuvasti. Lisäksi järjestelmien komponenttien autonomia suhteessa järjestelmään lisääntyy, joka vaikeuttaa resurssienhallintaa.

Monimutkaisuus tekee perinteisistä resurssienhallintatavoista epäkäytännöllisiä. Perinteiset tavat pyrkivät optimoimaan jotain järjestelmänlaajuista suorituskyvyn mittaria, kuten esimerkiksi vasteaikaa. Optimointi tehdään joko keskitetyllä algoritmilla tai hajautetulla konsensukseen pyrkivällä algoritmilla. Resurssien hallinnan monimutkaisuus ja järjestelmän epäyhtenäisyys tekee kuitenkin mahdottomaksi määrittää hyväksyttävää järjestelmänlaajuista suorituskyvyn mittaria, ja usein eri sovelluksilla onkin hyvin erilaiset tavat mitata omaa suorituskykyään.[5]

Sama resurssien jakamisen monimutkaisuus ilmenee myös ihmistalouksissa johtuen samoista syistä kuin tietokonejärjestelmissä, eli talouden hajautuneisuudesta ja toimijoiden ja resurssien epäyhtenäisyydestä. Yleisesti ottaen modernit taloudet jakavat resursseja järjestelmissä, joiden monimutkaisuus on tietokoneille käytännöllisesti katsoen mahdotonta käsitellä.[5]

Markkinapohjaiset mallit tarjoavatkin useita mielenkiintoisia apuja tietokonejärjestelmien resurssienjakoalgoritmeihin. Ensinnäkin mallit tarjoavat joukon työkaluja resurssien hallinnan monimutkaisuuden rajoittamiseksi. Toiseksi mallit tarjoavat joukon matemaattisia malleja jotka auttavat ymmärtämään resurssien jako-ongelmia paremmin.

Markkinataloudessa hajauttaminen syntyy itsenäisten toimijoiden pyrkiessä itsekkäästi toteuttamaan päämääriään. Toimijoita on kahdenlaisia: tuottajia ja kuluttajia. Tuottajat pyrkivät maksimoimaan voittonsa tarjoamalla resursseja kuluttajille maksua

vastaan. Kuluttajat puolestaan pyrkivät maksimoimaan tarpeentyydytyksensä ostamiaan resursseja kuluttamalla.[5]

Mallit käyttävät yleensä rahaa ja hinnanmuodostusta toimijoiden toiminnan ohjaamiseksi. Tuottajat omistavat resursseja, joiden käytöstä ne saavat kuluttajilta rahaa. Kuluttajat saavat rahaa, jota ne käyttävät resurssien ostamiseen. Resurssin hinta määräytyy sen kysynnän ja tarjonnan mukaan. Tuottajat rajoittavat pääsyä resursseihin hinnoilla, ja kuluttajat ostavat resursseja tarpeidensa tyydyttämiseksi. Tuottaja määrää hinnat kuluttajien kysyntää vastaaviksi, ja kuluttajien kysyntä määräytyy hintojen mukaan.

Järjestelmän resurssienhallintatehtävän helpottamisen ja tehostamisen lisäksi markkinapohjaisista resurssienhallintamalleista voi olla myös hyötyä joillekin sovelluksille. Merkittävä joukko muisti-intensiivisiä sovelluksia voisi hyötyä paljon mahdollisuudesta hallita omaa muistiaan. Tähän joukkoon kuuluvat esimerkiksi suuret simulaatiot ja tietokannanhallintajärjestelmät.[9]

Perinteinen tapa hoitaa järjestelmän laajuinen muistinhallinta on käyttää keskitettyä muistin hallintaa, joka pyrkii määrittämään sovellusten muistitarpeet tarkkailemalla niiden muistin käyttöä ja viittauskäyttäytymistä. Muistinhallinta on täysin sovellusten saavuttamattomissa, ja samaa mallia käytetään kaikkien sovellusten muistin hallintaan.

Perinteinen tapa toimi hyvin vähän muistia sisältävissä järjestelmissä, koska malli pystyi hyvin arvioimaan sovellusten muistikäyttäytymistä. Keskitetty järjestelmänlaajuinen muistinhallinta on kuitenkin hankalaa muisti-intensiivisille sovelluksille. Ohjelmien pitkän aikavälin käyttäytymistä on vaikea arvioida luotettavasti. Lisäksi muisti-intensiivisten sovellusten viittauskäyttäytyminen on liian monimutkaista arvioida yleisesti käytetyillä malleilla. Sovellusten muistikäyttäytymisen arvioimiseen käytetyt mallit eivät myöskään osaa ottaa huomioon joidenkin sovellusten mahdollisuutta sopeuttaa toimintaansa saatavilla olevan muistin mukaan.

Markkinapohjaisia lähestymistapoja käyttäessä sovelluksille voidaan suoda huomattavasti enemmän määräysvaltaa omaan muistin käyttöönsä. Markkinapohjaiset lähestymistavat voivat myös tarjota uudenlaisia mekanismeja resurssien jakamiseksi sovellusten kesken. Sovelluksille annettavilla tuloilla voidaan ohjata resurssien käyttöä järjestelmän ylläpitäjän toivoman toimintalinjan mukaan. Tulot määräävät sovellusten välisen tärkeyden rajoittaen sovellusten saatavilla olevia resursseja suhteessa järjestelmän muihin sovelluksiin. Lisäksi sovellusten vuokratessa muistia ennalta määrätyksi ajaksi järjestelmä saa suoraan tiedon sovelluksille osoitettavasta muistista ja sen vapautumisesta, eikä muistinhallintajärjestelmän tarvitse tehdä muistin jakamisen päätöksiä.

Aloitan aiheen tarkastelun tutustumalla ensin käyttöjärjestelmiin yleisesti luvussa 2. Käyttöjärjestelmällä on tietokonejärjestelmässä monta tehtävää, joista erityisesti kiinnitän huomiota resurssienhallintaan. Seuraavassa luvussa, luvussa 3, otan tarkasteltavaksi muistinhallinnan käyttöjärjestelmän resurssienhallinnan erityisosana. Käyn aluksi läpi muistinhallintaa yleisesti, virtuaalimuistia ja joitain perinteisiä sivunkorvausalgoritmeja. Luvun lopussa käyn läpi muistinhallintaa kahdessa käyttöjärjestelmässä: Mach ja NetBSD.

Luvussa 4 käyn läpi kansantaloustieteen perusteita kiinnittäen erityisesti huomiota markkinapohjaisten resurssienhallintamallien kannalta tärkeisiin osa-alueisiin: hinnanmuodostukseen ja resurssien jakautumiseen taloudellisissa järjestelmissä.

Luvussa 5, aikaisempien lukujen aiheet sitoutuvat yhteen muodostaen kuvan markkinapohjaisista malleista resurssienhallinnassa tietokonejärjestelmissä kiinnittäen erityistä huomiota käyttöjärjestelmien muistinhallintaan. Luvun lopuksi esittelen kaksi erilaista toteutusta markkinapohjaisten mallien käytöstä käyttöjärjestelmän muistinhallinnassa. Ensimmäinen on Kieran Hartyn ja David Cheritonin Stanfordin yliopistossa tekemä markkinapohjainen muistinhallintajärjestelmä, ja toinen on Utahin yliopistossa tehty muokkaus TENEX-käyttöjärjestelmään, jossa prosessoriaikaa ja keskusmuistia hallitaan markkinalähtöisesti.

Tutkimukseni teoriaosan jälkeen, luvussa 6, esittelen Hartyn ja Cheritonin mallin pohjalta itse suunnittelemani ja toteuttamani markkinapohjaisen muistinhallintajärjestelmän tyngän. Luvussa 7, tutkielmani lopuksi esitän joitain kysymyksiä, mitä asiaa käsitellessä mieleeni on noussut, sekä joitain asioita, jotka sopisivat mielestäni jatkotutkimuksien suuntaviivoiksi.

2 Käyttöjärjestelmistä

Käyttöjärjestelmä on ohjelma, joka toimii välittäjänä tietokoneen käyttäjän ja käyttäjän ohjelmistojen, ja tietokoneen laitteiston välillä. Käyttöjärjestelmän tarkoituksena on luoda ympäristö, missä käyttäjä voi suorittaa tarvitsemiaan ohjelmia. Käyttöjärjestelmän ensisijaisena tavoitteena on tehdä tietokoneesta kätevä käyttää ja toissijaisena tavoitteena on hyödyntää tietokoneen laitteistoa mahdollisimman tehokkaasti.[17]

Käyttöjärjestelmä voidaan jakaa pienempiin osiin joista jokaisella on oma hyvin määritelty tehtävä ja sen mukainen rajapinta:[17]

- Prosessienhallinnan tehtävänä on prosessien luominen, poistaminen, keskeyttäminen ja jatkaminen. Prosessienhallinta tarjoaa prosesseille myös mekanismeja prosessien keskinäiseen synkronointiin ja niiden väliseen kommunikointiin. Lisäksi prosessienhallinta jakaa prosessoriresurssit prosessien kesken.

- Muistinhallinta on vastuussa muistin osoittamisesta ohjelmille ja käyttäjille niiden tarpeiden mukaan, niin että muisti tulee käytettyä mahdollisimman tehokkaasti. Lisäksi muistinhallinta on vastuussa muistin käytön kirjanpidosta, eli siitä mitkä osat muistista ovat käytössä ja kenellä.

- Toissijaisen muistin hallinta pitää kirjaa vapaasta tilasta toissijaisessa muistissa, osoittaa sitä sitä tarvitseville, ja ajastaa kirjoitukset toissijaiseen muistiin ja lukemiset toissijaisesta muistista.

- Tiedostojärjestelmän vastuulla on tiedostojen ja hakemistojen luonti ja poistaminen sekä erilaisten tiedostojen ja hakemistojen käsittelytoimintojen tukeminen. Lisäksi tiedostojärjestelmän on kuvattava tiedostot toissijaiseen muistiin ja huolehdittava niiden tallentumisesta sinne.

- I/O-laitteiden hallintajärjestelmä muodostuu muistinhallintaosasta, joka hoitaa laitteen puskuroinnin, välimuistin ja sivuajon; yleisestä laiteajurirajapinnasta; sekä ajureista yksittäisille laitteille.

2.1 Resurssienhallintatehtävä

Yksi käyttöjärjestelmän tärkeimmistä tehtävistä on resurssien hallinta. Tietokoneet koostuvat useista resursseista kuten prosessoreista, muisteista, ajastimista, levyistä,

verkkoliittymistä, näytöistä, äänikorteista ja monista muista laitteista, jotka vaihtelevat järjestelmästä toiseen. Käyttöjärjestelmän tehtävänä on jakaa näitä lukuisia eri resursseja niitä tarvitseville prosesseille. Useiden käyttäjien käyttäessä tietokonetta samanaikaisesti resurssien hallinan tarve vain kasvaa. Yhtenä käyttöjärjestelmän resurssienhallinnan tehtävistä onkin pitää kirjaa kuka käyttää mitä resurssia, osoittaa resursseja prosessien käyttöön, pitää kirjaa resurssien käytöstä ja sovittaa yhteen sovellusten ja käyttäjien keskenään ristiriitaisia vaatimuksia.[19]

Resurssien hallinnalla tarkoitetaan käyttöjärjestelmän hallinnassa olevien rajallisten resurssien jakamista sovelluksille käyttäjän määräämän toimintalinjan mukaan. Käyttöjärjestelmällä on resurssien hallinnan suhteen kaksi tavoitetta: resurssien jaon globaali, järjestelmänlaajuinen optimointi, ja lokaali, sovelluskohtainen optimointi.[18]

Globaali optimointi tarkoittaa resurssien jakamista tehokkaasti käyttäjien hyödyn maksimoimiseksi. Käyttäjän hyödyn ajatellaan olevan kaikkien sen sovellusten hyötyjen summa. Usean käyttäjän järjestelmissä globaali optimi on kaikkien käyttäjien hyötyjen summa.[18]

Lokaali optimointi on sovelluskohtaista ja pyrkii osoittamaan sovellukselle resursseja niin että se pystyy täyttämään käyttäjän vaatimukset mahdollisimman hyvin.[18]

Kun tietokoneessa on samanaikaisesti useita käyttäjiä tai prosesseja, on niille kaikille osoitettava tarvittavat resurssit. Eri resursseja varten käyttöjärjestelmällä on erilaiset hallintamenetelmät. Joitain resursseja varten, kuten prosessoriaikaa, muistia ja tiedostojärjestelmää, käyttöjärjestelmä soveltaa kyseistä resurssia varten erikoistuneita hallintamenetelmiään, kun taas jotkut, kuten I/O-laitteet, käyttävät yleensä yleisempää pyyntö–vapautus -menetelmää.

Resurssien käytöstä pidetään usein myös kirjaa tilastointia, tutkimusta tai myöhempää tarvetta varten. Joskus resurssien käytöstä on pidettävä kirjaa myös laskutusta varten, jos tietokonejärjestelmä on yleisessä käytössä, ja käyttäjät joutuvat maksamaan käyttämistään resursseista. Kirjapidosta voi olla myös apua järjestelmän kehittämisessä.

Useiden käyttäjien suorittaessa useita sovelluksiaan samanaikaisesti eivät sovellukset saisi häiritä muiden sovellusten suoritusta. Yksi tärkeä osa resurssien hallintaa onkin resurssien ja niiden käytön suojaaminen. Resurssien suojaaminen ei ole tärkeää ainoastaan muilta sovelluksilta ja käyttäjiltä, vaan myös tietokoneen ulkopuolisilta häirintäyrityksiltä.

3 Muistinhallinta

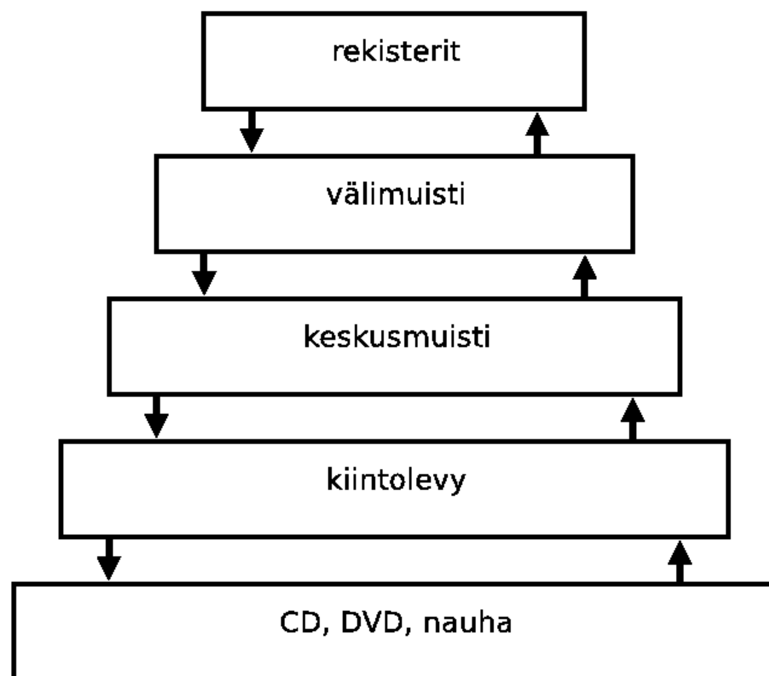
Muisti on yksi tietokoneen tärkeimmistä resursseista, ja sen hallintaan on kiinnitettävä erityistä huomiota. Muistinhallintajärjestelmän tehtävänä on pitää kirjaa mitkä osat muistista ovat sovellusten käytössä ja mitkä eivät, osoittaa sovelluksille muistia kun ne tarvitsevat sitä ja vapauttaa muisti muuhun käyttöön kun sovellukset eivät sitä enää tarvitse. Muistinhallinan tehtävänä on myös hoitaa muistin osien siirtäminen keskusmuistin ja toissijaisen muistin välillä tarvittaessa.

Keskusmuisti on ainoa suuri muisti, johon prosessori voi viitata suoraan. Keskusmuisti on suuri taulu tavuja, joihin kuhunkin voidaan viitata suoraan osoitteella. Keskusmuistia käytetään yksinkertaisilla luku- ja kirjoitus-käskyillä, jotka osoitetaan yksittäisiin osoitteisiin. Luku-käsky lukee tavun muistiosoitteesta prosessorin sisäiseen rekisteriin, ja kirjoitus-käsky puolestaan siirtää rekisterin sisällön haluttuun muistiosoitteeseen. Lisäksi prosessori lukee automaattisesti muistista käskyjä suoritettavaksi.[17]

Koska keskusmuisti on usein liian pieni säilyttämään kaikkea tietokonejärjestelmässä tarvittavia ohjelmia ja tietoa, ja koska sen sisältö tyhjenee kun tietokoneesta sammuu virta, on tietokoneessa oltava myös toissijainen muisti tiedon säilyttämistä varten. Suurin osa ohjelmista ja niiden käyttämästä tiedosta säilytetään toissijaisessa muistissa kunnes ne luetaan keskusmuistiin suoritusta varten. Suuri osa ohjelmista myös käyttää toissijaista muistia käyttämänsä tiedon lähteenä ja tulosten tallentamiseen.

Tietokoneen useat erilaiset muistit voidaan järjestää hierarkiaan nopeuden ja hinnan mukaan: korkeammalla hierarkiassa ovat nopeat, mutta kalliit muistit, ja alempana taas hitaat mutta halvemmat muistit. Hinnasta johtuen ylempien tasojen muistit ovat yleensä pienempiä kuin alempien. Muistin käyttötarkoitukseen ja asemaan hierarkiassa vaikuttaa hinnan ja nopeuden lisäksi myös muistin haihtuvuus. Tietyn tyyppiset muistit eivät säilytä sisältöään järjestelmän sammuttua, ja toisen tyyppiset taas säilyttävät. Käyttökertojen välillä ja sähkökatkosten varalta tiedot on tallennettava pysyvään muistiin, jotta ne eivät katoa.

Yleensä tietoa säilytetään ohjelmien suorituksen ajan tietokoneen ensisijaisessa muistissa, josta sitä kopioidaan tarvittaessa järjestelmän nopeisiin välimuisteihin käytön nopeuttamiseksi. Kun jotain tiettyä tietoa tarvitaan, tarkistetaan ensin, löytyykö se välimuistista, josta se saadaan nopeammin käyttöön. Jos tieto ei ole välimuistissa, se luetaan ensisijaisesta muistista ja kopioidaan samalla välimuistiin siltä varalta että sitä tarvitaan pian uudestaan.



Kuva 3.1: Muistihierarkia

Koska välimuistin koko on rajoitettu, ja sen toiminta keskeinen tekijä tietokonejärjestelmän suorituskyvyn kannalta, on välimuistin hallinta keskeinen tietokonejärjestelmien suunnitteluongelma. Hyvällä suunnittelulla voidaan iso osa hauista saada välimuistista, mikä nopeuttaa tiedon hakuja huomattavasti.[16]

Toisaalta ensisijainen muisti voidaan myös nähdä nopeana välimuistina toissijaiselle muistille, koska tieto täytyy siirtää toissijaiselta muistilta ensisijaiselle ennen kun sitä voidaan käyttää, ja sen on oltava ensisijaisessa muistissa ennen kun se voidaan kirjoittaa toissijaiseen muistiin.

Myös tietokonejärjestelmän eri oheislaitteissa voi olla omia välimuisteja. Esimerkiksi kiintolevyllä voi olla oma laitteiston itsensä ohjaama välimuistinsa, jonka tarkoitus on nopeuttaa tiedonsiirtoa kiintolevyn ja ensisijaisen muistin välillä.

Tiedon siirto eri tasojen muistien välillä voi tapahtua joko implisiittisesti tai eksplisiittisesti. Ensisijaisen muistin ja prosessorin välimuistin välillä siirrot tapahtuvat implisiittisesti täysin laitteiston ohjaamana. Ensisijaisen muistin ja toissijaisen muistin välillä tiedon siirrosta vastaa käyttöjärjestelmä. Siirrot tapahtuvat siis eksplisiittisesti käyttöjärjestelmän näkökulmasta, mutta implisiittisesti sovellusten ja käyttäjän näkökulmasta. Toissijaisen muistin ja erilaisten lisälaitteiden kuten CD- tai DVD-asemien välillä tiedon siirto tapahtuu yleensä täysin eksplisiittisesti käyttäjän ohjaamana.

Sama tieto voi usein sijaita usealla eri tasolla muistihierarkiassa samanaikaisesti. Kun tieto on kopioitu alemmalta tasolta korkeammalla, se sijaitsee molemmilla tasoil-

la samaan aikaan. Jos silloin tietoon tehdään muutoksia korkeammalla tasolla, eivät muutokset ole välittömästi näkyvissä alemmalla tasolla, vaan ne täytyy erikseen kopioida alaspäin. Ennen kopiointia alemmalle tasolle, eri tasojen tiedot eivät vastaa toisiaan.

Järjestelmässä, jossa on vain yksi prosessi kerrallaan, eivät eri tasojen eriävät versiot ole ongelma, koska aina käytetään kuitenkin korkeamman tason versiota. Jos taas prosesseja on useita saman aikaisesti, täytyy järjestelmän varmistaa, että kaikki prosessit saavat aina käyttöönsä uusimman version haetusta tiedosta. Tilanne muuttuu vielä monimutkaisemmaksi, jos järjestelmässä on useita prosessoreita, joilla jokaisella on oma prosessorikohtainen välimuistinsa. Koska prosessorit suorittavat samanaikaisesti, on järjestelmän varmistettava, että muutokset tietoon yhden prosessorin välimuistissa heijastuvat välittömästi myös muiden prosessorien välimuisteihin.

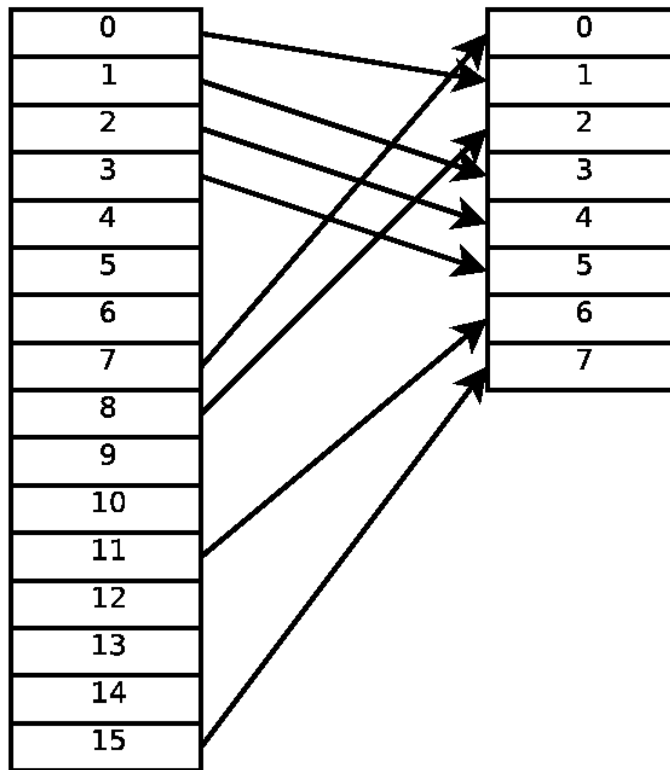
Muistinhallintaan on useita erilaisia menetelmiä, jotka soveltuvat erilaisiin käyttötarkoituksiin ja erilaisille laitteistoille. Useat muistinhallintamenetelmät vaativatkin jonkinlaista laitteistotason tukea. Useat tietokonejärjestelmät käyttävät jonkilaista virtuaalimuistitekniikkaa muistinhallinnan helpottamiseksi. Seuraavaksi kerron hieman virtuaalimuistista ja siihen keskeisesti kuuluvasta sivunkorvausongelmasta, johon markkinapohjaiset mallit mielestäni tarjoavat ratkaisun.

3.1 Virtuaalimuisti

Virtuaalimuistin tarkoituksena on erottaa ohjelmien käytössä oleva muistiavaruus fyysisestä muistista luomalla ohjelmien käyttöön erillinen virtuaalimuistiavaruus, jonka osat järjestelmä kuvaa tarpeen mukaan fyysiseen muistiin. Virtuaalimuistin avulla on tietokoneessa mahdollista ajaa ohjelmia, jotka eivät mahdu kerralla kokonaan fyysiseen muistiin, tai ajaa saman aikaisesti useita ohjelmia, jotka eivät mahdu kaikki samanaikaisesti tietokoneen fyysiseen muistiin.[16]

Virtuaalimuistin perusajatuksena on erottaa ohjelmien käyttämät muistiosoitteet fyysisisen muistin osoitteista. Ohjelmien käyttämät muistiosoitteet ovat virtuaaliosoitteita, joista muodostuu ohjelman käytössä oleva virtuaaliosoiteavaruus. Tietokone ei käytä virtuaaliosoitteita suoraan, vaan koneen MMU (memory management unit) kuvaa virtuaaliosoitteet fyysisiksi osoitteiksi, joita voidaan käyttää fyysiseen muistiin viittaamiseksi.

Virtuaaliosoiteavaruus on jaettu tasakokoisiin osiin, joita kutsutaan sivuiksi. Sivuja vastaavat osat fyysisessä muistissa ovat puolestaan nimeltään sivukehikoita. Sivut ja sivukehikot ovat aina samankokoisia. Muistin osien siirtäminen keskusmuistin ja toissijaisen muistin välillä tapahtuu aina sivuina.



Kuva 3.2: Virtuaalimuistin sivujen kuvaus fyysisen muistin kehikoihin

Kun prosessi viittaa sivuun, jota ei ole fyysisessä muistissa, se saa aikaan sivuvirheen. Sivuvirheen sattuessa käyttöjärjestelmä valitsee fyysisestä muistista vähän käytetyn sivun ja kirjoittaa sen sisällön takaisin toissijaiseen muistiin ja hakee tilalle viitatun sivun toissijaisesta muistista. Käyttöjärjestelmän vaihdettua virtuaaliosoitekuvauksen voidaan ohjelman suoritusta jatkaa sivuvirheen aiheuttaneesta kohdasta.

3.2 Sivunkorvaus

Sivuvirheen sattuessa käyttöjärjestelmän on valittava sivu poistettavaksi saadakseen tilaa uudelle sivulle. Lisäksi, jos poistettava sivu on muuttunut muistissa, se täytyy kirjoittaa takaisin toissijaiseen muistiin. Yleensä järkevintä on valita vähän käytetty sivu poistettavaksi, koska paljon käytettyä sivua tullaan todennäköisemmin tarvitsemaan pian uudestaan. Poistettavan sivun valinta ei kuitenkaan ole aivan yksinkertainen asia ja siihen onkin kehitetty lukuisia algoritmeja, joista esittelen seuraavaksi joitain Tanenbaumin oppikirjassa [19] kuvattuja.

Optimaalinen sivunkorvausalgoritmi valitsee aina korvattavaksi sivun, jota tullaan tarvitsemaan uudestaan pisimmän ajan kuluttua. Optimaalinen sivunkorvausal-

goritmi on kuitenkin mahdoton toteuttaa, koska käyttöjärjestelmä ei voi tietää, miten pitkän ajan kuluttua jotain tiettyä sivua tullaan tarvitsemaan uudestaan.

Useat tietokoneet keräävät laitteiston toimesta tietoa muistin käytöstä kahdella sivuihin liitetyllä bitillä, joista toinen kertoo, onko sivuun viitattu, ja toinen, onko sitä muutettu. Näiden kahden laitteiston ylläpitämän bitin avulla voidaan toteuttaa helposti **NRU-sivunkorvausalgoritmi** (not recently used). NRU on houkutteleva algoritmi, koska se on helppo ymmärtää, tehokas toteuttaa ja suorituskyvyltään yleensä riittävä.

NRU-algoritmissa kaikki sivut alkavat aina viitattu- ja muokattu-bitit nollattuina. Sivujen viitattu-bitit asetetaan päälle sitä mukaa, kun sivuihin viitataan, ja muokattu-bitit vastaavasti aina, kun sivuja muokataan. Yleensä tietokonelaitteisto pystyy tekemään tämän automaattisesti. Tietyin väliajoin kaikkien sivujen viitattu-bitit nollataan, jotta voidaan erottaa ne sivut, joihin on viitattu lyhyen ajan sisällä.

Sivuvirheen sattuessa sivut jaetaan neljään luokkaan niiden viitattu- ja muokattu-bittien perusteella:

1. ei viitattu, ei muokattu,
2. ei viitattu, muokattu,
3. viitattu, ei muokattu ja
4. viitattu, muokattu.

Poistettavaksi valitaan satunnaisesti joku alimman ei-tyhjän luokan sivu.

FIFO-sivunkorvausalgoritmissa käyttöjärjestelmä pitää listaa muistissa olevista sivuista järjestettynä sivun iän mukaan. Sivuvirheen sattuessa vanhin valitaan poistettavaksi. FIFO-algoritmin ongelmana on, että se saattaa poistaa muistissa kauan olleen, mutta paljon käytetyn sivun, jota tullaan tarvitsemaan taas pian uudestaan.

Ratkaisuna voidaan lisäksi tutkia sivujen viitattu- ja muokattu-bitit. Valittaessa poistettavaa sivua etsitään ensin listasta vanhin luokan 1. sivu. Jos luokan 1 sivuja ei löydy siirrytään seuraavaan luokkaan ja niin edespäin, kunnes sopiva sivu löytyy. Toinen ratkaisu on niin sanottu **second chance -sivunkorvausalgoritmi**, jossa vanhimman sivun ollessa viitattu, sen viitattu-bitti nollataan ja sivu siirretään listan päähän kuin se olisi juuri tuotu muistiin.

LRU-algoritmi (least recently used) perustuu havaintoon, että lähimenneisyydessä paljon käytettyä sivua tullaan todennäköisesti käyttämään paljon myös lähitulevaisuudessa, ja vastaavasti lähimenneisyydessä käyttämättömänä ollut sivu säilynee käyttämättömänä lähitulevaisuudessakin. Sivuvirheen sattuessa LRU-algoritmi poistaa

sivun, joka on ollut käyttämättömänä pisimmän aikaa. Algoritmi on raskas toteuttaa ohjelmistossa sellaisenaan, koska sivut on säilytettävä järjestetyssä listassa, jota on päivitettävä jokaisen muistiviittauksen yhteydessä.

LRU-algoritmi voidaan kuitenkin toteuttaa sopivan laitteiston avulla. Jos laitteistossa on laskuri, jota kasvatetaan jokaisen käskyn yhteydessä, ja laskurin arvo tallennetaan sivuun jokaisen muistiviittauksen yhteydessä, voidaan poistettavaksi valita sivu, jonka laskurin arvo on pienin. Toinen vaihtoehto on käyttää n sivua kohden laitteiston ylläpitämää $n * n$ matriisia. Kun sivuun k viitataan, laitteisto asettaa matriisissa rivin k kaikki bitit päälle, ja sen jälkeen sarakkeen k kaikki bitit pois päältä. Matriisin rivit tulkitaan binääriluvuiksi, ja poistettavaksi voidaan valita sivu, jota vastaavan matriisin rivin binääriluku on pienin.

LRU-algoritmia voidaan myös simuloida ohjelmistossa, jos laitteistotukea ei ole. **NFU-algoritmi** (not frequently used) liittää jokaiseen sivuun laskurin, jota kasvatetaan joka kellokeskeytyksellä sivun viitattu-bitin arvolla. Sivuvirheen sattuessa poistetaan sivu, jonka laskurin arvo on pienin. NFU-algoritmin ongelmana on ettei NFU koskaan unohda mitään, vaan vanhat laskurin arvot säilyvät myös myöhemmille jaksoille, ja uusien sivujen laskurin arvot saattavat jäädä jatkuvasti pieniksi. Muokattu NFU nimeltään **aging** ratkaisee ongelman siirtämällä laskureita oikealle yhdellä bitillä ennen viitattu-bitin lisäämistä ja lisää viitattu-bitin arvon laskurin vasemmanpuolimmaiseen bittiin.

Luvun lopuksi kuvailen kahden käyttöjärjestelmän muistinhallintajärjestelmiä antaakseni jonkinlaisen kuvan kahden erilaisen käyttöjärjestelmän muistinhallintajärjestelmästä. Ensimmäinen järjestelmistä on mikroydinkäyttöjärjestelmä Mach ja toiseksi olen valinnut BSD-perheeseen kuuluvan NetBSD-käyttöjärjestelmän ja sen muistinhallintajärjestelmän UVM.

3.3 Mach

Mach[14, 15, 16] on Carnegie Mellon yliopistossa kehitetty mikroydinkäyttöjärjestelmä, jonka yhtenä tarkoituksena on auttaa tutkimaan laitteistotason ja ohjelmistotason muistiarkkitehtuureja ja auttaa suunnittelemaan muistinhallintajärjestelmä, joka olisi helposti siirrettävissä yhtälailla moniprosessorisille kuin yksiprosessorisille tietokonejärjestelmille.

Mach-järjestelmä rakentuu viiden perusabstraktion varaan:

1. Tehtävä (task) on suoritusympäristö, jossa voidaan suorittaa säikeitä. Tehtävä on resurssien osoituksen perusyksikkö. Tehtävä sisältää sivutetun virtuaaliosoi-

teavaruuden ja suojatun pääsyn järjestelmäresursseihin kuten prosessorit, portti kyvyt ja virtuaalimuisti. Osoiteavaruus koostuu järjestetystä kokoelmasta kuvauksia muistiobjekteihin. Lähin vastine UNIX-järjestelmän prosessille Mach-järjestelmässä olisi tehtävä, jossa on yksi säie.

2. Säie (thread) on prosessorin käytön perusyksikkö. Kaikilla säikeillä on jaettu pääsy tehtävän resursseihin.

3. Portti (port) on kommunikointikanava, ytimen suojaama viestijono. Porttien perusoperaatiot ovat lähetä ja vastaanota.

4. Viesti (message) on tyypitetty kokoelma dataobjekteja, joita käytetään säikeiden väliseen kommunikointiin. Viestit voivat olla minkä kokoisia hyvänsä ja voivat sisältää osoittimia ja tyypitettyjä kykyjä.

5. Muistiobjekti (memory object) on kokoelma dataa. Muistiobjekti on palvelimen tarjoama ja hallinnoima, mutta se voidaan kuvata tehtävän osoiteavaruuteen.

Tehtävän, säikeen tai muistiobjektin luominen palauttaa oikeudet porttiin, joka kuvaa luotua objektia ja jota kautta sitä voidaan manipuloida. Kaikki muut paitsi viesteihin kohdistuvat operaatiot toteutetaan lähettämällä viestejä portteihin. Viestinvälitys mahdollistaa objektien sijaita mielivaltaisesti verkossa.

Mach-mikroytimen muistinhallintajärjestelmän järjestelmäkutsurajapinta on täysin 4.3BSD yhteensopiva ja sen ominaisuuksiin kuuluu esimerkiksi tuki suurille, väljille virtuaalimuistiavaruuksille, copy-on-write virtuaaliset kopiointi-operaatiot, copy-on-write ja read/write muistin jakaminen tehtävien välillä, muistikuvatut tiedostot ja käyttäjän toimittamat taustasäilöobjektit ja sivuttajat. Järjestelmä tekee vain vähän oletuksia käytössä olevasta muistinhallintalaitteistosta. Tärkeimpänä vaatimuksena on laitteiston kyky käsitellä ja toipua sivuvirheistä.

Järjestelmässä virtuaalimuistin hallinta on integroitu viestinvälityskommunikointiin, joka sallii suurten datamäärien lähettämisen viesteissä muistikuvauksen tehokkuudella. Viestinvälityksessä käytetään tavallisesti copy-on-write muistin jakamista, jolloin voidaan lähettää jopa kokonaisia osoiteavaruuksia joutumatta välttämättä oikeasti kopioimaan mitään.

Jokaisella tehtävällä on suuri osoiteavaruus, joka koostuu kuvauksista muistialueiden ja muistiobjektien välillä. Osoiteavaruuden kokoa rajoittaa vain laitteiston osoittamiskyky.

Tehtävä voi muokata osoiteavaruuttaan usealla tavalla. Mukaanlukien:

- osoittaa alueen virtuaalimuistia,

- vapauttaa alueen virtuaalimuistia,
- asettaa virtuaalimuistialueen suojauksen,
- määrittää virtuaalimuistialueen perinnän ja
- luoda ja hallita muistiobjektia, joka voidaan kuvata toisen tehtävän osoiteavaruuteen.

Ainoa muistialueille asetettu rajoitus on, että niiden pitää osua järjestelmän sivurajoihin. Järjestelmän käyttämä sivukoko määrätään käynnistysvaiheessa.

Read/write -jaettua muistia voidaan luoda osoittamalla muistialueen ja asettamalla sen perintäominaisuuden sopivaksi. Lapsitehtävät jakavat muistin sen perintäominaisuuden mukaan. Perintäominaisuus voidaan asettaa joko 'shared', 'copy' tai 'none' sivukohtaisesti. Oletuksena perintä on copy. Shared-merkityt sivut jaetaan read/write, copy-merkityt copy-on-write, ja none-merkittyjä sivuja ei jaeta ollenkaan lapsitehtävien kanssa.

Muistin suojaus määrätään myös sivukohtaisesti. Jokaiselle joukolle sivuja on kaksi suojausarvoa: nykyinen ja maksimi. Nykyinen ohjaa todellista laitesuojausta, ja maksimi taas määrää maksimiarvon, jonka nykyinen voi saada. Maksimia ei voi nostaa, mutta sitä voi laskea. Jos maksimia lasketaan alle nykyisen, nykyistä lasketaan vastaavasti. Suojaukset toteutetaan luku-, kirjoitus- ja suoritusoikeuksien yhdistelminä. Suojausten täytäntöönpano riippuu kuitenkin laitteiston tuesta.

Muistinhallintajärjestelmän perustietorakenteita on neljä:

1. Muistissa pysyvä sivutaulu on taulu, jolla pidetään kirjaa laitteistoriippumattomista sivuista.
2. Osoitekuvaus on kahteen suuntaan linkitty lista kuvausmerkintöjä, joista jokainen kuvaa kuvauksen osoitealueelta muistiobjektin alueeseen.
3. Muistiobjekti on taustasäilön yksikkö, jota hallinnoi ydin tai käyttäjätehtävä.
4. Koneriippuvainen muistikuvaustietorakenne `pmap`, eli fyysinen osoitekartta.

Virtuaalimuistin toteutus on järjestelmässä jaettu koneriippuvaiseen ja koneriippumattomaan osaan. Koneriippuvainen osa toteuttaa vain laitteiston vaatimien kuvausrakenteiden käsittelyn, eikä sillä ole mitään tietoa koneriippumattomista tietorakenteista tai kuvauksista virtuaalimuistista fyysiseen muistiin.

Tieto fyysisistä sivuista (muuttunut ja viitattu -bitit) ylläpidetään fyysisen sivunumeron mukaan indeksoidussa taulussa sijaitsevissa merkinnöissä, joista jokainen voi

olla linkitty joko muistiobjektilistaan, muistiosoitusjonoon tai hajautustaulun alkioon. Tiettyyn objektiin liittyvät sivumerkinnät on linkitty yhteen muistiobjektilistaan muistin vapauttamisen ja virtuaalikopiointien nopeuttamiseksi. Jokaiseen objektiin voi liittyä useita sivuja, mutta jokainen sivu voi kuulua vain yhteen objektiin. Muistiosoitusjonoissa säilytetään vapaat ja takaisin otettavat osoitetut sivut sivutustaustaprosessia varten. Objektiin kuuluvan sivun etsimisen nopeuttamiseksi käytetään hajautustaulua, jonka avaimena käytetään muistiobjektia ja tavupoikkeamaa.

Ytimen täytyy pitää kirjaa omasta sekä jokaisen tehtävän virtuaaliosoiteavaruudesta. Tehtävän osoiteavaruuden osoitteet kuvataan muistiobjektien tavupoikkemiksi osoitekuvauksella. Osoitekuvaus on kahteen suuntaan linkitty, järjestetty lista osoitekuvausmerkintöjä, joista jokainen kuvaa jatkuvan alueen virtuaaliosoitteita jatkuvaan alueeseen muistiobjektissa. Jokaisessa osoitekuvausmerkinnässä on suojaus ja perintä tiedot sen määräämästä muistialueesta. Yhden merkinnän kuvaamalla alueella on oltava samat ominaisuudet.

Osoitekartta toteuttaa tehtävän osoiteavaruuden yleisimmät toiminnot, kuten sivuvirhehaut, kopiointi ja suojaus toiminnot osoiteväleille ja osoitevälien osoitukset ja vapautukset. Järjestetty, linkitty lista mahdollistaa operaatiot osoiteväleille nopeasti ja yksinkertaisesti eikä rankaise isoja, väljiä osoiteavaruuksia. Lisäksi virheiden hakujen nopeuttamiseksi järjestelmä voi säilyttää vihjeitä viimeisimmistä virheistä. Koska jokainen merkintä voi kuvata ison alueen virtuaaliosoitteita, osoitekartta on tyypillisesti pieni. Osoitekartan ei myöskään tarvitse pitää kirjaa taustasäilöstä, koska taustasäilö on toteutettu muistiobjekteissa.

Virtuaalimuistiobjekti on säilytyspaikka datalle, jolle voidaan suorittaa luku- ja kirjoitustoimintoja. Objektin sisältöön voidaan viitata tavukohtaisesti ja se muistuttaa monella tapaa UNIX-järjestelmän tiedostoa. Jokaiselle objektille pidetään viittauslaskuria, mikä mahdollistaa roskienkeruun. Lisäksi usein tarvittavista objekteista pidetään välimuistia niiden uudelleenkäytön helpottamiseksi viittausten poistuttua. Tarvittaessa sivuttaja voi myös erikseen pyytää objektia säilytettäväksi.

Jokaiseen objektiin liitetään sivuttajatehtävä, joka mahdollistaa sivuvirheiden ja sivutuspyyntöjen käsittelyn ytimen ulkopuolella. Esimerkiksi muistikuvattu tiedosto voidaan toteuttaa luomalla virtuaalimuistiobjekti, jonka sivuttajaksi asetetaan tiedostojärjestelmä, ja sivuvirheet muutetaan pyynnöiksi tiedostojärjestelmälle. Pääsyä sivuttajaan kuvataan portilla (`paging_object`-portti), johon ydin voi lähettää viestejä datan pyytämiseksi tai ilmoittamiseksi objektin välimuistin muuttumisesta.

Järjestelmä ylläpitää tilainformaatiota ja listaa ensisijaisessa muistissa olevista sivuista, joita hallitaan ytimessä ytimen sivutustaustaprosessin välityksellä. Sivuja, jotka eivät ole ensisijaisessa muistissa, ylläpitää niiden sivuttaja, joka voi olla joko ytimessä

tai käyttäjätilan tehtävä.

Ydin tarjoaa joitain yksinkertaisia sivutuspalveluita muistille, jolla ei ole sivuttajaa. Muisti, jolla ei ole sivuttajaa, on automaattisesti nollatäytetty, ja sivutus tehdään inode-sivuttajalle, joka käyttää taustasäilönä tiedostojärjestelmää.

Copy-on-write -jakoa suoritettaessa, molemmat osoitekuvaukset osoittavat samaan muistiobjektiin. Jos molemmat tehtävät haluavat vain lukea dataa, ei ole tarpeen tehdä muita kuvauksia, mutta jos toinen tehtävistä kirjoittaa dataan, täytyy kirjoittavalle tehtävälle osoittaa uusi sivu muutoksia varten. Muutoksia varten luodaan erillisiä varjo-objekteja, jotka keräävät muutoksissa syntyneitä sivuja. Aluksi varjo-objekti on tyhjä ja ilman sivuttajaa, ja siitä on osoitin "varjostettavaan" objektiin.

Varjo-objekti ei kuitenkaan sisällä kaikkia määrittämänsä alueen sivuja, vaan muuttumattomia sivuja varten käytetään alkuperäistä objektia. Sivua etsittäessä seurataan varjo-objektien ketjua, kunnes etsitty sivu löydetään. Varjo-objekti voi siis varjostaa myös toista varjo-objektia.

Muistiobjektit eivät sovellu read/write -tyyppiseen muistin jakamiseen. Jaettuihin muistialueisiin kohdistuvat toiminnot saattavat vaatia useiden muistiobjektien kuvaamista tai uudelleenkuvaamista. Read/write -jakamiseen tarvitaankin kuvauksen kaltainen rakenne, johon voidaan viitata muista osoitekuvauksista. Osoitekuvausmerkinnät voivat viitata muistiobjektien lisäksi jakokuvauksiin, jotka taas osoittavat jaettuihin objekteihin. Jakokuvauksia voidaan jakaa ja yhdistää. Toiminnot, jotka vaikuttavat kaikkiin datan jakaviin kuvauksiin, suoritetaan jakokuvaukseen.

Mach-käyttöjärjestelmän muistinhallinan tekee mielenkiintoiseksi mahdollisuus toteuttaa osa muistinhallinnasta käyttäjätilan tehtävissä. Käyttöjärjestelmässä toissijaisen muistin objekti on yleensä kuvattu käyttäjätilan tehtävän virtuaalimuistiavaruuteen, ja sivuvirheen sattuessa järjestelmä lähettää viestin objektin porttiin, ja objektin tehtävänä on käsitellä virhe itse. Koska objekti voi olla käyttäjätilan tehtävän hallinnoima, tehtävä voi itse hoitaa objektin muistin sivutuksen. Myös objektin poistuessa sen sisältämän tiedon sivutuksen hoitaa sama tehtävä. Tehtävä voi myös valita jonkun muun sivun poistettavaksi, mutta jos se ei vähennä muistissa olevien sivujensa määrää pyydettäessä, ytimen oletussivuttaja poistaa tehtävältä vaadittavan määrän sivuja.

3.4 UVM

UVM-virtuaalimuistijärjestelmä[2] on ollut käytössä NetBSD-käyttöjärjestelmässä versiosta 1.4 eteenpäin. UVM tuli korvaamaan aikaisemmin käytössä ollutta 4.4BSD virtuaalimuistijärjestelmää, joka perustui Mach-mikroydinkäyttöjärjestelmän virtuaalimuistijärjestelmään. UVM-järjestelmä on erityisesti parempi sovelluksille, jotka käyttävät

paljon muistikuvattuja tiedostoja ja copy-on-write -muistia.

UVM on täysin uusi virtuaalimuistijärjestelmä, joka ottaa parhaat puolet BSD-käyttöjärjestelmän virtuaalimuistijärjestelmästä ja korvaa sen huonot puolet paremmilla. UVM sisältää myös joitain uusia ominaisuuksia, joita BSD-järjestelmässä ei ole ollenkaan. BSD-järjestelmästä on säilytetty jako koneriippuvaiseen ja koneriippumattomaan tasoon sekä kuvausrakenteet. Virtuaalimuistiobjekti, virheiden käsittely ja sivuttaja ovat korvattu paremmilla. Lisäksi uutena ominaisuutena UVM-järjestelmässä ovat virtuaalimuistipohjaiset tiedonsiirtomekanismit.

Koneriippuvainen kerros UVM-järjestelmässä (ja samoin BSD-järjestelmässä) on nimeltään `pmap`. Kerros hoitaa prosessorin MMU:n ohjelmoinnin, eli virtuaalimuistikuvausten ja fyysisten muistisivujen lisäys-, poisto-, muokkaus-, ja kyselytoiminnot. Koneriippuvaisella kerroksella ei ole mitään tietoa korkeamman, koneriippumattoman tason abstraktioista. Koska `pmap`-kerros on koneriippuvainen, on jokaiselle laitteistoarkkitehtuurille oltava oma, arkkitehtuurin erityispiirteet huomioon ottava `pmap`-kerros. UVM käyttää samaa `pmap`-kerrosta, joka on käytössä BSD- ja Mach-järjestelmissä.

Koneriippumaton kerros toteuttaa korkeamman tason muistinhallintatehtävät: prosessien tiedostokuvausten hallinan, tiedon hakemisen toissijaisesta muistista, muistin sivutuksen, fyysisen muistin allokoinnin ja copy-on-write -muistin käsittelyn. Koneriippumaton kerros voi olla sama kaikilla arkkitehtuureilla.

Koneriippumaton kerros määrittää viisi pääabstraktiota käytettäväksi prosessien muistinhallinnassa:

1. Virtuaalimuistiavaruus, `vmspace`, joka kuvaa prosessien virtuaalimuistiavaruuden koneriippuvaiset ja -riippumattomat osat. Virtuaalimuistiavaruus sisältää osoittimet prosessin `pmap` ja muistikuvaus -rakenteisiin sekä tilastoja prosessin muistin käytöstä.

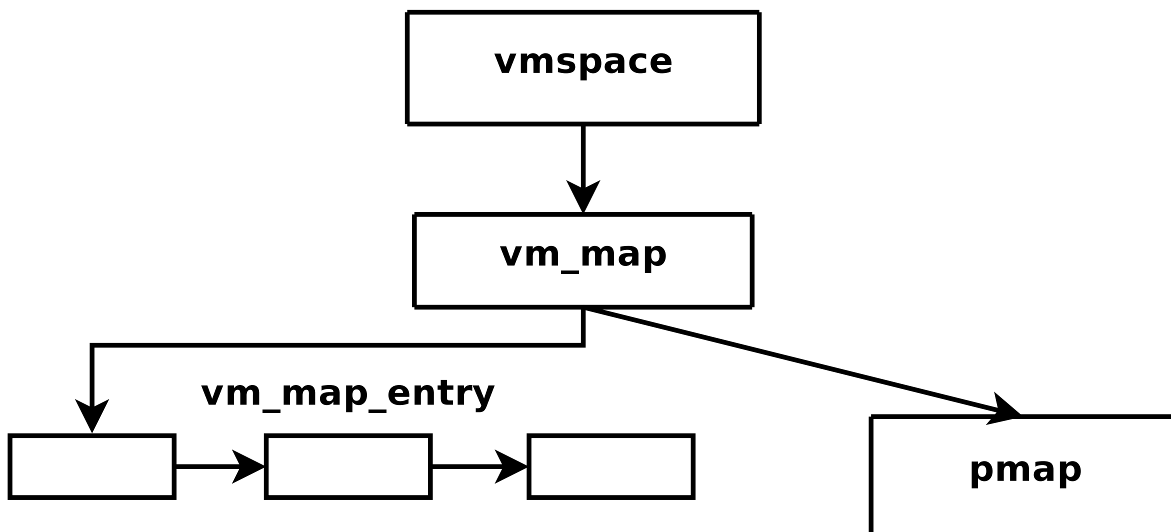
2. Muistikuvaus, `vm_map`, joka kuvaa prosessin virtuaalimuistiavaruuden koneriippumattoman osan. Muistikuvaus kuvaa muistiobjektit virtuaalimuistin alueisiin. Rakenne sisältää järjestetyn, kahteen suuntaan linkityn listan kuvausmerkintöjä, joista jokainen sisältää rekisterin kuvauksesta prosessin virtuaalimuistiavaruudessa. Rekisteri sisältää osoittimen kuvattuun objektiin, alku- ja loppuvirtuaaliosoitteet, sekä kuvauksen ominaisuudet (kuvauksen suojaukset, käyttömallin ja kiinnitysten lukumäärän). Käyttöjärjestelmän ytimellä ja jokaisella prosessilla on omat `vm_map` rakenteet.

3. Muistiobjekti, `uvm_object`, joka kuvaa tiedoston, nollatäytetyn muistialueen tai laitteen, jonka voi kuvata virtuaalimuistiavaruuteen. Muistiobjekti sisältää listan

sivuista, jotka sisältävät objektin dataa. Muistiobjekti-rakenne koostuu `vm_amap` tai `uvm_object` rakenteesta tai molemmista.

4. Sivuttaja, `uvm_pagerops`, joka kuvaa miten toissijaista muistia käsitellään. Jokaisella muistiobjektilla on sivuttaja, joka osoittaa funktioihin, joilla objektin sivuja siirretään ensisijaisen ja toissijaisen muistin välillä.

5. Muistisivu, `vm_page`, joka kuvaa yhtä sivua fyysistä muistia. Järjestelmän käynnistyessä jokaista fyysisen muistin sivua kohtaan luodaan `vm_page`-rakenne, joita virtuaalimuistijärjestelmä sen jälkeen käyttää muistinhallinnassaan.



Kuva 3.3: UVM-muistinhallintajärjestelmän eri abstraktioiden keskinäiset suhteet

Kun prosessi yrittää käyttää muistialuetta, jota ei ole kuvattu virtuaalimuistiavaruuteen, muodostetaan sivuvirhe. Sivuvirheen sattuessa prosessin `vm_map`-rakenteesta haetaan merkintä, joka vastaa virheen aiheuttanutta osoitetta (jos vastaavaa merkintää ei ole, annetaan virhesignaali). Halutun merkinnän löydyttyä selvitetään, onko haluttu data jo jossain sivussa, ja jos on, se kuvataan virtuaalimuistiavaruuteen. Jos taas halutun merkinnän dataa ei ole valmiina missään sivussa, objektin sivuttajaa pyydetään tuomaan haluttu data toissijaisesta muistista.

Yksi muistiobjekti voidaan kuvata useaan eri kohtaan muistiavaruudessa. Tällaista tarvitaan tilanteissa, joissa eri kuvauksilla on erilaiset ominaisuudet. Esimerkiksi, jos eri prosesseilla on erilaiset oikeudet samaan objektiin.

Kun yhden kuvausmerkinnän kuvaama virtuaalimuistin alue jaetaan useaan vierekkäiseen osaan, joista kullakin on oma kuvausmerkintä, syntyy kuvausmerkintöjen

pilkkoutumista. Tällaista tapahtuu, kun muutoksia tehdään vain osaan merkinnän kuvaamasta virtuaalimuistista: koska kaikilla yhden merkinnän sivuilla on oltava samat ominaisuudet, osan ominaisuuksien muuttuessa sivut on jaettava useammalle merkinnälle. Pilkkoutumisesta on haittaa järjestelmän suorituskyvylle, koska mitä enemmän merkintöjä yhdessä muistikuvassa on, sitä kauemmin siihen kohdistuvat toiminnot kestävät. Lisäksi merkintöjen pilkkomiseksi on luotava ja alustettava uusia merkintöjä, ja saatava uusia viittauksia taustaobjekteihin. Tämä on ongelma, koska ytimen käytössä olevien merkintöjen määrä on kiinnitetty. UVM-järjestelmässä ei pilkkoutuneita merkintöjä yritetä kuitenkaan koota yhteen, vaikka ominaisuudet muuttuisivatkin samoiksi.

Kiinnitetyllä muistilla tarkoitetaan virtuaalimuistia, jonka on pysyttävä fyysisessä muistissa, eikä sitä siksi voida sivuttaa toissijaiseen muistiin. Kiinnitettyä muistia käytettäessä syntyy paljon pilkkoutumista.

BSD-järjestelmässä on viisi eri tilannetta, joissa muistin on oltava kiinnitettyä. Neljässä näistä tilanteista UVM välttää pilkkoutumisen kokonaan:

1. Ytimen koodi, data ja tietorakenteet ovat aina kiinnitettyjä, joten sitä ei tarvitse erikseen huomioida `vm_map`-rakenteessa.

2. Jokaisella prosessilla on `user`-rakenne, joka sisältää sen ydin-pinon, signaalitiedot ja prosessikontrollialueen. Prosessin `user`-rakenteen on oltava kiinnitetty niin kauan, kun prosessi on ajettavassa tilassa. Tieto siitä, onko prosessin `user`-rakenne kiinnitetty, tallennetaan UVM-järjestelmässä sen `proc`-rakenteeseen, eikä sitä siis tarvitse tallentaa `vm_map`-rakenteeseen erikseen.

3. Ytimen tilan kyselyyn käytettävä `sysctl`-kutsu kiinnittää käyttäjän puskurin siksi aikaa, kun kutsun tulos kopioidaan sinne. UVM tallentaa tiedon siitä, onko puskuri kiinnitetty, ydin-pinoon eikä prosessin `vm_map`-rakenteeseen.

4. Tiedonsiirtoon laitteiden ja prosessin muistin välillä käytettävä `physio`-kutsu kiinnittää käyttäjän puskurin siirron ajaksi. UVM tallentaa tiedon siitä, onko puskuri kiinnitetty ydin-pinoon.

5. Ohjelmakoodin kriittisissä kohdissa muistin kiinnittämiseen käytettävä `mlock`-kutsu kiinnittää muistia haitallisten sivuvirheiden välttämiseksi. Tällaisissa tilanteissa tieto siitä, onko muisti kiinnitetty, on tallennettava prosessin `vm_map`-rakenteeseen.

Muistiobjektit on suunniteltu upotettaviksi laajempaan rakenteeseen, joka on tyypillisesti osa rakennetta, jota hallitaan virtuaalimuistijärjestelmän ulkopuolella. Näin

muistiobjekteja hallitaan yhdessä niiden sisältämän datan kanssa. Tekemällä muistiobjekteista tällä tavalla upotettavia, mistä tahansa ytimen abstraktiosta saadaan muistiin kuvattava. Upotettavuus aiheuttaa myös vähemmän ristiriitaa virtuaalimuistijärjestelmän ja muiden ytimen järjestelmien välille. Muistiobjektia käyttävästä rakenteesta huolehtivalle järjestelmälle annetaan koukku, jolla rakenteeseen liittyvä muistiobjekti voidaan tuhota, kun sitä ei enää tarvita.

Kun muistiobjektit upotetaan laajempiin rakenteisiin, voi virtuaalimuistijärjestelmä luottaa muiden järjestelmien ylläpitävän välimuistia viittaamattomista objekteista, eikä välimuistia tarvitse näin ollen toteuttaa virtuaalimuistijärjestelmässä. Näin viittaamattomille objekteille riittää yksi taso välimuistia, mikä taas vähentää ylimääräistä koodia. Tämä mahdollistaa lisäksi sivutusalgoritmien käytön rakenteille ja muistiobjekteille yhtäläisesti.

Muistia, joka vapautetaan heti kun siihen ei enää ole viittauksia, kutsutaan nimettömäksi muistiksi. Nimetön muisti sivutetaan toissijaiseen muistiin, jos muisti on vähissä. Nimetöntä muistia käytetään alustamattomalle datalle, pinoon, SysV jaettuun muistiin, ytimen muistin sivutettaviin alueisiin ja copy-on-write -kuvausten muuttuneisiin sivuihin.

Nimettömän muistin käsittelyssä käytetään SunOS-käyttöjärjestelmän virtuaalimuistijärjestelmän käyttämiä `anon` ja `amap` -abstraktioita: `anon` on tietorakenne, joka kuvaa yhtä sivua nimetöntä muistia, ja `amap` on tietorakenne, joka sisältää osoittimia `anon`-rakenteisiin, jotka ovat kuvattu yhteen virtuaalimuistissa. UVM kuitenkin eroaa SunOS-käyttöjärjestelmän virtuaalimuistijärjestelmän nimettömän muistin käsittelystä neljällä tavalla:

1. UVM-järjestelmässä on tuki muistin perinnälle ja `amap`-rakenteiden lykättyyn luontiin Mach-järjestelmän tapaan.

2. SunOS-käyttöjärjestelmässä nimettömän muistin hallinta ei ole näkyvissä muulle virtuaalimuistijärjestelmälle. UVM-järjestelmässä nimettömän muistin hallinta on kaikkien sivuttajien ja IPC- ja tiedonsiirto-järjestelmien käytössä.

3. SunOS-käyttöjärjestelmän sivutustietorakenne vaatii, että jokainen sivuttaja käsittelee omat virheensä. UVM-järjestelmässä on yleinen sivuvirhekäsittelijä, joka osaa käsitellä myös nimettömään muistiin liittyvät virheet.

4. UVM-järjestelmässä `amap`-rajapinta on erillään toteutuksesta, jotta toteutusta on helppo muuttaa tarvittaessa.

UVM-virtuaalimuistijärjestelmässä nimettömän muistin hallinta tapahtuu yksinkertaisen, kaksitasoisen kuvauksen avulla. Kuvaus koostuu ylemmästä `amap`-tasosta

ja alemmasta taustaobjekti-tasosta. Copy-on-write -merkinnässä on osoittimet `amap`-rakenteeseen ja alla olevaan objektiin, joista molemmat voivat myös puuttua. Jaetussa kuvauksessa `amap`-osoitin usein puuttuu, ja nollatäytetyssä kuvauksessa objekti-osoitin usein puuttuu. `anon`-rakenteessa on viittauslaskuri ja tieto datan sijainnista (ensisijainen tai toissijainen muisti). Nimetön muistisivu, johon on vain yksi viittaus, on kirjoitettava, ja sivut, joihin on useampia viittauksia, ovat copy-on-write. Copy-on-write -virheiden sattuessa sivun sisältämä data kopioidaan uuteen nimettömään sivuun, ja viittaus vanhaan sivuun poistetaan.

UVM-järjestelmässä on kolme uutta tiedonsiirtomekanismia, joita ei ollu vanhassa BSD-järjestelmässä:

1. Sivujen lainaaminen (page loanout) mahdollistaa prosessin turvallisesti jakaa copy-on-write -kopio muistisivusta muiden prosessien, tiedonsiirtojärjestelmän tai IPC-järjestelmän kanssa. Lainattava sivu voi olla kuvatusta tiedostosta, nimetöntä muistia tai niiden yhdistelmä. Sivuja voi lainata myös kiinnitettyihin sivuihin ytimen tiedonsiirtojärjestelmälle tai sivutettavana nimettömänä muistina muille prosesseille. Lainaaminen säilyttää copy-on-write -ominaisuuden sivuvirheiden, sivutuksen ja muistin huuhtomisen yhteydessä. Lainaaminen tarjoaa pääsyn muistiin sivutasolla aiheuttamatta pilkkoutumista tai häiritsemättä korkeamman tason tietorakenteita.

2. Sivujen siirto (page transfer) mahdollistaa muistin sisällyttämisen prosessin muistiavaruuteen tiedonsiirto- tai IPC-järjestelmistä tai muista prosesseista. Sivujen siirryttyä prosessille, niistä tulee nimetöntä muistia. Siirtomekanismi osaa myös käsitellä sivut, jotka on lainattu muualta. Jos siirtomekanismin annetaan itse valita minne sivut liitetään, sivujen siirto voidaan tehdä aiheuttamatta pilkkoutumista tai häiritsemättä korkeamman tason tietorakenteita.

3. Kuvausmerkintöjen siirto mahdollistaa prosessien ja ytimen vaihtaa isoja alueita virtuaalimuistiavaruuttaan virtuaalimuistijärjestelmän tietorakenteita käyttäen. Näin voidaan kopioida, siirtää tai jakaa mikä tahansa virtuaalimuistin alue. Myös copy-on-write alueiden jakaminen on mahdollista. Merkintöjen siirron sivukohtainen hinta on alhaisempi kuin lainauksen tai siirron, mutta se voi aiheuttaa pilkkoutumista, jos sitä käytetään pieniin määriin sivuja. Merkintöjen siirtoa ei kuitenkaan voida käyttää muistin jakamiseksi DMA:ta käyttävien ytimen järjestelmien kanssa.

NetBSD-käyttöjärjestelmän UVM-muistinhallintajärjestelmä toimii tässä esimerkkinä nykyaikaisesta versiosta perinteisen UNIX-tyyppisen käyttöjärjestelmän muistinhallinnasta. Muistinhallinta on keskitettyä, käyttöjärjestelmän ytimen hallitsemaa, eikä

käyttäjätilan prosesseilla ole juurikaan mahdollisuuksia saada siitä tietoa tai vaikuttaa sen tekemiin päätöksiin. UVM-on parantanut NetBSD-käyttöjärjestelmän muistinhallinan suoritustehoa verrattuna edelliseen BSD-muistinhallintaan, mutta uusi muistinhallintajärjestelmä ei kuitenkaan tuo mitään uutta globaalin ja lokaalin optimoinnin erillisyyden ongelmaan.

4 Kansantaloustiede

Kansantaloustiede tutkii ihmistalouksia ja sitä, miten niissä tuotetaan, jaetaan ja kulutetaan hyödykkeitä. Jokaisen talouden peruskysymykset ovat, mitä tuotetaan, kuinka tuotetaan ja kenelle tuotetaan. Käyn seuraavaksi läpi hieman kansantaloustieteen perusteita. Luku perustuu pääosin Pekkarisen ja Sutelan oppikirjaan Kansantaloustiede [13] sekä Beggin oppikirjaan Economics [1].

Tärkeimmät talousjärjestelmät ovat markkinatalous ja suunnitelmatalous. Markkinataloudessa talouden peruskysymykset, eli hyödykkeiden jakautuminen, ratkeavat markkinoiden tietä, kun päätöksiä tekevät omaa etuaan tavoittelevat talousyksiköt. Suunnitelmataloudessa puolestaan poliittiset päätöksentekijät hyväksyvät suunnitelman, jonka nojalla kysymykset ratkaistaan.

Tarpeita tyydytetään hyödykkeitä kuluttamalla. Koska hyödykkeitä on saatavilla vain rajallinen määrä, on niistä maksettava korvaus, hinta. Hinta on hyödykkeen niukkuuden osoitin, ja hyödykkeen hinta nousee siihen kohdistuvan tarpeen voimistuessa. Kuluttajan valintateorian mukaan kuluttaja tavoittelee mahdollisimman korkeata tarpeentyydytystä. Koska yksittäisen kuluttajan voimavarat tarpeiden tyydyttämiseksi ovat niukat, tuotannossa yhdistetään voimavaroja (eli niin sanottuja tuotannontekijöitä) hyödykkeiden aikaansaamiseksi.

Jos tuotannontekijöitä jää käyttämättä, tai niitä ei käytetä parhaalla mahdollisella tavalla, talous toimii tehottomasti. Tehokkaasti toimivassa taloudessa jonkin hyödykkeen tuotantoa on mahdollista lisätä vain vähentämällä jonkin toisen hyödykkeen tuotantoa. Voimavarojen tehokas käyttö tarkoittaa, että voimavarojen käytöstä saadaan mahdollisimman runsas tarpeentyydytys, tai tavoiteltu tarpeentyydytys saadaan aikaan mahdollisimman vähäisellä voimavarojen käytöllä.

Markkinoiden toimivuuteen vaikuttaa myös paljon vaihdannan vaatima aika, vaiva ja muut kustannukset, yhteiseltä nimeltään taloustoimikustannukset. Taloustoimikustannukset vaihtelevat paljon ja niihin kuuluu usein esimerkiksi kohteen määritteleminen, vaihdannan toisen osapuolen etsiminen ja kauppaehdoista sopiminen.

Erikoistumisesta, eli työnjaosta, on tuotannossa yleensä huomattavia etuja. Tuottajien luontaisista eroista seuraa, että jotkut pystyvät tekemään tietyt tehtävät tehokkaammin. Tiettyyn tuotannonalaan erikoistuessaan tuottajat voivat harjaantua siihen nimenomaiseen alaan ja pystyvät toimimaan tehokkaammin. Lisäksi keskittymisestä seuraava suurtuotanto tuo kustannusetuja. Tuotannon kasvaessa tuotteet voidaan val-

mistaa halvemmalla. Toisaalta jakaessaan tuotannon pieniin erillisiin osiin, työnjako tekee talouden monimutkaiseksi ja alttiiksi häiriöille. Lisäksi erikoistunut tuotanto joutuu helposti vaikeuksiin kysynnän heikentyessä.

Tyonjako edellyttää vaihdantaa. Ne hyödykkeet, joita ei itse tuoteta, saadaan vaihtamalla. Vaihto olisi periaatteessa mahdollista ilman rahaa, mutta taloustoimikustannukset olisivat silloin liian suuret. Rahan käyttö parantaa talouden tehokkuutta.

Kotitaloudet ovat kulutusyksiköitä, ja tuotanto keskittyy yleensä yrityksiin. Kotitaloudet myyvät tuotannontekijöitä yrityksille erityisten tuotannontekijämarkkinoiden kautta, ja saamillaan tuloilla kotitaloudet ostavat yritysten tuottamia hyödykkeitä hyödykemarkkinoilta. Kotitalouksien tuotannontekijät muodostavat kotitalouksilta yrityksiin kulkevan reaalivirran, jonka vastineeksi yrityksiltä virtaa kotitalouksiin tulovirta. Näiden virtojen kautta yritysten ja kotitalouksien välillä vallitsee jatkuva kiertokulku, ja tapahtumat yksillä markkinoilla vaikuttavat myös toisiin. Lisäksi kotitalouksien ja myös yritysten kesken tapahtuu paljon taloudellista toimintaa.

Kiertokulkumallia voidaan kehittää myös kattamaan muita talouden osia. Esimerkiksi pankit ottavat kotitalouksien säästöt ja välittävät ne käytettäväksi muualla, julkinen valta kerää veroja, jotka käytetään julkisten palveluiden ja tavaroiden tuottamiseen tai ostamiseen. Ulkomaankaupan kautta osa kierrosta siirtyy ulkomaille.

Mitä enemmän ja mitä niukempaa tuotannontekijää omistaa, sitä suuremman hinnan siitä saa tuotannontekijämarkkinoilla, ja sitä suuremmat tulot on. Yhden tuotannontekijän hinnan noustessa pyritään tuotantotekniikkaa muuttamaan sitä säästävään suuntaan.

Hyödykkeiden kysynnän muutos vaikuttaa niiden tuotannontekijöiden markkinoihin, joilla hyödykkeet valmistetaan. Tuotannontekijöiden kysyntä on johdettua, eli se on riippuvaista hyödkemarkkinoiden tilasta. Tuotannontekijämarkkinat eivät koskaan toimi vapaasti, niillä ei koskaan vallitse täydellinen kilpailu. Valtiovalta, ammattiliitot ja työnantajaliitot sopivat tärkeimmistä työehdoista, koska ne vaikuttavat merkittävästi kotitalouksien hyvinvointiin.

Taloudessa kuluttajat pyrkivät mahdollisimman suureen tarpeentyydytykseen, ja tuottajat puolestaan voittojen maksimointiin. Kuluttajien ja tuottajien itsenäisen, itsekkään toiminnan kautta toimiva hajautettu, omaa etua tavoitteleva päätöksenteko on markkinatalouden suuri voima, joka luo talouteen joustavuuden, uudistumiskyvyn ja luovuuden. Markkinamekanismi sovittaa miljoonien kotitalouksien ja yritysten toiminnot yhteen täysin automaattisesti kenenkään ohjaamatta. Se jopa johtaa kaikkien kannalta parhaaseen mahdolliseen lopputulokseen. Vaihtoon ryhdytään aina vapaaehtoisesti ja sitä jatketaan kunnes hyötymismahdollisuuksia ei enää ole.

Toisaalta osapuolten tehdessä kauppaa, molemmat ottavat yleensä huomioon vain

oman etunsa, eikä mahdollisia haitallisia ulkoisvaikutuksia (esimerkiksi luontoon tai yhteiskuntaan kohdistuvia ulkoisvaikutuksia) huomoida usein ollenkaan. Markkinoiden toimintaa haittaa myös se, että markkinoilla käytössä oleva informaatio on usein epätäydellistä ja osapuolten kesken epätasaisesti jakautunutta.

Toimivat markkinat vaativatkin moniulotteisen ympäristön, johon kuuluu rahataloudellinen vakaus, vakaa lainsäädäntö sekä monia kirjoitettuja ja kirjoittamattomia sääntöjä ja instituutioita. Erityisen tärkeätä markkinoiden kannalta on hyvin määritellyt ja riittävän vahvat omistusoikeudet. Omistusoikeuksiin kuuluu perinteisesti oikeus käyttää ja luovuttaa hyödyke ja ottaa itselle sen mahdollinen tuotto. Omistusoikeudetkaan eivät aina ole täydellisiä. Esimerkiksi jos kohteella on merkittäviä ulkoisvaikutuksia, valtio voi rajoittaa sen omistusoikeutta.

Markkinoita on useita eri tyyppejä. Jotkut markkinat ovat anonyymejä, joskus vaihto tapahtuu verkostossa, ja joskus taas alihankintasuhteessa. Oligopoli tarkoittaa että markkinoita hallitsee vain harvojen yritysten ryhmä. Monopoli puolestaan että markkinoilla on vain yksi tarjoaja.

Mielestäni tärkeimmät tekijät markkinapohjaisten mallien soveltamisessa tietokonejärjestelmien resurssienhallinnassa ovat resurssien hinnoittelu ja mahdollisuus saavuttaa tasapainotila, pareto-optimi järjestelmän resurssien jaossa. Hinnoittelu on keskeisessä asemassa, koska se määrää, miten järjestelmän toimijoilla on varaa hankkia tarvitsemiansa resursseja. Resurssien jakautuminen markkinoilla taas on keskeinen kysymys, koska markkinamekanismien käyttö on järkevää ainoastaan, jos ne pystyvät saavuttamaan vähintään pareto-optimaalisen resurssien jaon toimijoiden välille. Seuraavaksi tutustunkin tarkemmin hinnanmuodostukseen ja resurssien jakautumiseen markkinoilla.

4.1 Hinnanmuodostus

Koska markkinataloudessa suurin osa päätöksistä tehdään hintojen mukaan, on hintatason oltava vakaa tai muututtava ennustettavalla tavalla. Markkinat eivät voi toimia, elleivät hinnat sopeudu kysynnän ja tarjonnan mukaan. Kysyntä on sitä suurempi, mitä alhaisempi hinta on. Kysynnän noustessa taas markkinahinta nousee. Toisaalta taas hinnan kohotessa tarjottu määrä kasvaa.

Markkinat hoitavat erillisten toimintojen koordinoinnin muodostamalla hyödykkeille ja tuotannontekijöille hinnat, jotka ohjaavat talousyksiköiden päätöksiä. Markkinoiden muodostamilla hinnoilla on kolme keskeistä tehtävää:

- Hinnat välittävät tietoa kuluttajien tarpeiden muutoksesta yrityksille ja tuotan-

totekniikan muutoksista kotitalouksille.

- Hinnat toimivat kannustimina kotitalouksille säästää tai kuluttaa ja yrityksille puolestaan lisätä tai vähentää tuotantoa.
- Hinnat vaikuttavat myös tulonjakoon: mitä enemmän kotitalous omistaa tuotannontekijöitä, sitä suuremmat ovat sen tulot, ja mitä alhaisemmat ovat hyödykkeiden hinnat, sitä suurempi on tulojen ostovoima.

Täydellisen kilpailun markkinoilla markkinavoimat, hyödykkeen kysyntä ja tarjonta määräävät hinnan. Yksittäiset myyjät ja ostajat eivät tee lainkaan hinnoittelupäätöksiä eivätkä siis vaikuta hintaan. Täydellisessä kilpailussa kaikilla aloilla ansaitaan pitkällä aikavälillä yhtä suuria voittoja. Useimmilla markkinoilla on kuitenkin epätäydellinen kilpailu, koska ostajat ja myyjät tekevät hintapäätöksiä.

Täydelliselle kilpailulle ovat voimassa seuraavat ehdot:

- Ostajia ja myyjiä on riittävän monta, ja kunkin markkinaosuus on pieni.
- Kaikkien tuottama hyödykkeet ovat samanlaisia, ja yhdenlaiselle hyödykkeelle on vain yksi hinta.
- Kaikilla toimijoilla on täydellinen tietämys hyödykkeiden ominaisuuksista.
- Markkinoille on vapaa pääsy, ja markkinoilta voi poistua vapaasti.

Epätäydellisen kilpailun tärkeimpänä syynä on suurtuotannon edut. Suurilla yrityksillä on markkinoilla kilpailuetu, joka johtaa keskittymiseen. Joskus keskittyminen voi johtaa monopoliin. Suurtuotannon kustannuseduista syntyvää monopolia sanotaan luonnolliseksi monopoliksi. Suurtuotannosta seuraava taloudellinen tehokkuus puoltaa luonnollisten monopolien sallimista. Monopoli voi olla myös seurausta yrityksen patentin turvin hallitsemasta yksinoikeudesta. Useimmat monopolit ovat kuitenkin lakisääteisiä.

Täydellisen kilpailun oloissa yrityksellä ei ole lainkaan valtaa markkinoilla, eikä se voi päättää kuin omasta tuotannostaan. Kun markkinoilla on vain yksi tarjoaja, se voi määrätä yksin markkinahinnan, ja kuluttajat päättävät vain, kuinka paljon he haluavat sillä hinnalla ostaa. Täydellisen kilpailun yritys on hinnan ottaja, ja monopoli puolestaan hinnan asettaja.

Alenevan rajahyödyn olettamuksen mukaan yhden lisä hyödykeyksikön kuluttamisesta kuluttajalle seuraava hyöty — rajahyöty — vähenee. Koska kuluttajan rationaalinen valinta perustuu hinnan ja rajahyödyn vertaamiseen, kuluttaja haluaa maksaa hyödykeyksiköstä enintään hinnan, joka on hänen rahamääräisenä ilmaistun rajahyödyn

suuruinen. Toisaalta, jotta kuluttaja voisi tehdä valintansa rationaalisesti, on kuluttajan tunnettava kulutuksesta seuraava hyöty ja pystyttävä vertaamaan sitä hyödykkeen rahamääräiseen hintaan.

Yrityksen teorian mukaan yritysten oletetaan pyrkivän aina voittojensa maksimointiin. Yritykselle yhdestä myydystä lisäyksiköstä saatu tulonlisäys — rajatulo — on hinnan suuruinen, ja yhdestä tuotetusta lisäyksiköstä seuraava kustannusten lisäys — rajakustannus — on sen valmistuskustannusten suuruinen. Yritys saavuttaa tuotannossa tasapainon, kun tuotettavalla määrällä hinta ja rajakustannukset ovat yhtä suuret. Jos hinta ylittäisi rajakustannukset, tuotantoa kannattaisi laajentaa, ja jos taas rajakustannukset ovat hintaa korkeammat, tuotantoa kannattaa supistaa.

Rajatuottavuus kertoo, kuinka paljon yhden yksikön lisääminen tuotantoon lisää tuotosta, kun muiden tuotannontekijöiden määrä pysyy samana. Voittojen maksimoimiseksi tuotannontekijää on oltava niin paljon, että rajatuottavuus on yhtä paljon kuin tuotannontekijän hinta. Rajatuottavuus muodostaa tuotannontekijän kysynnän.

Kuluttajien ja tuottajien toiminta johtaa tasapainoon, jossa saavutettava rajaetu on yhtä suuri kuin rajahaitta, ja tällöin kokonaisetu on suurin mahdollinen. Vain tasapainossa markkinaosapuolten aikomukset ovat sopusoinnussa keskenään: kysytty määrä ja tarjottu määrä ovat yhtä suuret. Ostajat ovat halukkaita ostamaan tasapainohintaan täsmälleen saman määrän, kuin myyjät ovat halukkaita myymään.

Tasapaino saavutetaan hinnan sopeuttamisen kautta. Vallitsevan hinnan ollessa korkeampi kuin tasapainohinta vallitsee liikatarjonta, joka pakottaa hinnat laskemaan tasapainotasolle. Jos taas hinta on tasapainoa alhaisempi, vallitsee liikakysyntä, joka ajaa puolestaan hintoja ylöspäin kohti tasapainohintaa. Hinnan muutokset jatkuvat näin, kunnes tasapainohinta saavutetaan, ja kun tasapaino on saavutettu, se pyrkii säilymään.

Myös monopoli maksimoi voittonsa tuottamalla määrän, jolla rajakustannus on sama kuin rajatulo, mutta monopolille rajatulo on myyntihintaa pienempi. Monopolin on enemmän myydäkseen laskettava myyntihintaa, joten myynnin lisääminen laskee jokaisesta hyödykkeestä saatua hintaa. Monopolin markkinahinta on kuitenkin rajakustannuksia korkeampi. Monopolin makkinahinnan ja rajakustannusten ero muodostaa monopolivoiton.

Oligopolissa markkinoita ohjaa muutamat keskenään kilpailevat suuret yritykset, joiden lisäksi voi toimia suurikin määrä pieniä yrityksiä. Oligopoleillakin on merkittävästi markkinavaltaa, ja nekin voivat asettaa rajakustannukset ylittävät hinnat. Oligopoli on kuitenkin tavallinen ja vakaa markkinamuoto, vaikka oligopolin muodostavat yritykset eivät ole jatkuvasti samat vaan saattavat vaihtua aikojen saatossa. Oligopolitkin ovat seurausta suurtuotannon eduista, mutta eduilla on kuitenkin rajansa, ja se

jättää tilaa useammille tuottajille.

Niukkojen hyödykkeiden ollessa kyseessä, kysyntä on hinnasta riippuvainen. Näin hinta toimii keinona rajoittaa kysyntää tuotantomahdollisuuksien tasolle. Toisalta tarjonta on myös hinnasta riippuvainen. Kysynnän hintajousto kertoo, kuinka paljon hinnan muutos vaikuttaa hyödykkeen kysyntään. Kysynnän hintajousto on yleensä negatiivinen, mikä tarkoittaa, että hinnan noustessa kysyntä laskee. Jos hintajousto on itseisarvoltaan alle yhden, se tarkoittaa, että kysyntä muuttuu suhteessa vähemmän kuin hinta, eli hinnan noustessa kysyntä laskee suhteessa vähemmän, ja hyödykkeeseen käytetty rahamäärä kasvaa. Tällöin kysynnän sanotaan olevan joustamatonta. Jos taas hintajousto on itseisarvoltaan yli yhden, kysyntä muuttuu suhteessa enemmän kuin hinta, ja hinnan noustessa käytetty rahamäärä pienenee. Tällöin kysynnän taas sanotaan olevan joustavaa. Jos hintajousto on itseisarvoltaan tasan yksi, puhutaan yksikköjoustavasta kysynnästä, eikä hinnan muutos tällöin vaikuta hyödykkeeseen käytettyyn rahamäärään.

Kysynnän tulojousto taas kertoo, miten kysyntä muuttuu tulojen muuttuessa. Kysynnän tulojousto on normaalihyödykkeillä positiivinen ja inferiorisilla hyödykkeillä negatiivinen. Normaalihyödykkeet voidaan vielä jakaa ylellisyyshyödykkeisiin, joilla jousto on suuri, ja välttämättömyyshyödykkeisiin, joilla jousto taas on pieni.

Muita joustoja on esimerkiksi kysynnän ristijousto, joka kertoo, miten yhden hyödykkeen hinnan muutos vaikuttaa toisen kysyntään.

4.2 Resurssien jakautuminen

Beggin mukaan [1] Adam Smithin on sanonut, että itsekäs omien etujen tavoittelu ilman keskitettyä ohjausta pystyy luomaan yhtenäisen yhteiskunnan ja saamaan aikaan resurssien järkevän jakautumisen. Resurssien jakautumisella tarkoitetaan täydellistä kuvausta siitä, mitä kukakin yhteiskunnassa tekee, ja mitä kukakin saa. Yhteiskunnassa mahdolliset resurssien jaot riippuvat yhteiskunnan käytössä olevasta teknologiasta ja resursseista. Resurssien jaon lopullinen arvo riippuu siitä, miten kuluttajat eri asioita arvottavat.

Markkinoilla vaihtoa jatketaan vain niin kauan kun toinen hyötyy ilman että toisen hyöty vähenee. Näin saavutettua tilaa kutsutaan pareto-optimiksi. Täydellisen kilpailun markkinat johtavat aina pareto-optimaaliseen tilaan.

Resurssien jaon voidaan sanoa olevan pareto-optimaalinen, jos siitä on mahdoton siirtyä toisenlaiseen jakoon, jolla joillekin olisi osoitettuna enemmän resursseja, mutta kenellekään ei vähempää. Pareto-optimaalisuutta voidaan käyttää arvioitaessa resurssien jaon tehokkuutta. Jos talouden kaikki markkinat ovat vapaita markkinoita, niistä

seuraava tasapaino on pareto-optimaalinen.

Vapaasta kilpailusta seuraava tasapainotila on pareto-optimaalinen, koska tuottajien asettaessa itsenäisesti rajakustannukset samaksi kuin hinta, ja kuluttajien asettaessa rajahyödyn samaksi kuin hinta, varmistetaan että tuotteen rajakustannus vastaa sen rajahyötyä.

Markkinoiden epäonnistumisilla tarkoitetaan tilanteita, joissa vapaiden markkinoiden tasapainotila ei kykene saavuttamaan tehokasta resurssien jakautumista. Epäonnistumiset kuvaavat tilanteita, joissa markkinoiden vääristymät estävät 'näkymätöntä kättä' jakamasta resursseja tehokkaasti.

Mahdollisia vääristymiä voivat aiheuttaa:

- Epätäydellinen kilpailu. Täydellinen kilpailu saa yritykset asettamaan rajakustannukset vastaamaan kuluttajien rajahyötyä. Epätäydellisessä kilpailussa puolestaan tuottajat asettavat rajakustannuksen hintaa alhaisemmaksi, ja niin rajahyöty ylittää rajakustannukset. Koska tuotannon lisääminen lisäisi enemmän kuluttajien hyötyä kuin se lisäisi kustannuksia, tuotetaan vähemmän kuin olisi tehokasta.

- Sosiaaliset prioriteetit. Oikeudenmukaisuuden nimissä tehtävä verotus saa aikaan tehotonta resurssien jakoa tekemällä eroa sen välille, mitä kuluttaja tuotteesta maksaa ja mitä kuluttaja siitä saa.

- Ulkoisvaikutukset. Ulkoisvaikutukset kuten saaste, melu tai ruuhkat ovat taloudellisen toiminnan ulkopuolisiin kohdistuvia tarkoittamattomia seurauksia. Ukoisvaikutukset ovat ongelma, koska niille ei ole toimivia markkinoita eikä markkinahintaa, joten markkinat eivät voi varmistaa, että niiden rajahyöty vastaisi ulkopuolisille aiheutunutta rajakustannusta.

- Muut puuttuvat markkinat. Futuureille, riskeille ja informaatiolle ei ole markkinoita tai ne ovat puutteellisia.

Kuten näemme, vapaassa kilpailussa saavutetaan ihmistalouksissa aina vähintäänkin pareto-optimaalinen resurssien jako. Vaikeuksia voivat tosin aiheuttaa erilaiset markkinoiden vääristymät: epätäydellinen kilpailu, sosiaaliset prioriteetit, ulkoisvaikutukset ja tietyille tärkeille resursseille puutteelliset markkinat. Sosiaaliset prioriteetit ja ulkoisvaikutukset puuttuvat tietokonejärjestelmistä kokonaan, mutta muut markkinoita vääristävät tekijät voivat vaatia erityisiä korjaustoimenpiteitä. Erityisen huolestuttavana pitäisin täydellisen kilpailun riittävän monien toimijoiden vaatimusta: useissa yksinkertaisimmissa resurssienhallintamalleissa kullekin resurssille on vain yksi myyjä, jolla on resurssin markkinoilla monopoliasema.

5 Markkinapohjaiset mallit muistinhallinnassa

Järjestelmän muistin hallinta globaalin ja lokaalin optimin saavuttamiseksi on selvästikin vaikea ongelma. Perinteiset järjestelmät, kuten esimerkiksi UNIX, yrittävätkin ratkaista ainoastaan globaalin optimoinnin ongelman käyttäen keskitettyä, käyttöjärjestelmän ytimen osana olevaa muistinhallintajärjestelmää.[18]

Lokaalin ja globaalin optimoinnin välille olisi tehtävä selvä ero. Sen sijaan että järjestelmää ajateltaisiin kokonaisuutena, se jaetaan itsenäisiin agentteihin, jotka vastaavat ihmistalouksien toimijoita: kuluttajina toimiviin sovelluksiin ja tuottajina toimiviin resurssien hoitajiin. Sovellukset ja resurssien hoitajat olisivat vastuussa vain omasta lokaalista optimoinnistaan. Asettamalla sovellusten ja resurssien hoitajien optimointiongelmat sopivasti, niiden sivuvaikutuksena voidaan saavuttaa globaalin optimoinnin likiarvo.[18]

Markkinapohjaisten mallien yhtenä houkutuksena resurssien hallinnassa on, että ne johtavat pareto-optimaaliseen resurssien jakoon hyvin määriteltyjen ehtojen vallitessa.

Kuten yritykset ihmistalouksissa, jokainen resurssin hoitaja on vastuussa tulojensa maksimoinnista. Tuloja resurssin hoitaja saa myymällä käyttöoikeuksia resurssiin sovelluksille. Sovellukset puolestaan ovat vastuussa hyötynsä maksimoinnista kuten ihmistalouksien kuluttajat. Sovellukset pyrkivät maksimoimaan hyötynsä ostamalla käyttöoikeuksia tarvitsemiinsa resursseihin resurssien hoitajilta tai vaihdon kautta muilta sovelluksilta.

Sovellukset saavat rahaa resurssien ostamiseen käyttäjäagentilta, jonka vastuulla on toteuttaa käyttäjän toimintalinjaa. Usean käyttäjän järjestelmissä järjestelmänlaajuinen toimintalinja määrää eri käyttäjäagenteille osoitettavista rahamääristä.

Lokaalin ja globaalin optimoinnin erottamisesta markkinamenetelmällä seuraa selviä etuja. Jokainen sovellus on vastuussa vain omasta hyödystään, eikä sovellusten hyötyä tarvitse siis määrittää sovellusten ulkopuolella. Lisäksi erottaminen mahdollistaa sovellusten sopeuttaa toimintaansa resurssien saatavuuden mukaan.[18]

Vaikka markkinapohjaiset mallit eivät välttämättä tarjoa mitään etua suhteessa järjestelmän yleiseen monimutkaisuuteen, ne tarjoavat selvästi lisää hajautettavuutta jopa kahdella eri tavalla. Markkinapohjaisissa malleissa päätöksenteko hajaantuu itsenäisille agenteille, joiden ei tarvitse huolehtia kuin omasta hyödystään ja resurssien hinnoista. Muut agentit ovat poissuljettuja yhden agentin päätöksenteosta. Kun riittävät ehdot on täytetty, ja yksittäiset agentit ovat riittävän pieniä, hinnat toimi-

vat yhteenvetona koko järjestelmästä. Toiseksi markkinoiden hallintamekanismit ovat luonnostaan hajautettuja. Resurssien hinnat määräytyvät vain yksittäisen resurssin kysynnän ja tarjonnan mukaan.[21]

Markkinapohjaisista malleista on myös paljon hyötyä järjestelmiä suunnitellessa. Laskennan perusyksikkönä toimii agentti, jolla on hyvin määrätty joukko kykyjä ja resursseja, ja tarkkaan rajattu tavoite. Resurssien hoitajilla tavoitteena on voiton maksimointi, ja sovelluksilla lokaalin hyödyn maksimointi. Lisäksi kaikki agenttien välinen kommunikointi tapahtuu vaihtojen kautta markkinahinnoilla. Täydellisessä kilpailussa voimme myös selvittää helposti vaikuttaako uuden agentin lisääminen järjestelmään, tarkastelemalla sen tuloksellisuutta järjestelmän sen hetkisillä hinnoilla. Järjestelmän toimintaan voidaan myös vaikuttaa haluttujen tulosten aikaansaamiseksi kannustejärjestelmillä. Lisäksi järjestelmää voidaan analysoida taloustieteen menetelmillä.[21]

Tietokonejärjestelmän resurssien hallinnan markkinapohjaisessa mallissa tietokonejärjestelmä on itse tuottaja, joka myy resursseja, kuten prosessoriaikaa, muistia tai I/O-laitteiden käyttöoikeuksia kuluttajina toimiville sovelluksille.

Markkinamekanismilla takoitetaan huutokauppaa, ja huutokaupalla puolestaan tarkoitetaan kilpailuun perustuvaa tasapainoituskäytäntöä. Huutokaupan keskeisin tehtävä on säätää tuotteen hinta niin, että sen kysyntä vastaa tarjontaa. Huutokauppa on siis eräänlainen tasapainottamisprosessi, jossa kysyntä ja tarjonta saatetaan keskenään tasapainoon.[20]

Yksinkertaisessa vaihtotaloudessa resurssien hinnat määräytyvät kysynnän mukaan, ja kysyntä puolestaan määräytyy hintojen mukaan. Tällaisessa järjestelmässä pareto-optimaaliset resurssien jaot vastaavat hintatasapainoja. Tasapainotilassa yksikään osapuoli ei vastaa hintoihin muuttamalla kysyntäänsä niin, että se vaikuttaisi hintoihin. Tasapainohinnat kertovatkin suhteet, millä resurssit tulisi jakaa eri osapuolille pareto-optimaalisesti.[20]

Useiden resurssien keskinäistä tasapainotilaa kutsutaan yleiseksi tasapainotilaksi. Yhtä resurssia tarkastellessa puhutaan osittaisesta tasapainotilasta, joka on yleensä helpompi löytää, mutta jättää huomiotta keskeisiä resurssien välisiä vuorovaikutuksia.[21]

Hintatasapainottaminen ja pareto-optimointi mahdollistaa suuremman epäyhtenäisyyden osapuolten välille kuin yksinkertaisemmat resurssien jako menetelmät. Ensinnäkin pareto-optimointi ei vaadi lokaalien hyötyjen olevan keskenään vertailukelpoisia. Toisekseen huutokauppakäytäntö mahdollistaa osapuolten tehdä päätöksiä itsenäisesti yksityisen tilan perusteella. Osapuolten alussa omistamat vaihdettavat hyödykkeet ja niiden mieltymykset ja päätöksentekoprosessit voivat vaihdella järjestelmässä suuresti.[20]

Huutokauppa kuitenkin olettaa yleensä, että kysyntä on monotonisesti laskeva suh-

teessa hintaan vakaan tasapainotilan saavuttamiseksi. Lisäksi tarjoukset oletetaan tehtävän perustuen ainoastaan osapuolten itse omistamiin hyödykkeisiin ilman tietoa muiden osapuolten omistuksista.[20]

Markkinamekanismi ei ole täydellinen resurssienjakoalgoritmi vaan ainoastaan tasapainottamiskäytäntö. Kun huutokauppamenetelmä on valittu, on vielä useita yksityiskohtia, jotka on määrättävä, ennen kuin järjestelmä suorituskyvystä voidaan alkaa tekemään päätelmiä. Erityisen suuri merkitys järjestelmän suorituskykyyn on agenttien suunnittelulla ja niiden toiminnalle asetetuilla rajoituksilla.

Markkinamenetelmällä tarkoitetaan optimoinnin ja ohjauksen kokonaista loogista rakennetta, johon valittu markkinamekanismi on sisällytetty. Voimakas markkinamenetelmä on rakenteeltaan ja toiminnaltaan lähellä ihmistalouksia. Agentteja määrittää korkea itsenäisyyden taso, ja niiden välinen kanssakäynti on rajattu vaihtoihin ja vaihtotarjouksiin. Agentit myös toimivat ainoastaan omaa hyötyään edistävillä tavoilla. Voimakkaan markkinamentelmän piirteisiin kuuluu myös, että järjestelmä kunnoittaa omistusoikeuksia.

Näennäismarkkinamenetelmässä on puolestaan vähemmän vapauksia ja se hyödyntää markkinamekanismia vähemmän. Agentit ovat toiminnaltaan samankaltaisia, eroten ainoastaan joidenkin parametrien osalta. Agentit voivat myös olla enemmän yhteistoiminnallisia kuin voimakkaissa markkinamenetelmissä: ne eivät välttämättä toimi keskinäisessä kilpailussa ja saattavat luopua omasta hyödystään yleisen hyödyn vuoksi. Lisäksi rajoitukset agenttien käyttäytymiselle mahdollistavat yleisen hyödyn suoran maksimoinnin ja pareto-optimin olemassaolon varmistamisen.

Markkinamenetelmät eivät kuitenkaan aina täysin sovellu tietojenkäsittelyn ongelmien ratkaisemiseen. Tämä johtuu pääasiassa ihmistalouksien ja tietokonejärjestelmien erilaisista taustaoletuksista ja tavoitteista. Ihmistaloudet myös yleensä toimivat epätäydellisesti eikä tietokonejärjestelmä kykene parhaimmillaankaan kuin jonkin ihmistalouden osan abstraktioon.

Järjestelmän suunnittelijan on otettava huomioon sen kannustinrakenne, eli mitä etuja ja haittoja agentit havaitsevat toimintaansa suunnitellessaan. Hyvä kannustinsuunnittelu lisää järjestelmän vakautta. Ensinnäkin kannustinrakenne on enemmän päämääräsuuntautunut kuin prosessisuuntautunut järjestelmän ohjausmalli. Agenteille jää huomattavasti vapautta improvisoida toimintaansa, mitä riittävän älykkäät agentit pystyvät hyvin käyttämään hyödykseen. Toisekseen kannustimilla voidaan suojata järjestelmää agenteilta joiden edut ovat vastoin yleistä etua.[20]

Kannustimet eivät kuitenkaan mahdollista tietokonejärjestelmän agenttien poiketa ohjelmastaan, joten agenttien sopiva toiminta voitaisiin varmistaa myös ilman kannustimia. Joissain tilanteissa tietokonejärjestelmän agentteja kuitenkin voidaan pitää

älykkäinä suhteessa kannustimiin. Jos agentteja ohjaavat samaa järjestelmää käyttävät eri henkilöt, joiden edut eivät osu yhteen, voidaan kannustimilla saada eri käyttäjien ohjaamat agentit toimimaan yhteisen hyödyn mukaisesti.

Tietokoneen agenttien puutteellinen älykkyys tekee kuitenkin järjestelmän ohjaamisen kannustinjärjestelmillä vaikeaksi. Älykkään agentin voidaan olettaa käyttävän hyväkseen jokaista tilaisuutta oman hyötynsä lisäämiseksi, mihin tietokonejärjestelmien agentit eivät kykene samalla joustavuudella.

Kilpailuun perustuvan tasapainottamisen suurimmat ongelmat liittyvät tasapainotilan potentiaaliseen huonoon laatuun ja vaikeuteen ennakoida ratkaisun löytymisen nopeutta. Pareto-optimi voi erota suurestikin globaalista optimista. Pareto-optimi on kuitenkin ainoa mahdollisuus, jos agenttien hyödyt eivät ole keskenään vertailukelpoisia, ja jos agenttien hyötyjen suhteellisten arvojen määrittämiseksi ei ole mitään keinoa.[20]

Pareto-optimointi voi myös joissain tapauksissa olla jopa toivottavampaa kuin globaali optimointi. Pareto-optimointi mahdollistaa optimointiongelman hajoittamisen erillisiksi aliongelmiksi, joiden ratkaiseminen voidaan antaa itsenäisille hajautetuille ohjelmistoyksiköille. Globaali optimointi voi myös joissain tapauksissa johtaa tilanteeseen, jossa joidenkin agenttien hyöty jää pysyvästi alhaiseksi. Pareto-optimointi varmistaa, ettei yhdenkään agentin hyöty ainakaan laske vaihtojen edetessä.

Markkinamenetelmä toimii hyvin, jos se toistuvasti ja nopeasti saavuttaa hyviä tuloksia. Tasapainottamisen ollessa kyseessä, se tarkoittaa että järjestelmä saavuttaa nopeasti globaalin optimin tai pystyy seuraamaan sitä dynaamisesti.

Jos markkinamenetelmällä haetaan ratkaisua kiinteiden agenttien tulojen ja mieltymysten ennalta esitettyyn ongelmaan, tasapainotilan olemassaolo ja mahdollisuus lähestyä sitä on tärkeää. Jos taas markkinamenetelmää käytetään ratkaisemaan dynaamista ongelmaa, jossa agentit osallistuvat jatkuvaan resurssien vaihdantaan ja käyttöön (kuten käyttöjärjestelmän resurssien hallinnassa), tasapainotilan olemassaolon osoittaminen ei ole kuitenkaan niin tärkeää. Tasapainotilan lähestyminen dynaamisissa tilanteissa onkin vähemmän tunnettua. Tasainen lähestyminen on kuitenkin odotettavissa useissa eri tilanteissa olettaen agenttien käyttäytymisen olevan uskottavaa.[20]

5.1 Hinnanmääritys

Markkinamalleissa on kaksi päätapaa jakaa järjestelmän resurssit kilpailevien agenttien kesken. Ensimmäinen tapa on vaihtoon perustuva. Agentit saavat alussa jonkin osan järjestelmän resursseista, mitä ne sitten alkavat vaihtamaan keskenään. Vaihtoa jatketaan, kunnes molempia osapuolia hyödyttäviä vaihtoja ei voida enää tehdä, jolloin

on saavutettu pareto-optimi.[5]

Toinen tapa on resurssien hintaan perustuva. Resursseille määrätään hinnat niiden kysynnän ja tarjonnan ja järjestelmän vaurauden perusteella. Agenteille määrätään jokin osa vauraudesta, millä ne voivat ostaa tarvitsemiaan resursseja järjestelmältä. Useita resursseja jaettaessa on löydettävä kaikille resursseille hinnat, joilla saavutetaan tasapainotila kaikille resursseille. Koska resurssien kysyntä ja tarjonta riippuvat toisistaan, kasvaa resurssien jaon monimutkaisuus nopeasti resurssien määrän kasvaessa.

Hintatasapainottamista käytetään tilanteissa, joissa osallistuminen vaihdantaan on vapaaehtoista, ja toimiminen vaatii kaikkien osapuolten tuntevan hyötyvänsä vaihdosta. Järjestelmässä, jossa osapuolet ryhtyvät vaihtoon, vain jos kaikki kokevat hyötyvänsä siitä, kaikkien osapuolten hyöty kasvaa jokaisen vaihdon myötä. Hinnat ovat tasapainossa kun ne eivät enää muutu, ellei jokin ulkopuolinen voima muuta kysyntää tai tarjontaa. Järjestelmä siis kulkee kohti pareto-optimia.

Tunnustelu on hinnanmääritysmenetelmä, jossa agentit määrittävät oman kysyntänsä resursseille hyötynsä ja niiden käytettävissä olevan varallisuuden perusteella. Agenttien yhteenlaskettu kysyntä annetaan tuottajille, jotka määrittävät kysynnän perusteella resursseille uudet hinnat, jonka jälkeen agentit määrittävät kysyntänsä uudestaan uusien hintojen perusteella. Uusi kysyntä annetaan taas tuottajille, jotka määrittävät jälleen uudet hinnat. Prosessia jatketaan, kunnes kysynnän ja tarjonnan tasapainotila saavutetaan. Eli, kun uusi kysyntä ei enää muuta hintoja, tai uudet hinnat eivät enää muuta kysyntää.[5]

Tunnustelun suurimpana ongelmana on, että hintatasapainon on löydyttävä ennen, kun resursseja voidaan alkaa osoittamaan agenteille. Tunnustelu soveltuukin hyvin tilanteisiin, jossa agentit ja niille jaettavat resurssit voidaan määritellä kiinteästi ennen vaihdon aloittamista.

Resurssien hintojen määritys voidaan myös jättää resurssin hoitajalle, joka määrittää hinnat dynaamisesti agenttien esittämän kysynnän ja tarjonnan muuttuessa. Hinnanmäärityksen dynaamisen luonteen takia hinnanmääritysalgoritmi ei saa kuitenkaan käyttää liikaa aikaa tai resursseja, koska sitä joudutaan käyttämään usein.

Hinnanmääritysalgoritmilta pitäisi vaatia myös, että hinnat pysyisivät vakaina pitkällä aikavälillä, vaikka hetkellisiä heittoja hinnoissa kuuluukin tapahtua kysynnän ja tarjonnan vaihdellessa. Resursseille pitäisi myös pystyä määrittämään hinnat vaihteleville aikaväleille resurssista ja sovelluksesta riippuen.

Hinnoittelun tarkoituksena on tarjota käyttäjälle ja sovelluksille kannustimet toimia järjestelmän kannalta toivottavalla tavalla. Tämä tarkoittaa, että kun resurssin käyttö lisääntyy, resurssin hoitaja nostaa resurssin hintaa kannustaen käyttäjiä siirtymään käyttämään vähemmän käytettyjä ja näin ollen halvempia resursseja.

Dynaamisissa tilanteissa resurssien hinnat voidaan määrätä myös huutokaupalla. Huutokauppa voidaan järjestää usealla eri tavalla. Tuplahuutokaupassa myyjät tarjoavat matalampia ja matalampia hintoja samalla, kun ostajat tarjoavat korkeampia ja korkeampia hintoja, ja tarjouksia jatketaan kunnes ne kohtaavat. Englantilaisessa huutokaupassa ostajat tarjoavat korkeampia hintoja, kunnes tarjoukset saavuttavat lakipisteen, ja myyjä hyväksyy korkeimman tarjouksen. Hollantilaisessa huutokaupassa myyjä tarjoaa matalampia hintoja, kunnes joku ostajista hyväksyy tarjouksen.

Suljetussa huutokaupassa ostajat tekevät tarjouksensa toistensa tarjouksia tietämättä, ja korkein tarjous hyväksytään. Vaihtoehtoisesti, toiseksi korkeimman hinnan suljetussa huutokaupassa, korkeimman tarjouksen tekijä voittaa huutokaupan, mutta maksaa toiseksi korkeimman tarjouksen hinnan.

Englantilainen ja suljettu toiseksi korkeimman hinnan huutokauppa johtavat tehokkaasti toimiviin markkinoihin. Englantilaisessa huutokaupassa voittaja maksaa vain hieman yli toiseksi korkeimman hinnan, eli melkein saman, kuin toiseksi korkeimman hinnan suljetussa huutokaupassa.[3]

Hollantilainen ja korkeimman hinnan suljettu huutokauppa johtavat tehottomasti toimiviin markkinoihin, koska ostajille on hyödyllistä yrittää arvata muiden ostajien tarjouksia. Englantilaisessa ja toiseksi korkeimman hinnan suljetussa huutokaupassa taas ostajan on aina järkevä tarjota sen verran, kuin on valmis myytävästä hyödykkeestä maksamaan.[3]

Kiinteä, ennalta määrätty hinta resurssille on helpoin toteuttaa ja usein myös helpompi sovelluksille. Resurssien hintojen pysyminen vakiona helpottaa toiminnan ja kulutuksen suunnittelua ennalta. Kiinteä hinta on helpompi myös resurssien hoitajille, kun resurssien hintoja ei tarvitse määrittää jatkuvasti uusiksi, tai järjestää niistä huutokauppoja.

5.2 Agentit

Talouden suorituskyky on seurausta sen instituutioiden rakenteesta, markkinaympäristöstä ja agenttien käyttäytymisestä. Talouden instituutioiden rakenteen määräävät vaihtoa hallitsevat säännökset. Markkinaympäristön määrää agenttien mieltymykset, sekä niiden käytössä oleva informaatio ja resurssit. Agenttien käyttäytymisen määrää niiden kaupankäyntistrategia.

Tietokonejärjestelmän markkinoiden agenteilta saattaa puuttua joitain ihmistalouksien toiminnan kannalta oleellisia ominaisuuksia. Ihmistalouksissa agenttien oletetaan kykenevän taloudelliseen toimintaan älykkään itsenäisen päätöksenteon perusteella.

Älykkäällä toiminnalla tarkoitetaan yleensä kykyä toimia johdonmukaisesti omien etujen mukaisesti. Älykkyyden (tässä mielessä) oletetaan sisältävän johdonmukaisuuden (älykäs agentti ei usko että molemmat A ja ei-A) ja täydellisyyden (älykkäällä agentilla on uskomuksia kaikista relevanteista väitteistä).[20]

Heikomman määritelmän mukaan älykkyyden voidaan olettaa tarkoittavan, ettei agentti tarkoituksella tee mitään itselleen epäedullista. Voimakkaamman määritelmän mukaan agentin voidaan olettaa kykenevän uudenlaiseen ja mielivaltaisen monimutkaiseen pättelyyn kaiken saatavilla olevan tiedon perusteella.[20]

Taloudellisten ennusteiden tekemisessä ei kuitenkaan ole hyödyllistä olettaa liian voimakasta älykkyyden muotoa. Ensinnäkin on hyvin epäuskottavaa, että ihmiset soveltaisivat johdonmukaisesti korkealle kehittynyttä päättelyä päätöksiä tehdessään. Toisekseen ihmiskokeet ovat osoittaneet ihmisten päätöksenteossa merkittäviä poikkeamia optimaalisista strategioista.[20]

Vaikka ihmismarkkinat perustuisivatkin älykkäiden agenttien toimintaan, se ei tarkoita, etteikö markkinoita voitaisi perustaa yksinkertaisemmillekin agenteille. Erityistä huomiota onkin kohdistettava markkinoilla toimimiseen vaadittavaan minimi kyvykkyyteen ja monimutkaisuuteen.

Kokeet ovatkin osoittaneet markkinoiden lähestyvän nopeasti tasapainotilaa jopa hyvin yksinkertaisilla agenteilla. Tasapainotila voidaan löytää agenteilla, jotka eivät pysty kommunikoimaan keskenään, päättelemään taloudellisesti tai tiedä muiden agenttien mieltymyksistä. Joissain tapauksissa hintatasapaino saavutetaan vaikka agentit toimisivat lähes satunnaisesti.[12]

Markkinoiden tehokkuus johtuukin pääasiassa markkinamallista. Agenttien oppimiskyky, älykkyys tai edes hyödyn tavoittelu ei ole markkinoiden tehokkaan toiminnan kannalta välttämätöntä. Markkimamalli on myös tärkeässä asemassa hintojen lähestymisessä tasapainotilaa.[8]

5.3 Hartyn ja Cheritonin muistinhallintamalli

Hartyn ja Cheritonin esittelemässä markkinamallissa[9] muistin osoittaminen on erillisten segmenttihoitajien vastuulla. Segmenttihoitajat päättävät miten paljon ja miten pitkäksi aikaa sovelluksille osoitetaan muistia. Prosessoriaika on puolestaan alisteinen muistille: sovellus saa prosessoriaikaa vain jos sillä on muistia.

Muistimarkkinamallissa sovelluksilta laskutetaan niille osoitetusta fyysisestä muistista. Muistin hinta määräytyy etukäteen määrätyllä tavalla osoitetun muistin määrän, osoituksen ajan ja pyynnön tärkeyden mukaan, eivätkä kysynnän vaihtelut muuta hintaa. Sovellukset esittävät pyyntöjä tarvitsemastaan määrästä muistia haluamakseen

ajaksi. Jos sovelluksella on tarpeeksi rahaa, pyydetty muisti myönnetään sovelluksen käyttöön sovituksi ajaksi, ja sovellus luovuttaa muistin takaisin järjestelmälle vuokra-ajan päätyttyä.

Jokaisella sovelluksella on tili, jolta muistin kustannukset maksetaan. Tileille talletettavalla rahalla on yläraja, jottei yksi sovellus voi säästää itselleen niin paljoa rahaa, että voisi ottaa haltuun kaikkea muistia kohtuuttoman pitkäksi aikaa häiriten merkittävästi muiden sovellusten toimintaa.

Järjestelmän sivuvälimuistihoitaja tallettaa tileille rahaa säännöllisin väliajoin järjestelmän toimintalinjan mukaan. Tilit on suojattu ja niitä käsitellään sivuvälimuistihoitajan kautta. Sovellus voi kysyä sivuvälimuistihoitajalta tilinsä saldoa, tulojaan sekä muistisivujen hintaa.

Yksinkertaisessa kokoonpanossa kaikki yhden käyttäjän sovellukset jakavat yhden tilin. Sovelluksille voidaan kuitenkin luoda omia tilejä niiden eristämiseksi muista sovelluksista. Eristetty sovellus ei voi näin estää muita sovelluksia saamasta osaansa muistista, ja samalla sovellukselle voidaan taata tietty osa muistista.

Sovellus voi esittää minimin ja maksimin haluamalleen sivumäärälle pyytäessään muistia sivuvälimuistihoitajalta. Sivuvälimuistihoitaja osoittaa sovellukselle niin paljon muistia, kuin rajoitusten perusteella pystyy. Pyytäessään muistia sovellus voi myös jäädä odottamaan muistin vapautumista, jos vapaata muistia ei pyydettäessä ole.

Saatuaan tarvitsemansa muistin ja alettuaan suorituksensa, sovellus tyypillisesti asettaa ajastimen ilmoittamaan, milloin muistin käyttöoikeus päättyy. Sovellusten on sisällytettävä vuokra-aikaansa myös riittävästi aikaa likaisten sivujen kirjoittamiseksi toissijaiseen muistiin.

Sovellus, joka ei luovuta muistia ajoissa, joutuu maksamaan vuokra-ajan yli menevästä ajasta sakkoa. Jos muistia ei luovuteta ennalta määrätyn sakkoajan jälkeenkään, muisti otetaan sovellukselta väkisin, ja sovellus voidaan myös lopettaa kokonaan. Jos kuitenkaan vuokra-ajan jälkeen muistille ei ole muuta tarvetta, sovellus voi pitää muistia yliaikaa maksutta.

Muistin hinnoittelulla määrätään, miten muistin hinta määräytyy muistin määrän, vuokra-ajan ja pyynnön tärkeyden mukaan. Hinnoittelu on toteutettu kolmitasoisena, vastaten pyyntöjen kolmea eri tärkeystasoa: matala, normaali ja korkea. Kiinteä hinnoittelu valittiin, koska vaihteleva hinnoittelu tuntui vaikealta ohjelmien käsitellä. Ohjelmien oli vaikea suunnitella etukäteen suoritustaan ja rahankäyttöään. Kiinteä hinnoittelu myös yksinkertaistaa sivuvälimuistihoitajaa, kun sen ei tarvitse määrätä uusia hintoja kysynnän muuttuessa.

Kolmitasoinen hinnoittelu mahdollistaa sovellusten pyytää muistia eri tärkeyksillä riippuen sovelluksen muistin tarpeen laadusta. Matalan tärkeyden pyynnöt täytetään

ainoastaan, kun tärkeämpiä pyyntöjä ei ole täyttämättä. Matalan tärkeyden muistin vuokra-aikaa voidaan lyhentää tarvittaessa, jos muistille tulee korkeamman tärkeyden pyyntöjä. Matalan tärkeyden muisti on maksutonta.

Matalan tärkeyden muisti mahdollistaa sovellusten hyödyntää muistia silloin, kun sille ei ole muuta käyttöä, ja saada muistia käyttöönsä, kun niillä ei ole rahaa. Matalan tärkeyden muisti edesauttaa muistin tehokasta käyttöä antaen muistia maksutta sovellusten käyttöön muistin jäädessä muuten käyttämättömäksi. Sivuvälimuistihoitaja voi myös herättää prosesseja, jotka odottavat lisää rahaa, kun muistia on käyttämättömänä.

Maksutonta muistia voivat hyödyntää myös jotkut sovellukset, joiden suorituskykyä voidaan parantaa lisämuistilla, mutta joiden suoritukselle se ei ole välttämätöntä. Tällaisia sovelluksia ovat esimerkiksi sovellukset, jotka ylläpitävät omaa välimuistia.

Korkean tärkeyden pyynnöt tarjoavat nopeampaa palvelua sovelluksille, joiden vasteaikavaatimukset ovat tiukempia, kuten esimerkiksi vuorovaikutukselliset sovellukset. Korkean tärkeyden pyynnöt toteutetaan ennen alempien tärkeystasojen pyyntöjä.

Muistin ajoitustoimintalinja määrää muistin osoitusten alku- ja loppuajat. Ajoitustoimintalinja voi myös määrätä osoituksille maksimiajan, maksimimäärän, tai varata osan muistista erityisesti korkean tärkeyden pyyntöjä varten niiden täyttämisen nopeuttamiseksi.

Sivuvälimuistihoitaja ylläpitää alustavaa aikataulua pyyntöjen käsittelemiseksi. Uuden pyynnön tullessa se liitetään pyyntöihin korkeamman tärkeyden pyyntöjen jälkeen ja matalamman tärkeyden pyyntöjä ennen. Pyyntö otetaan vastaan vasta, kun sovelluksella on riittävästi rahaa sen maksamiseksi. Ajastusalgoritmi takaa, ettei pyyntö joudu odottamaan matalamman tärkeyden pyynnön täyttämisen takia.

Ajastin käyttää pyynnön alarajaa määrittääkseen osoituksen ajastuksen, mutta myöntää muistia niin paljon kun pystyy aina pyynnön ylärajaan saakka. Muistia pyritään myöntämään kuitenkin niin, ettei se viivytä turhaan seuraavia pyyntöjä.

Pyyntöjen koolle voidaan myös asettaa yläraja. Suuret sovellukset saattaisivat muuten estää pienempien suorituksen lähes kokonaan jos niiden sallitaan varata muistia liikaa yhdellä kertaa. Lisäksi saman tärkeyden pienemmät pyynnöt joutuvat odottamaan suuren pyynnön täyttämistä, vaikka muistia olisikin niiden tarpeisiin riittävästi.

Osoitusten kestoille asetetun ylärajan ollessa pitkä voivat vuorovaikutuksellisten sovellusten vasteajat kasvaa pitkiksi. Järjestelmä voikin varata osan muistista korkean tärkeyden pyynnöille, joita vuorovaikutukselliset sovellukset voivat käyttää pyyntöjensä käsittelyn nopeuttamiseksi.

5.4 Utahin yliopiston TENEX järjestelmä

Vuonna 1971 Utahin yliopisto vaihtoi DEC 1050 -järjestelmänsä TENEX-järjestelmään. TENEX-järjestelmän ajastimesta saatiin paljon valituksia käyttäjiltä ja siinä havaittiin olevan vakavia ongelmia. Käyttäjien sovelluksien suorituksissa saattoi ilmetä pitkiä, jopa viidentoista minuutin mittaisia taukoja järjestelmän käytön ollessa kohtuullisellakin tasolla. Havaittujen ongelmien korjaamiseksi TENEX-järjestelmää katsottiin tarpeelliksei korjata vastaamaan paremmin yliopiston tarpeisiin.[4]

Alkuperäinen järjestelmä asetti korkean tärkeyden vuorovaikutuksellisille prosesseille, ja pitkiä laskentajaksoja suorittavat prosessit joutuivat usein odottamaan tarpeettoman pitkiä aikoja sen seurauksena. Utahin yliopiston tarpeet olivat erilaiset kuin alkuperäisen TENEX-järjestelmän oletukset. Vuorovaikutukselliset prosessit olivat yleensä alhaisen tärkeyden ohjelmointiharjoittelijoiden prosesseja. Lisäksi yliopiston koneessa oli vähemmän muistia, kuin mille alkuperäinen TENEX-järjestelmä oli suunniteltu.

Kaikki tietynlaisten prosessien suosimista koskevat päätökset päätettiin poistaa ajastimelta, ja päätöksenteko siirrettiin ylläpitäjän vastuulle. Resurssien jakamisen malliksi valittiin markkinat, ja markkinoille luotiin rahayksikkö Boynton (muutokset ohjelmoineen Thomas L. Boyntonin mukaan). Resurssien arvo määräytyisi Boyntoneina. Koska resursseille ei pystytty määräämään etukäteen hintoja, resurssien hinnanmääritys jätettiin ajastimen tehtäväksi.

Jokaiselle käyttäjälle osoitettin tulo Boyntoneina, minkä tarkoitus oli ilmaista resurssien jakautumista käyttäjien kesken. Käyttäjän tulot määräävät käyttäjän mahdollisuudet kilpailla järjestelmän resursseista muiden käyttäjien kanssa. Tuloilla pystytään varmistamaan resurssien jakautuminen reilusti kilpailevien prosessien kesken. Käyttäjän tulot kertoo käyttäjän prosesseille joka millisekuntti osoitetavan Boynton määrän.

Markkinamallissa järjestelmän täytyy vain pyrkiä tienaamaan mahdollisimman paljon jokaisesta työstä. Tuloja järjestelmä saa veloittamalla prosesseja niiden käyttämistä resursseista. Järjestelmä pystyy myös myöntämään luottoa ja pystyy ajoittain maksimoimaan tulonsa pelkästään hinnat maksimoimalla.

Resurssien hinnat määräytyvät resurssin viimeaikaisen markkinahistorian perusteella. Jos riittävän suuri osa resursseista on osoitettu prosesseille, joiden tili on plussan puolella, resurssin hinta nousee, ja päinvastoin. Valittu hinnanmuodostusmekanismi valittiin huutokaupan sijaan tarvittavien muutosten yksinkertaisuuden vuoksi verrattuna huutokauppamekanismiin.

Uuden ajastimen myötä käyttäjien resurssien kulutuksen lisäksi pidetään kirjaa nii-

den Boynton kulutuksesta. Hintojen vaihtelua voidaan myös seurata niiden muuttuessa. Prosessien kulutusta ja hintoja seuraamalla voidaan havaita käyttäjät, jotka vaikuttavat muihin käyttäjiin, vaikka ne eivät näkyisikään prosessorin käyttötilastoissa. Käyttäjät voivat myös seurata hintojen kehitystä ja voivat käyttää tietoa toimintansa ohjaamiseen.

Käyttäjät on järjestetty puurakenteeseen yliopiston palkkalistan rakennetta vastaten. Käyttäjät on jaettu projekteihin, joilla jokaisella on projektijohtaja. Projektijohtajat vastaavat projektista päätutkijalle. Pienin mahdollinen tulo on 2 ja suurin 200 Boyntonia millisekunnissa. Päätutkija määrää projektien henkilöiden tuloille minimin, maksimin ja keskiarvon, missä puitteissa projektijohtaja asettaa projektin käyttäjien tulot.

Jokaiselle prosessille on kirjattu sen omistavan käyttäjän tulo jaettuna käyttäjän omistamien prosessien lukumäärällä (RATE) ja prosessin tilin saldo (POT). Käyttäjän tulot jaetaan sen prosessien kesken, jotteivat käyttäjät saisi enemmän tuloja, kuin niille on osoitettu, vain ajamalla useampia prosesseja. Prosessien tileille lisätään rahaa RATEn verran. Lisäys tehdään teoriassa joka millisekuntti, mutta POT lasketaan kuitenkin tosiasiassa vain tarvittaessa. Prosessin POT alustetaan nollaan, kun prosessi luodaan.

Prosessin POTin yläraja on MAXPOT, joka määräytyy hintojen perusteella. POT voi olla ajoittain myös negatiivinen (ja usein onkin), eikä sillä ole alarajaa.

Prosessi käyttää rahaa tililtään saamiensa resurssien maksamiseen niiden sen hetkisen hinnan mukaan. Prosessoriajasta prosessi joutuu maksamaan myös siltä ajalta, kun estää muiden prosessien pääsyn prosessorille, vaikka prosessi ei itse suorittaisikaan (esimerkiksi, jos se odottaa sivujen lukemista/kirjoittamista toissijaiseen muistiin).

Järjestelmä laskee resursseille uudet hinnat sekunnin välein kuluneen sekunnin tietojen perusteella. Hinnat lasketaan resursseille erikseen ennalta määrättyjen kaavojen mukaan.

Alkuperäisen TENEXin viiden, eri tärkeyksiä vastaavien prosessijonojen sijaan Utahin yliopiston TENEX-järjestelmän ajastin määrittää kuusi ryhmää, joihin prosessit jaetaan ja jotka määräävät prosessien tärkeyden. Ryhmien sisäinen järjestys määräytyy prosessien POTin perusteella. (THRESHOLD on rahamäärä, jonka annettu prosessi voisi kuluttaa yhden aikajakson aikana. THRESHOLD määräytyy työjoukon koon ja sen hetkisten hintojen mukaan.)

G6) Prosessille ei ole osoitettu muistia ja sen (POT - THRESHOLD) < 0.

G5) Prosessille on osoitettu muisti ja sen POT < 0.

G4) Prosessille on osoitettu muisti ja sen POT > 0.

G3) Prosessille ei ole osoitettu muistia ja sen (POT - THRESHOLD) > 0.

G2) Prosessilla on ohjelmistokeskeytys käsittelemättä ja muuten se täyttää G3) tai G4) vaatimukset.

G1) Prosesille on osoitettu muisti eikä sen aikajakso ole käytetty.

Valittaessa suoritettavaa prosessia, järjestelmä valitsee suurimman POTin omaavan prosessin tärkeimmästä ryhmästä.

6 Oma toteutukseni markkinapohjaisesta muistinhallintajärjestelmästä

Työni lopuksi esittelen itse tekemäni toteutuksen Hartyn ja Cheritonin markkinapohjaisen muistinhallintamallin[9] sivuvälimuistihoitajasta. En kuitenkaan ole toteuttanut kaikkia Hartyn ja Cheritonin kuvailemia ominaisuuksia, enkä ole kaikkea toteuttanut aivan samalla tavalla kuin he kuvailevat. Selvitän erot yksityiskohtaisesti toteutustani kuvaillessani. Toteutus on tehty Common Lisp ohjelmointikielellä käyttäen Movitz-ohjelmointiympäristöä. Aloitan esittelyn kertomalla hieman Intel Pentium arkkitehtuurin muistinhallintatekniikoista ja Movitz ympäristöstä. Lopuksi esittelen toteutukseni, ja miltä osin se toteuttaa Hartyn ja Cheritonin kuvailemaa mallia, ja miltä osin eroaa siitä.

6.1 Muistinhallinta Intel arkkitehtuurissa

Vuonna 1993 Intel esitteli uuden Pentium arkkitehtuuriin perustuvan prosessorinsa[10, 11]. Vuonna 1995 Pentium prosessoria seurasi Pentium Pro, jonka jälkeen olemme edenneet jo vuonna 2000 esiteltyyn Pentium 4 prosessoriin.

Pentium-arkkitehtuuri tukee fyysisen muistin suoraa osoittamista tai fyysisen muistin osoittamista virtuaaliosoitteiden kautta. Suoraa osoittamista käyttäessä lineaarista osoitetta käytetään suoraan fyysisenä osoitteena ilman tulkkausta. Virtuaaliosoitteita käytettäessä lineaarinen osoite tulkataan fyysiseksi osoitteeksi.

Arkkitehtuurin muistinhallintapalvelut jakautuvat kahteen osaan: lohkoihin jakoon ja sivutukseen. Jako lohkoihin mahdollistaa eri tehtävien muistialueiden eristämisen toisistaan, sekä yhden tehtävän eri muistialueiden (koodi, data, pino) eristämisen toisistaan. Sivutus puolestaan tarjoaa tarvittavat mekanismit tavallisen virtuaalimuistijärjestelmän toteuttamiseksi. Sivutusta käytettäessä tehtävien koodi, data ja pino lohkot sekä järjestelmälohko ja lohkotaulut voidaan sivuttaa.

En tässä mene syvällisesti lohkoihin jaon mekanismeihin, koska ne eivät olleet työni kannalta oleellisessa asemassa, vaan keskityn pääasiassa arkkitehtuurin tarjoamiin työkaluihin sivutuksen toteuttamiseksi.

Intel Pentium arkkitehtuuri tarjoaa neljän gigatavun fyysisen muistiavaruuden, johon prosessori pystyy suoraan osoittamaan järjestelmän osoiteväylän kautta. Fyysi-

sen osoiteavaruuden osoitteet ovat välillä $0 - \texttt{FFFFFFFF}$. Muistiavaruus voidaan jakaa lohkoihin ja sivuihin, jotka voidaan kuvata read/write, read-only ja muistikuvatuksi I/O:ksi.

Fyysisen osoitteen saadakseen prosessori käyttää kaksitasoista osoitteen tulkkausta. Muistin tavuihin osoitetaan loogisella osoitteella, joka koostuu 16-bittisestä lohkovalitsimesta ja 32-bittisestä poikkeamasta. Lohkovalitsin kertoo, missä lohkossa haluttu tavu sijaitsee, ja poikkeama kertoo tavun sijainnin lohkon sisällä. Tämän ensimmäisen tulkkausvaiheen kautta loogisesta osoitteesta saadaan lineaarinen osoite, joka taas on 32-bittinen osoite prosessorin lineaarisessa osoiteavaruudessa.

Prosessorin lineaarinen osoiteavaruus on jakamaton neljän gigatavun osoiteavaruus, jonka osoitteet ovat myös välillä $0 - \texttt{FFFFFFFF}$. Jos sivutusta ei käytetä, lineaarinen osoite kuvataan suoraan fyysiseksi osoitteeksi. Jos taas sivutus on käytössä, lineaarinen osoite tulkataan toisen tulkkausvaiheen kautta lopulliseksi fyysiseksi osoitteeksi. Lineaarisen osoitteen bitit 22-31 antavat poikkeaman sivuhakemistoon, bitit 12-21 poikkeaman sivutauluun ja bitit 0-11 poikkeaman sivuun, josta haluttu tavu löytyy.

Sivutusta käytettäessä prosessorin lineaarinen muistiavaruus on jaettu (yleensä) neljän kilotavun kokoisiin sivuihin, jotka voidaan kuvata joko fyysiseen muistiin tai toissijaiseen muistiin. Kun prosessorin suorittama tehtävän käsky osoittaa loogiseen osoitteeseen muistissa, prosessori tulkkaa osoitteen fyysiseksi osoitteeksi, ja jos haluttu sivu ei ole fyysisessä muistissa, prosessori synnyttää sivuvirheen. Sivuvirheen sattuessa poikkeuskäsittelijä pyytää käyttöjärjestelmää lataamaan sivun fyysiseen muistiin, minkä jälkeen suoritusta voidaan jatkaa.

Tieto lineaariavaruuden sivujen kuvauksista fyysiseen muistiin säilytetään fyysisessä muistissa sijaitsevissa sivuhakemistoissa ja -tauluissa. Sivuhakemisto on taulu 32-bittisiä sivuhakemistomerkintöjä yhdessä neljän kilotavun sivussa. Sivuhakemistoon mahtuu 1024 sivuhakemistomerkintää. Sivuhakemistomerkintä viittaa sivutaulun sijaintiin fyysisessä muistissa ja sisältää tiedot sivutaulun käyttöoikeuksista ja muistinhallintaan käytettäviä tietoja. Sivutaulu on taulu 32-bittisiä sivutaulumerkintöjä yhdessä neljän kilotavun sivussa. Sivutauluun mahtuu 1024 sivutaulumerkintää. Sivutaulumerkintä viittaa yhteen fyysisen muistin sivuun ja sisältää tiedot sivun käyttöoikeuksista ja muistinhallintaan käytettäviä tietoja.

Sivuhakemiston fyysinen osoite on tallennettuna CR3-rekisteriin (toiselta nimeltään PDBR-rekisteri), ja se on oltava tallennettuna ennen sivutuksen kytkemistä päälle. PDBR-rekisteriin voidaan tallentaa uusi arvo joko eksplisiittisesti `MOV`-käskyllä tai implisiittisesti tehtävän vaihdon yhteydessä. Sivuhakemiston on oltava fyysisessä muistissa, kun tehtävää aletaan suorittamaan ja koko sen suorituksen ajan.

6.2 Movitz

Movitz[7, 6] on Common Lisp toteutus, joka toimii suoraan x86 laitteistoarkkitehtuurilla ilman käyttöjärjestelmän tai muiden ohjelmistoympäristöjen tukea. Movitz on tarkoitettu toimimaan kehitysympäristönä käyttöjärjestelmille sekä sulautetuille ja yhden tarkoituksen sovelluksille.

Movitz kehitysympäristö koostuu kahdesta osasta: minimaalisesta suoritusympäristöstä, joka on suunniteltu x86 arkkitehtuurille, ja ristiinkääntäjästä, joka kääntää ohjelmia suoritusympäristölle. Suoritusympäristön tarkoituksena on toteuttaa koko ANSI Common Lisp määritys sekä tyypillisen käyttöjärjestelmän tarvitsemat toiminnot. Movitz on täysin omavarainen siinä mielessä, että se ei tarvitse muita ohjelmistopalveluita kuin mitä se itse toteuttaa.

Tällä hetkellä Movitziin kuuluu yksi ydin-sovellus: `los0`. Ydin-sovellus `los0` sisältää suoritusympäristön ja tarjoaa ainoastaan tavallisen Common Lisp komentotulkin, jonka kautta järjestelmää voidaan tutkia ajon aikana. Movitz vaatii vähintään 386-prosessorin ja vähintään kaksi megatavua muistia. Movitz voidaan käynnistää levykuvalta PC-emulaattorilla, oikealta levykkeltä tietokoneella tai tietokoneella käyttäen esilataajaa.

Movitziin kuuluva Common Lisp kirjasto määrittää kattavan joukon funktioita ja makroja. Kirjasto ei ole vielä täydellinen, mutta toteutetut osat pyrkivät toteuttamaan ANSI määritelmän mahdollisimman kattavasti. Joitain yksityiskohtia:

- Numeroluokista on toteutettu ainoastaan fixnum-luokka, eli 30-bittiset kokonaisluvut $(-(2^{29}) - (2^{29} - 1))$.
- Tuki erilaisille listatyypeille on puutteellinen: jotkut funktiot toimivat vain tietyillä tyypeillä, ja jotkut eivät hyväksy `FROM-END` avainsanaa.
- Common Lisp kirjaston `EVAL` ei tue makroja, vaan ainoastaan funktioita. Jotkut makrot ovat kuitenkin toteutettu tuettuina erikoisoperaattoreina kuten `SETF`.
- Common Lisp Object System, eli CLOS, on tuettu. Mukaanlukien täysin toimivat `defclass` ja `defmethod`.
- Koska Movitzissa ei ole tiedostojärjestelmää, ei kirjastossa ole myöskään mitään tiedostojärjestelmän käsittelyyn käytettäviä funktioita.

Movitz on vasta kasvamassa eikä vielä lähellekään valmis, mutta se toteutti kaiken omaa työtäni varten tarvittavan. Lisäksi kehitystyöhön osallistuva ryhmä, vaikka onkin pieni, vaikuttaa aktiiviselta sähköpostilistalla käytävän keskustelun perusteella.

Keskustelua käydään myös #movitz IRC-kanavalla Freenode-verkossa, ja kaikki osallistujat olivat aina valmiita auttamaan minua ymmärtämään Movtizin toimintaa paremmin.

6.3 Toteutus

Kuvailtuani käytössäni olevat laitteiston tarjoamat työkalut ja Movitzin tarjoamat ohjelmistopalvelut, siirryn kuvailemaan itse muistinhallintamallin toteutusta.

6.3.1 Kellokäsittelijä

Ensimmäisiä asioita, jonka havaitsin tarvitsevani muistin varausjärjestelmää suunnitellessani, oli jonkinlainen kello, jolla ajastaa muistin vuokrauksia. Intel Pentium arkkitehtuuri sisältää kellopiirin, jolta pystyin saamaan riittävän tasaisen ajastuksen tarpeitani varten. Oletuksena kellopiiri lähettää keskeytyksen noin 18.2 hertsin taajuudella.

Movitz tarjoaa riittävät työkalut kellokeskeytysten käsittelyyn, ja näin pystyin toteuttamaan sivuvälimuistihoitajan tarpeisiin riittävän kellokäsittelijän. Kellokäsittelijälle annetaan vastakutsu-funktio ja haluttu aika, jolloin käsittelijä halutun ajan kuluttua kutsuu annettua funktiota. Kellokäsittelijää käytetään sivuvälimuistihoitajassa kahteen tehtävään: tilien saldojen kasvattamiseksi tasaisin väliajoin ja ilmoittamaan seuraavasta vuokra-ajan umpeutumisesta.

6.3.2 Fyysinen muistiavaruus

Seuraavaksi loin kuvan tietokoneen fyysisestä muistiavaruudesta ja tarvittavat operaatiot fyysisten muistisivujen varaamiseksi ja palauttamiseksi. Abstraktio on äärimmäisen yksinkertainen eikä vielä tee minkäänlaista käsittelyä sivuille tai mitään sivutusta tai muistinhallintaa. Fyysinen muistiavaruus antaa sivuja kun niitä siltä kysytään niin kauan kuin niitä riittää ja ottaa vastaan sivut kun ne palautetaan. Lisäksi siltä voi kysellä vapaiden sivujen määrää.

6.3.3 Sivuvälimuistihoitaja

Toteutuksen keskeisin osa on sivuvälimuistihoitaja, jonka tehtävänä on hallinnoida asiakkaiden tilejä, ottaa vastaan muistipyynnöt ja vuokrata muistia järjestelmän toimintalinjan mukaisesti. Sivuvälimuistihoitaja toteuttaa asiakkaille kutsurajapinnan jolla asiakkaat voivat luoda tilejä, kysellä niiden saldoa ja tulojaan, sekä siirtää rahaa tai osan tuloistaan toiselle tilille. Asiakkaat voivat myös kysellä muistin hintaa ja muistia

odottavien pyyntöjen määrää. Ja tottakai asiakkaat voivat myös pyytää muistia, jatkaa olemassaolevan vuokrasopimuksen vuokra-aikaa ja sanoa irti vuokrasopimuksensa.

Tilit luodaan `spcm-create-account`-kutsulla, joka luo asiakkaalle tilin ja palauttaa tilin tunnuksen, jonka kautta tiliä voi käsitellä muilla sivuvälimuistihoitajan kutsuilla. Hartyn ja Cheritonin kuvaamasta mallista poiketen uudet tilit luodaan täysin ilman rahaa tai tuloja. Tarkoituksena on että järjestelmään luodaan aluksi yksi päätili, jonka tulot yhdessä aikajaksossa ovat riittävät kaiken muistin varaamiseksi siksi aikajaksoksi. Käyttäjille luodaan sen jälkeen tilejä, joille siirretään tuloja päätililtä, ja käyttäjät voivat itse luoda itselleen alitilejä, joille niiden on siirrettävä tuloja tai rahaa joltain toiselta omalta tililtään.

Muistia varataan `spcm-request-memory`-kutsulla. Kutsu koostuu vuokrausta rahoittavan tilin tunnuksesta, haluttujen sivujen vähimmäismäärästä ja enimmäismäärästä, vuokra-ajasta ja vastakutsusta, jota kutsutaan pyynnön tultua käsitellyksi, jos pyyntöä ei voida käsitellä saman tien. Pyynnön saadessaan sivuvälimuistihoitaja ensin tarkistaa onko muita pyyntöjä odottamassa muistia ja jos ei, onko sivuja vapaana tarpeeksi pyynnön täyttämiseksi saman tien. Jos pyyntö voidaan täyttää, tarkistetaan onko annetulla tilillä tarpeeksi rahaa pyynnön maksamiseksi ja varojen riittäessä pyyntö täytetään saman tien. Jos pyyntöä ei voida täyttää, se siirretään pyyntöjonoon odottamaan vuoroaan, tai jos pyynnön täyttämiseen ei ole varaa, siitä ilmoitetaan asiakkaalle ja pyyntö hylätään suoraan. Kaikella muistilla on vakiohinta, eikä eri tärkeyksiä tai maksutonta muistia muistin ollessa käyttämättömänä ole toteutettu.

Kun asiakas haluaa jatkaa olemassa olevaa vuokrasopimusta alkuperäisen vuokra-ajan yli, sivuvälimuistihoitajalle lähetetään `spcm-extend-lease`-kutsu ja siinä ilmoitetaan jatkoajalle tarvittavien sivujen määrä ja haluttu jatkoaika. Jatkoajalle pyydettävien sivujen määrä ei saa ylittää alkuperäiseen sopimukseen kuuluvien sivujen määrää. Kun alkuperäinen vuokra-aika päättyy, sivuvälimuistihoitaja tarkistaa onko sopimukseen merkattu jatko-aikaa ja onko asiakas kykeneväinen maksamaan siitä, jos on, aikaa jatketaan, ja tarvittaessa ylimääräiset sivut, joita ei haluttu jatkoajalle, palautetaan.

Asiakas voi myös lopettaa vuokrasopimuksen kesken vuokra-ajan, jolloin hänelle palautetaan maksu vuokra-ajan lopulta. (Asiakkaiden tileiltä veloitetaan heti vuokrauksen alkaessa koko vuokra-ajan maksu.) Vuokrasopimuksen keskeyttäminen tehdään `spcm-release-lease`-kutsulla, jolle annetaan vuokrasopimuksen tunnus.

6.3.4 Toteutuksen testaus

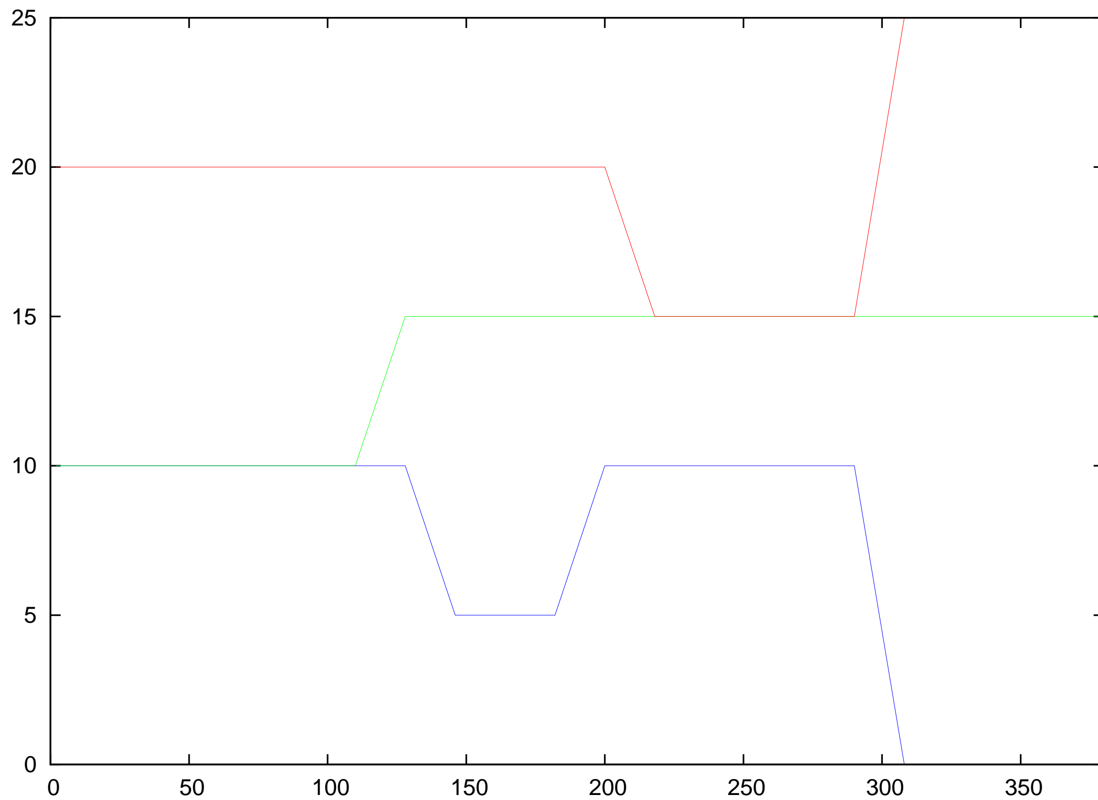
Testasin toteutustani luomalla kolme toimijaa, jotka säännöllisin väliajoin joko vuokrasivat uutta muistia tai jatkoivat vanhan vuokrauksen vuokra-aikaa sen mukaan, miten

paljon niillä oli käytettävissään rahaa muistin vuokraamiseksi. Toimijat eivät käyttäneet muistia mihinkään, vaan tehtyään varauksen tai vuokra-ajan jatkon, ne asettivat vain ajastimen ilmaisemaan kun vuokra-aika (tai sen jatkoaika) oli loppumaisillaan ja ajastimen ilmoittaessa vuokrasivat lisää muistia tai jatkoivat vanhan vuokran vuokra-aikaa. Ennalta määrääminäni hetkinä muutin toimijoille osoitetun rahan määrää ja seurasin miten se vaikutti toimijoille osoitetun muistin määrään.

Toimijoita oli kolme, joista ensimmäisen ja toisen tuloiksi asetettiin 10 ja kolmannen tuloiksi 20. Kaikille kolmelle asetettiin tilille kaksi kertaa tulojen verran rahaa, jotta ne pystyivät vuokraamaan muistia heti joutumatta ensin odottamaan rahojen karttumista tileilleen. Toimijat vuokrasivat aluksi tulojensa verran sivuja kahdeksi aikayksiköksi, ensimmäinen ja toinen 10 sivua ja kolmas 20 sivua, ja asettivat ajastimen ilmoittamaan yhden aikayksikön päästä tarpeesta lisätoimenpiteisiin. Ajastinsignaalin saatuaan toimijat tarkistivat varallisuutensa ja sillä saatavien sivujen määrän. Jos saatavien sivujen määrä oli pysynyt samana tai laskenut edellisestä kerrasta, toimija saattoi jatkaa edellisen vuokrasopimuksen vuokra-aikaa jälleen kahdella aikayksiköllä, jos taas rahat riittivät suurempaan määrään sivuja, toimija teki pyynnön uudesta vuokrasopimuksesta.

Ensimmäisessä muutoksessa, viiden aikayksikön kuluttua kokeen alusta, siirsin toimijan numero yksi tuloista viisi yksikköä toimijalle numero kaksi. Toisessa muutoksessa, kymmenen aikayksikön päästä alusta, siirsin toimijan numero kolme tuloista viisi yksikköä toimijalle numero yksi. Viimeisessä muutoksessa, viidentoista aikayksikön jälkeen kokeen alusta, siirsin toimijan numero yksi tuloista kymmenen yksikköä toimijalle kolme. Viimeisen muutoksen jälkeen toimijoiden tulot olivat seuraavasti: ensimmäisellä ei ollut tuloja ollenkaan, toisen tulot olivat viisitoista ja kolmannen toimijan tulot olivat kaksikymmentä viisi. Muutokset toimijoiden tuloissa näkyivät niille osoitettujen sivujen määrissä odotetusti (kts. kuva 6.1).

Yksinkertainen kokeeni mielestäni osoitti vähintäänkin että yksinkertaisille toimijoille osoitettavilla tuloilla voidaan ohjailla hyvin niille osoitettavan muistin määrää. Mielenkiintoisia kysymyksiä herättääkin miten suuremman älykkyyden osaavat toimijat osaisivat hyödyntää niiden käytössä olevia varoja tarvittavien muistiresurssien hankkimiseksi.



Kuva 6.1: Kolmelle toimijalle osoitettu muistin määrä vaihtelee niille osoitetun rahan muutosten mukaan. Käyrät kuvaavat toimijoille osoitettujen sivujen määrää. Pystyakseli on sivujen määrä ja vaaka-akseli aika.

7 Kysymyksiä ja jatkopohdintaa

Tutkimukseni on esitellyt miten markkinapohjaisilla malleilla voitaisiin tuoda kaivattua lisä joustavuutta käyttöjärjestelmien muistinhallintaan. Markkinapohjaiset mallit tarjoavat myös suoraviivaisen tavan vaikuttaa käyttäjän sovelluksien käytössä olevan keskusmuistin määrään. Esitys on kuitenkin jättänyt joitain epäselvyyksiä ja herättänyt joitain kiinnostavia kysymyksiä tulevaisuuden tutkimusta varten.

Yksi keskeisimmistä kysymyksistä, joka herää minun mielessäni kun luin artikkeleitä TENEXille toteutetusta markkinapohjaisesta muistinhallinnasta ja Hartyn ja Cheritonin kuvailemasta mallista, koskee fyysisten sivujen kuvaamista prosessin muistiavaruuteen. Miten kuvaukset tehdään, ja voivatko prosessit vaikuttaa niihin mitenkään? Kuvauksiin vaikuttaminen on kuitenkin oleellista sovelluksien mahdollisuuksille vaikuttaa omaan muistin käyttöönsä ja sopetua käytettävissä olevaan muistin määrään. Sovellukset saattaisivat esimerkiksi haluta pitää yllä pientä määrää sivuja tärkeätä koodia ja dataa varten, mutta vuokrata ylimääräisiä sivuja joidenkin toimintojen nopeuttamiseksi kun varoja on yli välttämättömän tarpeen.

Utahin yliopiston TENEX järjestelmässä kuvaukset ilmeisesti tehdään automaattisesti prosessien voimatta vaikuttaa niihin mitenkään. Markkinamalli ei anna prosesseille sen enempää päätäntävaltaa koskien niille osoitettuja muistisivuja, vaan ottaa ainoaksi tehtäväkseen muistin (ja prosessoriajan) reilun ja tehokkaan jakamisen prosessien kesken. Harty ja Cheriton puolestaan kuvailevat erillisessä artikkelissaan operaatiot joilla prosessit voisivat käsitellä käytössään olevaa fyysistä muistia ja sen kuvaamista omaan muistiavaruuteensa.

Toinen kysymys, joka tuli mieleeni etenkin omaa toteutustani suunnitellessa oli, miten prosessi voi pyytää lisää muistia, jos sen rahat on loppu, eikä sillä siten ole varaa vuokrata edes sen vertaa muistia että se voisi tehdä pyynnön lisä muistista? Hartyn ja Cheritonin mallissa tähän lienee vastattu nollahintaisella matalan tärkeyden muistilla, mutta en ole täysin vakuuttunut, että se voisi ratkaista ongelman tyydyttävästi. Jos prosessi joutuu odottamaan ettei muistille ole muuta käyttöä päästäkseen jatkamaan suoritustaan, se todennäköisesti joutuisi odottamaan hyvin kauan. Ainoana edes lähes tyydyttävänä ratkaisuna näkisin prosessin muistipyyntöjen säilyttämisen senkin jälkeen kun sillä ei ole varaa niitä maksaa. Rahaa odottavat pyynnöt käytäisiin läpi säännöllisin väliajoin, ja tarkistettaisiin olisiko niiden tileille kertynyt riittävä määrä rahaa pyynnön täyttämiseksi. Ongelmana näkisin tässä sen, että ilman rahaa olevat prosessit voivat

pyyntöjä esittämällä kuormittaa muistinhallintajärjestelmää turhaan.

Yksi kaikkien lukemieni tekstien taustalla oleva oletus on tietokonejärjestelmien resurssimarkkinoiden vastaavan ihmistalouden hyödykemarkkinoita, missä prosessit toimivat hyödykkeiden kuluttajina ja järjestelmän resurssienhoitajat tuottajina. Huomasin kuitenkin asiaan tutustuessani olevani taipuvaisempi vertaamaan resurssimarkkinoita ennemminkin ihmistalouksien tuotannontekijämarkkinoihin. Järjestelmän resurssienhoitajille sopisi paremmin mielestäni kotitalouksien rooli. Resurssienhoitajat omistavat tiettyjä resursseja, joita ne koittavat saada myytyä markkinoilla mahdollisimman suuren tulon saamiseksi, eikä niiden tarvitse miettiä resursseista saatavaa voittoa tulojen ja kulujen erotuksena. Mitä suuremmat tulot resurssienhoitajat resursseistaan saavat, sitä suurempi on niiden tarpeen tyydytys. Prosesseille puolestaan sopii paremmin yritysten rooli tuotannontekijöiden ostajina, joille on tärkeätä resurssien hinnan lisäksi niiden tehokas käyttö lopputuloksen tuottamisessa. Käyttäjät puolestaan ostavat prosessien toiminnan lopputuloksen ja maksavat niille siitä rahalla, jonka prosessit saavat käyttöönsä lisäresurssien ostamiseen.

Mielenkiintoisena kysymyksenä pidän myös markkinapohjaisen mallin laajentamista muihin tietokonejärjestelmän resursseihin. Utahin yliopiston TENEX-järjestelmä yhdistää samassa markkinamallissa muistin ja prosessoriajan. Mutta miten pitkälle resurssimarkkoinoita voitaisiin viedä? Verkkoliikenteen kaistanleveys ja kiintolevy tila tuntuvat sopivan mielikuvissani suhteellisen kivuttomasti samaan resurssimarkkinoiden malliin, mutta mitenkä kävisi esimerkiksi näytönohjaimen ja monitorin käytön siirtyessä vapaille markkinoille? Miller ja Drexler kuvailevat artikkelissaan[12] mallia, jossa kaikki resurssit on siirretty markkinoiden ohjattavaksi, ja lisäksi markkinoilla liikkuu erilaisia johdannaisia, joilla erilaiset toimijat voivat edistää toimintaansa.

Voitaisiinko jopa kokonainen käyttöjärjestelmä toteuttaa perusteistaan alkaen resurssimarkkinoina? Järjestelmän perusabstraktioina toimisivat toimijat ja niiden väliset sopimukset, ja ytimen tehtäväksi jäisi ainoastaan omistusoikeuksien ja niitä kuvaavien sopimusten turvaaminen ja markkinapaikan tarjoaminen toimijoiden kaupankäynnin helpottamiseksi. Itseasiassa markkinapaikat, eli keinot resurssien myyjien ja ostajien kohdata toisensa, voitaisiin todennäköisesti toteuttaa myös maksullisena palveluna.

Monimutkaisten markkinajärjestelmien lisäksi mielenkiintoista pohdittavaa olisivat markkinoiden toimijoiden erilaiset toimintastrategiat ja niiden omaavan "älykkyyden" taso. Minkälaisilla kaupankäyntistrategioilla resurssien hoitajat voisivat saada resursseistaan mahdollisimman suuret tulot? Entä minkälaisilla kaupankäyntistrategioilla prosessit voisivat saada tarvitsemansa resurssit käyttöönsä mahdollisimman halvalla? Tietysti nämä kysymykset ovat toisiinsa vahvasti kytköksissä. Resurssien hinnan

määritykseen ja niiden hoitajien kaupankäyntistrategiaan vaikuttaa varmasti resursseja ostavien prosessien strategiat, ja toisinpäin. Entä missä vaiheessa toimijoiden monimutkaisten strategioiden toteutusten vaatimat lisäresurssit ylittäisivät niistä saatavan lisähyödyn? Olisiko markkinoiden toiminnan kannalta kuitenkin parasta, että toimijat olisivat mahdollisimman yksinkertaisia?

8 Viitteet

[1] David Begg, Stanley Fischer, and Rudiger Dombusch. *Economics*. McGraw-Hill Publishing Co., 2000.

[2] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the Usenix Annual Technical Conference*, pages 117–130, June 1999.

[3] K. Eric Drexler and Mark S. Miller. Incentive engineering for computational resource management. In B. Huberman, editor, *The Ecology of Computation*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1988.

[4] C. M. Ellison. The Utah TENEX scheduler. In *Proceedings of the IEEE, Vol.63, Iss.6, June 1975*, pages 940–945, 1975.

[5] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.

[6] Frode V. Fjeld. The Movitz development platform, 2004. Available at http://common-lisp.net/project/movitz/files/movitz.pdf.

[7] Frode Vatvedt Fjeld. Movitz: A Common Lisp OS development platform, 2004. Available at http://80.68.86.115/project/movitz/movitz.html.

[8] Dhananjay K Gode and Shyam Sunder. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of Political Economy*, 101(1):119–37, February 1993. Available at http://ideas.repec.org/a/ucp/jpolec/v101y1993i1p119-37.html.

[9] Kieran Harty and David Cheriton. A market approach to operating system memory allocation. In Scott H. Clearwater, editor, *Market-based control: a paradigm for distributed resource allocation*, pages 126–155. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[10] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997. Available at http://developer.intel.com/design/pentium/manuals/24319001.pdf.

[11] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1999. Available at ftp://download.intel.com/design/PentiumII/manuals/24319202.pdf.

[12] Mark S. Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In B. Huberman, editor, *The Ecology of Computation*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1988.

[13] Jukka Pekkarinen and Pekka Sutela. *Kansantaloustiede*. WSOY, 2002.

[14] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[15] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, pages 31–39, Palo Alto, California, 1987.

[16] Abraham Siberschatz and Peter B. Galvin. *Operating System Concepts, 4th Ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[17] Abraham Siberschatz and Peter B. Galvin. *Operating System Concepts, 5th Ed.* Addison-Wesley Publishing Company, Boston, MA, USA, 1998.

[18] Neil Stratford and Richard Mortier. An economic approach to adaptive resource management. In *Workshop on Hot Topics in Operating Systems*, pages 142–147, 1999.

[19] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1987.

[20] P. Tucker and F. Berman. On market mechanisms as a software technique. Technical Report TR CS96-513, U. C. San Diego, 1996.

[21] Michael P. Wellman. Market-oriented programming: Some early lessons. In Scott Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, River Edge, New Jersey, 1996.

A Lähdekoodi

```
(in-package los0)

;; PHYSICAL MEMORY SPACE
;;
;; The physical memory space keeps a list of the pages of
;; physical memory in the system and how many of them are
;; free and how many in use.

;; A single page of physical memory
(defclass phys-mem-page ()
  ((address :initarg :addr ; the physical address of the page
            :reader phys-page-addr)))

;; Three lists of phys-mem-pages: one for all the pages in the
;; system, one for the pages that are free, and one for pages in use
(defclass phys-mem-space ()
  ((page-list :accessor phys-space-page-list
              :initform nil)
   (free-page-list :accessor phys-space-free-list
                   :initform nil)
   (reserved-page-list :accessor phys-space-res-list
                       :initform nil)
   ))

;; Create and initialize a phys-mem-space
(defun init-phys-mem-space ()
  (let* ((phys-mem-size (memory-size))
         (phys-mem-pages (* phys-mem-size #x100))
         (phys-mem-space (make-instance 'phys-mem-space)))
    (do* ((i 0 (1+ i))
          ; create a phys-mem-page for the address (* i #x1000)
          (page (make-instance 'phys-mem-page :addr (* i #x1000))
                (make-instance 'phys-mem-page :addr (* i #x1000))))
        ((>= i phys-mem-pages) phys-mem-space)
      ; add the page to the page list and the free page list
      (push page (phys-space-page-list phys-mem-space))
      (push page (phys-space-free-list phys-mem-space))
      )
    ))


(defvar *phys-mem-space* nil)

;; Return the phys-mem-space, or create one if needed
(defun phys-mem-space ()
  (if *phys-mem-space*
    *phys-mem-space*
    (setq *phys-mem-space* (init-phys-mem-space))))

;; phys-space-free-pages
;; Return the number of free pages
(defmethod phys-space-free-pages ((pspace phys-mem-space))
  (list-length (phys-space-free-list pspace)))

;; phys-space-get-pages
;; Reserve a number of physical pages
;;  - n : number of pages to reserve
;;  - :wait : whether or not to wait for pages if there isn't
;;            enough available (not implemented)
;;  - returns a list of n page addresses or nil if pages couldn't
;;    be allocated
(defmethod phys-space-get-pages ((pspace phys-mem-space) n
                                 &key wait)
  (let ((pages-free (list-length (phys-space-free-list pspace)))
        (plist nil))
    (if (>= pages-free n)
```

```lisp
    (dotimes (i n plist)
      ; pop a page from the free list and push it on the reserved
      ; list
      (let ((page (pop (phys-space-free-list pspace))))
        (push (phys-page-addr page) plist)
        (push page (phys-space-res-list pspace))))
    (when wait
      (error "wait flag not implemented for phys-space-get-pages")))
  plist))

;; phys-space-put-pages
;; Free a group of physical pages
;;  - pages : a list of the page addresses of the pages to free
;;  - returns the number of pages freed
(defmethod phys-space-put-pages ((pspace phys-mem-space) pages)
  (let ((n 0))
    ; find the freed pages in the reserved list and move them onto
    ; the free list
    (dolist (p pages n)
      (let ((page (find-if (lambda (x) (eql (phys-page-addr x) p))
                           (phys-space-res-list pspace))))
        (when page
          (setf (phys-space-res-list pspace)
                (delete page (phys-space-res-list pspace)))
          (push page (phys-space-free-list pspace))
          (incf n))))
    n))


;; CLOCK HANDLER
;;
;; The clock handler follows the ticks of the system clock and lets
;; programs set alarms


(defclass clock-handler ()
  ((callout-queue :initform nil
                  :accessor clock-handler-queue)
   (current-ticks :initform 0
                  :accessor clock-handler-ticks)))

;; clock-handler-add-callout
;; Add a new alarm
;; - fn : the function to call when the alarm goes off
;; - delta : the number of ticks after which the alarm should go off
;; - arg : an optional arg that is sent to the callout function when
;;    it is called
;; - returns an id for the alarm
(defmethod clock-handler-add-callout ((ch clock-handler) fn delta &optional arg)
  ; create a new callout and insert it into the callout list (sorted
  ; by calltime)
  (let* ((calltime (+ (clock-handler-ticks ch) delta))
         (co (make-instance 'callout
                            :fn fn
                            :arg arg
                            :calltime calltime))
         (predicate (lambda (co1 co2) (< (callout-calltime co1) (callout-calltime co2)))))
    (setf (clock-handler-queue ch)
          (merge 'list (clock-handler-queue ch) (cons co nil) predicate))
    (callout-id co)))

;; clock-handler-remove-callout
;; Remove an alarm
;; - callout-id : the id of the alarm to remove
;; - returns t
(defmethod clock-handler-remove-callout ((ch clock-handler) callout-id)
  (delete-if (lambda (x) (eql (callout-id x) callout-id)) (clock-handler-queue ch))
  t)

;; Start the clock handler
(defmethod clock-handler-start ((ch clock-handler))
  (setf (exception-handler 32) ; set the clock interrupt handler
        (lambda (exception-vector exception-frame)
          (declare (ignore exception-vector exception-frame))
          (let ((ticks (incf (clock-handler-ticks ch))))
            ; go through the list of callouts and call those that
```

```
            ; are due
            (dolist (co (clock-handler-queue ch) t)
              (if (>= ticks (callout-calltime co))
                (callout-call (pop (clock-handler-queue ch)) ticks)
                (return nil))))
          (pic8259-end-of-interrupt 0)))
  (setf (pic8259-irq-mask) #xfffe)
  (pic8259-end-of-interrupt 0)
  (with-inline-assembly (:returns :nothing) (:sti)))

;; Create a clock handler and start it up
(defun start-clock-handler ()
  (let ((ch (make-instance 'clock-handler)))
    (clock-handler-start ch)
    ch))

(defvar *callout-idcount* 0)

;; Clock handler callout function
(defclass callout ()
  ((id :initform (incf *callout-idcount*)
       :reader callout-id)
   (fn :initarg :fn
       :reader callout-fn)
   (arg :initarg :arg
        :initform nil
        :reader callout-arg)
   (calltime :initarg :calltime
             :reader callout-calltime)))

;; Call a callout
(defmethod callout-call ((co callout) ticks)
  (if (callout-arg co)
    (funcall (callout-fn co) ticks (callout-arg co))
    (funcall (callout-fn co) ticks)))

;;; MEMORY ACCOUNT
;; "A memory account is used to record the share of the system memory
;; resources that is allocated to a particular segment manager.
;; A memory account may be associated with an individual user, a set
;; of users or a subsystem such as a database management system."

(defvar *memory-account-idcount* 0)

(defconstant *memory-account-initial-balance* 0)
(defconstant *memory-account-initial-income* 0)
(defconstant *memory-account-max-balance* 1280)
; 1280 max balance = 5M for 1 time quantum

(defclass memory-account ()
  ((id :initform (incf *memory-account-idcount*)
       :reader memory-account-id) ;; account id
   (balance :initform *memory-account-initial-balance*
            :accessor memory-account-balance) ;; current balance
   (income :initform *memory-account-initial-income*
           :accessor memory-account-income) ;; share of system memory money
   ))

;; Increment the account balance by its income
(defmethod memory-account-increment ((acnt memory-account))
    (incf (memory-account-balance acnt)
          (min (memory-account-income acnt)
               (- *memory-account-max-balance* (memory-account-balance acnt)))))

;; Charge the given amount from the account
(defmethod memory-account-charge ((acnt memory-account) amount)
  (unless (<= (memory-account-balance acnt) 0)
    (decf (memory-account-balance acnt) amount)))

;; Return an amount of money to the account
(defmethod memory-account-refund ((acnt memory-account) amount)
    (incf (memory-account-balance acnt)
          (min amount
               (- *memory-account-max-balance* (memory-account-balance acnt)))))

;;; MEMORY REQUEST
```

```lisp
;; A request for memory.

(defclass memory-request ()
  ((account-id :initarg :account-id ; id of account funding the request
               :reader memory-request-account-id)
   (minpages :initarg :minpages ; min pages of request
             :reader memory-request-minpages)
   (maxpages :initarg :maxpages ; max pages of request
             :reader memory-request-maxpages)
   (leasetime :initarg :leasetime ; requested time for lease
              :reader memory-request-leasetime)
   (callback :initarg :callback ; function to call when request is fulfilled
             :reader memory-request-callback)
   ))

;;; MEMORY LEASE
;; A memory lease.

(defvar *memory-lease-idcount* 0)

(defclass memory-lease ()
  ((id :initform (incf *memory-lease-idcount*)
       :reader memory-lease-id) ;; lease id
   (account-id :initarg :account-id ; account funding the lease
               :reader memory-lease-account-id)
   (pages :initarg :pages
          :accessor memory-lease-pages) ;; pages associated with lease
   (lease-start :initarg :lease-start
                :accessor memory-lease-lease-start) ;; lease start time
   (lease-end :initarg :lease-end
              :accessor memory-lease-lease-end) ;; lease end time
;   (priority :initform nil
;             :accessor memory-lease-priority) ;; lease priority
   (ext-time :initform nil
             :accessor memory-lease-ext-time) ;; a pending lease extension
   (ext-pages :initform nil
              :accessor memory-lease-ext-pages)
   ))

;; Remove a number of pages from a lease
(defmethod memory-lease-revoke-pages ((lease memory-lease) npages)
  (let ((ret nil)
        (tmp nil))
    (dotimes (n npages ret)
      (if (setq tmp (pop (memory-lease-pages lease)))
        (push tmp ret)))
    ))


;;; SYSTEM PAGE-CACHE MANAGER
;; "The system page-cache manager."

(defclass system-page-cache-manager ()
  ((accounts :accessor spcm-accounts ;; account list
             :initform (make-hash-table))
   (sysaccount :accessor spcm-sysaccount ;; the system memory account
               :initform (make-instance 'memory-account))
   (leases :accessor spcm-leases
           :initform nil) ;; lease list
   (requests :accessor spcm-requests
             :initform nil) ;; request list
   (pspace :accessor spcm-pspace
           :initform (phys-mem-space))
   (clock :accessor spcm-clock
          :initform (start-clock-handler))
   (alarmid :accessor spcm-alrmid
            :initform 0)
   ; the price of a page of memory for one time quantum
   (price :reader spcm-price
          :initarg :price
          :initform 1)
   ; the length of a single time quantum in clock ticks
   (quantum :reader spcm-quantum
            :initarg :quantum
            :initform 18)
   ))
```

```
;; spcm init function that creates a clock callout routine for
;; incrementing the memory account balances
;; also initialize system memory account
(defmethod spcm-init ((spcm system-page-cache-manager))
  (clock-handler-add-callout (spcm-clock spcm) #'spcm-account-callout (spcm-quantum spcm) spcm)
  (setf (memory-account-income (spcm-sysaccount spcm))
        (* (spcm-price spcm) (list-length (phys-space-page-list (spcm-pspace spcm)))))
  t
  )

;; clock handler callouts for accounts and leases

;; Account callout. Increments memory accounts at set intervals
(defmethod spcm-account-callout (ticks (spcm system-page-cache-manager))
  (maphash
    (lambda (k v) (memory-account-increment v))
    (spcm-accounts spcm))
  (clock-handler-add-callout (spcm-clock spcm) #'spcm-account-callout (spcm-quantum spcm) spcm)
  )

;; Lease callout. Check for expired leases and revoke or extend them.
(defmethod spcm-lease-callout (ticks (spcm system-page-cache-manager))
  (let ((freed-pages nil)
        (reset-clock nil)
        (ext-leases nil))
    ; go through the list of leases sorted by end time
    (dolist (lease (spcm-leases spcm))
      ; when the first lease with end time in the future is reached,
      ; stop
      (when (> (memory-lease-lease-end lease) ticks)
        (return))
      (setq reset-clock t)
      (pop (spcm-leases spcm))
      (if (memory-lease-ext-time lease)
        ; the lease has a pending extension, handle that
        (let ((account (spcm-find-account spcm (memory-lease-account-id lease)))
              (extension-cost (* (memory-lease-ext-time lease)
                                 (memory-lease-ext-pages lease)
                                 (spcm-price spcm))))
          (if (>= (memory-account-balance account) extension-cost)
            ; the extension is funded
            (progn
              (setf (memory-lease-lease-end lease)
                    (+ (memory-lease-lease-end lease)
                       (* (spcm-quantum spcm) (memory-lease-ext-time lease))))
              (if (/= 0 (memory-lease-ext-pages lease))
                ; if the extension is for fewer pages than the lease
                ; free the extra pages
                (setf freed-pages
                      (memory-lease-revoke-pages lease
                                                 (- (list-length (memory-lease-pages lease))
                                                    (memory-lease-ext-pages lease)))))
              (print (list ticks "lease for account" (memory-lease-account-id lease)
                           "extended; pages:" (list-length (memory-lease-pages lease))
                           "time:" (memory-lease-ext-time lease)))
              (setf (memory-lease-ext-time lease) nil)
              (setf (memory-lease-ext-pages lease) nil)
              (memory-account-charge account extension-cost)
              (setq ext-leases (append ext-leases (list lease)))
              )
            ; extension doesn't have enough funding
            (print (list ticks "can't afford extension. b, i:"
                         (memory-account-balance account)
                         (memory-account-income account)))))
        ; the lease has no pending extension so it can be freed
        (progn
          (setf freed-pages (append freed-pages (memory-lease-pages lease)))
          (print (list ticks "terminated lease from account"
                       (memory-lease-account-id lease) "freeing"
                       (list-length (memory-lease-pages lease)) "pages"))
          )
        ))
    ; free freed pages
    (spcm-free-pages spcm freed-pages)
    ; insert extended leases in their places in the lease list
```

```lisp
    (mapcar (lambda (x) (spcm-insert-lease spcm x)) ext-leases)
    ; reset lease clock if necessary
    (when reset-clock
      (spcm-reset-lease-clock spcm))
    ))

;; Reset the lease clock handler callout
(defmethod spcm-reset-lease-clock ((spcm system-page-cache-manager))
  (if (spcm-leases spcm)
    (let ((delta (- (memory-lease-lease-end (first (spcm-leases spcm)))
                    (clock-handler-ticks (spcm-clock spcm)))))
      (clock-handler-remove-callout (spcm-clock spcm) (spcm-alrmid spcm))
      (setf (spcm-alrmid spcm)
            (clock-handler-add-callout (spcm-clock spcm) #'spcm-lease-callout delta spcm))
      ))
  )

;;; SPCM PRIVATE METHODS

;; Insert a new account in its place
(defmethod spcm-insert-account ((spcm system-page-cache-manager) account)
  (setf (gethash (memory-account-id account) (spcm-accounts spcm)) account))

;; Find an account by its id
(defmethod spcm-find-account ((spcm system-page-cache-manager) acnt-id)
  (gethash acnt-id (spcm-accounts spcm)))

;; Create a new memory lease
(defmethod spcm-create-lease ((spcm system-page-cache-manager) acnt-id
                                                                 pages
                                                                 lease-start
                                                                 lease-end)
  (let ((lease (make-instance 'memory-lease :account-id acnt-id
                                            :pages pages
                                            :lease-start lease-start
                                            :lease-end lease-end)))
    (spcm-insert-lease spcm lease)
    (print (list (clock-handler-ticks (spcm-clock spcm))
                 "created lease for account" acnt-id
                 "for" (list-length pages) "pages"))
    (spcm-reset-lease-clock spcm)
    (memory-lease-id lease)))

;; Insert a lease in its place
(defmethod spcm-insert-lease ((spcm system-page-cache-manager) lease)
  (let ((pos 0))
    (labels ((insert (l llist)
                (if (or (endp llist)
                        (> (memory-lease-lease-end (first llist)) (memory-lease-lease-end l)))
                  (cons l llist)
                  (progn
                    (incf pos)
                    (cons (car llist) (insert l (cdr llist)))))))
      (setf (spcm-leases spcm) (insert lease (spcm-leases spcm))))
    pos
  ))

;; Find a lease by its id
(defmethod spcm-find-lease ((spcm system-page-cache-manager) lease-id)
  (find-if (lambda (x) (= lease-id (memory-lease-id x))) (spcm-leases spcm)))

;; Remove a lease (no clean up)
(defmethod spcm-remove-lease ((spcm system-page-cache-manager) lease)
  (remove lease (spcm-leases spcm))
  t)

;; Handle a request for memory. Check for required funds and create a
;; lease if request can be funded.
(defmethod spcm-handle-request ((spcm system-page-cache-manager) req pages)
  (let* ((start-time (clock-handler-ticks (spcm-clock spcm)))
         (end-time (+ start-time (* (memory-request-leasetime req) (spcm-quantum spcm))))
         (account (spcm-find-account spcm (memory-request-account-id req)))
         (request-cost (* (length pages) (memory-request-leasetime req) (spcm-price spcm))))
    (if (< (memory-account-balance account) request-cost)
      ; if request isn't funded it is discarded
      (progn
```

```
          (spcm-free-pages spcm pages)
          'insufficient-funds)
        ; if the request is funded the request is granted
        (progn
          (memory-account-charge account request-cost)
          (spcm-create-lease spcm (memory-request-account-id req) pages start-time end-time)))))

;; Free a group of pages and check waiting requests if any can be
;; satisfied
(defmethod spcm-free-pages ((spcm system-page-cache-manager) freed-pages)
  (phys-space-put-pages (spcm-pspace spcm) freed-pages)
  (dolist (req (spcm-requests spcm))
    ; go through the list of waiting requests and grant as many as
    ; there are pages available
    (if (> (memory-request-min req) (phys-space-free-pages (spcm-pspace spcm)))
      ; the requests are organized in a fifo, so the oldest request
      ; is always granted first
      (return))
    (pop (spcm-requests spcm))
    (let* ((pages (phys-space-get-pages (spcm-pspace spcm)
                                        (min (memory-request-max req)
                                             (phys-space-free-pages (spcm-pspace spcm)))))
           (ret (spcm-handle-request spcm req pages)))
      (funcall (memory-request-callback req) ret))))


;;; SPCM PUBLIC INTERFACE

;; spcm-create-account
;; Create a new memory account
;; - returns a newly created memory account
(defmethod spcm-create-account ((spcm system-page-cache-manager))
  (let ((acnt (make-instance 'memory-account)))
    (spcm-insert-account spcm acnt)
    (memory-account-id acnt)))

;; spcm-account-balance
;; Query the balance of a memory account
;; - acnt-id : id of the account
;; - return account balance or 'no-such-account if id doesn't match
;;   any account
(defmethod spcm-account-balance ((spcm system-page-cache-manager) acnt-id)
  (let ((account (spcm-find-account spcm acnt-id)))
    (if account
      (memory-account-balance account)
      'no-such-account)))

;; spcm-account-income
;; Query the income of a memory account
;; - acnt-id : id of the account
;; - return account income or 'no-such-account if id doesn't match
;;   any account
(defmethod spcm-account-income ((spcm system-page-cache-manager) acnt-id)
  (let ((account (spcm-find-account spcm acnt-id)))
    (if account
      (memory-account-income account)
      'no-such-account)))

;; spcm-transfer-account
;; Transfer a portion of an accounts income or balance to another
;; account
;; - srcacnt-id : id of the source account
;; - dstacnt-id : id of the destination account
;; - :income : the amount of income to transfer
;; - :balance : the amount of balance to transfer
;; - return t or 'no-such-account if either account id doesn't match
(defmethod spcm-transfer-account ((spcm system-page-cache-manager) srcacnt-id dstacnt-id
                                                                   &key income balance)
  (let ((srcacnt (spcm-find-account spcm srcacnt-id))
        (dstacnt (spcm-find-account spcm dstacnt-id)))
    (if (not (and srcacnt dstacnt))
      'no-such-account
      (progn
        (when income
          (when (>= (memory-account-income srcacnt) income)
            (decf (memory-account-income srcacnt) income)
```

```lisp
            (incf (memory-account-income dstacnt) income)))
        (when balance
          (when (>= (memory-account-balance srcacnt) balance)
            (decf (memory-account-balance srcacnt) balance)
            (incf (memory-account-balance dstacnt) balance)))
        t))
    ))

;; spcm-memory-price
;; Query the price of a page of memory for a single time quantum
;; - return the price of memory
(defmethod spcm-memory-price ((spcm system-page-cache-manager))
  (spcm-price spcm))

;; spcm-request-memory
;; Make a request for memory
;; - account-id : id of the account that is funding the request
;; - minpages : minimum number of pages to accept
;; - maxpages : maximum number of pages needed
;; - leasetime : for how long the pages would be leased
;; - callback : function to call if the request cannot be satisfied
;;   immediately
;; - return : lease id if request could be satisfied
;;            'insufficient-funds if request cannot be funded
;;            'please-wait if the request if left waiting for more
;;                pages to be freed
(defmethod spcm-request-memory ((spcm system-page-cache-manager) account-id
                                                                  minpages
                                                                  maxpages
                                                                  leasetime
                                                                  callback)
  (let ((req (make-instance 'memory-request :account-id account-id
                                            :minpages minpages
                                            :maxpages maxpages
                                            :leasetime leasetime
                                            :callback callback)))
    (if (and (= 0 (list-length (spcm-requests spcm)))
             (>= (phys-space-free-pages (spcm-pspace spcm)) minpages))
      (let ((pages (phys-space-get-pages (spcm-pspace spcm)
                                         (min maxpages (phys-space-free-pages (spcm-pspace spcm))))))
        (spcm-handle-request spcm req pages))
      (progn
        (setf (spcm-requests spcm) (append (spcm-requests spcm) (list req)))
        'please-wait))))

;; spcm-requests-waiting
;; Query the number of requests waiting for memory
;; - return number of waiting requests
(defmethod spcm-requests-waiting ((spcm system-page-cache-manager))
  (list-length (spcm-requests spcm)))

;; spcm-lease-start
;; Query the start time of a lease
;; - lease-id : id of the lease
;; - return lease start time (in clock handler ticks) or
;;   'no-such-lease if lease id cannot be matched to any lease
(defmethod spcm-lease-start ((spcm system-page-cache-manager) lease-id)
  (let ((lease (spcm-find-lease spcm lease-id)))
    (if lease
      (memory-lease-lease-start lease)
      'no-such-lease)))

;; spcm-lease-end
;; Query the end time of a lease
;; - lease-id : id of the lease
;; - return lease end time (in clock handler ticks) or 'no-such-lease
;;   if lease id cannot be matched to any lease
(defmethod spcm-lease-end ((spcm system-page-cache-manager) lease-id)
  (let ((lease (spcm-find-lease spcm lease-id)))
    (if lease
      (memory-lease-lease-end lease)
      'no-such-lease)))

;; spcm-lease-pages
;; Query number of pages in a lease
;; - lease-id : id of the lease
```

```lisp
;; - return number of pages in lease or 'no-such-lease if lease id
;;   cannot be matched to any lease
(defmethod spcm-lease-pages ((spcm system-page-cache-manager) lease-id)
  (let ((lease (spcm-find-lease spcm lease-id)))
    (if lease
      (list-length (memory-lease-pages lease))
      'no-such-lease)))

;; spcm-lease-account-id
;; Query the account funding a lease
;; - lease-id : id of the lease
;; - return the id of the account associated with lease or
;;   'no-such-lease if lease id cannot be matched to any lease
(defmethod spcm-lease-account-id ((spcm system-page-cache-manager) lease-id)
  (let ((lease (spcm-find-lease spcm lease-id)))
    (if lease
      (memory-lease-account-id lease)
      'no-such-lease)))

;; spcm-extend-lease
;; Extend a memory lease
;; - lease-id : id of the lease
;; - npages : pages needed for extension
;; - leasetime : extension time
;; - return : lease-id if extension is succesful
;;            'too-many-pages if trying to request more pages than
;;              in the original lease
;;            'no-such-lease if lease id cannot be matched to any
;;              lease
(defmethod spcm-extend-lease ((spcm system-page-cache-manager) lease-id npages leasetime)
  (let ((lease (spcm-find-lease spcm lease-id))
        (ret nil))
    (cond
      ((<= npages 0)
       (setq ret 'too-few-pages))
      (lease
        (when (< npages (list-length (memory-lease-pages lease)))
          (setf (memory-lease-ext-time lease) leasetime)
          (setf (memory-lease-ext-pages lease) npages)
          (setq ret lease-id))
        (when (= npages (list-length (memory-lease-pages lease)))
          (setf (memory-lease-ext-time lease) leasetime)
          (setf (memory-lease-ext-pages lease) 0)
          (setq ret lease-id))
        (when (> npages (list-length (memory-lease-pages lease)))
          (setq ret 'too-many-pages)))
      ((not lease)
       (setq ret 'no-such-lease)))
    ret))

;; spcm-release-lease
;; Release a memory lease
;; - lease-id : id of the lease
;; - return t
(defmethod spcm-release-lease ((spcm system-page-cache-manager) lease-id)
  (let* ((lease (spcm-find-lease spcm lease-id))
          (pages (memory-lease-pages lease))
          (remaining-time (/ (- (memory-lease-lease-end lease)
                                (clock-handler-ticks (spcm-clock spcm)))
                             (spcm-quantum spcm)))
         (account (spcm-find-account spcm (memory-lease-account-id lease))))
    (spcm-remove-lease spcm lease)
    (memory-account-refund account (* remaining-time (list-length pages) (spcm-price spcm)))
    (spcm-free-pages spcm pages)
    (spcm-reset-lease-clock spcm))
  t)


;; TESTING

(defun make-account (spcm balance income)
  (let* ((account-id (spcm-create-account spcm))
          (account (spcm-find-account spcm account-id)))
    (setf (memory-account-balance account) balance)
    (setf (memory-account-income account) income)
    account-id))
```

```
;; tester
(defun tester-callout (ticks spcm-lease-account-id)
  (let* ((spcm (first spcm-lease-account-id))
         (lease-id (second spcm-lease-account-id))
         (account-id (third spcm-lease-account-id))
         (pages (min (spcm-account-balance spcm account-id)
                     (spcm-account-income spcm account-id)))
         (prev-pages (spcm-lease-pages spcm lease-id))
         (clock (spcm-clock spcm))
         (ret nil)
         (k 2)
         )
    (when (= 0 pages)
      (print (list ticks "account" account-id "out_of_money")))
    (when (eq prev-pages 'no-such-lease)
      (print (list ticks "missing_lease:" lease-id))
      (setq prev-pages 0))
    (if (> pages prev-pages)
      (progn
        (setq ret (spcm-request-memory spcm account-id pages pages 2
                                       (lambda (x) (print "something_went_wrong"))))
        (setq k 1))
      (progn
        (setq ret (spcm-extend-lease spcm lease-id pages 2))
        (setq k 2)))
    (case ret
      ('please-wait
       (print "please_wait"))
      ('insufficient-funds
       (print "insufficient_funds"))
      ('too-many-pages
       (print "too_many_pages"))
      ('no-such-lease
       (print "no_such_lease"))
      (otherwise
        (let ((delta (* k (spcm-quantum spcm))))
          (clock-handler-add-callout clock #'tester-callout delta
                                     (list spcm ret account-id))
          )))
    ))

(defun tester-start (spcm balance income)
  (let* ((account-id (make-account spcm balance income))
         (pages (min (spcm-account-balance spcm account-id)
                     (spcm-account-income spcm account-id)))
         (clock (spcm-clock spcm))
         (ret nil)
         )
    (setq ret (spcm-request-memory spcm account-id pages pages 2
                                   (lambda (x) (print "something_went_wrong"))))
    (case ret
      ('please-wait
       (print "please_wait"))
      ('insufficient-funds
       (print "insufficient-funds"))
      (otherwise
        (let ((delta (- (spcm-lease-end spcm ret)
                        (spcm-lease-start spcm ret)
                        (spcm-quantum spcm))))
          (clock-handler-add-callout clock #'tester-callout delta
                                     (list spcm ret account-id))
          )))
    account-id
    ))

(defun phase-one-callout (ticks spcm-acnts)
  (let* ((spcm (first spcm-acnts))
         (account-one-id (second spcm-acnts))
         (account-one (spcm-find-account spcm account-one-id))
         (account-two-id (third spcm-acnts))
         (account-two (spcm-find-account spcm account-two-id))
         (account-three-id (fourth spcm-acnts))
         (account-three (spcm-find-account spcm account-three-id)))
    (decf (memory-account-income account-one) 5)
```

```
    (incf (memory-account-income account-two) 5)
    ))

(defun phase-two-callout (ticks spcm-acnts)
  (let* ((spcm (first spcm-acnts))
         (account-one-id (second spcm-acnts))
         (account-one (spcm-find-account spcm account-one-id))
         (account-two-id (third spcm-acnts))
         (account-two (spcm-find-account spcm account-two-id))
         (account-three-id (fourth spcm-acnts))
         (account-three (spcm-find-account spcm account-three-id)))
    (decf (memory-account-income account-three) 5)
    (incf (memory-account-income account-one) 5)
    ))

(defun phase-three-callout (ticks spcm-acnts)
  (let* ((spcm (first spcm-acnts))
         (account-one-id (second spcm-acnts))
         (account-one (spcm-find-account spcm account-one-id))
         (account-two-id (third spcm-acnts))
         (account-two (spcm-find-account spcm account-two-id))
         (account-three-id (fourth spcm-acnts))
         (account-three (spcm-find-account spcm account-three-id)))
    (decf (memory-account-income account-one) 10)
    (incf (memory-account-income account-three) 10)
    ))

(defun test-main ()
  (let* ((spcm (make-instance 'system-page-cache-manager))
         (clock (spcm-clock spcm)))
    (spcm-init spcm)
    (let ((tester-one-acnt (tester-start spcm 20 10))
          (tester-two-acnt (tester-start spcm 20 10))
          (tester-three-acnt (tester-start spcm 40 20))
          (phase-one-delta (* 5 (spcm-quantum spcm)))
          (phase-two-delta (* 10 (spcm-quantum spcm)))
          (phase-three-delta (* 15 (spcm-quantum spcm))))
      (clock-handler-add-callout clock #'phase-one-callout phase-one-delta
                                 (list spcm tester-one-acnt
                                            tester-two-acnt
                                            tester-three-acnt))
      (clock-handler-add-callout clock #'phase-two-callout phase-two-delta
                                 (list spcm tester-one-acnt
                                            tester-two-acnt
                                            tester-three-acnt))
      (clock-handler-add-callout clock #'phase-three-callout phase-three-delta
                                 (list spcm tester-one-acnt
                                            tester-two-acnt
                                            tester-three-acnt)))
    spcm
    ))
```