TIES325 Tietokonejärjestelmä

Jani Kurhinen
Jyväskylän yliopisto
Tietotekniikan laitos

Kevät 2008

Luku 3

Tietokoneen ohjaaminen

Tietokone kehitettiin mekaanisista laskukoneista suorittamaan erilaisia toimintoja ohjatusti[1]. Perinteiset mekaaniset laskimet soveltuivat vain niihin tarkoituksiin, joihin ne olivat suunniteltu, mutta tietokone oli yleiskäyttöinen laite, joka voitiin ohjelmoida suorittamaan haluttuja tehtäviä. Ensimmäiset (ja edelleen yksinkertaisimmat) tietokonejärjestelmät lukivat muistiinsa suoritettavan koodin ja operoivat sen mukaan. Järjestelmien kehittyessä ja käyttäjien tarpeiden kasvaessa laitteiston ja suoritettavan ohjelmiston väliin tuli uusi kerros: käyttöjärjestelmä. Laitteiston näkökulmasta käyttöjärjestelmä on ainoastaan suoritettava ohjelma. Sovelluskehittäjän kannalta käyttöjärjestelmä puolestaan on alusta, jolle sovellusta kehitetään. Edelleen tarkasteltuna laitteisto suorittaa käyttöjärjestelmäohjelmaa, joka toimittaa sille prosessoitavaa dataa ulkoa tulevien syötteiden perusteella. Vastaavasti käyttöjärjestelmä peittää (suuressa määrin) laitteistotason sovelluskehittäjältä, jolle jää tehtäväksi varsinaisen toiminnallisuuden kuvaaminen ohjelmoitikielen avulla ilman, että hänen tulee huolehtia liikaa systeemitason yksityiskohdista, kuten ulkoisesta I/O:sta, välimuistista tai näytön ohjaamisesta (elleivät nämä kuulu sovelluksen toiminnallisuuteen).

3.1 Käyttöjärjestelmän rooli

Pohjimmiltaan käyttöjärjestelmä on siis vain ihmisen mukavuudenhalusta[2] syntynyt komponentti, jonka avulla tietokoneen käyttö on sekä sujuvampaa että tehokkaampaa. Kuten jo aiemmin luvussa 1.2 todettiin, käyttöjärjestemällä on rooli laitteiston ja sovellusohjelmoistijan rajapintana. Tarkastellaan tätä seuraavaksi hieman tarkemmin.

Käyttöjärjestelmän roolin alkaa siinä vaiheessa, kun sovellusohjelmaa kehitetään. Käyttöjärjestelmän päälle on rakennettu joukko työkaluja, jotka mahdollistavat nykymuotoisen sovelluskehityksen. Tähän työkalujen joukkoon kuuluvat editori, kääntäjä (tai tulkki) sekä debuggeri. Oikeastaan nämä työkalut eivät varsinaisesti liity käyttöjärjestelmään, mutta niiden käyttö on mahdollista käyttöjärjestelmän välityksellä. Varsinaisesti käyttöjärjestelmään kuuluva osa

[1] Tässä on kaksi painotettavaa asiaa: erilaisia toimintoja, eli siis yleiskäyttöisyys, sekä ohjattavuus, eli mahdollisuus hallita koneen tekemisiä.

[2] Aivan alussa taloudelliset tekijät olivat myös merkittäviä, sillä käyttöjärjeselmän avulla voitiin suorittaa tehtäviä sarjassa ilman kalliin tietokoneen hukkakäyntiä.

on soveluskehittäjän käytössä oleva systeemikutsujen kirjasto, joka mahdollistaa laitteistotason hallinnan.

Sovellusohjelman kannalta seuraava vaihe kehitystyön jälkeen on ohjelman käynnistäminen. Ilman käyttöjärjestelmää ohjelman käynnistys on sinänsä yksinkertaista, jos kone ainoastaan lähtee suorittamaan ulkoa saamaansa koodisyötettä. Tämän yksinkertaisuuden haittapuolena on jo edellä mainittu käytön jäykkyys: vain yksi käyttäjä pystyy suorittamaan yhtä ohjelmaa kerrallaan. Käyttöjärjestelmien myötä tilanne on muuttunut ja ohjelman käynnistämisestä on tullut yksi hyvin tärkeä lisävaihe. Käynnistysvaiheessa käyttöjärjestelmä lataa suoritettavan ohjelman koodin muistiin ja asettaa sen suoritusvuoroon.

Edellä mainittujen systeemikutsujen merkitys tulee esille sovelluksen suoritusvaiheessa. Systeemikutsujen suorituksen kautta ohjelma kykenee hallitusti käyttämään I/O-palveluita sekä tiedostojärjestelmää. Lisäksi käyttöjärjestelmällä on merkittävä osuus virhetilanteen sattuessa kohdalle. Koska käyttöjärjestelmä hallitsee ohjelmien suoritusta, tarkkailee se myös niiltä tulevia signaaleja. Saadessaan normaalin lopetussignaalin käyttöjärjestelmä voi olla tyytyväinen kaikkien osapuolien toimintaan ja jatkaa seuraavana listalla olevien töiden parissa. Havaitessaan virheen[3], käyttöjärjestelmän tulee reagoida siihen jollakin tapaa. Se, miten tulee reagoida, riippuu yleensä monista yksityiskohdista, mutta tavallisimpia reaktioita on ohjelman suorituksen keskeyttäminen, jonkin toiminnan uudelleensuoritus tai tilanteesta ilmoittaminen käyttöliittymän välityksellä.

Hyvän käyttöjärjestelmän tehtäviin kuuluu myös tapahtumien dokumentointi lokien avulla, jotta tietyistä operaatioista voidaan pitää lukua tai ongelmatilanteita voidaan tarkestella jälkekäteen. Tämä ei missään nimessä ole käyttöjärjestelmän pakollinen toiminto, mutta toisaalta koko käyttöjärjestelmäkerros on vain tietokoneen käyttäjän miellyttämiseksi.

Edellä kuvattiin käyttöjärjestelmän roolia käyttäjän ja laitteiston rajapintana. Toinen erittäin merkittävä rooli on toimia laitteiston resurssien jakajana. Tämä rooli on käyttöjärjestelmän kannalta sinänsä riskialtis, että prosessorin mielestä se on vain suorituksessa oleva ohjelma, eikä se näe mitään eroa sovelluksien ja käyttöjärjestelmän välillä. Koska prosessoriydin kykenee suorittamaan vain yhtä ohjelmamodulia kerrallaan, on käyttöjärjestelmän annettavan suorituksen hallinta prosessorin tehtäväksi Käyttöjärjestelmän on siis toimittava siten, että sen tulee pyrkiä varmistamaan suorituksen palautuminen itselleen tietyn sovellusohjelman lohkon suorituksen jälkeen[4].

3.2 Käyttöjärjestelmän kehitys

Käyttöjärjestelmän syntyyn tärkeä tarve alun perin oli kalliin tietokoneen mahdollisimman tehokas käyttö. Alussa tietokonetta valvoi operaattori, jolta varattiin tietty aikaväli suoritinaikaa. Oman vuoron alkaessa, piti ensin ladata oma sovellus tietokoneen muistiin ja sen jälkeen syöttää sovelluksen tarvitsemat parametrit. Jos työ ei valmistunut varatussa ajassa, työ keskeytettiin[5] ja yleensä

[3]Tämä virhe voi olla joko sovelluksen hallitusti lähettämän tai vakavammasta viasta johtuvan yllättä virhetapahtuma.

[4]Tähän perustuu virusten, matojen ja muiden haittaohjelmien toiminta. Haittaohjelma sisältää antaa prosessorille uusia ohjeita toiminnan jatkamisesta

[5]Tämä toki riippui välillisistä asianhaaroista, kuten siitä, kenen sovellus oli ajossa tai mitä oli jonossa.

mitään tuloksia ei tällaisessa tapauksessa ollut mahdollista saada. Tämä koko aikaikkuna oli siis mennyt hukkaan. Toisaalta työ saattoi valmistua varttitunnin etuajassa, jolloin seuraava varaaja ei välttämättä ollut vielä valmiina, ja jälleen tuhlautui kallista laskenta-aikaa tietokoneen seisottamisesta. Tosin tähän ongelmaan jonkinlaisen ratkaisun toi esimerkiksi vaatimus toimittaa suoritettava työ riittävän paljon etuajassa operaattorille.

Ensimmäiset versiot käyttöjärjestelmistä olivat ns. eräajojärjestelmiä (Batch system). Ensimmäisenä tällaisena pidetään General Motorsin 1950-luvun puolivälissä IBM:n 701-koneelle kehittämää järjestelmää. 1960-luvulle tultaessa tällaisia järjestelmiä oli jo käytössään lukuisilla toimijoilla.

Näiden ensimmäisten käyttöjärjestelmien ydin oli monitoriksi kutsuttu ohjelmakomponentti. Koneen operaattori syötti suoritettavia töitä tietokoneelle sarjana, josta monitori otti kerrallaan yhden työn käsittelyyn. Jokaisen sovelluksen oli palautettava hallinta monitorille oman suorituksensa jälkeen. Miten tämä sitten tapahtui? Monitorin (tai ainakin osan siitä) tuli olla koko ajan tietokoneen muistissa. Samalla kun monitori siirsi suoritusvastuun sovellukselle, se jäi odottamaan sovellusohjelman hyppykäskyä, joka osoitti monitoriin.

Onjelman suorituksenn liittyen monitorille annettiin ohjeet töiden suorittamiseksi erillisellä ohjauskielellä. Yksi työ sisälsi tyypillisesti (korkean tason kielellä kirjoitetun) ohjelmakoodin lataamisen, kääntäjän noutamisen tarvittaessa joltakin massamuistivälineeltä, ohjeltikoodin asettamisen suoritusjonoon ja sovelluksen syötteen lukemisen. Huomionarvoista näin nykypäivänä on, että esimerkiksi reikäkortteja käytettiin korkean tason kielen, kuten Fortran tai Cobol, lähdekoodin kuvaamiseen ja kääntäjälle syöttämiseen. Reikäkortit eivät siis sisältäneet pelkästään konekielistä koodia muutoin kuin aivan tietojenkäsittelyn alkuvaiheissa.

Seuraavassa vaiheessa käyttöjärjestelmät kehittyivät moniajaviksi eräjärjestelmiksi (Multiprogrammed batch system). Tähän kehitysvaiheeseen oli syynä tietokoneen suoritinajan tehostaminen entisestään. Vaikka vanhoissa yhden ohjelman kerrallaan suorittavissa järjestelmissä töiden vaihtamisesta tulevaa hukka-aikaa olikin saatu vähennettyä, yksittäisten ohjelmien sisälläkin suoritin saattoi olla jouten suuriakin aikoja. Seuraava esimerkki on lainattu Stallingsilta[6]. Oletetaan, että kesken ohjelmaa tulee lukea syöte nauhalta. Syötteen lukuaika on 1,5ms. Näiden syötteiden käsittelyyn tarvitaan 100 käskyä, joiden suorittaminen kestää 0,1ms. Operaation jälkeen data kirjoitetaan takaisin nauhalle, mikä kestää jälleen 1,5ms. Näinollen tämän rutiinin suoritus kokonaisuudessaa kesti 3,1ms, mutta keskussuoritinta vaadittiin ainoastaan 3,2% tästä ajasta. Vaikka yksittäiset odotusajat tuntuvatkin lyhyiltä, ohjelma, joka suorittaa miljoona levyoperaatiota hukkaa suoritinaikaa useta minuutteja.

Moniajavan eräjärjestelmän keskeinen ajatus oli siis saada edellä esitettyä hyötysuhdetta kasvatettua. Tämä voitiin toteuttaa lataamalla kaksi tai useampia ohjelmia samaan aikaa tietokoneen muistiin ja siirtämällä suoritusvuoro toiselle ohjelmalle esimerkiksi juuri I/O-operaation aikana. Jotta tämänlainen järjestelmä voitiin toteuttaa, vaadittiin uusia työkaluja. Näitä olivat kesketykset ja niiden erillinen käsittely sekä muistinhallinta.

Moniajavan eräajojärjestelmän ongelma oli edelleen se, että suoritettavat työt tuli asettaa jonoon edottamaan sopivaa hetkeä, jolloin monitori asetti työn

[6] William Stallings. Operating systems: internals and design principles, 3rd ed. Prentice-Hall. 1998.

ajoon. Erityisesti käyttäjän interaktiota vaatineiden sovelluksien kanssa tämä oli varsin hankalla. Ositusjärjestelmä (Time-sharing system) oli väline, jolla tähän ongelmaan lähtettiin etsimään ratkaisua. Siinä missä moniajavat eräajojärjestelmät pyrkivät maksimoimaan prosessorin käytön, ositusjärjestelmän tavoite oli minimoidan käyttäjän kokema viive. Ositusjärjestelmässä voitiin palvella useita käyttäjiä samanaikaisesti antamalla kullekin vuoronperään pieni jakso suoritinaikaa käytettäväkseen. Vaikka tietokoneen teho jaettiin useiden käyttäjien kesken, yhden käyttäjän kokema tehon väheneminen oli yleensä pienehkö johtuen I/O-operaatioiden ja erityisesti ihmisen reaktioajan hitaudesta. Käyttäjät olivat yhteydessä tietokoneeseen erillisten päätteiden kautta sen sijaan, että työt olisi ladattu yhdeltä konsolilta. Ensimmäisenä ositusjärjestelmänä pidetään MITissä 1962 kehitettyä ohjelmaa (Compatible Time-Sharing System, CTSS). Ositusjärjestelmä oli hyvin monimutkainen varrattuna aiempiin käyttöjärjestelmiin, ja ensimmäiset tällaiset järjestelmät olivatkin varsin alkeellisia, mutta ne loivat pohjan nykyaikaisille moniajaville monen käyttäjän järjestelmille.

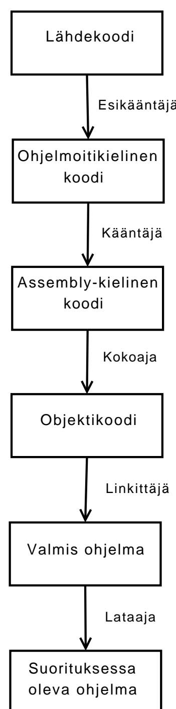
3.3 Sovellusohjelma

Edellä käsiteltiin sitä, miten käyttöjärjestelmä on kehittynyt palvelemaan nykyisiä tietoteknisiä tarpeitamme. Kehityksestä huolimatta käyttöjärjestelmän perustehtävänä on pysynyt yksi tehtävä: sovelluksien suorittaminen. Seuraavaksi tarkastelemam sitä, miten sovellus päätyy ohjelmistosuunnittelijan ideasta suoritettavaksi ohjelmaksi.

Tietokone osaa vain omaa kieltään, joka ei ihmiselle ole kovin intuitivinen. Tämän vuoksi sovellukset tehdään suurimmaksi osin jollakin korkeamman tason ohjelmointikielellä. Jotta korkeamman tason kielellä tehty ohjelma voitaisiin suorittaa tietokoneella, on se muunnettava tietokoneen konekielelle. Muuntaminen voidaan tehdä joko kääntämällä tai tulkkaamalla. Näiden kahden menetelmän ero on siinä, että kääntämällä ohjelma muunnetaan etukäteen kokonaisuudessaan konekielelle. Kääntämisen jälkeen ohjelma on valmis suoritettavaksi ja myös siirrettäväksi toiseen samaa arkkitehtuuria noudattavaan koneeseen. Tulkatessa ohjelmaa muunnetaan komento kerrallaan ajon aikana tulkkiohjelman avulla. Tulkatessa ei synny mitään erillistä pysyvää välitulosta, vaan tulkkaaminen on suoritettava jokainen kerta erikseen. Ohjelman siirtäminen toiseen koneeseen vaatii siten myös uudesta kohdekoneesta tulkin, mutta toisaalta tulkattava ohjelma voidaan siirtää myös eri arkkitehtuurilla toteutettuun laitteeseen, jos siinä vain on kyseisen ohjelmointikielen tulkki.

Tämän kurssin puitteissa kääntämistä tarkastellaan seuraavaksi. Sovellusohjelmaa käännettäessä tiedosto on kääntämisen yksikkö. Tämä tarkoittaa sitä, että yhteen tiedostoon kirjoitettu lähdekoodi käännetään kerrallaan. Mitä hyötyä tällaisesta sitten on? Edellähän mainittiin, että kääntämällä syntyy kokonaan uusi tiedosto, joka sitten suoritetaan. Erityisesti sovelluksen kehitysvaiheessa sen lähdekoodiin tehdään lukuisia muutoksia ja näiden muutosten vaikutusta halutaan tarkastella tai testata. Muutoksen vuoksi on siis tehtävä kokonaan uusi suoritettava tiedosto. Ohjelman kääntäminen on suhteellisen raskas prosessi ja pienen muutoksen vuosi erityisesti suurten ohjelmistoprojektin kääntäminen kokonaan on sekä hidasta että resursseja haaskaavaa. Lähdekoodin jakaminen osiin ja osien kääntäminen erikseen pienempinä kokonaisuuksina mahdollistaa ainoastaan muuttuneen osion uudelleenkääntämisen. Ohjelman

kääntäminen osissa komponentti kerrallaan mahdollistaa myös sen, että omassa sovelluksessa voidaan käyttää jo valmiiksi tehtyjä komponentteja, joiden lähdekoodia komponentin tekäjä ei kuitenkaan halua jakaa. Toisaalta tämä erillisten palasten kääntäminen tarkoitaa myös sitä, että ohjelman osat pitää myös liittää yhteen. Tätä vaihetta kutsutaan linkittämiseksi. Sovelluksen viimeinen vaihe kohti suoritukseen pääsyä on ohjelmakoodin lataaminen (yleensä joltakin ulkoiselta tallennusvälineeltä) tietokoneen muistiin. Kuvassa 3.1 on kuvattuna tämä elinkaari. Edellä mainittujen tehtävien lisäksi kuvasta löytyy joitakin muita välivaiheita ja lisäksi sekä kääntäminen että linkittäminen vaativat vielä hieman lisää selitystä.



Kuva 3.1: Sovellusohjelman tekemisen vaiheet

Kuvan 3.1 kaksi ensimmäistä laatikkoa ovat lähdekoodi sekä ohjelmoitikielinen koodi ja näiden välissä on esikääntäjä. Kirjoittaessamme ohjelmia, emmekö luokaan ohjelmointikielistä koodia, vaikka näin olemme tekevinämme? Vastaus on kyllä ja ei, vähän katsontatavasta riippuen. Joka tapauksessa tietyt lähdekoodin osat eivät sellaisenaan ole kääntäjän hyväksymää ohjelmointikielen syktaksia, vaan erityisiä ohjelmoitityötä helpottavia ja erilliselle esikääntäjälle suunnattuja ohjausdirektiivejä. Tällaisia ovat esimerkiksi C-kielen *#include* sekä *#define*. Näiden ohjausdirektiivien avulla voimme luoda "luettavampaa"tai "tiiviimpää"koodia esimerkiksi erillisten makrojen tai vakiotermien, kuten TRUE, FALSE, ON, OFF, avulla. Esikääntäjän tehtävänä on siis muuntaa nämä varsinaisen kääntäjän ymmärtämään muotoon, minkä lisäksi esikääntäjä poistaa myös itse sovelluksen kannalta ylimääräiset kommentit. Esikääntäjä ei

etsi tarkista koodin oikellisuutta tai järkevyyttä, vaan se suorittaa mekaanisen etsi-korvaa-tyyppisen operaation.

Seuraavassa vaiheessa kääntäjä tekee sovelluksesta assembly-kielisen version. Tämä välivaihe ei periaatteessa olisi välttämätön, mutta on toteutusteknisesti helpompaa ensin muuntaa sovellus symbolisen konekielen muotoon ja vasta sen jälkeen tehdä varsinainen konekielinen versio. Kääntäjän tehtävänä on varmistaa tässä vaiheessa, että syötteenä annettu koodi on kieliopillisesti oikein sekä tarkistaa siinä olevien aliohjelmakutsujen, muuttujien, ym. käyttö. Kääntäminen on siis ensimmäinen vaihe, jossa ohjelmistonkehitystyökalut voivat havaita ja ilmoittaa havaitsemastaan virheestä.

Sovelluksen rakentaminenkin pitää sisällään kääntämisen, symbolisesta konekielestä oikeaan tietokoneen ymmärtämään konekieleen. Rakentamisvaiheella on kuitenkin tärkeämpikin tehtävä, minkä vuoksi sitä tarkastellaan omana vaiheenaan. Ilman suurempaa pohtimista käännöksen symboliselta konekieleltä aidolle konekielelle voisi kuvitella olevan suoraviivainen tehtävä verrattuna edelliseen vaiheeseen, sillä assembly-kielinen koodi yleensä kääntyy 1:1 konekielelle (eli yhtä assembly-käskyä vastaa yksi konekielinen käsky). Kuitenkaan menetelmä, jossa kääntäjä lukee yhden käskyn, kääntää sen konekielelle ja jatkaa tätä prosessia niin kauan kunnes koko ohjelma on käännetty, ei valitettavasti toimi. Syy siihen löytyy käskyjen dataosista ja erityisesti siitä, että symbolinen konekieli on kuitenkin jo sen verran korkean tason määritys, että siinä voidaan viitata muuttuviin datatietoihin symbolisten leimojen (label) avulla sen sijaan, että käytettäisiin aitoja rekistereitä tai muistiosoitteita.

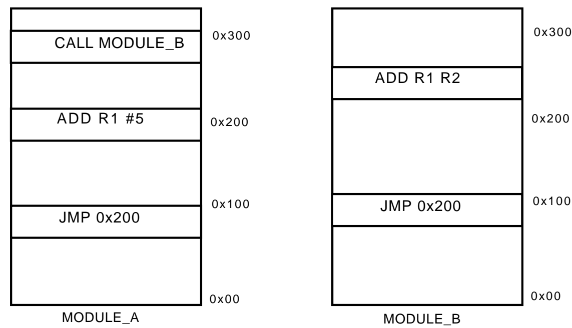
Tarkastellaan assembly-koodin kääntämistä konekielelle esimerkin avulla. Oletetaan, että meillä on käytössä joltakin kolmannelta osapuolelta hankittu kirjastomoduli, joka osaa laskea Fibonaccin lukuja. Edellä mainittiin, että sovellusohjelma voidaan kääntää pienissä paloissa, mikä mahdollistaa esimerkiksi ulkopuolisten kirjastojen käytön. Nyt meillä on juuri tämänlainen tilanne. Voimme siis luoda korkeamman tason ohjelmointikielellä lähdekoodin, jossa kutsumme ulkoista kirjastofunktiota ilman, että meillä on sen lähdekielistä toteutusta[7]. Edellä mainittiin myös tiedoston olevan kääntämisen yksikkö, joten on mahdollista kääntää tämä ilman funktion toteutusta oleva koodi assembly-kielelle. Alla on esitettynä ARM-assemblyllä fibonacci-funktion kutsu vakioarvolla 5.

```
mov     r0, #5
bl fibonacci
```

Ensimmäinen rivi, vakioarvon 5 sijoittaminen rekisteriin r0, on suoraviivaisesti käännettävissä konekielelle, mutta aliohjelmakutsun bl parametrille fibinacci ei voida tässä vaiheessa antaa oikeaa arvoa, sillä kokoajalla ei ole tiedossa aliohjelmarutiinin muistiosoitetta varsinaisen hypyn suorittamiseksi, ja kyseessä on ulkoisen viittauksen ongelma (external reference problem). Kokoajan on kuitenkin otettava jollakin tavoin kantaa edellä mainittuun aliohjelmakutsuun, sillä kokoajan tehtävähän oli tuottaa konekielistä koodia. Vastaava tilanne tulee vastaan yhdelläkin ohjelmamodulilla, jos aliohjelman (tai jonkin muun viittauksen) toteutus on myöhemmin kuin siihen viittaava kutsu[8]. Ennen kuin kokoaja on käynyt läpi ohjelmakoodia riittävän pitkälle, ei sillä ole mitään mahdollisuutta

[7]Vähän riippuen ohjelmoitikielestä funktion esittely (eli jonkinlainen otsikkotiedosto) on monesti oltava

[8]Tämähän on normaalia korkeamman tason kielillä, kuten C:llä. Funktion esittely on oltava ennen, mutta toteutus voi olla myöhemmin.



Kuva 3.2: Erilliset objektitiedostot omissa osoiteavaruuksissaan

tietää viitauksen osoitetta. Tästä käytetään nimitystä eteenpäinviittausongelma (forward reference problem).

Käytännössä kokoajan onkin luettava koodi kahteen kertaan[9]. Ensimmäisen lukukerran tehtävä on muodostaa erityinen symbolitaulukko, johon kerätään koodissa esiintyvien leimojen tiedot mukaan lukien (moduliin liittyvät suhteelliset) osoitetiedot (jos tiedossa) sekä näiden riippuvuussuhteet. Näiden tietojen perusteella kokoaja tekee toisella kierroksella käännetystä ohjelmakoodista objektitiedoston, joka sisältää sekä itse toiminnallisuuden että ohjeet siitä, miten leimoja tulee käsitellä suoritettavaa tiedostoa tehtäessä. Tähän vaiheeseen pääseminen ei vielä lopullisesti tarkoita sitä, että luotu koodi olisi kaikin puolin toimivaa ja valmiina suoritettavaksi, sillä seuraava vaihe on edelleen kriittinen.

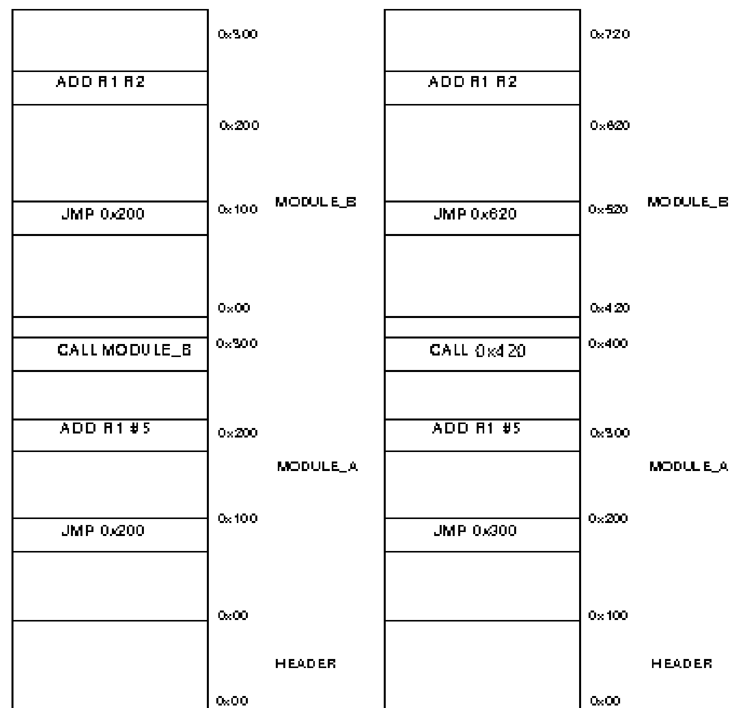
Objektitiedostojen luomisen jälkeen seuraava vaihe on valmiin ohjelman rakentaminen objektitiedostoista, jotka sisältävät jo valmiiksi prosessorin konekielistä koodia. Edellisessä vaiheessa konekielisiin käskyihin jäi vielä kohtia joita joudutaan tässä vaiheessa muokkaamaan. Ensinnäkin kukin objektitiedosto on luotu itsenäisesti omaan osoiteavaruuteensa. Samalla, kun linkittäjä yhdistää erilliset objektitiedostot toisiinsa, on sen muunnettava osien erilliset osoiteavaruudet yhteen yhteiseen osoiteavaruuteen. Tästä käytetään nimitystä uudelleensijoitusongelma (relocation problem).

Kuvissa 3.2 ja 3.3 kuvataan uudelleensijoitusprosessia. Kuvissa kaksi objektitiedostoa linkitetään toisiinsa siten, että modulissa A kutsutaan mobulissa B toteutettua rutiinia. Kuvan 3.3 ensimmäisessä vaiheessa modulit on liitetty toisiinsa ja lisätty suoritettavan ohjelman tarvitsemaa lisätietoa. Toisessa vaiheessa aliohjelma- ja hyppykäskyjen osoitteet on myös korjattu yhteiseen osoiteavaruuteen, minkä jälkeen ohjelma on valmis suoritukseen.

Edellä esitetty kuva objektitiedostosta on äärimmäisen yksinkertaistettu ja pitää sisällään myös jonkin verran ohjausinformaatiota. Erityisesti näistä on syytä mainita modulin ulkopuolelta käytettävien funktiokutsujen lista (entry point table), viittaukset ulkoisiin moduleihin (external reference table) sekä sisäisten leimojen uudelleensijoitustaulukko.

Viimeisenä ketjussa on lataaja, jonka tehtävänä on viedä tietokoneen prosessorille ohjeet ohjelman suorituksesta. Ohjelman latauksessa muistiin käyttö-

[9]Toinen vaihtoehto on kerätä ensimmäisellä lukukerralla kaikki muuttuva informaatio johonkin pienempään tilaan ja käydä tämä erillinen informaatio läpi koko koodin sijaan. Periaate molemmissa on kuitenkin sama.



Kuva 3.3: Objektitiedostot linkitettyinä

järjestelmä varaa riittävän kokoisen fyysisen muistilohkon ohjelmalle ja sijoittaa tämän ohjelman virtuaaliseen muistiavaruuteen. Tiedon sopivasta koosta lataaja saa suoritettavan ohjelmatiedoston ohjaustiedoista. Tämän jälkeen lataaja kopioi koodin ja datan muistiin, alustaa rekisterit, kopioi parametrit pinoon sekä suorittaa hypyn ohjelman alkuun.