Tito-kooste

Esipuhe

Tässä dokumentissa on esitelty tekstimuodossa (vrt. verkkoluentojen "kalvomuoto”) luentojen pääasiat, tai mitkä minun mielestäni ovat luentojen pääasiat. Luennoitsija saattaa olla eri mieltä siitä, joten tähän dokumenttiin ei voi vedota jos kokeessa kysytään jotain muuta. :-P

Dokumentin runko seuraa luentojen järjestykstä yksi kerrallaan, ja lopussa on vielä kooste siitä mitä tällä kurssilla pitäisi noin pääpiirteissään osata.

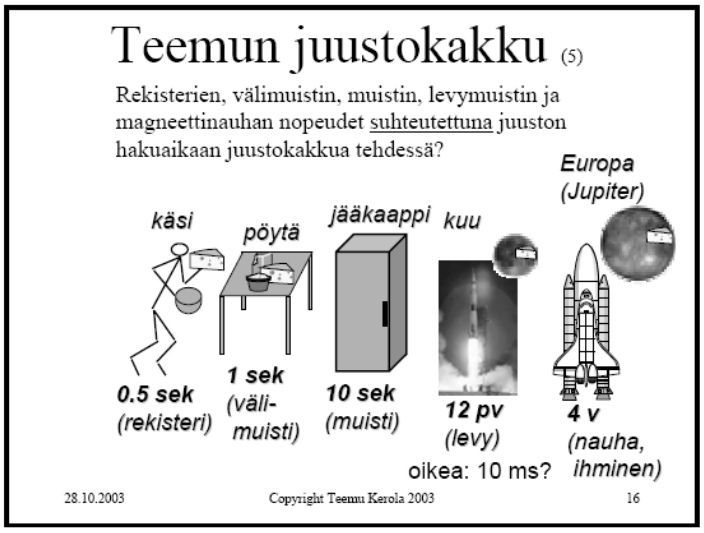
Versiohistoria:

20.2.2011: Luvut 1-3 ovat valmiina. Neljäs luku on puoliksi valmiina.
21.2.2011: Neljäs luku kirjoitettu loppuun. Luvut 5-6 lisätty.
23.2.2011: Pieniä korjauksia lukuun 6. Luvut 7-8 lisätty.
25.2.2011: Lisätty luvut 9-11 ja yhteenveto. Onnea tenttiin! ☺

Sisällysluettelo

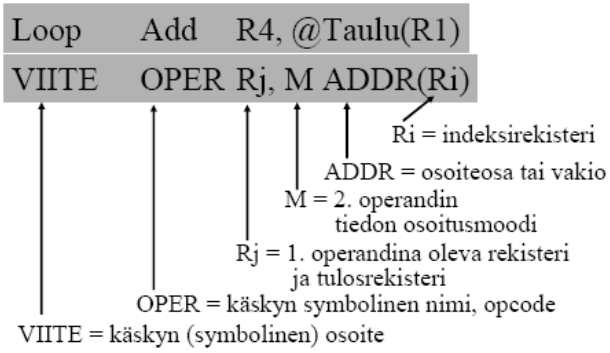
Luento 1: Tietokonejärjestelmän kokonaisrakenne

Muista Teemun juustokakku! Kaiken muun tältä luennolta voit ohittaa, mutta jos muistat juustokakun, voit siitä päätellä eri tietokoneen osien suhteelliset nopeudet vaikket tarkkoja lukuja muistaisikaan. Eivätköhän kaikki tietojenkäsittelytiedettä opiskelevat osaa muutenkin luetella tietokoneen oheislaitteita kuten näyttö, näppäimistö, jne. Järjestelmän rakenne käydään läpi luennossa 2 ja käännös/linkitys/lataus luennossa 10.



Luento 2: TTK-91 esimerkkitietokone ja sen simulaattori

Esimerkkitietokoneen käskyistä ks. ”ttk-91 komentoesimerkit” ja rakenteesta ”Keskusyksikön rakenne ja toiminta”. Niissä on selitetty asia yksityiskohtaisesti.



Tämäkin luku oli aika lailla johdantoa, tällä kertaa ttk-91 esimerkkikoneen käyttöön. Järjestelmän rakennetta ja toimintaa käsitellään tarkemmin myös luennossa 5. Toinen ihan hyvä muistaa asia on käskyn rakenne, joka näkyy yllä olevassa kuvassa, jos se ei ole vielä syöpynyt selkäytimeen asti TitoTrainer-harjoituksissa.

Luento 3: Konekielinen ohjelmointi

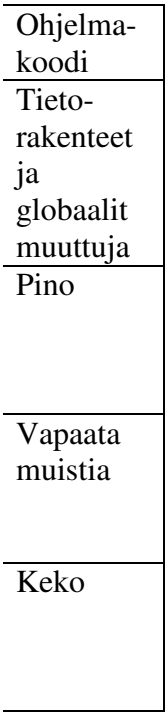
Tiedon sijainti ja siihen viittaus

Ennen kuin ruvetaan käsittelemään miten konekielellä ohjelmoidaan, tutkitaanpa vähän ohjelmakoodin ja sen käyttämän datan sijaintia ohjelman suorituksen aikana, sekä miten siihen voidaan viitata.

Ohjelman suorituksen aikana tieto voi sijaita joko rekistereissä tai muistissa (”keskusmuistissa”). Se ei voi sijaita esim. levymuistissa, flash-muistissa eikä verkossa, koska niistä tiedon hakeminen kestää ihan liian kauaa (muista se juustokakku-esimerkki!). Ohjelma voi toki pyytää käyttöjärjestelmää kopioimaan tiedon levymuistista tms. muistiin, mutta silloin se joutuu odottamaan mahdollisesti tuhansien konekäskyjen ajan. Käytännössä kun prosessi joutuu odottamaan näin hidasta operaatiota, suoritin luovutetaan toisen prosessin käyttöön.

Rekisterit ovat kaikkein nopeimpia, mutta niiden koko on rajoitettu. Lisäksi niitä on rakenteellisista syistä usein aika vähän. Esimerkiksi ttk-91:ssä niitä on vain 8 kappaletta. Muisti puolestaan on iso, mutta hidas (yleensä noin 10-20x hitaampi kuin rekisteri). Tästä herää tietysti kysymys siitä, milloin jotain muuttujaa kannattaisi pitää rekisterissä ja milloin muistissa?

Tietoon voidaan viitata usealla tavalla, riippuen siitä, missä se sijaitsee. Muistissa olevaan tietoon voidaan viitata suoraan muistiosoitteella (esim. ”LOAD R1, 100”, lue muistipaikasta, jonka osoite on 100, ja tallenna siinä oleva arvo rekisteriin R1) tai muuttujan nimellä (muuttuja on siis osoite, eikä arvo, jolloin tehdään siis yksi muistinouto käyttäen muotoa ”LOAD R1, X”). Luettava osoite voi myös sijaita muistissa jossain toisessa paikassa (eli annetaan osoite, josta löytyvää arvoa käytetään osoitteena, tätä kutsutaan pointterien käyttämiseksi), jolloin haluttuun arvoon päästään käsiksi kaksinkertaisella muistinoudolla.



Laitteistotasolla muistissa sijaitsevat tietorakenteet sijaitsevat ohjelmakoodin ja pinon välissä (ks. kuva). Pinon alapuolella on vapaata muistia ja sen alla keko. Sekä pino että keko voivat laajeta tähän muistialueeseen, kunnes ne kohtaavat toisensa. Jos tämä tila loppuu, tapahtuu ”stack overflow” virhe. Pino on aliohjelmien toteuttamista varten ja keko suoritusaikana dynaamisesti varattavia muistialueita varten. Esimerkkikoneessa TTK-91 ei ole kekoa.

Oikeassa tietokoneessa jos haluttu tieto on haettu muistista aikaisemmin, se saattaa löytyä myös välimuistista. Välimuisti toimii samalla tavalla kuin muistikin, paitsi että se on pienempi ja nopeampi (koska se sijaitsee samalla lastulla suorittimen kanssa). Muistihakua tehdessä ei kuitenkaan voida tietää löytyykö tieto välimuistista vai pitääkö se hakea keskusmuistista.

Rekisteriin puolestaan viitataan sen numerolla tai (symbolisessa konekielessä) nimellä, esim. R2 tai SP. Muistipaikkojen ja rekistereiden lisäksi tieto voi olla myös konekäskyn vakio-osassa, esim. ”LOAD R3, =10”.

Konekielisen ohjelmoinnin peruskäsitteet

Yksi tietokoneiden perustehtävistä on suorittaa laskennallisia tehtäviä, joten peruslaskutoimitukset ovat tärkeä konekielisen ohjelmoinnin peruskäsite. Näistä enemmän löytyy mm. ”ttk-91 komentoesimerkkejä” ja ”yksinkertaisia pikkuohjelmia” dokumenteista. Sama aritmeettinen lauseke voidaan toki laskea usealla tavalla, ja optimoivat kääntäjät yrittävät tehdä sellaisia ratkaisuja, jotka minimoivat käskyjen määrän, ja siten myös suoritukseen kuluvan ajan.

Toinen konekielisen ohjelmoinnin peruskäsitteistä ovat taulukot ja tietueet. Taulukot luodaan DS (data segment) käskyllä, ja niiden alkioihin viitataan tyylillä ”taulukon alkuosoite + alkion numero”, esim. ”LOAD R2, Table(R1)” kun alkion numero on rekisterissä R1. Tietueet ovat muuten samanlaisia kuin taulukot, mutta niihin viitataan toisin päin, esim. ”LOAD R2, age(R1)”, kun tietueen ”omistajan” osoite (eli tietueen ensimmäisen alkion osoite) on rekisterissä R1. Näihin yksityiskohtiin lienette tutustuneet TitoTrainer-tehtävien kanssa. Myös moniulotteisia taulukoita voidaan toteuttaa konekielellä tallentamalla taulukoita joko riveittäin tai sarakkeittain yhdeksi pötköksi. Sillä ei ole konekielen kannalta väliä kumpaa tapaa käytetään, mutta riveittäin voi olla mukavampi ihmisen hahmotettavaksi. Vieläkin monimutkaisempia tietorakenteita voidaan palauttaa näiden tietorakenteiden kombinaatioiksi (esim. taulukko, jonka sisällä on tietueita).

Kaikkia näitä tietorakenteita käytettäessä täytyy luonnollisesti sisältää tarkistuksia siitä, että pysytään halutun data-alueen sisällä, eikä siirrytä seuraavan tietorakenteen (tai pinon) muistialueelle.

Viimeinen tässä luennossa käsitelty asia on valinta- ja toistolausekkeet (if-else, switch, for, while jne.). Nämä kaikki toteutetaan konekielessä vertailu- ja hyppykäskyillä. Hyppyjä on sekä ehdottomia (JUMP) että ehdollisia (JZER, JGRE, JNGRE jne.). Ehdottomat hypyt hyppäävät joka tapauksessa, ehdolliset jos annetun rekisterin arvo on positiivinen/negatiivinen/nolla tai jos tilarekisterin bitit E, G tai L ovat päällä. Tarkempaa tietoa hyppykäskyistä katso esim. ”ttk-91 komentoesimerkit”. Loppujen lopuksi valinta- ja toistolausekkeiden toteuttamisessa vaikeinta on hahmottaa mitä tarkkaan ottaen haluat koodin tekevän. Kun olet pilkkonut sen päässäsi pienimpiin mahdollisiin osiin (esim. ”verrataan muuttujaa t lukuun 200, jos se on pienempi mennään tuonne, jos se on yhtä suuri kuin suurempi niin mennään tänne”), konekäskytason muoto on jo valmiina ja voit vain kirjoittaa sen ttk-91 syntaksilla. ☺

Luento 4: Aliohjelmien toteutus

Aliohjelmat ja parametrit

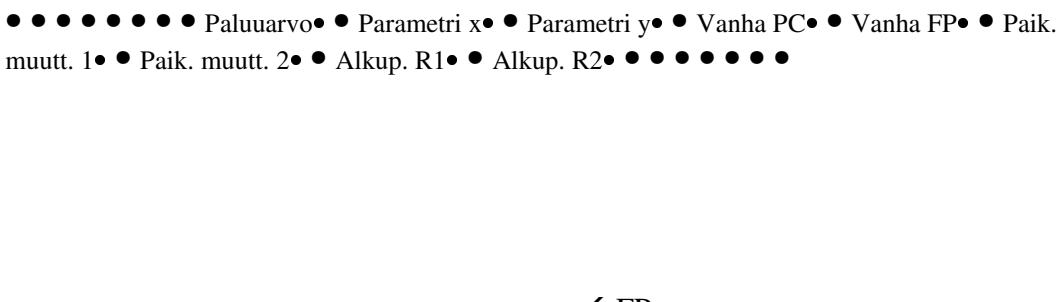
Korkean tason ohjelmointikielen tasolla aliohjelmia on kolmea tyyppiä: proseduureja (sisäänmenoparametreja ja ulostuloparametri paluuarvon sijasta), funktioita (parametreja ja paluuarvo) ja metodeja (parametreja, ehkä paluuarvo). Konekielessä näitä tyyppejä on vain kaksi, aliohjelma ja proseduuri. Näitä kaikkia voidaan kuitenkin vain kutsua aliohjelmiksi, ja sitten spesifioida erikseen puhutaanko aliohjelmasta, jolla on paluarvo, vai tallentaanko kyseinen aliohjelma jotain viiteparametrina saatuun osoitteeseen.

Parametrityyppejä on kolme: arvo-, viite- ja nimiparametrit. Arvoparametri on kopio todellisesta arvosta, eli alkuperäistä muuttujaa/muistiosoitteen arvoa ei muuteta aliohjelmassa. Viiteparametri puolestaan on muistipaikan osoite. Tällöin arvoa voidaan muuttaa, koska tiedetään sen osoite. Viiteparametri on mm. ainoa tapa muokata taulukkoja aliohjelmassa millään järkevällä tavalla, koska koko taulukon tunkeminen pinoon ja sitten sen poppaaminen ja kopioiminen alkio kerrallaan alkuperäiseen taulukkoon olisi todella työlästä ja aikaa vievää. Nimiparametrin tapauksessa välitetään parametrin arvon tai osoitteen sijasta sen nimi. Sitä käytetään nykyään lähinnä makroissa. Tämän kurssin puitteissa nimiparametreista ei tarvitse tietää muuta kuin että ne ovat olemassa. TTK-91 ei tue niitä.

Aktivointitietue

Järjestelmä toteuttaa aliohjelman käsitteen aktivointitietueen avulla. Aktivointietueen rakenne vaihtelee ohjelmointikielen ja järjestelmän mukaan, mutta tärkeintä onkin mitä aktivointitietueessa on, ei sen sisäinen järjestys. Luennossa (kuten koko kurssilla) käsitellään TTK-91:n esimerkkiä.

Käsitellään esimerkkialiohjelmaa int A(int x, int y). Eli A ottaa kaksi sisäänmenoparametria, x ja y, ja palauttaa yhden paluuarvon. Pääohjelmassa laitetaan ensin pinoon paikka paluuarvolle, sitten parametrit x ja y pushataan pinoon (joko arvo tai viiteparametreina, riippuen aliohjelman toteutuksesta). Seuraavaksi CALL-kutsu, joka aloittaa



aliohjelma, automaattisesti laittaa vanhan PC:n ja vanhan FP:n arvon pinoon, jotta suoritus voi jatkua oikeasta kohdasta aliohjelmasta palattaessa. PC:n arvoksi vaihtuu aliohjelman osoite, ja uusi FP osoittaa aktivointitietueen päälle (kutsuhetkellä, se ei siirry kun aliohjelmaa aletaan suorittaa). Aliohjelma voi sitten lisätä pinoon paikallisia muuttujia (A lisää niitä kaksi) ja lopuksi ennen kuin aliohjelman varsinainen suoritus alkaa, vanhat rekisterien arvot talletetaan pinoon (ohjelmoija määrittää aliohjelmassa mitkä rekisterit talletetaan).

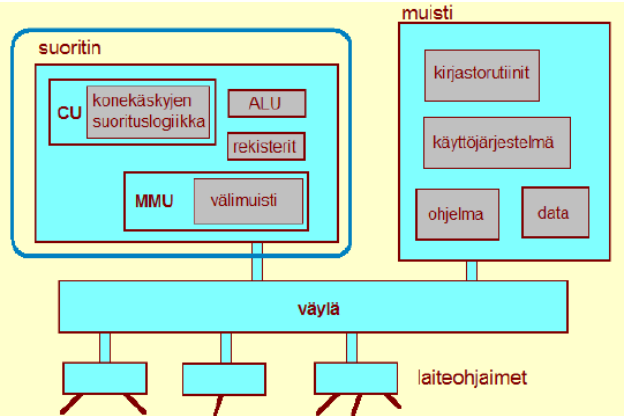
Eli parametreihin ja paikallisiin muuttujiin päästää aliohjelmassa käsiksi viittaamalla muistipaikkaan pinossa (FP:n suhteen, esim. viimeinen parametri on paikassa FP-2). Myös paluuarvon paikkaan päästään käsiksi tällä tavalla, ja siten voidaan tallettaa se pääohjelman popattavaksi. Koska pinoon pääsee käsiksi vain päältä, paluuarvon pitää luonnollisesti olla pohjimmaisena. Aliohjelman tehtyä mitä se nyt tekeekään, siitä poistutaan käskyllä EXIT, joka ottaa toisena operandina aliohjelmakutsun parametrien määrän (aliohjelman A tapauksessa =2), jotta se löytää oikean ”vanha PC” arvon. Luonnollisesti aliohjelman täytyy palauttaa rekisterit (POPR-käskyllä) ja poistaa paikalliset muuttujat pinosta ennen EXIT-käskyä. EXIT automaattisesti palauttaa vanhan PC:n ja vanhan FP:n paikalleen, ja pääohjelman suoritus voi jatkua siitä mihin se jäi.

Konekäskytasolla käyttöjärjestelmän palvelupyyntö, SVC, toimii aivan kuin aliohjelma. Ainut ero on se, että SVC asettaa suorittimen etuoikeutettuun tilaan, ja palvelupyynnöstä paluukäsky IRET palauttaa suorittimen käyttäjätilaan.

Käydään vielä läpi aliohjelman kutsun ja sieltä paluun päävaiheet:

- Laitetaan paluuarvolle paikka pinoon.
- Laitetaan parametri(t) pinoon.
- CALL-käsky, vanha PC ja vanha FP menevät automaattisesti pinoon.
- Aliohjelma alkaa.
    - Luodaan paikalliset muuttujat. Ne löytyvät kohdista FP+1, FP+2 jne.
    - Laitetaan rekisterit talteen PUSHR-käskyllä, tai ihan vain ”PUSH SP, R1”, ”PUSH SP, R2” jne. jos kaikkia rekisterejä ei tarvitse tallettaa.
    - Tehdään jotain työtä.
    - Tallennetaan paluuarvo sille varattuun paikkaan.
    - Valmistaudutaan poistumaan palauttamalla rekisterit POPR- käskyllä.
    - Poistetaan paikalliset muuttujat pinosta, esim. jos muuttujia olisi kaksi, ”SUB SP, =2” toimii.
    - Poistutaan aliohjelmasta EXIT-käskyllä, jolla annetaan operandina parametrien määrä.
- EXIT- käsky automaattisesti palautti vanhan PC:n ja vanhan FP:n rekistereihin PC ja FP. Pääohjelman suoritus jatkuu.

Luento 5: Suoritin ja väylä



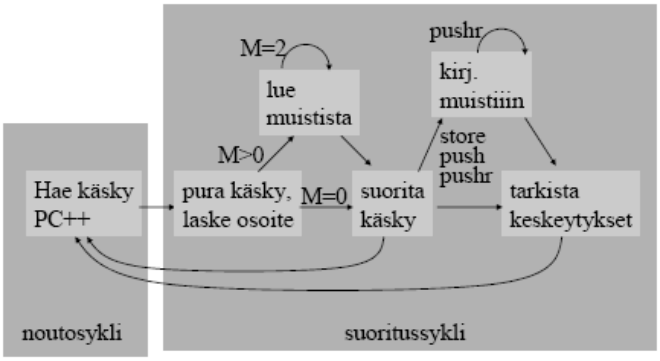
Järjestelmän rakenne

Tässä luennossa käsiteltiin järjestelmän rakennetta, erityisesti suorittimen toimintaa konekäskyn suoritussyklin tasolla, sekä väylän rakennetta yleisesti. Vieressä olevassa kuvassa nähdään keskusyksikön rakenne. Se koostuu suorittimesta, muistista, väylästä ja laiteohjaimista. Koska dokumentissa ”Keskusyksikön rakenne ja toiminta” on esitetty järjestelmän rakenne ja konekäskyn suoritus yksityiskohtaisesti, tässä koosteessa keskitytään yleistasoon ja keskeytyksiin.

Suoritin koostuu ohjausyksiköstä (CU), aritmeettis-loogisesta yksiköstä (ALU), muistinhallintayksiköstä (MMU) ja rekistereistä. Muistinhallintayksikköön voi kuulua välimuisti, joka on pienempi mutta nopeampi kuin varsinainen muisti, koska se on samalla piirilevyllä kuin suoritin. TTK-91:ssä ei ole välimuistia. Suorittimen eri osia yhdistää suorittimen sisäinen väylä. Ulkoinen väylä puolestaan yhdistää suorittimen muistiin ja laiteohjaimiin. Ulkoisessa väylässä on kolme osaa: kontrolliväylä (tunnetaan myös nimellä ohjausväylä), osoiteväylä ja dataväylä. Muistissa sijaitsee ohjelman ja sen käyttämä datan lisäksi myös käyttöjärjestelmän koodi ja data, sekä kirjastorutiinit. Jos suorituksessa on muitakin prosesseja (melkein aina on), niiden ohjelmakoodi ja data sijaitsevat myös muistissa. Kannattaa muistaa, että se miten muistia allokoidaan jokaiselle prosessille ja miten virtuaalisia muistisoitteita muutetaan fyysisiksi voidaan tehdä monella tavalla, esim. base-limit pari (jota TTK-91 käyttää), sivuttava tai segmentoiva virtuaalimuisti. Näiden toteutusyksityiskohdat tosin kuuluvat syventäville kursseille.

Konekäskyn nouto- ja suoritussykli

Konekäskyn suorittaminen on hyvin yksinkertainen. Ensin käsky noudetaan (noutosykli) ja sitten se suoritetaan (suoritussykli). Suoritussyklissä ensin käsky puretaan, sitten tehdään tarvittava määrä muistihakuja (0-2), käsky suoritetaan suorittimen piireissä, ja tarvittaessa tulos tallennetaan muistiin. Viimeiseksi tarkistetaan onko yksikään keskeytysbitti päällä. Jos on, keskeytykset käsitellään. Ja sitten koko sykli alkaa alusta noutamalla uusi käsky. Siinä on kaikki. Mitään muuta ei ole.



Kuten mainittu, tarkat rekisteritasoiset kuvaukset konekäskyjen suorittamisesta löytyvät ”Keskusyksikön rakenne ja toiminta” dokumentista.

Keskeytykset

Keskeytyksen määritelmä olisi suunnilleen ”mikä tahansa tilanne, jonka käsittely vaatii poikkeuksen käskyjen normaaliin suoritusjärjestykseen”. Kaikki keskeytystyypit on määritelty käyttöjärjestelmässä. Ei siis voi tulla sellaista tilannetta, jota käyttöjärjestelmä ei tunnistaisi. Ainut ”odottamaton” asia on se, milloin keskeytys tapahtuu.

Jokainen keskeytystyyppi on siis tunnettu, joten jokaiselle on oma käyttöjärjestelmän keskeytystenkäsittelyrutiini. Nämä keskeytystenkäsittelijärutiinit ovat oikeastaan koko järjestelmän tuntemia osoitteita. Keskeytysten käsittelyä voitaisiin siis ajatella yllättävänä aliohjelmakutsuna. Usein uusien keskeytysten käsittely estetään nykyisen keskeytyksen käsittelyn ajaksi. Keskeytykset voidaan toki estää muulloinkin kun se on tarpeen (esim. tietorakenteen sisäisen eheyden takia).

Keskeytykset voidaan luokitella kolmeen tyyppiin: käskyn aiheuttamat virhetilanteet, käskyn aiheuttamat muut poikkeustilanteet ja ulkoapäin suorittimelle tulleet signaalit. Virhetilanteita ovat mm. kelvoton operaatiokoodi, nollalla jako ja liukuluvun yli- tai alivuoto. Muita poikkeustilanteita ovat mm. SVC palvelupyynnöt ja Javan throw-catch lause, jolla voidaan heittää käyttäjän määrittelemiä keskeytyksiä. Suorittimen ulkopuolelta tulleita signaaleja puolestaan ovat mm. kellolaitekeskeytykset ja I/O-keskeytykset.

Vaikka tämä kaikki saattaa kuulostaa kauhean monimutkaiselta, loppujen lopuksi keskeytystenkäsittely on vain suorittimen tapa saada käyttöjärjestelmä vähän auttamaan asioissa, joiden tekemiseen suorittimella ei käyttäjä-tilassa ole oikeuksia. Kun suoritin siirtyy käsittelemään keskeytystä, se siirtyy etuoikeutettuun (eli käyttöjärjestelmä) tilaan, jossa se voi käyttää kaikkia konekäskyjä ja kaikkia muistiosoitteita. Keskeytyksen käsittelyn jälkeen suoritin palaa käyttäjätilaan.

Luento 6: Tiedon esitysmuodot

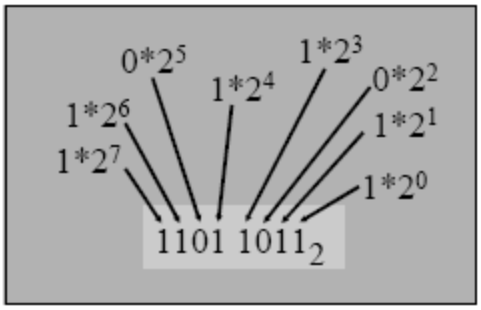
Mitä tahansa tietoa voidaan käsitellä laitteistossa kunhan sen muodosta sovitaan. On huomattava ero siinä, millaisena tietoa näytetään ihmiselle, ja millaisena järjestelmä tallentaa sen. Ajatellaanpa vaikka kuvaa. Tietokonelle se on vain iso pino ykkösiä ja nollia. Sano nyt siitä että onko se kenties omena vai ankka. Järjestelmällä on sitten säännöt, kuinka nämä ykköset ja nollat muunnetaan sellaiseksi kuvaksi, jonka ihminen näkee. Suoritin itsessään ymmärtää vain rajoitettua määrää eri tietotyyppejä, esim. TTK-91 ymmärtää vain kokonaislukuja. Monimutkaisemmat tietotyypit annetaan erilaisten aliohjelmien käsiteltäväksi.

Tietokoneessa kaikki tieto on siis tallennettu bitteinä, eli ykkösinä ja nollina. Kahdeksan bittiä muodostavat tavun. 1, 2, 4 tai 8 tavua muodostavat sanan (riippuen siitä kuinka vanha tai uusi kone on). Sana on muistiviittauksen perusyksikkö, joten muistista tuodaan ja sinne viedään aina kokonaisia sanoja. Suorittimen rekisterien pituus on yleensä yksi tai kaksi sanaa.

Numeromuunnokset

Tietokone käyttää binäärijärjestelmää ja ihmiset käyttävät (yleensä) kymmenjärjestelmää, joten on hyvä tietää miten numeroita muunnetaan näiden järjestelmien välillä.

Binääristä kymmenjärjestelmään on useimmille helpompi suunta. Lasketaan vain lukujen painoarvo suhteessa kahden potensseihin. Alla olevassa kuvassa esimerkiksi nähdään luvun 1101 1011 suhde kahden potensseihin. Siitä voidaan helposti laskea että 1101 1011 on $1*2^0+1*2^1+0*2^2+1*2^3+1*2^4+0*2^5+1*2^6+1*2^7=219$.



Kymmenjärjestelmästä binäärijärjestelmään on hivenen vaikeampi. Luennot tarjoavat kaavan ”jaa aina luku kahdella, laita jakojäännös muistiin, ja ota jakojäännökset käänteisessä järjestyksessä”. Se on hyvin toimiva kaava, mutta sitä on vaikea muistaa jos ei ymmärrä mistä se tulee. Yritän tässä vähän avata mihin metodi perustuu (käytetään luentojen esimerkkiä 57, koska 219:n loppumuoto ei mahdu yhdelle riville MathType:ssa):

$$
\begin{aligned}
&57\\
&=2*28+1\\
&=2*(2*14+0)+1\\
&=2*(2*(2*7+0)+0)+1\\
&=2*(2*(2*(2*3+1)+0)+0)+1\\
&=2*(2*(2*(2*(2*1+1)+1)+0)+0)+1
\end{aligned}
$$

Tässä siis ei tapahdu mitään muuta kuin jaetaan aina jokaisella rivillä sisimpien sulkujen sisällä oleva luku kahdella. Jakojäännökset kertyvät oikealle.

Ryhmitellään potenssit niin saadaan:

$$
\begin{aligned}
&2*(2*(2*(2*(2*1+1)+1)+0)+0)+1\\
&=2^5*1+2^4*1+2^3*1+2^2*0+2^1*0+2^0*1\\
&=111001_2
\end{aligned}
$$

Lukuja esitetään myös heksana, koska binääriluvuista tulee pakostakin todella pitkiä. Binääristä heksaan muunnetaan ottamalla aina 4 bittiä kerrallaan. Esimerkki:

0100	1100	1011	0001	1111
4	C	B	1	F

Heksa on siis 16-pohjainen (0000 = 0 ja 1111 = 15 = F) järjestelmä, joka menee 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Muistisäännöillä että 1000 = 8 ja 1111 = F ja maalaisjärjellä saa tehtyä muunnokset. Heksasta kymmenjärjestelmään tai toisinpäin tarvitsee joko laskimen, tai se pitää pyörittää binäärin kautta (ellet ole super matematiikkanero joka osaa laskea 16:n potensseja päässä ☺).

Tallennusmuodot

Aika usein tieto on pitempi kuin yhden tavun. Silloin pitää päättää tallennetaanko se eniten vai vähiten merkitsevä tavu ensimmäisenä. Jos eniten merkitsevä kirjoitetaan ensiksi, sitä kutsutaan Big-Endianiksi, jos vähiten, Little-Endianiksi. Big-Endian on intuitiivisesti helpompi ymmärtää, koska se toimii kuten olemme tottuneet lukemaan lukuja. Little-Endianin hyvä puoli puolestaan on se, että jos luku mahtuisi pienempäänkin määrään tavuja, sitä ”pienennettäessä” tiedon osoite ei muutu. Ajatellaan vaikkapa lukua 0000 0000 1101 0011.

Big-Endian: | 0000 | 0000 | 1101 | 0011 |
Little-Endian: | 0011 | 1101 | 0000 | 0000 |

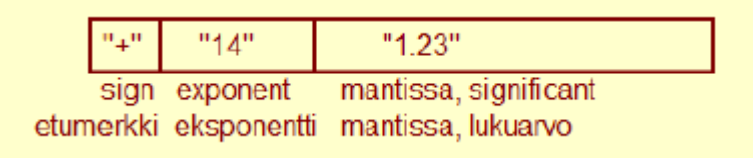
Jos Big-Endian muodossa oleva luku katkaistaisiin 2-tavuiseksi, sen osoite siirtyisi kaksi tavunpituutta eteenpäin alkuperäisestä paikasta. Little-Indian muodossa oleva luku pysyisi paikallaan, sen jäljestä vain vapautuisi 2 paikkaa.

Kokonaisluvuille ja liukuluvuille on myös eri merkintätavat. Käsitellään ensin kokonaislukuja.

Negatiiviset kokonaisluvut voidaan esittää neljällä tavalla:

- Etumerkkibitti erikseen. + on 0 ja – on 1.
- Yhden komplementti. Otetaan vastaava positiivinen luku ja käännetään sen bitit. Esim -57 on 1100 0110 ja +57 on 0011 1001. Tässäkin ensimmäinen bitti osoittaa etumerkkiä, mutta tämä muoto ei ole helposti ihmisen luettavissa.
- Kahden komplementissa komplementoidaan bitit samalla tavalla kuin yhden komplementissa, mutta lopuksi lisätään vielä 1. Tämä on yleisin tapa tallentaa luvut, koska sille on helpoin tehdä matematiikkapiirejä laskentaa varten.
- Vakiolisäys. Lisätään negatiiviseen kokonaislukuun jokin sovittu vakio että siitä tulee positiivinen.

Liukuluvuille on IEEE:n 32-bitin standardi:



”Liukulukumuunnos IEEE standardiin” dokumentista löytyy yksityiskohtainen selitys miten kymmenjärjestelmän luku muutetaan binääriseksi standardin mukaiseksi liukuluvuksi.

Luento 7: Tiedon tarkistus ja muisti

Tiedon tallennusmuodot jatkuu

Edellisessä luennossa asia tiedon tallennusmuodoista jäi vähän kesken. Käsiteltiin kokonaisluvut ja liukuluvut, sekä mainittiin vähän merkeistä, merkkijonoista, kuvista, ym. Mutta taulukot, tietueet ja oliot jätettiin käsittelemättä. Ei niissä tosin ole kovin paljon käsiteltävää. Konekielessä taulukoihin ja tietueisiin viitataan niiden ensimmäisen alkion osoitteella, ja sitten lasketaan ”alkuosoite + alkion numero”, että löydetään haluttu alkio muistista. Oliot toteutetaan yleensä tietueina. Ne tallennetaan kekoon.

Tiedon muuttumattomuus

Tämän luennon varsinainen asia on tiedon muuttumattomuus. (Ja vähän katsotaan muistia, mutta ei paljon, se on enemmän luennon 9 asia.) Tiedon tarkistuksien tavoitteena on pitää huolta siitä, että tieto ei ole muuttunut liikkuessaan laitteistossa. Tai sen puoleen verkossakaan, mutta se ei kuulu tähän kurssiin. Tiedon tarkistamiseen voidaan käyttää mm. pariteettibittejä, Hamminging koodia tai CRC:tä (Cyclic Rebundency Check). Kaikkien näiden tarkistustapojen perusidea on, että otetaan ylimääräisiä bittejä, jotka lasketaan datasta jollain tavalla, ja kun data on saapunut perille, lasketaan bitit uudestaan ja verrataan tuloksia. Jos kyseiset bittijonot ovat erilaisia, voimme todeta että tässä on nyt tapahtunut jokin virhe. Mitä enemmän bittejä otetaan tarkistuksen käyttöön, sitä enemmän virheitä voidaan havaita, ja ehkä korjatakin, mutta enemmän bittejä tarkoittaa myös enemmän ylimääräistä laskennallista työtä ja tilantarvetta.

Pariteettibitti on kaikkein yksinkertaisin tapa tarkistaa tiedon oikeellisuus. Pariteettibittejä on yksi bitti datayksikköä (voi olla sana tai tavu tai mitä vain) kohden. Parillinen pariteetti tarkoittaa, että ykkösbittejä on (pariteettibitti mukaan lukien) parillinen määrä. Pariton pariteetti puolestaan että ykkösbittejä on pariton määrä. Pariteettibitti huomaa yhden bitin virheen, muttei kahden (tietysti koska kaksi kääntynyttä bittiä kumoavat toisensa parittomuus/parillisuuslaskuissa). Sen kanssa ei myöskään tiedetä mikä biteistä on kääntynyt, sehän laskee vain ykkösten määrää.

Hamming-koodi onkin sitten huomattavasti monimutkaisempi tarkistustapa. Siinä jokaista bittiä ”vartioi” kaksi tai useampi pariteettibitti. Pariteettibitit ovat aina järjestysnumeroiltaan 2^i (eli 1, 2, 4, 8,....). Sen, mitkä pariteettibitit vartioivat kutakin bittiä voi laskea siitä, millä pariteettibittien summalla saataisiin kyseisen bitin järjestysnumero. Esim. bittiä 13 vartioivat 8, 4 ja 1. Bittiä 39 puolestaan 32, 4, 2 ja 1. Hetkisen miettimällä ja vertaamalla lukujen esittämiseen binäärinä voi todeta, että nämä

kombinaatiot todellakin ovat yksikäsitteisiä. ☺ Siinä on ihan kiva oikopolku: kun luvun muuttaa binääriksi, näkee suoraan ykkösten kohdista mitkä pariteettibitit vartioivat sitä.

Siinä on taas se käsite, ”pariteettibitti”. Hamming-koodi on oikeastaan vain iso kasa erillisiä pariteetteja. Jokaiselle pariteettibitille lasketaan kaikkien sen vartioimien bittien ykkösten määrä. Jos niitä on parillinen määrä, tarkistusbitiksi tulee 1. Jos pariton, tarkistusbitiksi tulee 0. Koska jokaista databittiä vartioi vähintään kaksi tarkistusbittiä, yhden bitin virhe voidaan aina havaita ja korjata. Esimerkiksi jos 9. bitti kääntyy nollasta ykköseksi, tarkistusbittien 1 ja 8 pariteetit ovat väärin. Silloin voidaan vain laskea 1+8 (koska jokaista bittiä vartioivat ne pariteettibitit joiden summana kyseisen bitin järjestysluku voidaan esittää) ja todeta, että 9. bitti on väärin, käännetään se takaisin.

Hamming-koodi on siinä rajoittunut, että se pystyy korjaamaan vain yhden bitin virheet, ja huomaamaan kahden bitin virheet (mutta silloin se ei pysty tunnistamaan niitä, koska ”vääristä” pariteettibiteistä voi muodostaa useampia kombinaatioita). Kolme tai enemmän saattavat jäädä siltä kokonaan huomaamatta jos ne kumoavat toisensa sopivasti. Esimerkiksi tiedonsiirrossa bittivirheet yleensä tapahtuvat isoina köntteinä. Tällöin voidaan käyttää CRC-menetelmää. Sen yksityiskohdat kuuluvat tietoliikenteen kurssille, mutta todettakoon, että se huomaa suurimman osan virheistä, usein enemmän kuin 99%.

Välimuisti ja muistityypit

Luennolla 5 mainittiinkin jo välimuisti. Se on yleensä suorittimen kanssa samalla piirilevyllä oleva pienempi muisti. Koska se käyttää suorittimen sisäistä väylää ulkoisen sijasta, se on huomattavasti keskusmuistia nopeampi. Välimuistissa pidetään kopioita viimeksi haetusta datasta, ja yleensä kyseisen datan ympärillä olevasta datasta (oletuksella, että on todennäköistä että halutun datan lähellä olevaa dataa halutaan aika pian). Suoritin ei kuitenkaan tiedä onko data välimuistissa vai ei, se vain lähettää käskyn noutaa data. Jos se löytyy välimuistista, hyvä, jos ei, mennään muistiin asti ja talletetaan haluttu data välimuistiin ennen kuin se annetaan suorittimen käyttöön. Tietysti myös kun dataa muutetaan, muutokset täytyy tehdä molempiin kohteisiin.

Muisti voidaan toteuttaa monella tavalla:

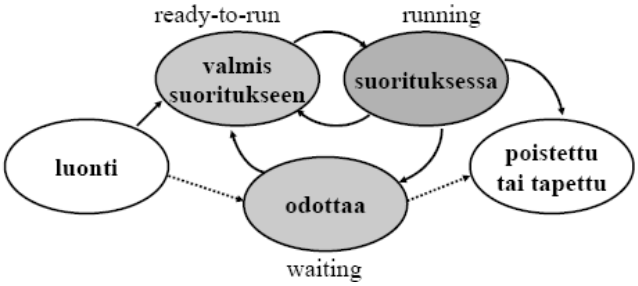
- RAM (Random Access Memory): häviävää muistia, kaikki tieto häviää virran katketessa.
    - DRAM (Dynamic RAM): halvempi ja hitaampi, ja tietoja pitää koko ajan virkistää. Useimmiten käytetään keskusmuistissa.
    - SRAM (Static RAM): kalliimpi ja nopeampi, tietoja ei tarvitse koko ajan virkistää. Useimmiten käytetään välimuisteissa.
- ROM (Read-Only Memory): Tieto säilyy ilman sähkövirtaa. Hitaampi lukea kuin RAM. Nimensä mukaisesti alkujaan ROM-muistit olivat muuttumattomia, ne kirjoitettiin vain kerran. Nykyään niihin saatetaan kirjoittaa useinkin. Nykyään kutsutaan myös Flash-muistiksi.

Tuon yksityiskohtaisemmin niitä ei tarvitse muistaa, onhan niitä Mask-ROMeja ja PROMeja ja EPROMeja ja ties mitä, mutta tärkein asia on tietää RAM:n ja ROM:n ero.

Luento 8: Ohjelman ja käyttöjärjestelmän toteutus

Prosessi

Prosessi on järjestelmässä olevan ohjelman esitysmuoto. Järjestelmässä voi olla yhtä aikaa useita prosesseja, joko samasta tai eri ohjelmista, mutta suorittimella niitä on kerrallaan vain yksi. Muut järjestelmässä olevat prosessit voivat olla esim. odottamassa vuoroaan, I/O-operaation valmistumista tai viestiä toiselta prosessilta. Suoritusvuorossa olevan prosessin vaihtamista kutsutaan prosessin vaihdoksi. Aikaisemmin suoritusvuorossa olevan prosessin tiedot kopioidaan muistiin, sen omaan prosessin kuvaajaan (PCB, Process Control Block) ja suorittimen rekisterit alustetaan vuoron saaneen prosessin tiedoilla (jotka saadan sen prosessin kuvaajasta). Prosessin vaihto tapahtuu hyvin usein, mutta viimeistään silloin kun prosessi ei voi syystä tai toisesta jatkaa suoritustaan (esim. prosessi on saanut suorituksensa loppuun tai sen pitää lukea tiedostosta jotain).



Prosessilla on viisi mahdollista tilaa. Ensin se luodaan, josta se yleensä siirtyy ”valmis suoritukseen” tilaan. Tosin joskus se joutuu ensin ”odottaa” jonoon, esimerkiksi jos käyttöjärjestelmä ei voi antaa sille muistia prosessin kuvaajalle. ”Valmis suoritukseen” jonosta prosessi voi luonnollisesti siirtyä suoritukseen. Suorituksesta prosessi voi joko siirtyä ”poistettu tai tapettu” tilaan (jos se sai työnsä valmiiksi tai käyttäjä tappoi sen), ”odottaa” tilaan (jos se jää odottamaan esim I/O:ta) tai ”valmis suoritukseen” tilaan (esim. jos sen aika suorittimella loppui). ”Odottaa” tilasta voidaan puolestaan siirtyä joko ”valmis suoritukseen” tilaan (jos prosessi sai sen mitä se odotti” tai ”poistettu tai tapettu” tilaan (jos käyttäjä tappoi sen).

Kuten vähän mainittiinkin, prosessien vaihto tapahtuu prosessin kuvaajien avulla. Niihin on talletettu kaikki, mitä prosessi tarvitsee jatkaakseen suoritustaan aivan kuin mitään keskeytystä ei olisi tapahtunutkaan. Prosessin kuvaajaan on talletettu mm. muistialueet, aukiolevat tiedostot, ja tietysti suorittimen rekisterien arvot silloin kun prosessi poistui suorituksesta. Jokaiselle prosessille on oma kuvaajansa, ja käyttöjärjestelmä käsittelee prosesseja juuri näiden kuvaajien kautta. Hauska yksityiskohta sinänsä, että prosessi ei pääse käsiksi omaan prosessin kuvaajaansa.

Käyttöjärjestelmä

Käyttöjärjestelmää voidaan ajatella monelta kantilta: Käyttäjä sitä harvemmin edes huomaa, ellei se toimi huonosti käytettävän ohjelman kanssa. Ylläpitäjää puolestaan

kiinnostaa enemmän käyttöjärjestelmän kokonaistoiminta, ei niinkään sen toiminta yhden ohjelman kanssa. Käyttöjärjestelmän voi myös ajatella olevan puhdas käyttöliittymä laitteistoon. Itse asiassahan käyttöjärjestelmä piilottaa laitteiston yksityiskohdat käyttäjiltä (myös ohjelmat ovat käyttäjiä!).

Käyttöjärjestelmä toimii monella tasolla. Se tekee laitteistojen käytön helpoksi, koska ohjelmat eivät silloin ole toteutuksessaan laiteriippuvaisia. Se huolehtii resurssien, kuten suoritinajana ja muistin, reilusta jaosta. Se myös toimii järjestelmän eheyden valvojana, huolehtien poikkeuksista (keskeytyksistä) ja pitäen huolta siitä, että yksikään prosessi ei pääse haittaamaan toisten prosessien toimintaa.

Vähän yksityiskohtaisemmalla tasolla ajatellen käyttöjärjestelmä huolehtii prosessien luonnista ja tuhoamisesta, niiden välisestä viestinnästä, suoritinajan jakamisesta, muistin jaosta, prosessien muistialueiden suojaamisesta toisiltaan, tiedostojen/laitteiden lukemisesta ja kirjoittamisesta, muiden järjestelmien kanssa kommunikoinnista (esim. verkon välityksellä) ja paljosta muusta.

Käyttöjärjestelmä koostuu joukosta KJ-prosesseja ja KJ-aliohjelmia. Tässä vaiheessa niistä tutuin on varmasti TitoTrainerin SVC SP, =halt. KJ-aliohjelmia kutsutaan SVC (supervisor call) käskyllä ja niistä poistutaan IRET (interrupt return) käskyllä. Käyttöjärjestelmää voidaan kutsua myös tavallisella CALL-aliohjelmakäskyllä, mutta tällöin se ei toimi etuoikeutetussa tilassa, mikä rajoittaa sen mahdollisuuksia tehdä asioita. Käyttöjärjestelmä voi siis olla lohkotettu etuoikeitettuun ja ei-etuoikeutettuun osaan. On vielä muutama muukin tapa kutsua käyttöjärjestelmää. Jos käyttöjärjestelmäpalvelu on toteutettu prosessina, sitä ei voida kutsua aliohjelmakutsulla. Tällöin käytetään SEND/RECEIVE-viestiparia. Kutsuva prosessi lähettää SEND-viestin, ja odottaa RECEIVE-rutiinissa vastausta. Myös keskeytykset aiheuttavat käyttöjärjestelmäkutsun.

Luento 9: Ulkoinen muisti ja I/O:n toteutus

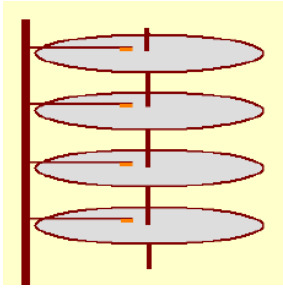
Ulkoinen muisti

Muistia on tähän mennessä käsitelty vain keskusmuistin ja välimuistin tasolla. Mutta tietenkin tietokoneissa on myös ulkoisia muistilaitteita. On levymuistit, CD-romput, DVD-romput, Flash-muistit yms. Suorittimen kannalta olisi mukavinta, jos kaikki tieto olisi keskusmuistissa, koska levyhaut kestävät suorittimen aikaskaalassa hyvin kauan (vrt. juustokakku ja kuu). Mutta usein kaikki ohjelman tarvitsema data ei mahdu keskusmuistiin. Tällöin suoritus täytyy keskeyttää siksi aikaa, että tarvittava tieto haetaan levymuistista.

Tätä ongelmaa voidaan helpottaa käyttämällä virtuaalimuistia, joka yrittää ennakoida mitä tietoa prosessi tulee tarvitsemaan. Virtuaalimuisti toimii käyttöjärjestelmän ohjauksessa. Tietenkin suorittimen aikaskaalassa prosessi joutuu odottamaan kun KJ käyttää suoritinta virtuaalimuistin hallinnointiin, mutta hyvin toimivan virtuaalimuistin kanssa ihminen ei juuri huomaa joutuvansa odottamaan. Virtuaalimuistin

toteutusyksityiskohdat kuuluvat syventäville kursseille, tällä kurssilla tarvitsee vain tietää, että virtuaalimuisti on kaksitasoinen: paljon tilaa vievä tukimuisti levyllä (johon on talletettu kopiot kaikista ohjelman tarvitsemista tiedostoista) ja vähän tilaa vievä osa keskusmuistissa. Näiden kahden välillä sitten kopioidaan tietoa.
Tiedostopalvelin toimii vähän samalla idealla, paitsi että sillä on yleensä useampia ”puskureita”, joissa tietoa säilytetään että sen noutaminen olisi nopeampaa. Tämä tietysti aiheuttaa omia ongelmiaan synkronoinnin kanssa. Miten pidetään huolta siitä, että kaikissa puskureissa on sama tieto?

Tämän sivujuonteen jälkeen palataan takaisin ulkoisiin muisteihin. Kiintolevy koostuu yhdestä tai useammasta magneettisesti käsiteltävästä levystä ja yhdestä tai useammasta kirjoituspäästä (yksi per levy). Kaikki nämä kirjoituspäät ovat kiinni samassa varressa.



Jokainen levypinta (joita levyssä voi olla 1 tai 2) muodostuu samankeskisistä urista (n. 2000-3000 per levypinta). Päällekkäiset urat muodostavat ns. sylinterin, ja niissä olevaa tietoa voidaan lukea samalla hakuvarren asennolla. Kukin ura on jaettu sektoreihin, mikä on pienin yksikkö joka levyltä voidaan kerrallaan lukea. Tiedon löytämiseksi pitää siis tietää levypinta, ura ja sektori. Käyttöjärjestelmän taulukoista löytyvät kaikki tarvittavat tiedot halutun ohjelman nimen perusteella.

Tiedon noutamiseksi hakuvarsi pitää ensin saada oikealle uralle. Tähän vaiheeseen menee ehkä 2 ms. Seuraavaksi levyä pyöritetään niin kauan, että oikea sektori on hakupään kohdalla. Jos sektori on valittu satunnaisesti, tähän vaiheeseen menee keskimäärin puoli pyörähdystä. Riippuen levyn pyörimisnopeudesta, tähän voi mennä n. 3-10 ms. Ja viimeisenä tieto luetaan. Tähän menee tietysti sen verran aikaa kun tarvitaan sen sektorin pyörittämiseen lukupään ohi, mikä on suhteellisen nopeaa. Kuten nähdään, suurin osa ajasta joka tarvitaan tiedon hakuun levyltä kuluu oikean kohdan etsimiseen. Siksi järjestelmä usein yrittää optimoida, ja hakee samalla kerralla paljon dataa.

Harvemmin on kuitenkaan kyseessä niin pieni määrä dataa, että se mahtuu yhteen sektoriin. Joten tiedostot jaetaan lohkoiksi, joista jokainen on sektorin jokin monikerta (1 lohko voi olla 1 sektori, tai 2 sektoria, tai useampi). Nämä lohkot yritetään sijoittaa vierekkäin, mutta aina niin ei käy. Mikä hidastaa tiedon hakua, kuten jokainen fragmentoituneen kovalevyn defragmentoinut tietää, kaikki toimii paljon nopeammin operaation jälkeen. ☺ Käyttöjärjestelmä pitää kirjaa siitä, missä lohkossa tai lohkoissa kukin tiedosto sijaitsee.

I/O

Järjestelmän rakenteesta puhuttaessa mainittiinkin jo laiteajurit. Tässä luennossa palataan niiden käyttöön.

Laiteajurit (järjestelmän puoli) ja laiteohjaimet (laitteen puoli) ovat sidospalikoita, jotka liittävät yhteen laitteen ja käyttöjärjestelmän. Laiteajuri on oikeastaan vain yksi

käyttöjärjestelmän palvelurutiini. Kun suorituksessa oleva prosessi tarvitsee I/O:ta, se kutsuu laiteajuria. Laiteajuri voi olla toteutettuna omana prosessina tai sitten aliohjelmana kutsuvalle prosessille.

Kuten harjoituksissa käytetyssä esimerkissä, driver.k91:ssä, laiteohjaimissa on kolme rekisteriä, kontrolli-, data- ja statusrekisterit. Laiteajuri ohjaa halutun oheislaitteen toimintaa kontrollirekisterin avulla, antamalla näihin käskyjä, jotka laiteohjain sitten toteuttaa. Statusrekisteri voi antaa erilaisia tilailmoituksia, esim. ”laite vapaa”. Varsinainen tiedonsiirto keskusmuistista laitteelle tai toisinpäin tapahtuu datarekisterin kautta. Laiteajuri antaa datan laiteohjaimelle, joka aikanaan tallentaa sen muistiin. Tai toisin päin.

Laiteohjain on pohjimmiltaan oikeastaan oma pieni prosessi. Laiteohjaimet ovat aina aktiivisina, toisin kuin laiteajurit, jotka voivat ”nukkua” odottaessaan I/O-keskeytystä, joka kertoo niille että laiteohjain on tehnyt työnsä ja odottaa laiteajurin tekevän jotain.

I/O-keskeytystä käyttävää I/O:ta sanotaan epäsuoraksi I/O:ksi. Siinä siis ajuri ”nukkuu” kunnes laiteohjain lähettää keskeytyksen. Ajuri ”herää” ja siirtää tiedon keskusmuistin. Kaksi muuta I/O-tyyppiä ovat suora I/O ja DMA. Suora I/O on kaikkein yksinkertaisin, siinä laiteajuri pyörii silmukassa tarkistaen jatkuvasti olisiko laiteohjain asettanut statusbittinsä kertomaan valmistuneesta työstä (kuten harjoitusten driver.k91). Suora I/O on suorittimen kannalta tehotonta, koska laiteajuri rohmuaa suorittimen käyttöönsä kunnes I/O-tehtävä on suoritettu. DMA-ohjaimet ovat huomattavasti kehittyneempiä kuin suora tai epäsuora I/O. Ne toimivat rinnakkain ajurin kanssa ja suorittavat itsenäisesti suuriakin I/O-töitä. DMA-ohjain osaa myös siirtää dataa suoraan ajurin ja muistin välillä, sen ei tarvitse kierrättää dataa suorittimen rekistereiden kautta kuten muiden ohjaintyyppien.

Luento 10: Käännös, linkitys ja lataus

Ensimmäisenä sanottakoon mikä minun mielestäni EI ole tärkeää tässä luennossa: makrot ja literaalit. Vaikka niistä harjoituksissa tehtiinkin iso numero, ne ovat pikkutilpehööriä (mikä valitettavasti ei estä niistä kysymisen kokeessa).

Makron määritelmä: Toistuva koodisarja, joka on liian pieni ollakseen aliohjelma mutta liian suuri että haluaisit kirjoittaa sen uudestaan ja uudestaan. Makron nimi koodissa korvataan käännösaikana sen rungolla.

Literaalin määritelmä: Tietyllä alueella muistissa (”literaalialueella”) sijaitseva vakio. Kaikki vakiot eivät ole literaaleja. Mitkä ovat literaaleja ja mitkä vakioita riippuvat ohjelmointikielestä, kääntäjästä ja kaikenlaisesta muusta mikä tekee niistä ei-yksiselitteisiä. Materiaali on erityisen sekava niistä, esimerkkien mukaan niitä luotaisiin DC-määreellä, mutta että niitä ei sitten saisi muuttaa? Siis häh?

Literaalien (teoreettinen) hyvä puoli on se että ne ovat kaikki samalla alueella muistissa, joten kyseiselle muistialueelle voidaan laittaa rajoituksia (esim. Read-Only), jotka

pätevät kaikkiin siellä oleviin muistipaikkoihin, eikä tarvitse käsitellä jokaista muistipaikkaa erikseen. Toinen hyvä (tai joskus huono) puoli on että ne ovat globaaleja, eli näkyvät kaikille.

Ja nyt asiaan. Kääntäminen, linkittäminen ja lataaminen ovat ne toimenpiteet, joilla korkean tason kielen ohjelmasta saadaan suorituskelpoinen ohjelma. Se on aika ”korkean tason” asiaa verrattuna suurimpaan osaan muista luvuista, joissa käsitellään asiaa rekisteritasolla.

Korkean tason kielen ohjelmaa tai moduulia kutsutaan käännösyksiköksi. Käännösyksikkö käännetään aina yhdellä kertaa (kuten nimikin sanoo). Kun yksi tai useampi käännösyksikkö käännetään konekielelle, siitä tai niistä tulee objektimoduuli. Tämä on siis ”käännösvaihe”.

Ohjelmasta voi tulla monta objektimoduulia, joten nämä pitää linkittää yhteen (”linkitysvaihe”), sekä mahdollisti pitää lisätä joitain kirjastomoduuleita. Staattisen linkityksen tuloksena saadaan ajomoduuli, jossa ei ole yhtään viittausta, joka ei johda minnekään. Tällöin ajomoduulista tulee todella iso. Dynaaminen linkitys on toinen vaihtoehto, siinä ei kiinnitetä kaikkia moduuleja paikalleen, vaan jätetään kutsukohdat auki, ja linkitetään tarvittavat palat ajon aikana. Tämä ratkaisu on tietysti paljon pienempi, mutta voi hidastua suorituksessa huomattavasti kun tarvittava objektimoduuli etsitään ja linkitetään.

Viimeisenä tulee ”latausvaihe”, jossa käyttöjärjestelmä luo ajomoduulista prosessin (jolla sitten on oma prosessinkuvaajansa), jonka suoritin voi ottaa suoritukseen.

No siinä olivat ne tärkeimmät. ☺ Voit nyt hypätä seuraavaan luentoon. Mutta käydään vähän vielä näiden vaiheiden yksityiskohtia läpi niille, joita yksityiskohdat kiinnostavat.

Käännösyksikön tarkka määritelmä on ”jollain ohjelmointikielellä kuvattu eheä kokonaisuus, joka halutaan aina kääntää yhdessä”. Esimerkiksi Javan yksi olio-luokka on käännösyksikkö. Käännösyksiköiden koko on tärkeä asia toiminnan sujuvuuden kannalta. Jos se on liian iso, sen kääntämiseen menee kauheasti aikaa joka muutoksen jälkeen. Toisaalta jos ohjelma on jaettu moneen kymmeneen käännösyksikköön, ne kyllä kääntyvät nopeasti, mutta linkittäminen on hankalaa. Juuri tämänhän takia ohjelmoinnin harjoitustyön ohjaajat suosittelevat että ohjelma koostuisi noin viidestä luokasta, vaikka syitä ei silloin selitetäkään. Käännösmoduulin ohjelmointikieli ei muuten ole tärkeä, ja sopivilla liitospalikoilla eri kielistä käännetyt objektimoduulit voidaan linkittää toisiinsa.

Käännöksen jälkeinen objektimoduuli koostuu ohjelman konekielisestä versiosta (binäärikoodia, ei symbolista konekieltä) sekä kaikista tiedoista, jotka tarvitaan moduulin linkittämiseen. Näitä ovat mm. uudelleensijoitustaulu, import/export-taulut ja symbolitaulu. Moduulin ulkopuoliset viitteet on merkitty import-tauluun. Kaikki sellaiset kohdat kyseisessä moduulissa, joihin toiset moduulit voivat viitata on merkitty export-tauluun. Uudelleensijoitustaulussa ovat kaikki ne osoitteet, jotka täytyy päivittää jos

kyseinen moduuli linkitetään toiseen moduuliin (koska jokaisen moduulin osoitteet alkavat nollasta). Ja symbolitaulussa ovat esitettynä kaikki symbolit ja niiden arvot (aivan kuten TitoTrainerissä).

Linkitysvaiheessa tarvitaan edellä mainittuja tauluja. Uudelleensijoitustaulun avulla osataan muuttaa osoitteet niin ettei niissä ole ristiriitoja. Import/export-taulujen avulla moduulien viitteet toisiin moduuleihin voidaan korvata todellisilla osoitteilla. Myös symbolit korvataan niiden arvoilla linkityksen aikana. Yleensä symbolitaulut tuhoutuvat tässä vaiheessa, ellei niitä haluta pitää mukana käyttäjäystävällisiä virheilmoituksia varten.

Latausvaiheessa ohjelmalle luodaan prosessinkuvaaja ja uusi prosessi siirretään ”valmis suoritukseen” jonoon odottamaan vuoroaan. Se ei vain ole sen monimutkaisempaa. ☺

Luento 11: Tulkinta ja emulointi

Tämä luento keskittyy Javaan ja sen tavukoodin ja Java-ohjelmien suorittamistapoihin. Vähän myös mainitaan C#:sta (Microsoftin wanna-be Java) ja TTK-91 arkkitehtuurin simuloinnista, mutta ne eivät ole kovin tärkeitä.

Kuten kaikki ovat varmasti tottuneet tekemään, ensin Java-ohjelma ”tavukooditetaan” javac-komennolla. Vaikka tämä on tavallaan ”käännös”, se ei luo objektimoduulia, kuten edellisessä luennossa käsiteltiin. Javan tavukoodimuoto on suorituslaitteistosta riippumaton (objektimoduuli on tietyn järjestelmän tuntemalle konekielelle tehty moduuli). Tätä tavukoodia voidaan sitten suorittaa Javan virtuaalikoneessa (JVM) usealla eri tavalla. Luennossa mainitaan näistä kolme.

Ensinnäkin on olemassa erityisiä Java-suorittimia, jotka suorittavat tavukoodia kuin se olisi konekieltä. Niiden käskykanta siis sisältää tavukoodin käskyt. Koska tavukoodi on liian tehotonta käyttöjärjestelmän suorittamiseen, käskykantaan kuuluu myös muita käskyjä. Näitä käytetään lähinnä joissakin pikkulaitteissa, jotka eivät tee juuri muuta kuin ajavat Java-sovelluksia. Esimerkiksi voisi olla teoreettinen ”Java-puhelin”, jonka kaikki ohjelmat toimisivat Javalla.

Toinen tapa on Java-käännös. Tässä tapauksessa tavukoodi käännetään kyseisen suorittimen tunnistamalle konekielelle, objektimoduuleiksi, jotka sitten linkitetään ja ladataan. Taas voidaan käyttää joko staattista (kaikki moduulit kerralla) tai dynaamista (käännetään moduuleja sitä mukaa kuin niitä tarvitaan) linkitystä. Dynaamisesta linkityksestä käytetään tässä tapauksessa nimeä Just-In-Time (JIT) käännös.

Kolmas tapa on Java-tulkki. Tämä on varmaan se kaikkein tutuin. Esimerkiksi selaimessa oleva Java on toteutettu Java-tulkilla. Tulkki on ihan tavallinen sovellus (yleensä kirjoitettu C:llä), joka lukee tavukoodia, ja simuloi sen käskyjä. Ihan kuin Titokone ja TitoTrainer. ☺

Oppimistavoitteet ja yhteenveto

Tietokoneen toiminta -kurssin oppimistavoitteet on määritelty alla. Tässä yhteenvedossa tarjoan ”in-a-nutshell” version jokaisen oppimistavoitteen sisällöstä.

Tietokonejärjestelmän rakenne ja toiminta

- Oppimistavoitteet
    - Osaa selittää laitteistokomponenttien väliset suuret nopeuserot ja niiden vaikutukset järjestelmään.
    - Osaa selittää välimuistin ja virtuaalimuistin toiminnan perusteet.
    - Osaa selittää, milloin ja miten prosessorin suoritustila vaihtuu.
    - Osaa selittää keskeytysten merkityksen ja toteutuksen konekäskyjen suorituksen tasolla.

Kannattaa yhäkin muistaa se juustokakku, johon suhteuttaa laitteistokomponenttien nopeudet. Yksittäiset nopeudet ovat kovin riippuvaisia laitteistosta, mutta niiden suhteet pysyvät suunnilleen vakiona. Rekisterit ovat kaikkein nopeimpia(juusto kädessä, ehkä puoli sekuntia että saadaan se kakkuun), välimuisti toiseksi (juusto pöydällä, n. 1-2 sekuntia), ja seuraavaksi muisti (juusto jääkaapissa, n. 10 sekuntia). Nämä kaikki ovat vielä suorittimen aikaskaalassa millisekunnin luokkaa, eli ihan hyviä suorituksen aikana. Levymuisti onkin sitten jo aika kaukana (juusto kuussa asti!) ja ihmisen antama syöte aivan älyttömän kaukana (juusto Jupiterin kuussa Europassa). Jos prosessi joutuu odottamaan levyä tai ihmisen syötettä, suoritus joudutaan keskeyttämään ja antamaan suoritusvuoro jollekin toiselle prosessille.

Välimuisti on suorittimen yhteyteen (eli samalla lastulle) toteutettu pieni muisti, joka käyttää suorittimen sisäistä väylää. Suorittimen sisäinen väylä on huomattavasti nopeampi kuin ulkoinen väylä, joten välimuisti on myös huomattavasti nopeampi kuin keskusmuisti. Suorittimen ohjausyksikkö ei kuitenkaan tiedä välimuistista mitään. Kun se pyytää tietoa muistista, MMU katsoo ensin välimuistista, ja jos tieto ei ole siellä, hakee sen keskusmuistista. Keskusmuistista haettu tieto myös kopioidaan välimuistiin. Jos välimuistissa olevaa tietoa muutetaan, muutettu tieto kopioidaan myös keskusmuistiin.

Keskusmuistin suuri ongelmahan on että se on liian pieni. Kaikki ohjelmassa tarvittava tieto ei yleensä mahdu kerralla keskusmuistiin. Tätä ongelmaa voidaan helpottaa käyttämällä virtuaalimuistia. Se toimii kaksitasoisesti: levymuistissa oleva tukimuisti (joka rohmuaa huomattavan määrän levytilaa) ja keskusmuistissa toimiva osa. Virtuaalimuisti yrittää ”arvata” mitä tietoa ohjelma tarvitsee seuraavaksi, ja kopioida sen keskusmuistiin, ettei prosessi joudu pysähtymään ja odottamaan.

Suorittimella (prosessorilla) on kaksi suoritustilaa: käyttäjä ja etuoikeutettu. Käyttäjätila on se ”normaali” tila. Etuoikeutettu on käyttöjärjestelmän tila, jossa voidaan käyttää kaikkia konekäskyjä ja kaikkia muistialueita. Prosessorin suoritustila vaihtuu esim. SVC- ja IRET-käskyillä, tai keskeytystenkäsittelyrutiinin alkaessa. Keskeytykset ovat mitä tahansa ”virhetilanteita”. Niitä on kolmenlaisia: käskyn aiheuttamat virhetilanteet , käskyn aiheuttamat muut poikkeustilanteet ja suorittimen ulkopuolelta tulevat signaalit

(liittyvät lähinnä I/O:een). Rekisteritasolla keskeytykset on toteutettu tilarekisterin biteillä. Jokaiselle keskeytystyypille on oma bittinsä, josta tulee ykkönen kun keskeytys tapahtuu. Jokaisen konekäskyn suorituksen jälkeen suoritin tarkistaa tilarekisterin bitit, ja käsittelee tarvittaessa keskeytykset.

Konekielitason ohjelmointi ja ohjelmien suoritus

- Oppimistavoitteet
    - Osaa toteuttaa tietueiden, olioiden ja moniulotteisten taulukoiden (eri tavoin talletettuna) tilanvaraukset ja niihin viittaamisen.
    - Osaa selittää globaalien ja paikallisten tietorakenteiden erot tilanvarauksessa ja niihin viittaamisessa ja osaa käyttää aktivointitietuetta ja aktivointitietuepinoa (rekursiivisten) aliohjelmien toteutuksessa.
    - Osaa selittää karkealla tasolla, kuinka (liikkuvat) kuvat ja äänet esitetään laitteistossa ja osaa selittää ja ottaa huomioon liukulukujen tarkkuuden muutokset laskennan aikana.
    - Osaa käyttää tiedon muuttumattomuuden suojauksen perusmenetelmiä (pariteettibitti, Hamming-koodi).

Tietueita ja taulukoita luodaan molempia DS-käskyllä, jolle annetaan parametriksi taulukon tai tietueen koko. Moniulotteiset taulukot voidaan tallentaa joko riveittäin tai sarakkeittain. Oliot voidaan tallentaa tietueina. Kaikkiin niihin viitataan ”ensimmäisen alkion osoite + paikka” tyylillä.

Globaalit muuttujat varataan muistin data-alueelta, paikalliset tietorakenteet puolestaan pinosta. Aktivointitietueen eri kohtiin voidaan viitata vertaamalla FP:n osoittamaan paikkaan (ks. luennon kuvaa). Aktivointitietueita voi kertyä pinoon useampia ”päällekkäin”, esim. rekursiivisessa aliohjelmassa. Ei niiden käyttämisessä mitään erityisempää ole, ne vain puretaan käänteisessä kutsumisjärjestyksessä.

Kuvat, liikkuvat kuvat ja äänet on yksinkertaisesti esitetty bitteinä, aivan kuten kaikki muukin. Miten ne ”puretaan” kuvaksi tai ääneksi on koodattu johonkin ohjelmistoon. Suoritin ei pysty niitä suoraan käsittelemään. Liukuluvut on tallennettu 32 bittiin (tai 64 bittiin, jos se on double tyyppi). Tällöin jos kaksi liukulukua, jotka ovat hyvin lähellä toisiaan vähennetään toisistaan, tapahtuu tarkkuusvirheitä. Jos tämä tulos vielä kerrotaan isolla luvulla, virhe moninkertaistuu.

Pariteettibitti laskee datassa olevia ykkösiä (parillinen tai pariton määrä). Pariteetti lasketaan ennen tiedon siirtoa ja tiedon siirron jälkeen. Jos näin saadut bitit ovat eri, voidaan todeta että jossain tapahtui virhe. Virhettä ei voida pariteettibitin tapauksessa korjata. Hammingin koodi on edistyneempi tapa suojata tieto, siinä voidaan korjata yhden bitin virhe, ja huomata kahden bitin virhe. Jos bittivirheitä on enemmän, Hammingin koodi ei huomaa niitä. Katso kyseistä luentoa Hamming-koodin tarkasta toimintatavasta.

Käyttöjärjestelmän perusrakenne ja toiminta

- Oppimistavoitteet
    - Osaa selittää, milloin ja miten suorituksessa oleva prosessi vaihtuu.
    - Osaa selittää käyttöjärjestelmän tavoitteet ja perustoiminnot sekä käyttäjän näkökulmasta että käyttöjärjestelmän ylläpitäjän näkökulmasta.
    - Osaa selittää käyttöjärjestelmäpalvelujen toteutusmekanismin erilaisten (etuoikeutettujen) aliohjelmien tai prosessien avulla.
    - Osaa selittää staattisen ja dynaamisen linkityksen edut ja haitat.
    - Osaa selittää käyttöjärjestelmän laitteistorajapinnan toteutuksen laiteajurien ja keskeytyskäsittelijöiden avulla. Osaa selittää tiedostopalvelimen ja tiedostovälimuistin perusidean.
    - Osaa selittää Java-virtuaalikoneen (JVM) perusrakenteen ja toiminnan sekä valita tiettyyn sovellukseen sopivan Java-ohjelmien suoritustavan.
    - Osaa selittää JIT-kääntämisen

Prosesseillahan on viisi mahdollista tilaa: luotu, valmis suoritukseen, suorituksessa, odottaa, ja lopettanut/tapettu. Suorituksessa oleva prosessi voi siirtyä valmis suoritukseen tilaan, jos sen suoritusaika loppui (kellokeskeytys) tai valmis suoritukseen -jonoon saapui prosessi, jolla on suurempi prioriteetti. Odottaa-tilaan prosessi voi siirtyä jos se joutuu odottamaan I/O:ta. Tietysti jos prosessi on suorittanut tehtävänsä, tai se tapetaan, se siirtyy lopettanut/tapettu-jonoon. Seuraava suoritukseen tuleva prosessi valitaan valmis suoritukseen -jonosta. Valmis suoritukseen – tai odottaa-jonoon siirtyvän prosessin suoritinympäristö tallennetaan muistiin sen prosessin kuvaajaan, ja uuden prosessin suoritinympäristö kopioidaan prosessin kuvaajasta suorittimelle.

Käyttöjärjestelmän tavoite käyttäjän näkökulmasta on yleensä pyörittää käyttäjän haluamia ohjelmia mahdollisimman hyvin. Käyttäjä harvemmin välittää mitä muita asioita KJ tekee, kunhan oma ohjelma toimii. Ylläpitäjän näkökulmasta KJ jakaa eri prosesseille muistia ja suoritinaikaa mahdollisimman reilusti, suojelee jokaisen prosessin omaa data-aluetta, että muut prosessit eivät sinne pääse, ja huolehtii yhteyksistä oheislaitteisiin.

Käyttöjärjestelmä voi toimia kokonaan etuoikeutetussa tilassa, tai osa sen prosesseista voi toimia käyttäjätilassa ja osa etuoikeutetussa tilassa. Näitä käyttöjärjestelmän palveluita voidaan kutsua aliohjelmakutsulla (CALL tai SVC) tai ne voidaan käynnistää omina prosesseinaan, jolloin prosessit ”keskustelevat” SEND/RECEIVE-käskyillä.

Kuten mainittua, käyttöjärjestelmä huolehtii yhteyksistä oheislaitteisiin. Tämä tapahtuu käyttöjärjestelmän puolelta laiteajurin ja laitteen puolelta laiteohjaimen avulla. Siihen voidaan käyttää joko suoraa I/O:ta (ajuri tutkii koko ajan ohjaimen statusrekisteriä, eli voisi ajatella että se koko ajan kysyy ”oletko nyt valmis?”) tai epäsuoraa I/O:ta, jossa laiteajuri menee ”nukkumaan” ja laiteohjain ”herättää” sen keskeytyksen avulla.

JVM, eli Java Virtual Machine, ei oikeastaan ole mikään oikea ”kone”. Se on määrittely, jonka avulla minkä tahansa käyttöjärjestelmän päälle voidaan rakentaa ”virtuaalikone”, joka suorittaa Javaa. JVM voi suorittaa Javan tavukoodia kolmella tavalla: java-

suorittimella (tavukoodi on sen ”konekieltä”), kääntämällä (tehdään käännös → linkitys → lataus aivan kuten normaalille ohjelmalle) tai tulkkaamalla.(emuloidaan koodia käsky kerrallaan). Kuten tavallisessa käännös/linkitys/lataus –paketissa, jos Java-ohjelma suoritetaan kääntämällä, se voidaan linkittää staattisesti (linkitetään kaikki moduulit kerralla) tai dynaamisesti (linkitetään ajon aikana sitä mukaa kuin tarvitaan). Dynaamista linkitystä kutsutaan myös JIT, eli Just-In-Time käännökseksi.

Lähteet

Auvo Häkkinen, Tietokoneen toiminta, luentomoniste (1998).
Teemu Kerolan verkkoluennot.
Teemu Kerolan luentokalvot (2003).
Tiina Niklanderin kertausluentokalvot (2011).