ITKA203 – Käyttöjärjestelmät
Kurssimateriaalia

(Tämä kopio on tuotettu 2. heinäkuuta 2011.)

Tässä versiossa on mukana jollain tavoin esitettynä kaikki kesällä 2011 käsitellyt asiat, poislukien demojen varaan jätetyt shell-, C-, ja skriptausasiat sekä harjoitustyö, joille on varattu vain otsikot liitteosiossa. Tunnettuja kauneusvirheitä on, ja ne korjataan siinä epätodennäköisessä tilanteessa että on aikaa. Mikäli kuitenkin löydät asiavirheitä tai epäselvyyksiä, joita ei ole sellaisiksi merkitty, otathan yhteyttä sähköpostitse, kiitos! Ne korjataan välittömästi!

Esipuhe

Tämä materiaali syntyi kesäopetusperiodilla 2011 Jyväskylän yliopiston Tietotekniikan laitoksella pidettävälle kurssille ITKA203. Se pohjautuu vahvasti kurssilla aiemmin käytettyyn materiaaliin sekä sisältörajaukseen (Jarmo Ernvall: ITKA203 – Käyttöjärjestelmät) sekä kesällä 2007 alustamiini lisäosiin. Materiaali sijoitetaan LaTeX-lähdekoodina YouSource-järjestelmään siinä toivossa, että sitä voitaisiin kehittää jatkossa yhteisvoimin sekä aina kulloisenkin luennoitsijan toimesta. Mikäli siis haluat selventää aihepiiriä nykyistä paremmin, ota rohkeasti yhteyttä projektiin, jotta pääset mukaan kirjoittamaan ja korjaamaan monistetta paremmaksi ja tämän hetken tarvetta vastaavaksi.

Jyväskylässä kesäkurssin 2011 alkaessa,
Paavo Nieminen <paavo.j.nieminen@jyu.fi>,
tohtorikoulutettava
kesäopettaja.

Sisältö

1 Johdanto

Aloitetaan mielikuvaharjoitteesta: Olet tietokoneen käyttäjä, ja tällä hetkellä olet kirjoittamassa opinnäytetyötäsi jollakin toimisto-ohjelmalla, vaikkapa Open Office Writerilla. Juuri äsken olet retusoinut opinnäytteeseen liittyvää valokuvaa piirto-ohjelmalla, esimerkiksi Gimpillä, ja tallentanut kuvasta tähän asti parhaan version siihen hakemistoon, jossa opinnäytteen kuvatiedostot sijaitsevat. Molemmat ohjelmat (toimisto-ohjelma, kuvankäsittely) ovat auki tietokoneessasi. Yhtäkkiä mieleesi tulee tarkistaa sähköpostit. Käynnistät siis WWW-selaimen ja suuntaat yliopiston Webmail-palvelimen osoitteeseen.

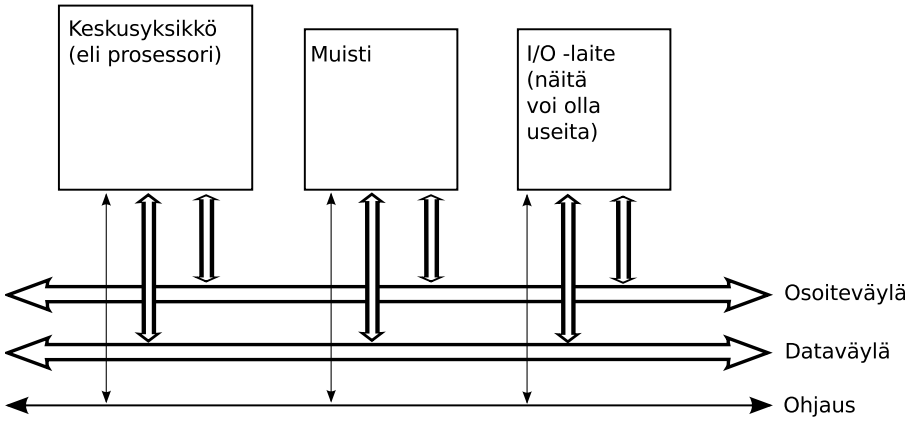
Edellä esitettyyn mielikuvaharjoitteeseen lienee helppo samaistua. Tietotekniikka on meille jokaiselle nykyään arkipäivää, jota ei tule ajateltua sen enempää. Teknologiaa vain käytetään, ja nykyään lapset voivat oppia klikkaamaan ennen kuin lukemaan. Tällä kurssilla kuitenkin mennään pintaa syvemmälle. Äsken kuviteltu tilanne näyttää ulkopuolelta siltä, että käyttäjä klikkailee ja näppäilee syöttölaitteita, esim. hiirtä ja näppäimistöä. Sitten "välittömästi" jotakin muuttuu tulostuslaitteella, esim. kuvaruudulla. Itse asiassa tietokonelaitteiston sisällä täytyy loppujen lopuksi tapahtua hyvinkin paljon jokaisen klikkauksen ja tulostuksen välisenä aikana. Tämän kurssin tavoite on, että sen lopuksi tiedät varsin tarkoin mm.

- miten näppäilyt teknisesti siirtyvät sovellusohjelmien käyttöön
- miten on teknisesti mahdollista että käytössä on monta ohjelmaa yhtä aikaa
- mitä on huomioitava, jos tehdään ohjelmia, jotka ratkaisevat samaa ongelmaa yhdessä ("rinnakkaisesti")
- mitä oikeastaan tarkoittaa että jotakin tallennetaan pysyvästi "tietokoneeseen"
- miksi pitäisi nostaa hattua jollekin, joka on saanut kehitettyä käyttökelpoisen käyttöjärjestelmän.

Lisäksi tavoitteena on lisätä monelta muultakin osin tietoteknistä yleissivistystä sekä ottaa haltuun perusteluineen ne ohjelmoinnilliset yksityiskohdat, joita hyvien ohjelmien tekeminen nykytietokoneille vaatii.

Tämän kurssin tavoitteita on nyt mainittu ylimalkaisesti. Joidenkin tavoitteiden ymmärtäminen saattaa vaatia jonkin verran perustietoa, jota ei vielä tähän johdantoon mahdu – miksi esimerkiksi usean ohjelman toimiminen samaan aikaan on jotenkin mainitsemisen arvoista, kun käytännössä tarvitsee vain klikata ne kaikki ohjelmat käyntiin? Jotta peruskäyttäjälle itsestäänselviä asioita (siis asioita, jotka *käyttäjälle tulee tarjota* että hän on tyytyväinen) osaisi arvostaa ohjelmien ja tietojärjestelmien toteuttajan näkökulmasta, täytyy ymmärtää jonkin verran siitä, millainen laite nykyaikainen tietokone oikeastaan on. Luku 2 tiivistää ja yleistää laitteiston olennaiset piirteet. Luku 3 kuvailee konekieltä, joka on ainoa rajapinta, jonka kautta tehtaalta toimitettua tietokonelaitteistoa on mahdollista komentaa[1]. osa-alueisiin yksi kerrallaan: luku 4 keskeytyskäsittelyyn ja kutsurajapintaan, luku 5 prosessien hallintaan ja vuorontamiseen, luku 6 synkronointiin ja kommunikointiin, luku 7 muistinhallintaan, luku 8 oheislaitteiden ohjaukseen ja luku 9 tiedostojärjestelmään. Aivan lopuksi, luvussa 10, voidaan toivottavasti jo ymmärtää käyttöjärjestelmältä edellytettävät osat ja jotakin niiden toiminnasta, jolloin päästään hieman miettimään millaisia korkean ja matalan tason suunnitteluratkaisuja niiden toteuttamiseen voitaisiin kuvitella, ja millaisia järjestelmäsuunnittelullisia tavoitteita ja ratkaisumenetelmiä toteuttamiseen voisi liittyä. Liitteissä on käytännön harjoitteita, joiden läpikäynti myös kuuluu kurssin sisältöön.

[1] Esitietokurssin "Tietokoneen rakenne ja arkkitehtuuri" käyneille nämä asiat lienevätkin jo tuttuja. Johtopäätös tulee olemaan yksinkertaistettuna, että tietokone on "tyhmä kasa elektroniikkaa", jolla ei käytännössä voi tehdä mitään hyödyllistä ilman "jotakin systeemiä", joka merkittävästi helpottaa elektroniikan käyttämistä. Luonnollinen nimi "systeemille" eli "järjestelmälle", joka helpottaa elektroniikan käyttämistä, voisi olla esimerkiksi ... "käyttöjärjestelmä".



Kuva 1: *Von Neumann -arkkitehtuurin yleiskuva: keskusyksikkö, muisti, I/O -laitteet, väylä.*

2 Tietokonelaitteisto

Tässä luvussa esitellään tietokoneen rakennetta ja toimintaperiaatteita, jotta voidaan tarkemmin nähdä, miksi käyttöjärjestelmää tarvitaan, mihin se sijoittuu tietotekniikan kokonaiskuvassa ja millaisia tehtäviä käyttöjärjestelmän tulee pystyä hoitamaan. Luku on pääosin tiivistelmä ja ”kertaus” esitietokurssina olleesta kurssista Tietokoneen rakenne ja arkkitehtuuri sekä aiemmin opetusohjelmassa olleesta Tietotekniikan Perusteet -kurssista. Tässä ei mennä kovin syvälle yksityiskohtiin tai teoriaan, vaan käsitellään aihepiiriä pintapuolisesti käyttöjärjestelmäkurssin näkökulmasta.

2.1 Yksinkertaisista komponenteista koostettu monipuolinen laskukone

Digitaalinen laskin (engl. *digital computer*) eli **tietokone** toimii tietyssä mielessä erittäin yksinkertaisesti. Kaikki niin sanottu ”tieto”, vielä enemmän ”informaatio” tai ”informaation käsittely”, jota digitaalisella laskimella ilmeisesti voidaan tehdä, on täysin ihmisen tekemää (ohjelmia ja järjestelmiä luomalla), eikä kone taustalla tarjoa paljonkaan älykkyyttä (vaikka onkin älykkäiden ihmisten luoma sähköinen automaatti, joka nykypäivänä sisältää suuren joukon ”apujärjestelmiä” jo sisälläänkin). Tietokone ensinnäkin osaa käsitellä vain **bittejä** eli kaksijärjestelmän numeroita, nollia ja ykkösiä. Bittejä voidaan toki yhdistää, esim. kahdeksalla bitillä voidaan ilmoittaa 256 eri lukua. Bitit ilmenevät koneessa sähköjännitteinä, esimerkiksi jännite välillä $0 - 0.8V$ voisi tarkoittaa nollaa ja jännite välillä $2.0 - 3.3V$ ykköstä.

Tietokone valmistetaan yksinkertaisista elektroniikkakomponenteista koostamalla. Toki kokonaisuus sisältää erittäin suuren määrän komponentteja, jotka muodostavat monimutkaisen, hierarkkisen rakenteen. Suunnittelussa ja valmistusteknologiassa on tultu pitkä matka tietokoneiden historian aikana. Peruskomponentit ovat kuitenkin yhä tänäkin päivänä yksinkertaisia, puolijohdetekniikalla valmistettuja, pienikokoisia elektronisia laitteita (esim. transistoreja, diodeja, johtimia) joista koostetaan **logiikkaportteja** (engl. *logic gate*) ja **muistisoluja** (engl. *memory cell*). Logiikkaportin tehtävä on suorittaa biteille jokin operaatio, esimerkiksi kahden bitin välinen AND (ts. jos portin kahteen sisääntuloon saapuu molempiin ykkönen, ulostuloon muodostuu ykkönen ja muutoin nolla). Muistisolun tehtävä puolestaan on tallentaa yksi bitti myöhempää käyttöä varten. Peruskomponenteista muodostetaan johtimien avulla yhdistelmiä, jotka kykenevät tekemään monipuolisempia operaatioita. Komponenttiyhdistelmiä voidaan edelleen yhdistellä, ja niin edelleen. Esimerkiksi 8-ytimisessä Intel Xeon 7500 -prosessorissa on valmistajan antamien tietojen mukaan yhteensä 2 300 000 000 transistoria, joista koostuva rakennelma pystyy tekemään biteille jo yhtä ja toista. Kuitenkin pohjimmiltaan kyseessä on ”vain” bittejä paikasta toiseen siirtelevä automaatti.

Nykyaikainen tapa suunnitella tietokoneen perusrakenne on pysynyt pääpiirteissään samana yli puoli vuosisataa. Käytettäköön tässä perusrakenteesta nimeä **Von Neumann -arkkitehtuuri** 1940-luvulla ensimmäisten tietokoneiden suunnittelussa vaikuttaneen henkilön mukaan. Kyseinen John Von Neumann ei toki keksinyt tietokonetta yksin, vaan perusarkkitehtuuri on monen henkilön pitkän työn tulos.

Kuvassa 1 esitetään Von Neumann -arkkitehtuuri, eli tietokoneen perusrakenne, kaikkein yleisimmällä tasolla, jonka komponenteilla on tietyt tehtävänsä. Tietokoneessa on **keskusyksikkö** (engl. *CPU, Central Processing Unit*) eli **prosessori** (engl. *processor*), **muisti** (engl. *memory*) ja **I/O-laitteita** (engl. *Input/Output modules*). Komponentteja yhdistää **väylä** (engl. *bus*). Keskusyksikkö hoitaa varsinaisen biteillä laskemisen, käyttäen apunaan muistia (myöhemmin varsin tarkoin selviävällä tavalla). I/O -laitteisiin lukeutuvat mm. näppäimistö, hiiri, näytönohjain, kovalevy, DVD, USB-tikut, verkkoyhteydet, printterit, ...

Väylää voi ajatella rinnakkaisina ”piuhoina” elektronisten komponenttien välillä. Kussakin piuhassa voi tietenkin olla kerrallaan vain yksi bitti, joten esimerkiksi 64 bitin siirtäminen kerrallaan vaatii 64 rinnakkaista sähköjohdinta. Piuhakokoelmia tarvitaan useita, että väylän ohjaus ja synkronointi voi toimia, erityisesti ainakin ohjauslinja, osoitelinja ja datalinja. Näillä voi olla kaikilla eri leveys. Esimerkiksi osoitelinjan leveys voisi olla 38 piuhaa (bittiä) ja datalinjan leveys 64 bittiä.

Prosessorin ulkopuolella kulkevan osoiteväylän leveys (bittipiuhojen lukumäärä) määrittelee tietokoneen fyysisen **osoiteavaruuden** (engl. *address space*) laajuuden eli montako muistiosoitetta tietokoneessa voi olla. Muistia ajatellaan useimmiten kahdeksan bitin kokoelmien eli **tavujen** (engl. *byte*) muodostamina lokeroina, joista jokaisella on oma osoite (biteiksi koodattu kokonaisluku $0, 1, 2, 3, \ldots$). Dataväylän leveys (bittipiuhojen lukumäärä) määrittelee kuinka paljon tietoa (eli peräkkäisiä tavuja) väylä korkeintaan voi siirtää kerrallaan. Tietokoneen **sananpituus** (engl. *word length*) on termi, jolla usein kuvataan tietyn tietokoneen kerrallaan käsittelemien bittien määrää (joka saattaa olla sama kuin ulkoisen väylän leveys). Tämä on hyvä tietää; kuitenkin tämän monisteen jatkossa tulemme käyttämään toista perinteistä määritelmää **sanalle** (engl. *word*): Sana olkoon meidän näin sopien aina kahden tavun paketti eli 16-bittinen kokonaisuus. Näin voidaan puhua myös tuplasanoista (engl. *double word*) 32-bittisinä kokonaisuuksina ja nelisanoista (engl. *quadword*) 64-bittisinä kokonaisuuksina.

Väylän kautta pääsee käsiksi moniin paikkoihin, ja osoiteavaruus jaetaan usein (laitteistotasolla) osiin siten, että tietty osoitteiden joukko tavoittaa **ROM-muistin** (engl. *Read-Only Memory*, vain luettavissa, tehtaalla lopullisesti kiinnitetty tai ainoastaan erityistoimenpitein muutettavissa), osa **RAM-muistin** (engl. *Random Access Memory*, jota voi sekä lukea että kirjoittaa ja jota ohjelmat normaalisti käyttävät), osa I/O-laitteet. Normaalit ohjelmat näkevät itse asiassa niitä varten luodun **virtuaalisen osoiteavaruuden** – tähän käsitteeseen palataan myöhemmin.

Tietokoneen prosessori toimii nopean kellopulssin ohjaamana: Aina kun kello ”lyö” eli antaa jännitepiikin (esim. 1 000 000 000 kertaa sekunnissa), prosessorilla on mahdollisuus aloittaa joku toimenpide. Toimenpide voi kestää yhden tai useampia kellojaksoja, eikä seuraava voi alkaa ennen kuin edellinen on valmis[2].

Myös väylä voi muuttaa piuhoissaan olevia bittejä vain kellopulssin lyödessä – väylän kello voi olla hitaampi kuin prosessorin, jolloin väylän toimenpiteet ovat harvemmassa. Ne kestävät muutenkin pidempään, koska sähkön on kuljettava pidempi matka ja aikaa kuluu ohjauslogiikan toimenpiteisiin. Operaatiot sujuvat nopeammin, jos väylää tarvitaan niihin harvemmin.

Jokainen mahdollinen toimenpide, jonka prosessori voi kerrallaan tehdä, on jotakin melko yksinkertaista — tyypillisimmillään vain rajatun bittimäärän (sähköjännitteiden) siirtäminen paikasta toiseen ("piuhoja pitkin"), mahdollisesti soveltaen matkan varrella jotakin yksinkertaista, ennaltamäärättyä digitaaliloogista operaatiota (joka perustuu siis komponenttien hierarkkiseen yhdistelyyn).

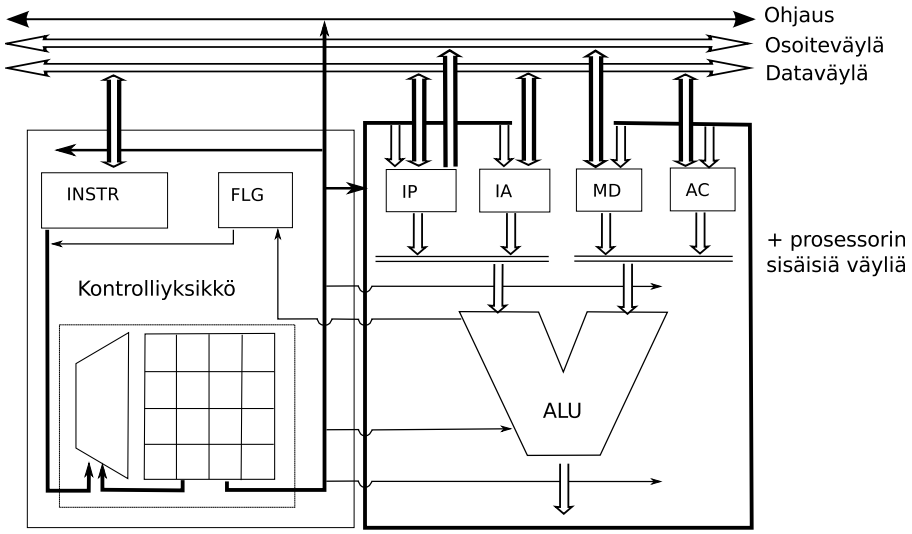
Kuva 2 tarkentaa vielä keskusyksikön jakautumista hierarkkisesti alemman tason komponentteihin. Näitä ovat **kontrolliyksikkö** (engl. *control unit*), **aritmeettislooginen yksikkö** (engl. *ALU, Arithmetic Logic Unit*) sekä moninaiset **rekisterit** (engl. *registers*). Tästä alempia laitteistohierarkian tasoja ei tällä kurssilla tarvitse ajatella, sillä tästä kuvasta jo löytyvät ne komponentit joihin ohjelmoija (eli sovellusohjelmien tai käyttöjärjestelmien tekijä) pääsee käsiksi. Alempien tasojen olemassaolo (eli koneen koostuminen yksinkertaisista elektronisista komponenteista yhdistelemällä) on silti hyvä tiedostaa.

Ylimalkaisesti sanottuna kontrolliyksikkö ohjaa tietokoneen toimintaa, aritmeettislooginen yksikkö suorittaa laskutoimitukset, ja rekisterit ovat erittäin nopeita muisteja prosessorin sisällä, erittäin lähellä muita prosessorin sisäisiä komponentteja. Rekisterejä tarvitaan, koska muistisoluthan ovat ainoita tietokoneen rakennuspalikoita, joissa bitit säilyvät pidempään kuin yhden hetken. Bittejä täytyy pystyä säilömään useampi hetki, ja kaikkein nopeinta tiedon tallennus ja lukeminen on juuri rekistereissä, jotka sijaitsevat prosessorin välittömässä läheisyydessä. Rekisterejä tarvitaan useita, ja joillakin niistä on tarkoin määrätyt roolit prosessorin toimintaan liittyen. Rekisterien kokonaismäärä (välttämättömien lisäksi) ja kunkin rekisterin sisältämien bittien määrä vaihtelee eri mallisten prosessorien välillä. Esimerkiksi Intel Xeon -prosessorissa rekisterejä on kymmeniä ja kuhunkin niistä mahtuu talteen 64 bittiä. Prosessorin sisällä on sisäisiä väyliä, joita kontrolliyksikkö ohjaa ja jotka ovat tietenkin paljon ulkoista väylää nopeampia bitinsiirtäjiä.

2.2 Suoritussykli (yhden ohjelman kannalta)

Nyt voidaan kuvan 2 terminologiaa käyttäen vastata riittävän täsmällisesti siihen, mikä itse asiassa on se ”yksittäinen toimenpide”, jollaisen prosessori voi aloittaa kellopulssin tullessa. Toimenpide on yksi kierros toistosta,

[2] Nykyiset prosessorit toki sisältävät teknologioita (esinouto, liukuhihnoitus, ennakoiva suoritus), joilla voidaan suorittaa useita käskyjä limittäin; tämän miettimisestä ei kuitenkaan ole käyttöjärjestelmäkurssin mielessä juuri hyötyä, joten pidämme esitystavan selkeämpänä valehtelemalla että operaatiot ovat vain peräkkäisiä; loogisessa mielessä joka tapauksessa ulospäin näyttää siltä!



Kuva 2: *Yksinkertaistettu kuva kuvitteellisesta keskusyksiköstä: kontrolliyksikkö, ALU, rekisterit, ulkoinen väylä ja sisäiset väylät. Kuva mukailee Tietotekniikan perusteet -luentomonistetta [2].*

jonka nimi on **nouto-suoritus -sykli** (engl. *fetch-execute cycle*)[3]. Mikäli prosessori on suorittanut aiemman toimenpiteen loppuun, se voi aloittaa seuraavan kierroksen, joka sisältää seuraavat päävaiheet (tässä esitetään sykli vasta yhden ohjelman kannalta; myöhemmin tätä hiukan täydennetään):

1. Käskyn **nouto** (engl. *fetch*): Prosessori noutaa muistista seuraavan konekielisen **käskyn** (engl. *instruction*) eli toimintaohjeen. Karkeasti ottaen käsky on bittijono, johon on koodattu prosessorin seuraava tehtävä. **Konekieli** (engl. *machine language*) on näiden käskyjen eli bittijonojen syöttämistä peräkkäin. Prosessori "ymmärtää" konekieltä, joka on kirjoitettu peräkkäisiin muistipaikkoihin. Ohjelmat voivat olla pitkiä, joten niitä ei voi säilyttää kovin lähellä prosessoria. Käskyt sijaitsevat siis muistissa, josta seuraava aina noudetaan väylän kautta[4]. Jotta noutaminen voi tapahtua, täytyy väylän osoitelinjaan kytkeä muistipaikan osoite. Jotta bittejä voidaan kytkeä johonkin, ne pitää tietenkin olla jossakin säilössä. Seuraavan käskyn osoite on tallessa rekisterissä, jonka nimi on **käskyosoitin** (engl. *IP, instruction pointer*). Toinen nimi samalle asialle (kirjoittajasta riippuen) voisi olla **ohjelmalaskuri** (engl. *PC, program counter*). Siis elektroniikka kytkee IP -rekisterin osoitelinjaan, odottaa että väylän datalinjaan välittyy muistipaikan sisältö, ja kytkee datalinjan rekisteriin, joka tunnetaan nimellä käskyrekisteri (engl. *INSTR, IR, instruction register*) tai vastaavaa. Seuraava käsky on nyt noudettu ja sitä vastaava bittijono on siis INSTR-rekisterin sisältönä. Prosessori noutaa muistista mahdollisesti myös käskyn tarvitsemat **operandit** eli luvut, joita operaatio tulee käyttämään syötteenään. [5]

2. Käskyn **suoritus** (engl. *execute*): Käskyrekisterin sisältö kytkeytyy kontrolliyksikön elektroniikkaan, ja yhteistyössä aritmeettisloogisen yksikön kanssa tapahtuu tämän yhden käskyn suorittaminen. Käsky saattaa edellyttää myös muiden rekisterien sisällön käyttöä tai korkeintaan muutaman lisätiedon noutoa muistista (väylän kautta, osoitteista jotka määräytyvät tiettyjen rekisterien sisällön perusteella; tästä on luvassa tarkempi selvitys seuraavassa luvussa esimerkkien kautta).

3. Tuloksen säilöminen ja tilan päivitys: Ennen seuraavan kierroksen alkua on huomattava, että käskyn suorituksen jälkeen prosessorin ulospäin näkyvä tila muuttuu. Ainakin käskyosoitinrekisterin eli IP:n sisältö on uusi: siellä on nyt taas seuraavaksi suoritettavan konekielisen käskyn muistiosoite. Usein erityisesti laskutoimituksissa muuttuvat tietyt bitit **lippurekisterissä** (engl. *FLAGS, FLG, FR, flag register*), josta käytetään myös englannin kielistä nimeä "Program status word, PSW". Jotta laskutoimituksissa olisi järkeä, niiden tulokset useimmiten tallentuvat johonkin rekisteriin (tai joskus suoraan muistiin, mikä taas edellyttää väylän käyttöä ja sitä että tulokselle tarkoitettu muistiosoite oli tallessa jossakin rekisterissä).

4. Sykli alkaa jälleen alusta.

Kohdassa kolme sanottiin että käskyosoittimen sisältö on uusi. Se, kuinka IP muuttuu, riippuu siitä millainen käsky suoritettiin:

[3] Tämä on taas hienoinen valhe asian yksinkertaistamiseksi. Itse asiassa nykyprosessorit suorittavat käskynsä ns. mikrokoodina, johon liittyy decode -vaihe, ja sykliä sanotaan joskus vastaavasti fetch-decode-execute -sykliksi. Lisäksi edellisessä alaviitteessä mainitut nopeutusteknologiat tekevät sisäisestä toiminnasta monimutkaisempaa. Näilläkään ei ole vaikutusta siihen, miltä toiminta näyttää ulospäin.

[4] Valehtelu jatkuu toistaiseksi: myöhemmin puhutaan ns. välimuisteista. Pidetään kuitenkin asia toistaiseksi yksinkertaisena.

[5] Tämä on perusidea, mutta yksinkertaistus, koska konekielisen käskyn pituus voi vaihdella ja tässä kohtaa saatettaisiin siis noutaa useita peräkkäisiä tavuja. Lisäksi prosessorin elektroniikka tekee tässä kohtaa muitakin valmisteluja.

- **Jokin peräkkäissuoritteinen käsky** kuten laskutoimitus tai datan siirto paikasta toiseen → "IP":ssä on juuri suoritettua käskyä seuraavan käskyn muistiosoite (siis siinä järjestyksessä kuin käskyt on talletettu muistiin).
- **Ehdoton hyppykäsky** → "IP":n sisällöksi on ladattu juuri suoritetun käskyn yhteydessä kerrottu uusi muistiosoite, esim. silmukan ensimmäinen käsky tms.
- **Ehdollinen hyppykäsky** → "IP":n sisällöksi on ladattu käskyssä kerrottu uusi osoite, mikäli käskyssä kerrottu ehto toteutuu; muutoin "IP" osoittaa seuraavaan käskyyn samoin kuin peräkkäissuorituksessa. (Ehto tulkitaan koodatuksi FLAGS-lippurekisterin johonkin/joihinkin bitteihin.)
- **Aliohjelmakutsu** → "IP":n sisältönä on käskyssä kerrottu uusi osoite (jonka tulee olla kutsuttavan aliohjelman ensimmäinen käsky; aliohjelmakutsussa prosessori tekee muutakin, mitä käsitellään kohta tarkemmin)
- **Paluu aliohjelmasta** → "IP" osoittaa taas siihen ohjelmaan, joka aiemmin suoritti kutsun, erityisesti kyseessä on aliohjelmakutsua välittömästi seuraava kutsuvan ohjelman käsky. (myöhemmin nähdään, miten tämä on voitu käytännössä pitää muistissa)

Aliohjelmakutsu ja paluu ovat normaalia käskyä hieman monipuolisempia toimenpiteitä, joissa prosessori käyttää myös pinomuistia (tarkennus tulee kohtapuoleen).

Esitetään muutamia huomioita rekisterien rooleista. Käskyrekisteri INSTR on esimerkki **ohjelmoijalle näkymättömästä rekisteristä**. Ohjelmoija ei voi mitenkään tehdä koodia, joka vaikuttaisi "suoraan" tällaiseen näkymättömään rekisteriin – sellaisia konekielikäskyjä kun ei yksinkertaisesti ole. Prosessori käyttää näitä rekisterejä sisäiseen toimintaansa. Näkymättömiä rekisterejä ei käsitellä tällä kurssilla enää sen jälkeen, kun suoritussykli ja keskeytykset on käyty läpi. Jonkinlainen "INSTR" on olemassa jokaisessa prosessorissa, jotta käskyjen suoritus olisi mahdollista. Lisäksi on mahdollisesti muita käyttäjälle näkymättömiä rekisterejä tarpeen mukaan. Niiden olemassaolo on hyvä tietää lähinnä esimerkkinä siitä että prosessorin toiminta, vaikkakin nykyään monipuolista ja taianomaisen tehokasta, ei ole perusidealtaan mitään kovin mystistä. Bittijonoiksi koodatut syöttötiedot ja jopa itse käsky noudetaan tietyin keinoin prosessorin sisäisten komponenttien välittömään läheisyyteen, josta ne kytketään syötteeksi näppärän insinöörijoukon kehittelemälle digitaalilogiikkapiirille, joka melko nopeasti muodostaa ulostulot ennaltamäärättyihin paikkoihin. Sitten tämä vain toistuu, nykyisin sangen tiuhaan tahtiin.

Käytännössä olemme kiinnostuneempia **ohjelmoijalle näkyvistä rekistereistä** (engl. *visible registers*). Jotkut näistä, kuten käskyosoitin IP ja lippurekisteri FLAGS, ovat sellaisia ettei niihin suoraan voi asettaa uutta arvoa, vaan ne muuttuvat välillisesti käskyjen perusteella. Jotkut taas ovat **yleiskäyttöisiä rekisterejä** (engl. *general purpose registers*), joihin voi suoraan ladata sisällön muistista tai toisista rekistereistä, ja joita voidaan käyttää laskemiseen tai muistin osoittamiseen. Joidenkin rekisterien päärooli voi olla esim. yksinomaan kulloisenkin laskutoimituksen tuloksen tallennus tai sitten yksinomaan muistin osoittaminen. Kaikki tämä riippuu suunnitteluvaiheessa tehdyistä ratkaisuista ja kompromisseista (mitä vähemmän kytkentöjä, sen yksinkertaisempi, pienempi ja halvempi prosessori – mutta kenties vaivalloisempi ohjelmoida).

Joidenkin rekisterien käyttö on sallittu vain käyttöjärjestelmälle. Sanotaan näitä vaikka **järjestelmärekistereiksi** (engl. *system registers*). Jatkossa keskitymme pääasiassa **käyttäjälle näkyviin rekistereihin** (engl. *user visible registers*). Näihin asioihin on kuitenkin palattava vielä vähän myöhemmin, kunhan saadaan ensin hanskaan sellaiset käsitteet kuin keskeytykset ja käyttöjärjestelmätila.

Tässä vaiheessa pitäisi olla jo selvää, että yksi prosessori voi suorittaa kerrallaan vain yhtä ohjelmaa, joten monen ohjelman yhdenaikainen käyttö ilmeisesti vaatii jotakin erityistoimenpiteitä. Yksi käyttöjärjestelmän tehtävä ilmeisesti on käynnistää käyttäjän haluamia ohjelmia ja jollain tavoin jakaa ohjelmille vuoroja prosessorin käyttöön, niin että näyttäisi siltä kuin olisi monta ohjelmaa "yhtäaikaa" käynnissä.

2.3 Prosessorin toimintatilat ja lippurekisteri

Eräs tarve tietokoneiden käytössä on eriyttää kukin normaali käyttäjän ohjelma omaan "karsinaansa", jotta ne eivät vahingossakaan sotke toisiaan tai järjestelmää. Tätä tarkoitusta varten prosessorissa on erikseen järjestelmärekisterejä ja toimintoja, joihin pääsee käsiksi vain käyttöjärjestelmän suoritettavissa olevilla konekielikäskyillä. Koska sama prosessorilaite suorittaa sekä käyttäjän ohjelmia että käyttöjärjestelmäohjelmaa, joilla on eri valtuudet, täytyy prosessorin voida olla ainakin kahdessa eri toimintatilassa, ja tilan on voitava vaihtua tarpeen mukaan.

Ohimennen voimme nyt ymmärtää, mitä tapahtuu, kun tietokoneeseen laitetaan virta päälle: prosessori käynnistyy niin sanottuun **käyttöjärjestelmätilaan** (engl. *kernel mode*). Muita nimiä tälle olisi suomeksi "todellinen

tila" (engl. *real mode*) tai "valvojatila" (engl. *supervisor mode*). Käynnistyksen jälkeen prosessori alkaa suorittaa ohjelmaa ROM-muistista (kiinteästi asetetusta fyysisestä muistiosoitteesta alkaen). Oletuksena on, että ROM:issa oleva, yleensä pienehkö, ohjelma lataa varsinaisen käyttöjärjestelmän joltakin ulkoiselta tallennuslaitteelta. Olet ehkä huomannut, että kotitietokoneiden ROM:issa on yleensä BIOS-asetusten säätöohjelmisto, jolla käynnistyksen yhteydessä voi määrätä fyysisen laitteen, jolta käyttöjärjestelmä pitäisi koettaa löytää (korppu, DVD, CD-ROM, kovalevyt, USB-tikku jne...). BIOS tarjoaa myös muita asetuksia, jotka säilyvät virran katkaisun jälkeen (esim. pariston avulla). Käynnistettäessä tietokone siis on vain tietokone, eikä esim. "Mac OS-X, Windows tai Linux -kone".

Käyttöjärjestelmän latausohjelmaa etsitään alkeellisilla, standardoiduilla laiteohjauskomennoilla tietystä paikasta fyysistä tallennetta. Siellä pitäisi olla siis nimenomaiselle prosessorille käännetty konekielinen latausohjelma, jolla on sitten vapaus säädellä kaikkia prosessorin systeemitoimintoja ja toimintatiloja. Sen pitäisi myös alustaa tietokoneen fyysinen muisti tarkoituksenmukaisella tavalla, ladata muistiin tarvittavat ohjelmistot, tehdä koko liuta muitakin valmisteluja sekä vielä lopulta tarjota käyttäjille mahdollisuus kirjautua sisään koneelle ja alkaa suorittamaan hyödyllisiä tai viihteellisiä ATK-sovelluksia. Esimerkiksi Unix-käyttöjärjestelmä jää käynnistyttyään odottamaan "loginia" eli käyttäjätunnuksen ja salasanan syöttöä päätteeltä, minkä jälkeen käyttöjärjestelmän login-osio käynnistää tunnistetulle käyttäjälle ns. "**kuoren**" (engl. *shell*) jota vakiintuneesti kutsutaan "shelliksi" myös suomen kielellä (esim. "bash", "tcsh" "ksh", tms., asennuksen ja valinnan mukaan). Englismi "shell" on jopa niin vakiintunut, että käytämme jatkossa ainoastaan sitä, kun puhumme "kuoresta". Käyttöjärjestelmä ohjaa päätteen näppäinsyötteet shellille ja shellin printtitulosteet näytölle. Käyttöliittymä voi toki olla graafinenkin, jolloin puhutaan **ikkunointijärjestelmästä** (engl. *windowing system*). Ikkunointi voi olla osa käyttöjärjestelmää, tai se voi olla erillinen ohjelmisto, kuten Unix-ympäristöissä usein käytetty ikkunointijärjestelmä nimeltä X. Nykypäivänä käyttäjä myös edellyttänee, että hänelle tarjotaan "**työpöytä**" (engl. *desktop manager*), joka on kuitenkin jo melko korkealla tasolla varsinaiseen käyttöjärjestelmän ytimeen nähden, eikä siten millään tavalla tämän kurssin aihepiirissä.

Kirjautumisen jälkeen kaikki käyttäjän ohjelmat toimivat prosessorin ollessa **käyttäjätilassa** (engl. *user mode*) jolle käytetään myös nimeä "suojattu tila" (engl. *protected mode*). Jälkimmäinen nimi viitannee siihen, että osa prosessorin toiminnoista on tällöin suojattu vahingossa tai pahantahtoisesti tapahtuvaa väärinkäyttöä vastaan. Prosessorin tilaa (käyttäjä-/käyttöjärjestelmätila) säilytetään jossakin yhden bitin kokoisessa sähkökomponentissa prosessorin sisällä. Tämä tila (esim. 0==käyttöjärjestelmä, 1==käyttäjätila) voi olla esim. yhtenä bittinä lippurekisterissä. (Itse asiassa esim. x86-64 arkkitehtuuri tarjoaa neljä suojaustasoa, 0–3, joista käyttöjärjestelmän tekijä voi päättää käyttää kahta (0 ja 3) tai useampaa. Olennaista kuitenkin on, että aina on olemassa vähintään kaksi – käyttäjän tila ja käyttöjärjestelmätila. Puhutaan jatkossa näistä kahdesta.)

FLAGS tallentaa myös muut prosessorin tilaan liittyvät on/off -liputukset. Prosessoriarkkitehtuurin määritelmä kertoo, miten mikäkin käsky muuttaa FLAGS:iä. Kolme tyypillistä esimerkkiä:

- Yhteenlaskussa (bittilukujen "alekkain laskeminen") voi jäädä muistibitti yli, jolloin nostetaan "carry flag" lippu – se on tietty bitti FLAGS-rekisterissä, ja sen nimi on usein kirjallisuudessa "CF". Samalla nousee luultavasti "overflow" "OF" joka tarkoittaa lukualueen ylivuotoa (tulos olisi vaatinut enemmän bittejä kuin rekisteriin mahtuu).
- Vähennyslaskussa ja vertailussa (joka on olennaisesti vähennyslasku ilman tuloksen tallentamista!) päivittyy FLAGS:ssä bitti, joka kertoo, onko tulos negatiivinen – nimi on usein "negative flag", "NF" (tai vastaavaa...)
- Jos jonkun operaation tulos on nolla (tai halutaan koodata joku tilanne vastaavasti) asettuu "zero flag", nimenä usein "ZF".

Liput ovat mukana prosessorin syötteessä aina kunkin käskyn suorituksessa, ja suoritus on monesti erilainen lippujen arvoista riippuen. Monet ohjelmointirakenteet, kuten ehto- ja toistorakenteet perustuvat jonkun testikäskyn suorittamiseen, ja vaikkapa ehdollisen hyppykäskyn suorittamiseen (hyppy tapahtuu vain jos tietty bitti FLAGS:ssä on asetettu). Käyttöjärjestelmälle varatut prosessoriominaisuudet eivät ole käytettävissä silloin kun FLAGS:n käyttäjätilalippu ei niitä salli.

Nykyaikaisissa prosessoreissa on myös muita käyttäjän tai käyttöjärjestelmän vaihdeltavissa olevia toimintatiloja, jotka vaikuttavat esimerkiksi siihen, miten suurta osaa rekisterien biteistä käytetään operaatioihin, ollaanko jossakin taaksepäin-yhteensopivuustilassa tai vastaavassa, ja sen sellaista, mutta niihin ei ole mahdollisuutta eikä tarvetta syventyä tällä kurssilla. *Olennaista on ymmärtää käyttöjärjestelmätilan ja käyttäjätilan erilaisuus fyysisen laitteen tasolla.* Siihen perustuu moniajo, virtuaalimuistin käyttö ja suuri osa tietoturvasta. Yksityiskohtiin palataan myöhemmin. Tässä vaiheessa riittäköön vielä seuraava ajatusmalli:

- Käyttöjärjestelmä ottaa tietokoneen hallintaansa pian käynnistyksen jälkeen, mutta ei aivan heti; laitteen

ROM-muistissa on oltava riittävä ohjelma käyttöjärjestelmän ensimmäiseksi suoritettavan osion lataamiseksi esim. kovalevyn alusta.

- Käyttöjärjestelmällä on vapaus käsitellä kaikkia prosessorin ominaisuuksia.
- Käyttöjärjestelmän täytyy käynnistää normaalit ohjelmat ja hoitaa ne toimimaan prosessorin käyttäjätilassa.
- Käyttöjärjestelmä isännöi tavallisia "käyttäjämaan" ohjelmia ja mahdollistaa moniajon yhteistyössä prosessorin kanssa (myöhemmin opittavalla menettelyllä).

2.4 Prosessorin fyysinen linkki ulkoiseen väylään

Nykyinen mikroprosessori on fyysiseltä kooltaan pieni – sen pitää olla pieni, koska sähköisen signaalin nopeus on rajoitettu, ja mitä pidempiä matkoja sähkön pitää matkata, sen hitaampaa on toiminta. Pienen pieni prosessori sijoitetaan tyypillisesti suhteellisen pieneen koteloon, joka voidaan lämpöä johtavasta kohdasta yhdistää jäähdytysjärjestelmään (vaikkapa tuuletin ja metallisiili). Nykyinen prosessori kuumenee niin paljon, että ilman jäähdytystä se menisi lähes välittömästi rikki. Pieni kotelo on kiinni isommassa kohtelossa, joka on kätevä asentaa kiinni muuhun laitteistoon. Koteloinnissaan olevan prosessorin kommunikaatio muun laitteiston kanssa voi tapahtua vain sähköjohtimia pitkin, joten johtavasta materiaalista on tehty "piuhat" pienen kotelon sisältä suuremman kotelon ulkopuolelle. Suuremmassa kotelossa jokainen piuha ilmenee kuparisena nastana, joka voidaan liittää emolevyssä (eräs kokonaislaitteiston osa) olevaan vastinkappaleeseen. Nastojen sijoittelulle on standardeja, jotta eri prosessorivalmistajat voivat koteloida prosessorinsa yhteensopivasti muiden laitteisto-osien valmistajia varten. Moderneissa prosessoreissa on nastoja melko paljon (useita satoja), ja fyysinen sijoittelu riippuu käytetystä standardista, mutta merkitykset ovat prosessorimerkistä riippumatta useimmiten samat:

- ulkoisen kommunikaation data-, osoite- ja ohjausbitit (Ennenvanhaan tämä tarkoitti ulkoisen väylän bittejä; uusimmissa prosessorimalleissa kuitenkin osa ulkoisen väylän ja muistin ohjauksesta tapahtuu prosessorisirulla, koska se on tämän päivän teknologialla tehokkaampaa ja edullisempaa; prosessorikoteloinnin nastat voidaan siis liittää suoraan muistikampoihin ja joihinkin ulkoisiin laitteisiin kuten näytönohjaimeen.)
- kellopulssit
- keskeytysilmoitukset
- lämpötilasensorit ym.
- käyttöjännite virtalähteeltä (nykyisin jännite voi tulla prosessorin eri osioihin eri nastojen kautta)
- maadoitus (mahdollisesti useita nastoja)

Väylä, väylän ohjaus, keskusmuisti ja kaikki I/O -laitteet ovat "historiallisissa tietokoneissa" prosessorin koteloinnin ulkopuolella. Nykyisissä tilanne voi siis olla hieman monimutkaisempi, mutta tämä ei vaikuta siihen, kuinka voimme abstraktisti ajatella arkkitehtuuria. Sulautettuja järjestelmiä varten voidaan prosessoreita valmistaa myös *system-on-a-chip* -periaatteella, jolloin koko tietokonearkkitehtuurin toteutus väylineen päivineen, jopa ROM-ohjelmistollakin varustettuna, valmistetaan yhdelle sirulle. Tällainen sirusysteemi voidaan suunnitella esim. tiettyä kännykkämallia, MP3-soitinta tai digikameraa varten. Tähän suuntaan on viime aikoina menty myös uusissa yleiskäyttöisissä prosessoreissa. *Ohjelman suorituksen kannalta kaikki näyttää kuitenkin yhä samalta*, olipa prosessori koteloitu erikseen tai yhdessä muiden laitteiden kanssa. On siis syytä ymmärtää Von Neumann -arkkitehtuuri, koska ainakin toistaiseksi tietokone näyttäytyy ohjelmoijalle juuri sen näköisenä laitteena.

2.5 Käskykanta-arkkitehtuureista

Tietty **prosessoriarkkitehtuuri** tarkoittaa niitä tapoja, joilla sen mukaisesti rakennettu fyysinen prosessorilaite toimisi: Mitä toimenpiteitä se voi tehdä (eli millaisia käskyjä sillä voi suorittaa), mistä ja mihin mikäkin toimenpide voi siirtää bittejä, ja miten mikäkin toimenpide muunnetaan (tavallisesti muutaman tavun mittaiseksi) bittijonoksi, joka sisältää **operaatiokoodin**, (engl. *opcode, "operation code"*) ja tiedot operoinnin kohteena olevista rekistereistä/muistiosoitteista. Prosessoriarkkitehtuurissa kuvataan mahdollisten, prosessorin ymmärtämien käskyjen ja operandiyhdistelmien joukko. Tätä kutsutaan nimellä **käskykanta** tai käskyjoukko (engl. *instruction set*). Toinen nimi prosessoriarkkitehtuurille voisi siis olla **käskykanta-arkkitehtuuri** (engl. *ISA, instruction set architecture*).

Prosessoriarkkitehtuureita ja niitä toteuttavia fyysisiä prosessorilaitteita on markkinoilla monta, ja niissä on merkittäviä eroja, mutta kaikissa on jollakin tavoin toteutettu edellä kuvaillut pakolliset piirteet, ja yleisrakenteeltaan ne vastaavat tänä päivänä sekä näköpiirissä olevassa tulevaisuudessa 1940-lukulaista Von Neumannin arkkitehtuuria! Kunkin prosessorin käskykanta-arkkitehtuuri kuvataan prosessorivalmistajan toimittamassa manuaalissa, jonka tarkoituksena on antaa riittävä tieto minkä tahansa toteutettavissa olevan ohjelman tekemiseen niitä nimenomaisia piinkappaleita käyttämällä, joita prosessoritehtaasta pakataan ulos. Mainittakoon myös nimeltä matemaatinen ala nimeltä **laskennan teoria**, joka antaa muurinlujia tuloksia siitä, mitä nykyisenkaltaisella tietokoneella voidaan tehdä ja mitä sillä toisaalta ei yksinkertaisesti voida tehdä. Mm. jokainen tietokone pystyy ratkaisemaan samat tehtävät kuin mikä tahansa toinen tietokone (jos unohdamme sellaiset ”pikkuseikat” kuin ratkaisuun kuluva aika). Tästä lisää algoritmikursseilla.

3 Konekielisen ohjelman suoritus

Edellisessä luvussa ”kerrattiin” varsin yleisellä tasolla esitietoja siitä, millainen laite tietokone yleisesti ottaen on. Tarkempi tietämys on hankittava oma-aloitteisesti tai tietotekniikan laiteläheisillä kursseilla. Tässä luvussa valaistaan konekielistä ohjelmointia lisää käytännön esimerkkien kautta. Esimerkkiarkkitehtuuri on ns. x86-64 -arkkitehtuuri. Yhtä hyvin esimerkkinä voisi olla mikä tahansa, jolla olisi helppo pyöräytellä esimerkkejä. Valinta tehdään kesällä 2011 tällä tavoin, koska Jyväskylän yliopiston Tietohallintokeskuksen kone “jalava.cc.jyu.fi“, johon opiskelijat pääsevät helposti käsiksi ja jossa kurssin harjoitukset voidaan tehdä, on tällä hetkellä malliltaan useampiytiminen Intel Xeon, jonka arkkitehtuuri on nimenomaan x86-64. Toivottavasti nykyaikaisen prosessorin käsittely on motivoivaa ja tarjoaa teorian lisäksi käytännön kädentaitoja tulevaisuutta varten.

3.1 Esimerkkiarkkitehtuuri: x86-64

Hieman x86-64:n taustaa: Prosessoriteknologiaan keskittyvä yritys nimeltä Intel on julkaissut mm. toisiaan seuranneet prosessorimallit (ja arkkitehtuurit) nimeltä 8086, 80186, 80286, 80386, 80486 ja Pentium. Intel itse on luonut sittemmin merkittävällä tavoin erilaisen prosessoriarkkitehtuurin nimeltä IA-64, jonka ei voi sanoa enää olevan suora perillinen edellisistä. Pitkäaikainen kilpailija ja ”klooniprosessoreja” valmistanut AMD esitteli kuitenkin ensimmäisenä arkkitehtuurin, joka perustuu vanhaan Intel-jatkumoon, mutta tuo uusia ominaisuuksia niin paljon, että pystyi kilpailemaan markkinoilla Intelin omaa erilaista uutuutta vastaan. Tällä kertaa Intel onkin ”kloonannut” AMD64-arkkitehtuurin nimikkeellä Intel 64, ja valmistaa prosessoreja, joissa AMD64:lle käännetty konekieli toimii lähes identtisesti. Koska Intel 64 ja AMD64 ovat lähes samanlaisia, niille on muodostunut yhteisnimi “x86-64“, joka kuvaa periytymistä Intelin x86-sarjasta ja leimallista 64-bittisyyttä (eli sitä, että rekistereissä ja muistiosoitteissa on 64 bittiä rivissä). Joitakin eroja on, mutta lähinnä niillä on merkitystä yhteensopivien kääntäjien valmistajille. Käytettäköön jatkossa siis arkkitehtuurien yhteisnimeä x86-64. Muista erilaisista nykyisistä prosessoriarkkitehtuureista mainittakoon ainakin IBM Cell (mm. Playstation 3:n multimediamylly) sekä ARM-sarjan prosessorit (jollainen löytyy monista sulautetuista järjestelmistä kuten kännyköistä).

Haasteelliseksi x86-64:n käyttämisen kurssin esimerkkinä tekee muun muassa se, että arkkitehtuuria ei ole suunniteltu puhtaalta pöydältä, vaan taaksepäin-yhteensopivaksi. Esimerkiksi 1980-luvulla tehdyt ja konekieleksi käännetyt 8086-arkkitehtuurin ohjelmat toimivat muuttamattomina x86-64 -koneissa, vaikka välissä on ollut useita prosessorisukupolvia teknisine harppauksineen. Käskykannassa ja rekisterien nimissä nähdään siis joitakin historiallisia jäänteitä, joita tuskin olisi tullut mukaan täysin uutta arkkitehtuuria suunniteltaessa.

Käyttäjän näkemät rekisterit x86-64 -arkkitehtuurissa

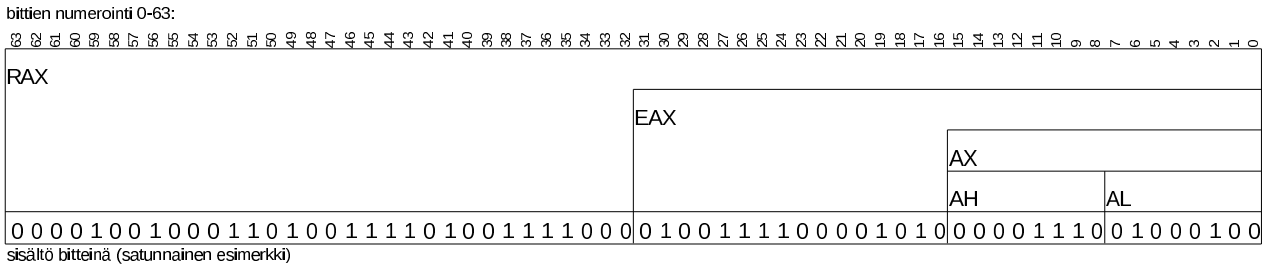
Nyt toivottavasti on riittävästi pohjatietoa, että voidaan vain esimerkinomaisesti listata eräässä prosessorissa käytettävissä olevia rekisterejä merkityksineen niillä lyhyillä nimillä, jotka prosessorivalmistaja on antanut. Taulukossa 1 on suurin osa rekistereistä, joita ohjelmoija voi käyttää Intelin Xeon -prosessorissa (tai muussa x86-64 arkkitehtuurin mukaisessa prosessorissa) aidossa 64-bittisessä tilassa. Yhteensopivuustiloissa olisi käytössä vain osa näistä rekistereistä, ja rekisterien biteistä käytettäisiin vain 32-bittistä tai 16-bittistä osaa, riippuen siitä monenko vuosikymmenen takaiselle x86-prosessorille ohjelma olisi käännetty.

Jokaisessa x86-64:n rekisterissä voidaan säilyttää 64 bittiä. Rekistereistä voidaan käyttää joko kokonaisuutta tai 32-bittistä, 16-bittistä tai jompaa kumpaa kahdesta 8-bittisestä osasta. Kuvassa 3 on esimerkiksi RAX:n osat ja niiden nimet; bitit on numeroitu siten, että 0 on vähiten merkitsevä ja 63 eniten merkitsevä bitti. Esim. yhden 8-bittisen ASCII-merkin käsittelyyn riittäisi “AL“, 32-bittiselle kokonaisluvulle (tai 4-tavuiselle Unicode-merkille) riittäisi “EAX“, ja 64-bittinen kokonaisluku tai muistiosoite tarvitsisi koko rekisterin “RAX“.

Jatkossa keskitytään lähinnä yleiskäyttöisiin kokonaislukurekistereihin. Käsittelemättä jätetään liukulukulaskentaan ja multimediakäyttöön tarkoitetut rekisterit (“FPR0-FPR7“, “MMX0-MMX7“ ja “XMM0“-“XMM15“). Esimerkiksi siinä vaiheessa, kun on kriittistä tehdä aiempaa tarkempi sääennuste aiempaa nopeammin, saattaa olla ajankohtaista opetella “FPR0-7“-rekisterit ja niihin liittyvä käskykannan osuus. Siinä vaiheessa, kun haluaa tehdä naapurifirmaa hienomman ja tehokkaamman 3D-koneiston tietokonepelejä tai lentosimulaattoria varten, on syytä tutustua multimediarekistereihin. Aika pitkälle ”tarpeeksi tehokkaan” ohjelman tekemisessä pääsee käyttämällä liukuluku- ja multimediasovelluksissa jotakin valmista kääntäjää, kirjastoa ja/tai virtuaalikonetta. Joka tapauksessa ohjelman *suoritusnopeus perustuu kaikista eniten algoritmien ja tietorakenteiden valintaan, ei jonkun algoritmin konekielitoteutukseen.* Lisäksi *nykyiset kääntäjät pystyvät ns. optimoimaan käännetyn ohjelman, eli luomaan juuri sellaiset konekieliset komennot jotka toimivat erittäin nopeasti.* Mutta älä koskaan sano ettei koskaan. . . voihan sitä päätyä töihin vaikka firmaan, joka nimenomaan toteuttaa noita kirjastoja, kääntäjiä

Taulukko 1: *x86-64:n 64-bittisen tilan rekisterejä.*

	Toiminnanohjausrekisterit:
RIP	Instruction pointer, ”IP”
RFLAGS	Flags, ”PSW”
	Yleisrekisterejä datalle ja osoitteille:
RAX	Yleisrekisteri; ”akkumulaattori”
RBX	Yleisrekisteri; ”epäsuora osoite”
RCX	Yleisrekisteri; ”laskuri”
RDX	Yleisrekisteri
RSI	Yleisrekisteri; ”lähdeindeksi”
RDI	Yleisrekisteri; ”kohdeindeksi”
RBP	Nykyisen aliohjelman pinokehyksen kantaosoitin
RSP	Osoitin suorituspinon huippuun
R8	Yleisrekisteri
R9	Yleisrekisteri
R10	Yleisrekisteri
R11–15	Vielä 5 kpl Yleisrekisterejä
	Muita rekisterejä:
MMX0-MMX7 /FPR0-FPR7	8 kpl Multimedia-/liukulukurekisterejä
YMM0-YMM15 /XMM0-XMM15	16 kpl Multimediarekisterejä
MXCSR ym.	Multimedia- ja liukulukulaskennan ohjausrekisterejä



Kuva 3: *x86-64 -prosessoriarkkitehtuurin erään yleisrekisterin jako aitoon 64-bittiseen osaan (”R”), 32-bittiseen puolikkaaseen (”E”), 16-bittiseen puolikkaan puolikkaaseen sekä alimman puolikkaan korkeampaan tavuun (high, ”H”) ja matalampaan tavuun (low, ”B”). Jako johtuu x86-sarjan historiallisesta kehityksestä 16-bittisestä 64-bittiseksi ja taaksepäinyhteensopivuuden säilyttämisestä.*

tai virtuaalikoneita [6].

Tällaisia rekisterejä siis x86-64 -tietokoneen sovellusohjelmien konekielisessä käännöksessä voidaan nähdä ja käyttää. Ne ovat esimerkkiarkkitehtuurimme käyttäjälle näkyvät rekisterit. Käyttöjärjestelmäkoodi voi käyttää näiden lisäksi systeemirekisterejä ja systeemikäskyjä, siis prosessorin ja käskykannan osaa, joilla muistinhallintaa, laitteistoa ja ohjelmien suojausta hallitaan. Systeemirekisterejä on esim. AMD64:n spesifikaatiossa eri tarkoituksiin yhteensä 50 kpl. Jos käyttäjän ohjelma yrittää jotakin niistä käyttää, seuraa suojausvirhe, ja ohjelma kaatuu saman tien (suoritus palautuu käyttöjärjestelmän koodiin). Tällä kurssilla nähdään esimerkkejä lähinnä käyttäjätilan sovellusten koodista. Käyttäjän ja käyttöjärjestelmän rekisterien lisäksi prosessorissa on oletettavasti sisäisiä rekisterejä väyläosoitteiden ja käskyjen väliaikaisia tallennuksia varten, mutta jätetään ne tosiaan tällä kertaa maininnan tasolle.

3.2 Konekieli ja assembler

Konekielen bittijonoa on järkevää tuottaa vain kääntäjäohjelman avulla. Käsityönä se olisi mahdottoman hankalaa – kielijärjestelmiä ja automaattisia kääntäjäohjelmia on aina tarvittu, ja siksi niitä on ollut olemassa lähes yhtä pitkään kuin tietokoneita. Sovellusohjelmoija pääsee lähimmäksi todellista konekieltä käyttämällä ns. **symbolista konekieltä** eli **assembleria** / **assemblyä** (engl. *assembly language*), joka muunnetaan bittijonoksi **assemblerilla** (engl. *assembler*) eli symbolisen konekielen kääntäjällä. Jokaisella eri prosessorilla on oman käskykantansa mukainen assembler. Yksi assemblerkielinen rivi kääntyy yhdeksi konekieliseksi käskyksi,

[6] Ja jos haluaa harrastuksen vuoksi hullutella, esim. ohjelmoida 4096 tavun kokoisia multimediateoksia eli ”4k introja”, on konekielen monipuolinen tuntemus vähintäänkin hyödyllistä kompaktin konekielen aikaansaamiseksi; ainakin kääntäjän koko-optimoinnin lopputulos on hyvä pystyä tarkistamaan.

ja käyttöjärjestelmän ohjelmakoodista pienehkö osa on kirjoitettava assemblerilla, joten tällä kurssilla ilmeisesti käsitellään sitä. Se on myös oiva apuväline prosessorin toiminnan ymmärtämiseksi (ja yleisemmin ohjelman suorituksen ymmärtämiseksi... ja myös korkeamman abstraktiotason kielijärjestelmien arvostamiseksi!). Assembler-koodin rivi voi näyttää päällisin puolin esimerkiksi tältä::

```
movq    %rsp , %rbp
```

Kyseinen rivi voisi hyvin olla x86-64 -arkkitehtuurin mukaista, joskin yhden rivin perusteella olisi vaikea vetää lopullista johtopäätöstä. Erot joissain yksittäisissä assembler-käskyissä ovat arkkitehtuurien välillä olemattomia. Prosessorivalmistajan julkaisema arkkitehtuuridokumentaatio on yleensä se, joka määrittelee symbolisessa konekielessä käytetyt sanat. Jokaisella konekielikäskyllä on **käskysymboli** (vai miten sen suomentaisi, ehkä "muistike" tjsp., englanniksi kun se on *mnemonic*). Yllä olevan esimerkin tapauksessa symboli on "movq". Käskyn symboli on tyypillisesti jonkinlainen helpohkosti muistettava lyhenne sen merkityksestä. Jos tämä olisi x86-64 -arkkitehtuurin käsky, "movq" (joka AMD64:n manuaalissa kirjoitetaan isoilla kirjaimilla "MOV" ilman "q"-lisuketta) olisi lyhenne sanoista "Move quadword". Sen merkitys olisi siirtää "nelisana" eli 64 bittiä paikasta toiseen. Tieto siitä, mistä mihin siirretään, annetaan **operandeina**, jotka tässä tapauksessa näyttäisivät x86-64:n määrittelemiltä rekistereiltä "rsp" ja "rbp" (AMD64:n dokumentaatiossa isoilla kirjaimilla "RSP" ja "RBP"). Käskyillä on useimmiten nolla, yksi tai kaksi operandia. Joka tapauksessa osa käskyn suorituksen syötteistä voi tulla muualtakin kuin operandeina ilmoitetusta paikasta – esim. FLAGS:n biteistä, tietyistä rekistereistä, tai jostain tietystä muistiosoitteesta. Jos operandina on rekisteri, jossa on muistiosoite, ja käskyn halutaan vaikuttavan muistipaikan sisältöön, puhutaan epäsuorasta osoittamisesta, (engl. *indirect addressing*). Tietyn prosessoriarkkitehtuurin dokumentaation käskykanta-osuudessa kerrotaan aina hyvin täsmällisesti, mitkä kunkin käskyn kaikki syötteet, tulosteet ja sivuvaikutukset prosessorin tai keskusmuistin seuraavaan tilaan ovat. Esimerkin tapauksessa nuo 64 bittiä siirrettäisiin prosessorin sisällä rekisteristä "rsp" rekisteriin "rbp". Sanotaan, että käskyn **lähde** (engl. *source*) on tässä tapauksessa rekisteri "rsp" ja **kohde** (engl. *destination*) on rekisteri "rbp". Koska siirto on rekisterien välillä, ulkoista väylää ei tarvitse käyttää. Siirtokäskyllä ei ole vaikutusta lippurekisteriin.

Prosenttimerkki "%" ylläolevassa on riippumaton x86-64:stä; se on osa tässä käytettyä yleisempää assembler-syntaksia, jota kurssillamme tänä kesänä käytettävät GNU-työkalut noudattavat.

Jotta ohjelmoijan maailma olisi tehty vaikeammaksi (tai ehkä kuitenkin muista historiallisista syistä) noudattavat jotkut assembler-työkalut ihan erilaista syntaksia kuin GNU-työkalut (GNU-työkalujen oletusarvoisesti noudattama syntaksi on nimeltään "AT&T -syntaksi" ja se toinen taas "Intel -syntaksi"). Ylläoleva rivi olisi siinä toisessa syntaksissa jotakuinkin näin::

```
movq    rbp , rsp
```

Erittäin merkittävä ero edelliseen on se, että *operandit ovat eri järjestyksessä!!* Eli lähde onkin oikealla ja kohde vasemmalla puolen pilkkua. Jonkun mielestä kai asiat ovat loogisia näin, että siirretään "johonkin jotakin" ja jonkun toisen mielestä taas niin, että siirretään "jotakin johonkin". Perimmäiset suunnitteluperusteet syntaksien luonteeseen, jos sellaisia ylipäätään on, ovat vaikeita löytää. Tänä päivänä käytetään molempia notaatioita, ja täytyy aina ensin vähän katsastella assembler-koodia ja dokumentaatiota ja päätellä jostakin, kumpi syntaksi nyt onkaan kyseessä, eli miten päin lähteitä ja kohteita ajatellaan. *Tämän luentomonisteen kaikissa esimerkeissä lähdeoperandi on vasemmalla ja kohde oikealla puolella pilkkua.* Käytämme siis AT&T -syntaksia, tarkemmin sen GNU-variaatiota [5], jota "gas" eli GNU Assembler käyttää.

Olipa syntaksi tuo tai tämä, assembler-kääntäjän homma on muodostaa prosessorin ymmärtämä bittijono symbolisen rivin perusteella. Paljastetaan tässä, että tuo ylläoleva rivi on ohjelmasta, johon se kääntyy seuraavasti::

```
400469:         48 89 e5                movq    %rsp,%rbp
```

Tässä tulosteessa ensimmäinen numero on käskyn muistipaikan osoite ohjelma-alueen alusta luettuna, virtuaaliosoite. Sitten seuraa heksaluvut, jotka kuvaavat kyseisen käskyn bittijonoa. Näköjään kyseisen käskyn bittijonossa on kolme tavua, jotka heksana ovat 48 89 e5. Siis bitteinä käsky on 0100 1000 1000 1001 1110 0101, jos monisteen kirjoittaja ei mokannut päässämuunnosta heksoista – tarkista itse; tämä on hyvä harjoitus lukujärjestelmistä. Lopussa on assembler-kielinen ilmaus eli käskyn lyhenne "mnemonic" ja operandit siinä muodossa kuin ne GNU assemblerissa kirjoitetaan x86-64 -käskykannalle.

Assembler-käännös taitaa olla ainoa ohjelmointikäännös, joka puolijärjellisellä tavalla on tehtävissä toisin päin: Konekielinen bittijono nimittäin voidaan kääntää takaisin ihmisen ymmärtämälle assemblerille. Sanotaan, että tehdään **disassembly**. Tällä tavoin voidaan tutkia ohjelman toimintaa, vaikkei lähdekoodia olisi saatavilla. Työlästähän se on, ja "viimeinen keino" debuggauksessa tai teollisuusvakoilussa, mutta mahdollista kuitenkin. Assembler-kielinen lähdekoodi sinänsä on kokeneen silmään ihan selkeätä, mutta ilman lähdekoodia tehdyssä disassemblyssä ei ole käytettävissä muuttujien tai muistiosoitteiden kuvaavia nimiä – kaikki on vain suhteellisia numeroindeksejä suhteessa rekisterien sisältämiin muistiosoitteisiin. Kokonaisuutta on silloin mahdoton

hahmottaa. Sillä tavoin tietokone käsittelee ohjelman suoritusta eri tavoin kuin ihminen – ihminen tarvitsee käsinkosketeltavia nimiä asioille, kone taas vain numeroita.

3.3 Esimerkkejä x86-64 -arkkitehtuurin käskykannasta

Edellä nähtiin esimerkki konekielikäskystä, "movq %rsp,%rbp". Mitä muita käskyjä voi esimerkiksi olla? Otetaan muutama poiminta AMD64:n manuaalin [4] käskykantaosiosta, tiivistetään ja suomennetaan tähän. Todellisuus on rikkaampi.

3.3.1 MOV-käskyt

Yksinkertainen bittien siirto paikasta toiseen tapahtuu käskyllä, jonka muistike (nimi, assembler-syntaksi) on useimmiten MOV. Ja GNU assemblerissa tähän lisätään vielä bittien määrää ilmaiseva kirjain. Esimerkkejä erilaisista tavoista vaikuttaa käskyn lähteeseen ja kohteeseen::

```
movq    %rsp, %rbp          # Rekisterin RSP bitit rekisteriin RBP

movl    %eax, %ebx          # 32 bitin siirto osarekisterien välillä
                            # ('l' == "long word", 32 bittiä)

movq    $123, %rax          # Käskyyn sisällytetyn vakioluvun siirto
                            # rekisteriin RAX; ylin bitti monistuu
                            # siirrossa joten kaikki 64 bittiä
                            # asettuvat vaikka luku 123 mahtuisi 8
                            # bittiin.

movq    %rax, -8(%rbp)      # Rekisterin RAX bitit
                            # muistipaikkoihin, joista ensimmäisen
                            # (virtuaali)osoite on RBP:n sisältämä
                            # osoite miinus 8. Viimeinen tavu
                            # sijoittuu paikkaan RBP-1. Missä
                            # keskinäisessä järjestyksessä
                            # 64-bittisen rekisterin 8 tavua
                            # tallentuvat noihin kahdeksaan
                            # muistipaikkaan?
                            #
                            # Tarkista itse prosessorimanuaalista
                            # kohdasta "byte order", mikäli haluat
                            # tarkan tiedon ...
                            #
                            # Myöhemmin tutustutaan pinokehysmalliin,
                            # jota noudattaen tuosta osoitteesta, eli
                            # RBP:n arvo miinus kahdeksan, voisi
                            # olettaa löytävänsä ensimmäisen
                            # nykyiselle aliohjelmalle varatun
                            # 64-bittisen lokaalin muuttujan...

movq    32(%rbp), %rax      # Rekisteriin RAX haetaan bitit
                            # muistipaikasta, jonka (virtuaali)osoite
                            # on RBP:n sisältämä osoite plus 32.
                            #
                            # Myöhemmin tutustutaan pinokehysmalliin,
                            # jota noudattaen tuosta osoitteesta voisi
                            # olettaa löytävänsä yhden pinon kautta
                            # välitetyistä aliohjelmaparametreista.
                            # (tosin kun parametreja on vähän, pinon
                            # kautta ei välttämättä välitetä mitään)
```

Esitellään tässä kohtaa vielä yksi tyypillinen käsky, LEA eli "load effective address"eli "lataa lopullinen osoite"::

```
lea     32(%rbp), %rax      # Osoite RBP + 32 lasketaan tässä
                            # valmiiksi, mutta sen sijaan, että
```

```
                    # siirrettäisiin osoitetun muistipaikan
                    # sisältö, laitetaankin tässä itse
                    # muistiosoite kohderekisteriin.
                    # Osoitteeseen voitaisiin sitten
                    # kohdistaa vielä laskutoimituksia ennen
                    # kuin sitä käytetään. Esimerkiksi
                    # voitaisiin ynnätä taulukon indeksin
                    # mukainen luku taulukon ensimmäisen
                    # alkion osoitteeseen ...
```

Näin ollen käskypari "lea 32(%rbp), %rdx" ja sen perään "movq (%rdx), %rax" tekisi saman kuin "movq 32(%rbp), %rax". Ja yksi käyttötarkoitus on siis esim. yhdistelmä::

```
lea    32(%rbp), %rdx    # Taulukon alkuosoite RDX:ään
addq   %rcx, %rdx        # Siirros RCX on laskettu valmiiksi esim.
                         # silmukkalaskurin päivityksen yhteydessä
movq   (%rdx), %rax      # Kohdistetaan haku taulukon sisällä olevaan
                         # muistipaikkaan.
```

3.3.2 Pinokäskyt

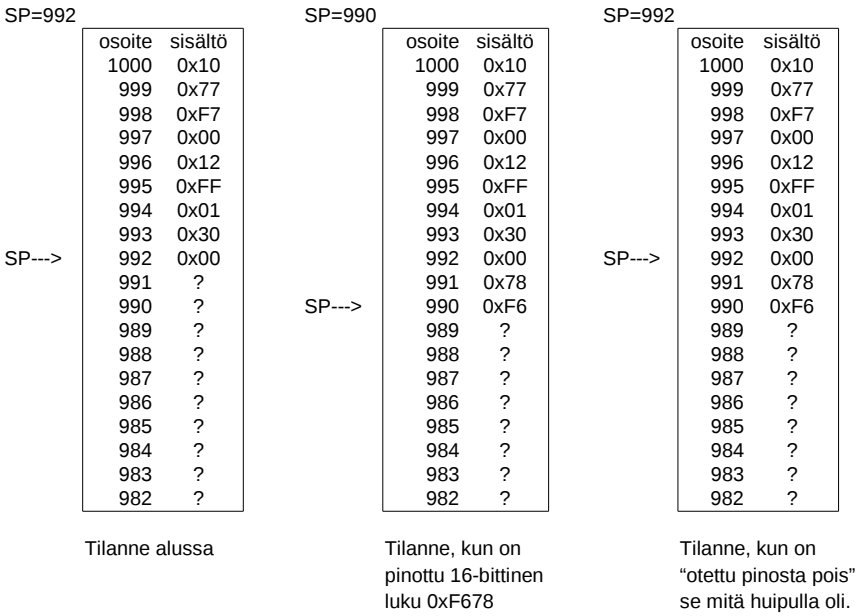
Yksi tapa siirtää bittejä paikasta toiseen on käyttää **suorituspinoa** (engl. *execution stack*). Silloin siirron lähde tai kohde on aina pinon huippu, jonka muistiosoite on rekisterissä RSP. Kun pinoon laitetaan jotakin tai sieltä otetaan jotakin pois, prosessori tekee automaattisesti siirron muistiin nimenomaan pinoalueelle tai vastaavasti sieltä johonkin rekisteriin. (Huomaa, että koko ajan tarkoituksella ohitetaan välimuistiin liittyvät tekniset yksityiskohdat!). Samalla se päivittää pinon huippua ylös tai alaspäin. Pari esimerkkiä::

```
pushq $12               # Ensinnä RSP:n arvo pienenee 8:lla,
                        # koska käskyssä on ``q'' mikä tarkoittaa
                        # 64-bittistä siirtoa. Muistiosoitteethan
                        # ovat aina yhden tavun eli 8 bitin
                        # kokoisten muistipaikkojen osoitteita.
                        #
                        # Siihen kohtaan muistia (osoite uusi RSP)
                        # menee sitten luku 12, eli 64-bittinen
                        # luku joka heksana on 0x000000000000000c.

pushl %edx              # RSP:n arvo pienenee 4:lla, koska
                        # käskyssä on ``l'' mikä tarkoittaa
                        # 32-bittistä siirtoa. Siihen kohtaan
                        # muistia menee sitten ne 32 bittiä, jotka
                        # ovat rekisterissä EDX eli RDX:n
                        # 32-bittinen puoli.
```

Kun pinoon laitetaan jotain, RSP tosiaan pienenee, koska pinon pohja on suurin pinolle tarkoitettu muistiosoite, ja pinon huippu kasvaa muistiosoitemielessä alaspäin. (Näin on siis useissa tyypillisissä prosessoreissa, mm. x86-64:ssä, oletettavasti joistakin historiallisista syistä; aivan kaikissa prosessoriarkkitehtuureissa suunta ei ole sama). Pinon päältä voi ottaa asioita vastaavasti::

```
popq %rbx               # Ensinnä prosessori siirtää RSP:n
                        # sisältämän muistiosoitteen kertomasta
                        # muistipaikasta 64 peräkkäistä bittiä
                        # rekisteriin RBX. Sen jälkeen se lisää
                        # RSP:n arvoon 8, eli tuloksena pinon
                        # huippu palautuu 64-bittisellä
                        # pykälällä kohti pohjaa.
```

Pinokäskyjen toimintaa havainnollistetaan kuvassa 4. Huomaa, että pino on käsitteellisesti samanlainen kuin normaali "pino"-tyyppinen (eli viimeksi sisään – ekana ulos) tietorakenne, jota normaalisti käytetään kahdella operaatiolla eli "painamalla" (push) asioita edellisten päälle ja "poksauttamalla" (pop) päällimmäinen asia pois pinosta, mahdollisesti käyttöön jossakin muualla. Toisaalta pino on vain peräkkäisten muistiosoitteiden muodostama alue, ja sitä käytetään varsinaisen huipun lisäksi myös huipun lähistöltä (ns. aktivaatiotietueen/-pinokehyksen sisältä, mikä selitetään myöhemmin). Kuvan esimerkissä on tyypillinen tapa esittää jokin alue



Kuva 4: *Prosessorin pino: muistialue, jota voidaan käyttää push- ja pop-käskyillä. SP osoittaa aina pinon ”huippuun”, joka ”kasvaa” muistiosoitteen mielessä alaspäin.*

muistiavaruudesta: muistiosoitteet kasvavat kuvan alalaidasta ylälaitaan päin. Näitä tulemme näkemään. Tässä kuvassa osoitteet ovat tavun kokoisten muistipaikkojen osoitteita, ja ne on ilmoitettu desimaalilukuina. Jatkossa siirrymme (tietokonemaailmassa) järkevämpään tapaan eli heksadesitykseen, kuten nyt onkin jo tehty muistin sisällön osalta – tavu kun on näppärää esittää kahdella heksanumerolla.

Muistin sisältönä pinon huippuun saakka on ”jotakin”, jolla yleensä on jokin merkitys. Kuvassa 4 tätä kuvaa satunnaisesti keksityt tavut. Muissakin osoitteissa on tietysti jotakin (aiemman historian mukaista) dataa, mutta niistä ei käytännössä välitetä, joten ne on kuvassa merkitty kysymysmerkeillä. Kuvan ensimmäinen tilanne vastaa tällaista ”alkutilannetta”. Kuvan esimerkissä pinoon laitetaan “push“ -käskyllä 16-bittinen luku. Kuten havaitaan, pino-osoitin SP on ensin saanut pienemmän arvon, jotta ”pinon huippu nousee”, ja sitten huipulle on tallennettu luvun tavut. Keskimmäisessä vaiheessa siis pinoon on laitettu jotakin, sen huippu on ”noussut” (joskin muistiosoitemielessä pienentynyt) ja se on valmiina vastaanottamaan taas jotakin uutta. Kolmannessa kuvassa puolestaan on esitetty “pop“-käskyn jälkeinen tilanne: 16-bittinen luku on kopioitu muistista jonnekin, oletettavasti johonkin rekisteriin, jotakin käyttötarkoitusta varten, ja pinon huippua on vastaavan verran ”lasketu” (joskin muistiosoitemielessä kasvatettu). Tästä huomataan, kuinka pinon muistialueelle kyllä aina jää sinne laitettu data, mutta datan merkitys on hävinnyt, sillä heti seuraava ”push” käyttäisi uudelleen samat muistipaikat. Tämä on olennaista ymmärtää, jotta myöhemmin on helpompi ymmärtää, miksi aliohjelman paikalliset muuttujat ovat ”unohtuneet” aliohjelmasta palaamisen jälkeen.

3.3.3 Aritmetiikkaa

Edellä olevat siirto- ja pinokäskyt vain siirtävät bittejä paikasta toiseen. Ohjelmointi edellyttää usein bittien muokkaamista matkalla. Pari esimerkkiä::

```
addq %rdx, -32(%rbp)      # Hakee muistipaikasta (RBP:n arvo - 32)
                          # löytyvät bitit, laskee ne yhteen
                          # rekisterissä RDX olevien bittien kanssa
                          # ja sijoittaa tuloksen takaisin
                          # muistipaikkaan (RBP:n arvo - 32).
                          # Muistia tarvitaan kolmessa kohtaa
                          # suoritussyklin kierrosta: käskyn nouto,
                          # operandin nouto, tuloksen tallennus.
```

```
addq $17, %rax          # Usein ynnätään "akkumulaattoriin" eli
                        # RAX–rekisteriin. Tässä luku 17 on mukana
                        # käskyn konekielikoodissa; se lisätään
                        # RAX:n arvoon ja tulos jää RAX:ään.
                        # Ylimääräisiä muistipaikkojen käyttöjä ei
                        # käskyn noudon lisäksi tarvita, joten tämä
                        # saattaa vaatia vähemmän kellojaksoja kuin
                        # edellä esitelty yhteenlasku suoraan
                        # muistiin.

subl 20(%rbp), %eax     # EAX–rekisterin arvosta vähennetään luku,
                        # joka haetaan ensin muistipaikasta RBP+20;
                        # tulos jää EAX–rekisteriin.
```

Prosessorit tarjoavat kokonaislukulaskentaan usein myös MUL-käskyn kertolaskulle ja DIV-käskyn jakolaskulle (tuloksena erikseen osamäärä ja jakojäännös tietyissä rekistereissä). Näistä on usein, mm. x86-64:ssä, erikseen etumerkillinen ja etumerkitön versio. Aritmeettiset käskyt vaikuttavat lippurekisterin RFLAGS tiettyihin bitteihin, esimerkkejä:

- Yhteenlaskun muistibitti jää talteen, *Carry flag*
- Jos tulos on nolla, asettuu *Zero flag*
- Jos tulos on negatiivinen, asettuu *Negative flag*

Liukulukujen laskentaan pitää käyttää erillisiä liukulukurekisterejä ja liukulukukäskyjä, joita ei tässä kuitenkaan käsitellä.

3.3.4 Bittilogiikkaa ja bittien pyörittelyä

Moniin tarkoituksiin tarvitaan bittien muokkaamista. Pari esimerkkiä::

```
notq  %rax          # Kääntää RAX:n kaikki bitit nollasta ykkösiksi
                    # tai toisin päin, siis bitittäinen looginen
                    # EI–operaatio.

andq  $15, %rax     # Bitittäinen looginen JA–operaatio. Tässä
                    # tapauksessa 15 on bitteinä 000...001111 eli
                    # neljä alinta bittiä ykkösiä ja loput 60 kpl
                    # nollia. Lopputuloksena RAX:n 60 ylintä bittiä
                    # ovat varmasti nollia ja puolestaan 4 alinta
                    # bittiä jäävät aiempaan arvoonsa, eli looginen
                    # JA toteuttaa bittien "maskaamisen". (Tämä btw
                    # on hyödyllinen kikka myös korkean tason
                    # kielillä ohjelmoidessa)

testq $15, %rax     # TEST tekee saman kuin AND, mutta ei tallenna
                    # tulosta mihinkään. Miksi näin? Liput eli
                    # RFLAGS päivittyy, eli esim. tässä tapauksessa
                    # jos tulos on nolla, Zero flag kertoisi käskyn
                    # jälkeen että mikään RAX:n neljästä alimmasta
                    # bitistä ei ole asetettu.

orq   %rdx, %rcx    # Bitittäinen looginen TAI–operaatio
                    # (Tätä voi käyttää bittien asettamiseen: ne
                    # jotka olivat ykkösiä RDX:ssä tulevat ykkösiksi
                    # RCX:ään, ja ne jotka olivat nollia RDX:ssä
                    # jäävät ennalleen RCX:ssä).

xorq  %rax, %rax    # Bitittäinen looginen JOKO–TAI –operaatio.
                    # Esimerkissä molemmat operandit ovat RAX, jolloin
                    # JOKO–TAI aiheuttaa RAX:n kaikkien bittien
```

```
                # nollautumisen, mikä vastaa luvun nolla
                # sijoittamista rekisteriin, mutta voi olla
                # nopeampi suorittaa (oli aikoinaan 286:ssa ym.
                # mutta en tiedä x86-64 -vehkeistä) ja
                # konekielinen koodi voi olla lyhyempi.
```

Muitakin bittioperaatioita on. Joitain esimerkkejä::

```
sarb   $3, %ah     # Siirtää 8-bittisen (``b'', byte) rekisteriosan
                   # bittejä kolmella pykälällä oikealle, eli jos
                   # siellä oli bitit 0110 0101 niin sinne jää
                   # käskyn jälkeen 0000 1100.

rolw   %cl, %ax    # Pyörittää 16-bittisen (``w'', word)
                   # rekisteriosan bittejä vasemmalle niin monta
                   # pykälää kuin 8-bittisen rekisteriosan CL viisi
                   # alinta bittiä kertovat. Siis esim. jos CL on
                   # 0100 0100 (eli viisi alinta bittiä ovat
                   # lukuarvo 4) ja AX on 1000 0011 0000 1110 niin
                   # pyöritetty tulos olisi 0011 0000 1110 1000
```

Pyörityksiä ja siirtoja on vasemmalle ja oikealle (SAR, SAL, ROR, ROL); näissä pyörityksen voi tehdä ilman Carry-lippua tai sitten voi pyörittää siten, että laidalta pois putoava bitti siirtyy Carry-lipuksi ja toiselta laidalta sisään pyöritettävä tulee vastaavasti Carrystä. Sitten on kokonaisluvun etumerkin vaihto NEG, ja niin edelleen. Tämä ei ole täydellinen konekieliopas, joten jätetään esimerkit tälle tasolle ja todetaan, että on niitä paljon muitakin, mutta että suurin piirtein tällaisia asioita niillä tehdään: suhteellisen yksinkertaisia bittien siirtoja paikasta toiseen.

3.3.5 Suoritusjärjestyksen eli kontrollin ohjaus: mistä on kyse

Tähän asti mainituista käskyistä muodostuva ohjelma suoritetaan **peräkkäisjärjestyksessä**, eli prosessori siirtyy käskystä aina välittömästi seuraavaan käskyyn. Tarkemmin: prosessori päivittää IP:n siten että ennen seuraavaa noutoa siinä on edellistä seuraavan käskyn osoite (eli juuri suoritetun käskyn osoite + juuri suoritetun käskyn bittijonon tarvitsemien muistipaikkojen määrä). Peräkkäisjärjestys ja käskyjen mukaan muuttuva tila onkin ohjelmoinnissa syytä ymmärtää alkuvaiheessa, kuten suorassa seisominen ja käveleminen ovat perusteita juoksemiselle, hyppäämiselle ja muulle vaativammalle liikkumiselle.

Ohjelmoinnista tietänet, että algoritmien toteuttaminen vaatii myös muita kuin peräkkäisiä suorituksia, erityisesti tarvitaan:

- **ehtorakenteet**, eli jotain tehdään vain silloin kun joku looginen lauseke on tosi.
- **toistorakenteet**, eli jotain toistetaan useaan kertaan, kunnes jokin lopetuskriteeriä kuvaava looginen lauseke muuttuu epätodeksi.
- **aliohjelmat** (tai **metodit**), eli suoritus täytyy voida siirtää toiseen ohjelman osioon väliaikaisesti, ja tuolle osiolle täytyy kertoa, mille tiedoille (esim. lukuarvoille tai olioille) sen pitää toimenpiteensä tehdä. Aliohjelmasta pitää myös voida sopivaan aikaan palata siihen kohtaan, missä aliohjelmaa alunperin kutsuttiin. Ideana on myös että aliohjelma voi palauttaa laskutoimitustensa tuloksen kutsuvaan ohjelmaan. Aliohjelmat voivat kutsua toisiaan (ja itseään, ”rekursio”). Kutsuista voidaan ajatella muodostuvan pino ”aktivaatioita”, jossa päällimmäinen, viimeksi kutsuttu aliohjelma on aktiivinen (rekursiossa samalla aliohjelmalla voi olla pinossa useita aktivaatioita) ja muut lojuvat pinossa kunnes niihin taas palataan.
- **poikkeukset**, eli suoritus täytyy pystyä siirtämään muualle kuin kutsuvaan aliohjelmaan, tai sitä kutsuneeseen tai niin edelleen...itse asiassa täytyy kyetä palaamaan niin kauas, että löytyy poikkeuksen käsittelijä.

Poikkeukset helpottavat ohjelmointia (tai vaikeuttavat, näkökulmasta riippuen...), mutta eivät sinänsä ole välttämättömiä ohjelmien tekemiselle. Kolme ensimmäistä ovat sangen välttämättömiä, ja nyt tutustutaan siihen, millaisilla käskyillä konekielessä saadaan aikaan ehto- ja toistorakenteet sekä aliohjelmien kutsuminen. Suorituksen on voitava siirtyä paikasta toiseen. Usein käytetty nimi tälle on **kontrollin siirtyminen** (engl. *control flow*). Jokin ohjelman kohta hallitsee eli kontrolloi laitteistoa kullakin hetkellä ja kontrollin siirtäminen osiolta toiselle on avain käyttökelpoiseen ohjelmointiin. Tässä monisteessa ”kontrollin siirtymisestä” puhutaan näköjään

vahingossa kaksi kertaa (ainoastaan, koska alunperin välttelin termiä), mutta englanninkielisessä kirjallisuudessa kontrolli on erittäin usein käytetty muoto suoritusjärjestyksen ohjaukselle.

3.3.6 Konekieltä suoritusjärjestyksen ohjaukseen: hypyt

Konekielikoodi sijaitsee tavuina keskusmuistin muistipaikoissa, joiden muistiosoitteet ovat peräkkäisiä. Ehdollinen suoritus ja silmukat perustuvat ehdollisiin ja ehdottomiin hyppykäskyihin, esimerkkejä::

```
jmp   MUISTIOSOITE      # Ehdoton hyppy "jump". Tämän käskyn
                        # suorituksen kohdalla prosessori lataa
                        # uudeksi käskyosoitteeksi (RIP-rekisteriin)
                        # osoitteen, joka käskyssä kerrotaan.
                        # Käännetyssä konekielessä osoite on
                        # tyypillisesti suhteellinen osoite
                        # hyppykäskyn oman muistipaikan osoitteeseen
                        # nähden, eli se on mallia "hyppää 48 tavua
                        # eteenpäin" tai "hyppää 112 tavua
                        # taaksepäin". Ensimmäisessä em. esimerkissä
                        # RIP päivittyisi RIP := RIP + 48 ja toisessa
                        # esimerkissä RIP := RIP - 112.

jz    MUISTIOSOITE      # Ehdollinen hyppy "jump if Zero". Hyppy on
                        # kuten jmp, mutta se tehdään vain silloin
                        # kun Zero flag on asetettu, eli kun
                        # edellisen aritmeettisen tai loogisen
                        # operaation tulos oli nolla. Jos RFLAGSin
                        # Zero-bitti ei ole asetettu, hyppyä ei
                        # tehdä vaan käskyn suorituksessa ainoastaan
                        # päivitetään RIP osoittamaan seuraavaa
                        # käskyä, ihan kuin peräkkäisesti
                        # suoritettavissakin käskyissä.

jnz   MUISTIOSOITE      # Ehdollinen hyppy "jump if not Zero".
                        # Arvatenkin hyppy tehdään silloin kun Zero
                        # flag -bitti ei ole asetettu eli edeltävä
                        # käsky ei antanut tulokseksi nollaa.

jg    MUISTIOSOITE      # "Jump if Greater" eli aiemmassa
                        # vertailussa (tai vähennyslaskussa)
                        # kohdeoperandi oli suurempi kuin
                        # lähde [Tai toisin päin, tämä on hankala
                        # muistaa tarkistamatta manuaalista].
                        # Ehto selviää tietysti RFLAGSissä
                        # olevista biteistä, kuten kaikissa
                        # ehdollisissa hypyissä.

jng   MUISTIOSOITE      # "Jump if not greater"

jle   MUISTIOSOITE      # "Jump if less or equal"

jnle  MUISTIOSOITE      # "Jump if not less or equal"

... ja niin edelleen ... näitä on melko monta variaatiota, jotka
kaikki toimivat samoin ...
```

Korkean tason kielellä kuten C:llä tai Javalla ohjelmoija ei tee itse lainkaan hyppykäskyjä, vaan hän kirjoittaa silmukoita silmukkasyntaksilla (esim. “for..“ tai “while..“) ja ehtoja ehtosyntaksilla (kuten “if .. else if..“). Kääntäjä tuottaa kaikki tarvittavat hypyt ja bittitarkistukset. Jos ohjelmoidaan suoraan assemblerilla, pitää hypyt ohjelmoida itse, mutta suhteellisia muistiosoitteita ei tarvitse tietenkään itse laskea, vaan assembler-kääntäjä osaa muuntaa symboliset nimet sopivasti. Esimerkiksi seuraava ohjelma laskisi luvusta 1000 lukuun 0 rekisterissä RAX::

```
ohjelman_alku:                  # symbolinen nimi muistipaikalle
        movq    $1000, %rax
```

```
silmukan_alku:                  # symbolinen nimi muistipaikalle
        subq    $1, %rax
        jnz     silmukan_alku   # Kääntäjä osaa laskea montako
                                # tavua taaksepäin on hypättävä
                                # että uudesta osoitteesta löytyy
                                # edelläkirjoitettu subq-käsky.
                                # Tuon miinusmerkkisen luvun se
                                # koodaa mukaan konekielikäskyyn.
```

Huomaa, että sama asia voidaan toteuttaa monella erilaisella konekielikäskyjen sarjalla – esim. edellinen lasku tuhannesta nollaan voitaisiin toteuttaa yhtä hyvin seuraavasti::

```
ohjelman_alku:
        movq    $1000, %rax

silmukan_alku:
        subq    $1, %rax
        jz      silmukka_ohi
        jmp     silmukan_alku

silmukka_ohi:
        ... tästä jatkuisi koodi eteenpäin ...
```

Silmukan konekielinen toteutus vaatii tyypillisesti joitakin käskyjä sekä silmukan alkuun että sen loppuun, vaikka korkean tason kielellä alku- ja loppuehdot kirjoitettaisiin samalle riville, esim::

```
for(int i = 1000; i != 0 ; i--)
{
  /* ... silmukan toistettava osuus ... */
}
```

3.4 Ohjelma ja tietokoneen muisti

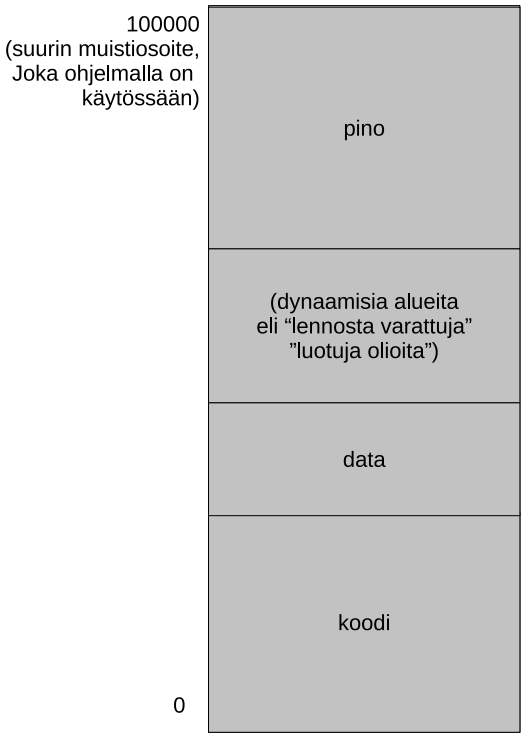
Luodaan katsaus siihen, miten tietokoneen muistin ja prosessorin yhteispeli oikein tapahtuu.

3.4.1 Koodi, tieto ja suorituspino; osoittimen käsite

Ohjelmaan kuuluu selvästi konekielikäskyjä prosessorin suoritettavaksi. Sanotaan, että tämä on ohjelman **koodi** (engl. *code*). Lisäksi ohjelmissa on usein jotakin ennakkoon tunnettua tai globaalia **dataa** (engl. *data*) kuten vakiomerkkijonoja ja varmasti tarvitaan vielä **paikallisia muuttujia** (engl. *local variables*) eli tallennustilaa useisiin väliaikaisiin tarkoituksiin. Tämä lienee selvää.

Mainitut koodi ja data voidaan ladata eri paikkoihin tietokoneen muistissa, ja paikallisille muuttujille varataan vielä ihan oma alue, jonka nimi on **pino** (sama kuin jo aiemmin mainittu suorituspino). Ohjelman tarvitseman muistin jako koodiin, dataan ja pinoon on perusteltua ja selkeätä ohjelmoijan kannalta; ovathan nuo selvästi eri tarkoituksiin käytettäviä ja erilaista dataa sisältäviä kokonaisuuksia. Lisäksi ohjelmat käyttävät useimmiten **dynaamista muistinvarausta** (engl. *dynamic memory allocation*) eli ohjelmat voivat pyytää (arvatenkin käyttöjärjestelmältä tai virtuaalikoneelta) käyttöönsä alueita, joiden kokoa ei tarvitse tietää ennen kuin pyyntö tehdään. Kuvassa 5 esitetään käsitteellinen laatikkodiagrammi ohjelman tarvitseman muistin jakautumisesta. Asia muodostuu jonkin verran täsmällisemmäksi, kun jatkossa puhutaan tarkemmin muistinhallinnasta.

Huomaa, että prosessorin kannalta dataa ei ole missään ”nimetyissä muuttujissa”, kuten lähdekoodin kannalta, vaan kaikki käsiteltävissä oleva data on rekistereissä, tai se pitää noutaa ja viedä muistiosoitteen perusteella. Muistiosoite on vain numero; useimmiten osoite otetaan jonkun rekisterin arvosta (eli rekisterin kautta tapahtuva epäsuora osoitus engl. *”register indirect addressing”*). Esim. pino-osoitin ja käskyosoiterekisteri ovat aina muistiosoitteita. Osoite voidaan myös laskea suhteellisena rekisterissä olevaan osoitteeseen nähden (tapahtuu rekisterin ja käskyyn koodatun vakion yhteenlasku ennen kuin osoite on lopullinen, ns. epäsuora osoitus ”kanta-rekisterin” ja siirrososoitteen avulla, engl. *”base plus offset”*). Lisäksi voi olla mahdollista laskea yhteen kahden eri rekisterin arvot (jolloin toinen rekisteri voi olla ”kanta” joka osoittaa esim. tietorakenteen alkuun ja toinen rekisteri ”siirros” jolle on voitu laskea tarpeen mukaan arvo edeltävässä koodissa; näin voidaan osoittaa tietorakenteen, kuten taulukon, eri osia).



Kuva 5: *Tyypilliset ohjelman suorituksessa tarvittavat muistialueet: koodi, data, pino, dynaamiset alueet. (periaatekuva, joka täydentyy myöhemmin)*

Operaatioiden tuloksetkin pitää tietysti erikseen viedä muistiin osoitteen perusteella. Kääntäjäohjelman tehtävänä on muodostaa numeerinen muoto osoitteille, joissa lähdekoodin kuvaamaa dataa säilytetään.

Assemblerilla muistiosoitteiden käyttö voisi näyttää esim. seuraavalta::

```
movq   $13, %(rbp)       # lähde "immediate",
                         # kohde "register indirect"

movq   $13, -16%(rbp)    # lähde "immediate",
                         # kohde "base plus offset"
```

C-ohjelmassa muistiosoitteita voi käyttää tietyllä syntaksilla, esim.::

```
int luku = 2;          /* lokaali kokonaislukumuuttuja nimeltä luku    */

int *osoitin;          /* lokaali muistiosoitin kokonaislukuun         */

osoitin = &luku;       /* otetaan luvun muistiosoite ja sijoitetaan se */

tulosta_osoitettu_luku(osoitin);
                       /* annetaan parametriksi muistiosoitin;
                          aliohjelma on tehty siten että se haluaa
                          parametrina osoittimen */

tulosta_luku(*osoitin);
                       /* annetaan parametriksi itse luku
                          eikä osoitetta; tähti on käänteinen
                          et-merkille                                  */

tulosta_osoitin(osoitin);
                       /* tässäkin annettaisiin parametriksi luku, mutta
                          kyseinen luku olisi muistiosoite.            */

lisaa_yksi_osoitettuun_lukuun(osoitin);
                       /* Tällä voitaisiin vaikuttaa paikallisen
                          muuttujan "luku" arvoon, johon osoitin
                          osoittaa.                                    */
```

```
lisaa_yksi_lukuun(luku);
                    /* Tällä ei tekisi mitään, jos tarkoitettu käyttö
                       olisi seuraavanlainen eikä parametri siis
                       olisi osoitin vaan primitiivimuuttuja:       */

luku = lisaa_yksi_lukuun(luku);
                    /* Tällä siis sijoitettaisiin paluuarvo.        */
```

Java-ohjelmassa jokainen viitemuuttuja on tavallaan ”muistiosoite” olioinstanssiin Javan **kekomuistissa** (engl. *heap*). Tai vähintäänkin sitä voidaan abstraktisti ajatella sellaisena. Esimerkki::

```
NirsoKapstyykki muuttujaA, muuttujaB, muuttujaC;
muuttujaA = new(NirsoKapstyykki(57));    /* instantoi                */
muuttujaB = new(NirsoKapstyykki(57));    /* instantoi samanlainen    */
muuttujaC = muuttujaA;                   /* sijoita                  */

tulosta_totuusarvo(muuttujaA == muuttujaB);       /* false */
tulosta_totuusarvo(muuttujaA == muuttujaC);       /* true  */
tulosta_totuusarvo(muuttujaA.equals(muuttujaB)); /* true, mikäli
                              NirsoKapstyykki toimii siten kuin
                              oletan sen toimivan Javassa..       */
```

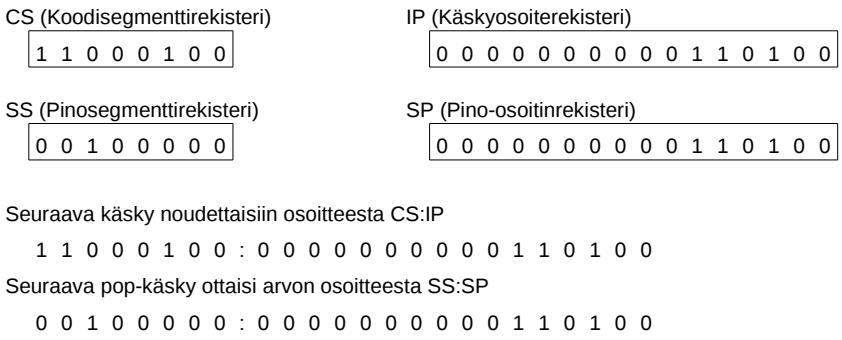
Ylläoleva Java-esimerkki pitäisi olla erittäin hyvin selkärangassasi, jos voit sanoa osaavasi ohjelmoida! Ja jos ei se vielä ole, voit ymmärtää asian yhtä aikaa kun ymmärrät muistiosoitteetkin (ja tulla siten askeleen lähemmäksi ohjelmointitaitoa): Esimerkissä “muuttujaA“, “muuttujaB“ ja “muuttujaC“ ovat **viittemuuttujia**, virtuaalikoneen sisäisessä toteutuksessa ehkäpä kokonaislukuja, jotka ovat indeksejä johonkin oliotaulukkoon tai muuhun vastaavaan. Viite eroaa muistiosoittimesta siinä, että se on vähän abstraktimpi käsite, eli se voisi olla jotain muutakin kuin kokonaisluku eikä ohjelmoijan tarvitse eikä pidä välittää niin kovin paljon sen varsinaisesta toteutuksesta ... Kuitenkin, kun yllä ensinnäkin instantoidaan kaksi kertaa samalla tavoin “NirsoKapstyykki“ ja sijoitetaan viitteet muuttujiin “muuttujaA“ ja “muuttujaB“, niin lopputuloksena on kaksi erillistä, vaikkakin samalla tavoin luotua, oliota. Kumpaiseenkin yksilöön on erillinen viite (sisäisenä toteutuksena esim. eri kokonaisluku). Sijoitus “muuttujaC = muuttujaA“ on nyt se, minkä merkitys pitää ymmärtää syvällisesti: Siinä sijoitetaan viite muuttujasta toiseen. Sen jälkeen viitemuuttujat “muuttujaA“ ja “muuttujaC“ ovat edelleen selvästi eri yksilöitä; nehän ovat Java-virtuaalikoneen suorituspinossa eri kohdissa ja niille on oma tila sieltä varattuna. Mutta se *olioinstanssi, johon ne viittaavat on yksi ja sama.* Eli sisäisen toteutuksen kannalta näyttäisi esimerkiksi siltä, että pinossa on kaksi samaa kokonaislukua eli kaksi samaa ”osoitinta” kekomuistiin. Sen sijaan “muuttujaB“ on eri viite. Rautalankaesimerkkinä pinossa voisi olla seuraava sisältö::

```
muuttujaA : 57686

muuttujaB : 3422

muuttujaC : 57686
```

Niinpä esim. muuttujien vertailut operaattorilla ja metodilla antavat tulokset siten kuin yllä on kommenteissa. Yritän siis kertoa vielä kerran, että:

- sekä JVM että konkreettiset tietokoneprosessorit ovat ”tyhmiä” vehkeitä, jotka tekevät peräkkäin yksinkertaisia suoritteita
- niissä on pinomuisti, koodialueita, dynaamisesti varattavia alueita
- näiden alueiden käyttö sekä rakenteisessa että olio-ohjelmoinnissa edellyttää ”viite” nimisen asian toteutumista jollain tavoin, olipa toteutus sitten muistiosoite, olioviite, kokonaisluku tai jokin mitä sanottaisiin ”kahvaksi”. Niiden toiminta ja ilmeneminen ovat monessa mielessä sama asia.

Ohjelmoinnin ymmärtäminen edellyttää abstraktin ”viite”-käsitteen ymmärtämistä, missä voi ehkä auttaa että näkee kaksi erilaista ilmenemismuotoa (tai edes yhden) konepellin alla eli laitteistotasolla (Javan tapauksessa laitteisto on virtuaalinen, eli JVM; C-kielen tapauksessa laitteisto on todellinen, esimerkiksi Intel Xeon; konekielen toiminnasta viimeistään selviää muistiosoittimen luonne).



Kuva 6: *Esimerkki segmenttirekisterien käytöstä: koodi ja pino voivat olla eri kohdissa muistia, vaikka IP ja SP olisivat samat. Segmentit ovat eri, ja alueet voivat kartoittua eri paikkoihin fyysistä muistia.*

3.4.2 Alustavasti virtuaalimuistista ja osoitteenmuodostuksesta

Olisi mukavaa, jos voitaisiin saada tuplavarmistuksia ja turvallisuutta siitä, että data- tai pinoalueelle ei voitaisi koskaan vahingossakaan hypätä suorittamaan niiden bittejä ikään kuin ne olisivat ohjelmakoodia. Pahimmassa tapauksessa pahantahtoinen käyttäjä saisi sijoitettua sinne jotakin haitallista koodia... haluttaisiin tosiaan myös, että koodialueelle ei vahingossakaan voisi kirjoittaa mitään uutta, vaan siellä sijaitsisi ainoastaan aikoinaan käännetty konekielinen ohjelma muuttumattomassa tilassa. (Ennakoidaan myös sitä tarvetta, että samassa tietokoneessa toimii yhtäaikaa useita ohjelmia, jotka eivät saisi sotkea toistensa toimintaa). Nykyisissä prosessoreissa tällaisia tuplavarmistuksia on saatavilla.

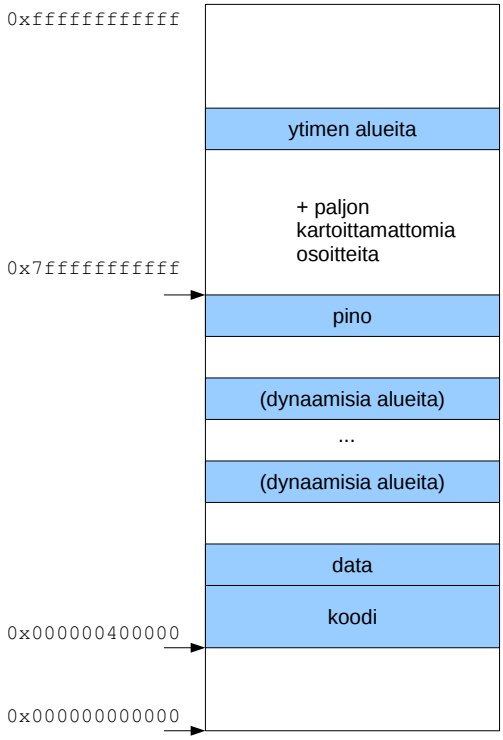
Moderneissa tietokoneissa sovellusohjelman konekielikäskyjen käsittelemät muistiosoitteet ovat ns. **virtuaaliosoitteita**: suorituksessa oleva ohjelma näkee oman koodinsa, datansa ja pinonsa omassa muistiavaruudessaan kuten kuvassa 5 summittaisesti esitettiin. Oikeat muistiosoitteet tietokoneen fyysisessä muistissa (siellä väylän takana) ovat jotakin muuta, ja prosessori osaa muuntaa virtuaaliset muistiosoitteet fyysisen muistin osoitteiksi. (Tämä tarkentuu myöhemmin).

Joissain arkkitehtuureissa voidaan kukin alue pitää omana segmenttinään, puhutaan **segmentoidusta muistista**. Tällöin esimerkiksi "IP":lle mahdolliset osoitteet alkavat nollasta ja päättyvät osoitteeseen, joka vastaa jotakuinkin ohjelmakoodin pituutta tavuina; tämähän on selkeätä, kun koodi alkaa nollasta ja jatkuu lineaarisesti aina käsky kerrallaan. Puolestaan "SP":lle mahdolliset osoitteet alkavat nollasta ja päättyvät osoitteeseen, joka vastaa pinolle varattua muistitilaa. Ohjelman alussa pino on tyhjä, ja "SP":n arvo on suurin mahdollinen; sieltä se alkaa kasvaa alaspäin kohti nollaa (alaspäin siis useissa, muttei kaikissa, arkkitehtuureissa). Myös data-alueen osoitteet alkavat nollasta. **Segmenttirekisterit** pitävät silloin huolen siitä, että pinon muistipaikka osoitteeltaan vaikkapa 52 on eri paikka kuin koodin muistipaikka osoitteeltaan 52. Kokonaisen virtuaalimuistiosoitteen pituus on silloin segmenttirekisterin bittien määrä yhdistettynä osoitinrekisterin pituuteen. Virtuaaliosoitteet olisivat siten esim. sellaisia kuin Kuvassa 6. Tämä on vaan hatusta vedetty esimerkki, jonka tarkoitus on näyttää, että IP ja SP voisivat segmentoidussa järjestelmässä olla samoja, mutta niiden tarkoittama virtuaaliosoite olisi eri, koska näihin liittyvät segmentit olisivat eri, koska segmentin numero kuuluu virtuaaliosoitteeseen.

Jätetään kuitenkin segmentoitu malli tuolle maininnan ja esimerkin tasolle. Esimerkkiarkkitehtuurimme x86-64 mahdollistaa segmentoinnin taaksepäin-yhteensopivuustilassa, koska siinä on haluttu pystyä suorittamaan x86-koodia, joka käytti segmenttejä. Kuitenkin 64-bittisessä toimintatilassa x86-64:n kullakin ohjelmalla on oma täysin lineaarinen muistiavaruutensa, joka käyttää 64-bittisen rekisterin 48 alinta bittiä muistiosoitteena.

Virtuaalinen osoiteavaruus sisältää siis osoitteet 0 – 281474976710655, heksalukuina kenties selkeämmin: 0x0 – 0xffffffffffff. Fyysistä muistia ei voi olla näin paljoa (281 teratavua), joten virtuaalimuistiavaruudesta kartoittuu fyysiseen muistiin vain osa. Muut muistiosoitukset johtavat virhetilanteeseen ja ohjelman kaatumiseen. x86-64:ssä ei ole mitään segmenttejä ja segmenttirekisterejä (tai itse asiassa on, mutta niiden on sovittu olevan sisällöltään nollia 64-bittisessä toimintatilassa). Koodi, pino ja tieto sijoittuvat kukin omaan alueeseensa 48-bittisessä virtuaaliosoiteavaruudessa.

Käyttöjärjestelmän ja kääntäjien valmistajat voivat tietenkin varioida muistiosoitteiden käyttöä, joten niiden varsinainen sijainti on sopimuskysymys. Esimerkiksi lähdettä [6] mukailevat sijainnit ohjelmakoodille, datalle ja pinolle on esitetty Kuvassa 7. Kuvassa on piirretty valkoisella kartoittamattomaksi jätetty muistin osa (jota suurin osa valtavasta muistiavaruudesta useimpien ohjelmien kohdalla tietenkin on), ja värein on esitetty kartoitetut alueet: koodi, data, pino, ja mahdolliset dynaamisesti varatut alueet. Ohjelmakoodin alueen pienin



Kuva 7: *Virtuaalinen muistiavaruus x86-64:ssä.*

osoite on 2^{22} = 0x400000. Sen alapuolelle on jätetty kartoittamatonta, mikä auttaa kaatamaan ohjelmat, joissa viitataan (bugin vuoksi) nollaa lähellä oleviin muistiosoitteisiin, sen sijaan että ohjelmat saisivat käytettyä muistia paikasta, josta ei ollut tarkoitus. Itse asiassa muistiosoitteessa 0 ei olisikaan järkeä olla mitään, koska tämä tulkitaan NULL-osoittimeksi eli ”ei osoita tällä hetkellä mihinkään”. Alaspäin kasvavan pinon ”pohjan” osoite $2^{47} - 1$ = 0x7fffffffffff on suurin muistiosoite, jonka 64-bittisessä esitysmuodossa eniten merkitsevät bitit ovat nollia; yhtä pykälää isomman osoitteen 0x800000000000 ylin eli 48. bitti nimittäin pitäisi AMD64:n spesifikaation [4] mukaan monistaa 64-bittisessä esityksessä muotoon 0xffff800000000000. Tällainen 64-bittinen luku voitaisiin tulkita negatiiviseksi, eli siitä seuraavat lailliset muistiosoitteet olisivatkin -0x800000000000 – -0x1. ”Negatiiviseen” virtuaalimuistiavaruuden puolikkaaseen voidaan sopia liitettäväksi käyttöjärjestelmäytimen tarvitsemia alueita.

Olipa kyse segmentoidusta tai segmentoimattomasta virtuaaliosoiteavaruudesta, selkeän, lineaarisen (eli peräkkäisistä muistiosoitteista koostuvan) osoiteavaruuden toteutuminen on prosessorin ominaisuus, joka helpottaa ohjelmien, kääntäjien ja käyttöjärjestelmien tekemistä. Muistanet toisaalta, että väylän takana oleva keskusmuisti sekä I/O -laitteet ym. ovat saavutettavissa vain fyysisen, osoiteväylään koodattavan muistiosoitteen kautta. Niinpä ohjelmassa käytetyt virtuaaliset osoitteet muuntuvat fyysisiksi osoitteiksi tavalla, jonka yksityiskohtiin palataan myöhemmin luvussa 7. Kuvassa 8 havainnollistetaan seikkaa. Prosessin näkemät muistiosoitteet ovat siistissä järjestyksessä, vaikka tiedot voivat sijaita sikin sokin todellisessa muistissa. Käyttöjärjestelmä ja prosessorilaite hoitavat osoitteiden muunnoksen, joten käyttäjän ohjelman tarvitsee huolehtia vain omista virtuaaliosoitteistaan.

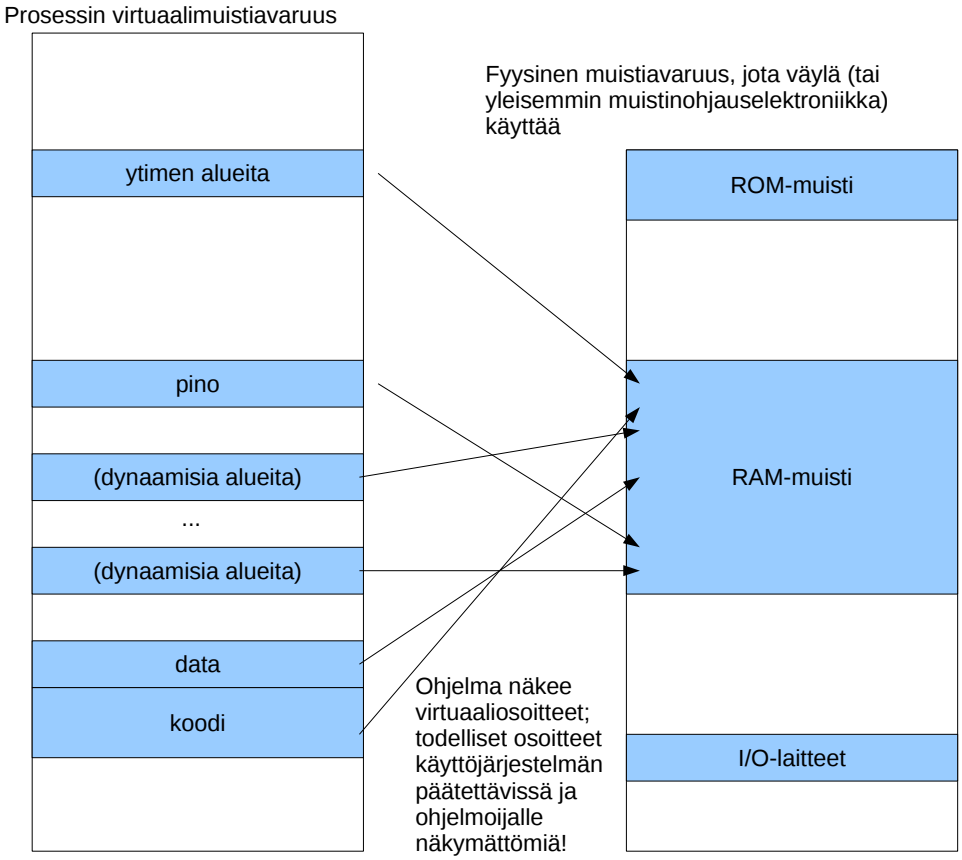
3.5 Aliohjelmien suoritus konekielitasolla

Aliohjelman käsite jollain tasolla lienee tuttu kaikille – olihan ”ohjelmointitaito” tämän kurssin esitietovaatimus. Jos ei ole tuttu, niin assembler-ohjelmoinnin kautta varmasti tulee tutuksi, kun alat ymmärtää, miten prosessori suorittaa ohjelmia ja kuinka aliohjelman suoritukseen siirtyminen ja sieltä takaisin kutsuvaan ohjelmaan palaaminen oikein toimii.

3.5.1 Mikäs se aliohjelma olikaan

Vähintään 60 vuotta vanha käsite **aliohjelma** (engl. *subprogram / subroutine*), joskus nimeltään **funktio** (engl. *function*) tai **proseduuri** (engl. *procedure*) ilmenee ohjelmointiparadigmasta riippuen eri tavoin:

- funktio-ohjelmoinnissa funktiot muodostavat puurakenteen, jonka lehtisolmuista lähtee määräytymään



Kuva 8: *Geneerinen esimerkki ohjelman näkemän virtuaaliosoiteavaruuden ja tietokonelaitteiston käsittelemän fyysisen osoiteavaruuden välisestä yhteydestä.*

pääfunktion (”juurisolmun” eli koko ohjelman) tulos. Tai sinne päin; en ole ihan asiantuntija; käykää halutessanne kurssi nimeltä Funktio-ohjelmointi, jossa ihminen kuulemma valaistuu lopullisesti. (Syksyllä 2011 kyseisellä kurssilla on tulossa myös ihan mielenkiintoinen opetusmenetelmäkokeilu!)

- imperatiivisessa ohjelmoinnissa aliohjelman avulla halutaan suorittaa jollekin datalle joku toimenpide. Aliohjelmaa kutsutaan siten, että sille annetaan mahdollisesti parametreja, minkä jälkeen kontrolli siirretään aliohjelman koodille, joka operoi dataa jollain tavoin, muuttaa mahdollisesti datan tilaa (”sivuvaikutus”) ja muodostaa mahdollisesti paluuarvoja.
- olio-ohjelmoinnissa olioinstanssille annetaan viesti, että sen pitää operoida itseään tietyllä tavoin joidenkin tarkentavien parametrien mukaisesti. Käytännössä olion luokassa täytyy olla toteutettuna viestiä vastaava **metodi** (engl. *method*) eli ”aliohjelma”, joka saa mahdolliset parametrit, muuttaa mahdollisesti olion sisäistä tilaa, ja palauttaa mahdollisesti paluuarvoja.

Ensiksi mainittuun funktio-ohjelmointiin ei tällä kurssilla kajota, mutta imperatiivisen ja olio-ohjelmoinnin näkökulmille aliohjelman käsitteestä pitäisi löytää yhteys. Olion instanssimetodin kutsu voidaan ajatella siten, että ikään kuin olisi olioluokkaan kuuluvien olioiden sisäistä dataa (eli attribuutteja) varten rakennettu aliohjelma, jolle annetaan tiedoksi (yhtenä parametrina) viite nimenomaiseen olioinstanssiin, jolle sen tulee operoida. Jotenkin näin se toteutuksen tasolla tapahtuukin, vaikkei sitä esim. Javan syntaksista huomaa. Luokkametodin (eli Javassa “static“-määreisen metodin) kutsu taas on sellaisenaankin hyvin lähellä imperatiivisen aliohjelman käsitettä, koska pelissä ei tarvitse olla mukana yhtään olioinstanssia.

Java-ohjelma ilman yhtään olion käyttöä (so. primitiivityyppisille muuttujille) pelkkiä luokkametodeja käyttäen vastaa täysin C-ohjelmointia ilman datastruktuurien (tai taulukoidenkaan) käyttöä. Se on ”pienin yhteinen nimittäjä”, jolla tavoin ei kummallakaan kielellä tietysti kummoisempaa ilotulitusta pysty toteuttamaan. Ilotulitukset tehdään Javassa luomalla olioita ja C:ssä luomalla tietorakenteita sekä operoimalla niille – toisin sanoen Javassa suorittamalla metodeja ja C:ssä suorittamalla aliohjelmia. Sekä olioista että tietorakenteista käytetään englanniksi joskus nimeä ”object” eli **objekti, olio**.

3.5.2 Aliohjelman suoritus == ohjelman suoritus

Käännös- ja ajokelpoinen C-ohjelma kirjoitetaan aina "main"-nimiseen funktioon, jolla on tietynlainen parametrilista. Käytännössä kääntäjän luoma alustuskoodi kutsuu sitä tosiaan ihan tavallisena aliohjelmana. Sama pätee Javassa: Ohjelma alkaa siten, että joku kutsuu julkista, "main"-nimistä luokkametodia. Aina ollaan suorittamassa jotakin metodia, kunnes ohjelma jostain syystä päättyy. Ei siis oikeastaan tarvitse tehdä mitään periaatteellista erottelua pää- ja aliohjelman välille prosessorin ja suorituksen näkökulmasta[7]. Minkä tahansa ohjelman suoritusta voidaan ajatella sarjana seuraavista:

- peräkkäisiä käskysuorituksia
- ehdollisia ja ehdottomia hyppyjä "IP":n arvosta toiseen
- aliohjelma-aktivaatioita, joista muodostuu kutsupino.

Peräkkäiset käskyt ja hyppykäskyt tulivatkin jo aiemmin esille x86-64:n esimerkkikäskyjen kautta. Tutkitaan seuraavaksi aliohjelmaan liittyviä käskyjä.

3.5.3 Konekieltä suoritusjärjestyksen ohjaukseen: aliohjelmat

Käydään tässä kohtaa läpi aliohjelmaan liittyvät x86-64 -arkkitehtuurin konekielikäskyt. Ensinnäkin suoranainen suoritusjärjestyksen ohjaus eli RIP:n uuden arvon lataaminen voi tapahtua seuraavilla käskyillä::

```
call MUISTIOSOITE       # Tämä on ehdoton hyppy, ihan kuin edellä
                        # esitetty jmp-käsky, mutta ennen kuin RIP:n
                        # arvo päivitetään uudeksi osoitteeksi,
                        # seuraavan käskyn osoite (joka
                        # peräkkäissuorituksessa ladattaisiin RIP:hen)
                        # painetaan pinoon. Siis ikäänkuin prosessori
                        # tekisi " pushq SEURAAVAN_KÄSKYN_OSOITE;
                        #          jmp MUISTIOSOITE "
                        # Kuitenkin molemmat asiat tapahtuvat yhdellä
                        # call-nimisellä käskyllä. Osoitteen laittaminen
                        # pinoon mahdollistaa palaamisen aliohjelmasta

ret                     # Tämä on paluu aliohjelmasta, ihan kuin edellä
                        # esitetty jmp-käsky, mutta RIP:hen laitettava
                        # arvo otetaan pinon päältä, eli muistipaikasta,
                        # jonka osoite on RSP:ssä. Eli ikäänkuin olisi
                        # "popq %rip", mutta käsky tosiaan on "ret".
```

Em. käskyillä siis hoidetaan suoritusjärjestys aliohjelmakutsun yhteydessä, ja tähän tarvitaan pinomuistia. Aliohjelmien tarpeisiin liittyy suoritusjärjestyksen lisäksi muuttujien käyttö kolmella tapaa:

- pitää voida välittää parametreja, joista laskennan lopputuloksen halutaan jollain tapaa riippuvan
- pitää voida välittää paluuarvo eli laskettu lopputulos takaisin aliohjelmaa kutsuneeseen ohjelman osaan
- pitää voida käyttää paikallisia muuttujia laskentaan.

Myöhemmin esitetään tarkemmin ns. pinokehysmalli. Siihen tulee liittymään seuraavat x86-64:n käskyt::

```
enter $540              # Tämä käsky kuuluisi heti aliohjelman alkuun.
                        # Se loisi juuri kutsutulle aliohjelmalle oman
                        # pinokehyksen, ja tässä tapauksessa varaisi
                        # tilaa 540 tavulle paikallisia muuttujia.
```

Em. ENTER-käsky tekee yhdessä operaatiossa kaikkien seuraavien käskyjen asiat, eli se on laitettu käskykantaan helpottamaan ohjelmointia tältä osin... ilman "enter"-käskyä pitäisi kirjoittaa seuraavanlainen rimpsu välittömästi aliohjelman alkuun::

[7] Hienoisia eroja toki on; esim. gcc:n kääntämän C-ohjelman main() on selkeästi uloin aktivaatio; sitä kutsutaan siten että edellisen pinokehyksen kantaosoite on nolla eli null-pointer

```
pushq %rbp                  # RBP talteen pinoon
movq  %rsp, %rbp            # Merkitään nykyinen RSP uuden pinokehyksen
                            # kantaosoitteeksi eli RBP := RSP
subq  $540, %rsp            # Varataan 540 tavua tilaa lokaaleille
                            # muuttujille.
```

Tähän komentosarjaan (tai samat asiat toimittavaan ENTER-käskyyn[8]) tullee lisää järkeä, kun luet myöhemmän pinokehyksiä käsittelevän kohdan.

Vastaavasti aliohjelman lopussa voidaan käyttää LEAVE-käskyä::

```
leave                       # Vapauttaa nykyisen pinokehyksen, eli
                            # hukkaa paikallisille muuttujille varatun
                            # tilan pinosta, ja palauttaa voimaan
                            # edellisen pinokehyksen. Käytännössä
                            # myös palauttaa pinon huipun sellaiseksi,
                            # että siitä löytyy RET–käskyn
                            # edellyttämä paluuosoite.
```

Tämä LEAVE-käsky tekisi yhdessä operaatiossa seuraavien käskyjen asiat; jos mietit asiaa hetken, huomannet, että nämä kumoavat kokonaan ENTERin tekemät tilamuutokset::

```
movq %rbp, %rsp             # RBP oli se aiempi RSP ... tilanteessa jossa
                            # oli juuri pinottu edeltävä RBP...
popq %rbp                   # Niinpä se edeltävä RBP saadaan palautettua
                            # pop–käskyllä. Ja POP–käskyssähän RSP
                            # palautuu yhdellä pykälällä pohjaa kohti.

                            # Tilanne on nyt sama kuin juuri aliohjelman
                            # alkaessa.
```

Ilo tästä on, että ENTERin ja LEAVEin välisessä koodissa SP on aina vapaana uuden kehyksen tekemiselle eli seuraavan sisäkkäisen aliohjelmakutsun tekemiselle. Tästä tosiaan lisää myöhemmin.

”Ohjelman suoritus konekielitasolla” on tämän kurssin yksi oppimistavoite. Ohjelman suoritus Java virtuaalikoneen eli JVM:n konekielitasolla on samankaltaista kuin ohjelman suoritus x86-64:n konekielitasolla tai kännykässä olevan ARM-prosessorin konekielitasolla, joten tarkoitus on oppia yleisiä ja laitteistosta riippumattomia periaatteita. Kuitenkin teemme tämän nyt valitun esimerkkiarkkitehtuurin avulla.

Ymmärretään toivottavasti, että jos kerran jokainen ohjelma on aliohjelma (tai yhtä hyvin metodi), niin ohjelmaa suoritettaessa ollaan suorittamassa aina aliohjelmaa. Kerrataan vielä ominaisuudet, joihin aliohjelmalla pitää olla mahdollisuus:

- se on saanut jostakin parametreja; ne pitää nähdä muuttujina aliohjelmassa, jotta niihin pääsee käsiksi
- se tarvitsee suorituksensa aikana paikallisia muuttujia
- sen pitää pystyä palauttamaan tietoja kutsujalleen
- sen pitää pystyä kutsumaan muita aliohjelmia.

Aliohjelmat (eli ohjelmat...) suoritetaan normaalisti käyttämällä kaikkeen ylläolevaan suorituspinoa (se kuvassa 4 esitelty lineaarinen muistialue, joka useimmiten täyttyy ovelasti osoitemielessä alaspäin). Perinteinen ja varsin siisti tapa hoitaa asia on käyttää aina aliohjelman suoritukseen **pinokehystä** (engl. *stack frame*) – toinen nimi tälle tietorakenteelle on **aktivaatiotietue** (engl. *activation record*). Rakenteen käyttöön tarvitaan virtuaalimuistiavaruudesta pinoalue ja prosessorista kaksi rekisteriä, jotka osoittavat pinoalueelle. Toinen on pinon huipun osoitin (“SP“), ja toinen pinokehyksen/aktivaatiotietueen **kantaosoitin** (joskus “BP“, engl. *base pointer*).

Perinteistä ideaa havainnollistetaan kuvassa 9. Idea on seuraavanlainen. Pinon huipulla (pienimmästä muistiosoitteesta vähän matkaa eteenpäin) sijaitsee tällä hetkellä suorituksessa olevan aliohjelman pinokehys, jolle pätee seuraavaa, kun rekisterit “BP“ ja “SP“ on asetettu oikein:

[8] Käsky on oikeasti vähän monipuolisempi; siinä voisi olla mukana toinen operandi, joka liittyisi pääsyyn aiempien toisiaan kutsuneiden aliohjelmien pinokehyksiin (eli ns. kutsupinossa taaksepäin...). Mutta ei mennä siihen nyt; riittää kun ajatellaan kerrallaan yhtä aliohjelmakutsua ja sen pinokehystä.

- Parametrit ovat pinoalueella muistissa (itse asiassa edellisen pinokehyksen puolella), mikäli aliohjelma on sellainen että se tarvitsee parametreja. Parametrien muistipaikkojen osoitteet saadaan lisäämällä kantaosoitteeseen "BP" sopivat arvot; yleensä prosessorikäskyt mahdollistavat tällaisen osoitusmuodon eli "rekisteri+lisäindeksi". Parametrien arvot saadaan laskutoimituksia varten rekistereihin siirtokäskyillä, joissa osoite tehdään tällä tavoin indeksoimalla, syntaksi esim. "16(%rbp)".
- Paikallisia muuttujia voidaan käyttää vastaavasti vähentämällä kantaosoitteesta sopivat arvot.
- Paluuosoite on tallessa tietyssä kohtaa pinokehystä.
- Edeltävän aliohjelma-aktivaation kantaosoitin on tallessa tietyssä kohtaa pinokehystä. Huomaa, että pinokehyksiä voidaan ajatella linkitettynä listana: Jokaisesta on linkki kutsuvan aliohjelman kehykseen. Pääohjelmassa linkki on NULL (ainakin gcc:n tekemillä C-ohjelmilla), jota voidaan ajatella listan päätepisteenä.
- Paikallisia muuttujia voidaan varailla ja vapauttaa tarpeen mukaan pinosta ja "SP" voi rauhassa elää "PUSH" ja "POP" -käskyjen mukaisesti.
- Uuden aliohjelma-aktivaation tekeminen on mahdollista tarvittaessa.

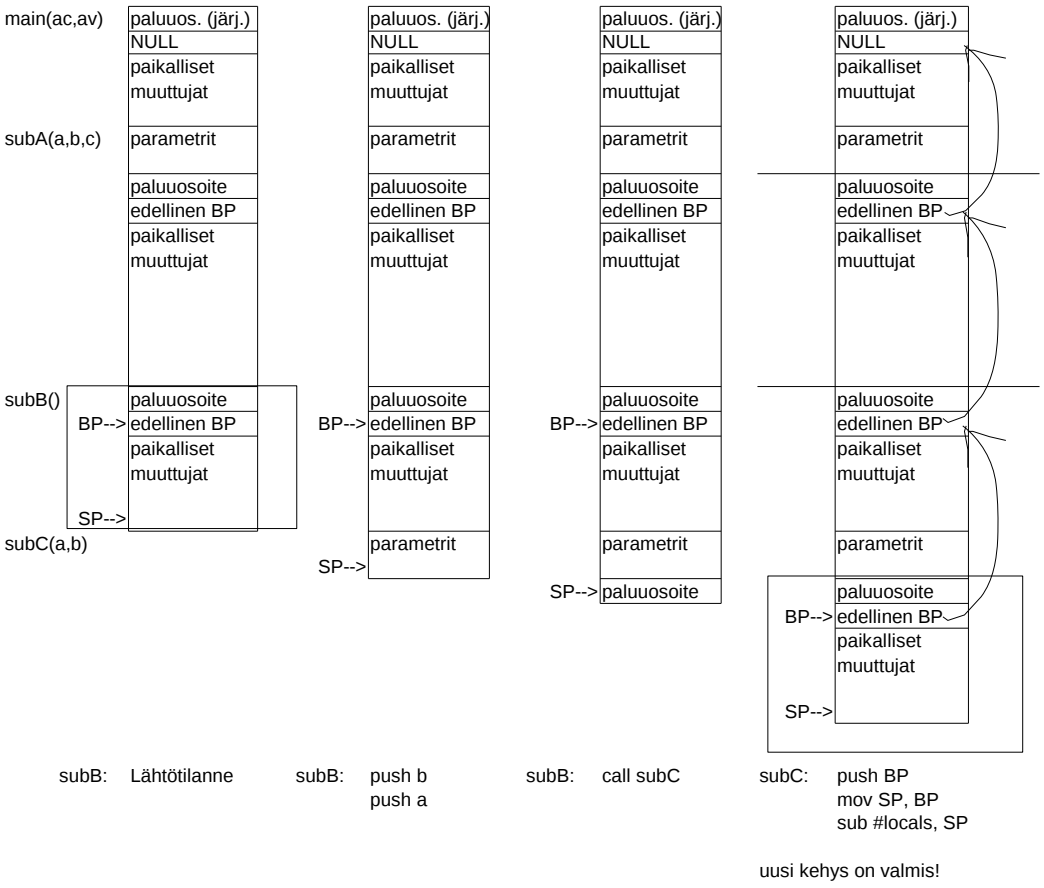
Homma toimii siis aliohjelman sisällä, vieläpä siten, että on tallessa tarvittavat tiedot palaamiselle aiempaan aliohjelmaan. Miten sitten tähän tilanteeseen päästään – eli miten aliohjelman kutsuminen (aktivointi) tapahtuu konekielisen ohjelman ja prosessorin yhteispelinä? Prosessorin käskyt tarjoavat siihen apuja, ja hyvätapaisen ohjelmoijan assembler-ohjelma tai oikein toimivan C-kääntäjän tulostama konekielikoodi osaavat hyödyntää käskyjä oikein. Tyypillisesti kutsumisen yhteydessä luodaan uusi pinokehys seuraavalla tavoin:

- kutsujan käskyt laittavat parametrit pinoon käänteisessä järjestyksessä (lähdekoodissa ensimmäiseksi kirjoitettu parametri laitetaan viimeisenä pinoon) juuri ennen aliohjelmakutsun suorittamista.
- Yleensä prosessori toimii siten, että "CALL" -käsky tai vastaava, joka vie aliohjelmaan, toteuttaa seuraavan käskyn osoitteen tallentamisen "IP":n sijasta pinon huipulle. "IP":hen puolestaan sijoittuu aliohjelman ensimmäisen käskyn osoite, joka annettiin käskyn mukana operandina.
- Seuraavassa prosessorin *fetch* -toimenpiteessä tapahtuu varsinaisesti suorituksen siirtyminen aliohjelmaan. Sanotaan, että kontrolli siirtyy aliohjelmalle.
- Aliohjelman ensimmäisen käskyn pitäisi ensinnäkin painaa nykyinen "BP" eli juuri äsken odottelemaan jääneen aktivaation kantaosoitin pinoon.
- Sen jälkeen pitäisi ottaa "BP"-rekisteri tämän uuden, juuri alkaneen aktivaation käyttöön. Kun siihen siirtää nykyisen "SP":n, eli pinon huippuosoitteen, niin se menee juuri niin kuin pitikin, ja ylläolevassa kuvassa oli esitelty.
- Ja siten "SP" vapautuu normaaliin pinokäyttöön.

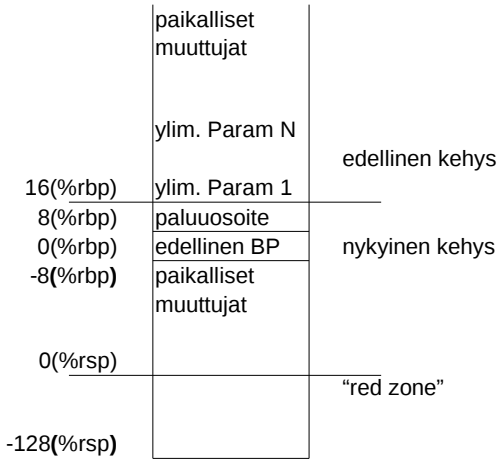
Kuten edellisessä osiossa nähtiin, pinokehyksen käyttöön on joskus tarjolla jopa prosessorin käskykannan käskyt, x86-64:ssä ENTER ja LEAVE, joilla pinokehyksen varaaminen ja vapauttaminen voidaan kätevästi tehdä.

Aliohjelmasta palaaminen tapahtuu käänteisesti:

- Aliohjelman lopussa voidaan löytää edeltävän aktivaation pinon huippu kohdasta, johon "BP" viittaa. Palautetaan siis BP:n sisältö SP:hen.
- Pinoon oli aliohjelman alussa laitettu edellisen aktivaation kantaosoitin. Nyt se voidaan poksauttaa pinon päältä takaisin BP:hen.
- Näin pinon päällimmäiseksi jäi paluuosoite, jonka "CALL" -käsky sinne laittoi. Voidaan suorittaa "RET" joka poimii IP:n seuraavan arvon pino päältä.
- Jos aliohjelma oli sellainen, että sille oli annettu parametreja pinon kautta, niin kutsuvan ohjelman vastuulla on vielä siivota oma kehyksensä, eli kutsusta palaamisen jälkeen SP:hen voi lisätä parametrien vaatiman tilan verran, millä tapaa siis parametrit "unohtuvat".
- Paluuarvo on jäänyt esim. rekisteriin, tai parametrina annettujen muistiosoitteiden kautta jokin tietorakenne on muuttunut. Aliohjelma on tehnyt tehtävänsä, ja ohjelman suoritus jatkuu (varsin todennäköisesti varsin pian uudella aliohjelmakutsulla).



Kuva 9: *Perinteinen pinokehys, ja kuinka se luodaan: eri vaiheet, osallistuvat ohjelman osat sekä "pseudo-assembler-koodi".*



Kuva 10: *Pinon käyttö x86-64:ssä kuten SVR4 AMD64 supplement sen määrittelee.*

3.5.4 Moderni laajennos: System V ABIn C-kutsumalli x86-64:lle

ABI eli **Application Binary Interface** on osa käyttöjärjestelmän määrittelyä; se kertoo mm. miten käännetty ohjelmakoodi pitää sijoitella tiedostoon, ja miten se tullaan suoritettaessa lataamaan muistiin. ABI määrittelee myös, miten aliohjelmakutsu tulee toteuttaa. Tämän asian standardointi on tarpeen, jotta eri kirjoittajien tekemät ohjelmat voisivat tarvittaessa kutsua toistensa aliohjelmia. Erityisesti voidaan tehdä yleiskäyttöisiä valmiiksi käännettyjä aliohjelmakirjastoja. Tämä ns. **kutsumalli** (engl. *calling convention*) määrittelee mm. parametrien ja paluuarvon välitysmekanismin. Malli voi vaihdella eri laitteistojen, käyttöjärjestelmien ja ohjelmointikielten välillä. Se on erittäin paljon sopimuskysymys. Siirrettävän ja yhteensopivan koodin tekeminen on vaikeaa, jos ei tiedä tätä asiaa sekä varoa siihen liittyviä sudenkuoppia. Mikä on se kutsumalli, jonka mukaista konekieltä kääntäjäsi tuottaa? Voitko vaikuttaa siihen jollakin syntaksilla tai kääntäjän argumentilla? Minkä kutsumallin mukaisia kutsuja aliohjelmakirjastosi olettaa? Mitä teet, jos työkalusi ei ole yhteensopiva, mutta haluat ehdottomasti käyttää löytämääsi binääristä kirjastoa?

Edellä esitettiin perinteinen pinokehysmalli aliohjelman kutsumiseen. Nykyaikainen prosessoriteknologia mahdollistaa tehokkaamman parametrinvälityksen: idea on, että mahdollisimman paljon parametreja viedään prosessorin rekistereissä eikä pinomuistissa – rekisterien käyttö kun on reilusti nopeampaa. GNU-kääntäjä, jota Jalavassa käytämme tällä kurssilla, toteuttaa kutsumallin, joka on määritelty dokumentaatiossa nimeltä "System V Application Binary Interface - AMD64 Architecture Processor Supplement"[9]. Olen tiivistänyt tähän olennaisen kohdan em. dokumentin draftista, joka on päivätty 3.9.2010.

Pinokehys ilmenee siten kuin kuvassa 10. Eli ihan samalta näyttää kuin yleinen pinokehysmalli. Kuitenkin nyt parametreja välitetään sekä muistissa että rekistereissä. Sääntöjä on useampia kuin tähän mahtuu, mutta todetaan, että esimerkiksi, jos parametrina olisi pelkkiä 64-bittisiä kokonaislukuja, juuri aktivoitu aliohjelma olettaa, että kutsuja on sijoittanut ensimmäiset parametrit rekistereihin seuraavasti::

```
RDI == ensimmäinen integer-parametri
RSI == toinen integer-parametri
RDX == kolmas integer-parametri
RCX == neljäs integer-parametri
R8  == viides integer-parametri
R9  == kuudes integer-parametri
```

Jos välitettävänä on enemmän kokonaislukuja, ne menevät pinon kautta. Jos välitettävänä on rakenteita, joissa on tavuja enemmän kuin rekisteriin mahtuu, sellaiset laitetaan pinoon – tai on siellä jotain muitakin sääntöjä, joiden mukaan parametri voidaan valita pinon kautta välitettäväksi vaikkei rekisterit olisi vielä sullottu täyteen. Paluuarvoille on vastaava säännöstö. Todetaan, että jos paluuarvona on yksi kokonaisluku, niin se palautetaan RAX:ssä kuten x86:n C-kutsumallissa aina ennenkin.

Näillä eväillä pitäisi pystyä tekemään kurssin perinteinen harjoitustyö, jossa käväistään hiukan syvempänä konekielen toiminnassa. Yritän tehdä aiheet sellaisiksi, että eksoottisempia säännöstöjä ei tarvitsisi käyttää. Parametreina olisi joko 64-bittisiä kokonaislukuja tai muistiosoitteita, jolloin em. kuvaus on riittävä.

[9] **System V** on 1980-luvulla tehty versio Unixista. Sitä voidaan pitää eräänlaisena standardina myöhempien Unix-variaatioiden tekemiselle, erityisesti sen versiota 4.0, jota sanotaan SVR4:ksi.

4 Käyttöjärjestelmän kutsurajapinta

Aiemmin tutustuttiin yhden ohjelman ajamiseen ja fetch-execute -sykliin. Sitä prosessori tekee ohjelmalle, ja yhden ohjelman kannalta näyttää ettei mitään muuta olekaan. Mutta nähtävästi koneissa on monta ohjelmaa yhtäaikaa, eikä nykyinen käyttäjä olisi muuten lainkaan tyytyväinen – miten se toteutetaan? Ilmeisesti käyttöjärjestelmän on jollakin tapaa hoidettava ohjelmien käynnistäminen ja hallittava niitä sillä tavoin, että ohjelmia näyttää olevan käynnissä monta, vaikka niitä suorittaisi vain yksi prosessori (nykyään prosessoreja voi olla muutamiakin, mutta niitä on selvästi vähemmän kuin ohjelmia tarvitaan käyntiin yhtä aikaa). Tässä luvussa käsitellään prosessorin toimintaa vielä sen verran, että ymmärretään yksi moniajon pohjalla oleva avainteknologia, nimittäin keskeytykset. Esityksen selkeyttämiseksi otamme käyttöön uuden sanan: **prosessi** (engl. *process*), jolla tarkoitamme yhtä suorituksessa olevaa ohjelmaa. Käsite tarkentuu myöhemmin, mutta vältämme jo keskeytyksistä puhuttaessa turhan ylimalkaisuuden sanomalla prosessiksi sitä kun prosessori suorittaa käyttäjän ohjelmaa. Huomaa, että jo arkihavainnon perusteella sama ohjelma voi olla suorituksessa useana ns. instanssina: esim. monta pääteyhteyttä eri ikkunoissa.

4.1 Keskeytykset ja lopullinen kuva suoritussyklistä

Pelkkä fetch-execute -sykli aiemmin kuvatulla tavalla ei oikein hyvin mahdollista kontrollin vaihtoa kahden prosessin välillä. Apuna tässä ovat keskeytykset. Esimerkiksi paljon laskentaa suorittava prosessi voidaan laittaa ”hyllylle” hetkeksi, ja katsoa tarvitseeko jonkun muun prosessin tehdä välillä jotakin. Tämä tapahtuu kun kellolaite keskeyttää prosessorin esimerkiksi 1000 kertaa sekunnissa. Tässä ajassa ohjelma on ehtinyt nykyprosessorilla tehdä esim. miljoona laskutoimitusta, joten se voisi hyvin lepäillä hetken. Lisäksi, jos ohjelman taas ei tarvitsekaan laskea, vaan ainoastaan odottaa I/O:ta kuten käyttäjän syöttämiä merkkejä, se pitäisi saada keskeytettyä siksi aikaa, kun jotkut toiset prosessit mahdollisesti tekevät omia operaatioitaan. Todetaan tässä kohtaa keskeytysten nivoutuminen prosessorilaitteiston toimintaan, tutkien kuitenkin vielä toistaiseksi pääasiassa yhtä prosessia; useiden prosessien tilanteeseen mennään luvussa 5.

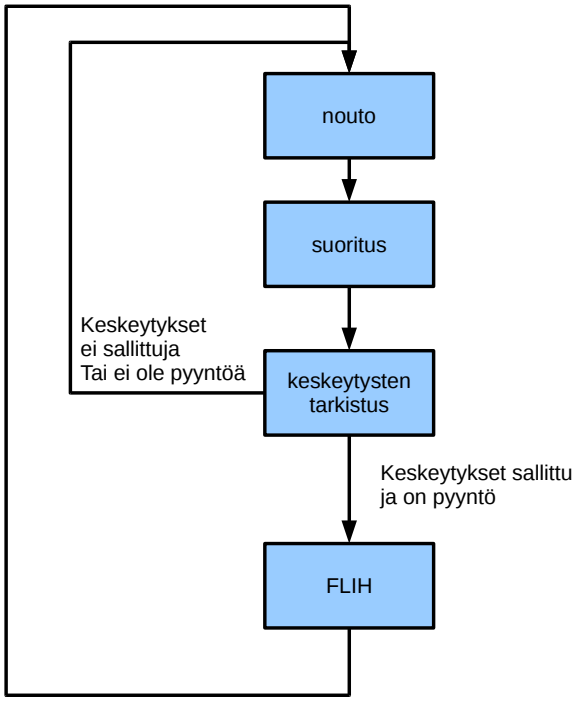
4.1.1 Suoritussykli (lopullinen versio)

Tietokonearkkitehtuuriin kuuluva ulkoinen väylä on kiinni prosessorin nastoissa, ja prosessori kokee nastoista saatavat jännitteet. Ainakin yksi nastoista on varattu **keskeytyspulssille** (engl. *interrupt signal*): Kun oheislaitteella tapahtuu jotakin uutta, eli vaikkapa ajoituskellon pulssi tai näppäimen painallus päätteellä, syntyy väylälle jännite keskeytyspulssin piuhaan kyseiseltä laitteelta prosessorille. Laite voi olla myös verkkoyhteyslaite, kovalevy, hiiri tai mikä tahansa oheislaite. Sillä on useimmiten väylässä kiinni oleva sähköinen kontrollikomponentti, jota sanotaan laiteohjaimeksi tai I/O -yksiköksi. Jos vaikka kovalevyltä on aiemmin pyydetty jonkun tavun nouto tietystä kohtaa levyn pintaa, se voi ilmoittaa keskeytyksellä olevansa valmis toimittamaan tavun dataväylälle, kunhan prosessori vain seuraavan kerran ehtii. Ja prosessori ehtii usein koko lailla välittömästi ... täydennämme aiemmin yhdelle ohjelmalle ajatellun nouto-suoritussyklin seuraavalla versiolla, joka esitetään visuaalisesti vuokaaviona kuvassa 11:

1. Nouto: Prosessori noutaa dataa “IP“-rekisterin osoittamasta paikasta
2. Suoritus: Prosessori suorittaa käskyn
3. Tuloksen säilöminen ja tilan päivitys: Käskyn suorituksen tuloksena rekisterien tila on muuttunut jollain tavoin; myös muistin sisältö voi olla muuttunut riippuen käskystä.
4. **Keskeytyskäsittely** (engl. *interrupt handling*): Jos keskeytysten käsittely on kielletty (eli kyseinen tilabitti “FLAGS“-rekisterissä kertoo niin), prosessori jatkaa sykliä kohdasta 1. Muutoin se tekee vielä seuraavaa:

   Jos prosessorin keskeytyspyyntö -nastassa on jännite, se siirtyy keskeytyskäsittelijään suorittamalla toimenpidesarjan, jonka yleisnimi on englanniksi **FLIH** eli **First-level interrupt handling**. Tarkempi selvitys alempana.
5. Tämän jälkeen prosessori jatkaa sykliä joka tapauksessa kohdasta 1, jolloin seuraava noudettava käsky on joko käyttäjän prosessin tai käyttöjärjestelmän keskeytyskäsittelijän koodia, riippuen edellä mainituista tilanteista (keskeytysten salliminen, keskeytyspyynnön olemassaolo).

Keskeytyspyynnön toteutus laitetasolla on ehkä mutkikkain prosessorin operaatio, mitä tällä kurssilla tulee vastaan (mutta ei sekään kovin mutkikas ole!). Jos haluat katsoa esim. AMD64:n manuaalia [4], löydät sieltä viisi



Kuva 11: *Prosessorin suoritussykli: nouto, suoritus, tilan päivittyminen, mahdollinen keskeytyksenhoitoon siirtyminen.*

sivua pseudokoodia, joka kertoo kaikki prosessorin toimenpiteet. Tällä kurssilla emme syvenny keskeytyspyyntöihin realistisen yksityiskohtaisesti, vaan todetaan, että kun keskeytys tapahtuu 64-bittisessä käyttäjätilassa (normaalin sovellusohjelman normaali suoritustila x86-64:ssä), sen jälkeen on voimassa seuraavaa:

- RSP:hen on ladattu uusi muistiosoitin, eli pinon paikka on eri kuin keskeyttävän prosessin pino; tästä alkaen käytössä on siis **Kernel-pino** (engl. *kernel stack*) (prosessori löytää uuden pino-osoitteen tietorakenteista, joiden sijainnin käyttöjärjestelmä on kertonut sille käyttäen tarkoitusta varten suunniteltuja järjestelmärekisterejä [10]).
- RIP:hen on ladattu muistiosoite, josta jatkuu pyydetyn keskeytyskäsittelijän suoritus. Perinteinen tapa hoitaa lataus on ollut ns. **keskeytysvektori** (engl. *interrupt vector*), käyttöjärjestelmän valmistelema muistialue, jossa on hyppyosoitteet ohjelmapätkiin, joilla oheislaitteiden pyytämät keskeytykset hoidetaan. Oheislaitteilla on omat indeksinsä, jonka perusteella prosessori käy noutamassa RIP:n osoitteen keskeytysvektorista. Nykyisissä prosessoreissa, kuten x86-64:ssä käytetään samankaltaista menettelyä: käyttöjärjestelmä valmistelee keskeytyskäsittelijöiden muistiosoitteet taulukkoon, jonka sijaintipaikan se kertoo prosessorille systeemirekisterien avulla. Keskeytyksen tullessa prosessori löytää uuden RIP:n tuosta tietorakenteesta.
- Keskeytyskäsittelyyn siirtyminen ei välttämättä edellytä siirtoa prosessista toiseen; käyttöjärjestelmän koodi on voitu liittää prosessin virtuaaliavaruuteen, ja samaa prosessia vain jatketaan nyt ”kernel running” -tilassa.
- Keskeytyskäsittelijän käyttämän pinon päällä (siis uuden RSP:n osoittamassa pinossa) on tärkein osuus keskeytetyn prosessin kontekstista:
  - RFLAGSin sisältö ennen keskeytystä
  - Ennen keskeytystä tulossa olleen seuraavan käskyn muistiosoite (eli se joksi RIP olisi päivitetty peräkkäissuorituksessa)
  - keskeytetyn prosessin pino-osoitin (eli RSP:n sisältö ennen INTin suoritusta).

  Muita rekisterejä ei ole laitettu mihinkään; niissä on yhä keskeytetyn prosessin tilanne.
- RFLAGS on päivitetty seuraavin tavoin: Prosessori on käyttöjärjestelmätilassa ja keskeytykset ovat toistaiseksi kiellettyjä (myöhemmin nähdään miksi keskeytykset on kiellettävä, eli miksi tarvitaan ns. atomisia toimenpiteitä, jotka prosessori suorittaa loppuun saakka ilman uutta keskeytystä)
- Muutakin voi olla, mutta tuossa on tärkeimmät asiat, joiden avulla keskeytys saadaan hoidettua, ja suoritus siirrettyä käyttäjän prosessilta käyttöjärjestelmälle.

[10] tarkemmin sanottuna x86-64:ssä on käytettävissä neljä eri suojaustasoa, joilla jokaisella on eri pino. Kaksikin jo silti riittäisi käyttökelpoisen käyttöjärjestelmän tekemiseksi, eli käyttäjän pino ja käyttöjärjestelmäpino.

- Käyttöjärjestelmän keskeytyskäsittelijä pääsee sitten suoritukseen.
- Keskeytynyt prosessi pääsee taas joskus myöhemmin jatkamaan tallentuneesta tilanteestaan, riippuen käyttöjärjestelmän tekemistä ratkaisuista.

RIP, RSP ja RFLAGS on välttämätöntä saada tallennettua atomisessa keskeytyskäsittelyssä (first-level interrupt handling), koska ne ovat kaikkein herkimmin muuttuvat rekisterit; esim. RFLAGS muuttuu melkein jokaisen käskyn jälkeen ja RIP ihan jokaisen käskyn jälkeen. Jos keskeytyskäsittelyssä pitää tehdä kontekstin vaihto eli vaihtaa suoritusvuorossa olevaa prosessia, käsittelijän pitää erikseen tallentaa kaikkien rekisterien sisältö ja sen on huolehdittava tarkasti siitä, että se itse ei ole vahingossa muuttanut (tai päästänyt uusia keskeytyksiä muuttamaan) rekisterien arvoja ennen kontekstin tallennusta.

Tämän kuvauksen tavoite oli antaa yleistietoa, jonka pohjalta on jatkossa mahdollisuus ymmärtää paremmin käyttöjärjestelmän osien toimintaa: prosessien vuorottelua, viestinvälitystä ja synkronointia, laiteohjausta sekä I/O:ta. Relevantti kysymys, joka voi herätä, on, voiko pyydetty keskeytys jäädä palvelematta, jos edellinen käsittely kestää pitkään siten että keskeytykset on kielletty. Tottahan toki. Tietokone voi kaatua lopullisesti, jos käyttöjärjestelmän keskeytyskäsittelijässä on ohjelmointivirhe, joka ”jää jumiin” eikä salli uusia keskeytyksiä. Jonkinlainen jonotuskäytäntö voi olla toteutettu laitteistotasolla. Useimmiten myös laitteilla on prioriteetit: korkeamman prioriteetin keskeytys voi aiheuttaa FLIHiin siirtymisen, vaikka prosessori olisi jo suorittamassa alemman prioriteetin keskeytystä. Yksityiskohtiin tässä ei mennä, vaan jätetään prioriteetit ohimennen mainituksi.

Joka tapauksessa esim. multimedialaitteiden keskeytyksiä on syytä päästä palvelemaan mahdollisimman nopeasti pyynnön jälkeen, jotta median tulostukseen ei tule katkoja. Keskeytyskäsittelijän koodi tulisi olla siten tehty, että mahdollisimman pian käsittelijään siirtymisen jälkeen se suorittaa “sti“ -konekäskyn (”set interrupt enable flag”), eli sallii prosessorille uuteen keskeytykseen siirtymisen vaikkei edellinen käsittely olisi kokonaan loppunutkaan.

Muutamia lisähuomioita keskeytyskäsittelystä:

- koskaan ei voi tietää etukäteen kuinka monta nanosekuntia, millisekuntia tai viikkoa esimerkiksi kestää ennen kuin käyttäjän ohjelman seuraava käsky suoritetaan, koska voi tulla käsiteltävä keskeytys joltakin laitteelta. Jos tarkka ajoitus on välttämätöntä, pitää olla käyttöjärjestelmä ja laitteisto, jotka voivat tarjota riittävän tarkan ajastuksen erityisenä palveluna (puhutaan reaaliaikakäyttöjärjestelmästä). Kriittisille ohjelmistoille on mahdollistettava ”etuajo-oikeus” eli prioriteetti, jolla ne voivat päästä suoritukseen juuri silloin kun niiden tarvitsee. [11]
- jos käytetään jaettuja resursseja (muistialueita, tiedostoja, I/O-kanavia, viestijonoja, ...) usean eri prosessin välillä, ei ilman käyttöjärjestelmältä pyydettyä poissulkupalvelua voida esim. tietää, mitkä kaikki muut prosessit ovat ehtineet kahden oman konekielikäskyn välissä käydä muuttamassa tai lukemassa resurssien sisältöjä.

4.1.2 Konekieltä suoritusjärjestyksen ohjaukseen: keskeytyspyyntö

Käsitellään vielä **ohjelmallinen keskeytyspyyntö**, joka on prosessorin käsky. Yleisiä nimiä ja assemblerkielisiä ilmauksia sille on esim. ”interrupt, INT”, ”supervisor call, SVC”, ”trap”. Keskeytyspyyntö on avain käyttöjärjestelmän käyttöön: kaikki käyttöjärjestelmän palvelut ovat saatavilla vain sellaisen kautta. Eli **käyttöjärjestelmän rajapinta** (engl. *system call interface*) näyttäytyy joukkona palveluita, joita käyttäjän ohjelmassa pyydetään ohjelmoidun keskeytyksen avulla. Keskeytyspyyntö näyttäisi x86-64:ssä seuraavanlaiselta::

```
int $10                 # Pyydetään keskeyttämään tämä prosessi
                        # ja suorittamaan keskeytyskäsittelijä
                        # numero 10. Oletus on, että jossain
                        # vaiheessa suoritus palaa tätä käskyä
                        # seuraavaan käskyyn, ja
                        # käyttöjärjestelmäkutsu on toteuttanut
                        # palvelunsa. Paitsi tietysti, jos pyyntö
                        # on tämän prosessin lopettaminen eli
                        # exit() -palvelu. Silloin oletus on, että
                        # seuraavaa käskyä nimenomaan ei koskaan
```

[11] Tai sitten on käytettävä jotakin muuta kuin moniajokäyttöjärjestelmää; sekin on tottakai mahdollista, riippuen rakennettavan järjestelmän vaatimuksista. 1980-luvulla silloiset hienoimmat tietokonepelit saatettiin toteuttaa kokonaan käyttöjärjestelmätilassa ilman moniajoa; niihin saatiin äärimmäisen tarkka ajoitus laskemalla kuinka monta kellojaksoa minkäkin koodipätkän suorittaminen kesti. Keskeytykset eivät niitä päässeet häiritsemään.

```
# suoriteta.
```

Prosessori suorittaa ohjelmoidun keskeytyksen kohdalla samanlaisen FLIH-käsittelyn kuin laitekeskeytyksessäkin. Ohjelmoidussa keskeytyksessä käsittelijän osoite riippuu INT-käskyn 8-bittisestä operandista, eli ohjelma voi pyytää mitä tahansa käsittelijää väliltä 0-255. Käyttöjärjestelmän palvelut löytyvät usein jollain tietyllä numerolla, tai muutamalla eri numerolla palveluiden tyypin mukaan jaoteltuna. Valinta riippuu siis täysin siitä, miten käyttöjärjestelmä on toteutettu.

Käyttöjärjestelmän palvelun tarvitsemat parametrit pitää olla laitettuna sovittuihin rekistereihin ennen keskeytyspyyntöä; tietenkin käyttöjärjestelmän rajapintadokumentaatio kertoo, mihin rekisteriin pitää olla laitettu mitäkin. Parametrina voi olla esim. muistiosoite tietynlaisen tietorakenteen alkuun, jolloin käytännössä voidaan välittää käyttöjärjestelmän ja sovelluksen välillä mielivaltaisen muotoista dataa. Toinen tyypillinen tapa on välittää kokonaisluvuiksi koodattuja "deskriptoreita" tai "kahvoja" jotka yksilöivät joitakin käyttöjärjestelmän kapseloimia tietorakenteita kuten semaforeja, prosesseja (PID), käyttäjiä (UID), avoimia tiedostoja (tiedostodeskriptori), viestijonoja ym.

Paluu käyttöjärjestelmäkutsusta tapahtuu seuraavasti::

```
iret            # Keskeytyskäsittelijän lopussa pitäisi olla tämä
                # käsky. Se on käänteinen INT–käskylle, eli prosessori
                # ottaa pinosta INTin aikoinaan sinne laittamat asiat
                # (tai siis olettaa että siellä on juuri ne)
                # ja sijoittaa ne asiaankuuluviin paikkoihin. Keskeytetyn
                # prosessin kannalta tämä näyttää siltä kuin mitään ei
                # olisi tapahtunutkaan: koko konteksti, mukaanlukien
                # RFLAGS, RSP, RIP ovat niinkuin ennenkin, paitsi jos
                # käyttöjärjestelmäpalvelu on antanut paluuarvon, joka
                # löytyy nätisti dokumentaatiossa kerrotusta rekisteristä,
                # tai se on muuttunut muistialueella, jonka osoite oli
                # kutsun parametrina.
```

Samalla käskyllä palataan sekä ohjelmoidun keskeytyksen että I/O:n aiheuttaman keskeytyksen käsittelijästä. Muistutus: prosessori on jatkuvasti yhtä "tyhmä" kuin aina, eikä se IRETin kohdalla tiedä, mistä se on siihen kohtaan tullut. Se kun suorittaa yhden käskyn kerrallaan eikä näe muuta. Käyttöjärjestelmän tekijän vastuulla on järjestellä keskeytyskäsittelyt oikeellisiksi, ja mm. IRET oikeaan paikkaan koodia, jossa käytössä olevasta pinosta löytyy paluuosoite.

Lopuksi todetaan kaksi käskyä keskeytyksiin liittyen::

```
cli             # Estää keskeytykset; eli kääntää RFLAGSissä olevan
                # keskeytyslipun nollaksi (clear Interrupt flag)

sti             # Sallii keskeytykset; eli kääntää RFLAGSissä olevan
                # keskeytyslipun ykköseksi (set Interrupt flag)
```

Nämä käskyt on sallittu vain käyttöjärjestelmätilassa (käyttäjän ohjelma ei voi estää keskeytyksiä, joten ainoa tapa saada aikaan atomisesti suoritettavia ohjelman osia on pyytää käyttöjärjestelmän palveluja, esimerkiksi MUTEX-semaforia, josta puhutaan myöhemmin). Kaikkia keskeytyksiä ei voi koskaan estää, eli on ns. "non-maskable" keskeytyksiä. Niiden käsittely ei saa kovin kummasti muuttaa prosessorin tai muistin tilaa, vaan keskeytyskäsittelijän on luotettava siihen, että jos keskeytykset on kielletty, niin silloin suoritettava käsittelijä on ainoa, joka voi muuttaa asioita järjestelmässä. Tiettyjä toteutuksellisia hankaluuksia syntyy sitten jos on monta prosessoria: Yhden prosessorin keskeytysten kieltäminen kun ei vaikuta muihin prosessoreihin, ja muistihan taas on kaikille prosessoreille yhteinen... mutta SMP:n yksityiskohtiin ei mennä nyt syvällisemmin; todetaan, että niitä varten tarvitaan taas tiettyjä lisukkeita käskykantaan, että muistinhallinta onnistuu ilman konflikteja. Prosessorissa on voitava suorittaa käskyjä, jotka tarvittaessa keskeyttävät ja lukitsevat muiden prosessorien sekä väylän toiminnan. (Monen prosessorin synkronointi osaltaan syö yhteistä prosessoriaikaa, mistä syystä kaksi rinnakkaista prosessoria ei ole ihan tasan kaksi kertaa niin nopea kokonaisuus kuin yksi kaksi kertaa nopeampi prosessori; rinnakkaisprosessoinnilla saadaan kuitenkin nopeutusta paljon halvemmalla kuin yhtä prosessoria nopeuttamalla, ja fysiikan lait rajoittavat yhden prosessorin nopeutta).

4.2 Tyypillisiä käyttöjärjestelmäkutsuja

Ilmeisesti käyttöjärjestelmältä tarvitaan sellaisia kutsuja, joilla ohjelmia voidaan käynnistää, lopettaa ja väliaikaisesti keskeyttää. Ohjelmien täytyy voida pyytää syötteitä I/O -laitteilta, ja niiden pitää voida käyttää tiedos-

toja. Niiden täytyy pystyä saamaan dynaamisia muistialueita kesken suoritusta. Toki niiden täytyy myös pystyä kommunikoimaan toisille ohjelmille. Kaikkiin näihin liittyy tietokonelaitteiston käyttöä, ja sitähän saa hallita vain käyttöjärjestelmä. Käyttöjärjestelmän **kutsurajapinta** (engl. *system call interface*) on "käyttäjän portti" käyttöjärjestelmän palveluihin. Edellä nähtiin prosessorin käskykannan vastaava käsky "int", jolla suoritettava ohjelma voi pyytää keskeyttämään oman suorituksensa ja siirtymään käyttöjärjestelmän keskeytyskäsittelijään. Katsotaan ensimmäisenä esimerkkinä C-kielisen ohjelman lopetusta erilaisin tavoin::

```
#include<stdlib.h>
int main(int argc, char *argv[]){

  /* Inline assembler for exiting without need of stdlib.
     Exit with code 13 this time. */
  asm (                                                \
       "movl $1,%eax\n"                                \
       "movl $13,%ebx\n"                               \
       "int $128\n"                                    \
       );

  /* Explicitly ask for shutdown with exit code 122*/
  exit(122);

  /* The "C-style" end-of-program with exit code 123*/
  return 123;
}
```

Tietenkin ohjelma on esimerkkinä järjetön: se ei tee mitään, ja sitten loppuu heti ensimmäiseen tapaan lopettaa, eikä siis sen jälkeistä koodia oikeasti suoritettaisi. Lisäksi se ilmoittaa päättyneensä virhetilanteeseen, jonka numero olisi epäonnen luku 13. Kuitenkin tässä nähdään "C:n laiteläheisyyden mahdollistama" suora konekielinen koodi eli "inline assembler", jolla voi suoraan tehdä käyttöjärjestelmäkutsun. Seuraavana on exit() -aliohjelman kutsuminen sekä lopuksi "C:n sopimuksen mukainen" lopetus main() -aliohjelman paluuarvona. Viimeistä tapaa tulisi käyttää C-ohjelmissa. Koodin keskimmäisen kohdan kautta päästään käsittelemään käyttöjärjestelmän kutsurajapinnan käytäntöä:

- Yleensä aliohjelmakirjastot hoitavat kutsujen tekemisen, joten ohjelmoijalle näyttää siltä että hänen ohjelmansa kutsuu esim. C:n aliohjelmaa (tai Javan/C#:n metodia) kuten tässä tapauksessa exit().
- Lopulta kuitenkin jossakin kirjastossa on ohjelmanpätkä, jonka konekielinen koodi sisältää nimenomaan ohjelmoidun keskeytyksen.[12]
- Käyttöjärjestelmäkutsun tekeminen on käsitteenä samanlainen kuin aliohjelman kutsuminen: pyydetään jotakin tiettyä palvelua, ja pyyntöön liitetään tietyt parametrit. Myös paluuarvoja voidaan saada, ja niitä voidaan käyttää hyödyksi kutsuvassa ohjelmassa.
- Parametrien välitys konekielitason käyttöjärjestelmäkutsulle on tapahduttava rekisterien kautta. Se on tehokasta, ja pino-osoitinhan muuttuu joka tapauksessa käyttöjärjestelmäkutsun kohdalla käytöjärjestelmän pinoksi, joten parametrien kaiveleminen toisesta pinosta olisi työlästä. Samoin paluuarvot tulevat rekisterissä.

Esimerkkinämme olevasta Linux-käyttöjärjestelmästä löytyy exit() -kutsun lisäksi paljon muita kutsuja, joista osa tulee tutuksi myöhemmissä luvuissa. Joillakin on selkeitä nimiä, kuten open(), close(), read(), write(). Näiden toiminta ja monien muiden merkitys ylipäätään avautuu vasta, kun mennään asiassa hieman eteenpäin.

[12] Tämä on yksi esimerkki laitteen ja ohjelmiston rajapinnasta sekä ohjelmille tyypillisestä kerrosmaisesta rakenteesta ("matalan tason kirjastot", mahdolliset "korkeamman tason kirjastot", jne.., ja päällä "korkean tason" sovellusohjelma).

5 Prosessi ja prosessien hallinta

Sana **prosessi** (engl. *process*) mainittiin jo aiemmin, koska esitysjärjestys tässä niin edellytti, mutta esitellään se vielä uudelleen, koska prosessi on käyttöjärjestelmän perusyksikkö – kuvaahan se sovellusohjelman suorittamista, jonka jouhevuus ilman muuta on päämäärä tietokoneen käyttämisessä.

5.1 Prosessi, konteksti, prosessin tilat

Konteksti (engl. *context*) on prosessorin tila, kun se suorittaa ohjelmaa. Kun prosessi keskeytyy prosessorin keskeytyskäsittelyssä, on tärkeää että konteksti säilyy, toisin sanoen johonkin jää muistiin rekisterien arvot (mm. IP, FLAGS, datarekisterit, osoiterekisterit). Huomioitavaa:

- jokaisella prosessilla on oma kontekstinsa.
- vain yhden prosessin konteksti on muuttuvassa tilassa yksiprosessorijärjestelmässä (kaksiprosessorijärjestelmässä kahden prosessin kontekstit, jne...)
- muiden prosessien kontekstit ovat ”jäädytettynä” (käyttöjärjestelmä pitää niitä tallessa, kunnes se päättää antaa prosessille taas suoritusvuoron, jolloin konteksti siirretään rekistereihin niin kuin mitään ei olisi tapahtunutkaan).

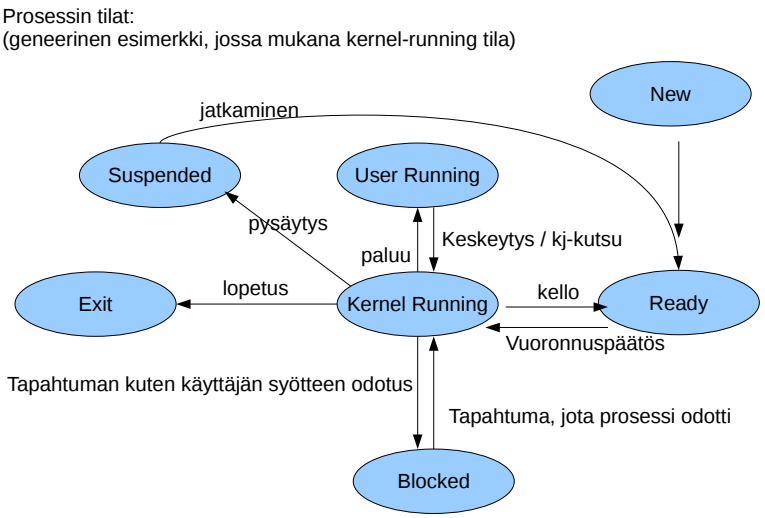
Prosessorissa täytyy tapahtua käyttöjärjestelmäohjelmiston koordinoima **kontekstin vaihto** (engl. *context switch*), kun prosessien suoritusvuoroja vaihdellaan. Kontekstin vaihdon lisäksi käyttöjärjestelmä suorittaa toki muutakin kirjanpitoa keskusmuistissa ja levyllä sijaitsevissa tietorakenteissa. Prosessia ei välttämättä tarvitse vaihtaa joka keskeytyksellä; se riippuu keskeytyksen luonteesta ja käyttöjärjestelmän vuorontajaan valituista algoritmeista. Yleensä mm. kellokeskeytys moniajojärjestelmässä kuitenkin tarkoittaa nykyisen prosessin aikaviipaleen loppumista, ja ainakin silloin vaihdetaan prosessia, mikäli joku toinen prosessi on valmiina suoritukseen. Toisaalta joku I/O, vaikkapa näppäinpainallus, voidaan lyhyesti kirjata käyttöjärjestelmän sisäiseen puskuriin odottamaan myöhempää käsittelyä, ja keskeytynyttä prosessia voidaan jatkaa ilman mitään vaihdosta aina aika-askeleen loppuun tai muuhun luonnolliseen vaihdokseen saakka (käytännössä siis prosessin tekemään ohjelmoituun keskeytykseen, jossa esim. odotellaan I/O:ta).

Kuvassa 12 on esimerkki tiloista, joissa kukin prosessi voi olla. Perustila ilmeisesti on ”User running”, jossa prosessin ohjelmakoodia suoritetaan. Keskeytyksen tullessa tilaksi vaihtuu ”Kernel running”, jossa prosessin kontekstissa (poislukien ohjelma- ja pino-osoittimet) suoritetaan käyttöjärjestelmän ohjelmakoodia. Tästä tilasta voi tulla paluu ”user running tilaan” tai sitten:

- Kellokeskeytyksen kohdalla prosessi voidaan siirtää pois suorituksesta odottelemaan seuraavaa vuoroaan ns. ”ready” -tilaan (valmiina suorittamaan seuraavalla aika-askeleella). Tällöin tapahtuu prosessin vaihto, mikä tarkoittaa että jokin toinen prosessi siirtyy ”ready” -tilasta ”running”-tilaan (”kernel-running”-tilan kautta ”user-runningiin”, vaikkei tällä yksityiskohdalla niin väliä olekaan).
- Käyttöjärjestelmäkutsun kohdalla, mikäli kutsuun liittyy odottelua (I/O tai muu syy), prosessi siirtyy ”blocked”-tilaan; sanotaan että kutsu blokkaa prosessin. Prosessi voi myös itse pyytää siirtymistä ”suspended” -tilaan.
- Riippuen toteutuksesta prosessi voidaan siirtää ”ready” -tilaan myös muiden keskeytysten kohdalla (esim. jos keskeytys tarkoitti että jonkun toisen prosessin odottama I/O tuli valmiiksi, saatetaan vaihtaa suoraan tuohoon toiseen prosessiin).

Siirtyminen pois ”blocked” -tilasta tapahtuu silloin kun prosessin odottama tapahtuma tulee valmiiksi (esim. I/O-keskeytyksen käsittelyn yhteydessä havaitaan että odotettu I/O-toimenpide on tullut valmiiksi; muita odottelua vaativia tapahtumia tullaan näkemään myöhemmin, kun käsitellään prosessien kommunikointia). Riippuen toteutuksesta, ”blocked”-tilasta voi tulla siirto suoraan ”running”-tilaan tai sitten ”ready”-tilaan.

”Suspended”-tilaan prosessi voi siirtyä omasta tai toisen prosessin pyynnöstä, kun sen suoritus halutaan jostain syystä väliaikaisesti pysäyttää kokonaan. Se on tila, jossa prosessi on ”syväjäässä” odottelemassa myöhempää eksplisiittistä palautusta tästä tilasta. Luonnollisesti prosessin tiloja ovat myös ”New”, kun prosessi on juuri luotu ja ”Exit”, kun prosessin suoritus päättyy.



Kuva 12: *Prosessin tilat (geneerinen esimerkki).*

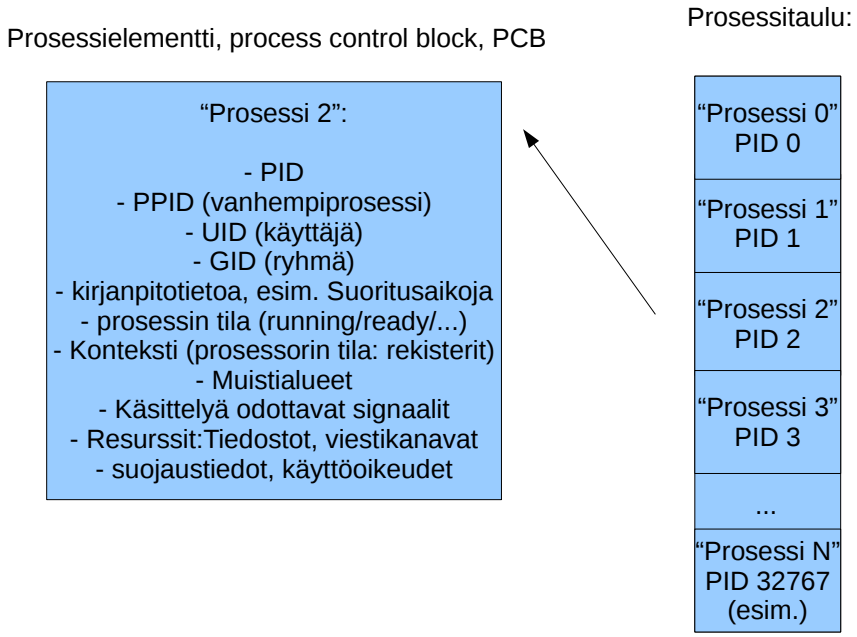
5.2 Prosessitaulu

Prosessitaulu (engl. *process table*) on keskeinen käyttöjärjestelmän ylläpitämä tietorakenne. Melkeinpä kaikki käyttöjärjestelmän osat käyttävät prosessitaulun tietoja, koska siellä on kunkin käynnistetyn ohjelman suorittamiseen liittyvät asiat. Prosessitaulu sisältää useita ns. **prosessielementtejä** (engl. *process control block, PCB*), joista kukin vastaa yhtä prosessia. Prosessitaulua havainnollistetaan kuvassa 13.

Prosessielementtien määrä on rajoitettu – esim. positiivisten, etumerkillisten, 16-bittisten kokonaislukujen määrä, eli 32768 kpl. Enempää ei pystyisi prosesseja luomaan. Esim. tuollainen määrä kuitenkin on jo aika riittävä. Esim. 3000 käyttäjää voisi käyttää yli kymmentä ohjelmaa yhtä aikaa. Luultavasti prosessoriteho tai muisti loppuisi ennemmin kuin prosessielementit.

Yhden PCB:n sisältö on seuraava:

- prosessin yksilöivä tunnus eli prosessi-ID, "PID". Voi olla toteutuksen kannalta PCB:n indeksi prosessitaulussa.
- konteksti eli prosessorin "user-visible registers"sillä hetkellä kun viimeksi tuli keskeytys, joka johti tämän prosessin vaihtamiseen pois Running-tilasta.
- PPID (parent eli vanhempiprosessin PID)
- voi olla PID:t myös lapsiprosesseista ja sisaruksista
- UID, GID (käyttäjän ja ryhmän ID:t; tarvitaan käyttöjärjestelmän vastuulla olevissa tietosuojatarkistuksissa)
- prosessin tila (ready/blocked/jne...) ja prioriteetti
- resurssit
  - tälle prosessille avatut/lukitut tiedostot (voivat olla esim. ns. deskriptoreita, eli indeksejä taulukkoon, jossa on lisää tietoa käsiteltävänä olevista tiedostoista)
  - muistialueet (koodi, data, pino, dynaamiset alueeet)
- viestit muilta prosesseilta, mm.
  - Sanomanvälitysjono
  - Signaalijono
  - putket



Kuva 13: *Käyttöjärjestelmän keskeisimmät tietorakenteet: prosessielementti ja sellaisista koostuva prosessitaulukko.*

5.3 Vuorontamismenettelyt, prioriteetit

Käyttöjärjestelmän osio, joka hoitaa prosessoriajan jakamista prosessorien kesken on nimeltään **vuorontaja** (engl. *scheduler*). Kuvassa 14 esitetään esimerkki yksinkertaisesta vuorontamismenettelystä nimeltä **kiertojono** (engl. *round robin*), joka jakaa tasavertaisesti aikaa kaikille laskentaa tarvitseville prosesseille. Prosessit sijaitsevat jonossa peräkkäin, ja jos aika-annos loppuu yhdeltä, siirrytään aina seuraavaan. Jos taas prosessi blokkautuu ennen aika-annoksen loppua, täytyy se tietenkin ottaa pois tästä suoritusvalmiiden kiertojonosta ja siirtää jonoon, jossa on tapahtumaa odottavia prosesseja (yksi tai useampia).

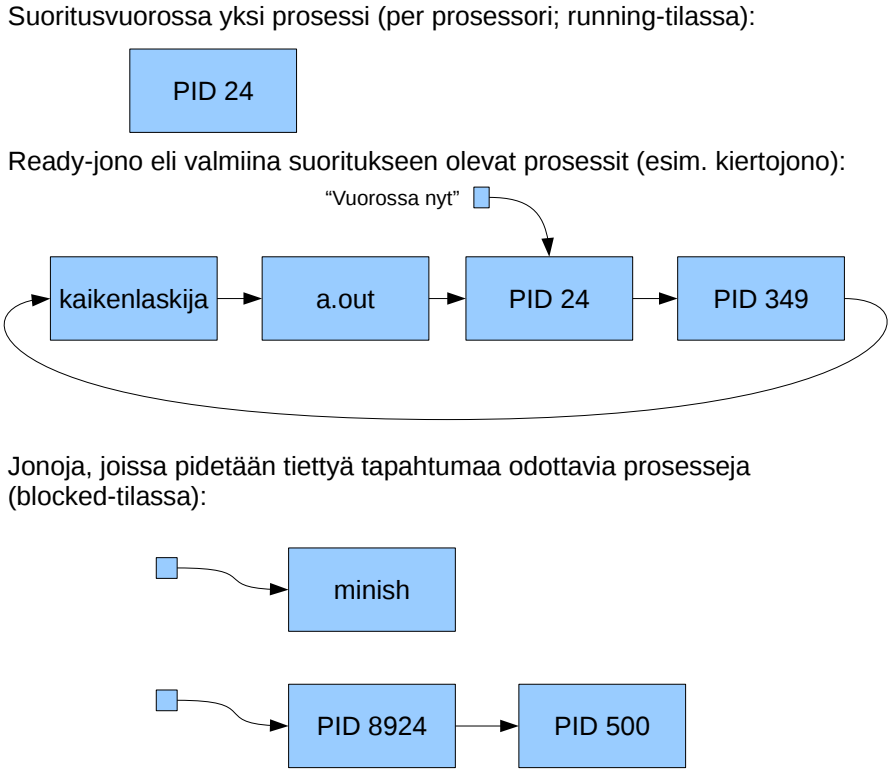
Round robin -menettelyssä ”ohiajot” eivät ole mahdollisia, joten se ei sovellu reaaliaikajärjestelmiin, kuten musiikkiohjelmien suorittamiseen. Myöhemmin, luvussa 10.2 palataan tutkimaan vaihtoehtoisia vuoronnusmenettelyjä.

5.4 Prosessin luonti fork():lla

Käyttöjärjestelmäkutsu fork() on ainoa tapa, jolla perus-Unixissa voi kukaan tehdä uuden prosessin. Linuxissa fork() toimii, koska yleensäkin unix-maiset käyttöjärjestelmäkutsut siinä toimivat – kuitenkin fork() on toteutettu Linuxissa erityistapauksena clone() -kutsusta, jolla voi tehdä myös säikeitä (Linuxissa säie on ”light weight process”; prosessin ja säikeen ”aste-erot” ovat hienosäädettävissä clone()-kutsun parametreilla, ja isoimmillaan ero on niin iso, että toteutuu perus-unixin fork()). Säikeistä lisää myöhemmin.

Käyttöjärjestelmä luo fork()-kutsua käsitellessään hiukan muutetun kopion nykyisestä prosessista (joka pyysi forkkausta eli haaroitusta). Kuvan 15 ensimmäinen siirtymä ylhäältä alaspäin havainnollistaa tapahtumaa. Uudella prosessilla on oma PID sekä omat PCB-tietonsa, onhan se uusi prosessiyksilö (koko forkin idea on luoda uusia prosesseja). PCB:n sisältö on kuitenkin pääasiassa kopio vanhempiprosessin tiedoista:

- PID on uusi
- vanhempiprosessin PID eli PPID (”parent PID”) on tietysti uusi
- prosessin konteksti on lähes sama kuin vanhempiprosessilla:
  - suoritus jatkuu samasta kohtaa, joten onnistuneen forkin jälkeen on prosessi todellakin ”kahdentunut”.
  - ainoa ero on fork() -kutsun paluuarvo eli esimerkiksi yhden paluuarvoa kuvaavan rekisterin sisältö.
- muut tiedot kopioituvat identtisinä; lapsiprosessilla on siis vanhempansa UID, GID (käyttäjän ja ryhmän ID:t) sekä samat resurssit (ml. tiedostot ja muistialueet kuten koodi, data, pino, dynaamiset alueet).



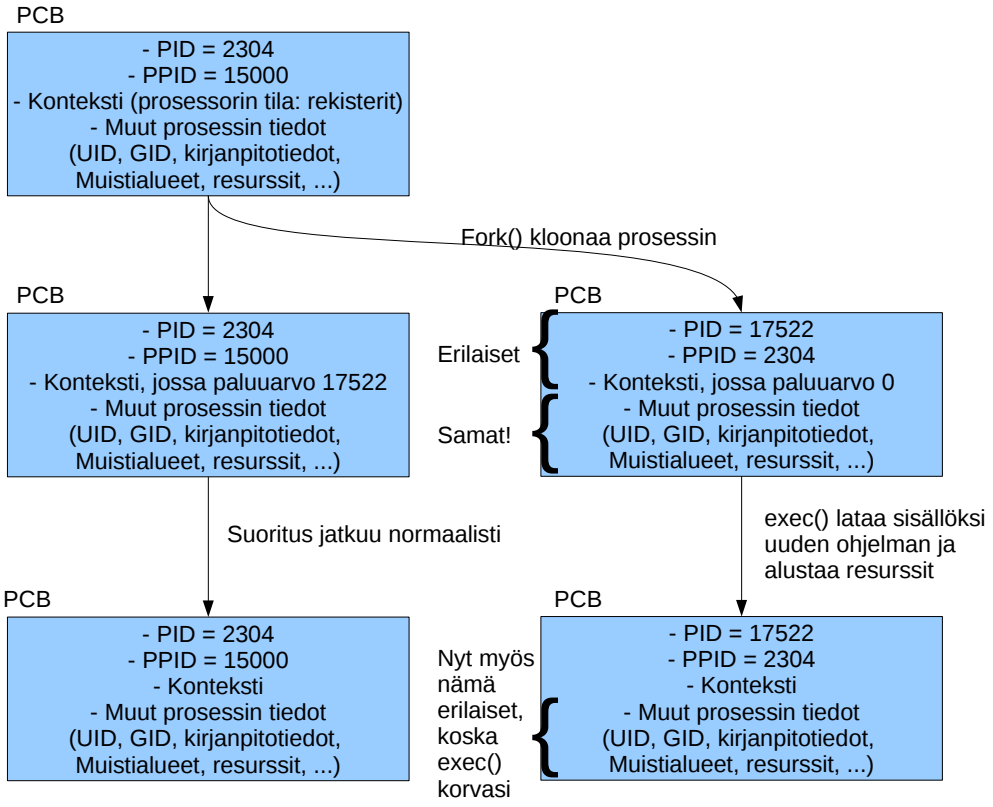
Kuva 14: *Kuva prosessien vuorontamisesta kiertojonolla (round robin). Prosesseja siirretään kiertojonoon ja sieltä pois sen mukaan, täytyykö niiden jäädä odottelemaan (eri jonoon, blocked-tilaan).*

Forkin jälkeen sekä vanhempi- että lapsiprosessi suorittavat samaa koodia samasta kohtaa, joten niiden täytyy uudelleen tunnistaa oma identiteettinsä kutsun jälkeen. Se tapahtuu fork() -kutsun paluuarvoa tulkitsemalla:

- fork() palauttaa lapsiprosessin kontekstissa nollan.
- fork() palauttaa vanhempiprosessin kontekstissa positiivisen luvun, joka on syntyneen lapsiprosessin PID.
- fork() palauttaa negatiivisen luvun, jos kloonia ei voitukaan jostain syystä luoda (ja tällöinhän on olemassa vain alkuperäinen prosessi).

Forkin käyttö esitellään vielä esimerkin kautta. Seuraavassa on "pseudo-C-kielinen" esimerkki ohjelmasta, joka lukee ohjelmatiedoston nimen argumentteineen ja pyytää käyttöjärjestelmää käynnistämään vastaavan ohjelman (eli kyseessä on "shell"-tyyppinen ohjelma). Ohjelmassa yritetään luoda lapsiprosessi forkkaamalla, ja jos se onnistuu, lapsiprosessin sisältö korvataan lataamalla sen paikalle uusi ohjelma. Lataaminen tapahtuu käyttöjärjestelmäkutsulla exec(). Siinä kohtaa käyttöjärjestelmä alustaa prosessin muistialueet (koodi, pino, data) sekä kontekstin, joten exec()-kutsua pyytänyt prosessi aloittaa uuden ohjelman suorituksen puhtaalta pöydältä. Huomattava on siis, että onnistuneen exec()-kutsun jälkeen prosessin aiempi ohjelmakoodi on unohdettu täysin, eli sen jälkeen tulevaa koodia ei tulla koskaan suorittamaan. Kannattaisi toki kirjoittaa sinne jokin käsittely tilanteelle, jossa exec() epäonnistuu esimerkiksi koska ohjelmatiedostoa ei löydy tai pysty lukemaan.

```
while(true){
  luekomento(komento, parametrit);    /* ''viiteparametrit'' saavat */
                                      /* uuden sisällön päätteeltä  */
  pid = fork();
  if (pid > 0) {      /* fork() onnistui, lapsiprosessin PID saatu. */
    status = wait(); /* odottaa että lapsiprosessi loppuu.          */
  } else if (pid == -1) {
    /* fork() epäonnistui; kenties prosessitaulu oli jo täynnä tai
     * muuta yllättävää...
     */
    exit(1)
  } else {
    /* fork() palautti 0:n, joten tässä ollaan lapsiprosessissa    */
```



Kuva 15: *Uuden prosessin luonti unixissa: käyttöjärjestelmäkutsu fork(). Uuden ohjelman lataaminen ja käynnistys edellyttää lisäksi lapsiprosessin sisällön korvaamista: käyttöjärjestelmäkutsu exec().*

```
        exec(komento, parametrit);    /*korvataan uudella ohjelmalla     */
        /* täällä pitäisi käsitellä exec()-kutsun epäonnistuminen       */
    }
}
```

Kuva 15 etenee ylhäältä alas, näyttäen onnistuneen forkin ja execin lopputuloksen, jossa käynnissä on kahtena prosessina kaksi eri ohjelmaa, joista toinen on toisen lapsi.

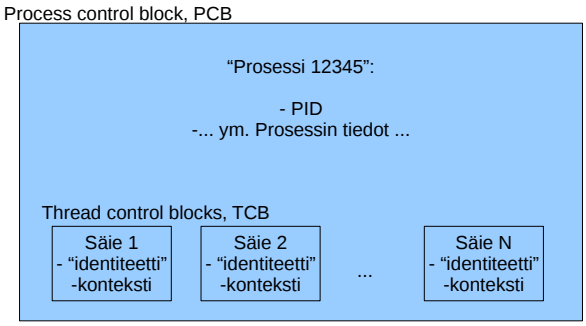
5.5 Säikeet

Yhdenaikainen suorittaminen on hyvä tapa toteuttaa käyttäjäystävällisiä ja loogisesti hyvin jäsenneltyjä ohjelmia. Mieti esim. kuvankäsittelyohjelmaa, joka laskee jotain hienoa linssivääristymäefektiä puoli minuuttia ... käyttäjä luultavasti voi haluta samaan aikaan editoida toista kuvaa eri ikkunassa, tai ladata seuraavaa kuvaa levyltä. Laskennan kuuluisi tapahtua "taustalla" samalla kun muu ohjelma kuitenkin vastaa käyttäjän pyyntöihin. Sovellusohjelman jako useisiin prosesseihin olisi yksi tapa, mutta se on tehottomampaa, esim. muistinkäytön kannalta raskasta, ja monilta osiltaan tarpeettoman monipuolista. Ratkaisu ovat **säikeet** (engl. *thread*), eli yhden prosessin suorittaminen yhdenaikaisesti useasta eri paikasta.

Muistamme, että prosessi on "muistiin ladatun ja käynnistetyn konekielisen ohjelman suoritus". Eli binääriseksi konekieleksi käännetty ohjelma ladataan käynnistettäessä tietokonelaitteistoon suoritusta varten, ja siitä tulee silloin prosessi. Nyt kun ymmärretään, miten tietokonelaitteisto suorittaa käskyjonoa, huomataan, että saman ohjelman suorittaminen useasta eri paikasta "yhtä aikaa" yhdellä prosessorilla on mahdollista – se edellyttää oikeastaan vain useampaa eri kontekstia (rekisterien tila, ml. suorituskohta, pino, liput, ...), joita vuoronnetaan sopivasti. Tällaisen nimeksi on muodostunut säie. Yhdellä prosessilla on aina yksi säie, tai sille voidaan haluta luoda useampia säikeitä.

Säikeet suorittavat prosessin ohjelmakoodia useimmiten eri paikoista (IP-rekisterin arvo huitelee eri kohdassa koodialueen osoitteita), ja eri paikkojen suoritus vuorontuu niin, että ohjelma näyttää jakautuvan rinnakkaisesti suoritettaviin osioihin, ikään kuin olisi useita rinnakkaisia prosesseja.

Säikeellä on oma:



Kuva 16: *Voidaan ajatella että säikeet (yksi tai useampia) sisältyvät prosessiin. Prosessi määrittelee koodin ja resurssit; säikeet määrittelevät yhden tai useampia rinnakkaisia suorituskohtia.*

- konteksti (rekisterit, mm. IP, SP, BP, jne..)
- suorituspino (oma itsenäinen muistialue lokaaleita muuttujia ja aliohjelma-aktivaatioita varten)
- ja tarvittava sälä säikeen ylläpitoa varten, mm. tunnistetiedot

Säie on paljon kevyempi ratkaisu kuin prosessi; sitä sanotaankin joskus **kevyeksi prosessiksi** (engl. *light-weight process*). Kun säikeessä suoritettava koodi tarvitsee prosessin resursseja, ne löytyvät prosessielementistä, joita on prosessia kohti vain yksi. Säikeillä on siis käytössään suurin osa omistajaprosessinsa ominaisuuksista:

- muistialueet
- resurssit (tiedostot, viestijonot, ym.)
- ja muut prosessikohtaiset tiedot.

Säie mahdollistaa moniajon yhden prosessin sisällä tehokkaammin kuin että olisi lapsiprosesseja, jotka kommunikoisivat keskenään. Kaikki resurssit kun ovat luonnostaan jaettuja.

Toteutustapoja:

- "User-level threads", ULT; Käyttöjärjestelmä näkee yhden vuoronnettavan asian. Prosessi itse vuorontelee säikeitään aina kun se saa käyttöjärjestelmältä ajovuoron.
  - Yksi prosessi yhdellä prosessorilla. Moniydinprosessori ei voi nopeuttaa yhden prosessin ajoa.
  - Toisaalta toimii myös käyttöjärjestelmässä, joka ei varsinaisesti ole suunniteltu tukemaan säikeitä.
  - Lisäksi säikeiden välinen vuorontaminen voidaan tehdä millä tahansa tavalla, joka ei riipu käyttöjärjestelmän vuoronnusmallista.
- "Kernel-level threads", KLT; Käyttöjärjestelmältä pyydetään säikeistys. Käyttöjärjestelmä näkee niin monta vuoronnettavaa asiaa kuin säikeitä on siltä pyydetty.
  - Moniprosessorijärjestelmissä voi kaikissa prosessoreissa ajaa eri säiettä kerrallaan. Mahdollista tehdä rinnakkaislaskennan kautta nopeammin suoritettavia prosesseja.
  - Toimii tietenkin vain käyttöjärjestelmässä, joka on suunniteltu tukemaan säikeitä vuoronnuksessa.
  - Nimeltään usein "light-weight process"

KLT-toteutuksessa käyttöjärjestelmällä voisi esimerkiksi olla tallessa PCB:n lisäksi ”säie-elementtien” (Thread Control Block, TCB) tiedot siten kuin kuvassa 16 on esitetty. TCB:tä voisi tosiaan sanoa suomeksi ”säie-elementiksi”, jollaisia sisältyy prosessikokonaisuuden PCB:hen eli prosessielementtiin yksi tai useampia.

6 Yhdenaikaisuus, prosessien kommunikointi ja synkronointi

Prosessien pitää voida kommunikoida toisilleen, esim. olisi kiva jos shellistä tai pääteyhteydestä käsin voisin pyytää jotakin toista prosessia suorittamaan lopputoimet ja sulkeutumaan nätisti. Esim. reaaliaikaisten chat-asiakasohjelmien täytyy pystyä kommunikoimaan toisilleen jopa verkkoyhteyksien yli. Jos tieteellinen laskentaohjelma ja sen käyttöliittymä toteutetaan eri prosesseina, toki käyttöliittymästä olisi kiva voida tarkistaa laskennan tilannetta ja ehkä pyytää myös välituloksia näytille... jne... eli ilmeisesti käyttöjärjestelmässä tarvitaan mekanismeja tähän tarkoitukseen, jota sanotaan **prosessien väliseksi kommunikoinniksi** (engl. *Inter-process communication, IPC*).

6.1 Tapoja, joilla prosessit voivat kommunikoida keskenään

Mainitsemme ensin nimeltä muutamia IPC-menetelmiä. Ensimmäisistä kerrotaan sitten täsmällisemmin:

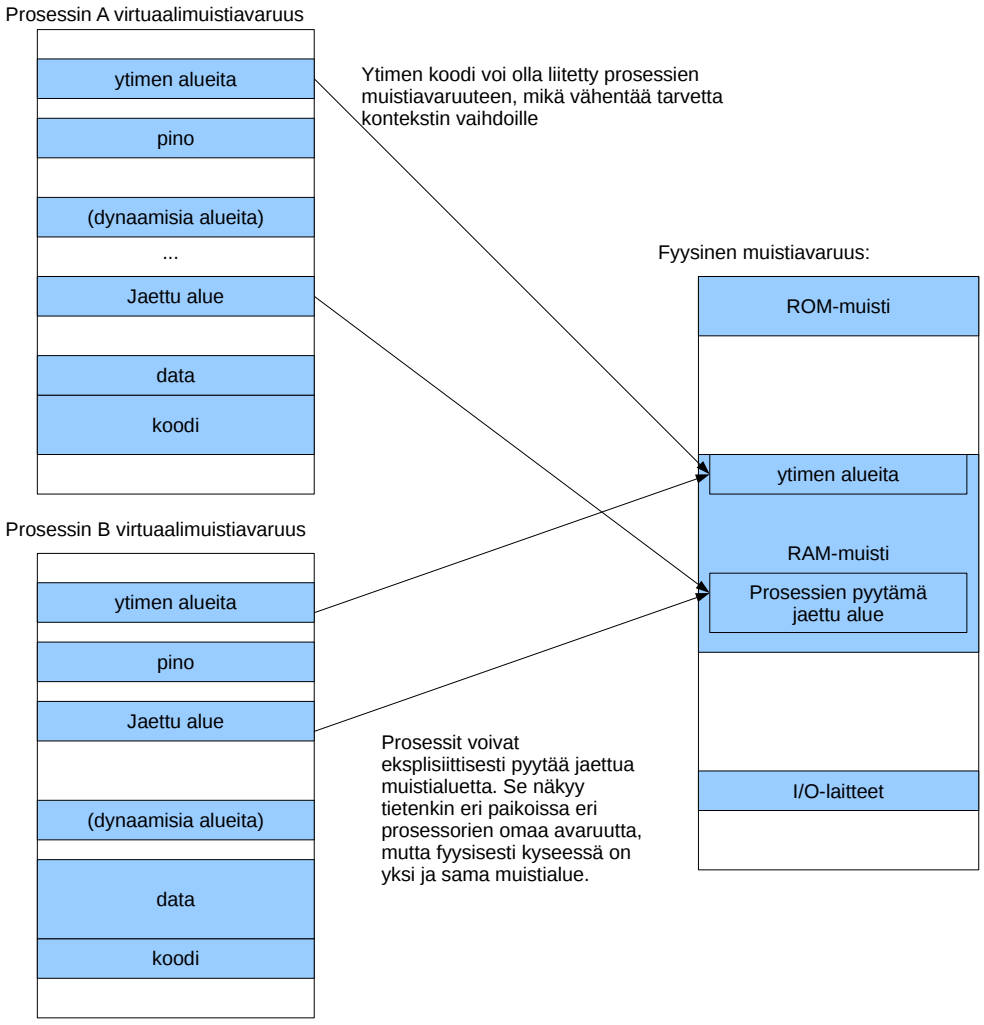
- **signaalit** (engl. *signal*) (asynkronisia eli "milloin tahansa saapuvia" ilmoituksia esim. virhetilanteista tai pyynnöistä lopettaa tai odotella; ohjelma voi valmistautua käsittelemään signaaleja rekisteröimällä käyttöjärjestelmäkutsun avulla käsittelijäaliohjelman)
- **viestit** (engl. *message*) (datapuskuri eli muistialueen sisältö, joka voidaan lähettää prosessilta toiselle viestijonoa hyödyntäen)
- **jaetut muistialueet** (engl. *shared memory*) (muistialue, jonka käyttöjärjestelmä pyydettäessä liittää osaksi kahden tai useamman prosessin virtuaalista muistiavaruutta)
- **putket** (engl. *pipe*) (putken käyttöä nähty mm. demossa, esim:: "ps -ef | grep bash | grep 'whoami' " ; käyttöjärjestelmä hoitaa putken operoinnin eli käytännössä tuottaja-kuluttaja -tilanteen hoidon; prosessi voi lukea putkesta tulevan datan standardisisääntulostaan, ja standardiulostulo voidaan yhdistää ulospäin menevään putkeen. Esim. Javassa nämä on kapseloitu olioihin System.in ja System.out
- **postilaatikko** (engl. *pipe*) (prosessi laittaa asioita "laatikkoon" ja yksi tai useampi voi käydä sieltä lukemassa)
- **portti** (engl. *port*) (postilaatikko, jossa lähettäjä tai vastaanottaja on yksikäsitteinen)
- **etäaliohjelmakutsu**, engl. *remote procedure call, RPC* (parametrit hoidetaan toisen prosessin sisältämän aliohjelman käyttöön ja paluuarvo palautetaan kutsujalle; voidaan tehdä myös verkkoyhteyden yli eli voi kutsua vaikka eri tietokoneessa olevaa aliohjelmaa).

6.1.1 Signaalit

Varmaankin yksinkertaisin prosessien kommunikointimuoto ovat signaalit. Ne ovat asynkronisia ilmoituksia, jotka ilmaisevat prosessille virhetilanteesta, yksinkertaisesta toimenpidepyynnöstä tai vastaavasta. Ohjelma voi valmistautua reagoimaan kuhunkin signaaliin omalla tavallaan. Signaalin saapuessa prosessille:

- Käyttöjärjestelmä huolehtii että seuraavan kerran kun signaalin kohteena oleva prosessi pääsee suoritusvuoroon, ei sen suoritus jatkukaan aiemmasta kohdasta vaan signaalinkäsittelijästä, jonka prosessi on rekisteröinyt.
- sovellusohjelmoijan pitää tietysti itse kirjoittaa ohjelmansa signaalinkäsittelijät sekä hoitaa tapahtumaan sellaiset käyttöjärjestelmäkutsut, joilla signaalinkäsittelijät pyydetään rekisteröimään.
- Jos ohjelma ei rekisteröi signaalikäsittelijöitä, tapahtuu tiettyjen signaalien kohdalla oletuskäsittely.

Prosessit voivat lähettää signaaleja toisilleen määrittelemällä kohdeprosessin PID:n sekä signaalin numeron. Signaaleilla on käyttöjärjestelmätoteutuksessa kiinnitetyt numerot; osa on kiinnitetty ohjelman erilaisia lopetusmenettelyjä varten, osa erilaisiin virhetilanteisiin, ja muutama on jätetty käyttäjän määriteltäväksi (eli ohjelman tekijä päättää, miten ohjelma vastaa näihin signaaleihin, ja ilmoittaa toiminnan sitten käyttöohjeissa). Unixeissa on apuohjelma "kill", jolla voi lähettää signaalin jollekin prosessille. Brutaalista nimestä huolimatta ohjelmalla voi lähettää minkä tahansa signaalin, olkoonkin että joskus joutuu viime hädässä komentamaan "kill -9", joka lähettäisi "tapposignaalin", jota ohjelma ei pysty itse poimimaan, vaan käyttöjärjestelmä lopettaa ohjelman väkivalloin (ilman että ohjelma ehtii tehdä mitään lopputoimia kuten tallentaa muuttuneita tietoja!) – täysin jumittuneelle ohjelmalle tämä voi joskus olla viimeinen vaihtoehto saada se loppumaan.



Kuva 17: *Muistialueita voidaan jakaa prosessien välillä niiden omasta pyynnöstä tai oletusarvoisesti (käyttöjärjestelmän alueet sekä dynaamisesti linkitettävät, jaetut kirjastot).*

6.1.2 Viestit

Viestit kulkevat käyttöjärjestelmän hoitamien viestiketjujen kautta, ja niihin pääsee käsiksi käyttöjärjestelmäkutsuilla, joiden nimiä voisivat olla esim seuraavat:

- msgget() pyytää käyttöjärjestelmää valmistelemaan viestijonon
- msgsnd() lähettää viestin jonoon
- msgrcv() odottaa viestiä saapuvaksi jonosta.

Viestien lähetys ja vastaanotto voivat sisältää lukitus- ja jonotuskäytäntöjä, joilla voidaan periaatteessa hoitaa samoja tehtäviä kuin seuraavassa luvussa esiteltävä semafori.

6.1.3 Jaetut muistialueet

Prosessit voivat pyytää käyttöjärjestelmää kartoittamaan fyysisen muistialueen kahden tai useamman prosessin virtuaalimuistin osaksi. Näin muistialueesta tulee prosessien jakama resurssi, johon ne molemmat voivat kirjoittaa ja josta ne voivat lukea. Tätä havainnollistetaan kuvassa 17. Kuvassa havainnollistetaan samalla aiemmin todettua seikkaa, että ytimen tarvitsemat muistialueet voidaan liittää prosessien virtuaalimuistiin, jolloin käyttöjärjestelmän ohjelmakoodiin siirtyminen ei vielä välttämättä vaadi prosessista toiseen vaihtamista.

6.2 Synkronointi: esimerkiksi kuluttaja-tuottaja -probleemi

Jaetun resurssin käyttö johtaa helposti erilaisiin ongelmatilanteisiin, jotka on jollain tavoin ratkaistava. Käytetään tässä esimerkkinä yksinkertaista tuottaja-kuluttaja -ongelmaa. Se on yksi perinteinen ongelma, joka voi syntyä käytännön sovelluksissa, ja jonka avulla voi testata synkronointimenetelmän toimivuutta:

- Yksi prosessi/säie tuottaa dataa elementti kerrallaan. Tämä voi olla hidas tai nopea toimenpide, ja dataelementin koko voi olla pieni tai suuri.
- Toinen prosessi/säie lukee ja käsittelee (="kuluttaa") tuotettua dataa elementti kerrallaan. Tämä voi olla hidas tai nopea toimenpide, erityisesti se voi olla paljon hitaampaa tai nopeampaa kuin tuottaminen, tai keskinäinen nopeus voi vaihdella.
- Tällä tavoin saavutetaan mm. modulaarisuutta ohjelmien tekemiseen, jakeluun ja suorittamiseen.
- Puolirealistinen esimerkki voisi olla että yksi prosessi/säie tuottaa kuvasarjaa fysiikkasimuloinnin perusteella (tuottamisen nopeus voi vaihdella esimerkiksi animaatiossa näkyvien esineiden määrän perusteella) ja toinen prosessi/säie pakkaa kuvat MP4-videoksi (pakkauksen nopeus voi vaihdella kuhunkin kuvaan sattuvan sisällön perusteella, esim. yksivärinen tai paikallaan pysyvä maisema menee nopeammin kuin erityisen liikkuva "kohtaus"; joka tapauksessa tuottaminen ja kuluttaminen tapahtuvat tässä oletettavasti keskimäärin eri nopeudella).
- Tietotekniikan realiteetit:
  - Datan siirtopuskuriin (muistialue, tiedosto tai muu) mahtuu vain äärellinen, ennalta päätetty määrä elementtejä.
  - moniajossa kumpikaan prosessi ei ilman erityistemppuja voi päättää vuorontamisesta; erityisesti tuottajaprosessi/-säie voi keskeytyä kun elementin kirjoittaminen on puolivalmis, ja myös kuluttaja voi keskeytyä kesken elementin lukemisen.

Mitä täytyy pystyä tekemään:

- Puskurin täyttyessä pitää pystyä odottamaan, että tilaa vapautuu. Muutoin ei ole mahdollista kirjoittaa uutta tuotosta mihinkään. Tuottajan pitää pystyä odottamaan.
- Puskurin ollessa kokonaan käsitelty, pitää pystyä odottamaan että uutta dataa ilmaantuu. Muutoin ei ole mitään kulutettavaa. Kuluttajan pitää pystyä odottamaan.
- Puskurin sisällön pitää olla koko ajan järkevä (ei puolivalmista dataa) ja myös täytyy olla järkevät osoittimet eli muistiosoitteet paikkaan, jota kirjoitetaan ja jota luetaan.

Esimerkiksi voidaan tuottaa "rengaspuskuriin"prosessien yhteisessä muistissa. Puskurin koko on kiinteä, "N kpl"elementtejä. Kun N:nnäs elementtipaikka on käsitelty, otetaan seuraavaksi taas ensimmäinen elementtipaikka. Siis muistialueen käyttö voitaisiin ajatella renkaaksi.

Puskurissa olevia tietoalkioita voidaan symboloida vaikkapa kirjaimilla::

```
|ABCDEFghijklmnopqrstuvwxyz|
      ^tuottaja tuottaa muistipaikkaan tALKU + ti
            ^kuluttaja lukee muistipaikasta kALKU + ki
```

Virtuaalimuistin hienoushan on, että sama fyysinen muistipaikka voi näkyä kahdelle eri prosessille (kommunikointi jaetun muistialueen välityksellä). Siis oletettavasti muistiosoitteiden mielessä “tALKU != kALKU“ mutta datan mielessä “tALKU[i] == kALKU[i]“. Eli tuottaja ja kuluttaja voivat olla omia prosessejaan. Ne näkevät puskurin alkavan jostain kohtaa omaa virtuaalimuistiavaruuttaan, ja niillä on oma indeksi tällä hetkellä käsittelemäänsä elementtiin. Mutta fyysinen muistiosoite on sama. (Muistinhallinnan yhteydessä tutustutaan tarkemmin ns. osoitteenmuodostukseen prosessin virtuaaliosoitteesta todelliseksi, joka menee prosessorista osoiteväylälle). Jos taas synkronointia tarvitaan saman prosessin säikeiden välille, toki kaikki muisti on jaettua säikeiden kesken, ja osoiteetkin ovat tällöin samat.

6.2.1 Semafori

Semafori (engl. *semaphore*) on käyttöjärjestelmän käsittelemä rakenne, jonka avulla voidaan hallita vuorontamista eli sitä, milloin prosessit pääsevät suoritukseen prosessorilaitteelle. Yhdessä semaforissa on arvo (”value”, kokonaisluku) ja jono prosesseista. Esimerkiksi semaforin tilanne voisi olla seuraavanlainen:

```
Arvo:   0
Jono:   PID 213 -> PID 13 -> PID 678 -> NULL
```

Semaforit pitää voida yksilöidä. Ne ovat saatavilla/käytettävissä KJ-kutsujen kautta. Semaforien luonnin ja yleisen hallinnan lisäksi käyttöjärjestelmä toteuttaa seuraavanlaisen pseudokoodin mukaiset käyttöjärjestelmäkutsut semaforin soveltamiseksi; niiden nimet voisivat olla "wait()"ja "signal()", mutta yhtä hyvin jotakin muuta vastaavaa... Kutsu on sovellusohjelmassa, ja sen parametrina on annettava yksi tietty semafori.

```
wait(Sem):

    if (Sem.Arvo > 0)
       Sem.Arvo := Sem.Arvo - 1;
    else {eli silloin kun  Sem.Arvo <= 0}
       Laita pyytäjäprosessi blocked-tilaan tämän semaforin jonoon.


signal(Sem):

   if (Jono on tyhjä)
       Sem.Arvo := Sem.Arvo + 1;
   else
       Ota jonosta seuraava odotteleva prosessi suoritukseen.
```

6.2.2 Poissulkeminen (Mutual exclusion, MUTEX)

Jos kaksi prosessia käyttää samaa jaettua resurssia jossakin ohjelmakoodin kohdassa, jonka luku- ja kirjoitusoperaatioita ei saisi päästä tekemään samanaikaisesti tai ”ristiin”, sanotaan tätä ohjelmakoodin osuutta **kriittiseksi alueeksi** (engl. *critical section*). Kriittistä aluetta saa päästä suorittamaan vain yksi prosessi kerrallaan, eli täytyy tapahtua prosessien **keskinäinen poissulkeminen** (engl. *mutual exclusion, ”MutEx”*). Oikeastaan ainoa tapa tähän on että ohjelmoija toteuttaa ohjelmaansa käyttöjärjestelmän palveluiden kutsumisen esimerkiksi seuraavasti::

```
    wait(semMunMutexi)   // "atominen käsittely",
                         // käyttäjän prosessit keskeytettynä.

    ... kriittinen alue, yksinoikeus ...

    signal(semMunMutexi) // "atominen käsittely"
```

Käydään läpi esimerkki, jossa on useita prosesseja, sanotaan vaikkapa PID:t 77, 123, 341 sekä 898, jotka suorittavat ylläolevan kaltaista koodia. Semafori “semMunMutexi“ on tietenkin sama yksilö ja kaikkien prosessien tiedossa. Alkutilanteessa semafori on “vapaa“:

```
semMunMutexi.Arvo:   1
semMunMutexi.Jono:   NULL
```

PID 77:n koodia suoritetaan, siellä on kutsu wait(semMunMutexi). Tapahtuu ohjelmallinen keskeytys, jolloin prosessi PID 77 siirtyy kernel running -tilaan, ja käyttöjärjestelmän koodista suoritetaan semaforin käsittely wait(). Ks. pseudokoodi yllä. Tässä tapauksessa seuraavaksi tilanne on:

```
semMunMutexi.Arvo:   0
semMunMutexi.Jono:   NULL
```

Käyttöjärjestelmästä palataan PID 77:n koodin suorittamiseen heti wait()-kutsun jälkeisestä käskystä (prosessia ei tarvinnut vaihtaa). Nyt PID 77:llä on yksinoikeus suorittaa semMunMutexi-semaforilla merkittyä kriittistä aluetta, koska semaforin arvosta 0 voi todeta jonkun prosessin olevan kriittisellä alueella.

Sitten esim. PID 898 tulisi jossain vaiheessa vuoronnetuksi suoritukseen ennen kuin PID 77 olisi valmis kriittisen alueen suorituksessa. Sitten PID 898:n koodi lähestyisi kriittistä aluetta, jossa sekin kutsuisi wait(semMunMutexi).

Jälleen tietenkin tulisi ohjelmallinen keskeytys, prosessi PID 898 menisi kernel running -tilaan, ja käyttöjärjestelmän koodista suoritettaisiin semaforin käsittely wait(). Tässä tapauksessa, kun semaforin arvo on 0, aiheutuukin seuraavanlainen tilanne:

```
semMunMutexi.Arvo:    0
semMunMutexi.Jono:    PID 898 -> NULL
```

Käyttöjärjestelmä siis siirtäisi prosessin PID 898 Blocked-tilaan, ja liittäisi sen semMunMutexin jonoon odottamaan myöhempää signal()-kutsua. Tämä (kuten ylipäätään käyttöjärjestelmäkutsu aina) tapahtuu sovellusohjelmien kannalta "**atomisesti**" (vai "atomaarisesti") (engl. *atomic operation*) eli mikään käyttäjän prosessi ei pääse suorittumaan ennen kuin käyttöjärjestelmä on tehnyt vaadittavat organisointi- ja kirjanpitotyöt.

Useilla prosesseilla voisi olla erilaisia toimenpiteitä semMunMutexilla suojattuun jaettuun resurssiin. Vuorontaja jakelisi prosesseille aikaa ja kaikki tapahtuisi nykyprosessorissa kovin nopeasti. Semafori kuitenkin on jo lukinnut alueen ensimmäiseksi ehtineen prosessin käyttöön, joten jossain vaiheessa tilanne voisi siis olla esimerkiksi seuraava:

```
semMunMutexi.Arvo:    0
semMunMutexi.Jono:    PID 898 -> PID 341 -> PID 123 -> NULL
```

Jonoon on kertynyt prosesseja. PID 77, joka ehti kutsumaan wait(semMunMutexi) ensimmäisenä, saa lopulta operaationsa valmiiksi jollakin ajovuorollaan, ja jos se on oikeellisesti ohjelmoitu, niin kriittisen alueen lopussa on kutsu “signal(semMunMutexi)“. Jälleen käyttöjärjestelmä atomisesti hoitaa tilanteeksi:

```
semMunMutexi.Arvo:    0
semMunMutexi.Jono:    PID 341 -> PID 123 -> NULL
```

PID 898 on siirretty blocked tilasta ready-tilaan, ja se on siirretty semMunMutexin jonosta vuorontajan ready-jonoon. (Tai, sekoittaaksemme päitämme, se voitaisiin ottaa suoraan suoritukseen, jos vuoronnus ja semaforit olisivat sillä tavoin toteutetut...) Semaforin arvo pysyy kuitenkin yhä 0:na, mikä tarkoittaa, että resurssi ei vielä ole vapaa. Siis joku suorittaa kriittistä aluetta, ja mahdollisesti sinne on jo jonoakin päässyt kertymään. Vasta, jos uusia jonottajia ei ole wait() -kutsun kautta tullut, ja aiemmat prosessit ovat yksi kerrallaan suorittaneet kriittisen alueensa ja kutsuneet signal(), niin aivan viimeinen signal() tapahtuu tietysti seuraavanlaisessa tilanteessa:

```
semMunMutexi.Arvo:    0
semMunMutexi.Jono:    NULL
```

Ja signalin jälkeen resurssi vapautuu täysin, sillä sittenhän tilanne on sama kuin aivan esimerkin alussa:

```
semMunMutexi.Arvo:    1
semMunMutexi.Jono:    NULL
```

6.2.3 Tuottaja-kuluttaja -probleemin ratkaisu

Tuottaja-kuluttaja -ongelma eli kahden prosessin välinen tietovirran synkronointi voidaan ratkaista semaforeilla seuraavaksi esitetyllä tavalla. Toinen perinteinen, erilainen ongelma-asettelu on "kirjoittajien ja lukijoiden"ongelma, jossa voi olla useita kirjoittajia ja/tai useita lukijoita (tuottaja-kuluttajassa tasan yksi kumpaistakin). Lisäksi on muita perinteisiä esimerkkiongelmia, ja todellisten ohjelmien tekemisessä jokainen yhdenaikaisuutta hyödyntävä sovellus saattaa tarjota uusia vastaavia tai erilaisia ongelmia, jotka on ratkaistava että ohjelma toimisi joka tilanteessa oikeellisesti. Myös ratkaisutapoja on muitakin kuin semaforit. Yksinkertaisuuden vuoksi Käyttöjärjestelmät -kurssilla käydään läpi vain yksi yksinkertainen ongelmatapaus ja yksi yksinkertainen ratkaisu siihen.

Tarvittavat semaforit:

```
MUTEX   (binäärinen)
EMPTY   (moniarvoinen)
FULL    (moniarvoinen)
```

Ohjelmoijan on muistettava näiden oikeellinen käyttö. Aluksi alustetaan semaforit seuraavasti:

```
EMPTY.Arvo := puskurin koko     // kertoo vapaiden paikkojen määrän

FULL.Arvo  := 0                 // kertoo täytettyjen paikkojen määrän
```

```
MUTEX.Arvo := 1                    // vielä ei tietysti kellään ole lukkoa
                                   // kriittiselle alueelle...
```

Tuottajan idea:

```
WHILE(1) // tuotetaan loputtomiin
    tuota()
    wait(EMPTY)      // esim. jos EMPTY.Arvo == 38 -> 37
                     // jos taas EMPTY.Arvo == 0 {eli puskurissa ei tilaa}
                     // niin blockataan prosessi siksi kunnes tilaa
                     // vapautuu vähintään yhdelle elementille.

    wait(MUTEX)      // poissulku binäärisellä semaforilla; ks. edell. esim
    Siirrä tuotettu data puskuriin (vaikkapa megatavu tai muuta hurjaa)
    signal(MUTEX)

    signal(FULL)     // esim. jos kuluttaja ei ole odottamassa FULLia
                     // ja FULL.Arvo == 16 niin FULL.Arvo := 17
                     // (eli kerrotaan vaan että puskuria on nyt
                     // täytetty lisää yhden pykälän verran)

                     // tai jos kuluttaja on odottamassa {silloin aina
                     // FULL.Arvo == 0} niin kuluttaja herättyy
                     // blocked-tilasta valmiiksi lukemaan.

                     // ... jolloin FULLin jono tyhjenee. Eli vuoronnuksesta
                     // riippuen tuottaja voi ehtiä monta kertaa suoritukseen
                     // ennen kuluttajaa, ja silloin se ehtii kutsua
                     // signal(FULL) monta kertaa, ja FULL.Arvo voi olla
                     // mitä vaan >= 0 siinä vaiheessa, kun kuluttaja
                     // pääsee apajille.
```

Kuluttajan idea:

```
WHILE(1)
    wait(FULL) // onko luettavaa vai pitääkö odotella,
               // esim. FULL.Arvo == 14 -> 13
               // tai esim. FULL.Arvo == 0 jolloin kuluttaja blocked
               // ja jonottamaan

    // tänne päädytään siis joko heti tai jonotuksen kautta (ehkä
    // vasta viikon päästä...) jahka tuottaja suorittaa signal(FULL)

    wait(MUTEX)  // tämä taas selvä jo edellisestä esimerkistä.
    käsitellään tietoalkio puskurista
    signal(MUTEX)

    signal(EMPTY) // Esim. jos EMPTY.Arvo == 37 ja tuottaja ei ole
                  // odottamassa, niin EMPTY.Arvo := 38
                  //
                  // Tai sitten tuottaja on jonossa
                  // {jolloin EMPTY.Arvo == 0}, missä tapauksessa ihan
                  // normaalisti semaforin toteutuksen mukaisesti
                  // tuottaja pääsee blocked-tilasta ja EMPTYn jonosta
                  // ready-tilaan ja taas valmiiksi suoritukseen.
```

Huomautuksia

Edellä oli pari esimerkkiä, mutta asian ymmärtäminen vaatii oletettavasti enemmän kuin vain esimerkkien läpiluvun. Mieti tarkoin, miten semafori toimii kussakin erityistilanteessa käyttöjärjestelmäkutsujen kohdalla, kunnes koet, että ymmärrät, miten ongelma tässä ratkeaa (ja tietenkin että mikä se ongelma lähtökohtaisesti olikaan).

Tässä oli ratkaisu kahteen pulmaan: resurssin johdonmukaiseen käyttöön poissulkemisen (Mutual exclusion, "MutEx") kautta, ja tasan kahden prosessin tai säikeen yksisuuntaiseen puskuroituun tietovirtaan eli tuottaja-kuluttaja -tilanteeseen. Todelliset IPC-ongelmat voivat olla tällaisia, mutta ne voivat olla monimutkaisempiakin: voi olla useita "tuottajia", useita "kuluttajia", useita eri puskureita ja useita sovellukseen liittyviä toimintoja. Olet nähnyt yksinkertaisia perusperusteita, joista toivottavasti syntyy jonkinlainen pohja ymmärtää monimutkaisempia tilanteita myöhemmin, jos joskus tarvitsee.

Tässä näimme semaforiperiaatteen, joka on yksi usein käytetty tapa ratkaista tässä nähdyt perusongelmat. Ota huomioon, että on myös muita tapoja näiden sekä monimutkaisempien ongelmien ratkaisemiseen. (Jälleen, tämä on yksinkertainen ensijohdanto kuten kaikki muukin Käyttöjärjestelmät -kurssilla). Muita tapoja on ainakin viestinvälitys ("send()"ja "receive()") sekä ns. "monitorit", jotka jätetään tässä maininnan tasolle.

6.3 Deadlock

Kuvaelma nimeltä "Ruokailevat nörtit": Pöydässä on Essi ja Jopi, joilla kummallakin on edessään ruokaa, mutta pöydässä on vain yksi haarukka ja yksi veitsi. Paikalla on myös Ossi, joka valvoo ruokailun toimintaa seuraavasti:

- Essi ja Jopi eivät saa tehdä yhtäaikaa mitään, vaan kukin vuorollaan, pikku hetki kerrallaan (vähän niin kuin Pros-essit tai "jobit"käyttöjärjestelmän eli OS:n vuorontamina).
- Veistä kuin myös haarukkaa voi käyttää vain yksi henkilö kerrallaan. Muiden täytyy jonottaa resurssin käyttövuoroa.

Jopi aikoo syödä omalla ruokailualgoritmillaan:

```
1. Varaa haarukka
2. Varaa veitsi
3. Syö ruoka
4. Vapauta veitsi
5. Vapauta haarukka
```

Essi puolestaan aikoo syödä omalla ruokailualgoritmillaan:

```
1. Varaa veitsi
2. Varaa haarukka
3. Syö ruoka
4. Vapauta haarukka
5. Vapauta veitsi
```

Kaikki menee hyvin, jos Essi tai Jopi ehtii varata sekä haarukan että veitsen ennen kuin Ossi keskeyttää ja antaa vuoron toiselle ruokailevalle nörtille. Mutta huonosti käy, jos...

```
1: Jopi varaa haarukan
2: Ossi siirtää vuoron Essille; Jopi jää odottamaan suoritusvuoroaan
3: Essi varaa veitsen
4: Essi yrittää varata haarukan
5: Ossi laittaa Essin jonottamaan haarukkaa, joka on jo Jopilla.
   Sitten Ossi antaa vuoron Jopille, joka on valmiina jatkamaan.
6: Jopi yrittää varata veitsen
7: Ossi laittaa Jopin jonottamaan veistä, joka on jo Essillä.
8: Sekä Jopi että Essi jonottavat resurssin vapautumista ja
   nääntyvät oikein kunnolla.
```

Mitään ei enää tapahdu; ruokailijat ovat ns. **deadlock** -tilanteessa eli odottavat toistensa toimenpiteiden valmistumista. Algoritmeja on säädettävä esim. ruokailu_MutEx -semaforin avulla:

Jopi:

```
1. Wait(ruokailu_MutEx) -- Ossi hoitaa yksinoikeuden
2. Varaa haarukka
3. Varaa veitsi
4. Syö ruoka
5. Vapauta veitsi
6. Vapauta haarukka
7. Signal(ruokailu_MutEx) -- Ossi hoitaa
```

Essi:

1. Wait(ruokailu_MutEx) -- Ossi hoitaa yksinoikeuden
2. Varaa veitsi
3. Varaa haarukka
4. Syö ruoka
5. Vapauta haarukka
6. Vapauta veitsi
7. Signal(ruokailu_MutEx) -- Ossi hoitaa

Nyt kun Jopi tai Essi ensimmäisenä varaa poissulkusemaforin, toinen ruokailija päätyy Ossin hoitamaan jonotukseen, kunnes ensimmäinen varaaja on ruokaillut kokonaan ja ilmoittanut lopettaneensa. Ossi päästää seuraavan jonottajan ruokailemaan eikä lukkiutumista tapahdu.

Vaarana on enää, että vain joko Jopi tai Essi joutuu hetken aikaa nääntymään nälkään, kunnes toinen on syönyt loppuun ja vapauttanut ruokailu_MUTEXin. Väliaikainen **nälkiintyminen** (engl. *starvation*) on siis kevyempi muoto lopullisesta **lukkiutumisesta**, joka on toinen, suomenkielisempi, sana deadlock-tilanteelle.

7 Muistinhallinta

7.1 Muistilaitteistosta: muistihierarkia, prosessorin välimuistit

Aiemmin on nähty prosessorin toimintaa. Havaittiin, että tarvitaan ns. rekisterejä, jotka ovat nopeita muistikomponentteja prosessorin sisällä, mahdollisimman lähellä laskentaa hoitavia komponentteja. Rekisterien määrää rajoittaa kovasti hinta, joka niiden suunnitteluun ja toteutukseen kuluu. Muistia tarvitaan tietokoneessa kuitenkin paljon, koska tietojenkäsittelyn tehtävät tarvitsevat lyhyt- ja pitkäaikaista tietojen tallennusta. Nykyisin keskusmuistiin mahtuu varsin paljon tietoa, mutta se on "kaukana" ulkoisen väylän päässä, joten sen käyttö on maailmallisista syistä johtuen hidasta. Kompromissinä prosessoreihin valmistetaan ns. **välimuisteja** (engl. *cache memory*), eli nopeampia (ja samalla hintavampia) muistikomponentteja, jotka sijaitsevat lähempänä prosessoria ja joissa pidetään väliaikaisesti osaa keskusmuistin sisällöstä. Tämä on järkevää, koska tavallisesti ohjelmat käyttävät enimmäkseen lähellä toisiaan olevia muistiosoitteita. Tälle havainnolle on nimi, **lokaalisuusperiaate** (engl. *principle of locality*). Siis suurin osa viittauksista ohjelman virtuaalimuistiin tapahtuu lähelle äskettäin viitattuja muistipaikkoja (tämä voidaan käytännön tasolla hyvin ymmärtää: Ajattele esim. koodin osalta peräkkäin suoritettavia käskyjä, muutamien käskyjen mittaisia silmukoita ja usein toistettavia aliohjelmia. Pinon osalta yleensäkin käytetään muutamia peräkkäin sijoittuvia aktivaatiotietueita. Datan osalta käsitellään suhteellisen pitkään tiettyä oliota/tietorakennetta ennen siirtymistä seuraavan käsittelyyn).

Rekisterejä voi siis olla vain muutama. Välimuistit maksavat enemmän kuin keskusmuisti, mutta nopeuttavat kokonaisuuden toimintaa. Prosessori hoitaa välimuistien toiminnan automaattisesti. Sovellusohjelmien tekijän ei tarvitse huolehtia siitä, ovatko virtuaalimuistiosoitteiden osoittamat tiedot keskus- vai välimuistissa. Välimuistien olemassaolo ja rajallinen koko on kuitenkin ymmärrettävä tiettyjä laskenta-algoritmeja tehdessä. Ohjelmat toimivat todella paljon nopeammin, jos suurin osa tiedoista todella löytyy välimuistista. Siis kannattaa tehdä algoritmit siten, että käsitellään dataa mahdollisuuksien mukaan pieni lohko kerrallaan ennen siirtymistä seuraavaan – eli hyppimättä kaukana toisistaan olevien muistiosoitteiden välillä.

Ns. massamuistia kuten kovalevytilaa on käytettävissä käytännön tarpeisiin lähes rajattomasti ilman äärimmäisiä kustannuksia. Voidaan puhua ns. **muistihierarkiasta** (engl. *memory hierarchy*), jossa muistikomponentit listataan nopeasta hitaaseen, samalla kalliista halpaan, seuraavasti:

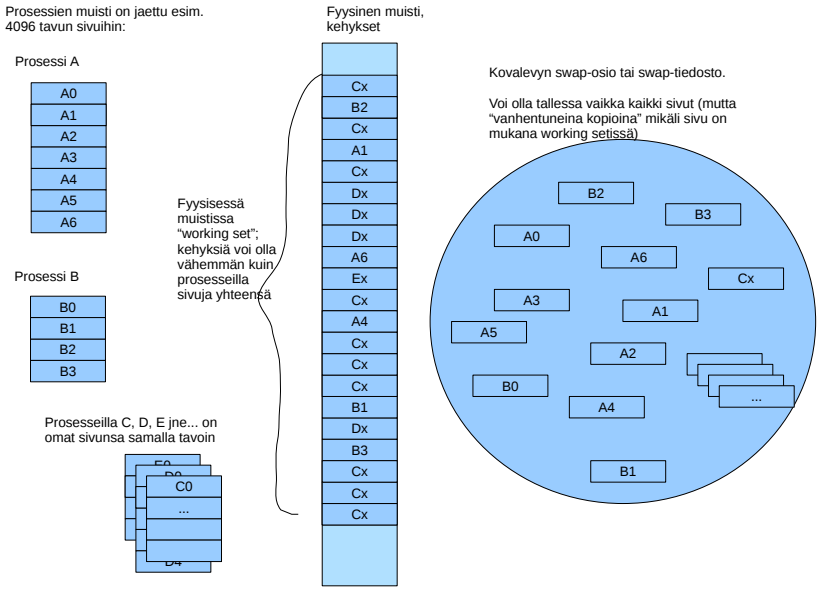
- Rekisterit
- Välimuistit (Level 1, Level 2; prosessoriteknologia hoitaa välimuistin käytön; ohjelma näkee virtuaalimuistiavaruuden)
- Keskusmuisti
- Massamuistit kuten kovalevyt

Koska on mahdotonta saavuttaa täydellistä tilannetta, jossa kaikki muisti olisi prosessorin välittömässä läheisyydessä, tarvitaan suunnittelussa kompromissiratkaisuja. Kalliita ja nopeita rekisterejä suunnitellaan järjestelmään muutamia, Level 1:n välimuistia jonkin verran ja Level 2 (ja ehkä Level 3) -välimuistia vielä vähän enemmän. Suunnittelu- ja tuotantokustannukset mutta samalla muistien käytön nopeus putoavat sitä mukaa kuin etäisyys prosessoriin kasvaa. Keskusmuistia on nykyään aika paljon, mutta ohjelmatkin tuppaavat kehittymään siihen suuntaan että niiden uudemmat ja hienommat versiot lopulta käyttävät niin paljon keskusmuistia kuin kulloisellakin aikakaudella on saatavilla. Prosesseja halutaan siis tyypillisesti suorittaa enemmän kuin keskusmuistiin mahtuu prosessien tarvitsemaa virtuaalimuistia. Massamuistit ovat äärimmäisen suuria ja halpoja, mutta myös keskusmuistiin nähden äärimmäisen hitaita. Tästä seuraa tarve jollekin fiksulle järjestelmälle, joka hyödyntää hidasta levytilaa pienen keskusmuistin apuna.

7.2 Sivuttava virtuaalimuisti

Lokaalisuusperiaatetta hyödyntävä sivuttavan virtuaalimuistin perusidea on esitetty kuvassa 18.

- Kunkin prosessin tarvitsema virtuaalimuisti jaetaan ns. **sivuihin** (engl. *page*), jotka ovat tyypillisesti esim. 4096 tavun mittaisia.
- Fyysinen muisti jaetaan sivun kokoisiin **kehyksiin** (engl. *page frame*), joissa kussakin voidaan säilyttää yhtä sivua jonkin prosessin "näinä aikoina" tarvitsemaa muistialuetta.
- Muistissa pidetään vain äskettäin tai "näinä aikoina" käytettyä koodia ja dataa, **työjoukkoa** (engl. *working set*), joka koostuu sivuista.



Kuva 18: *Sivuttavan virtuaalimuistin perusidea..*

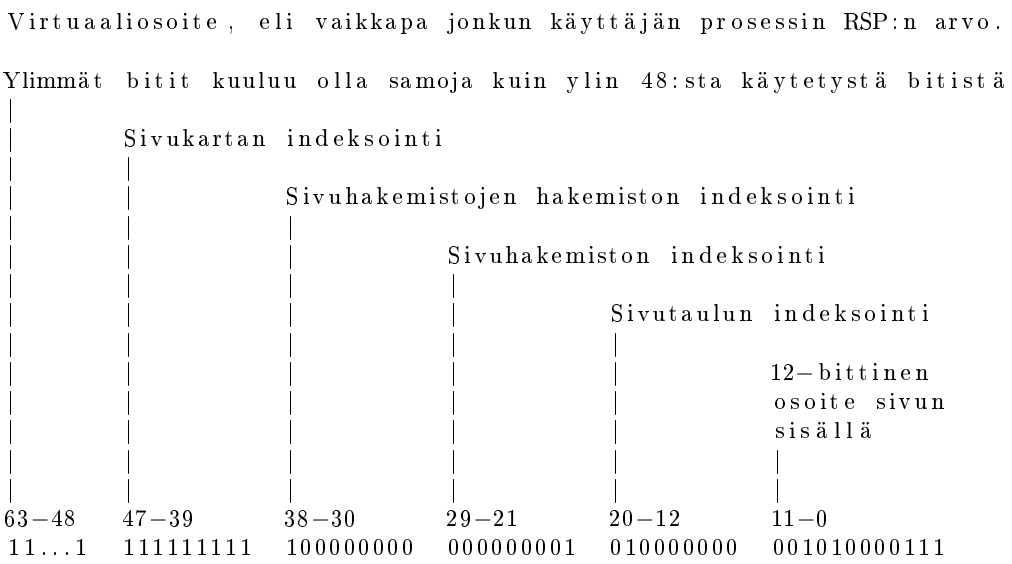
- Loput sivut voivat odotella levyllä ”jäädytettynä”.
- ”Suspended” -tilassa olevien prosessien kaikki sivut voi heittää levylle, koska niitä ei tällä hetkellä ylipäätään suoriteta.

Jotta tällaista sivutussysteemiä voidaan käyttää, tarvitaan tietyt ominaisuudet prosessorilta ja käyttöjärjestelmältä. Ensinnäkin käyttöjärjestelmän täytyy pitää yllä seuraavia tietorakenteita (perinteinen perusidea):

- **Sivutaulu** (engl. *page table*) jokaista prosessia kohden. Sivutaulussa on seuraavat tiedot jokaista prosessin sivua kohden:
  - ”muistibitti” eli onko sivu keskusmuistissa
  - fyysisen sivun numero, jos sivu on keskusmuistissa
  - sijainti levyllä (jokaisesta sivusta on levyllä tallessa kopio)
- Yksi **kehystaulu** (engl. *frame table*) jossa on tiedot jokaista keskusmuistin kehystä eli sivun fyysistä lokeroa kohden:
  - minkä prosessin käytössä tämän kehyksen sisältämä sivu on, ja mitä prosessin virtuaalimuistin sivua se vastaa
  - ”kirjoitusbitti” (engl. *dirty bit / modified bit*): Prosessori asettaa tämän, jos muistiin kirjoitetaan tälle sivulle; sivusta tulee ”likainen” siinä mielessä että muistissa oleva versio ei enää ole sama kuin levyllä tallessa oleva ”jäädytetty” sivu.
  - ”seinäkelloaika”: Prosessori päivittää ajan, kun sivua luetaan/kirjoitetaan
  - suojaustiedot: Esim. saako sivulle kirjoittaa, saako sieltä noutaa konekäskyjä suoritukseen (”no execute”-bitti); näillä voi jonkin verran rajoittaa perinteisiä tietomurtotapoja.

Tällaisten taulujen käyttö edellyttää laitteistolta joitakin melko hienostuneita ominaisuuksia (joita on pitänyt prosessoritekniikkaan kehitellä aikojen varrella, muistihierarkian luomiseksi ja entistä tehokkaamman tietotekniikan mahdollistamiseksi):

- Osoitteenmuodostus aina taulujen avulla, joita itse prosessori osaa käsitellä!!
- **Sivunvaihtokeskeytys**, toiselta nimeltään ”**sivuvirhe**” (engl. *page fault exception*), jolla käyttöjärjestelmä pääsee lataamaan ja tallentamaan sivuja levyltä/levylle.



Kuva 19: *Nelitasoinen osoitteenmuodostus AMD64-prosessorissa (eräs x86-64). Arkkitehtuuri tukee muitakin sivukokoja, mutta tässä on esimerkki 4096 tavun (2^{12}) eli neljän kilotavun kokoisten sivujen käytöstä.*

Sivuvirhe on käsitteellisesti aivan normaali keskeytys: prosessi, jonka koodissa normaali osoitteenmuodostus johtaa sivulle, joka ei olekaan fyysisessä muistissa, keskeytetään, ja prosessori alkaa suorittaa käyttöjärjestelmän muistinhallinta -osion koodia. Sivun sisältö pitää silloin ladata fyysisen muistin vapaaseen kehykseen. Käyttöjärjestelmän ylläpitämässä kehystaulussa on operaatioon tarvittavat tiedot. Käyttöjärjestelmän täytyy silloin huolehtia seuraavista toimenpiteistä:

- Jos vapaata kehystä ei ole, pitää ensin valita joku käytössä olevista ja vaihtaa (engl. *swap*) sen sisältämä sivu puolestaan levylle jemmaan.
- Esim. **LRU, least-recently-used**, eli käyttöjärjestelmän muistinhallintakomponentti heittää käyttöajankohdan mukaan kauimmin käyttämättä olleen sivun levylle ja lataa sen tilalle uuden. Levyosoite löytyy sen prosessin sivutaulusta, joka aiheutti sivuvirheen.
- Kehystaulun tiedot tietysti on päivitettävä vastaavasti. Ja jos jotain heitettiin levylle pois fyysisestä muistista, on kyseisen prosessin sivutaulua myös muutettava.

Muistinhallintaan siis liittyy jopa hieman mutkikastakin logiikkaa, joka käyttöjärjestelmän on hoidettava. Tämä on välttämätöntä, jotta muistin osalta rajallisella tietokonelaitteistolla voidaan suorittaa riittävä määrä prosesseja yhtäaikaa.

7.3 Esimerkki: x86-64:n nelitasoinen sivutaulusto

Nykypäivän prosessoreissa virtuaalimuistiavaruus on laaja, ja osoitteenmuodostus voi tapahtua monitasoisen taulukkohierarkian kautta. Esimerkiksi x86-64:ssa tapahtu nelitasoinen osoitteenmuodostus, joka esitetään kuvassa 19. 64-bittisessä muistiosoitteessa on nykyisellään 48 bittiä käytössä. 9 bitin mittaiset pätkät ovat indeksejä, joilla löytyy aina seuraavan tason taulukko ja lopulta sivutaulusta fyysisen kehyksen osoite keskusmuistissa. Taulukot sijaitsevat keskusmuistissa ja prosessori toimintansa nopeuttamiseksi lataa niitä myös välimuistiin ja käyttää erityisiä teknologioita (mainittakoon nimeltä TLB (engl. *translation look-aside buffer*)) osoitteenmuodostuksessa. Moniprosessoreissa tämä asettaa tiettyjä synkronointihaasteita: Jos prosessori käyttää jotakin sivua ja päivittää sivun tietoja, päivitys tapahtuu luonnollisesti kyseisen prosessorin välimuistissa. Kuinka muut prosessorit saadaan tietoisiksi tästä muutoksesta, jos niissä on jokaisessa oma välimuisti... todetaan, että prosessorien manuaaleissa on nykyään sivukaupalla tekstiä asiasta. Synkronointi on hoidettavissa, mutta vaatii käyttöjärjestelmän tekijältä huolellisuutta ja tietoa prosessoriin tarkoitusta varten rakennetuista ominaisuuksista.

8 Oheislaitteiden ohjaus

8.1 Laitteiston piirteitä

Aloitetaan muutamilla havainnoilla I/O (input/output) eli syöttö- ja tulostuslaitteista:

- Ne on liitetty prosessoriin ja muistiin väylän kautta.
- Laitteiden toimintajakso erilainen kuin tietokoneen – jopa satunnainen (esim. käyttäjän klikkailut ja näppäinpainallukset).
- Monenlaisia laitteita eri tarkoituksiin (kategorisoitavissa esim. tiedonsiirtotavan mukaan merkki- / lohkolaitteisiin; näppäimistö on merkkilaite, koska sieltä saapuu dataa yksi kerrallaan; kovalevy on lohkolaite, koska siellä luonnostaan on peräkkäin paljon dataa, jonka voidaan ajatella koostuvan tietynmittaisista lohkoista).
- I/O -laitteet ovat alttiita häiriöille (mekaaniset komponentit alttiimpia kuin esim. prosessorin elektroniset komponentit; lisäksi I/O-laitteet usein liitäntäjohdon päässä, mistä aiheutuu mm. johdon yllättävän irtoamisen vaara). Niiden tarkkailu ja virheenkorjaus ovat näin ollen tarpeen.

Väylän päässä ovat itse asiassa **laiteohjaimet** (engl. *device controller*) tai **sovittimet** (engl. *adapter*), laajemmissa järjestelmissä myös "**kanavat**" (engl. *channel*). Laiteohjaimessa on jokin **kontrollilogiikka**, ikään kuin minitietokone, jonka avulla ohjain kommunikoi yhdelle tai useammalle fyysiselle laitteelle. Prosessorilta voi antaa laiteohjaimen kontrollilogiikalle väylän kautta komentoja, jotka ovat numeeriseksi koodattuja toimenpidepyyntöjä sekä näiden parametreja. Riippuu toki laitteen suunnittelusta, kuinka korkean tai matalan tason operaatioita siltä voidaan pyytää, ja paljonko taas jää toteutettavaksi ohjelmallisesti. Esimerkiksi joissain ääniohjaimissa on mukana mikseri tai syntetisaattori, kun taas joissain äänisignaali on laskettava ohjelmallisesti CPU:n toimesta ja lähetettävä lopullisena ääniohjaimelle. Tyypillisesti I/O -laitteelle annettavat komennot ovat pyyntöjä kirjoittaa tai lukea jotakin ("merkkilaitteissa" tavu kerrallaan, "lohkolaitteissa" lohko kerrallaan).

Suuremman datamäärän siirron tekee usein väylään liitetty **DMA**-järjestelmä (engl. *Direct memory access*), joka osaa käyttää väylää bittijonon kopioimiseksi muistin ja I/O-laitteiden välillä puskurista toiseen. DMA on näppärä, huolimatta "kellojaksovarkauksista" (cycle stealing) operoinnin aikana, eli siitä että DMA käyttää väylää siirtoon eikä prosessori välttämättä pääse käyttämään muistia väylän kautta aina tarvitessaan. Prosessorin välimuistit luonnollisesti auttavat pienentämään ulkoisen väylän ruuhkaa.

Idea I/O:ssa ohjelmiston ja prosessorin kannalta on:

- käyttöjärjestelmän I/O:ta hoitava ohjelmakoodi antaa I/O -laitteen kontrollilogiikalle käskyn (tyypillisesti lue/kirjoita + lähteen ja kohteen tiedot)
- koska fyysisen I/O:n kestoa ei tiedetä, I/O:ta tarvinneen ohjelman prosessi kannattanee laittaa odottelemaan (blocked-tilaan) ja päästää jokin muu prosessi suoritusvuoroon.
- I/O -laite tekee fyysisen toimenpiteensä
- Valmistuttuaan I/O -laite antaa prosessorille keskeytyksen, joten käyttöjärjestelmän I/O -koodi voi jatkaa tarvittavilla toimenpiteillä (eli mm. siirrellä odottavan prosessin taas blocked-tilasta suoritusvalmiiksi ja valmistella sille tiedon operaation onnistumisesta)

8.2 Kovalevyn rakenne

Tutustutaan kovalevyn rakenteeseen toisaalta yhtenä käytännön esimerkkinä I/O-laitteesta ja toisaalta valmisteluna tiedon tallentamiseen liittyvän järjestelmämoduulin esittelyyn. Kovalevyn toiminta perustuu magnetoituvaan kalvoon, jonka pienen alueen magneettikentän suunta tulkitaan bittinä. Pyörivän kalvon kulkiessa luku/kirjoituspään ohi voidaan bitit lukea (tulkita bitit kalvon alueista) tai kirjoittaa (vaihtaa kentän suuntaa).

Kovalevyn rakenteeseen liittyvät **ura** (engl. *track*), eli yksi ympyräkehän muotoinen jono bittejä pyörivällä kiekolla, **sektori** (engl. *sector*), joka on tietynmittainen peräkkäisten bittien pätkä yhdellä uralla, ja **sylinteri** (engl. *cylinder*), joka koostuu pakassa päällekkäin pyörivien levyjen päällekkäisistä kohdista. Huomioita:

- Sektori on pienin kerrallaan kirjoitettava tai luettava alue (esim. 512 tavua – valmistajasta riippuen)

- Sektori sisältää tarkistetietoa virheiden havaitsemista ja korjaamista varten
- Fyysisen levyn osoitteet ovat sektorikohtaisia
- Tyypillisesti ohjelmisto (esim. käyttöjärjestelmän tiedostonhallinta) niputtaa muutamia sektoreita lohkoiksi (block)
- (RAID-järjestelmissä on useita kovalevyjä, joita "juovitetaan", "peilataan"ja "tarkistussummataan"jotta saadaan nopeampi ja toimintavarmempi tietovarasto; voi olla toteutettu laitteistotasolla tai ohjelmallisesti)

Hakuaikaan eli siihen, kuinka nopeasti levyltä saadaan luettua jotakin, vaikuttavat:

- lukupään siirtoaika uralta uralle
- pyörähdysviive, eli kuinka nopeasti uralla oleva sektori pyörähtää lukupään alle
- siirtoaika sisäiseen puskurimuistiin.

Laitteesta voidaan mitata ja ilmoittaa keskimääräinen siirtoaika (seek), pyörähdysaika (rotation) sekä sektorien määrä uralla (sectors). Keskimääräinen sektorin lukuaika on tällöin:

```
seek + rotation/2 + rotation/sectors
```

Sektorien määrä voi vaihdella levyn eri osien välillä, mikä hieman mutkistaa laskutoimituksia todellisuudessa.

8.3 Käyttöjärjestelmän I/O -osio

Käyttöjärjestelmän I/O:ta hoitavan ohjelmakoodin tyypillinen kerrosmainen rakenne on esitetty kuvassa 20. Edellä kuvattu I/O -operaation suoritus kulkee ohjelmistokerrosten läpi: Käyttäjän prosessi käyttää käyttöjärjestelmän kutsurajapintaa, joka abstrahoi alla olevat laitteet. Ne ovat ”vain jotakin” johon voi kirjoittaa tai josta lukea. Tämä ns. **laitteistoriippumaton ohjelmiston osa** delegoi sitten pyynnöt ajureille, joiden täytyy tuntea laitteiston rajapinta. Tämä ns. **laiteriippuva ohjelmiston** osa täytyy olla siis laitteen valmistajan tukemaa (joko he kirjoittavat ajurinsa tai julkaisevat laitteen rajapinnasta riittävän dokumentaation avoimien ajurien tekemiseksi). Laiteriippuva osio voi komentaa fyysistä laitetta väylän kautta. Tyypillisesti sen jälken tapahtuu fyysisen tapahtuman odottelua, jonka päättymisestä tulee tieto keskeytyksenä. Keskeytyksen hoitaa käyttöjärjestelmän keskeytyskäsittelijä, jonka tehtävänä on siirtää kontrolli jälleen laiteriippuvalle ohjelmistolle, jonka puolestaan on osattava muuntaa I/O -tulos muotoon, jota ylempi kerros eli laitteistoriippumaton osa käsittelee. Lopulta kontrolli tietysti palaa käyttäjän ohjelmalle, joka voi olla autuaan tietämätön laitteiston yksityiskohdista.

8.4 Laitteistoriippumaton I/O -ohjelmisto

Laitteistoriippumattoman ohjelmiston tehtävä on luoda yhtenäinen tapa käyttää laitteita (uniform interfacing). Mitä vaatimuksia tällaiselle tavalle asetetaan:

- tiedostojärjestelmä (file system; datan looginen organisointi tiedostoiksi ja hakemistoiksi)
- laitteiden nimeäminen (device naming; millä tavoin käyttäjän prosessi voi tietää fyysisesti asennettuna olevat I/O-laitteet ja päästä niihin käsiksi)
- käyttöoikeudet (protection; käyttäjillä oltava omat tietonsa joita muut eivät pääse sotkemaan, ja käyttäjillä voi olla eri valtuuksia lukea/kirjoittaa/suorittaa toimenpiteitä tietokoneella ja sen I/O-laitteiden osajoukolla)
- varaus ja vapauttaminen (allocating and releasing; tyypillinen I/O-laite on voitava varata yksinoikeudella prosessin käyttöön, jotta sen toiminnassa on järkeä)
- virheraportointi (error reporting; virheitä tapahtuu fyysisen maailman pakottamana ja niiden raportointi ja toipumiskeinot on tarjottava)
- laitteistoriippumaton lohkokoko (block size; erilaisten kovalevyjen ym. tallennusvälineiden käyttö yhtenäisen kokoisiin datamöykkyihin jaoteltuna)

Kuva 20: *I/O -operaatioon osallistuvat kerrokset, niiden väliset rajapinnat, ja operaation suoritusvaiheet ja osapuolet aikajärjestyksessä (1–7).*

- puskurointi (buffering; esim. peräkkäisten näppäinpainallusten tallennus tai yhden tavun lukeminen kovalevyltä, joka antaa kuitenkin sektorin kerrallaan)
- yhtenäinen ajuriliitäntä (device driver interface; erilaisten laitteiden valmistajien, tai ainakin ajurien tekijöiden, täytyy tietää millaisen rajapinnan toteuttamista käyttöjärjestelmä vaatii ajuriohjelmistolta)

9 Tiedostojärjestelmä

9.1 Unix-tiedostojärjestelmä, i-solmut

Esimerkkinämme olkoon kuvassa 21 havainnollistettu perinteinen Unix-tiedostojärjestelmä, jossa kullakin tiedostolla on **i-numero** (engl. *i-number*), eli indeksi **i-solmujen** (engl. *i-node*) taulukkoon. I-solmutaulukko on tiedostojärjestelmän tietorakenne, joka sisältää jokaista tiedostoa (tai tarkemmin i-solmua) kohden seuraavat tiedot:

- tyyppi eli tavallinen/hakemisto/linkki/erikoistiedosto
- suojaustiedot eli omistava käyttäjä (UID, user ID) ja ryhmä (GID, group ID).
- aikaleimat (käyttö, tiedoston sisällön muutos, i-solmun muutos)
- koko (size; tavuina)
- lohkojen määrä (block count)
- lohkojen suorat osoitteet (direct blocks; esim. 12 kpl, pienille tiedostoille riittävä)
- yksinkertainen epäsuora osoite (single indirect; viittaa yhteen lisätaulukkoon lohkojen osoitteita)
- kaksinkertainen epäsuora osoite (double indirect; viittaa taulukkoon, jonka alkiot viittaavat taulukkoihin lohkojen osoitteista)
- kolminkertainen epäsuora osoite (triple indirect; taulukko, josta taulukoihin, joista taulukoihin, joissa lohkojen osoitteita)

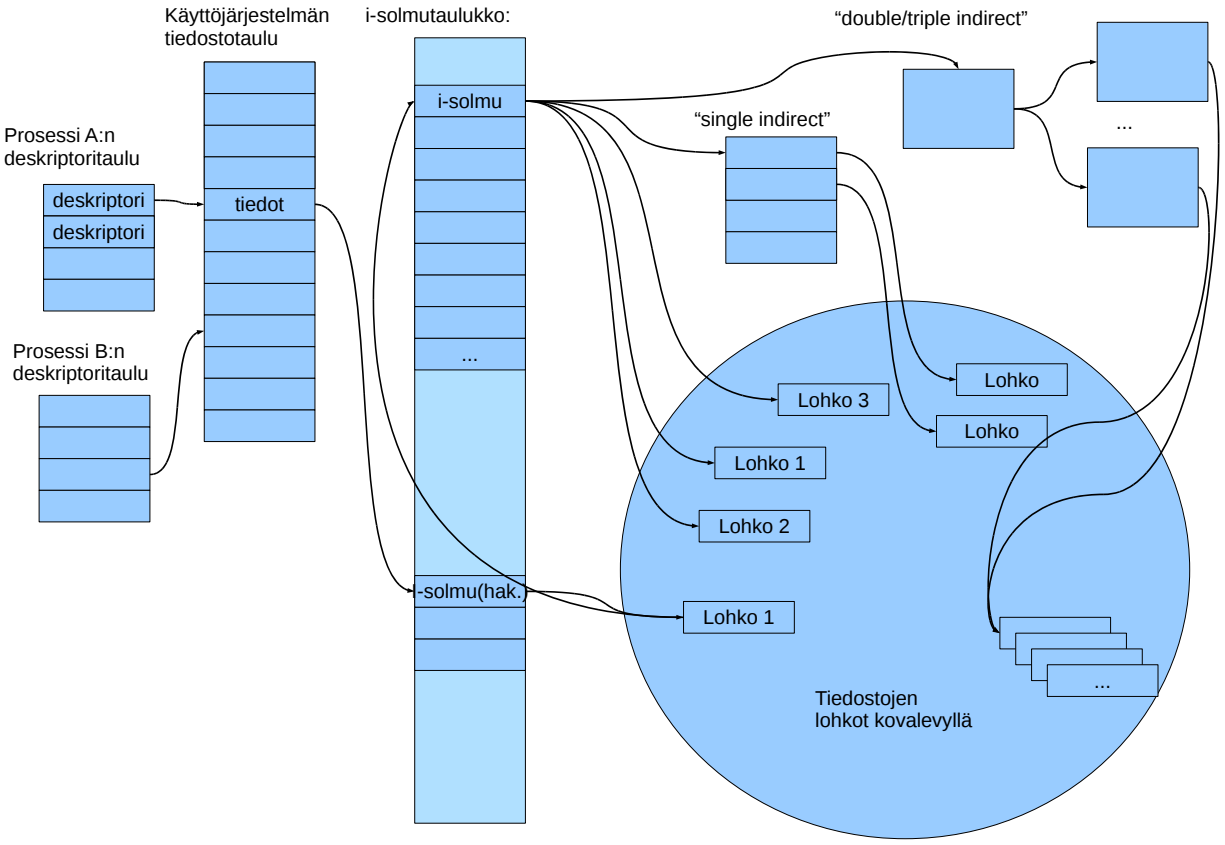
Tiedostojen tyypit Unix-tiedostojärjestelmässä voivat olla seuraavat:

- tavallinen (esim. tekstitiedosto kuten lähdekoodi, valokuva, videotiedosto, ...; olennaisesti bittijono jossa i-solmujen ilmoittamien lohkojen fyysinen sisältö peräkkäin)
- hakemisto (hakemistojen idea on järjestellä tiedot hierarkkisesti t. puumaisesti). Hakemisto Unixissa on käytännössä taulukko, jossa on jokaista hakemiston sisältämää tiedostoa kohden seuraavat tiedot:
  - tiedoston nimi
  - i-solmun numero

  Huomaa, että tässä järjestelmässä tiedoston nimi määräytyy hakemistotiedon perusteella! itse ”tiedosto” on lohkoissa lojuva bittijono, jolla on kyllä tietyt i-solmun metatiedot mutta ei nimeä.
- linkki (uusi viite fyysiseen tiedostoon, jolla on toinenkin sijaintipaikka; käytännössä nimi, jota kautta pääsee käsiksi tiedostoon jolla on vähintään yksi toinenkin nimi)
- erikoistiedosto (vastaa laitetta tai muuta käyttöjärjestelmän tarjoamaa rajapintaa)

Tiedostojen (perinteiset) oikeudet Unixeissa ja vastaavissa:

- oikeudet: kirjoitus / luku / suoritus (read / write / execute)
- tasot: omistaja / ryhmä / muut (user / group / other)
- oikeuksien ja tasojen kombinaatioita on $2^9 = 512$. Nämä on tyypillistä kirjoittaa oktaalilukuna, joka vastaa kolmiosaista (omistaja-ryhmä-muut) kolmijakoista (kirjoitus-luku-suoritus) bittijonoa. esim. 0755 on ohjelmatiedostolle sopiva: kaikki voivat lukea ja suorittaa tiedoston mutta vain omistava käyttäjä voi tehdä ohjelmatiedostoon muutoksia.
- setuid - bitti: jos tämä bitti on asetettu ohjelmatiedostossa, käynnistetyn prosessin omistajaksi tulkitaan (ns. effective user) tiedoston omistaja, ei se käyttäjä joka varsinaisesti ajaa ohjelman
- setgid - bitti: jos tämä bitti on asetettu ohjelmatiedostossa, käynnistetyn prosessin omistavaksi ryhmäksi (ns. effective group) tulkitaan tiedoston omistava ryhmä, ei siis ajavan käyttäjän oletusryhmä.



Kuva 21: *Käyttöjärjestelmän tietorakenteita tiedostojärjestelmän toteuttamiseksi.*

- sticky - bitti: alkuperäisessä ideassa, jos tämä bitti on asetettu ohjelmatiedostossa, ohjelmaa ei poisteta muistista sen suorituksen loputtua (nopeampi käynnistää uudelleen). Nykyvariaatioissa kuitenkin eri käyttötarkoitukset – erityisesti esim. Linuxissa jos tämä bitti on asetettu hakemistolle, rajoittuu sisällön poisto ja uudelleennimeäminen vain kunkin tiedoston omistajalle tai pääkäyttäjälle, riippumatta kunkin tiedoston asetuksista.

Tämä on siis yksi esimerkki tiedostojärjestelmästä. Muitakin on paljon, ja toteutuksen yksityiskohdista riippuu mm. tiedostojen maksimikoko, oikeuksien asetuksen tarkkuus ym. seikat.

Tiedostojen järkevä käyttö ohjelmissa edellyttää joitakin käyttöjärjestelmän sisäisiä tietorakenteita. Ensinnäkin jokaisella prosessilla on **deskriptoritaulu** osana prosessielementtiä (deskriptoritauluja on siis yhtä monta kuin on prosesseja). Prosessi käyttää deskriptoritaulun indeksejä avaamiensa tiedostojen yksilöintiin.Deskriptoritaulussa on avatun / varatun tiedoston osalta viite käyttöjärjestelmän **tiedostotauluun**. Tiedostotauluja tarvitaan yhteensä yksi kappale, jossa on tiedot jokaista tiedostoa kohden, jonka jokin prosessi on avannut:

- todellisen avatun tiedoston i-numero (unix-järjestelmässä)
- tiedosto-osoitin (ts. missä kohtaa tiedostoa luku/kirjoitus on menossa)
- millaiset oikeudet prosessi on saanut avatessa tiedostoa.

NFS:ssä (Network file system) i-solmu voidaan laajentaa osoitteeksi toisen tietokoneen tiedostojärjestelmässä. Apuna ovat käyttöjärjestelmän verkkoyhteyspalvelut (networking) ja prosessien välisen kommunikoinnin palvelut (ipc) sekä asiakaspäässä että palvelijapäässä. Yhteys perustuu määriteltyihin kommunikaatioprotokolliin, joten NFS:llä kytketyillä tietokoneilla voi olla eri käyttöjärjestelmät, kunhan molemmissa on tuki NFS:lle.

Tiedostojärjestelmän käyttöjärjestelmäkutsuja ovat esimerkiksi seuraavat:

```
create()   – luo tiedoston
open()     – avaa tiedoston ja tarvittaessa lukitsee
read()     – lukee tiedostosta nykyisen tiedosto–osoittimen kohdalta
close()    – "sulkee" tiedoston, ts. vapauttaa lukituksen
```

Esimerkki hakemiston käytöstä, kun prosessi haluaa avata tiedoston (esim. nimeltä `grillikuvat/2011/nakit.jpg`):

1. Nykyisestä hakemistosta (esim. /home/ktunnus/Pictures/) käyttöjärjestelmä etsii rivin, jonka nimi on seuraavaksi etsittävä "grillikuvat".
2. Jos nimi "grillikuvat" löytyy, rivillä on uuden i-solmun osoite, jonka pitäisi nyt olla hakemistotyyppinen; käytetään tätä uutta hakemistoa etsimään taas seuraava nimi. (Jos ei nimeä löydy, tulkitaan pyyntö virheelliseksi ja hoidetaan tieto epäonnistumisesta eteenpäin.)
3. Toistetaan muillekin mahdollisille hakemistonimille. Lopulta vastaan tulee viimeinen osio, eli varsinaisen tiedoston nimi, tässä tapauksessa "nakit.jpg". Tiedostoa vastaava i-solmu on nyt löytynyt.

9.2 Käyttäjänhallintaa tiedostojärjestelmissä

Tiedoston oikeuksien asettaminen unixissa tapahtuu komennolla `chmod`. Esimerkkejä:

```
chmod u+x skripti.sh      # käyttäjälle suoritusoikeus
chmod ugo+r nakit.jpg     # kaikille lukuoikeus
chmod go-x hakemisto      # muilta kuin käyttäjiltä
                          # suoritusoikeus pois
```

Symbolinen linkki tosiaan on vain linkki toiseen tiedostoon, ja käyttöoikeudet määräytyvät tuon kohdetiedoston oikeuksien mukaan. Hakemiston oikeuksissa suoritus (x) merkitsee oikeutta päästä hakemiston sisältöön käsiksi (jos tietää siellä olevan tiedoston nimen). Erillisenä tästä on hakemiston lukuoikeus (r), joka sallii hakemiston sisällön listaamisen.

Windowsin graafisella tiedostoselaimella ("Windows Explorer") voi klikata tiedostoa oikealla napilla ja säätää oikeuksia Security -välilehdeltä.

Unixin "**mounttaus**" ja "**unmounttaus**: Komennolla `mount` voi unixeissa/linuxeissa liittää tallennuslaitteita haluamaansa kohtaan hakemistohierarkiaa. Käyttöjärjestelmä tulkitsee laitteiden sisältönä olevan tietyn tiedostojärjestelmän mukaista dataa. Komennolla `umount` vastaavasti voi katkaista yhteyden laitteeseen. Mountteja voi normaalisti tehdä vain pääkäyttäjä/ylläpitäjä.

9.3 Huomioita muista tiedostojärjestelmistä

Tiedostojärjestelmiä on monia! Esim. Linuxeissa yleiset ext2, ext3, ext4; Windowsissa tyypillinen ntfs; Verkon yli jaettava nfs; yksinkertainen fat; muistia kovalevyn sijaan käyttävä ramfs; lisäksi mm. 9p, afs, hfs, jfs, ... Tiedostojärjestelmät ovat erilaisia, eri tarkoituksiin ja eri aikoina syntyneitä. Niiden mahdollisuudet ja rajoitukset kumpuavat toteutustavasta.

Tiedostojärjestelmien näkyviä eroja ovat mm. tiedostonimien pituudet, kirjainten/merkkien tulkinta, käyttöoikeuksien asettamisen hienojakoisuus ym. Syvällisempiä eroja ovat mm. päämäärähakuiset toteutusyksityiskohdat:

- nopeus/tehokkuus eri tehtäviin, tilansäästö levyllä/muistissa (haku, nimen etsintä, tiedoston luonti, kirjoitus, luku, poisto, isot vai pienet tiedostot)

  → "Yksi koko ei sovi kaikille" - tiedostojärjestelmä on valittava kokonaisjärjestelmän käyttötarkoituksen mukaan. Esim. paljon pieniä tiedostoja voi toimia paremmin eri järjestelmässä kuin isojen tiedostojen käsittely.

- toimintavarmuus (mitä tapahtuu, jos tulee sähkökatko tai laitevika)

  → joissakin tiedostojärjestelmissä on toteutettu "transaktioperiaate" eli **journalointi**: Kirjoitusoperaatiot tehdään atominen kirjoituspätkä kerrallaan. Ensin tehdään kirjoitus yhteen paikkaan, "ennakkokirjoitus" (ei vielä siihen kohtaan levyä, mihin on lopulta tarkoitus) Kirjoitetaan myös tieto, mihin kohtaan on määrä kirjoittaa. Jos todellinen kirjoitus ei ehdi jostain syystä toteutua, se voidaan suorittaa alusta lähtien uudelleen käyttämällä ennakkoon tehtyä ja tallennettua suunnitelmaa, tai perua kokonaan jos itse suunnitelma oli jostain syystä pilalla.

10 Käyttöjärjestelmän suunnittelusta

10.1 Tavoiteasetteluja ja pohdintoja

Esimerkiksi käyttöjärjestelmän vuoronnukselle (tai jopa yleisemmin jollekin resurssia, kuten prosessoria tai hissiä, hyötykäyttävälle järjestelmälle) voidaan asettaa esimerkiksi seuraavanlaisia tavoitteita:

- käyttöaste (utilization): kuinka paljon prosessori pystyy tekemään hyödyllistä laskentaa vs. odottelu tai hukkatyö
- tasapuolisuus (fairness): kaikki saavat suoritusaikaa
- tuottavuus (throughput): aikayksikössä loppuun saatujen tehtävien määrä
- läpimenoaika (turnaround): suoritukseen kulunut aika
- vasteaika (response time): odotus ennen toimenpiteen valmistumista
- odotusaika (waiting time): kokonaisaika, jonka prosessi joutuu odottamaan

Kaikki tavoitteet ovat ilmeisen perusteltuja, mutta ne ovat myös silminnähden ristiriitaisia: Esim. yhden prosessin läpimenoaika saadaan optimaaliseksi vain huonontamalla muiden prosessien läpimenoaikoja (ne joutuvat odottamaan enemmän). Prosessista toiseen vaihtamiseen kuluu oma aikansa (täytyy huolehtia mm. tilannetiedon tallennuksesta ja palautuksesta jokaisen vaihdon yhteydessä). Siis käyttöaste heikentyy, jos pyritään vaihtelemaan kovin usein ja usean prosessin välillä.

Käyttöjärjestelmän algoritmit ovat jonotuksiin ja resursseihin liittyviä valintatehtäviä (vrt. pilvenpiirtäjän hissit, liikenteenohjaus tai lennonjohto). **Operaatiotutkimus** (engl. *operations research, OR*) on vakiintunut tieteenala, joka tutkii vastaavia ongelmia yleisemmin. Kannattaa huomata yhteys käyttöjärjestelmän ja muiden järjestelmien tavoitteiden sekä ratkaisumenetelmien välillä!

10.2 Esimerkki: Vuoronnusmenettelyjä

Tutkitaan esimerkkinä joitakin käyttöjärjestelmän vuoronnusmenettelyjä:

- FCFS (First come, first serve): prosessi valmiiksi ennen siirtymistä seuraavan suoritukseen; "eräajo"; historiallinen, nykyisin monesti turha koska keskeytykset ovat mahdollisia ja toisaalta moniajo monin paikoin perusvaatimus.
- Kiertojono (round robin): tuttu aiemmasta esittelystä: prosessia suoritetaan aikaviipaleen loppuun ja siirrytään seuraavaan. Odottavista prosesseista muodostuu rengas tai "piiri", jota edetään aina seuraavaan.
- Prioriteetit: esim. kiertojono jokaiselle prioriteetille, ja palvellaan korkeamman prioriteetin jonoa useammin tai pidempien viipaleiden verran. (vaarana alemman prioriteetin nääntyminen)::

```
Prioriteetti 0  [READY0] -> PID 24 -> NULL
Prioriteetti 1  [READY1] -> PID 7 -> PID 1234 -> PID 778 -> NULL
Prioriteetti 2  [READY2] -> PID 324 -> PID 1123 -> NULL
...
Prioriteetti 99 [READY99] ->    NULL
```

- Dynaamiset prioriteetit: kiertojono jokaiselle prioriteetille, mutta prioriteetteja vaihdellaan tilanteen mukaan:
  - prosessit aloittavat korkealla prioriteetilla I/O:n jälkeen (oletetaan ”nopea” käsittely ja seuraava keskeytys; esim. tekstieditori, joka lähinnä odottelee näppäinpainalluksia)
  - siirretään alemmalle prioriteetille, jos aika-annos kuluu umpeen, ts. prosessi alkaakin tehdä paljon laskentaa
  - lopputulemana on kompromissi: vasteajat hyviä ohjelmille, jotka eivät laske kovin paljon; hintana on se että runsaasti laskevien ohjelmien kokonaissuoritusaika hieman pitenee (pidennys riippuu tietysti järjestelmän kokonaiskuormasta eli paljonko prosesseja yhteensä on ja mitä ne kaikki tekevät; jos prosessori olisi muuten "tyhjäkäynnillä", saa matalinkin prioriteetti tietysti lähes 100% käyttöönsä).

Prosessorin lisäksi muitakin resursseja täytyy vuorontaa. Esim. **kovalevyn vuoronnus** (engl. *disk scheduling*):

- esim. kaksi prosessia haluaa lukea gigatavun eri puolilta levyä
- gigatavun lukeminen kestää jo jonkin aikaa
- lukeeko ensin toinen prosessi kaiken ja sitten vasta toinen pääsee lukemaan mitään ("FCFS"), vai vaihdellaanko prosessien välillä lukuvuoroa?
- kun lukuvuoroa vaihdellaan, kuinka suurissa pätkissä (lohko vai useampia) ja millä prioriteeteilla...
- kovalevyn vuoronnukseen (disk scheduling) liittyy fyysisen laitteen nopeusominaisuudet: esim. kokonaisuuden throughput pienenee, jos aikaa kuluu lukupään siirtoon levyn akselin ja ulkoreunan välillä; peräkkäin samalla uralla sijaitsevaa tietoa pitäisi siis suosia, mutta tämä voi laskea vasteaikaa muilta lukijoilta. Arvatenkin tarvitaan taas jonkinlainen kompromissi.

10.3 Reaaliaikajärjestelmien erityisvaatimukset

Reaaliaikajärjestelmä (engl. *real time system*) on sellainen järjestelmä, esimerkiksi tietokonelaitteisto ja käyttöjärjestelmä, jonka pitää pystyä toimimaan ympäristössä, jossa asiat tapahtuvat todellisten ilmiöiden sanelemassa ”reaaliajassa”. Esimerkkejä ”reaaliajasta”:

- Robottiajoneuvo kulkee eteenpäin, ja sitä vastaan tulee este; esteen havaitseminen ja siihen reagoiminen ilmeisesti täytyy tapahtua riittävän ajoissa, koska muuten tapahtuu törmäys.
- Syntetisaattoriohjelman on tarkoitus tuottaa äänisignaalia millisekunnin mittaisissa aikaikkunoissa; ääniohjain ilmoittaa tulostuspuskurin olevan pian tyhjä, jolloin rumpukoneohjelman on pystyttävä kirjoittamaan seuraava pätkä riittävän ajoissa, koska muuten ääniohjaimen on pakko puskea tyhjä tai puolivalmis puskuri kaiuttimiin, joista kuuluu tällöin ikävä rasaus. Näytönpäivityksen osalta tilanne on vastaava, mutta lyhyt grafiikan nykäys on usein vähemmän häiritsevä kuin voimakas häiriö äänentuotossa.
- Kuuraketin nopeussensori huomaa suunnan kallistuvan vasemmalle; ohjausjärjestelmälle on riittävän pian saatava komento korjausliikkeestä, koska muuten kallistus saattaa kärjistyä katastrofaalisesti eikä matkustajille käy hyvin.

Reaaliaikajärjestelmän yleisiä vaatimuksia ovat seuraavat:

- determinismi (determinism); olennaiset toimenpiteet saadaan suoritukseen aina riittävän pian niitä tarvitsevan ilmiön (esim. prosessorin keskeytyksen) jälkeen
- vaste/responsiivisuus (responsiveness); olennaiset toimenpiteet saadaan päätökseen riittävän pian käsittelyn aloituksesta
- hallittavuus (user control); käyttäjä tietää, mitkä toimenpiteet ovat olennaisimpia - tarvitaan keinot kommunikoida nämä käyttöjärjestelmälle, mikäli kyseessä on yleiskäyttöinen käyttöjärjestelmä (ja dedikoitu järjestelmä olisi varmaan alun alkaenkin suunniteltu käyttäjiensä sovellustietämyksen perusteella)
- luotettavuus (reliability); vikoja esiintyy riittävän harvoin/epätodennäköisesti
- vikasietoinen toiminta (fail-soft operation); häiriön tai virheen ilmetessä toiminta jatkuu - ainakin jollain tavoin. Esim. vaikka robottiajoneuvon vaste oli kertaalleen liian pitkä ja se törmäsi esteeseen ja meni osittain rikki, niin ohjausta pitäisi edelleen jatkaa ettei vauhdissa tule lisää vaurioita.

Edellä olevassa ”riittävä” tarkoittaa reaaliaikailmiön luonteesta riippuen eri asioita. Joissain sovelluksissa esim. mikrosekunti on tämä riittävän lyhyt aika, toisissa taas minuutti tai tuntikin saattaa riittää. Käytännössä hankalinta on tietysti hallita lyhyitä aikajaksoja, joihin mahtuu pieni määrä prosessorin kellojaksoja tai prosessien aikaviipaleita.

Normaali interaktiivinen tietokoneen käyttö ei edellytä ”reaaliaikaisuutta”. Determinismi- ja vastevaatimukset eivät ole lukkoon lyötyjä eivätkä kriittisiä esim. WWW-sivujen lataamisen ja katselun tai tekstinkäsittelyn kannalta. Kriittisemmäksi tilanne muuttuu, jos laitetta käytetään esim. multimediaan; esim. toimintapelien elämyksellisyys voi vaatia riittävän nopeata kuvan ja äänen päivitystä.

Reaaliaikaiseen käyttöjärjestelmään liittyy termi **pre-emptiivisyys** (engl. *preemption/pre-emption*) mikä tarkoittaa että prosessin toiminta voi keskeytyä kun suuremman prioriteetin prosessi tarvitsee palvelua.

Pre-emptioksi voidaan sanoa jo sitäkin kun prosessi ylipäätään voi keskeytyä kesken laskennan (siis aikaannoksenkin loppumiseen); reaaliaikajärjestelmissä pre-emption rooli on kuitenkin merkityksellisempi - mm. keskeytyskäsittelyjä ja muita käyttöjärjestelmän toimenpiteitä (joita ei välttämättä tarvitsisi keskeyttää ei-reaaliaikajärjestelmässä) täytyisikin voida keskeyttää, jos tulee "se tärkeä keskeytys".

10.4 Käyttöjärjestelmien historiaa ja tulevaisuutta

"Käyttöjärjestelmien historia ja tulevaisuus" liittyy ilman muuta tämän kurssin oppimistavoitteisiin. Esitysjärjestys tällä kertaa ei kuitenkaan ollut kronologinen vaan lähdimme liikkeelle hands-on esimerkkien kautta nykypäivästä ja olemassa olevista artefakteista: mitä meillä on (tietokone), ja millainen ohjelmisto sen käyttämiseksi nykytarpeisiin tarvitaan (nykyaikainen käyttöjärjestelmä). Niinpä tämä tyypillisesti (esim. oppikirjassamme) ensimmäinen luku käsitellään tässä vaiheessa, kun jo tiedetään nykyaikaisen käyttöjärjestelmän piirteet.

Käyttöjärjestelmien historia liittyy luonnollisesti elimellisesti tietokonelaitteistojen historiaan:

1940-luku, 1950-luku: Ensimmäiset tietokoneet, ei käyttöjärjestelmää. Ongelmia mm. ajankäytön kannalta: Miljoonien arvoista konetta varattiin kalenterista kuin Kortepohjan pyykkikonetta, esim. tunti kerrallaan. Jos tunnin aikaikkunassa tehtiin vartin työ, jäi 45 minuuttia hukattua aikaa. Ongelmana oli siis kallis hukka-aikaa, joka johti jonkinlaisen automaattisen vuoronnuksen tarpeeseen.

1950-luku: "monitoriohjelmat" – ensimmäiset koko ajan muistissa olevat "proto-käyttöjärjestelmät". Monitori odotti että yksi ns. **työ** (engl. *job*) tuli suoritetuksi loppuun, minkä jälkeen se automaattisesti latasi sisään seuraavan työn ja käynnisti sen.

1960-luku: monitoriohjelmia sekä "JCL", job control language, ideana 'compile, load, and go". Ohjelmoinnin helpottamiseksi oli käytössä erilaisia ohjelmointikieliä ja näitä varten tehtyjä kääntäjäohjelmia. Ohjelmat luettiin sisään massamuistista (nauhat, reikäkortit) ihmisen ymmärtämänä lähdekoodina, esim. FORTRAN-kielisenä. Lisäksi JCL-kielinen ohjausohjelma, jossa mm. ilmoitettiin seuraavalle ohjelmalle käytettävä kääntäjä. Käyttöjärjestelmä hoiti kääntämisen, lataamisen ja käynnistämisen. Edelleen mm. ei moniajoa eikä muistinsuojausta. Havaittuja ongelmia olivat mm. seuraavat:

- muistin suojaus – käyttäjän ohjelma saattoi huolimattomalla muistiin sijoittamisella rikkoa monitoriohjelman ohjelmakoodin tai JCL-kieliset ohjeet.
- oli siis selkeä tarve prosessorin käyttöjärjestelmätilalle – esim. käyttäjän ohjelma ei saisi vahingossa(kaan) kajota suoritusvuorolistaan. Tulevaisuuden käyttöjärjestelmä tarvitsisi tulevaisuuden prosessorin, joka toimii vähintään kahdessa suojaustilassa.
- tasapuolisuus – pitäisi voida ajaa välillä yhtä ohjelmaa ja välillä toista.
- keskeytykset – pitäisi voida jättää esim. I/O:ta tarvitseva ohjelma odottelemaan ja ajaa muita sillä välin. Tulevaisuuden käyttöjärjestelmä tarvitsisi tulevaisuuden prosessorin, jossa suoritus voidaan keskeyttää ja siirtyä käyttöjärjestelmän hallintaosion suoritukseen.

1960-luku (edelleen): aikajakojärjestelmät (time-sharing systems). Koneet olivat yhä isoja ja kalliita, mutta moni työntekijä olisi pystynyt tekemään tehokasta työtä niiden avulla. Ratkaisuna tähän päätteet ja käyttäjien kesken jaetut aikaikkunat/aikaviipaleet. Syntyi prosessin käsite; mainittakoon merkkipaaluna Multics -järjestelmä.

Useiden prosessien ja jaettujen resurssien käytöstä syntyivät luonnollisesti ongelmat synkronoinnissa, poissulussa ja lukkiutumisessa. Lisäksi muistin sekä käyttäjien ja käyttöoikeuksien hallinta tuli entistä tärkeämmäksi.

1970-luku: Järjestelmiä olivat mm. Multics sekä kehitteillä olevat UNIX ja MS-DOS. Käyttöjärjestelmästä oli tulossa selvästi aiempaa monipuolisempi järjestelmä. Siis oli aiempaa suurempi koodimassa, jollaista on hankalampi hallita. Uusia haasteita oli siis tarvittavan kokonaisuuden jäsentäminen. Tuloksena käyttöjärjestelmän piirteiden luettelointi ja kerroksittainen jäsennysmalli.

1980-luvulta alkaen: moniajon ja suojausten kehittämistä, moniydinprosessorit ja niiden asettamat haasteet, hajautetut käyttöjärjestelmät, oliopohjaisuus. Mikrotietokoneet ja oheislaitteiden kirjo. Verkkoyhteydet ja internet.

2000-luku: kännykät ym. sulautetut laitteet. Digikamerassakin on oltava vähintään tiedostojärjestelmä, jotta kuvat saadaan tallennettua muistikortille (joka itsessään osaa vain tallentaa pötkön bittejä).

Tulevaisuus: ei voitane ennustaa? Jotakin muuta kuin tänään. Ilmeisesti jatkuvia trendejä ovat hajautus, liikkuvat laitteet sekä moniydinprosessorit.

10.5 Millaisia ratkaisuja käyttöjärjestelmän tekemisessä voidaan tehdä

Kuvissa 22 – 25 on kuvailtu joidenkin käyttöjärjestelmien, eli vanhan ja uuden Unixin, Linuxin sekä Windows 2000:n moduulien jakautumista karkealla tasolla. Havaintoja:

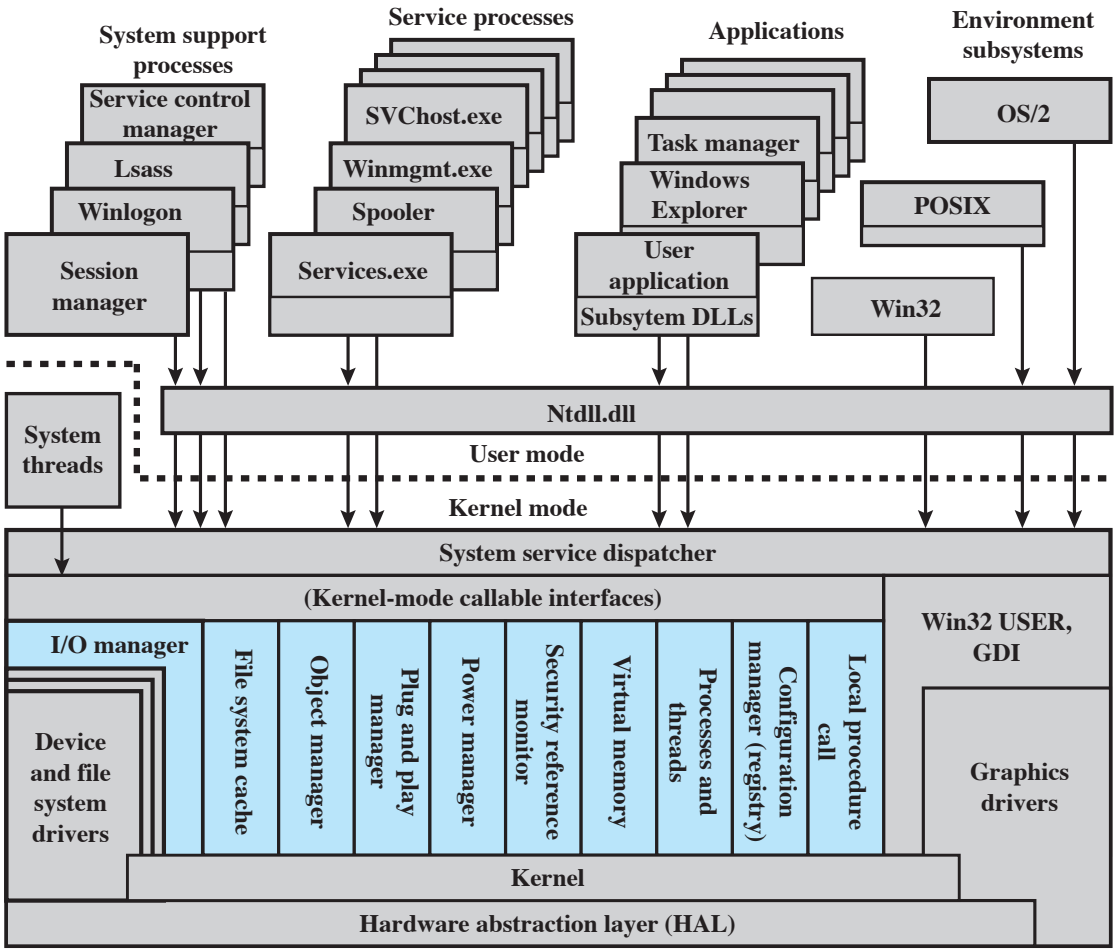
- Kaikissa esimerkeissä näkyy jollain tapaa hierarkkinen rakentuminen ja kerrokset (kuten aikalailla kaikessa ohjelmoinnissa).
- Kuitenkin on varsin erilaisia tapoja niputtaa moduulit. Jopa siinä määrin että jossain järjestelmässä graafinen käyttöliittymä ajatellaan kuuluvaksi käyttöjärjestelmään, ja joissain taas ei.
- Windowsiin liittyvässä moduulijaossa nähdään ero käyttäjätilassa ja käyttöjärjestelmätilassa toimiviin moduuleihin.

Erityispohdintana tarkastellaan vielä **monoliittista** (engl. *monolithic*) ydintä ja **microkernel**-ydintä, eli miten organisoida käyttöjärjestelmän osiot ajon aikana: Monoliittinen käyttöjärjestelmä on yksi kokonaisuus, joka sisältää kaikki ominaisuudet ja palvelut yhdessä koodiköntässä. Käyttöjärjestelmäkutsu vastaa jotakuinkin aliohjelman kutsumista. Sen sijaan microkernel-mallissa käyttöjärjestelmätilassa toimiva ”ytimen ydin” on mahdollisimman pieni, ts. vain vuoronnus (ym. prosessorin rajoittamatonta käskykantaa ja rekisterejä tarvitseva osuus) ja IPC-menettely jolla mahdollistuu prosessien kommunikointi ja synkronointi. Muu toiminnallisuus, ml. laiteajurit (tietenkään keskeytyskäsittelijöitä lukuunottamatta), tiedostojärjestelmä sekä muistinhallinnan logiikka, ovat käyttäjätilassa toimivia prosesseja. ”Käyttöjärjestelmäkutsun” sijasta prosessit voivat lähettää viestejä käyttöjärjestelmän palveluprosesseille.

Hyveitä microkernel-mallissa ovat mm.

- modulaarisuus (mm. pakko eriyttää toiminnot rajapintojen taakse),
- turvallisuus (käyttöoikeudet jaettavissa eri toimia hoitavien prosessien kesken),
- vikasietoisuus (palveluprosessin kaatuminen ei kaada koko järjestelmää)

Käytännössä microkernel-ominaisuuksia on mukana nykyisissä järjestelmissä, mutta tehokkuussyistä ytimissä on mukana suhteellisen paljon ominaisuuksia; ei siis ole täydellisen hyveellistä microkerneliä vaan kompromissiratkaisuja (jollaisia aina tarvitaan).



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link libraries

Colored area indicates Executive

Figure 2.13 Windows 2000 Architecture [SOLO00]

Kuva 22: *(Stallingsin oppikirjasta [1] otettu, kun olis vähän kova homma piirtää uusiksi.)*

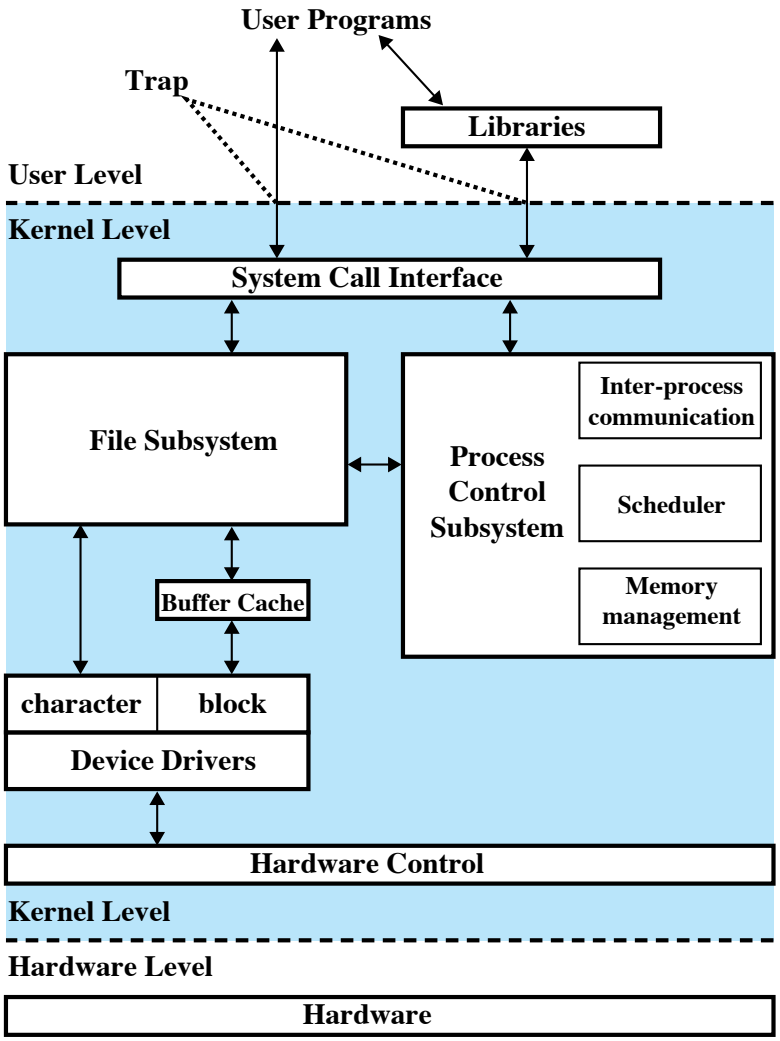


Figure 2.15 Traditional UNIX Kernel [BACH86]

Kuva 23: *(Stallingsin oppikirjasta [1] otettu, kun olis vähän kova homma piirtää uusiksi.)*

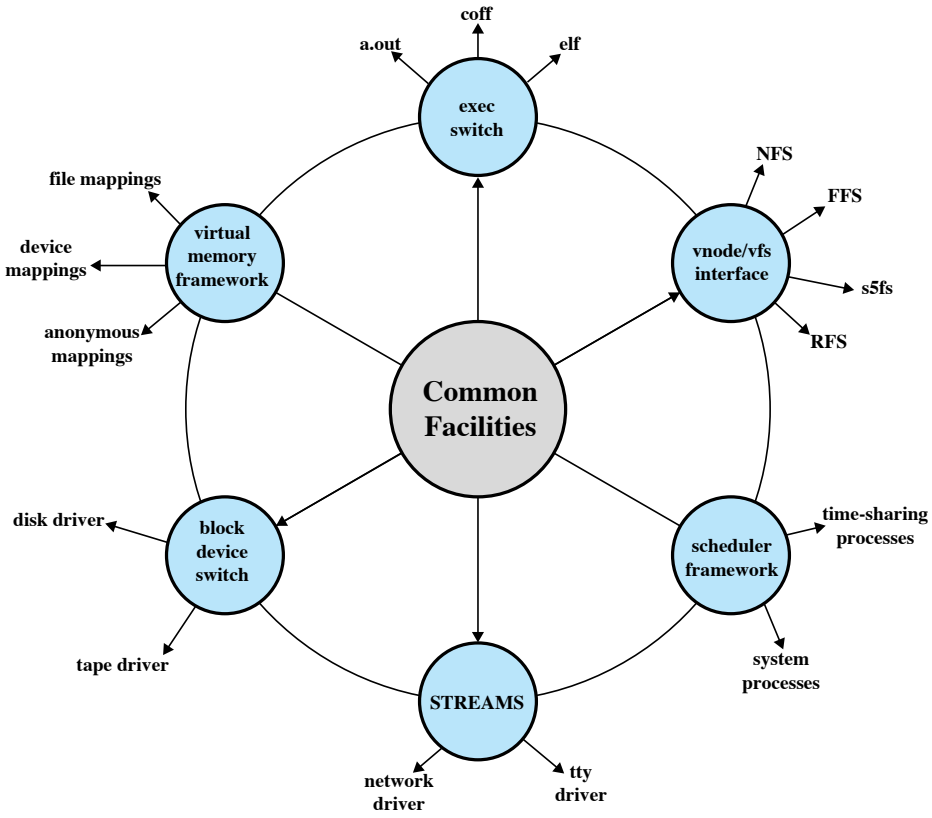


Figure 2.16 Modern UNIX Kernel [VAHA96]

Kuva 24: *(Stallingsin oppikirjasta [1] otettu, kun olis vähän kova homma piirtää uusiksi.)*

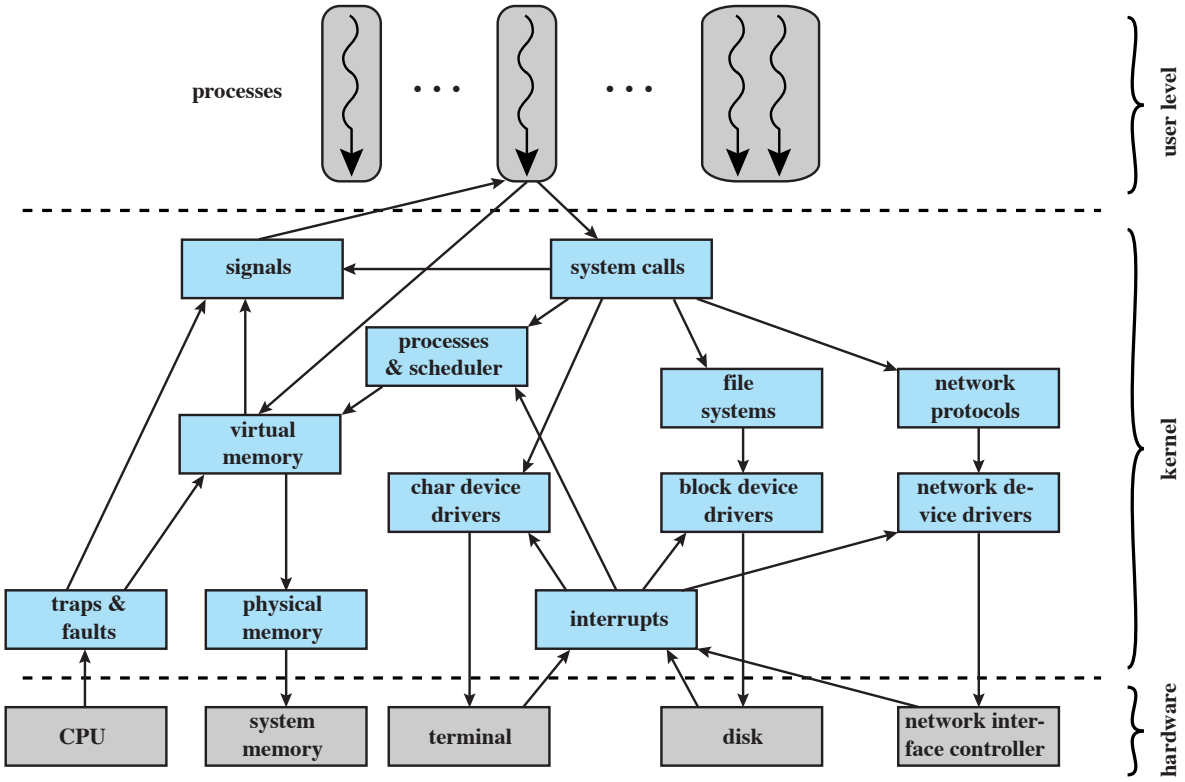


Figure 2.18 Linux Kernel Components

Kuva 25: *(Stallingsin oppikirjasta [1] otettu, kun olis vähän kova homma piirtää uusiksi.)*

11 Epilogi

11.1 Yhteenveto

Toivottavasti on tähän mennessä nähty, että vaikka tietokone (edelleenkin, jopa aikojen saatossa syntyneine lisäteknologioineen) on pohjimmiltaan yksinkertainan laite, logiikkaportteihin perustuva bittien siirtäjä, on siihen ja sen käyttöön aikojen saatossa kohdistunut uusia vaatimuksia ja ratkaistavia haasteita. Tuloksena on laaja ja monimutkainen järjestelmä, jonka kokonaisuuden ja yksittäiset osa-alueet voi toteuttaa erilaisin tavoin. Haasteet muuttuvat aikojen myötä, joten käyttöjärjestelmien piirteitä on jatkuvasti tutkittava. Alan konferensseja ja lehtiä voi kiinnostunut lukija varmasti löytää internetistä esimerkiksi hakusanoilla ”operating system journal”, ”operating system conference” ja yleisesti ”operating system research”.

11.2 Mainintoja asioista, jotka tällä kurssilla ohitettiin

Monet asiat käsiteltiin pintapuolisesti, koska kurssin opintopistemäärä ei mahdollista kovin suurta syventymistä. Myöskään emme voi toisen vuoden kurssilla esimerkiksi teettää harjoitustyönä omaa käyttöjärjestelmää, kuten joillakin vastaavilla kursseilla on tapana. Tarkoitus olikin antaa yleiskuva siitä, mikä oikein on käyttöjärjestelmä, mihin se tarvitaan, ja millaisia osa-alueita sellaisen on hallittava. Terminologiaa ja käsitteitä esiteltiin luettelonomaisesti, jotta ne olisi tämän jälkeen ”kuultu” ja osattaisiin etsiä lisätietoa itsenäisesti.

Käsittelemättä jätettiin myös joitakin suositeltuja aihekokonaisuuksia, mm.

- tietoturvaan liittyvät seikat (security / security models) (”policyt”, tietoturvalaitteet, kryptografia, autentikointi) pääasiassa ohitettiin.
- Sulautettujen järjestelmien (embedded systems) erityistarpeita ei juurikaan käsitelty osa-alueiden yhteydessä. Pääasiassa käsiteltiin työasemien ja palvelimien näkökulmaa.
- Järjestelmän suorituskyvyn analysoinnista (performance evaluation) (tarpeet, menetelmät) ei ollut varsinaisesti puhetta.
- ”Sähköisestä todisteaineistosta” (digital forensics) (kerääminen, analysointi) ei ollut puhetta.

Viitteet

[1] William Stallings, 2009. Operating Systems – Internals and Design Principles, 6th ed.

[2] Pasi Koikkalainen ja Pekka Orponen, 2002. Tietotekniikan perusteet. *Luentomoniste.* Saatavilla WWW:ssä osoitteessa `http://users.ics.tkk.fi/orponen/lectures/ttp_2002.pdf` (linkin toimivuus tarkistettu 5.6.2011)

[3] Pentti Hämäläinen, 2010. Tietokoneen rakenne ja arkkitehtuuri. *Luentomoniste.*

[4] AMD64 Architecture Programmer's Manual Vol 1–5 `http://developer.amd.com/documentation/guides/pages/default.aspx`

[5] GAS assembler syntax. `http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_16.html`

[6] Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell (eds.). System V Application Binary Interface – AMD64 Architecture Processor Supplement (Draft Version 0.99.5) September 3, 2010. `http://www.x86-64.org/documentation/abi-0.99.pdf`

A Yleistä tietoa skripteistä

Nämä yleiset skripteihin liittyvät asiat eivät välttämättä tulleet demo1:n ohjeessa ihan näillä sanoin, joten liitetään tähän:

Shelleistä yleisesti ottaen:

- shell tarkoittaa "kuorta", joka "ympäröi"käyttöjärjestelmän ydintä ja jonka kautta käyttöjärjestelmää voidaan komentaa.
- Shelliä käytettäessä ollaankin varsin lähellä käyttöjärjestelmän rajapintoja.
- Merkittäviä shellejä ovat olleet mm. Bourne Shell (sh), sen laajennokset csh, ksh ja zsh sekä nykyisin varsin suosittu GNU Bourne Again Shell (bash). Paljon muitakin shellejä on kehitetty. Pääpiirteissään ne toimivat hyvin samalla tavoin. (Syntakseissa ja ominaisuuksissa on eroa)

THK:n nykyinen suositus on käyttää bashiä jalavassa. Oletus on kuitenkin monilla tcsh, historiallisista syistä. Katsottiin kuinka shellin voi vaihtaa salasana.jyu.fi -palvelussa, jos haluaa jatkossakin käytellä suorakäyttökoneita.

Shellejä voi käyttää interaktiivisesti eli kirjoittamalla komento kerrallaan, mutta niillä voi myös hiukan ohjelmoida. Shell-ohjelma on periaatteessa pötkö komentoja, joiden ympärille voi lisätä ohjelmointirakenteita kuten muuttujia, ehtoja, toistoja ja aliohjelmia. Shell osaa tulkita ja suorittaa tällaisen ohjelman, jota sanotaan skriptiksi (joskus erityisesti shell-skriptiksi).

Miksi tehdään skriptejä:

- usein tehtävät komentosarjat (esim. tiedostokonversiot, varmuuskopiot) on mukava sijoittaa helposti ajettavaan skriptiin.
- ajoitetut tehtävät (esim. varmuuskopiot klo 5:30) voidaan kirjoittaa skriptiin, joka suoritetaan automaattisesti tiettyyn aikaan (ajoitusapuohjelmalla, luonnollisestikin).
- konfigurointi (esim. käyttöjärjestelmän palveluiden ylösajo)
- itse shellin konfigurointi, esim. ympäristömuuttujien asetus.
- ohjelmistoasennukset.

Skriptejä tehdessä on syytä olla huolellinen ja huomioida erityistapaukset ja -tilanteet! Tästä nähtiin pieni esimerkki, jossa höpö skriptini päätyi luomaan tiedostot nimeltä “*.c“ ja “*.java“ mikä kyllä varmasti varautumatonta käyttäjää hämmentäisi. Nähtiin myös että skriptin ajaminen uudelleen samoille tiedostoille alkuperäisessä tarkoituksessaan hajotti tiedostot, itse asiassa melko lopullisesti. Eli (kuten ohjelmoinnissa ylipäätään) on paljon kiinni varautumisesta erilaisiin lähtötilanteisiin, ja loppukäyttäjältähän ei yleensä voi olettaa minkäänlaista osaamista tai ymmärrystä, joten varsinkin käyttäjän "tyhmyyteen"kannattaa useimmiten luottaa, joskus myös omaansa. (Tämä ei tietysti estä tekemästä lyhyttä ja turvatonta skriptiä itselleen tässä ja nyt, jos sellaisesta on hyötyä).

Huomioitiin, että skriptejä voi tehdä shellin lisäksi millä tahansa muulla tulkattavalla ohjelmointikielellä (perl, python, ...). Shellin käyttö on perusteltua, jos ei voida olettaa että hienompia alustoja olisi asennettu koneelle, jossa skriptit tarvitsee ajaa. Esim. bash löytyy todella monista Unix/Linux -koneista ja se on saatavilla myös Windowsille.

B Käytännön harjoituksia

Harjoitukset kuuluvat kurssin sisältöön, ja niissä on tietoa, jota muissa monisteen osissa ei ole käsitelty, joten niihin on syytä tutustua. Teknisesti niitä ei vielä ole liitetty tähän monisteen liitteeksi, joten toistaiseksi nämä on katsottava erikseen kurssin nettisivulta:

B.1 Sormet Unixiin

B.2 Sormet C:hen

B.3 Sormet skripteihin

B.4 Miniharjoitustyö: assembler-ohjelma ja debuggaus