ITKA203 – Käyttöjärjestelmät Kurssimateriaalia: ”luentomoniste”

Tämän version tilanne: Tämä versio monisteesta on käyttökelpoinen vuoden 2019 kurssille.

Kehityskohteita ja muutosideoita on kerääntynyt iso joukko, mutta niitä käydään läpi niin sanotusti silloin, kun jollain on riittävästi luppoaikaa… ei ehkä tarvitse pidätellä hengitystä suurempia muutoksia odotellessa. Jotakin pientä voi muuttua kurssin aikana, mutta ei kuitenkaan niin, etteikö ensimmäisenä kurssipäivänä omalle koneelle ladattu tai paperille tulostettu versio olisi käyttökelpoinen alusta loppuun saakka.

Merkittävät asiavirheet korjataan välittömästi – ilmoita heti, jos löydät sellaisen!

Esipuhe

Tämä on Jyväskylän yliopiston Informaatioteknologian tiedekunnan kurssia ITKA203 Käyttöjärjestelmät tukeva teksti. Kädessäsi oleva versio on tulostettu LaTeX -ladontajärjestelmällä päivämäärällä 25. maaliskuuta 2019. **Tämän lisäksi kurssin sisältöön kuuluvat sekä luennoilla nähtävät esimerkit että demot laajoine opastusteksteineen**. Koko kurssimateriaalin viimeisin kehitysversio löytyy versionhallinnasta seuraavasta sijainnista:

https://yousource.it.jyu.fi/itka203-kurssimateriaalikehitys/itka203-kurss

Edellä mainitussa sijainnissa löytyy myös kohtalaisen tarkoin määritellyt osaamistavoitteet, joiden saavuttamiseen demot, esimerkit ja myös tämä teksti tähtäävät.

Keväästä 2014 alkaen materiaali on ollut YouSource-järjestelmässä siinä toivossa, että sitä kehitettäisiin yhteisvoimin kulloisenkin vastuuopettajan koordinoimana. Mikäli haluat selventää aihepiiriä nykyistä paremmin, ota rohkeasti yhteyttä projektiin, jotta pääset mukaan kirjoittamaan ja korjaamaan monistetta paremmaksi ja alan kehittyviä tarpeita vastaavaksi! Mikäli löydät asiavirheitä tai epäselvyyksiä, joita ei ole sellaisiksi merkitty, ota yhteyttä välittömästi!

Tekijät sitoutuvat sijoittamaan osuutensa avoimen lisenssin alle sekä käyttämään yhteistä versionhallintajärjestelmää muutosten tekemiseen. Hyvistä kontribuutioista voitaneen kurssin yhteydessä antaa bonuspisteitä tenttiin; näistä on neuvoteltava vastuuopettajan kanssa etukäteen.

Lisenssi

Tämä teos on lisensoitu Creative Commons Nimeä-JaaSamoin 4.0 Kansainvälinen -käyttöluvalla.

Tekijät ja kiitokset

Allekirjoittanut on kasannut materiaalin vuosina 2007-2019 hyödyntäen aiempien opettajien (Jarmo Ernvall, Pentti Hämäläinen) aiherajauksia sekä luentomateriaaleja. Vahvana vaikuttimena on William Stallingsin oppikirja [1], tosin etenkin käytännön osioissa materiaali on originaalia ja perustuu suoraan laite- ja rajapintadokumentaatioon. Kevään 2014 aikana merkittäviä uudistuksia teki tuntiopettaja Juha Rautiainen. Emma Lindfors kävi syksyllä 2014 monisteen läpi opiskelijan näkökulmasta antaen arvokkaita ehdotuksia täsmennyksistä ja rakenteen selkeyttämisestä. Keväällä 2015 Ari Tuhkala piirsi havainnekuvat aiempaa kauniimmin ja Tomi Lundberg auttoi dokumentin strukturoinnissa sekä tuotti demoihin kaivatun vim-tekstieditorin alkuopastuksen. Keväällä 2019 Jonne Itkonen teki erittäin merkittäviä ilmiasuparannuksia ja sisältötäsmennyksiä. Hämmentävän läheisesti vastaavan kaltainen kurssimateriaalin kehitysprosessi vaikuttaa johtaneen ilmaiseen englanninkieliseen verkko-oppikirjaan [2], jota allekirjoittanut onkin taipuvainen suosittelemaan lisämateriaaliksi.

Muutama sana terminologiasta ja alaviitteistä

Tärkeintä on aina käsitteet termien takana, mutta alalla tarvitaan myös oma erityissanasto, jolla voidaan kommunikoida asiat lyhyesti. Tälläkin *kurssilla tulee vastaan huikea määrä uusia sanoja*, joilla kutakin esiteltyä käsitettä tai rakennelmaa symboloidaan puhutussa ja kirjoitetussa kielessä. Käsite- ja sanastovyöryyn on syytä varautua kurssin ensi hetkistä alkaen!

Monet käsitteet ja sanat ovat syntyneet englanninkielisissä ympäristöissä. Alkuperäissanat ovat enemmän tai vähemmän loogisia analogioita reaalimaailman ilmiöistä. Suomalaiseen sanastoon on poimittu enemmän tai vähemmän loogisin perustein englanninkielisiä lainasanoja tai sanayhdistelmiä. Suomennoksiakin on tehty,

enemmän tai vähemmän loogisin perustein, ja on pyritty vakiinnuttamaan suomalaiseen suuhun taipuvaa terminologiaa ammattikieleen.

Vaikka olenkin yleensä kielipoliisi pahimmasta päästä, aiheen työstäminen suomeksi asettaa haasteita: Pitäisikö puhua esimerkiksi suorittimesta vai prosessorista, vuorontajasta vai skedulerista, näennäismuistista vai virtuaalimuistista, näennäis- vai virtuaalikoneesta, kuoresta vai shellistä, lokitiedoista vai logitiedoista. . . Kieltäm monet sanoista on peräisin omasta ja lähipiirini puheenparresta, jonka syntyhistoria paikallisessa tietojenkäsittelykulttuurissa on vanhempi kuin minä itse. Osittain moniste siis pitää yllä tuota kyseistä perinnettä sen sijaan mitä kielitoimiston suositukset mahdollisesti sanovat. Siihen olen pyrkinyt, että jokaisesta asiasta käytetään säännönmukaisesti samaa sanaa sen jälkeen, kun käsitteen ensiesittelyn yhteydessä on listattu myös muut yleisesti käytetyt variaatiot. Olen tietoisesti käyttänyt lainasanoja ja englismejä niin runsaalla kädellä, että se sattuu jo omaan sieluun, mutta tällä olen halunnut kaventaa eroa kurssilla vastaan tulevan termin ja englanninkielisessä kirjallisuudessa esiintyvän sanaston välillä.

Lukija, jota monisteen lukuisat alaviitteet häiritsevät, ohittakoon ne huoletta[1].

Jyväskylässä keväällä 2016,
Paavo Nieminen `<paavo.j.nieminen@jyu.fi>` ja kevätkurssin tun-

[1]Alaviitteitä voi ajatella varsinaisen monisteen “kommenttiraitana”, jossa perustelen päätekstissä tehtyjä yksinkertaistuksia, kerron laajemmasta kontekstista, ja muutoinkin avaan taustoja. Varsinaisia “lillukanvarsia” olen pyrkinyt välttämään, mutta kuulun itse sellaiseen ryhmään, joka on omana opiskeluaikanaan nauttinut reunahuomautuksista, jopa lillukanvarsista, ja löytänyt niistä inspiraatiota varsinaiseen tekemiseen. Pyrin tietoisesti tarjoamaan näitä sekä luennoilla että materiaalissa. Parhaimmillaan olen onnistunut sijoittamaan nämä alaviitteisiin päätekstin sijaan. Demojenkin reunahuomautukset toivottavasti näyttävät sellaisilta jo päälle päin.

tiopettajat.

Sisältö

Sisältö **6**
0.1 Motivointia ja sijoittamista kokonaiskuvaan 8
0.2 Esitietoja 18
0.3 Hei maailma – johdattelua tietokoneeseen 64
0.4 Konekielisen ohjelman suoritus 89
0.5 Ohjelma ja tietokoneen muisti 119
0.6 Käyttöjärjestelmä 146
0.7 Keskeytykset ja käyttöjärjestelmän kutsurajapinta . 164
0.8 Prosessi ja prosessien hallinta 179
0.9 Yhdenaikaisuus, prosessien kommunikointi ja synkronointi 199
0.10 Muistinhallinta 228
0.11 Oheislaitteiden ohjaus 258
0.12 Tiedostojärjestelmät 273
0.13 Käyttöjärjestelmän suunnittelusta 297
0.14 Shellit ja shell-skriptit 309
0.15 Epilogi 324
.1 Pullantuoksuinen pehmojohdanto 331
.2 Koodiliite 360

Kirjallisuutta **406**

Hakemisto **408**

Kuvat **418**

0.1 Motivointia ja sijoittamista kokonaiskuvaan

Käyttöjärjestelmä on ohjelmisto, joka toimii rajapintana laitteiston ja sovellusohjelmien välillä. Sen tehtävä on tarjota palveluita, joiden kautta tietokonelaitteiston käyttö on muille ohjelmille suoraviivaista, tehokasta ja turvallista. Siinä se kaikessa lyhykäisyydessään oli. Loppu tästä kurssista on tämän lauseen avaamista. Karkea yleiskuva ei nimittäin riitä silloin, kun ratkaistavana on johonkin yksityiskohtaan liittyvä ongelma!

Tärkeätä on heti alkuun todeta, mitä tältä kurssilta *ei* kannata odottaa: Täällä ei ensinnäkään käsitellä graafisia käyttöliittymiä. Ei yhden yhtäkään ikkunaa painikkeineen suunnitella tai sen taustaväriä mietitä. Täällä ei mietitä, miten verkkokauppaa käytetään mobiililaitteella tai WWW-selaimella tai millaisia ajatuksia tietokonetta klikkailevan käyttäjän päässä liikkuu. Sen sijaan tämä on *matka ytimeen*, syvälle kohti puolijohteista valmistettua aparaattia, joka murskaa numeroita miljardi kertaa sekunnissa tikittävän kellon piiskaamana. Sieltä tullaan takaisin vain hieman sen rajapinnan yläpuolelle, jonka alapuolella on rauta ja yläpuolella softa. Asioista puhutaan ohjelmakoodilla, konekielellä ja heksaluvuilla.

Teknisesti orientoituneita, laitteiden toiminnasta kiinnostuneita opiskelijoita tämä aihepiiri yleensä kiehtoo luonnostaan. Muille asia saattaa tuntua lähtökohtaisesti vastenmieliseltä ja tarpeettomaltakin... se on kuitenkin illuusio, sillä sen verran olennaisesta informaatioteknologian koneiston rattaasta käyttöjärjestelmissä on kyse. Tärkeätä on joka tapauksessa ymmärtää heti aluksi, mitä tuleman pitää. Pidä edeltävän ohjelmointikurssin oppikirjasta kiinni, koska pian lähdetään laskeutumaan syvemmälle! Jos et innostukseltasi jaksa pysyä tuolilla, voit jatkaa luvusta 0.2. Muussa tapauksessa ilmeisesti epäröit, joten tarvitset lisää motivointia.

Aloitetaan mielikuvaharjoitteesta: Kirjoittelet opinnäytetyötäsi jollakin toimisto-ohjelmalla, esimerkiksi Open Office Writerilla. Juuri äsken olet retusoinut opinnäytteeseen liittyvää valokuvaa piirto-ohjelmalla, esimerkiksi Gimpillä. Tallensit kuvasta tähän asti parhaan version siihen hakemistoon, jossa opinnäytteen kuvatiedostot sijaitsevat. Molemmat ohjelmat (toimisto-ohjelma, kuvankäsittely) ovat auki tietokoneessasi. Mieleesi tulee tarkistaa sähköpostit. Käynnistät siis lisäksi WWW-selaimen ja suuntaat yliopiston Webmail-palvelimen osoitteeseen. Taskussasi piippaa, ja huomaat että kaverisi on kirjoittanut viestin Facebookin kautta suoraan omaan älypuhelimeesi ...

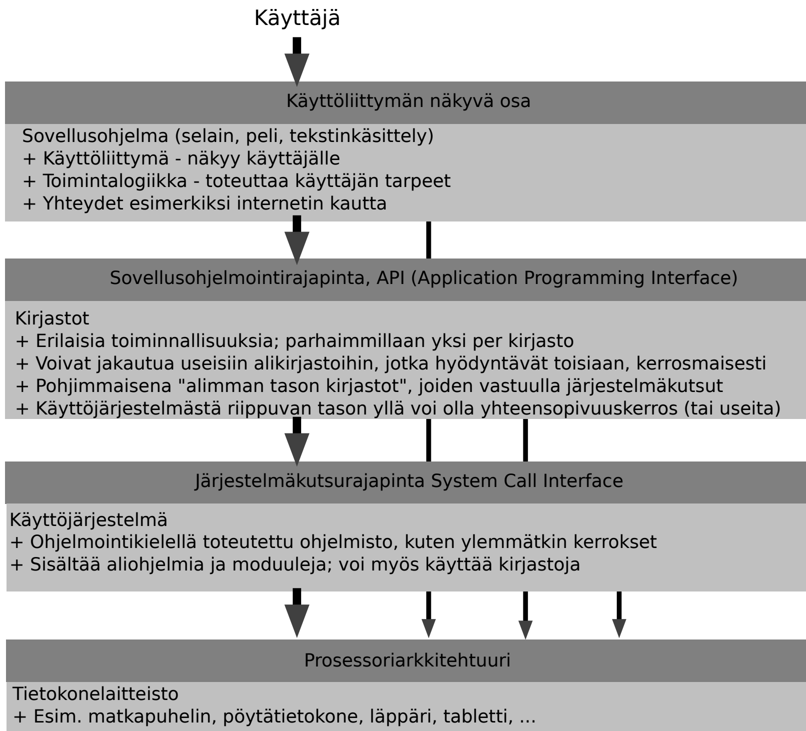
Edellä esitettyyn mielikuvaan lienee helppo samaistua. Teknologia on arkipäivää, jota ei tule ajateltua sen enempää. Sitä ”vain käytetään”, ja lapset saattavat oppia klikkaamaan ennen kuin lukemaan. Tällä kurssilla kuitenkin mennään pintaa syvemmälle. Äsken kuviteltu tilanne näyttää ulkopuolelta siltä, että käyttäjä klikkailee ja näppäilee syöttölaitteita, esim. hiirtä ja näppäimistöä tai älypuhelimen kosketusnäyttöä. 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 koko pitkän matkansa sovellusohjelmien käyttöön
- miten on teknisesti mahdollista, että käytössä on monta ohjelmaa yhtä aikaa (ja miksi se ei oikeastaan ole mitenkään itsestäänselvää)
- mitä on huomioitava, kun tehdään ohjelmia, jotka ratkaisevat samaa ongelmaa yhdessä (”yhdenaikaisesti”)

- mitä oikeastaan tarkoittaa se, että jotakin tallennetaan pysyvästi "tietokoneeseen"
- miksi pitäisi nostaa hattua jollekin, joka on saanut kehitettyä käyttökelpoisen käyttöjärjestelmän.

Lisäksi tavoitteena on lisätä monelta muultakin osin alan yleissivistystä sekä ottaa haltuun perusteluineen ne ohjelmoinnilliset yksityiskohdat, joita hyvien ohjelmien tekeminen nykytietokoneille vaatii. Ensimmäisen ohjelmointikurssin antamaa perustaitoa laajennetaan antamalla kevyt yleiskäsitys kahdesta erilaisesta ohjelmointikielestä: laiteläheisestä C-kielestä ja moninaisiin ylläpitotehtäviin soveltuvasta Bourne Again Shell -skriptikielestä. Jonkin verran sivutaan myös symbolista konekieltä.

Yritetäänpä sijoitella tätä kurssia kartalle informaatioteknologian kokonaiskuvassa. Arkipäivää ovat esimerkiksi verkkopankin käyttö, yhteydenpito sosiaalisessa mediassa, digitaalisten muistojen tallentaminen ja jakaminen (esim. valokuvat, videot) tai digitaalisten pelien pelaaminen. Nämä ovat selkeästi informaatioteknologian (eli tietokone- ja tietoliikennelaitteistojen sekä ohjelmistojen) sovelluksia. Puhelu sukulaiselle kulkee nykyään kännykästä kännykkään radiolinkkien ja valokuitukaapelin kautta. Lentokoneiden, laivojen ja autojen ohjaamisesta on tehty täsmällisempää, helpompaa ja turvallisempaa informaatioteknologian avulla. Vuokraamosta lainattu (tai nettipalvelusta ladattu) video katsotaan laitteella, jota ohjaa jonkinlainen tietokone. Universumin salat avautuvat tietokoneella tehtävän laskennan avulla, samoin kuin huomisen päivän sääennuste. Talot, joissa asumme, on suunniteltu tietokoneen avulla. Informaatioteknologian kenttä on laaja, mutta kaikissa tilanteissa on aina mukana sekä ihminen että jonkinlainen, useimmiten toisiin vastaaviin yhteydessä oleva, tietokone. Alalla olemme (sattuneesta syystä) tottuneet käyttämään ihmisestä nimeä "käyttä-



Kuva 0.1: *Kerrokset ja rajapinnat käyttäjän ja tietokonelaitteiston välillä (yksinkertaistus).*

jä”, koska hän käyttää näitä rakentamiamme järjestelmiä. Toisessa päässä kuviota, kauempana arkihavainnosta, on tämän kaiken mahdollistava laitteisto, jonka eräästä olennaisesta komponentista käytämme tuota tietyllä tapaa hassua ilmaisua ”tietokone”. Välissä tarvitaan erinäinen valikoima jotakin, mitä sanomme ”ohjelmistoksi”.

Kuva 0.1 on karkea yleistys ”tasoista” tai kerroksista, joihin käyttäjän ja tietokonelaitteiston välissä oleva osuus kokonaiskuvasta voidaan ajatella jaettavan. Kuva rajoittuu tilanteeseen, jossa yksittäinen käyttäjä hyödyntää yksittäistä sovellusohjelmaa yksittäisellä tietokonelaitteella. Tämä on yleinen ja helposti samaistuttava informaatioteknologian sovellus, joskaan ei missään mielessä ai-

noa. Arkipäivän tutuissa sovelluksissakin käyttäjän tavoite voi olla yhteydenpito muihin käyttäjiin (esim. email, some. . .) palveluntarjoajiin (esim. verkkopankki, musiikkikauppa. . .), jolloin välissä voi olla kilometreittäin verkkoyhteyksiä ja järjestelmään voi kuulua monia tietokoneita ja tietokantoja. Onneksi asiat voidaan opiskella (ja toteuttaakin) yksi pienempi pala kerrallaan. Olipa informaatioteknologian sovelluksen kokonaiskuva tai sen merkitys ihmisen elämässä kuinka laaja tahansa, siihen kuuluu aina olennaisena osana yksi tai useampi sähköllä toimiva tietokone, joka on piirretty kerroksittaisen kuvan alimmalle tasolle. Laitteisto on teknisistä syistä yksinkertainen, ja jotta monimutkaisempien ohjelmistojen olisi helppoa sitä käyttää, tarvitaan tietynlainen erityisohjelmisto, jota sanotaan käyttöjärjestelmäksi. Ylipäätään kerroksittainen rakenne on historian mittaan osoittautunut hyväksi tavaksi tehdä ohjelmistoja. Alempi kerros niin sanotusti tarjoaa palveluja, joita ylempi kerros voi hyödyntää tarjotakseen puolestaan palveluja seuraavalle ylemmälle kerrokselle. Kerrosten väliin määritellyt **rajapinnat** (engl. *interface*) mahdollistavat kerrosten tarkastelun (suunnittelun, toteutuksen, opiskelun) erillisinä kokonaisuuksina. Rajapinta tarkoittaa niitä sovittuja, alemman tason valmistajan määrittelemiä, keinoja, joilla tason palveluita käytetään. Rajapinnan dokumentaatio sisältää käyttöohjeet, joissa kuvaillaan tarkoin kerroksen ominaisuudet ja keinot, joilla ylempi taso voi niitä hyödyntää.

Kuvattua kerrosmaista rakennetta tai sen osaa esim. pohjalta johonkin tiettyyn kirjastoon asti voidaan sanoa ohjelmistopinkaksi engl. *software stack*. Abstraktisti voidaan puhua myös virtuaalikonehierarkiasta, koska jokainen rajapinta tavallaan määrittelee yhdenlaisen virtuaalisen koneen, piilottaen alempien kerrosten "astetta todellisemmat" koneet.

Tämän kurssin esitietona edellytämme jonkinlaisen ohjelmoinnin

peruskurssin, jossa on nähty ensinnäkin jonkin ohjelmointikielen rajapinta: sopimus *syntaksista*, jolla kyseistä kieltä voidaan kirjoittaa, ja *semantiikasta* eli siitä, mitä kirjoitettu ohjelma tarkoittaa tietorakenteiden ja suorituksen kannalta. Lisäksi ohjelmoinnin peruskurssilla on varmasti nähty kurssilla käytössä olleelle ohjelmointikielelle tarjolla olevien *peruskirjastojen* rajapintoja, eli metodeita tai aliohjelmia, joilla tehdään joitakin usein tarvittavia operaatioita – yksinkertaisimmillaan esimerkiksi tekstin tulostaminen kuvaruudulle tai kirjoittaminen tiedostoon. Nykyisellä Ohjelmointi 1 -kurssillamme[2] hyödynnetään C#-kielen ja .NET -alustan peruskirjastojen lisäksi paikallisen yhteisön kehittämää Jypeli-nimistä kirjastoa, jonka avulla voidaan luoda erilaisia kaksiulotteisia pelejä. Jypeli-kirjaston rajapinta määrittelee, miten ja millaisia peliobjekteja voidaan luoda ja miten niiden yhteistoimintaa voidaan koordinoida.

Tällä kurssilla kiinnostuksen kohteena on ensinnäkin käyttöjärjestelmän niin sanottu **ydin** (engl. *kernel*), joka käyttää alempaa tasoa eli fyysistä tietokonejärjestelmää niin sanotun käskykanta-arkkitehtuurin välityksellä ja tarjoaa ylemmälle tasolle eli ”matalan tason kirjasto-ohjelmistolle” niin sanotun järjestelmäkutsurajapinnan. Myös vähintäänkin tuon kyseisen matalan tason kirjasto-ohjelmiston voidaan ajatella olevan osa käyttöjärjestelmän ylöspäin tarjoamaa rajapintaa. Lisäksi rajapinta voi sisältää apuohjelmistoja, erityisesti jonkinlaisen **kuoren** (engl. *shell*) eli komentokielen ja joukon apuohjelmia, joilla käyttäjä voi kaikkein suorimmin päästä käsiksi ytimen hallitsemaan laitteistoon.

Käyttöjärjestelmäohjelmisto on käytännön syistä muodostunut välttämättömäksi välittäjäksi sovellusohjelman ja tietokonelaitteiston välille. Tehdäkseen mitään laitteistoon liittyvää (esim. näppäinpainallusten vastaanottaminen), sovelluksen on kuljettava käyttöjär-

[2] Tilanne keväällä 2016 Jyväskylän yliopiston IT-tiedekunnassa

jestelmän tarjoaman rajapinnan kautta. Käytännössä toki sovellus käyttää kirjastoja, jotka käyttävät alemman tason kirjastoja, ... vasta aivan alimman tason kirjasto hoitaa teknisesti yhteyden käyttöjärjestelmän ytimeen.

Ylöspäin käyttöjärjestelmän ydin tarjoaa ”järjestelmäkutsurajapinnan”, joka ei ole määrittelytavaltaan kovin erilainen kuin kirjastojen toisilleen ja sovellusohjelmille tarjoamat rajapinnat eli julkisten aliohjelmien/metodien nimet, parametrilistat ja luvatut vaikutukset dataan. Teknisen toteutuksen osalta järjestelmäkutsurajapinta on hieman erilainen kuin normaali aliohjelman tai metodin kutsuminen. Tämä kaikkein alin ohjelmistorajapinta tarjoaa (alemman tason kirjastojen välittämänä) sovellusohjelmien ainoat keinot vaikuttaa tietokonelaitteiston osiin ja sen resurssien (laskenta-aika, muisti, syöttö-, tulostus- ja tallennusvälineet) käyttöön.

Käyttöjärjestelmä on ohjelmisto siinä missä muutkin – erotuksella että sillä (ja yksin sillä) on lupa tehdä tiettyjä resurssijakoon liittyviä toimenpiteitä. Tämä ”yksinoikeudellinen lupa” on historian saatossa nähty tarpeelliseksi mm. tietoturvasyiden vuoksi, ja se on suunniteltu sisään syvälle nykyisten tietokoneiden rakenteeseen. Nykyaikaisen laitteiston ylöspäin tarjoama rajapinta (sanotaan tätä karkeasti vaikkapa ”käskykanta-arkkitehtuuri” tai ehkä mieluummin ”prosessoriarkkitehtuuri”) tukee suoraan nykyaikaisen käyttöjärjestelmäohjelmiston tarvitsemia erivapauksia.

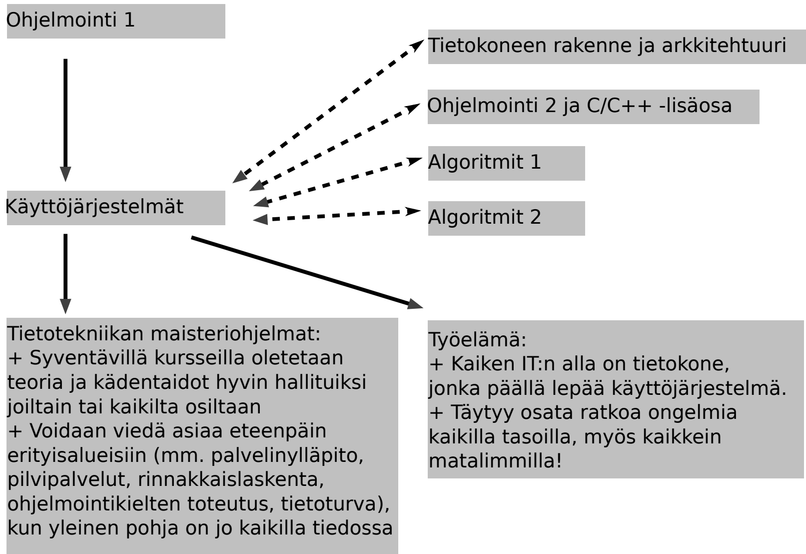
Joidenkin kurssin osaamistavoitteiden arvostaminen saattaa vaatia jonkin verran perustietoa, jota ei vielä tähän johdantoon mahdu – miksi esimerkiksi usean ohjelman toimiminen samaan aikaan on jotenkin mainitsemisen arvoista, kun käytännössä ”tarvitsee vain klikata ne kaikki ohjelmat käyntiin”... Jotta peruskäyttäjälle itsestäänselviä asioita (eli asioita, jotka *käyttäjälle tulee tarjota et-*

tä hän on tyytyväinen) osaisi arvostaa ohjelmien ja tietojärjestelmien toteuttajan näkökulmasta, täytyy ymmärtää jonkin verran siitä, millainen laite nykyaikainen tietokone oikeastaan on. Liite .1 tarjoaa “pullantuoksuisen pehmojohdannon”, joka on ehdottoman suositeltavaa alkulukemistoa, mikäli tekninen lähestymistapa pelottaa. Liitteessä käydään läpi tämän kurssin sisältöä yllättävän pitkälle menemättä lainkaan teknisiin yksityiskohtiin.

Varsinainen asia alkaa luvusta 0.2, joka kuvailee tiivisti sen fyysisen tietokonelaitteiston olennaiset piirteet, jota käyttöjärjestelmillä halutaan komentaa. Luvun alku on kurssin Tietokoneen rakenne ja arkkitehtuuri ”kertausta” tai ”ennakkospoileria”, riippuen siitä, onko kurssi jo käyty vai ei. Samoin käydään läpi joidenkin muiden kurssien sisältöä niiltä osin kuin myöhempien lukujen seuraaminen vähimmillään vaatii. Kuvaan 0.2 on merkitty muita kurssejamme, joihin käyttöjärjestelmät vahvasti liittyvät. Ilman Ohjelmointi 1 -kurssin hyvää hallintaa tämän kurssin asiat menevät todennäköisesti ohi ”korkealta ja kovaa”. Kuvaan on myös merkitty päämäärät: Tämän kurssin sisältö ja kädentaidot oletetaan kokonaisuudessaan tunnetuksi ainakin tietotekniikan maisteriohjelmien syventävillä kursseilla. Työelämärelevanssiakin tällä on, koska kurssin tietojen pohjalta on mahdollisuus tarttua nopeammin sellaisten ongelmien ratkaisemiseen, jotka liittyvät alimpiin ohjelmistokerroksiin.

Esitietojen varmistuksen jälkeen luku 0.3 käy läpi teknisiä näkökulmia ohjelmien tekemiseen ja suorittamiseen tietokoneessa. Luku 0.4 kuvailee konekieltä, joka on viime kädessä ainoa rajapinta, jonka kautta tehtaalta toimitettua tietokonelaitteistoa on mahdollista komentaa[3]. Luvussa 0.5 käydään täsmällisesti läpi yksit-

[3]Esitietokurssin ”Tietokoneen rakenne ja arkkitehtuuri” käyneille ensimmäisten lukujen asiat lienevätkin jo tuttuja. Johtopäätös tulee olemaan yksinkertaistettuna, että tietokone on ”tyhmä kasa elektroniikkaa”, jolla ei käytännössä voi tehdä mi-



Kuva 0.2: *Käyttöjärjestelmien suhde muihin kursseihin. Ohjelmointi 1 on välttämätön esitieto. Muut mainitut auttavat, mutta niiden sisällöstä sovelletaan vain joitain osia.*

täisen ohjelman konekielinen suoritus ja virtuaalimuistiavaruuden käyttö yhden prosessoitavan ohjelman näkökulmasta. Luvussa 0.6 käydään tietotekniikan historian kautta läpi ohjelmien suorittamiseen liittyviä tavoitteita ja haasteita, joiden ratkaisuna nykyaikaisen käyttöjärjestelmän piirteet ovat syntyneet. Luku 0.7 käsittelee käyttöjärjestelmän ja fyysisen laitteiston kättelypintaa eli keskeytysjärjestelmää ja käyttöjärjestelmän kutsurajapintaa konekielen tasolla. Luvussa 0.8 ryhdytään puhumaan omassa muistiavaruudessaan suorituksessa olevasta ohjelmasta prosessina, käydään läpi abstraktiin prosessin käsitteeseen liittyviä ominaisuuksia sekä tapoja, joilla ominaisuudet voidaan käytännössä toteuttaa käyttöjärjestelmän ohjelmakoodissa. Luvussa 0.9 havaitaan yhdenaikaisten prosessien tai säikeiden käytön hyötyjä sekä tällaisesta yhdenai-

---

tään hyödyllistä ilman ”jotakin systeemiä”, joka merkittävästi helpottaa elektroniikan käyttämistä. Luonnollinen nimi ”systeemille” eli ”järjestelmälle”, joka helpottaa elektroniikan käyttämistä, voisi olla esimerkiksi . . . ”käyttöjärjestelmä”.

kaisuudesta mahdollisesti aiheutuvia ongelmia ratkaisuineen. Viimeisissä luvuissa käydään läpi muutamia tärkeimpiä yksityiskohtia ja menettelytapoja, jotka liittyvät käyttöjärjestelmän tärkeimpiin osajärjestelmiin: luku 0.10 käsittelee muistinhallintaa, luku 0.11 syöttö- ja tulostuslaitteita ja luku 0.12 tiedostojärjestelmää. Luvussa 0.13 tehdään aiempien lukujen pohjatietoihin vedoten huomioita suunnitteluperusteista ja kompromisseista, joita käyttöjärjestelmän ja sen osioiden suunnitteluun liittyy. Luku 0.14 mainitsee sanan tai pari skripteistä, jotka kurssilla varsinaisesti käsitellään demotehtävien kautta.

0.2 Esitietoja

Osaamistavoitteet: Luvun luettuaan opiskelija:

- tietää, mitä hänen tulisi jo osata ennen tämän kurssin loppuosan seuraamista; osa asioista on tullut vastaan kaikille pakollisella esitietokurssilla Ohjelmointi 1, mutta osa käydään tässä välttämättömiltä osin läpi uutena asiana niille, joilla Ohjelmointi 1:n lisäksi muita suositeltuja esitietoja ei mahdu sivuaineopintopakettiin. Näiltä kursseilta (Ohjelmointi 2, Tietokoneen rakenne ja arkkitehtuuri, Algoritmit 1) tarvitaan vain muutamaa erityisasiaa, joita voidaan tällä kurssilla käsitellä soveltaen tai ennakoivasti, ilman mainituilla kursseilla tarjottavaa teoriapohjaa.

Tässä luvussa esitellään joitakin esitietoja, joita myöhempien lukujen ja kurssin käytännön esimerkkien täysipainoinen ymmärtäminen luultavasti edellyttää. Mikäli missään näistä esitiedoista on puutteita, on vaikea hahmottaa, kuinka kurssin myöhemmistä luvuista, luennoista tai demotehtävistä voisi olla hyötyä. Ymmärryksen syntyminen on lopulta tärkeintä, ja se voi syntyä vain aiempien palasten päälle.

Aivan alkupuolella, luvussa 0.2 muistellaan avainsanojen tasolla, mitä ensimmäisellä ohjelmointikurssilla on opittu ohjelmien suunnittelusta ja kirjoittamisesta rakenteisella ohjelmointikielellä. Seuraavaksi luvussa 0.2 käydään pintapuolisesti läpi abstrakteja tietorakenteita, joita käyttöjärjestelmän toiminta ja sen ymmärtäminen edellyttää. Lopuksi luvussa 0.2 esitellään tietokoneen fyysistä rakennetta ja toimintaperiaatteita, jotta voidaan tarkemmin nähdä, miksi käyttöjärjestelmää tarvitaan, mihin se sijoittuu tietotekniikan kokonaiskuvassa ja millaisia tehtäviä käyttöjärjestelmän

tulee pystyä hoitamaan. Kuvaus on pääosin tiivistelmä ja tavallaan kertaus suositeltuna esitietokurssina olleesta kurssista Tietokoneen rakenne ja arkkitehtuuri. Tässä ei mennä kovin syvälle yksityiskohtiin tai teoriaan, vaan käsitellään aihepiiriä pintapuolisesti käyttöjärjestelmäkurssin näkökulmasta. Laitteiston esittelyn yhteydessä käydään läpi myös digitaalilogiikkaan liittyviä lukujärjestelmiä ja niiden notaatioita, joita on kyllä ehkä sivuttu Ohjelmointi 1 -kurssilla, mutta todennäköisesti kevyemmin kuin tällä kurssilla tarvitaan.

Pari sanaa ohjelmoinnista (välttämätön esitieto)

Oman tiedekuntamme kurssi Ohjelmointi 1 tai jokin vastaava ohjelmoinnin peruskurssi on ehdoton esitieto tämän kurssin seuraamiselle. Kertaa siis mahdollisimman pian, mikäli erityisesti seuraavat asiat (poimittuna suoraan kevään 2015 Ohjelmointi 1 -kurssin sisällysluettelosta[4]) eivät ole vähintään käsitteen ja C#/Java/C++/C -kielisen lähdekoodin kirjoittamisen tasolla tuttuja. Hakasulkeissa on lisätty kommentteja suhteesta tähän kurssiin:

- Ohjelman **kääntäminen** ja **ajaminen** [ainakin Visual Studiossa tai muussa IDE:ssä; tällä kurssilla tehdään samoja asioita komentoriviltä. Myös useissa IDEissä käännös tapahtuu konepellin alla loppujen lopuksi komentorivillä, jossa IDE:n graafisessa asetusikkunassa säädettyjen muutosten perusteella kääntäjäohjelmalle annetaan käännöskomennon yhteydessä argumentteja, jotka vaikuttavat siihen, millainen lopputuote pullahtaa ulos. Esimerkiksi julkaistava lopputuote tehdään eri argumenteilla kuin debugattava kehitysversio.]

[4] https://trac.cc.jyu.fi/projects/ohj1/wiki/sisallys

- Ohjelman rakenne: nimiavaruus, luokka, **pääohjelma**, **aliohjelmat** [tämän kurssin työkaluilla periaatteessa ei ole käytettävissä nimiavaruuksia eikä luokkia, mutta pää- ja aliohjelmia on; eli siinä mielessä on jopa hiukan yksinkertaisempaa kuin Ohjelmointi 1:llä:)]

- Algoritmeista: Algoritminen ajattelu, Tarkentaminen, Yleistäminen, Algoritmin kirjoittaminen ja suunnittelu [liittyy **kaikkeen ohjelmien tekoon**, mm. käyttöjärjestelmiin]

- Kirjastot [käsitteen tasolla]

- Aliohjelmat: **Kutsuminen**, Kirjoittaminen [+erityisesti **parametrien välitys** kutsun yhteydessä]

- [Muuttujat]: Muuttujan määrittely, [esim. C#:n] alkeistietotyypit, Nimeäminen, Arvon asettaminen muuttujaan, Näkyvyys [erityisesti erot luokkamuuttujan, aliohjelman lokaalin muuttujan ja aliohjelman parametrien välillä], Vakiot, Aritmeettiset lausekkeet

- Oliotietotyypit: Mitä oliot ovat, Luominen, **Oliotietotyypit vs alkeistietotyypit**, Metodin kutsuminen. [Olion tuhoaminen ja roskienkeruu hyvä tietää; käyttöjärjestelmissä puhutaan näitä sivuavista asioista jonkin verran laiteläheisestä muistinhallinnan näkökulmasta]

- **Aliohjelman paluuarvo**

- **Debuggaus** [eli mistä debuggauksessa on kyse; mm. mitä tarkoittaa ohjelman askeltaminen rivi kerrallaan, miten asetetaan breakpointteja ja tarkastellaan muuttujien/olioiden sisältöä ohjelman ollessa pysäytettynä; tällä kurssilla debugataan tekstipohjaisella gdb-ohjelmalla, jossa tehdään olennaisesti ihan samoja ykköskurssilta tuttuja asioita!]

- Syntaksivirheiden etsintä, Koodin täydennystyökalut ja koodimallit [napatkaa tästä idea tehokäytön tavoiteltavuudesta, näppäinyhdistelmistä, ympäristön lisäasetuksista ym.; tällä kurssilla emme käytä IDEä, mutta vastaavalla tavalla opetellaan bashin ja tekstieditorien tehokäyttöä]

- **Merkkijonot** [muistettava perusidea ”jonosta, joka muodostuu peräkkäisistä merkeistä” sekä lähdekoodiin kirjoitetun merkkijonoilmaisun (engl. *string literal*) kirjoittamisesta esim. C#:ssa lainausmerkkien väliin; tällä kurssilla käydään läpi merkkijonon mahdollista toteutusta laitteistotasolla ja erityisesti C-kielessä; shellin käytössä on ymmärrettävä merkkijonon rooli myös]

- **Ehtolauseet**: **`if`**, **`if-else`**, **`if-else if-...-else`**, **`switch-ca`** Vertailuoperaattorit. Loogiset operaatiot.

- **Taulukot**: Luominen, **Alkioon viittaaminen**, Moniulotteiset taulukot

- **Toistorakenteet**: **`while`**, **`do-while`**, **`for`**, **`for`**`-each`, Sisäkkäiset silmukat, **`break`**- ja **`continue`** -lauseet, *ikuinen silmukka* [näiden on oltava tarkasti selvillä käsitteen ja jonkin rakenteisen ohjelmointikielen koodin tasolla; tällä kurssilla sovelletaan samaa C-kielen ja bash-skriptikielen syntakseilla]

- **Lukujen esitys tietokoneessa**, ASCII-koodi (hmm, joko UTF pitäisi lisätä mukaan?)

- Rekursio [Tällä kurssilla on mahdollista saada oivallinen konkreettinen esimerkki, jolla aiempaa alustavaa ymmärrystä rekursiosta saataisiin ehkä syvennettyä; toivottavasti ehditään näkemään sellainen kurssin mittaan]

- Dynaamiset tietorakenteet [olennaista on idea dynaamisuudessa eli vaihtuvakokoisuudessa; tällä kurssilla tätä konkretisoidaan muistin tarpeen ja muistinhallinnan kannalta]

- (Tällä nimenomaisella kurssilla tarvitsemme todennäköisesti hyvin vähän seuraavia Ohjelmointi 1:n sinänsä tärkeitä osioita: Merkkijonojen pilkkominen, Järjestämisalgoritmi, Valmiit järjestysalgoritmit, Poikkeukset, Olioluokkien dokumentaatio)

  [Paitsi tarkemmin ajatelle. . . merkkijonoja ehkä tullaan pilkkomaan shellin työkaluilla tavalla tai toisella, työkaluohjelmaa `sort` käytetään jo ensimmäisessä demossa syöterivien leksikografiseen järjestämiseen, poikkeusten lähtökohtainen syy voi useimmitenkin olla käyttöjärjestelmän hallinnoimassa laitteistossa, . . . ja mitä tulee *dokumentaatioon*, niin tällä kurssilla silmäillään vähintään läpi POSIXia ja AMD64 -prosessoriarkkitehtuuria, joten kaukana ei tosiaankaan olla dokumentaation lukemisesta (ja omat ohjelmathan tulee aina dokumentoida tarpeen vaatimalla tavalla)]

Kerratkaa, kerratkaa, kaikki edellä mainitut kohteet ohjelmointi 1:n kurssimateriaalista tarvittaessa. Jatkossa oletetaan edellämainitut asiat tunnetuiksi *sillä tasolla, millä ne on ehditty esitietokurssilla harjaannuttaa!* – Aivan luonnollista on, että tässä vaiheessa käsitteet saattavat olla vielä sekaisin ja asiat levällään. Kokemus on tämän realismin näyttänyt, ja opettajat ymmärtävät sen kyllä. Olemme valmistautuneet auttamaan myös esitietojen kertaamisessa (ja mahdollisesti ymmärtämisessä ensimmäisen kerran, jos niikseen tulee. . .). Rohkeasti eteenpäin vain – kokonaisuus hahmottuu asia kerrallaan!

Konkreettisia ja abstrakteja tietorakenteita

Seuraavassa tietorakenteiden esittelyssä tulee ensimmäistä kertaa vastaan tärkeä termi **muistiosoite** (engl. *memory address*), jolle haetaan tarkempaa ymmärrystä pitkin kurssia, kun puhutaan enemmän C-ohjelmoinnista, konekielestä ja käyttöjärjestelmän vastuulla olevasta muistinhallinnasta. Tämän aliluvun tarpeisiin riittää mielikuvamalli, jossa ymmärrät muistiosoitteen olevan kokonaisluku, joka ilmoittaa yhden pienen tietoalkion sijainnin tietokoneessa. Ajattele vaikka ruutupaperia, johon mahtuu jokaiseen ruutuun yksi kaksinumeroinen luku (siis 00–99). Ensimmäisen muistiruudun osoite on 0, seuraavan 1, sitä seuraavan 2 ja niin edelleen. Paperiarkissa on vain tietty määrä ruutuja, sanotaan N kappaletta, jolloin viimeisen paperiin mahtuvan tiedonmurusen osoite on $N - 1$. Peräkkäiset osoitteet vastaavat siis paperissa olevia peräkkäisiä ruutuja, joissa olevia tietoja voidaan osoittaa osoitenumeroinnin avulla. Mielikuvamalli tietokoneen muistista ja muistiosoitteista tarkennetaan kurssin mittaan teknisesti oikeellisemmaksi, mutta se vaatii pohjustusta enemmän kuin tähän esitietolukuun mahtuu.

Alkeistyyppinen data, oliot, tietorakenteet

Edeltävän ohjelmointikurssin pohjalta oletetaan täysin selväksi, miten yksittäistä muuttujaa tai tietoalkiota käsitellään. Puhutaan siis esimerkiksi seuraavanlaisesta ohjelmakoodista jollakin yleisesti tunnetulla tyypitetyllä ohjelmointikielellä kirjoitettuna:

```
int a = 10;  // muuttujan esittely ja arvon alustaminen
a = 5;       // arvon asettaminen uudelleen.
a = 3;       // ja taas uudelleen, peräkkäin.
```

Tässä esimerkissä annetaan nimi `a` muuttujalle, jonka tyypiksi tulee kokonaisluku (engl. *integer*) eli **`int`** ja arvoksi 10 koodinpätkän

alussa. Koodia suoritetaan C:n tapaisissa kielissä, kuten C++:ssa, C#:ssa tai Javassa, peräkkäin, jolloin edellisten koodirivien jälkeen muuttujan `a` sisältö tulisi olemaan 3. Näissä kielissä suoritusjärjestystä muutellaan tarvittaessa kirjoittamalla ohjelmaan kontrollirakenteita, kuten ehtoja, silmukoita, aliohjelmakutsuja ja muita Ohjelmointi 1:ltä tuttuja syntaktisia rakenteita. Yksittäisen lohkon sisällä suoritus kulkee yleensä riviltä toiselle ylhäältä alaspäin.

Muuttujan tyyppi voi olla muutakin kuin kokonaisluku, esimerkkejä:

```
  double pi_oma_arvioni = 3.1416;  // tarkahko liukuluku (des
  string otsikko = "Kertomuksiani␣ohjelmoinnista"; // mer
  MonimutkainenVekotin v = new MonimutkainenVekotin(17);
// Vekotin
```

Esimerkin ensimmäinen rivi käsittelee liukulukutyyppiä, joka voi olla kokonaisluvun tapaan yksinkertainen, tietyllä tapaa koodattu datankappale, esim. 64 peräkkäistä bittiä, jotka kuvaavat reaalilukua mahdollisimman läheisesti vaikkapa IEEE 754 -standardin mukaisen koodauksen mukaan. Jälkimmäiset kaksi esimerkkiä ovat kuitenkin mm. C#:ssa ”vaikeampia” tapauksia[5]: Ne ovat **viitteitä** (engl. *reference*) tiettyyn **luokkaan** (engl. *class*) kuuluviin **olioihin** (engl. *object*), joiden tarvitsema tallennustila on varattu omasta muistialueestaan, ns. **keosta**. Olion tilaa käsitellään luokan määräämiä menettelytapoja eli **metodeja** (engl. *method*) kutsumalla. Luokat määritellään yksinkertaisimmillaan alkeistietotyypeistä ja olioviitteistä koostamalla seuraavan esimerkin kaltaisesti:

```
class Kokonaislukupari {
  int a = 10;   // kenttä nimeltä "a", oletusarvo 10
```

[5]Ainakin ne tekevät tämän monisteen kirjoittajalle tyyppien kertaamisen lyhyin sanamuodoin kovin hankalaksi, kun ei voida puhua kaikesta yhtäläisesti oliona, vaan on erikseen primitiivitietotyypit ...

```
  int b = 5;    // kenttä nimeltä "b", oletusarvo 5
}

class Henkilotiedot {
  int syntymavuosi;
  string nimi;
  string tyopaikka;
}
```

Esimerkissä on käytetty jonkinlaisen C#-maisen oliokielen syntaksia määrittelemään kaksi erilaista **tietorakennetta** (engl. *data structure*), joita tosiaan oliokielessä sanotaan luokiksi. Ensimmäinen näistä kuvaa kokonaislukuparia, esimerkiksi ruutupaperin yhden ruudun sijaintia ruutupaperin vasemmasta yläkulmasta lukien (”10 ruutua oikealle ja 5 alaspäin, niin olet perillä”). Jälkimmäinen voisi kuvailla yhden henkilön henkilötietoja jonkin sovelluksen tarpeeseen.

Informaation piilottamista, perintää, rajapintoja ja muita oliokielten herkkuja varten tarvitaan menettelyjä, joihin voi syventyä erikseen muilla kuin tällä kurssilla. Ohjelmointi 2 kertoo perusasiat sovellusohjelmoijan näkökulmasta ja niiden hiomista jatketaan esimerkiksi kursseilla Graafisten käyttöliittymien ohjelmointi ja Websovellukset. Olio-ominaisuuksien toteuttamiseen kääntäjäteknologian näkökulmasta meillä on (JY/kevät 2015) kokonainen maisteriohjelma nimeltä Ohjelmointikielten periaatteet, jossa saa lopulta vaikka gradun verran tutkia moniperinnän ihmeellistä maailmaa tai automaattista koodin generointia suunnitteludiagrammien perusteella. . . *Tällä kurssilla päästään helpommalla*, koska näitä oliomallinnuksen hienouksia ei tosiaankaan tarvitse juuri nyt miettiä.

Tämän kurssin *oliot* ovat pääasiassa yksinkertaisia, kiinteän mittaisia datapötköjä, joiden sisäinen rakenne määritellään C-kielessä

seuraavankaltaisesti (toimivaksi varmistettuja, C99-standardin mukaisia koodiesimerkkejä tulee sitten demoissa):

```
struct Kokonaislukupari {
  int a;    // kenttä nimeltä "a"
  int b;    // kenttä nimeltä "b"
}

struct Henkilotiedot {
  int syntymavuosi;
  char nimi[15];
  char tyopaikka[15];
}
```

Ero edelliseen olioesimerkkiin on, että luokan (**`class`**) sijasta puhutaankin yksinkertaisemmasta konseptista eli tietueesta (**`struct`**), joka luokan kaltaisesti määrittelee nimiä ja tyyppejä olion ominaisuuksille. Kuitenkaan esimerkiksi metodeja ei voisi määritellä, eikä väkisin piilottaa tietoja rajapinnan taakse (ts. **`private`** / **`protected`** -ominaisuudet eivät ole olemassa vaan kaikki on **`public`** eikä muuta voi), eikä uusia ominaisuuksia voisi helposti lisätä muuten kuin muokkaamalla rakenteen alkuperäistä määritelmää (perintää, engl. *inheritance* / engl. *extends*, tai luokkarajapintoja, engl. *implements* tmv., ei ole mukana kielessä).

Kokonaislukuparihan määritellään tässä olennaisesti samoin kuin aiemmin. Henkilötietojen merkkijonosisältö määritellään kuitenkin nyt kiinteän mittaiseksi: Nimelle ja työpaikalle on varattu tasan 15 merkkiä, joten pidemmät merkkijonot on esimerkiksi katkaistava keskeltä poikki. Sisältönä tässä rakenteessa voisi olla esimerkiksi seuraavaa[6]:

[6]Tällaisenaan merkkijonot eivät olisi C-kielessä valideja, koska lopusta puuttuu merkkijonon päättävä nk. nollamerkki; luentoesimerkeissä ja demossa tarkennetaan.

```
  1979, "Paavo Juhani Ni", "JY              "
```

Tällainen rakenteisen datan tallennusmuoto ei selvästi ole nykypäivän standardein ajateltuna ollenkaan riittävän monipuolinen – mm. mielivaltaisen pitkiä merkkijonoja ei voi käyttää. Toisaalta lyhyen merkkijonon, kuten esimerkissä ”JY”, säilömiseen riittäisi vähempikin tila kuin mitä nyt on varattu. Onneksi sovellusohjelmia tehdään nykyään oliokielillä, joiden alustakirjastot tukevat mm. merkkijonojen käyttöä monipuolisesti!

Käyttöjärjestelmät saatetaan kuitenkin mm. tehokkuussyistä tehdä ns. matalamman tason kielillä, esimerkiksi ”1970-lukulaisella” C-kielellä. Se, että C on vuosikymmeniä vanha, ei tarkoita sitä, että se olisi hyödyttömäksi havaittu ohjelmointikieli – sitä myös kehitetään alati. Uusin standardiversio C18[7] on vuoden 2018 paikkeilta. Uusin POSIX-standardi edellyttää vuonna 1999 kiinnitetyn C99-standardin mukaisen C-kääntäjän löytymistä järjestelmästä (ja käynnistymistä shell-komennolla `c99`). Standardointisyistä keväästä 2015 alkaen tämän kurssin esimerkit yrittävät noudattaa C99-standardia. *Ei ole kovin nopeaa lukea tarkoin noita monituhatsivuisia standardeja, joten valjastetaan nyt myös kaikkien kurssilaisten silmät varmistamaan standardinmukaisuus ja ilmoittamaan mahdollisista standardirikkomuksista, jotta ne saadaan korjattua! Osa kurssilaisista saattaa olla näiden speksien kanssa tekemisissä päivittäin, kun taas luennoitsija on vain pari kuukautta kerran vuodessa ... vasta-alkajat ottakoon standardit kuitenkin vain silmäilyn tasolla, koska kurssin aika on rajallinen! Syventymiseen on aikaa koko loppuelämä.*

Esim. tietokannassa tai tällaiseksi sovitussa tiedostoformaatissa tilanne voisi kuitenkin olla tämänkin esimerkin kaltainen.

[7] Virallisemmin ISO/IEC 9899:2018.

Käyttöjärjestelmäkurssin osalta C-kielen ja kiinteän kokoisten datamöhkäleiden käyttö selkeyttää asioita – päästään nimittäin näkemään yksinkertaisesti, mutta konkreettisesti, miten rakenteisesti organisoitu data näyttäytyy tietokoneen muistissa. Silloin, kun kiinteänkokoiset datamöhkäleet riittävät sovelluksen tarpeisiin, ovat ne myös kaikkein nopeimpia. Laitteiston, mukaanlukien muistin, luonne ja käskyttäminen on käyttöjärjestelmän tehtävä ja siten tämän kurssin ydinasiaa! Siitä kertyy myös ymmärryspohja, jonka kautta myöhempien, korkeamman abstraktiotason kurssien seuraaminen on toivon mukaan miellyttävämpää tai ainakin vähemmän mystistä.

Siitä huolimatta, ja osittain sen ansiosta, että esimerkeissä käytetään laiteläheistä “matalan abstraktiotason” kieltä eli C:tä, tällä kurssilla saadaan toivon mukaan mm. vahva selvyys siihen, miten kaikki data täytyy loppuviimein olla tallennettuna tietokoneen muistissa, miten uusille olioille (tai ylipäätään uudelle datalle) täytyy konkreettisesti varata uutta tilaa muistista ja miten oliot voivat kaikkein yksinkertaisimmillaan viitata toisiin muistissa oleviin olioihin. C-kielen ja assemblerin katseleminen tekee tällaisten tarkastelujen tekemisen helpoksi välttämättömän ja toisaalta helposti nähtäville tulevan konkretian kautta. Prosessorin ja muistin toiminnan ymmärtäminen mahdollistaa myös jämerän ymmärryksen syistä, joiden takia yhdenaikaisten ohjelmien käyttö voi johtaa kilpa-ajotilanteisiin ja sitä kautta lukitusten tarpeeseen.

Perintää ja tiedon piilotusta ei helposti saada, mutta C-kielen tietorakenteeseen voidaan kyllä erittäin helposti määritellä *viitteitä* muihin rakenteisiin, kuten seuraavassa esimerkissä::

```
struct Henkilotiedot {
  int syntymavuosi;
  char *nimi;
  char *tyopaikka;
```

```
}
```

Nyt mielivaltaisen pitkät nimet olisivatkin mahdollisia! Tähtimerkki eli asteriski tietorakenteen kenttien `*nimi` ja `*tyopaikka` edessä tarkoittaa, että kyseessä on **osoitin** (engl. *pointer*) jossakin muualla sijaitsevaan dataan. *Idea on periaatteessa täysin sama kuin olioviitteissä.* Toteutus vain on sangen yksinkertainen ja tietokoneen perustoiminnan mukainen: Viimeisen esimerkin mukaisen tietorakenteen määrittelemässä oliossa on tasan kolme kokonaislukua: ensimmäinen on henkilön syntymävuosi, kaksi jälkimmäistä kokonaislukua ovat muistiosoitteita eli kertakaikkiaan sijaintipaikkoja tietokoneen muistissa, joihin voidaan varata tilaa vaikka miljoonan merkin mittaiselle merkkijonolle. Loppuun asti ei tarvitse asiaa ymmärtää vielä, mutta se on *tavoite hahmottaa demojen ja luentoesimerkkien kautta heti, kun mahdollista.*

Taulukot

Selvää aiemman kurssin pohjalta tulee olla muuttujista muodostettujen **taulukoiden** (engl. *array*) kayttö ohjelmissa. Esimerkiksi:

```
int[] lottorivi = {7, 13, 22, 24, 30, 31, 32};
// Hups, kun onnennumeroni ei olekaan 32 vaan 36:
lottorivi[6] = 36;
```

Taulukossa on peräkkäin samantyyppisiä tietoyksiköitä, joihin päästee käsiksi indeksin perusteella. Esimerkissä tehdään ensin seitsemän kokonaislukua sisältävä taulukko, jossa on lottorivi. Sitten taulukon viimeiseen alkioon sijoitetaan uusi arvo. Taulukon olennainen piirre on, että se on kiinteän mittainen ja jokaiseen alkioon päästään käsiksi indeksillä (suoraan, kulkematta kiertoteitä eli todennäköisesti myös salamannopeasti). Indeksit alkavat C-mäisissä kielissä, mm. C#:ssa, nollasta, joten `lottorivi[6]` tarkoittaa seit-

Lottorivi, “int lottorivi[7]”

Indeksi	0	1	2	3	4	5	6
Data	7	13	22	24	30	31	32

Esimerkiksi pätee:
lottorivi [3] == 24

Viisi sijaintia kokonaislukukoordinaatistossa, “Kokonaislukupari pari[5]”

Indeksi	0	1	2	3	4
Data	7	12	8	1	0	0	7	5	1	62

Esimerkiksi pätee:
pari[0].a == 7;
pari[0].b == 12;
pari[4].a == 1;
pari[4].b == 62;

Kuva 0.3: *Kiinteänmittaisia taulukoita sisältöineen*

semättä alkiota ensimmäisestä eteenpäin laskien. Varmistu asiasta kynällä ja paperilla tarvittaessa.

Kuvassa 0.3 on esimerkit kahdesta taulukosta, jotka on koostettu kiinteänmittaisista tietotyypeistä. Ensimmäiseksi on kokonaislukutaulukko, joka kuvaa lottoriviä. Jokainen alkio on kokonaisluku. Yksinkertaisimmillaan taulukko voidaan tallentaa tietokoneen muistissa seitsemään peräkkäiseen kokonaisluvun mittaiseen tilaan (esim. 32 bittiä jokaista kokonaislukua kohden). Toisena esimerkkinä on taulukko, jossa on peräkkäin kahdesta kokonaisluvusta muodostettuja pareja. Kumpikin kokonaisluku vaatii esimerkiksi 32 bittiä tallennustilaa, joten jokainen kokonaisuus vaatii $2 \times 32 = 64$ bittiä. Koko taulukko voidaan tallentaa peräkkäisiin sijainteihin muistissa, samoin kuin lottorivikin. Kukin viidestä alkiosta on nyt kuitenkin tietorakenne, jossa on kaksi 32-bittistä kokonaislukua. Yhteensä se vaatii muistia siis $5 \times 2 \times 32$ bittiä.

Kiinteän mittaisten C-kielisten tietorakenteiden tallennus muistiin kiinteän mittaisena taulukkona on selkeätä: ensimmäisen elemen-

tin kenttien arvot sijaitsevat peräkkäin ensimmäisestä muistipaikasta alkaen. Sen jälkeen tulevat samassa sisäisessä järjestyksessä kaikkien muiden taulukossa sijaitsevien elementtien kenttien arvot. Tällaisessa taulukossa kaikki elementit ovat samantyyppisiä, joten ne ovat myös keskenään saman mittaisia. Taulukon alusta voidaan siis erittäin helposti laskea, mikä on sen muistipaikan osoite, josta tietyn elementin tiedot alkavat:

```
elementin_alun_osoite =
    ensimmaisen_elementin_osoite
    + elementin_indeksi * elementin_koko
```

Abstrakteja tietorakenteita, joita tällä kurssilla tarvitaan

Edellä mainitut alkeistietotyypit, alkeistietotyyppien ja mahdollisesti muistiosoitteina toteutettujen viitteiden muodostamat yhdistelmätietorakenteet sekä edellämainituista muodostuvat indeksoidut taulukot ovat kaikkein yksinkertaisimpia tapoja tiedon säilyttämiseen ja käsittelemiseen tietokonelaitteistossa. Ne otetaan tällä kurssilla selkäytimeen konkretian kautta C- ja konekielidemoissa. Lisäksi esimerkiksi tämän kurssin kaikkein keskeisimmät tietorakenteet, prosessielementti ja niistä muodostuva prosessitaulu voitaisiin toteuttaa jopa näillä erittäin simppeleillä rakenteilla. Arvosanan 1 perusvaatimukset ylittävänä päämääränä kurssilla on ymmärtää jotakin esim. oikean Linux-ytimen tietorakenteiden määrittelystä ja hyödyntämisestä tutkimalla konkreettista lähdekoodia.

Monissa sovelluksissa, mukaanlukien käyttöjärjestelmän joidenkin osioiden toiminta, tarvitaan kuitenkin aavistuksen verran monimutkaisempia tietorakenteita kuin pelkät peräkkäiset datapötköt. Tarkempi teoria monipuolisempiin tietorakenteisiin sekä niiden käsittelyyn käydään läpi jatkokursseilla nimiltään Algoritmit 1 ja 2. Tällä kurssilla pärjätään, kun ymmärretään vähintään käsitteelli-

sellä ja kaaviokuvien tasolla aivan muutama ns. **abstrakti tietorakenne** (engl. *abstract data structure*). Teoriastakaan ei tarvitse toistaiseksi paljoa tietää, koska asiat ovat käyttöjärjestelmien osalta pitkälti arkijärjellä ymmärrettävissä[8].

Termissä **abstrakti tietorakenne** osuus *abstrakti* tarkoittaa sitä, että tällaisella tietorakenteella halutaan yleistää eli abstrahoida jokin ilmenevä tai toivottu käyttäytymis- tai toimintamalli riippumatta siitä, millaisia olioita kyseiseen käyttäytymiseen osallistuu. Abstrakteista rakenteista tällä kurssilla tullaan käsittelemään ainakin kolmea: jono, pino ja puu. Käydään seuraavaksi läpi olennaisten piirteiden, reaalimaailman analogioiden, kaaviokuvien ja tähän kurssiin liittyvien ennakoivien mainintojen kautta läpi jokainen näistä. Kuvassa 0.4 näkyy ensimmäisenä lista, jonka erityistapauksia kuvassa myös esitettyjen jonon ja pinon voidaan ajatella olevan.

Lista (engl. *list*) on tietorakenne, joka kuvaa järjestettyä, vaihtuvankokoista, joukkoa joitakin olioita. Jos lista ei ole tyhjä, sen pää (engl. *head*) viittaa listan ensimmäiseen alkioon. Listan muut alkiot kuin ensimmäinen muodostavat listan hännän (engl. *tail*). Vaikkei välttämätöntä, on näppärää pitää yllä myös viitettä loppuun, eli listan viimeiseen elementtiin (engl. *last*). Reaalimaailman esimerkki on helppo löytää vaikkapa suomenkielisestä lauseesta ”Luentosalista löytyivät tiistaina henkilöt Tomi, Paavo, Seppo ja Jarno”. Kevään 2015 eräällä luennolla nimettyjä, salista löytyneitä henkilöitä on tässä listattu suomenkielisenä järjestettynä listana, jossa Tomi on keulassa eli kirjoitusjärjestyksessä ensimmäisenä

---

[8]Tervemenoa jatkokursseille, mikäli asia kiinnostaa tarkemmin; tietoteknikoillahan algoritmien analysointi kuuluu pakolliseen yleissivistykseen. Teoriaa ei ole missään nimessä syytä aliarvioida, koska juuri siihen perustuu mm. erilaisten toteutustapojen paremmuuden vertailu tehokkuuden, skaalautuvuuden ym. reaalimaailmassa tärkeiden kriteerien suhteen!

mainittu. Jokaisen henkilön osalta voidaan ainakin lauserakenteen mukaisesti sanoa, kuka on seuraava (se on aina se, jonka nimi on kirjoitettu pilkun jälkeen). Listaa yleisesti ottaen voidaan muokata esimerkiksi poistamalla sieltä olio, esim. Paavon poistamisen jälkeen em. listaan jäisi järjestyksessä ”Tomi, Seppo, Jarno”. Listaan voi myös lisätä olioita eri kohtiin. Esimerkiksi voitaisiin lisätä ”Ismo” johonkin kohtaan, vaikkapa ”Sepon” jälkeen, jolloin lopputulemana olisi lista ”Tomi, Seppo, Ismo, Jarno”. Listan ominainen piirre on, että sitä voidaan käydä läpi ainoastaan järjestyksessä, keulasta alkaen, kustakin oliosta yksi kerrallaan seuraavaan selaamalla. Indekseillä ei voi peruslistan keskelle hypätä, vaan täytyy selata läpi indeksin mukainen määrä viitteitä aina seuraavaan elementtiin.

Jono (engl. *queue*) on lista, johon lisätään aina loppuun ja otetaan pois keulilta. Reaalimaailman esimerkki: Piato-ravintolan ruokajonossa on tällä hetkellä Tomi, Paavo, Seppo ja Jarno. Tomia palvellaan ja hän saa lounaan; poistetaan ruokajonosta, jolloin jonossa on Paavo, Seppo ja Jarno. Kaisa saapuu jonoon, jolloin jonossa on Paavo, Seppo, Jarno ja Kaisa. Ulla saapuu jonoon, jolloin jonossa on Paavo, Seppo, Jarno, Kaisa ja Ulla. Paavoa palvellaan ja hän saa lounaan, jolloin jonossa on Seppo, Jarno, Kaisa ja Ulla. Seppoa palvellaan ja hän saa lounaan, jolloin jonossa on Jarno, Kaisa ja Ulla . . . Tähän kurssiin liittyvä ennakoiva sovellus: (1) prosessoriaikaa odottavien prosessien jono (2) viestijono, jossa prosessille tarkoitetut viestit odottavat pääsyä prosessin käsittelyyn. (3) resurssin lukituksen aukenemista odottavien prosessien jono.

Pino (engl. *stack*) on lista, johon lisäys ja poisto tapahtuvat samasta päästä. On sitten toteutusyksityiskohta, tapahtuuko lisäys ja poisto listan alkuun vai loppuun. Esim. jos toteutustapa on taulukko, on edullisinta lisätä ja poistaa viimeinen alkio, mutta jos toteutustapa on linkitetty lista, kannattaa lisäys ja poisto tehdä lis-

tan alkuun. Reaalimaailman esimerkki: pelikorttipino, johon tietyn pasianssin sääntöjen mukaan saa laittaa kortin päällimmäiseksi – alempia kortteja ei saa alta edes nähdä. Vain päällimmäisenä näkyvän kortin saa ottaa käyttöön pelin sääntöjen mukaisesti. Tähän kurssiin liittyvä ennakoiva sovellus: (1) Aliohjelma-aktivaatioista muodostuva kutsupino abstraktina käsitteenä (2) pinon konkreettinen toteutus muistialueena prosessin virtuaalimuistissa (3) erillinen käyttöjärjestelmäpino järjestelmäkoodin aktivaatioille.

Puu (engl. *tree*) on matemaattisesti ”verkko, jossa ei ole yhtään silmukkaa”. Reaalimaailman esimerkki kuvasta 0.5: Puussa on juuri, haaroja ja lehtiä. Teknisesti näitä rakenteen solmupisteitä voi kutsua **solmuiksi** (engl. *node*). Kuvassa 0.6 on abstrakti puu piirrettynä tavanomaisella tavalla, jossa **juurisolmu** (engl. *root node*) on ylimpänä ja puu ”kasvaa” alaspäin kohti **lehtisolmuja** (engl. *leaf node*). Tähän kurssiin liittyvä ennakoiva sovellus: (1) hakemistopuu, esim. JY:n suorakäyttökoneilla toimii komento `tree ~`, jolla voi nähdä oman kotihakemiston haaromisen alihakemistoihin ja lehtisolmuina ilmeneviin tiedostoihin. (2) prosessipuu, esim. komennolla `ps -ef --forest` voi visualisoida suorakäyttökoneen kaikkien käyttäjien kaikki prosessit puurakenteena.

Abstraktit tietorakenteet voidaan kuvailla ohjelmointikielessä viitteiden – ja laiteläheisessä C-kielessä yksinkertaisesti muistiosoitteiden – avulla. Esimerkiksi listan elementti voisi olla seuraavanlainen:

```
struct ListaElementti{
  Sisaltotyyppi *varsinainen_sisalto;
  ListaElementti *seuraava_elementti;
}
```

Hieman helpommaksi listan käsittely saadaan laittamalla elementteihin tieto seuraajan lisäksi myös edeltäjästä:

```
struct ListaElementti{
  Sisaltotyyppi *varsinainen_sisalto;
  ListaElementti *seuraava_elementti;
  ListaElementti *edellinen_elementti;
}
```

Tämän viimeksi mainitun listaelementin koko on 64-bittisessä tietokoneessa $3 \times 64 = 192$ bittiä eli 24 kasibittistä tavua. Miksi? Siihen sisältyy kolme 64-bittisenä muistiosoitteena ilmenevää viitettä. Ensimmäinen viite osoittaa varsinaiseen olioon, jota tämä yleisen listamallin elementti käsittelisi. Se voi itsessään olla minkä tahansa kokoinen; lista toteuttaa vain abstraktin *listamaisuuden* ja itse oliot voivat olla mitä tahansa tarkoitusta varten.

Tämä riittänee meille toistaiseksi tietorakenteista. Demoissa ja luennoilla pitäisi tulla sitten lisää konkretiaa esimerkkien kautta. Paljon lisää erilaisista abstrakteista tietorakenteista ja niiden toteuttamisesta ja soveltamisesta tulee siis kursseilla Algoritmit 1 ja 2.

Tietokonelaitteisto

Digitaalinen laskin (engl. *digital computer*), jota on totuttu nimittämään sanalla **tietokone**, toimii tietyssä mielessä erittäin yksinkertaisesti. Kaikki niin sanottu *tiedon* tai varsinkin *informaation* käsittely, jota digitaalisella laskimella ilmeisesti voidaan tehdä, on täysin ihmisen toteuttamaa (ohjelmia ja järjestelmiä luomalla), eikä kone taustalla tarjoa paljonkaan tietoa tai älykkyyttä, vaikka onkin älykkäiden ihmisten luoma sähköinen automaatti, joka nykypäivänä sisältää suuren joukon apujärjestelmiä jo sisälläänkin. Tietokone ensinnäkin osaa käsitellä vain **bittejä** (engl. *binary digit, bit*) eli kaksijärjestelmän numeroita, nollia ja ykkösiä. Bittejä voidaan yhdistää – esim. kahdeksalla bitillä voidaan koodata 256 eri kokonaislukua. Bitit ilmenevät koneessa sähköjännitteinä, esi-

merkiksi jännite välillä 0 – 0.8V voisi tarkoittaa nollaa ja jännite välillä 2.0 – 3.3V ykköstä. Bittejä voidaan tallentaa myös pitkäaikaisiin tallenteisiin, jolloin esimerkiksi magnetoituvan materiaalin magneettikenttä käännetään enemmän tai vähemmän pysyvästi yhteen suuntaan silloin kun bitti on ykkönen ja vastakkaiseen suuntaan kun se on nolla – magneettinauhat ja kovalevyt perustuvat tähän. Vielä pidempiaikainen tallenne saadaan, kun fyysisesti kaiverretaan johonkin materiaaliin kuoppa nollabitin kohdalle ja jätetään kaivertamatta ykkösbitin kohdalta. Kiveen kaiverrettuna peräkkäiset bitit säilyisivät käytännössä ikuisesti, toisin kuin nykyisissä magneettisissa välineissä, joiden magneettikentillä on taipumus jollain aikavälillä palautua fysiikan lakien mukaisesti epämääräiseen, satunnaiseen orientaatioon[9].

Bittien käsittelyltä ei voida tietokoneissa välttyä, joten käydään seuraavaksi lyhyesti läpi lukujärjestelmien perusteita.

Lukujärjestelmät

Ihmiset ovat tottuneet laskemaan kymmenjärjestelmän luvuilla, mutta kun ollaan tietokoneiden kanssa tekemisissä, on ymmärrettävä myös binäärijärjestelmää. Idea on helppo ja sama kuin kaikissa kantalukuun perustuvissa lukujärjestelmissä: yksi bitti on ykkönen tai nolla. Jos luvussa on useampia bittejä, ilmoittaa jokainen bitti (0 tai 1) aina, kuinka monta vastaavaa kantaluvun kaksi potenssia lukuun sisältyy. Kymmenjärjestelmän luvuissahan jokainen numero (0, 1, 2, 3, 4, 5, 6, 7, 8 tai 9) ilmoittaa, kuinka

[9]NASA:n Voyager-luotaimeen kiinnitetty viesti vieraalle sivilisaatiolle on normaali äänilevy, jossa äänet ja kuvat on kaiverrettu analogisena signaalina levyn pintaan, mutta levyn käyttöohjeisiin on kaiverrettu tiettyjä binäärilukuja siinä toivossa, että kaksijärjestelmä olisi yksinkertaisuudessaan universaali ja intergalaktinen. Onhan siinä vain kaksi symbolia. Myös Aurinkokunnan sijainti on ilmoitettu viestissä binäärilukujen avulla. (Kyllä, tätä monistetta on kirjoitettu öiseen aikaan ja Wikipedia on ollut saatavilla.)

monta vastaavaa kantaluvun kymmenen potenssia lukuun sisältyy. Esimerkki samasta lukumäärästä 123 kymmenjärjestelmän ja binäärijärjestelmän kirjoitusasussa: Kymmenjärjestelmässä $123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 100 + 20 + 3 = 123$. Binääri- eli kaksijärjestelmässä $1111011_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 64 + 32 + 16 + 8 + 0 + 2 + 1 = 123$. Kantalukua ilmaiseva alaindeksi on tässä ja myöhemmissä esimerkeissä kirjoitettu pääsääntöisesti vain muihin kuin kymmenjärjestelmän lukuihin.

Binääriluvut ovat siis ainoa tietokoneen ymmärtämä lukujärjestelmä. Vähänkään suuremmat binääriluvut ovat pitkiä ja siksi tikkuisia tulkita ja kirjoittaa. Nykyiset tietokoneet käsittelevät monia asioita 64-bittisinä ja mm. salausavaimia sitäkin useammilla biteillä. Paljon helpompaa bittien tulkitseminen on kuusitoistajärjestelmässä eli heksadesimaalijärjestelmässä[10]. Puhutaan ammattijargonilla lyhyesti **heksaluvuista** eli heksoista. Kyseessä on ihan samanlainen järjestelmä kuin muutkin lukujärjestelmät, jossa jokainen luvun numero (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E tai F) ilmoittaa, montako vastaavaa kantaluvun kuusitoista potenssia lukuun sisältyy. Kirjainten A-F (tai pienten kirjainten a-f) käyttäminen lukumääriä 10, 11, 12, 13, 14 ja 15 ilmaisevina heksanumeroina on vain käytäntö – yhtä hyvin symbolit voisivat olla sydämiä, peukkuja ja hymynaamoja. Pitäydytään kuitenkin vallitsevan käytännön mukaisissa heksanumeroissa, niin ei tule sekaannuksia. Esimerkiksi luku 123 on heksajärjestelmässä $7B_{16} = 7 \cdot 16^1 + 11 \cdot 16^0 = 123$. Symboli 'B' vastaa siis lukumäärää 11 meille tutuimmassa kymmenjärjestelmässä.

[10] *Desimaali* ei tässä sanassa liity 10-järjestelmään, vaan kantalukuun kuusi-toista eli jotakuinkin *hexa - decimal*. Tämä tieto nyt on Wikipediasta, joten ei niellä purematta; lisäksi Wiki kertoo meille, että heksa tulisi kreikan kielestä ja decimal latinasta. Tietoteknikot eivät ole alan sanastoa luodessaan olleet ainakaan mitään kielipoliiseja...

Miten niin heksaluvut ovat näppäriä, kun mietitään bittejä? Koska kantaluku 16 on kakkosen potenssi, tarkkaan ottaen neljäs potenssi, $16 = 2^4$, vastaa jokainen heksanumero yksi-yhteen neljää bittiä luvun binääriesityksessä, esim. $111\ 1011_2 = 7B_{16}$. Nyt ei tarvitse muistella kerrallaan kuin jokaisen heksanumeron vastaavuutta neljän bitin sarjan kanssa, niin voidaan kuinka tahansa pitkiä bittilukuja muuttaa vaivatta pitkästä binääriesityksestä lyhyeen heksaesitykseen. Tässä kirjoittajan apinahakkauksella toteuttama satunnainen binäärilukuesimerkki muunnettuna heksoiksi päässälaskien: $10\ 1010\ 0101\ 1011\ 0001\ 0101_2 = 2A5B15_{16}$. Tarkista itse, menikö muunnos oikein.

Joissain yhteyksissä voi olla mielekästä käyttää myös oktaali- eli kahdeksanjärjestelmää eli **oktaalilukuja**. Jokainen oktaaliluvun numero (0, 1, 2, 3, 4, 5, 6 tai 7) ilmoittaa tietenkin montako kertaa kantaluku kahdeksan vastaava potenssi esiintyy luvussa. Kantaluku kahdeksan on kakkosen kolmas potenssi, joten oktaaliluvun numerot vastaavat kolmen bitin yhdistelmiä. Edelliset esimerkit ovat siis $1\ 111\ 011_2 = 173_8$ ja $1\ 010\ 100\ 101\ 101\ 100\ 010\ 101_2 = 12455425_8$.

Ohjelmointikielissä ja kirjallisuudessa on vakiintunut useita erilaisia tapoja ilmoittaa, että jokin vakioluku on kirjoitettu jossakin tietyssä edellä käsitellyistä lukujärjestelmistä. Oletus on yleensä kymmenjärjestelmä, mikäli on kirjoitettu pelkkä luku, vaikkapa 123. Heksaluvut kirjoitetaan aika usein (mm. C#, Java, C++ ja C -lähdekoodeissa) niin, että niiden alkuun lisätään merkit `0x` eli nolla ja pieni äksä, esimerkiksi `0x7B` tai `0x2A5B15`. Oktaaliluvut kirjoitetaan aika usein niin, että niiden alkuun lisätään merkki ”0” eli ylimääräinen nolla, esimerkiksi `0173`. Varo siis vaaraa, kun ohjelmoit: lähdekoodissasi `0123` tarkoittaa luultavasti aivan erilaista lukumäärää kuin kymmenjärjestelmän 123! Jossain kielissä (tuskin kuitenkaan C:ssä ja jälkeläisissä. . .) binääriluvut kirjoitetaan

laittamalla niiden perään pieni b-kirjain, esim. 1111011b. Binäärilukuja kuitenkin harvemmin kirjoitetaan sellaisenaan. Ohjelmoijalle on selvää, että `0xFF` tarkoittaa 8-bittistä lukua, jonka kaikki bitit ovat ykkösiä, `0xFE` tarkoittaa 8-bittistä lukua, jossa kaikki paitsi vähiten merkitsevä bitti ovat ykkösiä, `0xAAAA` tarkoittaa 16-bittistä lukua, jossa joka toinen bitti on ykkönen ja joka toinen nolla (ja niin edelleen...). Jos käytössä on 64-bittinen tallennustila, suurin mahdollinen etumerkitön kokonaisluku, jota voidaan käsitellä on `0xFFFFFFFFFFFFFFFF`. Yritys lisätä tähän ykkönen ja tallentaa syntyvä 65-bittinen tulos `0x10000000000000000` aiempaan 64-bittiseen tallennustilaan tyypillisesti pyöräyttää lukualueen vaivihkaa ympäri ja tuloksena onkin nolla[11].

Joissain yhteyksissä, varsinkin työkaluohjelmien tulosteissa tai salausavaimissa ymv., ei ole mitään erityistä merkintää siitä, että luvut ovat heksoja. Usein nämä tulosteet tulevat kuitenkin vastaan siinä vaiheessa, kun jo tiedät, että haluat nähdä tai kirjoittaa nimenomaan heksalukuja. Nykyisin bittejä on tyypillistä ajatella kahdeksan bitin paketteina eli **tavuina**. Jokaista tavua vastaa näppärästi kaksinumeroinen heksaluku. Tietokoneen datan tarkastelu mikrotasolla on helppoa juuri tavujen heksaesityksiä tutkimalla. Kurssin luennoilla ja demoissa tullaan tekemään tätä käytännössä työkaluohjelmien avulla.

Yksinkertaisista komponenteista koostettu monipuolinen laskukone

Tietokone valmistetaan yksinkertaisista elektroniikkakomponenteista koostamalla. Toki kokonaisuus sisältää erittäin suuren määrän

[11] Tällaisesta kokonaisluvun ylivuodosta engl. *integer overflow* voi aiheutua vaarallisiakin toimintahäiriöitä, jos siihen ei ole varauduttu. Toistaiseksi kalleimmaksi käyneitä lienee ensimmäisen Ariane 5 -tyyppisen avaruusraketin räjähtäminen vuonna 1996.

komponentteja, jotka muodostavat monimutkaisen, osin hierarkkisen ja osin modulaarisen rakenteen. Suunnittelussa ja valmistusteknologiassa on tultu pitkä matka tietokoneiden historian aikana. Peruskomponentit ovat kuitenkin yhä tänä päivänä 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. Kuvassa 0.7 esitetään tietokone kaikkein yleisimmällä tasolla, jonka komponenteilla on tietyt tehtävänsä. Tietokoneessa on **keskusyksikkö** (engl. *CPU, Central Processing Unit*) eli **suoritin** eli **prosessori** (engl. *processor*), **muisti** (engl. *memory*) ja **syöttö- ja tulostuslaitteita** eli **I/O-laitteita** (engl. *Input/Output modules*). Komponentteja yhdistää **väylä** (engl. *bus*)[12]. Keskusyksikkö hoitaa varsinai-

[12]Todellisuudessa ns. emolevyyn (engl. *motherboard*) on yhdistetty myös muita ohjauskomponentteja. Ulkoinen väylä voi jakautua esim. ns. pohjoissillan (engl. *northbridge*) ohjaamaan nopeampaan väylästöön (muisti, grafiikkaohjain) ja eteläsillan (engl. *southbridge*) ohjaamaan hitaampaan väylästöön (näppäimistö, kovalevyt, ym.). Esimerkki AMD64:n 940-nastaisen paketin spesifikaatiosta: `http://support.`

sen biteillä laskemisen, käyttäen apunaan muistia. I/O -laitteisiin lukeutuvat mm. näppäimistö, hiiri, näytönohjain, kovalevy, DVD, USB-tikut, verkkoyhteydet, printterit ja monet muut laitteet. Toimintaa ohjaavalle prosessorille nämä erilaiset I/O -laitteet näyttävät kaikki hyvin samanlaisilta, mihin asiaan palataan kurssilla myöhemmin.

Väylää voi ajatella rinnakkaisina johtimina, eli kaapelina elektronisten komponenttien välillä. Kussakin johtimessa voi tietenkin olla kerrallaan vain yksi bitti, joten esimerkiksi 64 bitin siirtäminen kerrallaan vaatii 64 rinnakkaista sähköjohdinta. Kaapeleita 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 johdinta (bittiä) ja datalinjan leveys 64 bittiä. Käytännössä tämä tarkoittaisi, että väylää pitkin voi päästä käsiksi korkeintaan 2^{38} eri paikkaan ja kuhunkin osoitettuun paikkaan (tai sieltä toiseen suuntaan) voisi siirtää kerrallaan maksimissaan 64 bittiä.

Sanotaan, että prosessorin ulkopuolella kulkevan osoiteväylän leveys (bittijohdinten lukumäärä) määrittelee tietokoneen **fyysisen osoiteavaruuden** (engl. *physical address space*) laajuuden eli montako muistiosoitetta tietokoneessa voi olla. Muistia ajatellaan useimmiten kahdeksan bitin kokoelmien eli **tavujen** (engl. *byte*) muodostamina lokeroina, jollaista jokainen osoite osoittaa. Tästä aiheutuu se, että vaikka dataväylän leveys (bittijohdinten lukumäärä) mahdollistaisi useiden tavujen siirtämisen kerralla johonkin osoitteeseen, on osoitettavissa olevien yksittäisten bittien määrä tavallisissa järjestelmissä osoiteavaruuden koko $\times$ 8 bittiä. Esimerkiksi 64-bittinen siirto käyttäisi kahdeksaa peräkkäistä 8-bittistä muistipaikkaa, joista ensimmäisen paikan osoite annetaan ja seu-

---

`amd.com/TechDocs/31412.pdf` – voidaan huomata, että tämän kurssin tarpeisiin riittävä mielikuvamalli on varsin yksinkertainen todellisuuteen nähden!

raavia seitsemää peräkkäistä paikkaa käytetään automaattisesti.

Tietokoneen **sananpituus** (engl. *word length*) on termi, jolla kuvataan tietyn tietokonemallin kerrallaan käsittelemien bittien määrää (joka saattaa olla sama kuin ulkoisen dataväylän leveys). Tämä on hyvä tietää, kun lukee alan kirjallisuutta, prosessorimanuaaleja ja vastaavia. 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. Mm. esimerkkinä käytetyn AMD64-prosessorin dokumentaatio käyttää näitä määritelmiä.

Väylän kautta pääsee käsiksi moniin paikkoihin, ja osoiteavaruus jaetaan usein (laitteistotasolla) osiin siten, että tietty osoitteiden joukko tavoittaa **ROM-muistin** (engl. *Read-Only Memory*, joka on 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 prosessorin ulkopuolella sijaitsevat lisälaitteet. Nykypäivänä normaalit ohjelmat näkevät itse asiassa niitä varten luodun **virtuaalisen osoiteavaruuden** – tähän käsitteeseen palataan myöhemmin.

Tietokoneen prosessori toimii nopean kellopulssin ohjaamana: Aina kun kello ”tikittää” 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[13].

[13]Nykyiset prosessorit toki sisältävät teknologioita (nimeltään mm. esinouto (prefetch), liukuhihnoitus (engl. pipelines), ennakoiva suoritus (engl. speculative execu-

Myös väylä voi muuttaa johtimissaan olevia bittejä vain kellopulssin lyödessä – väylän kello voi olla hitaampi kuin prosessorin, jolloin väylän toimenpiteet ovat harvemmassa. Ne kestävät muutenkin pidempään, koska sähkösignaalin on kuljettava pidempi matka perille oheislaitteeseen ja aikaa kuluu myös väylän ohjauslogiikan toimenpiteisiin. Operaatiot sujuvat siis 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). Tai tältä se ohjelmien näkökulmasta näyttää. Todellinen toteutustapa on villi vekotin, yli 70-vuotisen digitaalielektroniikan kehitystyön kompleksinen tulos, mutta se on *piilossa rajapinnan takana*, eikä meidän tarvitse tietää kuin rajapinta voidaksemme käyttää järjestelmää! Rajapinta puolestaan on tarkoituksella suunniteltu (rajoitteiden puitteissa) mahdollisimman yksinkertaiseksi.

Kuva 0.8 tarkentaa vielä keskusyksikön jakautumista hierarkkisesti alemman tason komponentteihin. Näitä ovat **kontrolliyksikkö** (engl. *control unit*), **aritmeettislooginen yksikkö** (engl. *ALU, Arithmetic Logic Unit*) sekä tiettyihin tarkoituksiin välttämättömät **rekisterit** (engl. *registers*). Tästä alempia laitteistohie-

---

tion), 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! Laitteet on suunniteltu näyttämään ulospäin peräkkäisiltä, eikä tämän kurssin tarkoitus ole mennä pitkälle prosessoriteknologian nyansseihin. Oikeasti esim. if-else -ehtolauseessa tietokone saattaa laskea etukäteen molemmat lopputulemat, mutta automatiikka unohtaa jälkikäteen tuloksista sen, jota ei tarvittu.

rarkian tasoja ei tällä kurssilla tarvitse ajatella, sillä tästä kuvasta jo löytyvät esimerkit tärkeimmistä komponenteista joihin ohjelmoija (eli sovellusohjelmien tai käyttöjärjestelmien tekijä) pääsee käsiksi. Alempien tasojen olemassaolo (eli monimutkaisen 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 pienen 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.

Suoritussykli (yhden ohjelman kannalta)

Nyt voidaan kuvan 0.8 terminologiaa käyttäen vastata riittävän täsmällisesti siihen, mikä itse asiassa on se *yksittäinen toimenpide*, jollaisen prosessori voi aloittaa kellopulssin tullessa. Toimenpide on yksi kierros toistosta, jonka nimi on **nouto-suoritus -sykli** (engl.

fetch-execute cycle)[14]. 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 ne eivät mahdu kokonaan talteen kovin lähelle prosessoria. Käskyt sijaitsevat siis muistissa, josta seuraava käsky aina noudetaan väylän kautta[15]. Jotta noutaminen voi tapahtua, täytyy väylän osoitelinjaan kytkeä käskyn muistipaikan osoite. Jotta bittejä voidaan kytkeä johonkin, ne pitää tietenkin olla jossakin säilössä. Seuraavan käskyn osoite on tallessa rekisterissä, jonka nimi on **käskyosoitin** (`IP`, engl. *instruction pointer*). Toinen nimi samalle asialle (kirjoittajasta riippuen) voisi olla **ohjelmalaskuri** (`PC`, engl. *program counter*). Siis elektroniikka kytkee `IP` -rekisterin osoitelinjaan, odottaa että väylän datalinjaan välittyy muistipaikan

[14]Tämä on taas hienoinen valhe asian yksinkertaistamiseksi. Itse asiassa nykyprosessorit suorittavat käskynsä ns. mikrokoodina, johon liittyy decode -vaihe, ja sykliä sanotaan joskus vastaavasti engl. *fetch-decode-execute* -sykliksi. Lisäksi edellisessä alaviitteessä mainitut nopeutusteknologiat tekevät sisäisestä toiminnasta monimutkaisempaa. Näilläkään ei ole vaikutusta siihen, miltä toiminta näyttää ulospäin. Positiivista on se, että monenlaiset prosessoreissa havaittavat suunnitteluviat eivät ole ”aitoja laitevikoja” vaan ne voidaan korjata tehtaalta tulon jälkeenkin päivittämällä prosessorin mikrokoodi.

[15]Valehtelu jatkuu toistaiseksi: myöhemmin puhutaan ns. välimuisteista. Pidetään kuitenkin asia toistaiseksi yksinkertaisena, miltä se yhä edelleenkin on tehty näyttämään ohjelman tekijälle päin!

sisältö, ja kytkee datalinjan rekisteriin, joka tunnetaan esimerkiksi nimellä käskyrekisteri (`INSTR`, `IR`, engl. *instruction register*). Seuraava käsky on nyt noudettu ja sitä vastaava bittijono on `INSTR`-rekisterin sisältönä. Prosessori noutaa muistista mahdollisesti myös käskyn tarvitsemat **operandit** eli luvut, joita operaatio tulee käyttämään syötteenään. Luonnollisesti operandien tulee sijaita sisäisissä rekistereissä tai suoraan väylän datalinjalta ALUn portteihin kytkettynä. [16]

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 myöhemmin 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** (`FLAGS`, `FLG`, `FR`, engl. *flag register*), josta käytetään myös englanninkielistä nimeä engl. *Program status word, PSW*. Jotta laskutoimituksista olisi hyötyä, niiden tulokset täytyy saada talteen johonkin rekisteriin (tai suoraan muistiin, mikä taas edellyttää väylän käyttöä ja sitä

[16] 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.

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:

- **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 ilmoitettu uusi muistiosoite, esim. silmukan ensimmäinen käsky tms.

- **Ehdollinen hyppykäsky** → `IP`:n sisällöksi on ladattu käskyssä ilmoitettu uusi osoite, mikäli käskyssä ilmoitettu ehto toteutuu; muutoin `IP` osoittaa seuraavaan käskyyn samoin kuin peräkkäissuorituksessa. (Ehto tulkitaan koodatuksi FLAGS-lippurekisterin johonkin/joihinkin bitteihin.)

- **Aliohjelmakutsu** → `IP`:n sisältönä on käskyssä ilmoitettu uusi osoite (jonka tulee olla kutsuttavan aliohjelman ensimmäinen käsky; aliohjelmakutsussa prosessori tekee muutakin, mitä käsitellään tällä kurssilla 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 ns. pinomuistia. Myös pinomuistin käyttöä tutkitaan tarkemmin tulevissa demoissa ja luennoissa.

Lippurekisteri (`FLAGS` tmv.) puolestaan tallentaa prosessorin tilaan liittyviä on/off -liputuksia, jotka voivat muuttua tiettyjen käskyjen suorituksen seurauksena. Prosessoriarkkitehtuurin määritelmä kertoo, miten mikäkin käsky muuttaa `FLAGS`:iä. Kolme tyypillistä esimerkkiä voisivat olla seuraavat:

- Yhteenlaskussa (bittilukujen ”alekkain laskeminen”) voi jäädä muistibitti yli, jolloin nostetaan engl. *carry flag* lippu – se on tietty bitti `FLAGS`-rekisterissä, ja sen nimi on usein kirjallisuudessa `CF`. Samalla nousee luultavasti ylivuodon (engl. *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 engl. *negative flag*, `NF`, etumerkkilippu (tai vastaavaa. . .)

- Jos jonkun operaation tulos on nolla (tai halutaan koodata joku tilanne vastaavasti) asettuu nollalippu (engl. *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 ehtoa testaavan käskyn suorittamiseen, ja vaikkapa ehdollisen hyppykäskyn suorittamiseen testin jälkeen (hyppy

tapahtuu vain jos tietty bitti `FLAGS`:ssä on asetettu). Käyttöjärjestelmälle varatut prosessoriominaisuudet eivät ole käytettävissä silloin kun FLAGS:n käyttäjätilalippu ei niitä salli.

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 aina seuraavan käskyn osalta, 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 muistiosoitteen ilmaisemiseen. 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 ja hitaampi suorittamaan toimenpiteitä).

Yhteenveto: Ohjelmien suoritukseen kykenevässä prosessorissa on oltava ainakin `IP`, `FLAGS` ja lisäksi rekisteri, joka voi sisältää dataa tai osoittaa muistipaikkaa, jossa data sijaitsee. Tyypillisesti nykyprosessoreissa on joitain kymmeniä rekisterejä yleisiin ja erityisiin käyttötarkoituksiin.

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.

Prosessorin toimintatilat ja käynnistäminen

Eräs tarve tietokoneiden käytössä on eriyttää kukin normaali käyttäjän ohjelma omaan 'karsinaansa', jotta ohjelmat eivät vahingossakaan sotke toisiaan tai järjestelmän kokonaistoimintaa. Tätä tarkoitusta varten prosessorissa on erikseen **järjestelmärekisterejä** (engl. *system registers*) ja toimintoja, joihin pääsee käsiksi vain käyttöjärjestelmän suoritettavissa olevilla konekielikäskyillä eli **järjestelmäkäskyillä** (engl. *system instructions*). 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** eli **ydintilaan** (engl. *kernel mode*). Englanninkielinen nimi viittaa nimenomaan käyttöjärjestelmän ytimeen (engl. *kernel*) eli siihen osaan käyttöjärjestelmän ohjelmakoodia, jota suoritetaan ydintilassa. Osa käyttöjärjestelmän palveluista on vähemmän kriittisiä, joten niitä ei ole pakko suorittaa ydintilassa. Muita tunnettuja nimiä käyttöjärjestelmätilalle ovat (vapaahkosti suommennettuna) 'todellinen tila' (engl. *real mode*) tai 'valvojatila' (engl. *supervisor mode*). Käynnistyksen jälkeen prosessori alkaa suorittaa ohjelmaa ROM-muistista (kiinteästi asetetusta fyysisestä muistiosoitteesta alkaen). RAM-muisti on tyhjentynyt tai satunnaistunut virran ollessa katkaistuna. Oletuksena on, että ROM:issa oleva, yleensä pienehkö, ohjelma lataa varsinaisen käyttöjärjestelmän joltakin ulkoiselta tallennuslaitteelta[17]. Käynnistettäessä tietokone siis on vain prosessoriarkkitehtuurinsa mukainen digitaalinen laskin, eikä esim. ”macOS, Windows tai Linux -kone”.

Käyttöjärjestelmän latausohjelmaa etsitään melko 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 RAM-muisti tarkoituksenmu-

[17] Olet ehkä huomannut, että kotitietokoneiden ROM:issa on yleensä BIOS-asetusten säätöohjelmisto, jolla käynnistyksen yhteydessä voi määrätä fyysisen laitteen, jolta käyttöjärjestelmä pitäisi koettaa löytää (DVD, CD-ROM, kovalevyt, USB-tikku jne...). BIOS tarjoaa myös muita asetuksia, jotka säilyvät virran katkaisun jälkeen (esim. pariston avulla; pariston tyhjentyessä tietokone alkaa menettämään näitä asetuksia ja kellokin saattaa näyttää tammikuun ensimmäistä vuonna 2000 tai jotain muuta ihmeellistä).

kaisella tavalla ja ladata muistiin seuraavissa vaiheissa tarvittavat ohjelmiston osat.

Alkulatauksen (engl. *bootstrapping*, engl. *booting*)[18] jälkeen käyttöjärjestelmäohjelmiston pitää vielä tehdä koko liuta muitakin valmisteluja sekä lopulta tarjota käyttäjille mahdollisuus kirjautua sisään koneelle ja alkaa suorittamaan hyödyllisiä tai viihteellisiä ATK-sovelluksia. Esimerkiksi ilman erillistä graafista käyttöliittymää varustettu 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 (lue: ammattislangilla) kutsutaan 'shelliksi' myös suomen kielellä. Englismi on jopa niin vakiintunut, että käytämme jatkossa ainoastaan sitä, kun puhumme kuoresta. Käyttöliittymältään ja toiminnallisuuksiltaan jonkin verran erilaisia shellejä on syntynyt aikojen saatossa monta (mm. *bash*, *tcsh*, *ksh*). Käyttöjärjestelmä ohjaa päätteen näppäinsyötteet shellille ja shellin tulosteet 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 aiemmin paljon käytetty ikkunointijärjestelmä nimeltä X tai sen uudempi korvaaja Wayland. Nykypäivänä käyttäjä myös edellyttänee, että hänelle tarjotaan **työpöytä** (engl. *desktop manager*), joka on kuitenkin jo melko korkealla tasolla varsinaiseen käyttöjärjestelmän ytimeen nähden, eikä siten kovinkaan paljon tämän kurssin aihepiirissä.

Kirjautumisen jälkeen kaikki käyttäjän ohjelmat toimivat proses-

[18] Termi tulee joko vanhassa kojeessa olleesta uudelleenkäynnistyskytkimestä, jota oli tapana potkaista, tai paroni von Münchhausenin uroteosta nostaa itsensä ja ratsunsa suosta saappaanraksista vetämällä, mutta tämäkin on valetta. Herra Murphee sen sijaan on nostellut itseään raksista joen tai aidan yli 1800-luvun Amerikassa.

sorin 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. Käyttäjätilassa toimii mahdollisesti myös osa käyttöjärjestelmän palveluohjelmista. Käyttöjärjestelmää, jossa suurin osa palveluista toimii käyttäjätilassa ja käyttöjärjestelmätilassa toimii vain minimaalinen määrä koodia, sanotaan ymmärrettävällä logiikalla **mikroydinkäyttöjärjestelmäksi** (engl. *microkernel operating system*). Vastakohta, jossa kaikki käyttöjärjestelmän palvelut toimivat prosessorin ollessa käyttöjärjestelmätilassa on nimeltään **monoliittinen käyttöjärjestelmä** (engl. *monolithic operating system*). Mikroydinjärjestelmä on turvallisempi ja toimintavarmempi. Monoliittisessa puolestaan on mahdollista maksimoida suorituskyky. Käytännön toteutuksissa haetaan näiden ääripäiden väliltä kompromissi, joka toimii riittävän tehokkaasti, mutta osa palveluista hoidetaan rajoitetussa käyttäjätilassa, missä mm. palvelukohtaisten oikeuksien määrittäminen on mahdollista.

Prosessorin tilaa (käyttäjä-/käyttöjärjestelmätila) säilytetään jossakin vähintään yhden bitin kokoisessa sähkökomponentissa prosessorin sisällä. Tämä tila (esim. 0==käyttöjärjestelmä, 1==käyttäjätila) voi olla esim. yhtenä bittinä lippurekisterissä. Itse asiassa kurssilla esimerkkinä käytettävä AMD64-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.

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. Tästä kerrotaan myöhemmin lisää.

Käyttäjätilassa konekielisissä ohjelmissa saa olla vain käyttäjätilassa sallittuja käskyjä ja käskyjen operandeina voi käyttää vain **käyttäjälle näkyviä rekisterejä** (engl. *user visible registers*). Prosessorin ollessa käyttäjätilassa oheislaitteiden suora käyttäminen on mahdotonta, samoin kuin muiden sellaisten muistiosoitteiden käyttö, joihin käyttäjätilan ohjelmalle ei nimenomaisesti ole annettu lupaa. Lupia voi muuttaa vain, kun prosessori on käyttöjärjestelmätilassa. Mitä luvat käytännössä tarkoittavat, on kurssin loppupuolen asiaa. . . pienenä spoilerina voidaan tässä vaiheessa todeta, että ne ilmenevät muistiosoitteisiin liittyvinä yksittäisinä

bitteinä, esim. 0 = ei saa käyttää; 1 = saa käyttää. Suoranainen yllätys tämä asia toivottavasti ei kuitenkaan enää ollut, kun on puhuttu tietokoneen luonteesta yksinkertaisena bittiautomaattina.

Käskykanta-arkkitehtuureista

Tietty **prosessoriarkkitehtuuri** tarkoittaa niitä tapoja, joilla sen mukaisesti rakennettu fyysinen prosessorilaite toimisi: Mitä toimenpiteitä se voisi tehdä (eli millaisia käskyjä sillä voisi suorittaa), mistä ja mihin mikäkin toimenpide voisi siirtää bittejä, ja miten mikäkin toimenpide muunnettaisiin (tavallisesti muutaman tavun mittaiseksi) bittijonoksi, joka sisältää **operaatiokoodin**, (engl. *opcode*, engl. *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 onkin **käskykanta-arkkitehtuuri** (engl. *ISA, instruction set architecture*).

Tarkkaavainen lukija huomasi, että edellinen kappale oli kirjoitettu konditionaalissa: 'toimisi', 'voisi tehdä', ... Tämä johtuu siitä, että arkkitehtuuri on nimenomaan sopimus siitä, kuinka laitetta voitaisiin käyttää. Se, onko arkkitehtuurin mukaisia laitteita olemassa, on eri asia. Luultavasti uusi prosessorimalli on olemassa ensin arkkitehtuuridokumentaationa, sen jälkeen elektroniikkasuunnitelmana ja simulaattorina ja vasta lopputulemana fyysisenä laitteena tehtaan paketissa. Laitteelle voidaan periaatteessa tehdä konekielisiä ohjelmia ja kääntäjiä jo ennen kuin yhtään laitetta on valmistettu. Ei tarvita kuin dokumentoitu prosessoriarkkitehtuuri.

Prosessoriarkkitehtuureita ja niitä toteuttavia fyysisiä prosessorilaitteita on markkinoilla monta, ja niissä on merkittäviä eroja,

mutta kaikissa on jollakin tavoin toteutettu edellä kuvaillut pakolliset piirteet. Yleisrakenteeltaan ne vastaavat tänä päivänä sekä näköpiirissä olevassa tulevaisuudessa 1940-lukulaista perusarkkitehtuuria! Kunkin prosessorin käskykanta ja muu konekieliseen ohjelmointiin tarvittava 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 tai tarvittavan muistin määrä. Tästä lisää algoritmikursseilla!

Hieman realistisempi kuva: moniydinprosessorit, välimuistit

Kuvassa 0.9 on hieman edellistä realistisempi kuva nykyaikaisesta tietokoneesta. Olennainen ero on, että tässä kuvassa keskusyksiköitä eli CPUita on monta rinnakkain. Kutakin niistä sanotaan ytimeksi (engl. *core*) ja kokonaisuutta moniydinprosessoriksi (engl. *multicore processor*). Mikäli prosessorit ovat keskenään samanlaisia ('symmetrisiä'), niin tällaisen tietokoneen yhteydessä puhutaan symmetrisestä moniprosessoinnista (engl. *symmetric multiprocessing*, SMP). Moniydinprosessorien suunnittelu ja valmistaminen on tietysti monimutkaisempaa kuin yhden yksittäisen prosessorin. Myös käyttöjärjestelmien suunnitteluun moniprosessointi tuo omat detaljinsa. Osoittautuu kuitenkin, että normaalien sovellusohjelmien tekijälle ei ole suurtakaan väliä sillä, onko alustalait-

teessa yksi vai monta ydintä. Myöskään tällaisen johdantokurssin asioihin ei tuo merkittävää eroa, onko prosessoreita yksi vai kaksi. Yksinkertaisuuden mielessä voidaan siis jatkossa käsitellä suurimmaksi osaksi yhden prosessorin (tai prosessoriytimen) toimintaa.

Kuvassa on myös täsmennetty muistilaitteiston jakautumista rekistereihin, välimuisteihin, keskusmuistiin ja massamuisteihin. Niistä on syytä kertoa seuraavaksi aivan erikseen.

Muistilaitteistosta: muistihierarkia, prosessorin välimuistit

Havaittiin, että prosessorissa tarvitaan ns. rekisterejä, jotka ovat nopeita muistikomponentteja prosessorin sisällä, mahdollisimman lähellä laskentaa hoitavia komponentteja. Rekisterien määrää rajoittaa kovasti hinta, joka niiden suunnitteluun ja toteutukseen liittyy. 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. Kompromissina prosessoreihin valmistetaan ns. **välimuisteja** (engl. *cache memory*), eli nopeampia (ja samalla kalliimpia) muistikomponentteja, jotka sijaitsevat lähempänä prosessoria ja joissa pidetään väliaikaisesti osaa keskusmuistin sisällöstä. Tämä on järkevää, koska ohjelmat käyttävät useimmiten lähellä toisiaan olevia muistiosoitteita aika pitkään, ennen kuin ne siirtyvät taas käsittelemään joitakin toisia (nekin taas lähellä toisiaan olevia) osoitteita. Riittää vaihtaa tuo uusi 'lähimaasto" välimuistiin edellisen 'lähimaaston' tilalle. Tälle havainnolle on nimikin, **lokaalisuusperiaate** (engl. *principle of locality*). Käytännön tasolla ilmiön taustat on helppo ymmärtää: Ajattele esim. koodin osalta peräkkäin suoritettavia käskyjä, muutamien käskyjen mittaisia silmukoita ja usein toistettavia

aliohjelmia. Datan osalta taas käsitellään suhteellisen pitkään tiettyä tietorakennetta ennen siirtymistä seuraavan käsittelyyn. Metodien sisäiset paikalliset muuttujat ovat nimensä mukaisesti myös paikallisia, koodissa lähekkäin määriteltyjä ja ajallisesti lähekkäin käyteltyjä.

Rekisterejä voi siis olla kustannussyistä vain muutama. Välimuistit maksavat enemmän kuin keskusmuisti, mutta nopeuttavat kokonaisuuden toimintaa. Nykyinen tietokonelaitteisto hoitaa välimuistien toiminnan automaattisesti. Sovellusohjelmien tekijän ei tarvitse huolehtia siitä, ovatko muistiosoitteiden 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 löytyy välimuistista suurimman osan aikaa. 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**(engl. *mass memory*), 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, joissakin prosessoreissa myös Level 3; prosessori- ja muistiteknologia hoitaa välimuistin käytön; ohjelman ei tarvitse, eikä se viime kädessä edes pysty huolehtimaan siitä, onko sen tarvitseman muistiosoitteen sisältö jo välimuistissa vai onko se toistaiseksi kauempana)

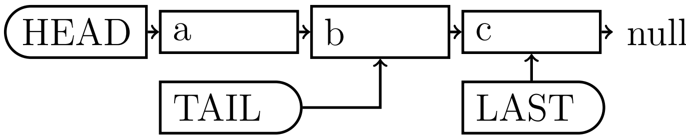
- Keskusmuisti
- Massamuistit kuten kovalevyt

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 mahdollisesti 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. Ohjelmia halutaan siis tyypillisesti suorittaa enemmän kuin keskusmuistiin mahtuu ohjelmien tarvitsemaa dataa. Massamuistit ovat äärimmäisen suuria ja halpoja, mutta myös keskusmuistiin nähden äärimmäisen hitaita. Tästä seuraa tarve jollekin fiksulle järjestelmälle, joka hyödyntää hidasta levytilaa nopean, mutta rajallisen, keskusmuistin apuna. Avain on käyttöjärjestelmän ja prosessorin yhteisesti hallitsema sivuttava virtuaalimuisti, johon syvennytään myöhemmässä luvussa.

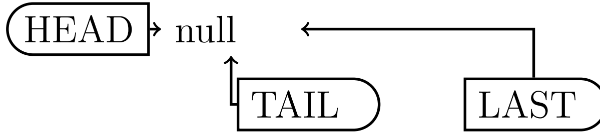
Virtuaalikoneet

TODO: Tähän jossain vaiheessa ehkä jotakin virtuaalikoneista ja virtualisoinnista. (?) Jotain poimintoja esim. aiemmalta vierailuluennolta, joka kertoi JY:n konesalin asteittaisesta virtualisoinnista ja sen hyödyistä? (*todennäköisesti ei vuonna 2016;* kiinnostuneet pyytäkööt vaikka linkin ja polkuavaimen vuoden 2014 vierailuluentoon. Virtuaalikoneita käsitellään kuitenkin tarkemmin myöhemmillä kursseilla, mm. Tietoverkkoturvallisuus.)

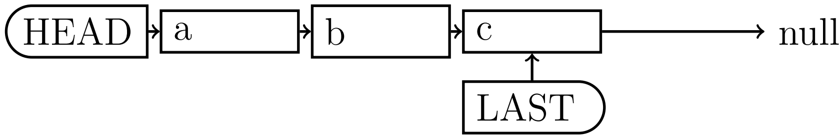
(a) *Lista*



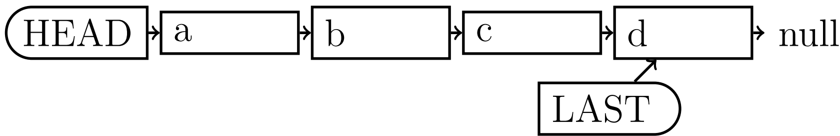
(b) *Tyhjä lista*



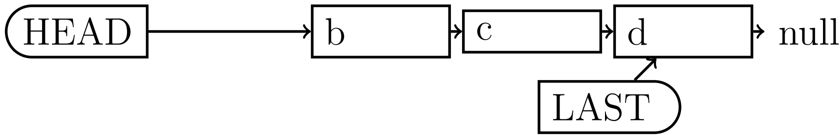
(c) *Jonon voi toteuttaa listana.*



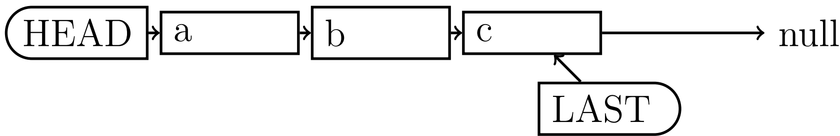
(d) *Lisäys jonoon viimeiseksi alkioksi.*



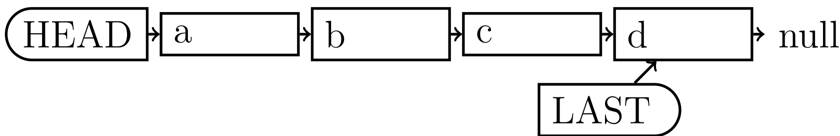
(e) *Käsittely jonosta, eli otetaan alkio jonon keulilta.*



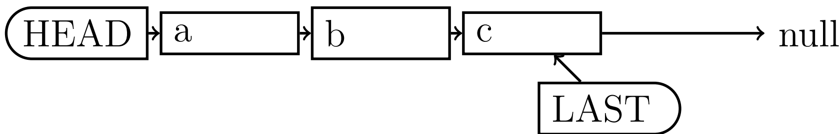
(f) *Pino*



(g) *Lisätään alkio pinon päälle.*



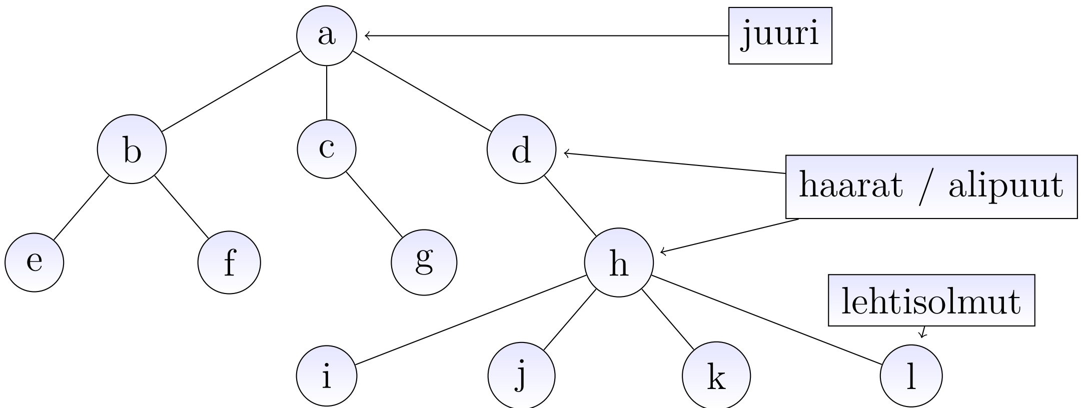
(h) *Käsittely pinosta, eli otetaan alkio pinon päältä.*



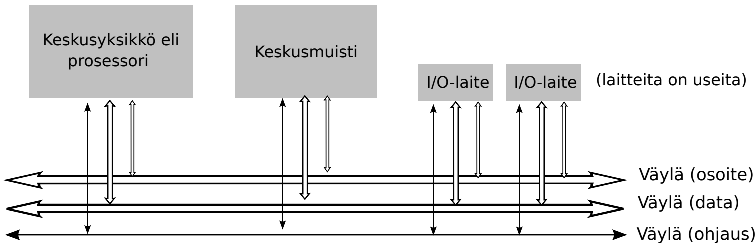
Kuva 0.4: *Lista, jono ja pino periaatepiirroksina.*

"Alipuu"
Lehtiä
Haaroja
Juuri

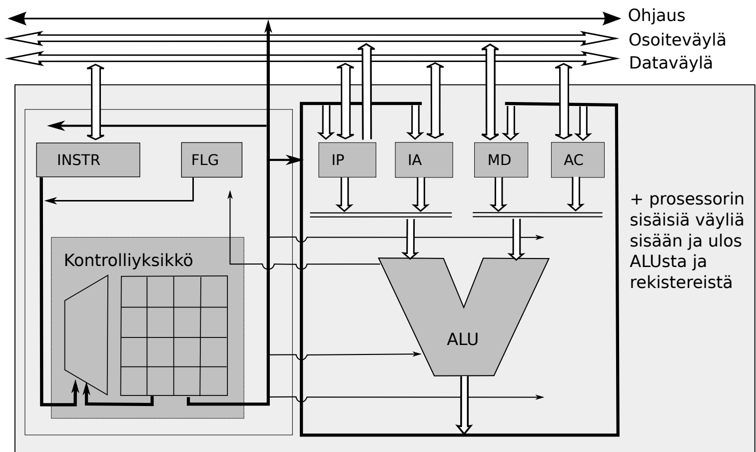
Kuva 0.5: *Reaalimaailman puurakenne periaatepiirroksena*



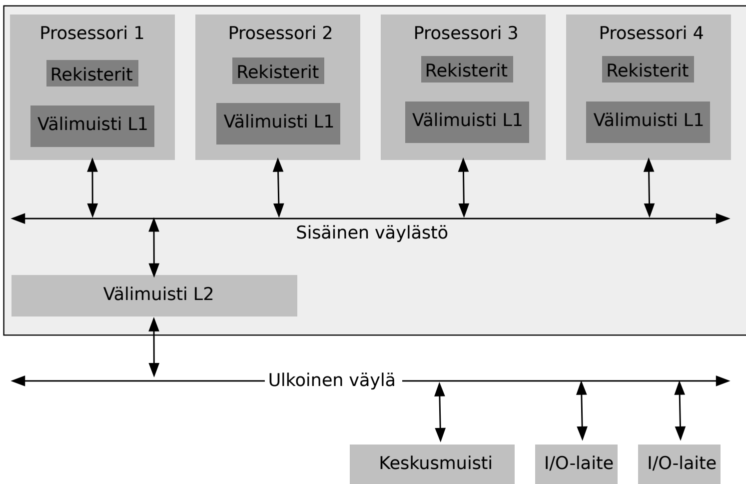
Kuva 0.6: *Puurakenne abstraktina tietorakenteena*



Kuva 0.7: *Yleiskuva tietokoneen komponenteista: keskusyksikkö, muisti, I/O -laitteet, väylä.*



Kuva 0.8: *Yksinkertaistettu kuva kuvitteellisesta keskusyksiköstä: kontrolliyksikkö, ALU, rekisterit, ulkoinen väylä ja sisäiset väylät. Kuva mukailee Tietotekniikan perusteet -luentomonistetta [3], jossa määritellään myös käskykanta-arkkitehtuuri vastaavalle pikkuprosessorille.*



Kuva 0.9: *Yksinkertaistettu kuva modernimman tietokoneen komponenteista: useita rinnakkaisia keskusyksiköitä, keskusmuisti, välimuistit, I/O -laitteet, väylä.*

0.3 Hei maailma – johdattelua tietokoneeseen

Avainsanat: ikkunointijärjestelmä, graafinen shell, tekstimuotoinen shell, tiedosto, lähdekoodi, kohdekoodi (objekti), kääntäminen, linkittäminen, lataaminen, kirjastot, jaetut objektit, tulkkaus, skripti, IDE

Osaamistavoitteet: Luvun luettuaan opiskelija:

- (alustavasti) ymmärtää ohjelman suorituksen muistissa sijaitsevien konekielisten käskyjen jonona sekä tyypillisten ohjelmointirakenteiden ilmenemisen konekielessä; tunnistaa konekielisen ohjelman sellaisen nähdessään (AT&T tai Intel -tyyppisenä assemblynä tai disassemblynä) [ydin/arvos2]
- osaa selittää vaiheet ja mahdolliset ongelmatilanteet suoritettavan ohjelmatiedoston lataamisessa, dynaamisessa linkittämisessä ja käynnistämisessä [ydin/arvos2]

  FIXME: Taitaa vielä puuttua kuvaus noista mahdollisista ongelmatilanteista?

Tässä luvussa kirjoitetaan, jälleen kerran, sovellusohjelma nimeltä ”Hei maailma!”. Ohjelma on sopivan yksinkertainen koeputkiesimerkki, jonka avulla päästään tutkimaan näkymiä tietokoneen osien toimintaan ja käyttöjärjestelmäohjelmiston vastuulla oleviin tehtäviin. Tässä luvussa tehdään vasta esityö ja kerätään havaintoja. Tarkempi ymmärrys on tarkoitus hankkia vasta seuraavissa luvuissa, joissa käydään yksityiskohtia tarkemmin läpi.

Peruskäyttäjän näkökulma: ikkunat, työpöytä, ”resurssienhallinta”

Totuttu tapa ohjelmien käynnistämiseen on klikata niiden kuvaketta jonkinlaisessa valikossa, joka näyttää mahdollisia ohjelmia. Ohjelmat aukeavat sitten yleensä kukin omaan ikkunaansa, jota voi siirrellä näytöllä. Ohjelmilla käsitellään useimmiten tiedostoja, esimerkiksi tekstidokumentteja, valokuvia tai videoita, joita voi tallentaa levyille, muistitikuille ym. tai siirtää verkon yli. Näitä tiedostoja voi selata, siirrellä hakemistosta toiseen, poistaa ym., erityisellä ohjelmalla, jonka nimi on esimerkiksi ”resurssienhallinta”. Tiedostot sijaitsevat hakemistopuussa, jossa niillä on kaikilla yksilöivä nimi ja ”tiedosto-osoite” eli sijainti puussa, joka alkaa jonkin alipuun juuresta – laajimmillaan koko tiedostojärjestelmän juuresta. Toisaalta osoite voi olla suhteessa myös esimerkiksi omaan kotihakemistoon tai vaikkapa yhden kurssin tehtävien vastaustiedostoja sisältävän alipuun juureen. Tiedoston osoitteen sanotaan olevan **absoluuttinen tiedostonimi** (engl. *absolute file name*), jos se ilmoitetaan suhteessa ylimpään juureen ja **suhteellinen tiedostonimi** (engl. *relative file name*), jos se ilmoitetaan suhteessa johonkin muuhun hakemistoon.

Graafisten ikkunoiden hallitseminen tapahtuu tyypillisimmin niin sanotun **ikkunointijärjestelmän** (engl. *window manager*) kautta. Vaikka kyseinen järjestelmä saattaa olla osa käyttöjärjestelmää, se on jo niin korkealla tasolla ytimen yläpuolella, ettei meitä tällä kurssilla liiemmin sen yksityiskohdat kiinnosta.

Nämä tutut ohjelmien käynnistämiseen ja yleiseen tiedostojen hallintaan liittyvät graafiset välineet ovat esimerkki **graafisista kuoriohjelmista** (engl. *graphical shell*), joissa voidaan hiirellä klikkailemalla tai kosketusnäyttöä näppäimellä päästä käsiksi järjestelmässä sijaitseviin kohteisiin. **Kuori** (engl. *shell*) ylipäätään tar-

koittaa matalamman tason järjestelmän ympärille rakennettua rajapintaa, jonka kautta järjestelmän hallitsemia kohteita on yhden pykälän verran helpompi käsitellä.

Käyttäjän näkökulma tällä kurssilla: tekstimuotoinen shell

Tällä kurssilla käsitellään tekstimuotoista kuorta, koska teksti on rajapintana lähempänä tietokoneen ja käyttöjärjestelmän sisäistä "totuutta" kuin kuvakkeet, joten teoreettiset asiat tulevat tekstikuoren kautta konkreettisemmiksi. Toisekseen tekstimuotoisia kuorikomentoja on helpompi kirjoittaa ylös, toistaa ja varioida kuin graafisia klikkailusekvenssejä. Niistä voidaan rakentaa skriptejä, joilla toimintojen sarjoja voidaan yhdistellä, parametrisoida ja toistaa aina tarvittaessa tai automaattisesti tietyin väliajoin. Tekstimuotoiselle kuorelle on myös vakiintuneita käytäntöjä, jotka on mm. POSIX-standardissa kiinnitetty yhteensopivien sovellusten tekemisen pohjaksi. Yhteensopivilla menettelyillä voidaan siis tehdä järjestelmänhallintaan liittyviä automaatioita, jotka toimisivat samalla tavoin esim. Mac OSX:ssä, FreeBSD:ssä ja joissain riittävän hyvin POSIXia tukevissa Linux-jakeluissa.

Keväällä 2015 kuoren eli shellin käyttöä opetellaan ensimmäistä kertaa demoissa 1 ja 2, nimiltään "Superpikaintro interaktiivisen Unix-shellin käyttöön" ja "Superpikajatko interaktiivisen Unix-shellin käyttöön". Kyseiset demot olisi hyvä olla tehtynä siinä vaiheessa, kun tämä luentomonisteen vaihe on käsittelyssä.

"Hello world!" lähdekooditiedostona

C-kieli kiinnostaa tällä kurssilla ainakin neljästä syystä:

- Konkreettisen tietokonelaitteiston toimintaan päästään sen avulla käsiksi ilman käsienheiluttelua, koska C:n lähtökohtainen tavoite ja nykyinen rooli on olla abstraktiotasoltaan matala kieli, joka käsittelee dataa melkein kuin tietokonelaite.

- C-kieli on ollut vahva vaikuttaja myöhemmin kehitettyjen lohkorakenteisten oliokielten, kuten C#:n ja Javan, määrittelyssä, joten C-kieleen tutustuminen antaa historiallista perspektiiviä nykypäivään sekä selityksiä ilmiöille, joita ei oikein pysty selittämään muuten kuin että ”C:ssä tämä piti kirjoittaa näin, joten kirjoitetaan se edelleen näin, vaikka se olisi ehkä fiksumpaa tehdä jollain muulla tavalla”. Yksi esimerkki ”vahvasti C-mäiseksi” tehdystä kielestä on myös esim. tietokonegrafiikassa tarpeellinen GLSL-varjostinkieli.

- Käyttöjärjestelmät saatetaan tehokkuussyistä (ja suoremman laiteohjauksen toteutumiseksi) haluta tehdä C-kielellä. Mm. Linux on pääosin C-kieltä, joten olemassaolevaa käyttöjärjestelmäkoodia C:llä löytyy paljon. Puhtaalta pöydältä aloitettaviin laiteläheisiin projekteihin kannattanee nykyään tutustua korkeamman tasoon kieliin, joille löytyy kääntäjä halutulle prosessoriarkkitehtuurille.

- POSIX-standardi, jota kurssilla käytetään esimerkkinä laiteriippumattomasta käyttöjärjestelmän sovellusrajapinnasta, määrittelee sovellusohjelmien tekemiseksi nimenomaan C-kääntäjän ja tiettyjen alemman tason kirjastojen ominaisuudet.

Seuraavassa on esimerkki perinteisestä, yksinkertaisesta sovellusohjelmasta C-kielellä:

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

```
    printf("Hello world!\n");
    return 0;
  }
```

Erot esimerkiksi C#:lla tehtyyn vastaavaan sovellukseen ovat pieniä, ja niitä mietitään tarkemmin tämän kurssin demossa 3. Demossa 2 havaituin keinoin voidaan varmistua siitä, miltä esimerkiksi tällainen lähdekooditiedosto näyttää tietokoneen näkökulmasta. Kevään 2015 luennoilla käytiin läpi jotakuinkin seuraavaa komentorivisessiota vastaavat esimerkit. Mukana on tässä sekä komennot että niiden tulosteet, jotta lukija voi seurata esimerkin kulkua:

```
[nieminen@halava esimerkit]$ pwd
/nashome3/nieminen/charragit/itka203-kurssimateriaali-avoin/2015/esimerkit
[nieminen@halava esimerkit]$ hexdump -C l04_helloworld.c
00000000  23 69 6e 63 6c 75 64 65  3c 73 74 64 69 6f 2e 68  |#include<stdio.h|
00000010  3e 0a 69 6e 74 20 6d 61  69 6e 28 69 6e 74 20 61  |>.int main(int a|
00000020  72 67 63 2c 20 63 68 61  72 20 2a 2a 61 72 67 76  |rgc, char **argv|
00000030  29 7b 0a 20 20 70 72 69  6e 74 66 28 22 48 65 6c  |){.  printf("Hel|
00000040  6c 6f 20 77 6f 72 6c 64  21 5c 6e 22 29 3b 0a 20  |lo world!\n");. |
00000050  20 72 65 74 75 72 6e 20  30 3b 0a 7d 0a           | return 0;.}.|
0000005d
[nieminen@halava esimerkit]$ stat l04_helloworld.c
  File: "l04_helloworld.c"
  Size: 93              Blocks: 16         IO Block: 131072 tavallinen tiedosto
Device: 1ah/26d Inode: 23581927    Links: 1
Access: (0600/-rw-------)  Uid: (29067/nieminen)   Gid: (   99/  nobody)
Access: 2015-03-26 15:52:58.000000000 +0200
Modify: 2015-03-26 15:54:19.000629000 +0200
Change: 2015-03-26 15:54:19.000629000 +0200
[nieminen@halava esimerkit]$
```

Alussa tulostetaan shellin nykyinen työhakemisto (komento pwd, ”print current working directory”). Se kertoo lähdekooditiedoston *sijainnin hakemistopuussa*, jolla ei ole oikeastaan mitään tekemistä tiedoston *sisällön* kanssa. Tiedostojahan voi siirrellä paikasta toiseen hakemistopuussa tai vaikka eri tietokoneelle verkon yli. Olisi ihan suotavaa, että sisältö ei näissä operoinneissa muutu.

Koko sisältö näkyy seuraavaksi apuohjelman ”hexdump” avulla tasan siten kuin se on: Vasemmassa sarakkeessa on 8-bittisten tavujen juokseva numerointi tiedoston alusta alkaen heksalukuna (indeksit alkavat nollasta, ja koska rivillä näytetään aina 16 tavua, on seuraavan rivin ensimmäinen indeksinumero aina 16 suurempi kuin edellinen - heksana siis 10_{16} eli 0x10 suurempi. Kaikki tiedoston sisältämät tavut, ei mitään vähemmän eikä enemmän, on tässä vedoksessa mukana heksoina. Oikeanpuoleisessa sarakkeessa joka rivillä on tavuja vastaavat 16 merkkiä, mikäli tavujen ilmoittamat luvut osuvat ASCII-merkistöstandardin mukaisiin ”tulostettaviin merkkeihin”. Ei-tulostettavien merkkien kohdalla on piste. Tässä tapauksessa melkein kaikki merkit ovat tulostettavia, mutta rivinvaihtojen kohdalla nähdään koodaus 0x0a, joka siellä tyypillisesti tulee, kun tiedosto on tehty unixiin pohjautuvassa järjestelmässä ASCII- tai UTF8-merkistökoodauksella. Rivinvaihdon, kuten tekstimerkkienkin, koodaus vaihtelee järjestelmien välillä, joten tiedon tuottamisessa käytetty koodaus tulee tuntea. Tavallisen tekstitiedoston sisältöön tieto koodauksesta ei nähtävästikään sisälly! Jos koodauksesta on epäselvyyttä, ei auta kuin katsoa sisältöä vaikkapa heksavedoksena ja yrittää tehdä johtopäätöksiä.

Havaitaan, että myöskään tiedoston *nimi ei kuulu sisältöön*. Samalle numeroina koodatulle datapötkölle voidaan antaa mikä tahansa nimi, eikä se vaikuta sisältöön. On siis aivan sama, onko tiedoston nimi ”helloworld.c”, ”helloworld.cs”, ”helloworld.txt” tai pelkkä ’helloworld”, vaikka monissa sovelluksissa (mukaanlukien monet ohjelmointikielten kääntäjät) onkin sovittu, että nimen pitää olla tietyn muotoinen erityisesti loppuosastaan.

Nimi tai mitään muutakaan kuvailevaa lisätietoa eli *metatietoa* ei tiedoston sisällössä kuitenkaan ole mukana. Ymmärrettävistä syistä tärkeitä tietoja ovat mm. tiedoston käyttöoikeudet, luonti- ja muokkausajankohdat ja koko. Nämä saadaan selvästi kaivet-

tua esille komennolla `ls` tai Linuxissa komennolla `stat` kuten esimerkissä yllä. Käyttöjärjestelmän täytyy ylläpitää kaikkia näitä tietoja, kuten myös tiedoston nimeä ja sisältöä jollakin tavalla. Tapa, jolla kaikki tarvittava tiedostoihin liittyvä tieto organisoidaan ja tallennetaan, on nimeltään **tiedostojärjestelmä** (engl. *file system*), johon liittyy sopimukset tallennustavasta, osoitteista ja mahdollisuuksista asettaa metatietoja. Tiedostojärjestelmiä on kehitetty useita. Niiden ominaisuudet ja käyttötarkoitukset poikkeavat toisistaan, ja uudemmissa on tietysti laajemmat mahdollisuudet kuin vanhemmissa. Järkevän käyttöjärjestelmän täytyy tukea ainakin yhtä tiedostojärjestelmää, koska tiedostojen käsittely on ymmärrettävästi aika tärkeä sovellus. Yleiskäyttöinen käyttöjärjestelmä tukee oletettavasti useitakin erilaisia tiedostojärjestelmiä, jotta vanhoja tai muutoin muissa järjestelmissä tallennettuja tietoja pystytään lukemaan. Osa metatiedoista saattaa tietenkin hukkua matkalla, mikäli tiedosto siirretään yhdestä tiedostojärjestelmästä toiseen, joka ei osaakaan tallentaa juuri samoja metatietoja. Tämän kurssin loppupuolella käsitellään, jos aikaa jää, yhtä konkreettista tiedostojärjestelmää esimerkinomaisesti.

Nyt toivottavasti on poistunut kaikki epäselvyys ja mystisyys siitä, miten tiedostot, esimerkiksi tavallinen teksti ja aivan samalla tavalla myös ohjelmointikielellä kirjoitettu **lähdekoodi** (engl. *source code*), näyttäytyvät tietokoneen näkökulmasta ja miten niitä voi halutessaan tutkia ”konepellin alta”.

Korostettakoon nyt vielä tässäkin, että laitteisto ei ”ymmärrä” tiedosta tai tiedostoista mitään muuta kuin pötkön bittejä. Kaikki sitä korkeamman tason asiat hoitaa ohjelmisto – matalimmalla tasolla siis käyttöjärjestelmäohjelmisto. Erityisesti tuollainen ihmisen ymmärtämällä korkean abstraktiotason kuvauskielellä kirjoitettu Hei maailma -lähdekoodi ei sellaisenaan pysty ohjaamaan tietokoneen toimintaa. Se pitää kääntää lähteestä **kohdekoodiksi**

(engl. *object code*), joka on tyypillisesti jonkin yksinkertaisemman järjestelmän ymmärtämää. Esimerkiksi C-kääntäjän tehtävänä on kääntää lähdekoodi kohde- eli objektikoodiksi, jota jokin tietty tietokonelaitteisto ymmärtää suoraan, ts. tietyn prosessoriarkkitehtuurin mukaiseksi konekielikoodiksi.

"Hello world!" lähdekoodista suoritukseen

Ohjelmia ei voi kirjoittaa kerralla alusta loppuun, sana sanalta järjestyksessä. Sen sijaan ohjelmaan saatetaan tehdä ensin runko, jonka sisältöä myöhemmin muokataan ja siihen lisätään aina pätkä kerrallaan, kunnes ohjelman oikeellisesta ja tavoitteen mukaisesta toiminnasta voidaan olla riittävän varmoja[19]. Tavoitteet puolestaan muuttuvat, joten luonnostaan ohjelmia täytyy muokata sieltä täältä aina seuraavaa versiota varten, eivätkä ne siten ole koskaan valmiita[20]. Ihminen tekee luonnostaan huolimattomuus- ja ajatusvirheitä, aina väärien näppäimien painamisesta lähtien. Näin ollen ohjelmakoodia pitää riittävän usein testata käytännössä. Vähintäänkin se pitää **kääntää** (engl. *compile*) ja **ajaa** (engl. *run*), ennen kuin voi tietää, että ohjelma on edes syntaksiltaan oikein.

Kevään 2015 luennolla esiteltiin koeputkiesimerkkinä vaiheittain C-kielisen Hei maailma -ohjelman synty ja mm. kääntäjän varoitukseen reagoiminen korjaamalla ohjelmaa. Lopputuloksena lopul-

[19]Tällä kurssilla emme mene siihen, kuinka ohjelmistojen tavoitteita yleisesti ottaen (muuten kuin käyttöjärjestelmien osalta) määritellään ja niiden toteutumista mitataan tai kuinka ohjelmistoja tai niiden kehitystyötä organisoidaan tavoitteiden toteuttamiseksi; syventävät opintojaksot vaatimuksista, testauksesta, arkkitehtuureista ja projektimalleista kertovat näistä erikseen.

[20]Ellei tekijä testamenttaa ohjelmistoa määritelmän mukaan valmiiksi, kuten Donald Knuth ilmeisesti on nettisivunsa mukaan tehnyt TeX ja METAFONT -järjestelmilleen: `http://www-cs-faculty.stanford.edu/~uno/abcde.html`

ta ohjelma vielä kerran käännettiin ja suoritettiin seuraavan komentorivisession mukaisesti:

```
[nieminen@halava esimerkit]$ c99 -g -o helloworld.suoritettava l04_helloworld.c
[nieminen@halava esimerkit]$ ./helloworld.suoritettava
Hello world!
[nieminen@halava esimerkit]$
```

IDEssä tällainen sekvenssi on tietysti automatisoitu klikkauksen tai pikanäppäimen taakse. Tällä kurssilla nähdään, mitä konepellin alla tapahtuu: kääntäjätyökalu suoritetaan ja sitä ohjataan jollain tapaa, esimerkiksi komentoriviargumenttien avulla. Tässä optiolla `-g` pyydetään sisällyttämään käännettyyn tiedostoon lisätietoja debuggausta varten, mm. viittaukset lähdekooditiedostoon. Kaupan hyllylle toimitettavassa ohjelmassa esim. tätä vipua kenties ei olisi, mutta ohjelmistokehittäjän omissa käännöksissä kylläkin. Esimerkiksi juuri tällaisia eroja IDE saattaa konepellin alla tehdä riippuen tehdäänkö käännös 'Release' -valinnalla tai 'Development' / 'Debug' -valinnalla. Lisäksi argumenttiparilla `-o helloworld.suoritettava` valitaan kuvaava nimi käännetylle ohjelmalle.

Edellä katsottiin, miten tiedostossa oleva lähdekoodi näyttäytyy tietokoneen näkökulmasta – sehän oli numeroita, jotka kuvaavat merkkejä tietyn merkistökoodauksen mukaisesti. Miten sitten näyttäytyy suoritettava konekielinen ohjelma? Tiedoston sisältöä voidaan tietenkin tutkia jälleen komentoriviltä esimerkiksi hexdump-ohjelmalla. Tiedoston ensimmäiset 64 tavua näyttävät seuraavalta:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  e0 03 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  d8 0c 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  08 00 40 00 25 00 22 00  |....@.8...@.%.".|
...
```

Tiedoston alussa on neljä ”taikamerkkiä”, jotka lupailevat sen lukijalle, että loppuosa koostuu täsmälleen tietyssä standardissa (Executable and Linkable File Format, ELF) määrätyistä osioista, joilla on sovittu sisältö. Kiinnostunut lukija löytänee tarkempia tietoja esimerkiksi Internetistä hakemalla. Heksavedoksen selailustakin voi päätellä, että tiedostossa on mm. toisteisia osioita, jotka itse asiassa ovat määrämittaisten tietorakenteiden muodostamia taulukoita. Lisäksi siellä on selväkielisiä merkkijonoja, kuten versiotietoja käytetystä kääntäjäohjelmasta. Osa merkkijonoista näyttäisi olevan tiedostosijainteja. Käyttöjärjestelmien mielessä mielenkiintoinen suoritettavaan ohjelmatiedostoon sisältyvä merkkijono on mm. ”/lib64/ld-linux-x86-64.so.2”, joka löytyy luentoesimerkissä ”jännästi” tasan 512 tavun päässä (heksana 0x200) tiedoston alusta lukien:

```
00000200  2f 6c 69 62 36 34 2f 6c  64 2d 6c 69 6e 75 78 2d  |/lib64/ld-linux-|
00000210  78 38 36 2d 36 34 2e 73  6f 2e 32 00 04 00 00 00  |x86-64.so.2.....|
```

Tietysti myös teksti ”Hello world” löytyy suoritettavan tiedoston sisältä jostakin kohtaa, kuten olettaa sopii.

Nyt voidaan ennakoivasti hahmottaa muutamaa yksityiskohtaa vaille, mitä esimerkiksi Linux tekee, kun shellin kautta annetaan komentona suoritettavan ohjelmatiedoston nimi: Se nimittäin

- koettaa hakemiston ja nimen perusteella etsiä tiedostoa tiedostojärjestelmästä
- varmistaa tiedostojärjestelmän erikseen tallennetuista metatiedoista, että käyttäjällä on suoritusoikeus tiedostoon
- tutkii tiedoston sisällön alusta muutaman taikanumeron ja päättää minkä ohjelman se käynnistää seuraavaksi:

- Jos tiedoston alussa on heksat 0x23 0x21 eli ASCII-merkistön merkit #!, käyttöjärjestelmä lukee tiedostoa ensimmäiseen rivinvaihtoon asti ja käynnistää huutomerkin ja risuaidan väliin kirjoitetun ohjelman, jolle annetaan koko tiedosto syötteeksi. Nyt ymmärretään, miksi demossa 2 piti aloittaa skripti kirjoittamalla ensimmäiselle riville #!/bin/bash. (Tämä ei ole POSIXin mukaan aivan sallittua, mutta se tekee jotkut meidän Linux-esimerkeistä helpommiksi)
- Jos tiedoston alussa on heksat 0x7f 0x45 0x4c 0x46, joista viimeiset siis merkkijono ”ELF”, käyttöjärjestelmä käynnistää lataajaohjelman, jolla ohjelma haluaa itsensä ladattavan. Esimerkkimme tapauksessa ohjelmaan on tallennettu merkkijono ”/lib64/ld-linux-x86-64.so.2” juuri tätä tarkoitusta varten. Kyseinen lataajaohjelma sitten käynnistyy ja koko suoritettava tiedosto menee syötteeksi lataajalle. Nyt ehkä myös ymmärretään, miksi yhdelle käyttöjärjestelmälle käännettyä suoritettavaa ohjelmakoodia ei ainakaan suoraan pysty käynnistämään toisessa käyttöjärjestelmässä. (Toki esimerkkinä on nyt vain Linux, mutta vastaavia määritelmiä ja komponentteja on muissakin käyttöjärjestelmissä, eivätkä ne tosiaan ole keskenään suoraan yhteensopivia).
- Muussa tapauksessa ainakin jotkut Linuxit nähtävästi olettavat, että ohjelma on tavallinen shell-skripti, olipa sen alussa #! tai ei.

Jokainen näistä vaiheista voi epäonnistua monista syistä. Esimerkiksi:

- Pyydetyn nimistä tiedostoa ei löydy paikoista, joita sitä etsitään. (virheilmoitus esim. ”command not found”)
- Käyttäjällä ei ole suoritusoikeutta tiedostoon. (virheilmoitus esim. ”Permission denied”)
- Löytyneen ELF-tiedoston muoto ei vastaa sitä, mitä Linux edellyttää. Näin käy esimerkiksi, jos ohjelma on käännetty eri prosessoriarkkitehtuuria varten kuin millä sitä yritetään suorittaa.
- ELFissä ilmoitettua latausohjelmaa tai skriptissä #!:lla ilmoitettua tulkkia ei syystä tai toisesta ole olemassa tai sitä ei pysty käynnistämään. (virheilmoitus esim. ”bad interpreter”)
- Muita ongelmia voidaan ymmärtää myöhemmin paremmin, kun ymmärretään lisää järjestelmän rajoitteista (mm. tarvittavia kirjastoja ei välttämättä ole asennettu, maksimimäärä prosesseja saattaa olla jo käynnissä, muisti voi olla täynnä, laitevika on aina mahdollinen. . .)

Suoritettava ohjelmakoodikin on tiedostossa ollessaan vielä vain pötkö määrämuotoon (tässä ELF) aseteltua dataa. Jotta päästään näkemään, miltä ohjelma näyttää lataamisen jälkeen tietokoneen muistissa, on käytettävä ”intrusiivista” työkalua, joka kertakaikkiaan antaa käyttöjärjestelmän ladata ohjelman suoritusvalmiiksi, mutta ottaa ohjelman sitten kontrolliinsa analysointia varten. Päätarkoitus on virheiden etsiminen ja korjaaminen, joten historiallisista syistä tällaista ohjelmaa sanotaan ”virheenpoistajaksi” eli **debuggeriksi** (engl. *debugger*). Muita käyttötarkoituksia voi olla esimerkiksi hakkerointi, toimintaperiaatteen selvittäminen ilman alkuperäistä lähdekoodia (”reverse-engineering”). Äärimmäisessä hätätilanteessa, jos jonkin elintärkeän ohjelman toiminta ei

jostain syystä saa missään nimessä katketa hetkeksikään, voi siitä debuggerilla periaatteessa korjata vian tai toimintahäiriön ronkkimalla suoraan muistiin ladattua ja toiminnassa olevaa koodia tai dataa. Tehotyökalu siis on kyseessä. Arvatenkin tällä kurssilla käytämme debuggeria komentoriviltä. Tarkempia ohjeita tulee demoissa. Luennolla nähtiin seuraavanlainen debuggerilla tehty tuloste Hei maailma -ohjelmasta:

```
(gdb) disassemble /mr main
Dump of assembler code for function main:
2       int main(int argc, char **argv){
   0x00000000004004c4 <+0>:     55              push   %rbp
   0x00000000004004c5 <+1>:     48 89 e5        mov    %rsp,%rbp
   0x00000000004004c8 <+4>:     48 83 ec 10     sub    $0x10,%rsp
   0x00000000004004cc <+8>:     89 7d fc        mov    %edi,-0x4(%rbp)
   0x00000000004004cf <+11>:    48 89 75 f0     mov    %rsi,-0x10(%rbp)

3         printf("Hello world!\n");
   0x00000000004004d3 <+15>:    bf e8 05 40 00 mov    $0x4005e8,%edi
   0x00000000004004d8 <+20>:    e8 db fe ff ff callq  0x4003b8 <puts@plt>

4         return 0;
   0x00000000004004dd <+25>:    b8 00 00 00 00 mov    $0x0,%eax

5       }
   0x00000000004004e2 <+30>:    c9              leaveq
   0x00000000004004e3 <+31>:    c3              retq

End of assembler dump.
```

Tässä tulosteessa näkyy rivinumeroin varustettuna pääohjelman C-kieliset koodirivit. Kunkin koodirivin alla on siihen riviin liittyvät konekieliset käskyt, jotka kääntäjäohjelma on tuottanut. Vasemmassa sarakkeessa on heksanumerona muistiosoite, eli 8-bittisten tavujen juokseva numerointi, niistä muistipaikoista, joissa ohjelman konekieliset käskyt suorituksen aikana sijaitsevat. Lukemista helpottaa suluissa kymmenjärjestelmän lukuna ilmoitettu siirros

aliohjelman ensimmäisen käskyn sijainnista alkaen. Seuraavassa sarakkeessa on konkreettiset konekieliset käskyt, eli tavun tai muutaman tavun mittaiset numerosarjat, joiden perusteella prosessori tekee aina jotakin hyvin yksinkertaista kerrallaan. Konkreettisten konekielisten tavusarjojen jälkeen tulosteessa on sama käsky **symbolisella konekielellä** eli **assemblerilla** (engl. *assembly language*) ilmaistuna.

Symbolinen konekieli käyttää käskyistä symboleita, joiden on tarkoitettu olevan ihmisen ymmärrettävissä ja kirjoitettavissa. Käyttöjärjestelmästä täytyy kirjoittaa assemblerilla aivan kaikkein matalimman abstraktiotason osuus, jonka tarvitsee kajota suoraan laitteiston konkreettisiin osiin, kuten esimerkiksi juuri tiettyyn rekisteriin. Myös prosessoriarkkitehtuurin dokumentaatio käyttää tyypillisesti assembleria selväkielisenä versiona prosessorin ominaisuuksien ja käskyjen toiminnan selittämiseksi. Manuaali kertoo tietenkin myös, miten nämä assemblerin tasolla kuvatut toiminnot muutetaan konkreettiseksi konekieleksi, jota valmistajan tekemä laite pystyy suorittamaan. Konekielisten tavujonojen tasolla laitteen rajapinta tarvitsee tuntea käytännössä vain, kun tehdään laitteelle kääntäjäohjelmistoa tai yritetään selvittää metatietojen puutteessa jostakin konekielisestä ohjelmanpätkästä, että millähän laitteella sitä mahdollisesti olisi tarkoitus ajaa. Esimerkiksi kaikissa ohjelmissa yleinen aliohjelmiin siirtyminen tai ns. pinomuistin käyttö voi tapahtua kussakin prosessoriarkkitehtuurissa leimallisella tavuyhdistelmällä, josta voi päätellä, mikä prosessoriarkkitehtuuri on kyseessä.

Ohjelman kääntäminen, objekti, kirjasto, linkittäminen ja lataaminen

Edellä nähtiin, kuinka tekstitiedostosta saatiin luotua suorituskelpoinen ohjelma, joka selvästi toimii niin kuin sen pitikin. Lisäksi katseltiin, miltä nämä tiedostot päällisin puolin näyttivät. Kaivetaan nyt vielä kerrosta syvemmältä: Mitä oikein tapahtui missäkin vaiheessa ja miksi?

Kääntäjäohjelma käynnistettiin komennolla `c99`. Tämän nimisen työkalun olemassaolohan on jotakin, minkä POSIX-standardi lupaa yhteensopivassa järjestelmässä. Komennolla pitää käynnistyä nimenomaan C99-standardin toteuttava C-kääntäjä.

Käytännössä Jyväskylän yliopiston suorakäyttökoneella keväällä 2015 tämä komento on itse asiassa shell-skripti! Kyseinen skripti löytyy tiedostosijainnista `/usr/bin/c99`, ja voit vaikka tulostaa skriptin sisällön ja ihmetellä, miten se on toteutettu. Se varmistaa, että käyttäjä ei ole argumenteillaan pyytänyt minkään muun standardiversion kuin C99:n mukaista C-käännöstä. Sitten se käynnistää GNU-projektin tekemän työkalun nimeltä `gcc` ja varmistaa, että kyseiselle ohjelmalle annetaan argumenttina `-std=c99` sekä muut skriptin suorittajan antamat argumentit. Tuo varsinainen konekielinen kääntäjäohjelma tiedostosijainnissa `/usr/bin/gcc` hoitaa sitten loput ja käyttäytyy kuin C99-kääntäjä, koska sille on annettu standardin määrittävä argumentti.

Kääntäjäohjelma `cc` tekee tässä tapauksessa konepellin alla muutakin kuin pelkän käännöksen. Hieman tarkemmin sanottuna:

- Yksittäinen C-kielinen lähdekooditiedosto käy ensin läpi ns. *esikäännöksen*, jossa poistetaan kommentit ja avataan ja muokataan koodia aika paljon. Yksinkertaisimpana esimerkkinä

’Hei maailma’ -ohjelman ensimmäisen rivin sisältö **#include<st** korvautuu tässä vaiheessa koodilla, joka luetaan otsikkotiedostosta ’stdio.h’. Myös otsikkotiedosto voi lukea muita otsikkotiedostoja, jotka kertakaikkiaan liitetään mukaan siihen kohtaan, missä tulee vastaan **#include<tiedostonimi>**. Tässä tapahtuu myös ns. *makrojen* avaaminen, mutta siitä lisää myöhemmin ja demojen yhteydessä. Esikäännöksen jälkeen koodi voi olla paljon alkuperäistä pidempi, eikä se sisällä enää yhtään kommenttia eikä yhtään esikääntäjän ohjauskomennoksi tarkoitettua riviä, jotka alkavat risuaidalla ’#’. Tämä on aina C-käännöksen ensimmäinen vaihe.

- Samaan aikaan tai heti sen jälkeen koodi jäsennetään C:n syntaksin mukaisesti *yhdellä* läpikäynnillä koodin alusta sen loppuun – tämä on syy siihen, että C:ssä on pakko määritellä kunkin nimen luonne (eli onko se tyyppi, muuttuja vai aliohjelma) *ennen* kuin nimeä käytetään muihin tarkoituksiin kuin ensimäärittelyyn. C-kääntäjä ei yksinkertaisesti tiedä, mitä nimi tarkoittaa, mikäli sen määrittely tulee koodissa jäljempänä. Siksi on tyypillistä kirjoittaa aliohjelmat ennen pääohjelmaa, kirjastojen otsikkotiedostot on luettava mukaan **#include**:lla heti lähdekooditiedoston alussa jne.

- Kääntäjä saattaa muodostaa jonkin laitteistoriippumattoman välikielisen muodon eli jonkinlaisen ’meta-assembler’ -koodin, joka vastaa melko läheisesti nykyisten tietokoneiden konekielten yhteisiä piirteitä, mutta on kuitenkin vielä hieman yleisemmässä muodossa. Välikielen käyttö helpottaa kääntäjän muokkaamista uuden prosessoriarkkitehtuurin mukaiseksi, koska konkreettinen konekieli on tuotettavissa välikielestä pienin muunnoksin verrattuna siihen, että pitäisi tuottaa sitä suoraan C-koodista. Välikieli selkeyttää myös uusien ohjelmointikielten lisäämistä kääntäjäinfrastruktuuriin: jos mille

tahansa ohjelmointikielelle saadaan aikaan välikielinen käännös, niin ohjelmat saadaan välittömästi toimimaan kaikissa konkreettisissa prosessoriarkkitehtuureissa, joille välikieltä voidaan käyttää. Tämä on jälleen yksi esimerkki kerrosmaisen rakenteen käyttökelpoisuudesta.

- Kääntäjä tuottaa välikielestä kohteena olevan konkreettisen prosessoriarkkitehtuurin mukaista konekieltä. GNU-kääntäjä itse asiassa lisää tähän kohtaan vielä yhden kerroksen: se tuottaa ensin prosessoriarkkitehtuurin mukaista assembleria, joka lopulta käännetään konekieleksi assembler-kääntäjällä.

- Lopullinen objektitiedosto on assembler-kääntäjän tuottama. Siihen mennessä C-koodin (tai muun kielen) semantiikka on muuttunut 'putkiaivoisen' assembler-kääntäjän ymmärtämään muotoon, jossa ohjelma on jaettu dataa ja koodia sisältäviin pätkiin, joissa jokainen rivi muuttuu pienehköksi pötköksi peräkkäisiä tavuja joko dataosioon tai koodiosioon[21]. Objektitiedoston sisältämät tavut on organisoitu tiettyjen muotosääntöjen mukaan – meidän esimerkissämme ELF-formaatin mukaan.

- Objektitiedoston ei välttämättä tarvitse vielä olla suoritettavissa. Viimeisenä vaiheena tapahtuu nimittäin objektien, eli ohjelmaosioiden, liittäminen toisiinsa eli linkitys eli **linkittäminen** (engl. *linking*).

- Varsinkin kirjastot on suotuisaa liittää mukaan vain ennakoivana pyyntönä lataus- tai suoritusvaihetta varten.

Kunkin vaiheen suorittaa erillinen osio kääntäjäohjelmasta tai jopa erillinen apuohjelma, jonka kääntäjän julkisivuohjelma `gcc` käyn-

[21] Jako dataan ja koodiin on pieni yksinkertaistus; data ja koodi voivat jakautua hienojakoisempiinkin osioihin, mutta kahdella pärjää yksinkertaisissa tilanteissa

nistää kunkin vaiheen tarpeisiin. Useimmiten ohjelmat ovat niin laajoja, että ne on tarkoituksenmukaista jakaa useisiin lähdekooditiedostoihin. Tyypillistä on, että kukin lähdekooditiedosto käännetään ensin erilliseksi, ei vielä suoritettavaksi, objektitiedostoksi, ja sitten lopuksi kaikki nämä objektitiedostot liitetään toisiinsa linkitysvaiheessa. Välivaiheissa syntyviä tiedostoja kääntäjä säilyttää järjestelmän hakemistossa `/tmp/`, mutta tuhoaa ne siinä vaiheessa, kun lopulliset objektit on kirjoitettu käyttäjän haluamaan paikkaan.

Edellä mainittiin, että linkitys voidaan tehdä ohjelman viimeistelyvaiheen lisäksi myös ennakoivana pyyntönä latausvaiheen osalta. Tämä on tärkeää varsinkin kirjastojen osalta. Olisi suurta resurssihukkaa liittää isot ja yleisesti käytetyt apukirjastot, mukaanlukien C:n standardikirjastot tai monessa ohjelmassa käytettävät graafiset kirjastot mukaan kaikkiin niitä tarvitseviin ohjelmiin! On paljon kompaktimpaa, jos kirjastot asennetaan tietokoneelle yhteiskäyttöön ja ne liitetään ohjelmiin vastsa siinä vaiheessa, kun ohjelmat ladataan muistiin suoritettavaksi prosessiksi – siinä vaiheessahan niitä vasta tarvitsee käyttää. Tällöin puhutaan **dynaamisesta linkittämisestä** (engl. *dynamic linking*) ja siitä on tullut nykyinen maailman tapa. Unix-maailmassa dynaamisesti linkitettävän kirjaston nimi on 'jaettu objektitiedosto' (engl. *shared object, "so"*) ja Windows-puolella 'dynaamisesti linkitettävä kirjasto' (engl. *dynamically linked library, DLL*). Molempien termien etymologia lienee edeltävän selostuksen perusteella järkeenkäypä. Kokenut ohjelmoija yleistää päässään kumman tahansa sanan samaan käsitteeseen.

Kun käyttöjärjestelmä käynnistää suoritettavan ohjelmatiedoston prosessiksi, se oikeastaan hyödyntää osiota tai työkaluohjelmaa nimeltä **lataaja** (engl. *loader*), jonka tehtäväksi käännöslinkkeri on jättänyt dynaamisten kirjastojen yhdistämisen. Käynnistysvai-

heessa tapahtuu jotakuinkin seuraavaa:

- Ensinnäkin tutkittuaan suoritettavaksi tarkoitettua tiedostonimeä, käyttöjärjestelmä on tullut siihen tulokseen, että nimen mukainen tiedosto aivan oikeasti on suoritettavissa (se on olemassa ja luettavissa tiedostojärjestelmässä, tunnistetussa suorituskelpoisessa formaatissa ja käyttöoikeuksiltaan suoritusta pyytäneen käyttäjän sallituissa rajoissa). Meidän esimerkissämme formaatti on ELF, kuten Linuxissa yleensä. Windowsissa formaatti olisi Portable Executable eli PE-formaatti.

- Sitten alkaa lataaminen: ELF (ja Windowsissa PE) -formaatti määrittelee merkityksen tiedoston datapötkön osioille, jotka lataaja lukee ja tulkitsee.

- ELF-tiedostoon sisältyy paljonkin tietoja, elintärkeimpinä datan ja koodin sijainti tiedostossa, nämä ovat kokonaislukuja, jotka kuvaavat tavujen sijaintia tiedoston alusta lukien sekä vaatimukset siitä, mihin muistiosoitteisiin nämä pätkät tulisi sijoittaa.

- Lataaja varaa järjestelmästä muistia niin paljon kuin ohjelma ilmoittamiensa tietojen mukaan tarvitsee. Sitten se lukee kunkin ohjelmaosion muistiin ja laittaa osiot sijaitsemaan muistiosoitteissa, joihin ELF-tiedosto ne toivoo. Tässä vaiheessa täytyy herätä relevantti kysymys moniajosta: Eivätkö yhtäaikaa suoritettavat ohjelmat voi mennä ikävästi keskenään ristiin, jos ne vahingossa pyytävät niiden koodia latautumaan samoihin muistiosoitteisiin? Tämä ei ole kuitenkaan millään tavalla ongelma, koska muistiosoitteet ovat ns. virtuaalisia. Aiheeseen palataan vahvasti, mutta tässä vaiheessa on pakko vielä hetkeksi jättää kysymys kiusaamaan mieltä.

- ELFiin sisältyy myös tieto niistä dynaamisista kirjastoista, joita ohjelma haluaisi käyttää. Lataaja voi ladata myös pyydetyt dynaamiset kirjastot (”epälaiska linkittäminen”), tai vähintään se tallentaa niiden sijainnista tiedot, joiden perusteella lataamista voidaan yrittää siinä vaiheessa, kun ohjelma haluaa ensimmäisen kerran käyttää kirjaston palveluita (”laiska linkittäminen”). Lataaja päättää, mihin muistiosoitteeseen kirjastojen koodi ja data tulevat niitä hyödyntävän prosessin käytettäviksi. Kirjastotiedostot ja niistä tarvittavat aliohjelmat ja vakiodatat ilmaistaan ELFissä merkkijonosymboleina (”selväkieliset” nimet, joita objektitiedostossa voi nähdä)

- Objektitiedostossa ei voi olla konkreettisia numeroita muistiosoitteina, ainakaan kirjastoihin viittaavia osoitteita, koska kirjastojen muistisijaintia ei ole mahdollisuutta tietää siinä vaiheessa, kun ohjelma käännetään. Lataajan tehtävänä on siis samalla yhdistellä oikein kaikki ohjelman tarvitsemat muistiosoiteviitteet. Osa sovellusohjelman koodista voi viitata ennalta laskettuihin osoitteisiin, mutta tämä edellyttää, että ohjelma ladataan aina juuri tiettyyn osoitteeseen; dynaamisten kirjastojen koodissa ei mitenkään voi olla konkreettisia ”kovakoodattuja” muistiosoitteita, koska ne ladataan aina tilanteen mukaan vapaana oleviin osoitteisiin.

- Kun kaikki on valmista, lataaja päästää ohjelman rullaamaan muistiosoitteesta, jonka ELF ilmoitti halutuksi **sisäänmenopisteeksi** (engl. *entry point*). Siis prosessorin `IP`-rekisteriin (meidän esimerkkilaitteessamme nimeltään `RIP`) ladataan sisällöksi ELFissä ilmoitettu aloitusosoite, jolloin kontrolli siirtyy ladatulle käyttäjän ohjelmalle. Tässä vaiheessa ohjelman suoritusta sanotaan prosessiksi, kun koodi on ladattu ja käynnistetty ja se rullailee omana instanssinaan.

Tässä listassa oli melko karkea yleistys tapahtumista ohjelmakoodin käynnistämisessä. Mieleen pitäisi jäädä, että ohjelman tie lähdekoodista suoritukseen on monivaiheinen ja että ohjelma ilmenee matkalla varsin erilaisissa muodoissa vielä senkin jälkeen, kun se on käännetty suoritettavaksi. Lopullinen konekielimuoto on olemassa vasta sen jälkeen, kun lataaja on käynnistyksen yhteydessä laittanut paikoilleen oikeat muistiosoitteet, kuten esimerkiksi 'Hei maailma':n ensimmäisen H-kirjaimen osoitteen. Linkittämisen ja lataamisen yksityiskohtia voi halutessaan opiskella vaikkapa kirjasta Linkers and Loaders [7], mikäli aihe alkaa kiinnostaa enemmän.

Käännös- ja linkitysjärjestelmät, IDE:t

Ohjelmointi 1:ltä tuttu tapa ohjelmien tekemiseen on C#:n IDE[22]. Ohjelmointi 2:lla vastaan tulee uusi erilainen, mutta toiminnoiltaan hyvin samankaltainen IDE. Tieteellistä laskentaa tehdään paljon Matlabilla, joka on yhdenlainen IDE. IDE:t organisoivat ja visualisoivat koodia ohjelmakehittäjän kannalta hyödyllisellä tavalla ja automatisoivat mahdollisimman pitkälti toimintoja, kuten kääntämistä, linkittämistä, lataamista.

Tällä kurssilla päästään näkemään konkreettisesti, mitä myös IDE:t tekevät konepellin alla: Lähdekoodi, debuggaustieto ja suoritettavat kehitys-/tuotantoversiot sijaitsevat tiedostojärjestelmässä paikoissa, joista IDE pitää huolta grafiikkansa takana. Ohjelmat tarvitsevat toimiakseen kirjastoja, joiden sijainnista ja löytymisestä IDE:n on pidettävä huolta. Erilaisia säätöjä tarvitaan vähintään kehitys- ja tuotantoversion välillä. Nämä klikataan graafisen käyttöliittymän kautta, mutta varsin usein lopputulemana IDE käynnistelee kääntäjä- ja linkitysohjelmia erilaisilla komentoriviargu-

[22] Tilanne Jyväskylän yliopistossa, keväällä 2016.

menteilla. IDEn tarvitsemat säädöt ja tieto kuhunkin projektiin liittyvien tiedostojen sijainneista ovat tallessa jonkinlaisessa kuvaustiedostossa tai useammassa.

Samat vaiheet on mahdollista, ja joskus pakkokin, tehdä suoraan komentoriviltä, jos vaikka jonkun ongelman syy liittyy välivaiheeseen, jonka tulosta on vaikea hahmottaa IDE:n käyttöliittymästä. Demossa 4 käydään läpi useita lähdekooditiedostoja sisältävän C-kielisen ohjelman kääntämistä suoraan komentoriviltä. Käytännössä homma kuitenkin tarvitsee apuvälineitä, kun ohjelman laajuus kasvaa mielenkiintoisemmaksi. Minimaalinen apu ohjelman lähdekooditiedostojen automaattiseen kääntämiseen ja linkittämiseen on ohjelmantekotyökalu `make`, jonka käyttöä demossa 4 myös katsotaan alustavasti. Projektin sisältö kuvataan silloin tietyn muotoisessa ns. tekemisohjetiedostossa, jonka nimi on tyypillisesti 'Makefile'. Kun makefile on kunnossa, isommankin ohjelman muunnos lähdekoodikasasta toimivaksi ohjelmaksi tapahtuu komentamalla shellissä `make` – ja muuta ei tarvita. Ilman IDEä tehtävään ohjelmien kääntämiseen on yleisessä käytössä vielä make-työkaluakin tehokkaampia välineitä, mutta jätetään ne sitten omatoimisen tai työtehtävissä tapahtuvan täsmäopiskelun varaan. Suurin osa ohjelmistokehityksestä tapahtuu kuitenkin todennäköisesti jossakin IDEssä[23].

Ohjelman toimintaympäristö

Ohjelman toimintaan voi vaikuttaa käynnistyksen yhteydessä komentoriviargumenteilla ja lisäksi ympäristömuuttujilla. Näiden käyttö on tarkoitus oppia C-kielen osalta esimerkin kautta demossa 3 ja shell-skriptien osalta myöhemmässä demossa.

[23]. . . vaikka voidaan myös sanoa, että tehokkaan tekstieditorin ja sopivien apuohjelmien avulla Unix-tyyppinen järjestelmä sisältää ohjelmakehitykseen soveltuvan tekstimuotoisen IDEn, joka ei juurikaan häviä tehokkuudessaan graafisille sisarilleen.

Käännettävät ja tulkattavat ohjelmat; skriptit

Edellä nähtiin, että C-kielinen ohjelma täytyy kääntää konekielelle ennen kuin sitä voidaan suorittaa tietyssä prosessorissa ja käyttöjärjestelmässä. Kieli on siis **käännettävä ohjelmointikieli** (engl. *compiled programming language*). Kun käännös on fyysisen tietokonelaitteen konekieltä, puhutaan lähdekoodista tehdystä "natiivikäännöksestä", "natiivikoodista" ja "natiivista ohjelmasta" (engl. *native code*). Toinen yleinen esimerkki natiivikäännöksiä tukevasta kielestä on olio-ohjelmointiin tarkoitettu C++, joka on täysin eri kieli kuin C, vaikka se historiansa johdosta sisältää C:n osajoukkonaan ja kykenee tarvittaessa samanlaiseen laiteläheisyyteen.

Jotkut käännettävät kielet, kuten C# ja Java, käännetään muotoon, joka ei ole natiivia koodia vaan kuvitteellisen (ts. näennäisen, virtuaalisen) prosessorin konekieltä, joka on määritelty vastaamaan läheisesti tyypillisten prosessorien toimintaa, mutta on hiukan simppelimpi käyttää ja mm. muistinhallinnan osalta "älykkäämpi" kuin mitä oikea kone pystyisi olemaan. Lisäksi se mahdollistaa käännettyjen ohjelmien suorittamisen uudella laitealustalla, mikäli virtuaalikone saadaan toteutettua kyseiselle alustalle. Ohjelma käynnistetään ja suoritetaan virtuaalisen laitteistorajapinnan toteuttavan ohjelman eli virtuaalikoneen kautta. Virtuaalikone tulkitsee virtuaalista konekieltä käsky käskyltä lennosta ja kääntää sitä natiivikoodiksi "juuri ajoissa" **ajonaikaisella kääntämisellä** (engl. *just-in-time compilation, JIT*). Näissäkin käännös tavukielelle ja kirjastojen linkittäminen tapahtuvat samantyyppisillä mekanismeilla kuin natiiveissa konekielisissä ohjelmissa. Virtuaalikoneen itse on oltava natiivi ohjelma, joka käyttää allaan olevaa laitteistoa ja käyttöjärjestelmää ja välittää sen palveluita sovellukselle, jota sen päällä puolestaan ajetaan. Nämä virtuaalikoneet ovat yksi hyvä esimerkki kerrosmaisesta rakenteesta laitteiston ja sovellusten välillä. Rakenne voi nykyisellään olla hyvinkin syvästi

kerrostunut.

Jotkut kielet ovat vielä tätäkin puhtaammin **tulkattavia kieliä** (engl. *interpreted languages*) siinä mielessä, että niissä ihmisen ymmärtämässä muodossa oleva lähdekoodi tulkitaan rivi kerrallaan lennosta, ilman välttämätöntä tarvetta kääntää lähdekoodia natiiviksi eikä edes minkään virtuaalikoneen tavukoodiksi ennen ohjelman suorittamista. Tunnettuja ja tiettyihin tarkoituksiin suosittuja tulkattavia kieliä ovat esimerkiksi Python, Perl ja PHP – nämäkin tosin voivat hyödyntää konekielimäistä välikieltä toimintanopeuden optimoimiseksi. Tulkattavat kielet mahdollistavat luonnostaan niiden interaktiivisen käyttämisen: käyttäjä voi antaa kielen syntaksin mukaisia komentoja rivi kerrallaan ja nähdä tulokset välittömästi. POSIXin määrittelemä shell syntakseineen ja pakollisine apusovelluksineen on yksi esimerkki tulkattavasta kielestä. Syntaksiltaan se on rankempaan sovellusohjelmointiin suunniteltuja sukulaisiaan jäykempi, ja suorituskyky ei ole kummoinen, mutta se taipuu helposti moniin käyttöjärjestelmän ylös- ja alasajoon, ylläpitoon ja ohjelmien asennusten tarvitsemiin tehtäviin. Ja se on määritelty rajapintastandardissa, joten yhteensopivassa järjestelmässä se on aina asennettuna ja toimii tunnetulla tavalla!

Tässä kohtaa on hyvä tehdä myös lisähuomio sovellusten ja kirjastojen kerroksellisuudesta.. Tulkattavan kielen tulkkiohjelma voisi itsekin toimia tulkin tai virtuaalikoneen päällä, virtuaalikone toisen virtuaalikoneen päällä. Kerroksia voi olla päällekkäin vaikka kuinka monta ihan suorituksen aikanakin. Sovelluksen tekijän tarvitsee olla kiinnostunut vain tarpeeseen valitun alustan rajapinnasta.

Natiivikaan koodi ei voi ”tietää”, toimiiko se virtuaalisessa tai simuloidussa prosessorissa vai ”oikeassa”. Viimeisessä lauseessa ”oikea” on lainausmerkeissä, koska nykyisessä prosessorissa itsessään

on kerrosmainen ja osin rinnakkainen rakenne, jossa todellisuudessa suoritetaan välissä vielä erilaista konekieltä eli ns. **mikrokoodia** (engl. *micro code*). Virtuaalikoneen tavoin myös oikea fyysinen prosessori palastelee ja optimoi sille syötettyä tavukoodia, eikä esimerkiksi käskyjen todellista suoritusjärjestystä voi tarkkaan määrätä. Rajapinta lupaa kuitenkin, että kaikkien ulkomaailmaan näkyvien ilmiöiden suhteen *näyttää* aina siltä, että suoritusjärjestys olisi se, mitä prosessorille on manuaalin ohjeiden mukaan syötetty.

0.4 Konekielisen ohjelman suoritus

Avainsanat: symbolinen konekieli eli assembler, käskysymboli, operandi, lähde ja kohde (käskyn), takaisinkääntäminen (engl. *disassembly*), suorituspino, peräkkäisjärjestys, ehtorakenne, toistorakenne, aliohjelma, poikkeus, kontrollin siirtyminen, ehdoton ja ehdollinen hyppykäsky

Osaamistavoitteet: Luvun luettuaan opiskelija:

- ymmärtää ohjelman suorituksen muistissa sijaitsevien konekielisten käskyjen jonona sekä tyypillisten ohjelmointirakenteiden ilmenemisen konekielessä; tunnistaa konekielisen ohjelman sellaisen nähdessään (AT&T tai Intel -tyyppisenä assemblynä tai disassemblynä) [ydin/arvos2]
- osaa lukea lyhyitä (so. 2-10 käskyä) suomen kielellä rivi riviltä kommentoituja konekielisiä ohjelmanpätkiä ja muodostaa mielessään (tai kynällä ja paperilla) niiden jäljen sekä niiden suorittamisen aiheuttamat tilapäiset ja pysyvät vaikutukset tietokoneen rekistereihin ja muistiin [edist/arvos4]

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 konekielen piirteitä lisää käytännön esimerkkien kautta. Esimerkkiarkkitehtuuri on ns. x86-64 -arkkitehtuuri. Yhtä hyvin esimerkkinä voisi olla mikä tahansa, jolla olisi helppo pyöräytellä esimerkkejä. Valinta tehdään nyt tällä tavoin, koska Jyväskylän yliopiston IT-palveluiden tarjoamat palvelinkoneet, joihin opiskelijat pääsevät helposti käsiksi ja joissa kurssin harjoituksia voidaan tehdä, ovat tällä hetkellä malliltaan

useampiytimisiä Intel Xeon -prosessoreja, joiden arkkitehtuuri on nimenomaan x86-64. Toivottavasti nykyaikaisen prosessorin käsittely on motivoivaa ja tarjoaa teorian lisäksi käytännön kädentaitoja tulevaisuutta varten.

Kurssin ydinaineksen osalta käytännön tekeminen ja ajattelu viedään loppuun demossa 5, jossa olisi tarkoitus ajella ohjelmaa debuggerilla ja katsella sen jälkeä lokitiedostosta.

Esimerkkiarkkitehtuuri: x86-64

Hieman x86-64:n taustaa: Prosessoriteknologiaan keskittyvä yritys nimeltä Intel julkaisi aikoinaan mm. toisiaan seuranneet prosessorimallit (ja arkkitehtuurit) nimeltä 8086, 80186, 80286, 80386, 80486 ja Pentium. Malleissa ominaisuudet laajenivat teknologian kehittyessä, sananleveyskin muuttui 80386:n kohdalla 16 bitistä 32 bittiin, mutta konekielitason yhteensopivuudesta aiempien mallien kanssa pidettiin huolta. Tämän tuote- ja arkkitehtuurijatkumon nimeksi on ymmärrettävistä syistä muodostunut ”x86-sarja”. Muiden muassa toinen tunnettu prosessorivalmistaja, nimeltään AMD, toteutti omia prosessorejaan, joiden ulkoinen rajapinta vastasi Intelin suosittua arkkitehtuuria. Samat ohjelmat ja käyttöjärjestelmät toimivat eri yritysten valmistamissa prosessoreissa samalla tavoin ihan konekielen tasolla.

Sittemmin juuri AMD kehitti sananleveydeltään 64-bittisen arkkitehtuurin, joka jatkaa Intelin x86-jatkumoa nykyaikaisesti mutta edelleen yhteensopivasti vanhojen x86-arkkitehtuurien kanssa. Tällä kertaa Intel on ’kloonannut’ tämän AMD64:ksi nimetyn arkkitehtuurin nimikkeellä Intel64 ja valmistaa prosessoreja, joissa AMD64:lle käännetty konekieli toimii lähes identtisesti. Intel64 ja AMD64 ovat rajapinnaltaan lähes samanlaisia, ja niille on ainakin linux-maailman puolella napattu AMD:n alkuperäisestä dokumen-

taatiosta yhteisnimeksi `x86-64`, joka kuvaa toisaalta periytymistä Intelin x86-sarjasta ja toisaalta leimallista 64-bittisyyttä (eli sitä, että rekistereissä ja muistiosoitteissa on 64 bittiä rivissä). Joitakin eroja Intelin ja AMD:n variaatioissa on, mutta lähinnä niillä on merkitystä yhteensopivien kääntäjien valmistajille sekä erityisten nopeusoptimointien tarvitsijoille. Muita käytössä olevia nimityksiä samalle arkkitehtuurille ovat mm. `x64`, `x86_64`. Käytettäköön tämän monisteen puitteissa jatkossa nimeä 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

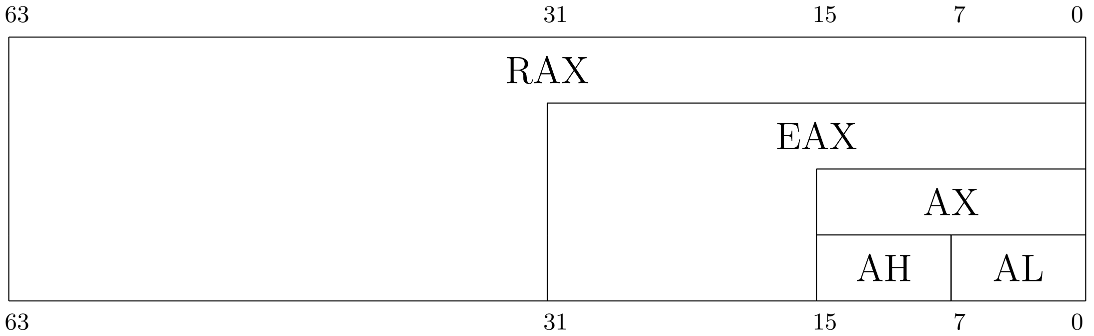
Nyt toivottavasti on riittävästi pohjatietoa, että voidaan vain esimerkinomaisesti listata eräässä prosessorissa käytettävissä olevia rekisterejä merkityksineen niillä lyhyillä nimillä, jotka prosessorivalmistaja on antanut. Taulukossa 0.1 on suurin osa rekistereistä, joita ohjelmoija voi käyttää Intelin 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.

Taulukko 0.1: *x86-64:n 64-bittisen tilan rekisterejä.*

	Toiminnanohjausrekisterit:
`RIP`	Instruction pointer, IP
`RFLAGS`	Liput, 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–R15`	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

Jokaisessa x86-64:n rekisterissä voidaan säilyttää 64 bittiä. Rekistereistä voidaan käyttää joko kokonaisuutta tai 32-bittistä, 16-



Kuva 0.10: *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, 'L'). Jako johtuu x86-sarjan historiallisesta kehityksestä 16-bittisestä 64-bittiseksi ja taaksepäin-yhteensopivuuden säilyttämisestä.*

bittistä tai jompaa kumpaa kahdesta 8-bittisestä osasta. Kuvassa 0.10 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-FPR7`-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 tai virtuaalikoneita [24].

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, koska niihin ei ole pääsyä ohjelmointikeinoin, eivätkä ne siten kuulu julkisesti dokumentoituun (laitteisto)rajapintaan.

[24] 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 kokooptimoinnin lopputulos on hyvä pystyä tarkistamaan ja kokeilla pystyykö itse vaikuttamaan lopputulemaan varioimalla lähdekoodia.

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. Rivi voi myös sisältää dataa, jonka assembler-kääntäjä sisällyttää objektitiedostoon sellaisenaan. Käyttöjärjestelmän ohjelmakoodista pienehkö mutta sitäkin tärkeämpi 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 suoritukseen vaikuttavista 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 mahdolliset syötteet, tulosteet ja sivuvaikutukset prosessorin tai keskusmuistin seuraavaan tilaan ovat. Esimerkin tapauksessa nuo 64 bittiä kopioitaisiin 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, joten se on erittäin nopea toimenpide suorittaa. Siirtokäskyllä ei ole vaikutusta lippurekisteriin. Vaikka käskyn virallinen nimi viittaa siirtämiseen, on toimenpidettä syytä ajatella *kopioimisena*, sillä *lähdepaikka ei siirtokäskyssä tyhjene tai muutu muutenkaan*, vaan *ainoastaan kohdepaikan aiempi sisältö korvautuu lähdepaikassa olleilla biteillä.*

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 maailmasta olisi saatu vaikeampi (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 on ’Intel-syntaksi’. Ylläoleva rivi olisi siinä toisessa syntaksissa jotakuinkin näin:

```
        movq   rbp, rsp
```

Prosenttimerkki puuttuu, mutta merkittävämpi 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’. Jonkun ikivanhan prosessoriarkkitehtuurin konekielen tavujen järjestys on saattanut vaikuttaa myös. 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 kuten kuvassa 0.11 sivulla 98 on esitetty.

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 **takaisinkäännös** (engl. *disassembly*). Tällä tavoin voidaan tutkia ohjelman toimintaa, vaikkei lähdekoodia olisi saatavilla. Työlästähän se on, ja ”viimeinen

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

Assembler-kielinen ilmaus, "mnemonic" ja operandit.

Käsky konekieleksi käännettynä. Näköjään kyseisen käskyn bittijonossa on kolme tavua, jotka heksana ovat 48 89 e5, bitteinä 0100 1000 1000 1001 1110 0101 jos en mokannut päässämuunnosta heksoista. Tarkista itse (= hyvä harjoitus lukujärjestelmistä).

Käskyn suhteellinen muistiosoite ohjelma-alueen alusta luettuna, virtuaaliosoite, heksalukuna.

Kuva 0.11: *Assembler-rivin rakenne, jossa näkyy osoite, konekielinen tavuesitys ja assembler-käsky GNU Assemblerin mukaisena.*

keino” debuggauksessa tai teollisuusvakoilussa, mutta mahdollista kuitenkin. Assembler-kielinen lähdekoodi sinänsä on kokeneen silmään ihan selkeätä, mutta ilman lähdekoodia tehdyssä takaisinkäännöksessä ei ole käytettävissä muuttujien tai muistiosoitteiden kuvaavia nimiä – kaikki on vain suhteellisia numeroindeksejä suhteessa rekisterien kullakin hetkellä sisältämiin muistiosoitteisiin. Kokonaisuutta on silloin aika mahdoton hahmottaa ilman suurta päättelytyötä. Sillä tavoin tietokone käsittelee ohjelman suoritusta eri tavoin kuin ihminen – ihminen tarvitsee käsinkosketeltavia nimiä asioille, kone taas vain numeroita.

”Hello world!” suoraan assemblerilla

C-kielen sekaan voi useimmissa kääntäjissä lykätä kääntäjän omalla syntaksilla jonkin prosessoriarkkitehtuurin mukaista assemble-

ria suoraan rivien väliin (engl. *inline assembly*). Esimerkkejä tästä löytyy kurssimateriaalin kevään 2015 esimerkkihakemistosta viidennen luennon kohdalta, mikäli niihin haluaa palata. Käyttötarkoituksia kielten sekoittamiselle löytyy kuitenkin hyvin harvoin. Kiinnostuneempia olemme standardin mukaisesta C:stä ja puhtaasta esimerkkiarkkitehtuurimme eli x86-64:n assemblerista. Pääasiassa kurssilla katsomme vain takaisinkäännöksiä, mutta niiden ymmärtäminen voi olla helpointa aloittaa katsomalla ensiksi konreettista, kokonaista assemblerilla tehtyä ohjelmaa. ”Hei maailma” soveltuu tähänkin tarkoitukseen.

Seuraavassa on suoraan x86-64:n assemblerilla, GNU-työkalujen ymmärtämällä syntaksilla toteutettu ”Hei maailma”, jonka avulla tartutaan kiinni käyttöjärjestelmän **järjestelmäkutsurajapintaan** (engl. *system call interface*). Se on käyttäjän oikeuksilla käynnistetyn prosessin ainoa tapa saada kutsuttua mitään laitteistoon liittyvää palvelua, joihin ”Hei maailmassa” kuuluu merkkien tulostaminen ja ohjelman oman elinkaaren lopettaminen hyvien tapojen mukaisesti:

```
        .globl  _start
        .text

hei_mun_maailmani:
        .string "Hello,␣hello,␣hello!\n"
hei_mun_maailmani_loppu:

_start:
        movq    $1, %rax
        movq    $1, %rdi
        movq    $hei_mun_maailmani, %rsi
        movq    $hei_mun_maailmani_loppu - hei_mun_maailmani, %rdx
        syscall

        movq $60,%rax
        movq $0,%rdi
        syscall
```

Ensimmäinen rivi julkaisee ohjelmasta yhden nimen `_start`, jonka perusteella käännösvaiheen linkkeri osaa kirjata objektina syntyvään ELF-tiedostoon sen muistiosoitteen, josta ohjelmaa pitää alkaa suorittamaan. Koodissa tälle muistiosoitteelle on annettu tuo kyseinen nimi `_start`, ja tosiaan lopullinen muistiosoite riippuu myös siitä, mihin kohtaan muistia käännösvaiheen linkkeri haluaa ladatuttaa ohjelman koodin aina käynnistyksen yhteydessä. Muihin tarkoituksiin koodissa on pari muutakin symbolista osoitetta. Lopullisilla osoitteilla ei ole lähdekoodin kirjoittajalle väliä, koska työkalut hoitavat ne symbolisilla, selväkielisillä nimillä. Merkkijonon pituus voidaan tässä laskea kahden muistiosoitteen erotuksena, koska erotus ei riipu tavujen absoluuttisesta sijainnista, vaan siitä kuinka paljon kahden osoitteen välissä on tavuja. Lopulliset osoitteet ovat ylipäätään tiedossa vasta lataamisen ja käynnistyksen yhteydessä tapahtuvan linkittämisen jälkeen. Joissain kohdissa osoitteen tilalla voi olla ohjelmatiedostossa nollia, ja ELF-tiedostomuodon määräämässä paikassa on sitten tieto, mitkä nollat tulee latausvaiheessa täydentää milläkin lopullisella osoitteella. Tyypillistä on myös osoitteiden ilmoittaminen suhteessa ohjelmatiedoston tai sen osion alkuun. Tällöin lopullinen osoite on laskettavissa suhteellisen siirrosmäärän ja lopullisen alkuosoitteen summana.

Toinen rivi sanoo assembler-kääntäjälle, että sitä seuraavat tavut pitää sijoittaa ohjelman koodialueelle (nimeltään itseasiassa `text`). Sitten se tuuttaa sinne pötkön merkkejä ja konekielistä koodia, joka syntyy rivi riviltä minimaalisella, prosessoriarkkitehtuurin manuaalin mukaisella muunnoksella. Tässä ohjelmassa kaikki tavut, myös tulostettava merkkijono, ovat koodialueella, mikä on ihan OK. Ohjelman suoritus alkaa `.string`-määreellä lisätyn merkkijonon loppumista ilmaisevaa nollamerkkiä seuraavasta muistiosoitteesta, eivätkä merkkijonon merkit siis häiritse prosessorin toimin-

taa, vaikkei se niistä mitään ymmärtäisikään. Paha ohjelmointivirhe olisi hypätä suorittamaan tavujonoa `Hello, hello, hello!\n\0` ikään kuin se olisi konekieltä. Onneksi kääntäjä ja linkkeri hoitavat ensimmäiseksi suoritettavaksi käskyosoitteeksi symbolin `_start` mukaisen paikan eikä varsinaista koodialueen alkua.

Näinhän siinä tosiaan käy, mikä voidaan verifioida debuggerin tulosteesta (tulostetta kaunisteltu ihan hieman):

```
(gdb) disassemble /r _start
Dump of assembler code for function hei_mun_maailmani_loppu:
   0x00000000004000ea <+0>:     48 c7 c0 01 00 00 00    mov    $0x1,%rax
   0x00000000004000f1 <+7>:     48 c7 c7 01 00 00 00    mov    $0x1,%rdi
   0x00000000004000f8 <+14>:    48 c7 c6 d4 00 40 00    mov    $0x4000d4,%rsi
   0x00000000004000ff <+21>:    48 c7 c2 16 00 00 00    mov    $0x16,%rdx
   0x0000000000400106 <+28>:    0f 05                   syscall
   0x0000000000400108 <+30>:    48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
   0x000000000040010f <+37>:    48 c7 c7 00 00 00 00    mov    $0x0,%rdi
   0x0000000000400116 <+44>:    0f 05                   syscall
End of assembler dump.

(gdb) x/s 0x4000d4
0x4000d4 <hei_mun_maailmani>:   "Hello, hello, hello!\n"

(gdb) x/68xb 0x4000d4
0x4000d4:       0x48    0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x68
0x4000dc:       0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x68    0x65
0x4000e4:       0x6c    0x6c    0x6f    0x21    0x0a    0x00    0x48    0xc7
0x4000ec:       0xc0    0x01    0x00    0x00    0x00    0x48    0xc7    0xc7
0x4000f4:       0x01    0x00    0x00    0x00    0x48    0xc7    0xc6    0xd4
0x4000fc:       0x00    0x40    0x00    0x48    0xc7    0xc2    0x16    0x00
0x400104:       0x00    0x00    0x0f    0x05    0x48    0xc7    0xc0    0x3c
0x40010c:       0x00    0x00    0x00    0x48    0xc7    0xc7    0x00    0x00
0x400114:       0x00    0x00    0x0f    0x05
```

Ohjelma on debuggerissa valmiiksi ladattuna, joten takaisinkäännös pystyy näyttämään ohjelman niissä peräkkäisissä muistiosoitteissa, joihin se on käytännössä ladattu tietokoneen muistiin. Takaisinkäännöksen kolmanteen käskyyn on viimeistään latausvai-

heessa laskettu merkkijonon sijaintipaikka kokonaislukuna[25]. Toinen komento pyytää debuggeria tutkimaan (x, 'examine') muistia kyseisestä muistipaikasta alkaen, tulkiten sisällön merkkijonona (s, 'string'). Kolmas komento pyytää debuggeria tutkimaan muistia 68 tavua eteenpäin samasta paikasta alkaen, tulkiten sitä vain peräkkäin sijaitsevina tavuina (x, 'hex format'; b, 'byte'). Tästä voidaan varsin konkreettisesti huomata, että assembler on tuutannut ulos merkkijonon, merkkijonon päättymistä tarkoittavan nollatavun, ja heti näiden perään suoritettavan koodin rivi kerrallaan käännettynä. ELF-objektiin se on kirjoittanut ylös, että suorituksen tulee alkaa muistiosoitteesta `0x4000ea`.

Ensi töikseen tämä ohjelma pyytää käyttöjärjestelmää tulostamaan merkkijonon käyttäen Linuxin järjestelmäkutsua nimeltä `write`. Mitään nimeähän tässä koodissa ei näy, vaan kutsulle sovittu järjestysnumero 1. Samalla kutsulla tulostettaisiin tiedostoihin ja muihin tietovirtoihin. Standardiulostulolle on sovittu järjestysnumero 1, ja se on sillä numerolla auki jo ohjelman käynnistymisvaiheessa. Siihen kirjoitetut tavut menevät pääteyhteydelle tai ne voi putkittaa toisen ohjelman syötteeksi tai ohjata tiedostoon esimerkiksi shellin kautta. Lisäksi `write()` tarvitsee tiedon kirjoitettavien tavujen sijainnista muistissa sekä lukumäärän tavuista, jotka sen on kirjoitettava. Lukumäärä tarvitaan, koska `write()` kirjoittaa vain tavujonoa, eikä ymmärrä mitään esimerkiksi rivinvaihdoista tai merkkijonon päättymisistä. Kutsu tarvitsee siis neljä parametria, joista yksi on itse kutsun identifioiva numero. Linuxin rajapinnassa on sovittu, että käyttöjärjestelmäkutsun parametrit toimitetaan

[25]Okei, tässä yksinkertaisessa esimerkissä kääntäjä on laskenut osoitteen valmiiksi objektitiedostoon ja virittänyt ELFin otsikot niin, että osoite löytyy aina oikeasta paikasta ilman latauslinkitystä. Lisäksi kierolainen on mennyt omin lupineen optimoimaan 64-bittisiksi kirjoitetut siirtokäskyt 32-bittisiksi. Pittäis katsoa, voiko tuollaisen kieltää suorakäyttökoneidemme työkaluilla ylipäätään. Muilta osin teksti pitänee paikkansa esimerkkiin nähden.

käyttöjärjestelmän käyttöön laittamalla ne tiettyihin rekistereihin juuri ennen varsinaista kutsua (eli `syscall` -käskyn suoritusta). Järjestys, jossa rekisterit tässä täytetään on täysin ohjelman kirjoittajan päättämä; olennaista on, että siirtokäskyjen kohderekisterit ovat Linuxin rajapinnan mukaiset.

Varsinainen magiikka tapahtuu käskyn `syscall` kohdalla – tai ei siinä magiikkaa tapahdu, vaan tietyt tarkoin määritellyt toimenpiteet, joita AMD64-prosessorin manuaali kuvailee noin viiden sivun verran. Tässä kohtaa mm. käyttäjän prosessin suorittaminen keskeytyy ja prosessori hyppää suorittamaan käyttöjärjestelmän koodia. Toiveissa olisi, että osoitettu merkkijono ennemmin tai myöhemmin kopioituisi prosessin omasta muistista jonnekin kohteeseen, vaikkapa käyttäjän päätteelle ja että sen jälkeen Hei maailma -sovellus jatkuisi heti seuraavasta käskystä `syscall`'in jälkeen. Iso osa loppukurssista pyörii mm. sen parissa, mitä muita käyttöjärjestelmäkutsuja kuin `write()` tarvitaan, miten ne tekevät toimenpiteensä, missä vaiheessa keskeytynyt käyttäjän prosessi pääsee jatkumaan ja sen sellaista.

Tulostuksen jälkeen esimerkin ohjelma tekee vielä toisen käyttöjärjestelmäkutsun. Se on nimeltään `exit()` ja sen sovittu järjestysnumero on 60. Kutsun jälkeen Linux tulkitsee sovittuun rekisteriin laitetun lukuarvon olevan prosessin tiedottama virhekoodi ja tekee tarvittavat toimenpiteet prosessin sulkemiseksi ja sen varaamien resurssien vapauttamiseksi. Resurssien luonne ja niiden varaamiseen tarvittavat käyttöjärjestelmäkutsut ovat toinen iso teema loppukurssilla.

Huomautus: Kun alussa on näiden konkreettisten esimerkkien kautta ymmärretty prosessorin toiminta ja käyttäjän prosessin ja käyttöjärjestelmäkoodin yhteispeli, palataan järjestelmäkutsuissa POSIX-standardin mukaisiin C-kielen matalan tason kutsuihin.

Osa niistä on toteutettu lähes sellaisenaan Linuxissa, mutta osa puolestaan tarvitsee väliin vielä yhteensopivuuskirjaston, joka muuntaa standardit palvelukutsut todellisiksi Linuxin käyttöjärjestelmäkutsuiksi ja saattaa tarvita jotakin ylimääräistä muunnostyötä ja kikkailua[26]. Tällainen *yhteensopivuuskerros* on jälleen yksi esimerkki kerrosmaisen rajapintoihin perustuvan suunnittelun hyödyistä. Haetaan tähän kohtaan ajatus siitä, että *POSIXin määräämää C-kielistä käyttöjärjestelmärajapintaa käytetään kutsumalla oletettavasti C-kielellä toteutettua matalahkon tason kirjastoa, joka varsin oletettavasti kutsuu vielä käyttöjärjestelmätoteutuksen tarjoamaa viеläkin matalamman tason yhteensopivuuskirjastoa. Loppujen lopuksi kaikkein matalimman tason apukirjaston täytyy POSIXissa luvatun palvelun hoitamiseksi tehdä yksi tai useampia konkreettisia käyttöjärjestelmäkutsuja, esimerkiksi suorituttaa x86-64 -prosessorilla tämän "Hei maailma" -esimerkin mukaisesti konekielikäsky `syscall`. Ilman käyttöjärjestelmäkutsun tapahtumista käyttäjän ohjelma ei kertakaikkiaan voi lukea eikä kirjoittaa mitään mistään/mihinkään oman rajatun pelikenttänsä ulkopuolella.* Kerroksittaisen kirjaston olemassaolo ja käyttöjärjestelmäkutsun tekninen toteutus meidän tulee tuntea yleissivistyksen vuoksi, mutta korkeammalla abstraktiotasolla määritelty standardi rajapinta vapauttaa miettimästä näitä yksityiskohtia normaaleja sovelluksia kirjoitettaessa.

Esimerkkejä x86-64 -arkkitehtuurin käskykannasta

Edellä nähtiin esimerkki konekielikäskystä, `movq %rsp,%rbp`. Mitä muita käskyjä voi esimerkiksi olla? Otetaan muutama poiminta

[26]Konkretiaa viimeisen päälle haluaville kerrottakoon, että esimerkiksi osoitteessa `https://filippo.io/linux-syscall-table/` näyttäisi löytyvän kaunis käsin koottu lista nimenomaan Linuxin käyttöjärjestelmäkutsuista.

AMD64:n manuaalin [4] käskykantaosiosta, tiivistetään ja suomennetaan tähän. Todellisuus on rikkaampi. Tähän tiivistelmään voi palata demossa 5, jossa tutkitaan aliohjelmia sisältävän ohjelmakoodin toimintaa debuggerilla tuotetun lokitiedoston perusteella.

`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  välill ä
                       # ('l'  == "long word", 32 bittiä)
                       # AMD64 määrittää, että RBX:n 32 ylintä
                       # bittiä asettuvat nollaksi ,  kun tehdään
                       # tällainen 32 bitin  sijoitus.

movq  $123, %rax       # Käskyyn sisällytetyn vakioluvun siirto
                       # rekisteriin  RAX; kaikki 64 bittiä
                       # asettuvat, vaikka luku 123 mahtuisi
                       # käskyssä 8 bittiin.  Lähde tulkitaan
                       # etumerkillisenä.

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 j ärjestyksess ä
                       # 64−bittisen rekisterin  8 tavua
                       # tallentuvat noihin kahdeksaan
                       # muistipaikkaan?
                       #
                       # Tarkista itse  prosessorimanuaalista
                       # kohdasta "byte order", mikäli haluat
```

```
                         # tarkan tiedon ... käytäntö vaihtelee.
                         # x86−sarja on alusta lähtien noudattanut
                         # ns. little −endian −käytäntöä.
                         #
                         # Myöhemmin tutustutaan pinokehysmalliin,
                         # jota noudattaen tuosta osoitteesta, eli
                         # RBP:n arvo miinus kahdeksan, voisi
                         # olettaa löytävänsä vaikkapa ensimmäisen
                         # nykyiselle aliohjelmalle varatun
                         # 64−bittisen lokaalin muuttujan arvon

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` 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.
```

Pinokäskyt

Tyypillinen lähde tai kohde siirtokäskyille on **suorituspinon** (engl. *execution stack*) huippu, jonka muistiosoite on rekisterissä `RSP`. Nimi tälle rekisterille voisi olla **pinon huipun osoitin** tai lyhyemmin **pino-osoitin** (engl. *stack pointer*, `SP`). Pinomaiseen käyttöön liittyy myös huipun siirtäminen ylös tai alaspäin samalla kun pinoon viedään tai sieltä tuodaan tietoa. Huipun käsittelylle ja tiedon siirrolle AMD64:ssä on erityiset käskyt **`push`** ja **`pop`**. Kun pinoon laitetaan jotakin tai sieltä otetaan jotakin pois näillä erityiskäskyillä, prosessori tekee automaattisesti siirron muistiin nimenomaan `RSP`:n osoittamaan kohtaan tai vastaavasti sieltä johonkin rekisteriin. Samalla se päivittää pinon huippua ylös tai alaspäin. (Huomaa, että koko ajan tarkoituksella ohitetaan välimuistiin liittyvät tekniset yksityiskohdat, koska sovellusohjelmoijan ei ole tarkoituskaan nähdä, miten muistijärjestelmä toimii konkreettisessa prosessorissa!). Pari esimerkkiä:

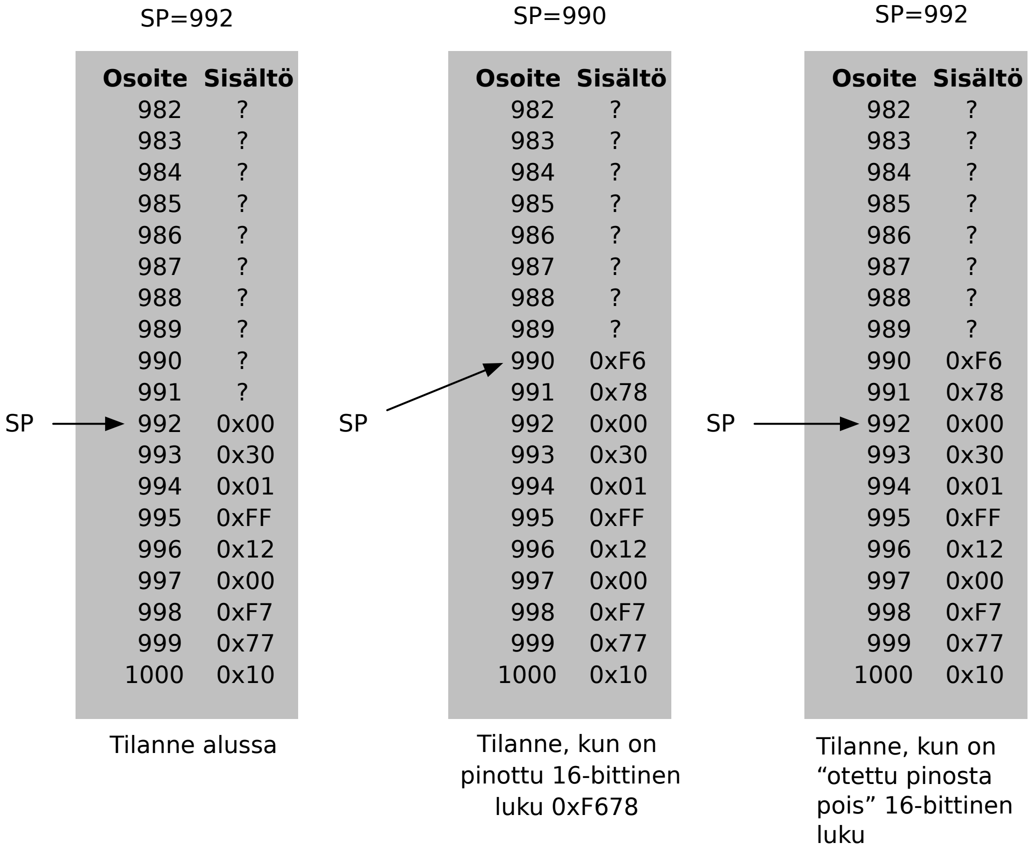
```
pushq $12              # Ensinnä RSP:n arvo pienenee 8:lla,
                       # koska käskyssä on \lstinline:q: mikä tarkoittaa
                       # 64−bittistä siirtoa.  Muistiosoitteethan
                       # ovat aina yhden tavun eli 8 bitin
                       # kokoisten muistipaikkojen osoitteita.
                       #
                       # Siihen kohtaan muistia (osoite uusi RSP)
                       # menee sitten luku 12, eli  64−bittinen
                       # luku joka heksana on 0x000000000000000c.
                       #
                       # HUOM: AMD64, toisin kuin edeltävät x86:t,
                       # tekee kaikki pino−operaatiot 64−bittisinä
                       # vaikka operandi olisi  lyhyempi. Operandi
```

```
                          # tulkitaan etumerkillisenä ja laajentuu
                          # 64-bittiseksi toisintamalla ylintä bittiä
                          # riittävästi. Esim tavu 0x80 painuisi
                          # pinoon muodossa 0xffffffffffffff80.

    pushw %dx             # RSP:n arvo pienenee 2:lla, koska
                          # käskyssä on \lstinline:w: mikä tarkoittaa
                          # 16-bittistä siirtoa. Siihen kohtaan
                          # muistia menee sitten ne 16 bittiä, jotka
                          # ovat rekisterissä DX eli RDX:n 16 alinta
                          # bittiä.

    pushl %edx            # ERROR: Mahdoton toteuttaa AMD64:ssä,
                          # koska arkkitehtuurissa ei ole olemassa
                          # operaatiokoodia tälle. (ks. manuaalin
                          # Volume 3, PUSH-käskyn referenssi).
                          #
                          # 32-bittiselle x86-arkkitehtuurille tämä
                          # kääntyisi, kuten olettaa sopiikin: \lstinline:l:
                          # tarkoittaa 32-bittistä siirtoa, joten
                          # ESP (ei AMD64:n laajennettu RSP) vähenisi
                          # 4:llä ja osoitettuun kohtaan muistia
                          # menisi 32-bittisen rekisterin EDX sisältö.
```

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 mukaisesta
                          # 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 0.12. Huomaa,



Kuva 0.12: *Suorituspino: muistialue, jota voidaan käyttää push- ja pop-käskyillä. SP osoittaa aina pinon ”huippuun”, joka ”kasvaa” muistiosoitteen mielessä alaspäin.*

että pino on käsitteellisesti samanlainen kuin normaali pinotyyppinen (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 muistiavaruudesta: muistiosoitteet kasvavat kuvan ylälaidasta ala-

laitaan päin. Näitä tulemme näkemään. Tässä kuvassa osoitteet ovat tavun kokoisten muistipaikkojen osoitteita, ja ne on ilmoitettu desimaalilukuina. Jatkossa siirrymme (tietokonemaailmassa) järkevämpään tapaan eli heksalukuihin, 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 0.12 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 laskettu (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.

Aritmetiikkaa

Edellä olevat siirto- ja pinokäskyt vain siirtävät bittejä paikasta toiseen. Ohjelmointi edellyttää usein bittien muokkaamista mat-

kalla. Pari esimerkkiä:

```
addq %rdx, -32(%rbp)    # Hakee muistipaikasta (RBP:n arvo − 32)
                        # löytyvät bitit, laskee ne yhteen
                        # rekisteriss ä 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 lis ätään
                        # 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 esimerkiksi seuraavasti:

- 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.

Bittilogiikkaa ja bittien pyörittelyä

Moniin tarkoituksiin tarvitaan bittien muokkaamista. Pari esimerkkiä:

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

andq $15, %rax   # Bitittäinen looginen JA-operaatio. Tässä
                 # tapauksessa 15 on bitteinä 000...001111 eli
                 # neljä alinta  bitti ä 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
                 # kielill ä 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.  Tarkoittaa mm., että
                 # siell ä oleva luku on jaollinen  kuudellatoista.

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. Ei kannata
                    # ihmetellä, jos kääntäjä tekee nollaamisen juuri
                    # tällä tavoin. Se on vain järkevää.
```

Muitakin bittioperaatioita on. Joitain esimerkkejä:

```
    sarb $3, %ah    # Siirtää 8-bittisen (\lstinline :b:, byte) rekisteriosan
                    # bittejä kolmella pykälällä oikealle , eli jos
                    # siellä oli bitit 0110 0101 niin sinne jää
                    # käskyn jälkeen 0000 1100. Vastaa kokonaisluvun
                    # jakamista kakkosen potenssilla 2^3 eli 8:lla.

    rolw %cl, %ax   # Pyörittää 16-bittisen (\lstinline :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 (muistibitti, muistinumerolippu) 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.

Suoritusjärjestyksen eli kontrollin ohjaus: mistä on kyse

Tähän asti mainituista käskyistä muodostuva ohjelma suoritetaan **peräkkäisjärjestyksessä**, eli prosessori siirtyy käskystä aina vä-

littömästi 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.

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, C#:lla 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 assemblerkääntäjä 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 ... */
}
```

Yhteenveto: Jokaisella eri prosessorilla on oman käskykantansa mukainen assembler, ja yksi assembler-kielinen rivi kääntyy yhdeksi konekieliseksi käskyksi.

Assembler-käännös toimii toiseenkin suuntaan: Takaisinkääntäjän (engl. *disassembler*) avulla on mahdollista tutkia ohjelman toimintaa, vaikka lähdekoodia ei olisikaan saatavilla. Tällöin käytössä on tosin vain suhteellisia numeroindeksejä, joiden kanssa kone operoi.

Pino on muistialue, jota voidaan käyttää muiden käskyjen lisäksi **`push`**- ja pop-käskyillä. Kun pinoon laitetaan luku (**`push`**), rekisterin `RSP` sisältämä muistiosoite pienenee, koska pinon huippu 'kasvaa alaspäin', kun tarkastellaan muistiosoitteita. Pinon huipulta otettaessa (**`pop`**), `RSP` suurenee.

Peräkkäisjärjestyksessä suoritettavien käskyjen lisäksi ohjelmoinnissa vaaditaan myös ehto- ja toistorakenteita, aliohjelmia sekä poikkeuksia ja niiden käsittelyä. Laitteiston kontrollin siirtäminen ohjelman osiolta toiselle on keskeistä käyttökelpoisessa ohjelmoinnissa. Käskyosoitin `RIP` muuttuu jokaisen käskyn kohdalla. Rakenteet hoidetaan hypyillä, joissa `RIP`:n uudeksi arvoksi ladataan (asetetaan) jotakin muuta kuin seuraavan peräkkäisen käskyn osoite. Hypyt voidaan suorittaa ehdollisesti riippuen `RFLAGS`in bittien tilasta. Kokonaislukujen aritmeettiset ja binääriloogiset käskyt vaikuttavat `RFLAGS`in bitteihin.

0.5 Ohjelma ja tietokoneen muisti

Avainsanat: koodi, data, paikallinen eli lokaali muuttuja, pinomuisti, dynaaminen muistinvaraus, kekomuisti, viite, virtuaaliosoite, segmentoitu muisti, segmenttirekisteri, aliohjelma eli funktio eli proseduuri, metodi, objekti eli olio, pinokehys eli aktivaatiotietue, kantaosoitin, kutsumalli

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa selittää prosessin virtuaalimuistiavaruuden tyypillisen jaon segmentteihin (koodi, data, pino, keko, käyttöjärjestelmäosio) sekä antaa esimerkin segmenttien sijoittumisesta avaruuden osoitteisiin [ydin/arvos2]
- ymmärtää suorituspinoon päällekkäin allokoitavien aktivaatiotietueiden käytön aliohjelmien parametrien ja paikallisten muuttujien säilytyksessä; osaa soveltaa tätä tietoa tekstipohjaista debuggeria käyttäen ja kykenee näkemään yhteyden aiemmin opitun oliokielen (esim. C#) viitteiden ja rakenteisen, laiteläheisen kielen (esim. C) muistiosoittimina toteutuvien viitteiden välillä [edist/arvos4]

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

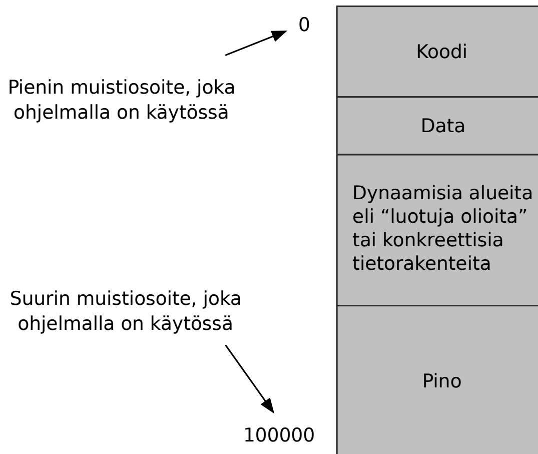
Tyypilliset segmentit eli osiot prosessin muistiavaruudessa

Edellä on jo kerrottu, että kääntäjän ja käännösvaiheen linkkerin tuottama objektitiedosto, mm. lopullinen suoritettava ohjelmatiedosto, pitää sisällään tavujonoja, jotka on objektitiedoston formaa-

tissa merkattu ladattavaksi tiettyihin muistipaikkoihin. Käyttöjärjestelmän lataaja osaa tulkita objektitiedoston, varata ohjelmalle riittävästi muistia ja ladata pätkät tiedostosta pyydettyihin osoitteisiin. Käydään seuraavaksi läpi minimaalisin joukko osioita eli 'muistisegmenttejä', joita melkeinpä kaikki ohjelmat tarvitsevat.

Koodi, tieto ja suorituspino; osoittimen käsite

Ohjelmaan kuuluu ilmeisestikin konekielikäskyjä prosessorin suoritettavaksi. Sanotaan, että tämä on ohjelman **koodi** (engl. *code*), jota joskus sanotaan myös 'tekstiksi' (engl. *text*). Lisäksi ohjelmissa on usein jotakin ennakkoon tunnettua tai globaalia **dataa** (engl. *data*) kuten merkkijonoja ja lukuarvoja. Varmasti tarvitaan vielä **paikallisia muuttujia** (engl. *local variables*) eli tallennustilaa useisiin väliaikaisiin tarkoituksiin. Tämä lienee selvää.



Kuva 0.13: *Tyypilliset ohjelman suorituksessa tarvittavat muistialueet: koodi, data, pino, dynaamiset alueet. (periaatekuva, joka täydentyy myöhemmin)*

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 alustakirjaston mustinhallintaosiolta (ja viimekädessä käyttöjärjestelmältä) käyttöönsä alueita, joiden kokoa ei tarvitse tietää ennen kuin pyyntö tehdään. Kuvassa 0.13 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 kokonaisluku; useimmiten osoite otetaan jonkun rekisterin arvosta (eli rekisterin kautta tapahtuva epäsuora osoitus engl. *register indirect addressing*). Esim. pino-osoitinrekisteri `SP` ja käskyosoiterekisteri `IP` sisältävät aina muistiosoitteen. Osoite voidaan myös laskea suhteellisena rekisterissä olevaan osoitteeseen nähden (tapahtuu rekisterin ja käskyyn koodatun vakion yhteenlasku ennen kuin osoite on lopullinen, ns. epäsuora osoitus **kantarekisterin** 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).

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. */
```

C#- ja Java-ohjelmassa jokainen viitemuuttuja on tavallaan 'muistiosoite' olioinstanssiin **kekomuistissa** (engl. *heap*). Tai vähintäänkin sitä voidaan abstraktisti ajatella sellaisena.[27]

Nykyään tiedekunnassamme aloitetaan ohjelmointi C#:lla, joten seuraava Java-esimerkki voi olla hieman vaikea hahmottaa. Operaattori `==` toimii Javassa samalla tavoin kuin C#:sta löytyvä metodi `Object.ReferenceEquals(A,B)`. Metodi `A.equals(B)` puolestaan toimii Javassa samalla tavoin kuin operaattori `==` C#:ssa. Jos seuraava esimerkki meinaa sotkea päätä enemmän kuin selkeyttää, voit ohittaa sen ja yrittää selvittää muistiosoitteen käsitteen C- ja assemblerkielisten demojen 4 ja 5 kautta.

Kaikesta huolimatta otetaan tässä kohtaa oliokielen viitteitä käsittelevä esimerkki Javalla:

```
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 voisi
                              olettaa Java−sovelluksessa.. */
```

Ylläoleva Java-esimerkki pitäisi olla erittäin hyvin selkärangassasi, jos voit sanoa osaavasi ohjelmoida Javalla! Ja jos ei se vielä ole, voit ymmärtää asian yhtä aikaa kun ymmärrät muistiosoitteetkin (ja tulla siten askeleen lähemmäksi ohjelmointitaitoa): Esimerkissä

[27] Joissakin oliokielessä toki viite osoittaa jonkinlaiseen taulukkoon, josta selviää kaikki luokan perintähierarkian mukaiset rajapinnat. C-kielessä tällaisia ei ole valmiiksi toteutettuna, vaan muistiosoite on siinä mielessä paljon suoraviivaisempi kuin aidon oliokielen viite.

`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 eli kopioidaan *viite* yhdestä muuttujasta toiseen. Sen jälkeen *viitemuuttujat* `muuttujaA` ja `muuttujaC` ovat edelleen selvästi erillisiä – nehän ovat Java-virtuaalikoneen suorituspinossa eri kohdissa ja niille on oma tila sieltä varattuna. Mutta se *olioinstanssi, johon ne viittaavat on yksi ja sama konkreettinen yksilö*. 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 Java-virtuaalikoneen pinossa voisi olla seuraava sisältö:

```
muuttujaA : 57686

muuttujaB : 3422

muuttujaC : 57686
```

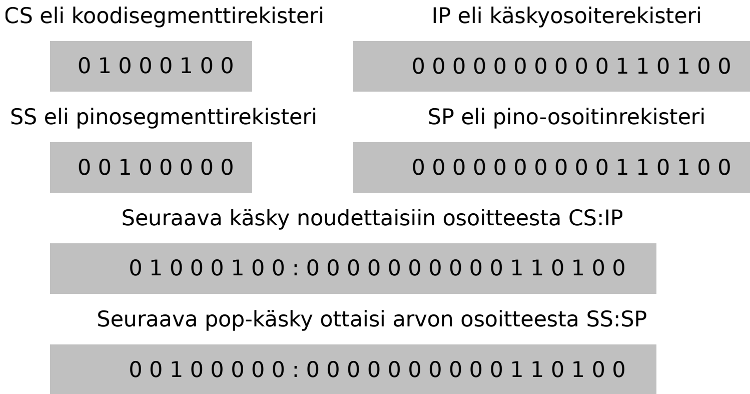
Niinpä esim. muuttujien vertailut operaattorilla ja metodilla antavat tulokset siten kuin yllä on kommenteissa. Yritän siis kertoa vielä kerran, että:

- 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' tai 'deskriptoriksi'. 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).

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



Kuva 0.14: *Esimerkki segmenttirekisterien käytöstä (esim. 80286-prosessori): 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.*

useita ohjelmia, jotka eivät saisi sotkea toistensa toimintaa). Nykyisissä prosessoreissa tällaisia tuplavarmistuksia on saatavilla.

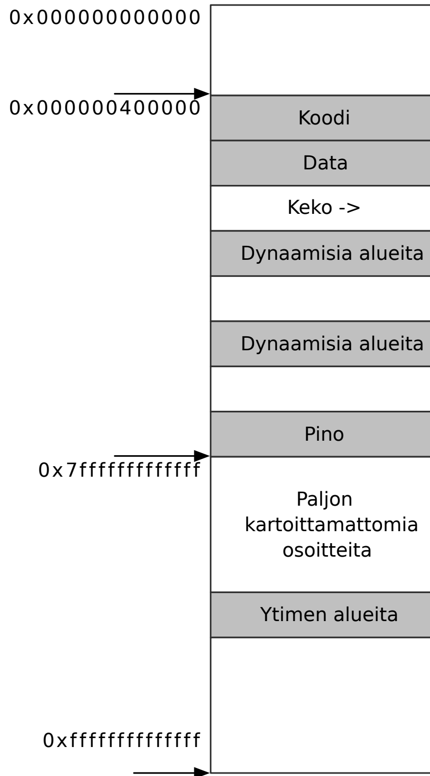
Moderneissa tietokoneissa sovellusohjelman konekielikäskyjen käsittelemät muistiosoitteet ovat ns. **virtuaaliosoitteita**: suorituksessa oleva ohjelma näkee oman koodinsa, datansa ja pinonsa omassa muistiavaruudessaan kuten kuvassa 0.13 summittaisesti esitettiin. Oikeat muistiosoitteet tietokoneen fyysisessä 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 laitteistotasolla, jolloin 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 pi-

no 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 0.14. 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. Segmenttirekisterit ovat edelleen olemassa taaksepäin yhteensopivassa prosessorissa, mutta manuaali määrää, että 64-bittisessä toimintatilassa 'muinoisten' segmenttirekisterien sisältönä pitää olla 0.

Esimerkkiarkkitehtuurimme virtuaaliset osoitteet ovat siis $0, 1, \ldots, 28$ heksalukuina kenties selkeämmin: 0x0, 0x1, . . . , 0xffffffffffff. Fyysistä muistia ei voi olla näin paljoa (281 teratavua), joten virtuaalimuistiavaruudesta kartoittuu fyysiseen muistiin vain osa. Muut muistiosoitukset johtavat virhetilanteeseen ja ohjelman kaatumiseen. Koodi, pino ja tieto sijoittuvat kukin omaan yhtenäiseen alueeseensa 48-bittisessä virtuaaliosoiteavaruudessa.

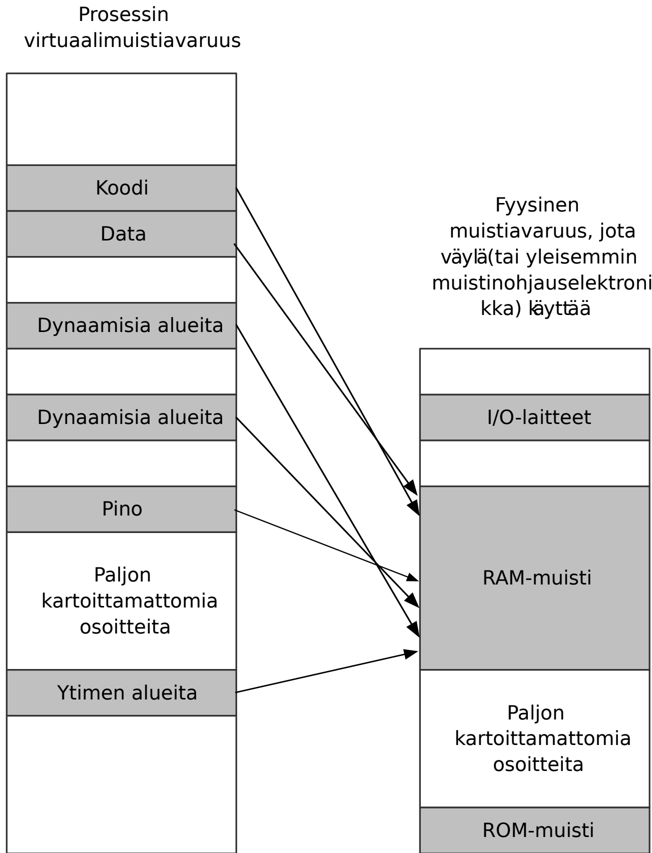


Kuva 0.15: *Virtuaalinen muistiavaruus x86-64:ssä tietyn ABI-sopimuksen mukaan. (XXX: Kuvassa väärät heksat, pitäisi olla 48-bit osoitteet, ei 56-bit.)*

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 0.15. 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 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 paikas-

ta, 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, -0x7fffffffffff, . . . , -0x1. 'Negatiiviseen' virtuaalimuistiavaruuden puolikkaaseen voidaan sopia liitettäväksi käyttöjärjestelmäytimen tarvitsemia alueita, jolloin osoitteen eniten merkitsevästä bitistä voidaan helposti päätellä, kuuluuko muistiosoite sovellusohjelman vai käyttöjärjestelmän käyttöön.

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 0.10. Kuvassa 0.16 havainnollistetaan seikkaa. Prosessin näkemät muistiosoitteet ovat siististi järjestyksessä, vaikka tiedot voivat sijaita sikin sokin todellisessa muistissa. Prosessorilaite hoitaa osoitteiden muunnoksen käyttöjärjestelmän ohjaamana, joten käyttäjän ohjelman tarvitsee huolehtia vain omista virtuaaliosoitteistaan.



Kuva 0.16: *Geneerinen esimerkki ohjelman näkemän virtuaaliosoiteavaruuden ja tietokonelaitteiston käsittelemän fyysisen osoiteavaruuden välisestä yhteydestä.*

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.

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 (puhtaassa sellaisessa) funktio on pääroolissa: ohjelmalla ei ole lähtökohtaisesti muuttuvaa tilaa, vaan se on kuvaus syötteestä tulosteeksi. Toki tila voidaan sitten mallintaa ja suoritusjärjestystä emuloida ns. monadeilla. Tästä on erillinen kurssi Funktio-ohjelmointi, jossa ihminen kuulemma valaistuu lopullisesti.

- imperatiivisessa / rakenteisessa 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 tehdä itselleen jokin toimenpide 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 (sivuvaikutus), 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**.

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 alustuskoodi 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 nä-

kökulmasta[28]. Minkä tahansa ohjelman suoritusta voidaan ajatella sarjana seuraavista:

- peräkkäisiä käskysuorituksia
- ehdollisia ja ehdottomia hyppyjä `IP`:n arvosta toiseen
- aliohjelma-aktivaatioita, joista muodostuu kutsupino.
- (nykyaikaisissa kielissä poikkeustilanteiden käsittelyjä, joissa voi tapahtua palaaminen poikkeuskäsittelijään usean pinossa olleen aktivaation yli)

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

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"
```

[28]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

```
                          # 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 ja siitä palaamisen 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:

```
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[29]) 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 siit ä l öytyy RET−käskyn
                          # edellyttämä paluuosoite.
```

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

```
movq %rbp, %rsp           # RBP oli se aiempi RSP ... tilanteessa  jossa
                          # oli  juuri  pinottu edelt ävä 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 **ENTER**in ja **LEAVE**in välisessä koodissa **SP** on aina vapaana uuden kehyksen tekemiselle eli seuraavan sisäkkäisen

[29]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.

aliohjelmakutsun tekemiselle. Tästä tosiaan lisää myöhemmin.

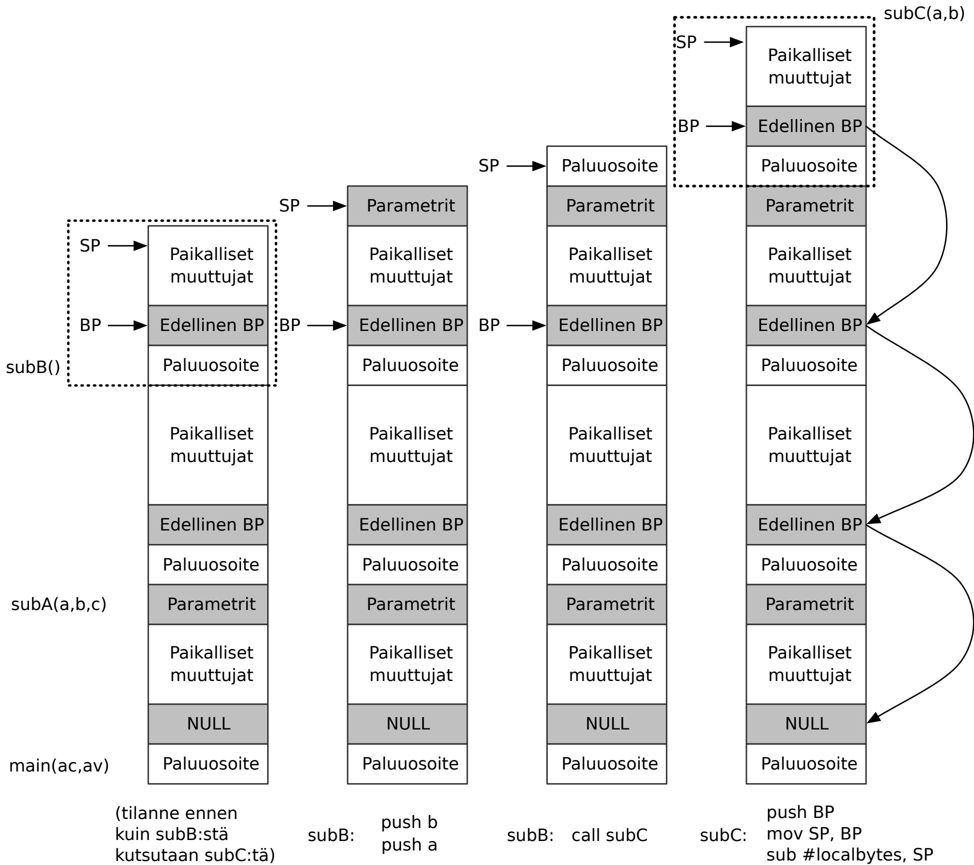
”Ohjelman suorituksen ymmärtäminen konekielitasolla” on tämän kurssin yksi osaamistavoite. Ohjelman suoritus C#:n tai Javan virtuaalikoneen konekielitasolla on samankaltaista kuin ohjelman suoritus x86-64:n konekielitasolla tai kännykässä olevan ARM-prosessorin konekielitasolla, joten tarkoitus on oppia yleisiä ja laitteistosta riippumattomia periaatteita. Kuitenkin teemme tämän nyt valitun esimerkkiarkkitehtuurin avulla.

Ohjelmaa suoritettaessa ollaan käynnistymisen jälkeen suorittamassa aina jotakin aliohjelmaa, alkaahan ohjelma yleensä `main`-aliohjelmasta tai metodista. 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 tai vaikuttaa olioihin sivuvaikutuksen kautta
- sen pitää pystyä kutsumaan muita aliohjelmia.

Aliohjelmat suoritetaan normaalisti käyttämällä kaikkeen ylläolevaan suorituspinoa (se kuvassa 0.12 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*)[30].

[30]HUOM: Jos ollaan aivan tarkkoja, niin aktivaatiotietue on laajempi käsite kuin pinokehys. Aktivaatiotietueeseen ajatellaan sisältyväksi kutsuvassa aliohjelmassa pi-



Kuva 0.17: *Perinteinen pinokehys, ja kuinka se luodaan: eri vaiheet, osallistuvat ohjelman osat sekä "pseudo-assembler-koodi".*

Rakenteen käyttöön tarvitaan virtuaalimuistiavaruudesta pinoalue ja prosessorista kaksi rekisteriä, jotka osoittavat pinoalueelle. Toinen on pinon huipun osoitin (`SP`), ja toinen pinokehyksen **kantaosoitin** (joskus `BP`, engl. *base pointer*).

Perinteistä ideaa havainnollistetaan kuvassa 0.17. Idea on seuraavanlainen. Pinon huipulla (pienimmästä muistiosoitteesta vähän matkaa eteenpäin) sijaitsee tällä hetkellä suorituksessa olevan alioh-

notut ja rekistereissä siirrettävät parametrit, mutta pinokehykseen vain se puoli aktivaatiotietueesta, joka on kutsutun aliohjelman varaamaa tilaa. Tässä sarjakuvaan on merkitty laatikoituna pinokehys, ei tarkkaan ottaen aktivaatiotietuetta.

jelman pinokehys, jolle pätee seuraavaa, kun rekisterit `BP` ja `SP` on asetettu oikein:

- 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:

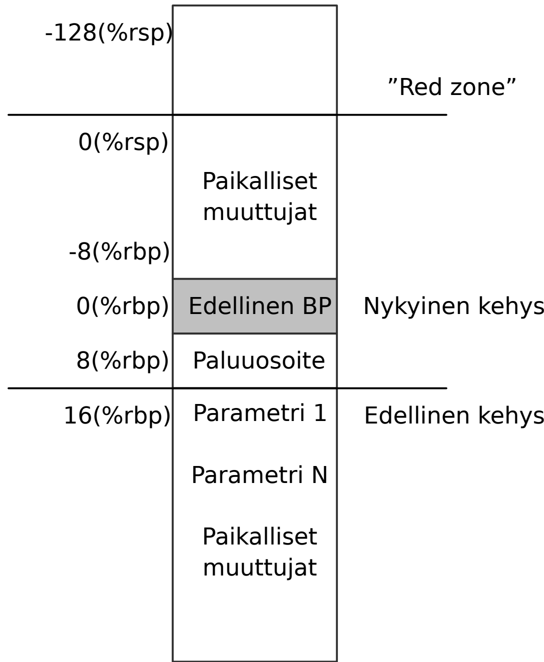
- Aliohjelman lopussa voidaan löytää edeltävän aktivaation pinon huippu kohdasta, johon **`BP`** viittaa. Palautetaan siis BP:n sisältö SP:hen.
- Pinoon oli aliohjelman alussa laitettu edellisen aktivaation kantaosoitin. Nyt se voidaan poksauttaa pinon päältä takaisin BP:hen.
- Näin pinon päällimmäiseksi jäi paluuosoite, jonka **`CALL`** -käsky sinne laittoi. Voidaan suorittaa **`RET`** joka poimii `IP`:n seuraavan arvon pinon 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, jossa jälleen toistetaan kutsuun liittyvät vaiheet eri parametreilla, minkä jälkeen hypätään johonkin aliohjelmaan).

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** tai **kutsusopimus** (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 ja osaa 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 valmiiksi konekielistä kirjastoa, jonka lähdekoodia ei ole saatavilla?

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"[31].

[31] **System V** on 1980-luvulla tehty versio Unixista. Sitä voidaan pitää eräänlaisena standardina myöhempien Unix-variaatioiden tekemiselle, erityisesti sen versiota 4.0,



Kuva 0.18: *Pinon käyttö x86-64:ssä kuten SVR4 AMD64 supplement sen määrittelee.*

Olen tiivistänyt tähän olennaisen kohdan em. dokumentin draftista, joka on päivätty 3.9.2010.

Mainitun ABIn mukainen pinokehys ilmenee siten kuin kuvassa 0.18. Hyvin pitkälti 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
```

jota sanotaan SVR4:ksi.

```
R8  == viides integer-parametri
R9  == kuudes integer-parametri
```

Jos välitettävänä on enemmän kokonaislukuja, ne menevät seitsemännestä alkaen pinon kautta. Rakenteet palastellaan 64-bittisiin osiin, jotka välitetään rekistereissä, mikäli niihin mahtuu. Jos välitettävänä on rakenteita, joissa on tavuja enemmän kuin rekistereihin mahtuu, sellaiset laitetaan pinoon – ja on siellä ABIssa 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.

Mikäli aliohjelma ei tule missään olosuhteissa kutsumaan muita aliohjelmia, sillä on lupa käyttää kuvaan merkittyä 128 tavun kokoista 'punaista aluetta' omiin tarkoituksiinsa, esim. väliaikaisille muuttujille. SysV AMD64 ABIssa on nimittäin sovittu, että mikään muu taho ei käytä tuota aluetta mihinkään tarkoituksiin (edes käyttöjärjestelmän palveluiden avulla, mikä olisi muutenkin ainoa sallittu mahdollisuus käyttää prosessin pinoa muiden kuin itse prosessin toimesta). Tämän sopimuksen perusteella voi tehostaa lyhykäistä aliohjelmaa sen verran, että siitä voi sallitusti jättää pois useiden toimenpiteiden mittaiset **`ENTER`** ja **`LEAVE`** -osiot. Rajoitteina punaisen alueen käytössä tosiaan on, että aliohjelma tarvitsee korkeintaan 128 tavua muistia paikalliseen laskemiseen eikä itse kutsu mitään muuta aliohjelmaa. Kääntäjäohjelma voi todeta tämän lähdekoodista varsin helposti ja suorittaa optimoinnin automaattisesti tulostamaansa konekieleen.

Näillä eväillä pitäisi pystyä tekemään kurssin perinteinen harjoitustyö (käytännössä kurssin 5. pakollinen demo), jossa käväistään hiukan syvemmällä konekielen toiminnassa. Kurssin tehtävien muoto

on laadittu sellaiseksi, että eksoottisempia säännöstöjä ei tarvitsisi käyttää kuin mitä tässä kuvailtiin – parametreina olisi joko 64-bittisiä kokonaislukuja tai muistiosoitteita, jolloin em. kuvaus, jopa ilman punaisen alueen käyttöä, on riittävä. Punaisen alueen käyttö estetään demon esimerkkialiohjelmassa tekemällä sieltä ylimääräinen aliohjelmakutsu, joka itsessään ei tee mitään muuta kuin palaa takaisin.

Yhteenveto: Ohjelman suorituksessa tarvitaan tyypillisesti seuraavia muistialueita: koodialue, ennakkoon tunnettu data, pinomuisti ja dynaamiset alueet, joiden kokoa ei ole välttämätöntä tietää etukäteen. Dynaamisten osalta puhutaan **keosta**. Dynaamisesti varattuihin alueisiin pääsee käsiksi muistiosoitteiden kautta, jotka alustakirjaston muistinvarausaliohjelma palauttaa löydettyään riittävästi tilaa. Alustakirjasto voi tarvittaessa pyytää käyttöjärjestelmän muistinhallintaosiolta lisää muistipaikkoja keossa käytettäväksi.

Prosessorin kannalta käsiteltävä data on rekistereissä tai se pitää noutaa ja viedä muistiosoitteen perusteella. Operaatioiden tulokset, joita tarvitaan pidemmän ajan päästä, on vietävä muistiin osoitteen perusteella.

Sovellusohjelman konekielikäskyjen käsittelemät muistiosoitteet ovat virtuaaliosoitteita, eli suorituksessa oleva ohjelma näkee oman koodinsa, datansa ja pinonsa omassa muistiavaruudessaan, joka lineaarisessa osoitusmallissa alkaa nollasta ja päättyy osoitteeseen, jossa kaikki muistiosoittimen käyttämät bitit (esim. 48 kpl) ovat ykkösiä. Virtuaalimuistiavaruudesta kartoittuu fyysiseen muistiin vain osa. Esimerkiksi ylin puolisko muistiavaruudesta voidaan kartoittaa käyttöjärjestelmän käyttöön, johon sovellusohjelma ei voi kajota. Pinomuisti kartoitetaan tyypillisesti sovellukselle sallitun alueen loppuun ja pinon täyttö alkaa viimeisestä sovellukselle sal-

litusta osoitteesta. Koodi ja data sijaitsevat pienehköissä osoitteissa. Dynaamisesti varattava kekomuisti sijaitsee keskialueella osoiteavaruutta.

Aliohjelman täytyy voida saada kutsujalta parametreja, välittää paluuarvo takaisin sitä kutsuneelle ohjelmalle sekä käyttää paikallisia muuttujia laskentaan. Lisäksi sen tulee voida kutsua muita aliohjelmia.

Aliohjelmien kutsuminen voidaan hoitaa (esimerkiksi erään x86-64:lle sovitun ABIn mukaisesti) luomalla pinon huipulle uusi pinokehys eli aktivaatiotietue:

1. *Kutsuvan aliohjelman* käskyt laittavat parametrit pinoon (käänteisessä järjestyksessä lähdekoodiin nähden), jos ne eivät mahdu rekistereihin nopeampaa välitystä varten.

2. Aliohjelmaan vievä käsky (`CALL`) myös tallentaa seuraavan käskyn osoitteen pinon huipulle. Sen jälkeen `IP`:hen sijoittuu aliohjelman ensimmäisen käskyn osoite.

3. Suoritus siirtyy aliohjelmaan luonnostaan, kun prosessori noutaa `IP`-rekisterin perusteella seuraavan käskynsä.

4. *Juuri kutsutun aliohjelman* ensimmäinen käsky painaa nykyisen `BP`:n eli kantaosoittimen pinoon. `BP`-rekisteri siirtyy uuden aktivaation käyttöön, kun siihen siirretään nykyinen `SP` (eli pinon huippuosoite).

5. Pino-osoittimesta voidaan vähentää aliohjelmassa tarvittavien paikallisten muuttujien tarvitsema tila, jonka jälkeen `SP` vapautuu käytettäväksi seuraavassa aliohjelmakutsussa.

Aliohjelmasta palaaminen tapahtuu karkeasti ottaen tekemällä kumoavat operaatiot käänteisessä järjestyksessä.

0.6 Käyttöjärjestelmä

Avainsanat: peräkkäiset ohjelmasuoritukset eli työt, yhdenaikaisuus, vuoronnus, aikataulutus, eräajo, moniajo, aikajakojärjestelmä, prosessi, tavoiteasettelu ja metriikat, kompromissit

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa kuvailla tyypillisen nykyaikaisen käyttöjärjestelmän päätehtävät ja kykenee pääpiirteissään selittämään historian, jonka kautta käyttöjärjestelmien nykymuoto on syntynyt [ydin-/arvos1]
- tunnistaa ja osaa antaa esimerkkejä ristiriitaisista tavoitteista ja kompromisseista, joita käyttöjärjestelmän toteutukseen ja asetusten tekemiseen välttämättä liittyy [ydin/arvos1]

Käyttöjärjestelmien historiaa ja tulevaisuutta

”Käyttöjärjestelmien historia ja tulevaisuus” liittyy ilman muuta tämän kurssin osaamistavoitteisiin. Historia voi tietysti kuulostaa kuivalta ja tarpeettomalta, mutta toisaalta vain sitä ymmärtämällä voidaan saada tarpeellinen kuva alati jatkuvasta kehityskaaresta, jonka osana toimimme vuonna 2016. On myös niin, että jokainen nykyisten käyttöjärjestelmien piirre on syntynyt johonkin tarpeeseen, joka jossain vaiheessa on ilmennyt, eivätkä aiemmat tarpeet yleensä poistu vaan niitä tulee aina vain lisää. Uudemmat järjestelmät lainaavat aiemmista niissä hyväksi havaittuja menettelyjä, ja vastaavasti huonoksi havaittuja täytyy pyrkiä analysoimaan ja kehittämään. Historia täytyy siis tuntea, että voi tehdä uutta.

Tietokonelaitteistoa ohjaamaan tarkoitettujen käyttöjärjestelmäohjelmistojen historia liittyy tietenkin elimellisesti itse laitteistojen historiaan. Seuraavassa tiivistetään suomen kielellä Stallingsin oppikirjan [1] historiakatsaus. Joitakin näkökulmia on lisätty Arpaci-Dusseau -pariskunnan kirjan [2] ja satunnaisten (alkuperäisten) Internet-lähteiden perusteella.

1940-luku, 1950-luku: Maailman ensimmäiset nykyisenkaltaiset tietokoneet rakennettiin. Niissä ei ollut erillistä käyttöjärjestelmää: Aluksi ohjelmat, joilla ratkottiin esim. matemaattisia yhtälöitä, käännettiin ja koostettiin (”assembloitiin”) osin käsipelillä, ladattiin suoraan tietokoneen muistiin reikäkorttipinkasta ja käynnistettiin. Sitten odotettiin, kun ohjelma laski ja tulosti esimerkiksi paperinauhalle tai binääriselle valotaululle (lamppu päällä tai pois). Ohjelman päättymisen jälkeen pysähtyi myös itse tietokone.

Konekieltä mielekkäämmät ohjelmointikielet nähtiin jo alkuvaiheessa tarpeellisiksi, ja syntyivät mm. kuuluisat LISP ja FORTRAN -kielet. Ohjelman automaattisesta kääntämisestä ihmisen ymmärtämästä lähdekoodista konekielen bittisarjaksi tuli osa tietotekniikan arkipäivää. Samoin havaittiin, että usein tarvittavista yleiskäyttöisistä ohjelman osista oli järkevää tehdä kirjastoja, joita saattoi käyttää samanlaisina eri sovelluksissa. Yksi tulkinta on, että kunkin tutkimuskeskuksen kirjasto-ohjelmisto muodosti silloisen ”esikäyttöjärjestelmän” paikallisen tietokoneen käytön helpottamiseksi, vaikka nimi ”käyttöjärjestelmä” vakiintui vasta myöhemmin.

Tämä kaikki oli uutta ja mullistavaa, mutta kehityskohteitahan tietokoneiden käytössä oli jo heti alkuun havaittavissa:

- Miljoonien arvoista laitetta varattiin paperikalenterista kuin Kortepohjan pyykkikonetta muinoin, esim. tunniksi kerral-

laan. Tietokoneen käyttö oli siis luonteeltaan vahvasti **peräkkäistä** (engl. *serial processing*). Jos tunnin aikaikkunassa tehtiin vartin työ, jäi 45 minuuttia hukattua aikaa. Ongelmana oli siis mm. kallis hukka-aika, joka johti jonkinlaisen automaattisen **vuoronnuksen** tai **aikataulutuksen** (engl. *scheduling*) tarpeeseen.

- Jokainen ohjelma eli **työ** (engl. *job*) täytyi erikseen laittaa toimintakuntoon, eli koneeseen piti ensin ladata kääntäjäohjelma ja sen jälkeen syöttää kirjastot ja sovellus kääntäjälle, ennen kuin päästiin aloittamaan itse ohjelman suorittaminen. Ohjelmointivirheen tai laitevian takia homman saattoi joutua aloittamaan alusta, mikä tarkoitti turhautumisen lisäksi taas lisää hukka-aikaa ja ongelmia aikataulujen kanssa – työ kun piti saada myös loppumaan ennen seuraavana vuorossa olevan kollegan aikaikkunaa.

1950-luku: Ensimmäisten haasteiden ratkaisuna käyttöön olivat tulleet ”monitoriohjelmat” eli ensimmäiset ”proto-käyttöjärjestelm
Monitorista osa oli pysyvästi tietokoneen muistissa (toki sekin piti laitteen käynnistämisen jälkeen sinne ladata). Monitori latasi sitten muistiin aina seuraavan työn eli ohjelman, joka ajettiin alusta loppuun, minkä jälkeen järjestelmän kontrolli tuli siirtää takaisin monitorille, jotta se pystyisi automaattisesti lataamaan muistiin seuraavan työn ja käynnistämään sen. Ohjelmat sijoitettiin peräkkäin ajettavaksi ”eräksi”, esim. reikäkorttipakkaan, ja puhuttiin **eräajosta** (engl. *batch processing*), jossa usean työn erä saatiin ajettua automaattisesti ilman hukka-aikaa töiden välissä. Hinta, joka tästä oli maksettava, oli että monitoriohjelman tarvitsema muistitila ei ollut käytettävissä itse töiden suoritukseen. Aikakauden tietokoneissa oli todella vähän muistia, joten se oli erittäin kallisarvoinen resurssi. Lisäksi eräajo vaati sopimuksia ajettavien

ohjelmien hyvästä käytöksestä: suorituksen loputtua tapahtuvasta kontrollin siirrosta takaisin monitorille sekä siitä, että ajettava työ ei vahingossa riko monitoriohjelmaa kirjoittamalla sotkua sen päälle. Tietokoneen muistin jakaminen käyttöjärjestelmän osaan ja käyttäjän ohjelman osaan nähtiin tarpeelliseksi. Ihminen kuitenkin tekee virheitä välttämättä, joten täydelliseksi kuvio voisi tulla vasta, jos laitteisto pystyisi teknisesti estämään ohjausjärjestelmälle varattuihin muistiosoitteisiin kirjoittamisen.

1960-luku: Käytössä oli edelleen monitoriohjelmia sekä työnohjauskieli (”JCL”, job control language), jolla monitorille saattoi kertoa esim., että seuraavaksi sille tulee syötteenä FORTRAN-kielinen ohjelma, joka pitää kääntää ja ajaa. Käännetyt ohjelmat oli mahdollista tallentaa vaihtoehtoisesti myös massamuistiin (magneettinauhalle ja sittemmin kovalevylle), jolloin niistä saattoi tehdä vähän isompia – massamuistista ne voitiin nimittäin ladata jälkikäteen muistiin kääntäjäohjelman tilalle. Myöskään kaikkien kielten kääntäjiä ei tarvinnut säilyttää muistissa, koska niistäkin voitiin ladata massamuistista vain se, jota kulloinkin tarvittiin.

Silloinen käyttöjärjestelmä siis hoiti ohjelman kääntämisen, lataamisen ja käynnistämisen. Eräajossa ohjelman tarvitsema data sijoitettiin mukaan syötepakkaan, heti koodin perään, jotta käynnistetty ohjelma pääsi lukemaan omaa syötettään siinä järjestyksessä kuin se korteissa luki. Edelleenkään ei ollut mm. moniajoa eikä muistinsuojausta. Havaittuja ongelmia olivat:

- muistin suojaus – käyttäjän ohjelma ei saisi huolimattomalla muistiin sijoittamisella rikkoa monitoriohjelman ohjelmakoodia. Prosessoriin tarvittiin ominaisuus, jolla laittomat muistiviittaukset voidaan havaita ja siirtyä saman tien tilanteen käsittelyyn virhetapahtumana. Yhden työn päättyminen vir-

heeseen ei saanut estää seuraavan työn normaalia lataamista ja käynnistämistä.

- aikakatkaisu – käyttäjän ohjelma ei saisi vahingossa jäädä pyörimään ikuisiksi ajoiksi, mikä estäisi seuraavan työn lataamisen. Sen sijaan ohjelman alussa pitäisi voida asettaa maksimiaika, jonka jälkeen prosessori pystyisi keskeyttämään työn väkipakolla ja siirtämään kontrollin käyttöjärjestelmälle. Laitteistoon tarvittiin siis ajastinkello ajastettua keskeytystä varten ja prosessorin suoritussykliin kellon tilanteen tarkistava vaihe.

- suojatut laitekomennot – käyttäjän ohjelma ei mielellään saisi vahingossakaan tehdä sellaisia I/O -toimintoja, jotka haittaavat muita käyttäjiä (esimerkiksi lukea syötettä sen yli mitä omalle ohjelmalle oli tarkoitettu; siitä menisi kirjaimellisesti pakka sekaisin, koska reikäkortteja ei voinut oikein selata edestakaisin). Prosessoriin haluttiin ominaisuus, jolla käyttäjän ohjelman tekemiä toimintoja voidaan rajoittaa ja pakottaa esimerkiksi syötöt ja tulostukset kulkemaan aina käyttöjärjestelmäohjelman kautta.

- oli siis selkeä tarve, että prosessori toimisi tarpeen mukaan jommassa kummassa kahdesta toimintatilasta: **käyttäjätilassa** (engl. *user mode*), jossa normaali ohjelma toimisi vain rajoitetuin oikeuksin tai **valvontatilassa** (engl. *supervisor mode*), jossa käyttöjärjestelmä pystyisi hallitsemaan koko laitteistoa. Käyttäjätilan ohjelma ei saisi pystyä vahingossakaan kajoamaan suoritusvuorolistaan tai muihin ohjausohjelmiston osiin. Tulevaisuuden käyttöjärjestelmä tarvitsisi siis tulevaisuuden prosessorin, joka voi toimia jommassa kummassa kahdesta erilaisesta suojaustilasta. Suljetummassa toimintatilassa täytyisi olla tekniset esteet joidenkin muistiosoitteiden

käytölle, ja vapaammassa tilassa pitäisi tietysti olla suoritettavissa käskyjä, joilla määritetään, minkä osoitteiden käytön suljettu tila estää ja mitkä se sallii.

- prosessorin ominaisuudeksi haluttiin **keskeytykset** (engl. *interrupt*), joilla aikaikkunan sulkeuduttua, virhetilanteen ilmetessä, tai I/O-laitteen käytön yhteydessä pystyttäisiin automaattisesti keskeyttämään meneillään olevan ohjelman suoritus ja siirtämään suoritus käyttöjärjestelmälle.

1960-luku (edelleen): Havaittiin, että hukka-aikaa menee myös I/O-toimenpiteiden suorittamiseen, koska mm. massamuistit ovat prosessoriin verrattuna hitaita (ja mikä tahansa interaktiivinen syöte kuluttaa ennalta tuntemattoman, käyttäjästä riippuvan, ajan). Optimitilanteessa muistissa olisi useita ohjelmia, joista yhtä voitaisiin suorittaa samaan aikaan kun toiset ohjelmat odottelevat esim. pyytämänsä I/O-toimenpiteen valmistumista. Tämän ratkaisuna olisi **moniajo** (engl. *multitasking*), toiselta nimeltään **moniohjelmointi** (engl. *multiprogramming*).

Moniajokäyttöjärjestelmä tarvitsee toimiakseen prosessorin, jossa meneillään oleva suoritus voidaan keskeyttää ja siirtää prosessori suorittamaan käyttöjärjestelmän koodia. Samalla hetkellä ilmeisesti täytyy tapahtua prosessorin tilan vaihto käyttäjätilasta valvontatilaan (kun järjestelmäohjelmiston yleinen rakenne ja terminologia on sittemmin vakiintunut, valvontatilaa voidaan sanoa myös käyttöjärjestelmätilaksi tai ydintilaksi). Ohjelmat eivät saa haitata toistensa toimintaa, joten tietokonelaitteistossa täytyy olla ominaisuudet myös muistinhallintaa varten. Moniajoa tukevan käyttöjärjestelmän pitää ottaa tietokoneen muisti haltuunsa ja jaella sitä hallitusti ohjelmien käyttöön.

Syntyi **aikajakojärjestelmiä** (engl. *time-sharing systems*). Tie-

tokoneet olivat yhä isoja ja kalliita, joten niitä ei voitu hankkia jokaista työntekijää varten. Kuitenkin moni työntekijä olisi pystynyt tekemään tehokkaampaa työtä tietokoneen avulla. Ratkaisuna tähän olivat **päätteet** (engl. *terminal*), joiden kautta käyttäjät saattoivat hyödyntää samaa keskustietokonetta eri tehtäviin samanaikaisesti. Käytännössä ohjelmat toimivat vuorotellen, ja käyttäjien kesken jaettiin aikaikkunoita/aikaviipaleita. Syntyi **prosessin** (engl. *process*) käsite kuvaamaan yhden ohjelman suoritusta (ja tiettyjä siihen läheisesti liittyviä asioita). Mainittakoon merkkipaaluna Multics -järjestelmä, jossa prosessia nimitettiin ensimmäisen kerran juuri tuolla nimellä, joka on jäänyt käyttöön aina siitä saakka.

Useiden prosessien ja käyttäjien jakamista laiteresursseista syntyivät luonnollisesti lisähaasteet prosessien yhteistoiminnan koordinoinnissa. Lisäksi tietokoneen muistin sekä käyttäjien ja heidän käyttöoikeuksiensa hallinta tuli entistä tärkeämmäksi. 1960-luvun aikana myös kehittyivät merkittävällä tavoin keinot, joilla välimuisteja voidaan hyödyntää suorituksen nopeuttamiseen. Välimuistien käytön teoreettisena pohjana on empiirisesti tutkittu ja havaittu ns. **lokaalisuusperiaate** (engl. *principle of locality*), joka tutkimuksen kautta hahmottui noin 1960-luvun loppuun mennessä. Olennainen havainto oli, että tyypilliset tietokoneohjelmat käyttävät koodin ja datan käsittelyssä kohtalaisen pitkään samaa ”lokaalia” aluetta ennen kuin (verrattain harvakseltaan) siirtyvät käsittelemään eri aluetta. Toimintoja voidaan nopeuttaa pitämällä pientä peräkkäistä datapätkää nopeassa ja lähellä sijaitsevassa muistissa. Periaatteeseen nojaavia välimuistijärjestelmiä löytyy nykyään joka paikasta – tietokonelaitteiston lisäksi välimuisteilla voidaan nopeuttaa tiedon saantia mm. Internetissä ja tietokannoissa. Lisätietoja saa esimerkiksi käsitteen kehittäjän, Peter Denningin, kirjoittamasta historiikista [10].

Nykyisten moniydinprosessorien kaltaisia laitteita oli jo käytössä, joten niidenkin koordinointiin liittyvä teoria ja käytännön ratkaisut kehittyivät 1960-luvulla.

1970-luku: Käytössä olevia järjestelmiä olivat mm. Multics sekä kehitteillä olevat UNIX ja MS-DOS. Käyttöjärjestelmästä oli tulossa selvästi aiempaa monipuolisempi järjestelmä. Aiempaa suurempaa koodimassaa oli hankalampi hallita, joten uutena haasteena oli tarvittavan kokonaisuuden jäsentäminen siten, että käyttöjärjestelmän laatu ja laajennettavuus saatiin säilymään. Tuloksena oli käyttöjärjestelmältä edellytettyjen piirteiden ja tehtävien täsmällinen luettelointi ja kerroksittainen jäsennysmalli. Uusia käyttöjärjestelmiä kehitettiin poimimalla mukaan parhaita käytäntöjä aiemmista ja luomalla uusia esimerkiksi paikallisen tutkimusryhmän tarpeisiin. Mielenkiintoista lisälukemista on esimerkiksi Unixin ja C-kielen varhaisia kehitysvaiheita kuvaileva Richien ja Thompsonin artikkeli [11] vuodelta 1974. Nykyistä tulkintaa käyttöjärjestelmän jäsennyksestä moduuleihin vastaavat jossain määrin esimerkiksi tämän luentomonisteen tai jonkin prosessorimanuaalin järjestelmäohjelmointia kuvaavan osan sisällysluettelo tai Linuxlähdekoodin hakemistorakenne karkeimmalla tasollaan.

1980-luku: Moniajon ja suojausten tehokkuutta oli alati kehitettävä monen käyttäjän järjestelmissä. Laitteistopuolella yleistyivät moniydin- ja rinnakkaisprosessorit mahdollisuuksineen ja toisaalta uusine haasteineen. Mikrotietokoneet rynnistivät koteihin ja yrityksiin, samoin kuin mitä moninaisimpien oheislaitteiden kirjo. Jokainen erilainen oheislaite (tietokoneen kannalta I/O-laite) tarvitsee oman ajurinsa, joka on käyttöjärjestelmän osa tai ainakin läheisessä yhteistyössä käyttöjärjestelmän kanssa, joten laitetarjonta omalta osaltaan kasvattaa huimasti käyttöjärjestelmäkoodin määrää. Laiterajapintoja on toki aina pyritty standardoimaan, mutta

standardien neuvottelu oli varsinkin alkuaikoina hitaampaa kuin uusien tuotteiden julkistaminen. Oheislaitestandardit koskivat lähinnä fyysistä liittämistä eli kuparinastojen sijoittelua väylän liitännöissä; se, mitä ohjauskomentoja laitteille piti antaa, ei ollut yhteisesti sovittua.

Suuren mullistuksen toivat myös verkkoyhteydet ja internet. Tutkimuksen kohteeksi tulivat hajautetut käyttöjärjestelmät ja oliopohjaisuus (ylipäätään ohjelmistoissa ja luonnollisesti myös käyttöjärjestelmissä). Tietokonevirukset melkeinpä konkreettisesti ilmaistuna ”karkasivat laboratorioista”, joissa ne olivat vielä 1970-luvulla pysyneet. Tietoturvan ja suojausten tarve on siten luonnostaan kasvanut, vaikuttaen laitteiston, käyttöjärjestelmien, ohjelmointityökalujen ja sovellusohjelmien kehitykseen[32].

Varhaisten kodeissa ja yrityksissä käytettyjen henkilökohtaisten tietokoneiden (engl. *PC, personal computer*) mukana toimitetut käyttöjärjestelmät, ikävä kyllä, eivät monin paikoin vielä 1980-luvulla toteuttaneet aiemmin hyväksi havaittuja käytäntöjä muistinsuojauksen ja moniajon suhteen. Tieto ja tutkimus oli saatavilla, mutta henkilökohtaisten koneiden käyttöjärjestelmissä ohjelmat ja käyttäjät pääsivät suojausten puuttuessa helposti sotkemaan toistensa toimintaa, joko vahingossa tai pahantahtoisuuden vuoksi.

Vuonna 1983 Richard Stallman ilmoitti aloittaneensa Unix-yhteensop mutta lähdekoodiltaan vapaan, käyttöjärjestelmän tekemisen ja tarvitsevansa apua projektin toteuttamiseen. Aloitetun tuotteen

[32]Perillä ei selvästi vieläkään olla, mikä huomattiin mm. kevään 2014 käyttöjärjestelmäkurssin mittaan, kun ns. ”Heartbleed” -tietoturva-aukosta uutisoitiin maailmalla laajasti. Ja onhan näitä aukkoja ikävä kyllä aina, kun ihmiset tekevät järjestelmiä. Jatkokursseilla Tietoverkkoturvallisuus ja Ohjelmistoturvallisuus perehdytään siihen, mitä sovellusohjelmoija voi yrittää tehdä minimoidakseen turvamokien mahdollisuutta. Laajempia näkökulmia tarjotaan Informaatioturvallisuus-maisteriohjelman lukuisilla kursseilla.

nimeksi hän ilmoitti tavoitteen mukaisesti GNU (“Gnu’s Not Unix”). Projektin mittavuudesta voi saada kuvaa silmäilemällä vaikka nykyistäkin POSIX-standardia, joka kuvailee ominaisuudet, jotka olisi käytännössä kirjoitettava puhtaalta pöydältä, jotta yritysten omistamaa suljettua koodia ei joutuisi mukaan.

1990-luku: Myös henkilökohtaisten tietokoneiden yleisten käyttöjärjestelmien uudet versiot alkoivat pikkuhiljaa toteuttaa 1960- ja 1970-luvulla opittuja periaatteita. Yleisissä prosessoreissa oli pitkälti valmiina nykyisenkaltaiset ominaisuudet, mukaan lukien kotitietokoneissa yleinen Intelin määrittelemä 80386-arkkitehtuuri. Myöhempi kehitys on ollut lähinnä laitteiden pienentymistä, nopeutumista, useampien prosessoriytimien lisäämistä ja perusarkkitehtuurin mukaisia toimintoja nopeuttavien laitekomponenttien lisäämistä kokonaisuuteen.

Linux syntyi vuonna 1991, kun suomalainen Linus Torvalds teki itselleen käyttöjärjestelmäytimen tietokoneelleen, jossa oli 80386-prosessori. Pohjana oli Andrew S. Tanenbaumin akateeminen oppikirjaesimerkki Minix. Linux-ydin on ollut alkuajoista lähtien kytköksissä GNU-projektin unix-maisiin apuohjelmistoihin, joita Torvalds käytti ytimensä päällä toimivana käyttäjärajapintana sekä ytimen kehitystyöhön. GNU:n kehittäjien mielestä Linux-ytimen ympärille rakenneltua käyttöjärjestelmäjakelua tulisi kutsua yhdistetyllä nimellä GNU/Linux silloin, kun merkittävässä roolissa on tyypilliset GNU:sta lainatut työkaluohjelmat.

Vuoden 1993 paikkeilla myös Berkeleyn yliopiston aiemmin suljettu käyttöjärjestelmä BSD (”Berkeley Software Distribution”) oli pääosin muunnettu avoimeksi lähdekoodiksi kehittäjiensä toimesta. Mm. BSD:n kautta unix-maiset ideat ja toiminnallisuudet ovat suotautuneet moniin myöhempiin käyttöjärjestelmiin, mukaanlukien Windowsin lukuisiin versioihin ja ehkä vielä suuremmin osin

Mac OS X:ään.

Käyttöjärjestelmien (mukaanlukien siis apuohjelmistot) lisäksi muidenkin ohjelmistojen lähdekoodin vapautta ja avoimuutta ajavia yhteisöjä syntyi 1990-luvulla. Nykyään suuri osa tavanomaisista tietokoneen käyttötarkoituksista voidaan hoitaa jollakin ohjelmalla, jonka lähdekoodia käyttäjän on mahdollista itse tutkia, muutella ja kehittää eteenpäin[33].

2000-luku: Kännykät (nykyisin älypuhelimet) ym. sulautetut laitteet yleistyivät. Yksinkertaisissakin laitteissa, kuten digikamerassa, on oltava vähintään tiedostojärjestelmä, jotta kuvat saadaan tallennettua muistikortille (joka itsessään on vain ”normaali massamuisti”). Arkipäiväisissä laitteissa, auton jarruista ja kerrostalon ilmastoinnista alkaen, on tietokoneita, joihin on toteutettava käyttöjärjestelmä tai ainakin käyttöjärjestelmän perustehtäviä hoitavia koodin osia. Sulautetut, akuilla toimivat, laitteet tuovat mukanaan joitakin haasteita ja kompromisseja: Jokainen tietokoneen suorittama käsky vaatii jonkin verran sähköä sen lisäksi, mitä menee muistissa olevan datan ylläpitoon. Akkukeston mielessä olisi suotavaa, että esimerkiksi älypuhelimen, tabletin tai kannettavan tietokoneen prosessori ei tekisi liiemmin ylimääräistä työtä. Nykyisen (ainakin kuluttakäyttöön tarkoitetun) prosessorin ominaisuutena tarvitaan virransäästötiloja, ja käyttöjärjestelmän, kuten muidenkin ohjelmistojen, on pidettävä toimintansa maltillisena, ylimääräisiä käskyjä ja ”odottelusilmukoita” välttäen. Jossain vaiheessa mm. siirryttiin tasaisesta kellokeskeytyksestä (käyttöjärjestelmä herää esim. 100 kertaa sekunnissa tarkistamaan, tarvi-

[33]Esimerkkejä ovat mm. WWW-palvelin Apache, selain Firefox, toimisto-ohjelmistot Open Office ja Libre Office, 3D-mallinnusohjelmisto Blender, pikseligrafiikkaohjelma Gimp, vektorigrafiikkaohjelma Inkscape, sävellysohjelma Rosegarden, audioraituri Ardour, matemaattiset ohjelmistot Octave ja R, ladontajärjestelmä TEX ja monet muut.

taanko toimenpiteitä) tarpeen mukaiseen keskeytykseen (käyttöjärjestelmä herätetään vaikkapa vain näppäimen painallukseen), jolloin tietokone voi todellakin ”vain nukkua” silloin, kun siltä ei edellytetä jotakin toimintoa[34].

Nykyhetki (2016; monisteen kirjoittajan tulkinta): Pilvipalveluista on tullut arkipäivää (Google, Facebook, Twitter, Dropbox, ...) ja voitaneen ajatella, että se kaikkein henkilökohtaisin tietokone kulkee nykyään taskussa älypuhelimen muodossa. Tuo henkilökohtainen tietokone kommunikoi radioteitse linkkimaston ja Internetin kautta pilvipalveluiden ja niiden kautta muiden henkilökohtaisten tietokoneiden kanssa. Aika paljolti ollaan siis hajautetussa, tietoverkon yli kommunikoivassa maailmassa, jossa laitteet ja sähköiset esineet ovat välittömässä yhteydessä toinen toisiinsa, mutta mahdollisesti hyvinkin etäällä toisistaan maantieteellisesti. Myös pilvipalvelut ovat sisäisesti hajautettuja, johtuen mm. suurten käyttäjämäärien tarvitsemasta laskentakapasiteetista. Päivän sana onkin ehkä juuri hajautus ja massiivinen rinnakkaislaskenta. Hajautuksen tuomat haasteet eivät ole enää historian mielessä mitenkään uusia, mutta ne ovat selvästi kohteita, joihin tällä hetkellä on hyvä kiinnittää huomiota. Globaali informaatioavaruus tuo mukanaan myös verkkorikollisuutta, -sodankäyntiä ja -terrorismia, joilta suojautuminen voidaan nähdä tämän päivän kuumana aiheena. Niin laitteiston kuin niitä ohjaavan käyttöjärjestelmän tulee kehittyä ainakin näitä tavoitteita kohti. Tämän kurssin loppuosassa otamme haltuun käyttöjärjestelmien peruskäsitteistön, jonka pohjalta on mahdollista siirtyä syventäville jatkokursseille esimerkiksi

[34]Lisähaasteita tulee myös totaalisemmista virransäästötiloista (ts. esim. kannettavissa koneissa käytetyt uni- ja horrostilat), joissa myös I/O-laitteet on sammutettava ja herätettävä, kukin omilla laitekomennoillaan, oikeassa järjestyksessä, mutta kuitenkin niin että työskentelyä voi jatkaa samasta tilanteesta. Täytyy toteuttaa siis järjestelmän normaalin käynnistyksen ja sammuttamisen lisäksi ”osittaisia” alas- ja ylösajotapoja.

hajautukseen, pilvipalveluihin, rinnakkaislaskentaan tai kyberturvallisuuteen liittyen.

Tulevaisuus: Tulevaisuutta tuskin voidaan ennustaa sen kummemmin kuin kotitietokoneiden, Internetin tai älypuhelimien ilmaantumista aikoinaan voitiin. Ilmeisesti jatkuvia trendejä ovat hajautus, liikkuvat laitteet sekä moniydinprosessorit. Kulman takana voi kuitenkin olla jotakin uutta mullistavaa, joka tuo taas uusia haasteita ratkottavaksi. Seuraava fundamentaali harppaus eteenpäin 1940-lukulaisesta bittilogiikkatietokoneesta voi olla kvanttitietokone, joka operoi ”kvanttibiteillä” (engl. *quantum bit, qubit*). Mitä tämä tuo arkipäivän tietotekniikkaan ja kvanttilaskentaa hyödyntävän tietokoneen käyttöjärjestelmään, jää nähtäväksi, mahdollisesti suhteellisen piankin. Tässä vaiheessa tutkimus aiheen parissa on erittäin kuumaa teknisen laitetoteutuksen puolella. Mikäli historia toistaa itseään, myös sovelluksiin ja ”kvanttiohjelmistotekniikkaan” liittyvät perusideat syntyvät mahdollisesti jo nyt, vaikka niitä päästään hyödyntämään vasta vuosien päästä, tekniikan tultua markkinoille. Etenkin fysiikkaa pidemmälle lukeneet opiskelijat voivat päästä näinä päivinä (2016) eturintamaan tutkimaan ja luomaan tulevaisuuden kvanttitietotekniikkaa ensimmäisten joukossa.

Yhteenveto käyttöjärjestelmän tehtävistä

Historiaa ovat ohjannet uusien teknologioiden ja laitteiden sekä myös käyttötarpeiden ilmaantuminen. Lisäksi käyttöjärjestelmä on itsessään laaja ohjelmisto, johon tulee helposti (jopa väistämättä) monentasoisia vikoja, joita on korjattava, mahdollisesti pohjasuunnittelun tasolta alkaen. Laitteiston, ohjelmiston ja sovelluskohteiden kehityskaaren tuloksena käyttöjärjestelmäohjelmiston vastuik-

si ovat muodostuneet ainakin seuraavat tehtävät (sovellettu aika suoraan Stallingsin kirjasta):

- Ohjelmien suorittaminen ja yhteistoiminnan koordinointi.
- Apuohjelmakirjastojen (".DLL", ".so") hallinnointi.
- Syöttö- ja tulostuslaitteiden (eli I/O-laitteiden) hallinta. Kaikki syötöt ja tulostukset kulkevat jossain vaiheessa käyttöjärjestelmän kautta.
- Tiedostojen hallinta. Fyysisten tallennuslaitteiden ohjaamisen lisäksi tähän kuuluu tiedostojen sisällön ja metatietojen fyysinen organisointi (tiedostojärjestelmä) ja osoitteiden määritys tiedostoille (tiedostojen sijainti jonkinlaisessa loogisessa hakemistorakenteessa).
- Järjestelmän käyttäjien ja käyttöoikeuksien hallinta. "Matti ei saa lukea Liisan tiedostoja ilman Liisan lupaa". (Ja järjestelmän asetusten muuttaminen pitää olla sallittua vain ylläpitäjälle.)
- Virhetilanteiden havainnointi ja käsittely - sisältää ohjelmistovirheiden lisäksi laitevikojen käsittelyn.
- Järjestelmän toimintojen tarkkailu ja kirjanpito, mm. lokitietojen ylläpito.
- Sovellusten binäärirajapinta (Application binary interface, ABI) eli tavat, joilla mm. käyttöjärjestelmää ja kirjastoaliohjelmia kutsutaan (sisältää järjestelmäkutsurajapinnan).

Lisäksi voidaan ajatella, että seuraavat voisivat olla käyttöjärjestelmän tehtäviä tai ainakin siihen hyvin läheisesti liittyviä:

- Sovellusohjelmarajapinta (Application programming interface, API) - vaikka tällainen on suurimmaksi osaksi jokaisen korkean tason kirjaston itselleen määrittämä rajapinta, osa kirjastoista keskustelee alaspäin suoraan käyttöjärjestelmän kanssa, joten osa API:sta täytyy olla käytettävän käyttöjärjestelmän ja prosessoriarkkitehtuurin mukaisesti toteutettu.
- Ohjelmakehityksen työkalut, mm. kääntäjät, tulkit, debuggerit, editorit, käyttöjärjestelmän ja laitteiston ohjaukseen liityvät kirjastot rajapintoineen.

Esimerkiksi käyttöjärjestelmästandardi POSIX käyttää määrittelyssään C-kielen lähdekoodirajapintaa ja tiettyjen C-kielisten kirjastojen rajapintaa. Se myös määrittää tiettyjen apusovellusten joukon, mukaanlukien shell-komentotulkin, C-kääntäjän ja ”visuaalisesti orientoituneen” editoriohjelman eli ”vi”:n olemassaolon yhteensopivassa käyttöjärjestelmässä.

Tavoiteasetteluja ja väistämättömiä kompromisseja

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

- maksimaalinen **käyttöaste** (engl. *utilization*): kuinka paljon prosessori pystyy tekemään hyödyllistä laskentaa vs. odottelu tai hukkatyö (eli kirjanpito ym. ”tehtävän kannalta tarpeeton” tekeminen)
- maksimaalinen **tasapuolisuus** (engl. *fairness*): kaikki saavat suoritusaikaa

- maksimaalinen **tuottavuus** (engl. *throughput*): aikayksikössä loppuun saatujen tehtävien määrä
- minimaalinen **läpimenoaika** (engl. *turnaround*): yksittäisen tehtävän suoritukseen kulunut aika
- minimaalinen **vasteaika** (engl. *response time*): odotus ennen toimenpiteen valmistumista
- minimaalinen **odotusaika** (engl. *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. Jotkut prosessit vaativat nopeita vasteaikoja, esimerkiksi ääntä ja kuvaa toistava prosessi ei saisi ”pätkiä” vaan uusi kuva on saatava ruutuun odottelematta – tämä vaatii prosessien priorisointia, mikä taas käy vastoin tasapuolisuutta ja muiden, vähemmän reaaliaikaisiksi katsottujen, prosessien vaste- ja odotusaikoja.

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!

Yhteenveto: Ensimmäisissä tietokoneissa 1940-luvulla ei ollut erillistä käyttöjärjestelmää, vaan käsin käännetyt konekieliset ohjelmat ladattiin muistiin ja käynnistettiin suoraan reikäkorttipinkasta. Konekieltä korkeamman tason ohjelmointikielet nähtiin jo varhain tarpeellisiksi. Lisäksi ohjelmien yleiskäyttöisistä osista alettiin tehdä kirjastoja.

1950-luvulla käytettiin ”monitoriohjelmia”, joiden avulla pystyttiin automatisoimaan ohjelmien peräkkäistä suorittamista järjestetty ”erä” töitä kerrallaan. Havaittiin tarve mm. tietokoneen muistin jakamiseen laitteistotasolla käyttöjärjestelmän ja käyttäjän ohjelman saavutettavissa oleviin alueisiin.

1960-luvulla käyttöjärjestelmä hoiti ohjelman kääntämisen, lataamisen ja suorittamisen. Havaitut ongelmat liittyivät muistinsuojaukseen (vahinkoviittaukset toisen ohjelman muistialueille), aikakatkaisuun (”ikuiset silmukat”), I/O -laitteiden suojaukseen (luku tai kirjoitus ohi sallitun alueen). Vuosikymmenellä havaittiin tarve prosessorin käyttäjä- ja valvontatilan erottamiselle laitetasolla ja tiettyjen muistiin ja I/O -laitteistoon liittyvien toimenpiteiden estämiseen käyttäjän ohjelmissa. Moniajoa kehitettiin: aikajakojärjestelmässä yhteiset laitteistoresurssit jaettiin useiden käyttäjien ja prosessien kesken aikaikkunoihin. Käyttäjät olivat pääkoneeseen yhteydessä tekstiä syöttävän ja tulostavan päätteen kautta. Haasteeksi muodostui prosessien yhteistoiminnan koordinointi. Lokaalisuusperiaate hahmottui ja sitä alettiin soveltaa mm. välimuisteissa.

1970-luvulla käyttöjärjestelmältä edellytetyt piirteet hahmottuivat uusien järjestelmien kehityksen myötä. Varhaiset UNIX-versiot olivat käytössä ja MS-DOS kehitteillä.

1980-luvulta lähtien mikrotietokoneiden ilmaantuminen ja Internet ovat mullistaneet kehityskaarta. Käyttöjärjestelmäkoodin määrä

ja monimutkaisuus on kasvanut entisestään mm. erilaisten laitteiden yleistyttyä. Tietoturvan ja suojausten tarve on lisääntynyt. Hajautetut järjestelmät ja oliopohjainen ohjelmointi ovat tulleet ajankohtaisiksi.

2000-luvulla sulautetut järjestelmät yleistyivät ja tarve optimoida prosessorin ja ohjelmien toimintaa mm. akkukeston suhteen kasvoi. Nykyään käyttöjärjestelmiltä vaaditaan hajautuksen ja massiivisen rinnakkaislaskennan tukemista. Nykypäivän kehitysongelmat ovat mm. hajautuksen ja tietoturvan teemoissa.

Kehityksen seurauksena käyttöjärjestelmien tehtäviksi ovat vakiintuneet ohjelmien suorittaminen ja yhteistoiminnan koordinointi, tiedostojen, apuohjelmakirjastojen ja I/O-laitteiden hallinta, käyttäjien ja käyttöoikeuksien hallinnointi, virhetilanteiden käsittely, järjestelmän toimintojen tarkkailu ja kirjanpito, sekä sovellusten binäärirajapinta.

Käyttöjärjestelmän toiminnalle voi asettaa mm. seuraavanlaisia mitattavia tavoitteita: käyttöaste, tasapuolisuus, tuottavuus, läpimenoaika, vasteaika ja odotusaika. Tavoitteet ovat usein keskenään ristiriitaisia, ja niihin pyritään vastaamaan kompromisseilla.

0.7 Keskeytykset ja käyttöjärjestelmän kutsurajapinta

Avainsanat: avainsanat: keskeytykset, keskeytyspulssi, keskeytyskäsittely, FLIH (First-level interrupt handling), Kernel-pino, keskeytysvektori, ohjelmallinen keskeytyspyyntö

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa kertoa, milloin tietokoneen prosessori suorittaa ohjelmakoodia käyttöjärjestelmätilassa ja milloin käyttäjätilassa sekä mitä eroja näillä tiloilla on keskenään [ydin/arvos1]
- ymmärtää keskeytyskäsittelyn hyödyt ja toisaalta implikaatiot sovellus- ja järjestelmäohjelmistojen kannalta [ydin/arvos1]
- tietää, missä olosuhteissa ja millä mekanismilla prosessori siirtyy keskeytyskäsittelyyn [ydin/arvos2]

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 (t. ”ytimiä”) 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[35]. Huomaa, että jo arkihavainnon perusteella sama ohjelma voi olla suorituksessa useana ns. instanssina: ohjelman voi klikata käynnistysvalikosta päälle monta kertaa esim. useiden tekstinkäsittelydokumenttien muokkaamista varten.

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 odotella pysähdyksissä ainakin sen aikaa, kun käyttöjärjestelmä tarkistaa, onko jonossa muita prosesseja odottamassa suoritusvuoroa. 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

[35]Käyttäjän ohjelmien lisäksi myös käyttöjärjestelmän koodia voidaan ajaa eri prosesseissa, joista kukin hoitaa jotakin tiettyä tehtävää. Käyttöjärjestelmän prosessit voivat tapahtua prosessorin ollessa käyttöjärjestelmätilassa tai sitten käyttäjätilassa tasa-arvoisesti käyttäjän ohjelmien kanssa. Jälkimmäinen mahdollistaa hienojakoisemmat suojaukset ja käyttöoikeudet käyttöjärjestelmän osioiden välille.

vielä toistaiseksi pääasiassa yhtä prosessia; useiden prosessien tilanteeseen mennään luvussa 0.8.

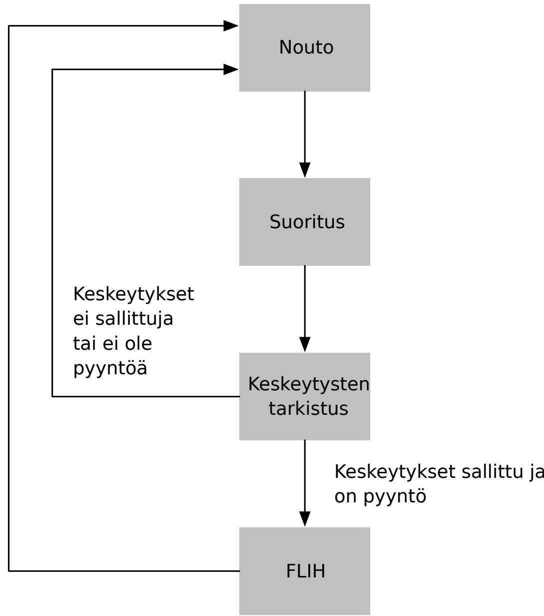
Suoritussykli (lopullinen versio)

Tietokonearkkitehtuuriin kuuluva ulkoinen väylä on kiinni prosessorin nastoissa, ja prosessori kokee nastoista saatavat jännitteet. Ainakin yksi nastoista on varattu **keskeytyspulssille** (engl. *interrupt signal*). Muita nimiä tälle voisivat olla **keskeytyspyyntö** (engl. *interrupt request, IRQ*) tai **keskeytyssignaali** (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 ottaa sen vastaan. Ja prosessori ehtiikin usein koko lailla välittömästi . . . täydennämme aiemmin yhdelle ohjelmalle ajatellun nouto-suoritussyklin seuraavalla versiolla, joka esitetään visuaalisesti vuokaaviona kuvassa 0.19:

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.



Kuva 0.19: *Prosessorin suoritussykli: nouto, suoritus, tilan päivittyminen, mahdollinen keskeytyksenhoitoon siirtyminen.*

4. **Keskeytyskäsittely** (engl. *interrupt handling*): Jos keskeytysten käsittely on kielletty (eli kyseinen tilabitti “FLAGS“-rekisterissä kertoo niin), prosessori jatkaa sykliä joka tapauksessa 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 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 [36]).

- 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.

[36]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.

- Keskeytyskäsittelyyn siirtyminen ei välttämättä edellytä siirtoa prosessista toiseen; käyttöjärjestelmän koodi on voitu liittää prosessin virtuaaliavaruuteen, ja samaa prosessia vain jatketaan nyt ”kernel running” -tilassa. Käyttäjän koodi ja käyttäjän näkemä konteksti on kuitenkin jäädytetty ja prosessori noutaa käskyjä virtuaalimuistiosoitteista, joissa on yksinomaan käyttöjärjestelmän koodia.

- 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 FLIHin 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.

- Käyttöjärjestelmän keskeytyskäsittelijä pääsee sitten suoritukseen.

- Keskeytynyt prosessi pääsee taas joskus myöhemmin jatkamaan tallentuneesta tilanteestaan, riippuen käyttöjärjestelmäkoodin 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. Siksi koko rumba täytyy tapahtua prosessorin suoritussyklin osana ennen seuraavan käskyn noutoa. Kaikki operaation osaset vaativat oman aikansa, joten FLIH ei tee enempää kuin pelkät välttämättömyydet. Joka tapauksessa jokainen FLIH on sovellusten kannalta ”hukka-aikaa” ja hukattua sähköä, mistä syystä esimerkiksi kiinteää, tiheään tahtiin tapahtuvaa kellokeskeytystä ei välttämättä kannata käyttää järjestelmässä, joka ei sellaista tarvitse.

Jos keskeytyskäsittelyssä pitää tehdä **kontekstin vaihto** (engl. *context switch*) eli vaihtaa suoritusvuorossa olevaa prosessia, käyttöjärjestelmän keskeytyskäsittelijän 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. Myös kontekstin vaihto on sovellusten kannalta hukattua aikaa, mikä johtaa kompromisseihin ainakin käyttöasteen, tuottavuuden, läpimenoaikojen ja tasapuolisuuden välillä.

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. Totta-

han 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 tai etenkään nauhoitukseen ei tule katkoja. Keskeytyskäsittelijän koodi tulisi olla siten tehty, että mahdollisimman pian käsittelijään siirtymisen jälkeen se suorittaa konekielikäskyn, joka jälleen sallii prosessorille uuteen keskeytykseen siirtymisen vaikkei edellinen käsittely olisi kokonaan loppunutkaan. Esimerkiksi AMD64:ssä tämä käsky on nimeltään STI (”set interrupt enable flag”), ja sellainen voidaan nähdä esim. Linuxin keskeytyskäsittelijöiden lähdekoodissa, toivon mukaan oikeisiin kohtiin kirjoitettuna.

Muutamia lisähuomioita keskeytyskäsittelystä:

- koskaan ei voi tietää etukäteen kuinka monta nanosekuntia, millisekuntia tai viikkoa esimerkiksi kestää ennen kuin käyttäjän ohjelman seuraava käsky suoritetaan, koska milloin tahansa 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 suori-

tukseen juuri silloin kun niiden tarvitsee. [37]

- 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.

Konekieltä suoritusjärjestyksen ohjaukseen: keskeytyspyyntö

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

```
syscall         # Pyydetään keskeyttämään tämä prosessi
                # ja siirt ämään suoritus ytimen koodiin.
                # Kutsun parametrit on oltava ennen tätä
```

[37] 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. Tällainen ajoitus oli jopa täsmällisempi kuin mihin nykyisin on mahdollista päästä. Käskyn vaatimat kellojaksot kun riippuvat mm. välimuistien ja liukuhihnoituksen hetkellisestä tilasta. Toisaalta nykyiset koneet ovat niin nopeita, että moisille kikkakolmosille ei liene normaalisti tarvettakaan. Tietoturvasyistä nykyisen käyttöjärjestelmän ei myöskään missään nimessä tule tarjota mahdollisuutta käynnistää ohjelmaa käyttöjärjestelmätilassa.

```
                              # laitettuna käyttöjärjestelmän tekij än
                              # määräämiin rekistereihin. Sovellus  voi
                              # olettaa, ett ä 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, ett ä
                              # seuraavaa käskyä nimenomaan ei koskaan
                              # suoriteta.  Sen sijaan  sovellus  ei  voi
                              # tietää,  kuinka kauan kestää ennen kuin
                              # sen suoritus  taas  jatkuu.  Se ei  myöskään
                              # ilman omia lisätarkistuksia  voi  tiet ää,
                              # onnistuiko pyydetty operaatio.
```

Prosessori suorittaa ohjelmoidun keskeytyksen kohdalla periaatteessa samankaltaisen FLIH-käsittelyn kuin laitekeskeytyksessäkin. Näin varmistuu luonnostaan, että prosessi keskeytyy kauniisti ja sen tilannetieto tallentuu. Luonnostaan tapahtuu myös prosessorin toimintatilan muuttuminen käyttöjärjestelmätilaan.

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::

```
sysret          # Ohjelmoidun keskeytyksen eli käyttöjärjestelmäkutsun
                # käsittelij än lopussa pit äisi  olla  tämä käsky.
```

```
                    # Se on käänteinen SYSCALL−käskylle, eli prosessori
                    # olettaa, että SYSCALLin aikoinaan tallentamat asiat
                    # ovat siellä, mihin SYSCALL ne laittoikin, ja sijoittaa
                    # ne takaisin alkuperäisiin sijainteihin. 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. Lisäksi tietyt AMD64−manuaalissa
                    # ilmoitetut rekisterit menettävät aiemman sisältönsä
                    # (muistaakseni RCX ja R11; tarkista itse halutessasi).
```

Ulkopuolisen keskeytyksen käsittelijästä palataan seuraavalla käskyllä:

```
iret                # Keskeytyskäsittelijän lopussa pitäisi olla tämä
                    # käsky. Se on käänteinen ulkoisesta syystä tapahtuneelle
                    # FLIH−käsittelylle (esim. I/O−operaation valmistuminen),
                    # eli prosessori ottaa pinosta FLIHin 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. Keskeytys voi
                    # tulla milloin tahansa, joten vaikka prosessi ei normaalia
                    # keskeytystä huomaa, se ei voi olettaa etteikö minkä tahansa
                    # käskyn jälkeen prosessori vaeltaisi väliaikaisesti muihin
                    # tehtäviin, mahdollisesti pitkäksikin aikaa.
```

Läheisesti samalla mekanismilla 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. SYSRET ja IRET oikeaan paikkaan koodia, jossa paluuosoite, RFLAGS ja muut tarvittavat tiedot ovat löytyvillä

sovituissa paikoissa (ennen muinoin pinossa; nykyisin mahdollisesti tietyissä rekistereissä, joista palauttaminen on nopeampaa kuin pinomuistista).

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 tuplasti nopeampi prosessori; rinnakkaisprosessoinnilla saadaan kuitenkin nopeutusta paljon halvemmalla kuin yhtä proses-

soria nopeuttamalla. Fysiikan laitkin rajoittavat yksittäisen prosessorin nopeutta paljon enemmän kuin rinnakkaisten yksiköiden määrää piirilevyllä.

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 tiedostoja. 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 x86-64 -prosessorin käskykannan vastaava käsky "syscall", jolla suoritettava ohjelma voi pyytää prosessoria keskeyttämään ja siirtymään käyttöjärjestelmän keskeytyskäsittelijään.

Perusidea käyttöjärjestelmäkutsuissa on seuraava:

- Yleensä aliohjelmakirjastot hoitavat kutsujen tekemisen, joten ohjelmoijalle näyttää siltä että hänen ohjelmansa kutsuu esim. C:n aliohjelmaa (tai Javan/C#:n metodia) kuten aiemman Hei maailma -esimerkin write() ja exit().
- Lopulta kuitenkin jossakin kirjastossa on ohjelmanpätkä, jonka konekielinen koodi sisältää nimenomaan ohjelmoidun keskeytyksen.[38]

[38]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).

- 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.

Yhteenveto: Moniajon kannalta eräs keskeinen teknologia ovat keskeytykset. Prosessorin suoritussykli on seuraava: nouto, suoritus, tilan päivittyminen ja mahdollinen keskeytyksenkäsittelyyn siirtyminen, mikäli keskeytyspyyntöjä on jonossa ja keskeytyksiä on sallittua tehdä.

Siirtyminen keskeytyskäsittelyyn (FLIH) sisältää seuraavat prosessorin toimenpiteet, jotka tapahtuvat käyttäjän ohjelman suorittaman käskyn jälkeen, ennen käyttöjärjestelmän käskyjen suorittamista:

1. Ennen keskeytystä voimassa ollut konteksti on tallessa: erityisesti RFLAGSin, RIP:n ja RSP:n sisällöt, eli seuraavan käskyn ja nykyisen pinon huipun muistiosoitteet on tallennettu.

2. RSP:hen on ladatty käyttöjärjestelmäkoodin käyttämän pinon huipun osoite.

3. RIP:hen on ladattu muistiosoite, josta pyydetyn keskeytyskäsittelijän suoritus alkaa.

4. RFLAGS on päivitetty siten, että prosessori on käyttöjärjestelmätilassa ja uudet keskeytykset ovat toistaiseksi kiellettyjä.

Keskeytyskäsittelijän koodi pääsee suoritukseen ja voi tarvittaessa esim. tallentaa muiden rekisterien sisältöjä (kontekstin loppuosa) muistiin prosessitauluun, mikäli prosessia tarvitsee vaihtaa.

0.8 Prosessi ja prosessien hallinta

Avainsanat: prosessi, konteksti, kontekstin vaihto, prosessitaulu, prosessielementti, vuorontaja, kiertojono, säikeet

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa kertoa, kuinka moniajo voidaan toteuttaa yksiprosessorijärjestelmässä ja tietää, kuinka moniprosessorijärjestelmä eroaa yksiprosessorijärjestelmästä sovellusohjelmoijan näkökulmasta [ydin/arvos1]
- osaa kuvailla prosessin elinkaaren vaiheet tilakaavion avulla sekä tulkita annettua tilakaaviota [ydin/arvos1]
- tunnistaa ja osaa luetella tilanteita, joissa samanaikaisten prosessien tai säikeiden käyttäminen on edullista ongelman ratkaisemiseksi (sisältäen muiden muassa lähtökohtaisen tarpeen moniajokäyttöjärjestelmälle) [ydin/arvos1]
- osaa selittää prosessin käsitteen sekä käyttöjärjestelmän prosessielementin ja prosessitaulun rakenteen pääpiirteet; tunnistaa prosessielementin nähdessään sellaisen ja osaa arvioida, onko kyseisessä elementissä mukana kaikki tarpeellinen tyypillisimpien käyttöjärjestelmätehtävien hoitamiseksi [ydin/arvos1]
- osaa selittää prosessin ja säikeen eron [ydin/arvos1]
- ymmärtää ja osaa selittää ohjelman käynnistämisen vaiheet unix-tyyppisten fork() ja exec() -käyttöjärjestelmäkutsujen tapauksessa [edist/arvos3]

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.

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).

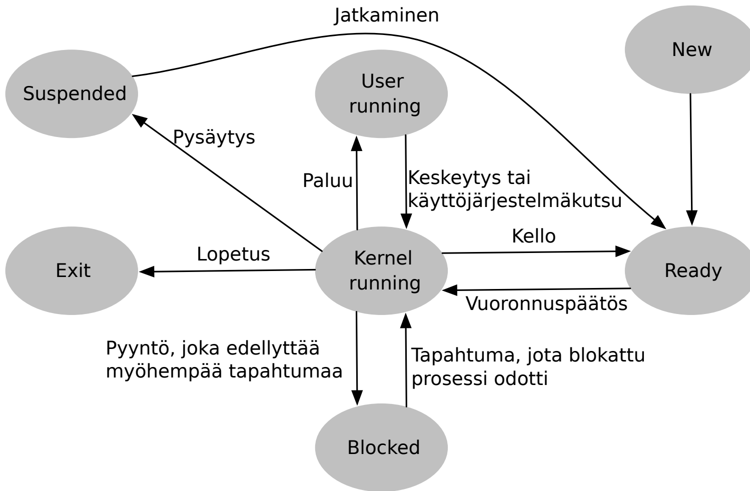
Prosessorissa täytyy tapahtua käyttöjärjestelmäohjelmiston koordinoima **kontekstin vaihto** (engl. *context switch*), kun prosessien suoritusvuoroja vaihdellaan. Pelkän käyttäjälle näkyvän prosessorikontekstin eli rekisterien sisältöjen lisäksi täytyy prosessorin muistinhallintayksikölle eli MMU:lle ilmoittaa seuraavaksi vuoroon tulevan prosessin muistiavaruuden kartta, jotta virtuaaliosoitteet

alkavat muuntua fyysisiksi oikealla tavalla. Muistin kartoittamisesta kerrotaan tarkemmin luvussa 0.10.

Prosessien välillä vaihtaminen on lukuisten operaatioiden vuoksi suhteellisen hidas toimenpide, joka on aina kaikkien varsinaisten ohjelmien kannalta ”hukka-aikaa”. Ilmiötä pahentaa sekin, että nopeakäyttöisissä välimuisteissa on todennäköisesti pääosin kopioita nykyisen prosessin muistisisällöstä, joten kontekstin vaihdon jälkeen tapahtuu myös välimuistien täyttämistä hitaammasta keskusmuistista.

Onneksi prosessia ei välttämättä tarvitse vaihtaa joka keskeytyksellä. Riippuu keskeytyksen luonteesta ja käyttöjärjestelmän vuorontajaan valituista algoritmeista, onko vaihdon tekeminen ajankohtaista. Yleensä mm. kellokeskeytys moniajojärjestelmässä kuitenkin tarkoittaa nykyisen prosessin aikaviipaleen loppumista, jolloin on syytä ottaa suoritukseen jokin toinen vuoroaan odottava prosessi. Toisaalta joku I/O, vaikkapa näppäinpainallus, voidaan lyhyesti kirjata käyttöjärjestelmän sisäiseen puskuriin odottamaan myöhempää käsittelyä, ja laitekeskeytyksen takia keskeytynyttä prosessia voidaan jatkaa ilman suurempaa vaihdosta. Tällaiset pikapalvelut mahdollistuvat sillä, että käyttöjärjestelmän koodi ja data on kartoitettu jokaisen prosessin muistiavaruuteen – käyttöjärjestelmän koodi suoritetaan ”tavallisen prosessin kontekstissa”, jolloin siirtymä on paljon pienempi kuin olisi vaihto erilliseen ”käyttöjärjestelmän kontekstiin”. Käyttäjän ohjelmalla ei luonnollisesti ole itsellään mitään oikeuksia käyttöjärjestelmälle kartoitettuun muistiavaruuden osaan, vaan se avautuu käyttöön vasta kun prosessori siirtyy käyttöjärjestelmätilaan keskeytyskäsittelyn yhteydessä. Mahdollisuudet riippuvat tietysti aina siitä, miten prosessoriarkkitehtuuri ja käyttöjärjestelmän toteutus on haluttu tehdä[39].

[39]Johdantokurssilla meidän kannattaa ajatella päätekstissä esitettyä perusideaa.



Kuva 0.20: *Prosessin tilat (geneerinen esimerkki).*

Kuvassa 0.20 on esimerkki tiloista, joissa kukin prosessi voi olla. Perustila on tietenkin "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. Tarkemmin ottaen "user runningiin" päästään aina vain "kernel-

Pienoinen lisätutustuminen x86-64:n toteutukseen ja siihen, miten Linux on sen päälle toteutettu, vaikuttaisi kertovan, että pikasiirtymä on mahdollinen vain SYSCALL-käsittelyssä. Laitekeskeytykset x86-64 tekee pidemmän kaavan kautta, siis raskaammin ja hitaammin. Linuxin keskeytyskäsittelyn siirtäminen x86-64:lle ei ole muutenkaan ollut aivan helppo temppu laiterajapinnan rajoitteiden vuoksi, vaikka alkuperäinen Linux oli jopa alunperin tehty x86-64:n edeltäjälle 386:lle. AMD64:n prosessorimanuaalin järjestelmäohjelmointiosa kertoo totuuden mm. keskeytyskäsittelystä.

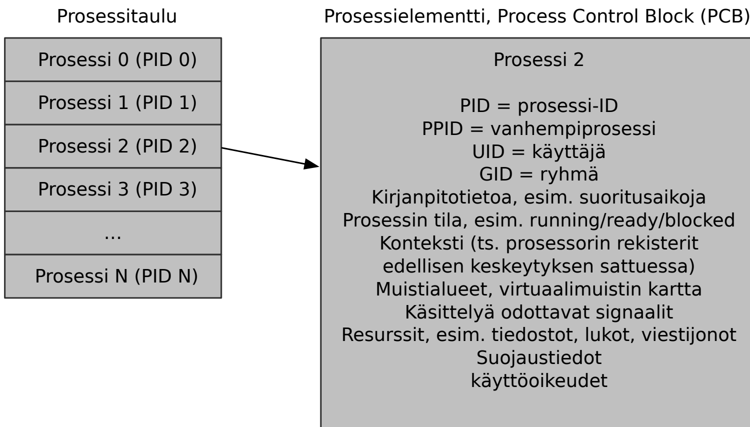
running"-tilan kautta, koska käyttöjärjestelmäkoodin on tehtävä kontekstin vaihtoon liittyvät operaatiot ennen kuin kontrolli on siirrettävissä käyttäjän koodin puolelle.

- 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, 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). Käyttöjärjestelmän toteutuksesta riippuu, tuleeko siirto "blocked"-tilasta suoraan "running"-tilaan vai ensin "ready"-tilaan odottamaan aikaikkunaa. Tällaisella toteutusvalinnalla on vaikutusta laitetta odottavan prosessin vasteaikaan ja vastakkainen vaikutus tasapuolisuuteen aikaikkunaa jonottavia prosesseja kohtaan.

"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 "syväjäädytettynä" odottelemassa myöhempää eksplisiittistä palautusta tästä tilasta. Luonnollisesti prosessin tiloja ovat myös "New", kun prosessi on juuri luotu ja "Exit", kun prosessin suoritus päättyy.



Kuva 0.21: *Käyttöjärjestelmän keskeisimmät tietorakenteet: prosessielementti ja sellaisista koostuva prosessitaulukko.*

Tässä esitetyt tilojen nimet ovat geneerisiä ja yleisesti ymmärrettyjä. Eri toteutuksissa nimet saattavat vaihdella, esimerkiksi ”Exit”-tila voi olla nimeltään ”Zombie” tai ”Terminated”. ”Running” ja ”Ready” voivat olla ”Running (executing)” ja ”Running(ready)” jne. Esitetyt tilasiirtymät vastaavat nykyisten monen käyttäjän järjestelmien tarpeita. Toteutus ei välttämättä tarvitse ”New” -tilaa, jos ohjelma voidaan ladata ja käynnistää suoraan ”Ready” tai ”Running” -tilaan.

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 0.21.

Prosessielementtien määrä on rajoitettu – esim. positiivisten, etu-

merkillisten, 16-bittisten kokonaislukujen määrä, eli 32768 kpl. Enempää prosesseja ei voi järjestelmässä olla yhtäaikaa. 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[40].

Yhden PCB:n sisältö on vähintään seuraava:

- prosessin yksilöivä tunnus eli prosessi-ID, "PID". Voi olla toteutuksen kannalta vaikkapa 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 oikeustarkistuksissa)
- prosessin tila (ready/blocked/jne...) ja prioriteetti
- resurssit

[40]Johdantokurssilla on soveliasta ajatella prosessien tietoja taulukkona; se olisi myös selkein ja tehokkain toteutustapa, mutta ei tietenkään skaalautuvin. Esimerkiksi Linuxissa prosessielementit ovat kiinteänmittaisen taulukon sijasta dynaamisesti allokoitavassa linkitetyssä listassa, joten niiden määrää rajoittaa viimekädessä muistitilan sijasta identiteettinumeron bittileveys. Oletusarvoisesti raja on myös Linuxissa 32786, mutta järjestelmänvalvojan on helppo nostaa sitä tarvittaessa; yksi shell-komento riittää tähän.

  - 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

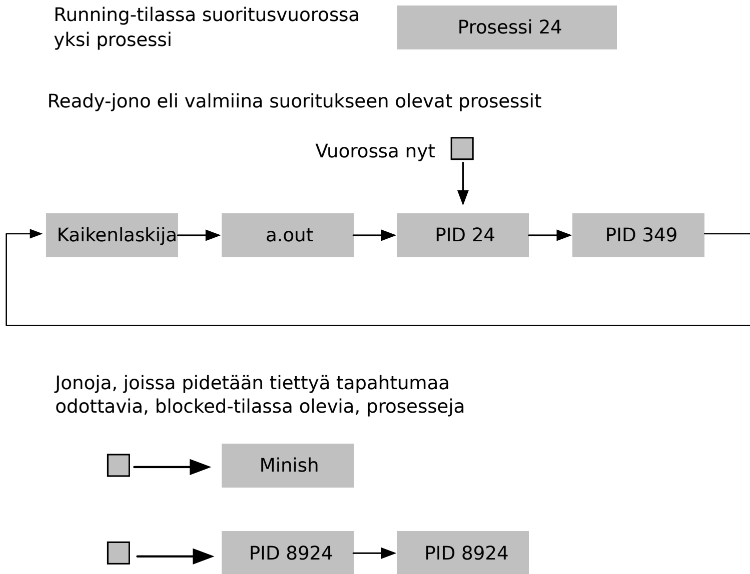
Käyttöjärjestelmän toteutuksesta riippuu, mitä muuta PCB:hen mahdollisesti sisältyy[41].

Vuorontaja

Käyttöjärjestelmän osio, joka hoitaa ajallisen resurssin, kuten prosessoriajan, jakamista prosessien kesken on nimeltään **vuorontaja** (engl. *scheduler*). Prosessivuorontajasta käytetään englanniksi myös nimeä "matkaanlähettäjä"(engl. *dispatcher*). Jotkut suomenkieliset tekstit käyttävät lainasanaa "skeduleri". Kuvassa 0.22 esitetään esimerkki yksinkertaisesta vuorontamismenettelystä nimeltä **kiertovuorottelumenetelmä** (engl. *round robin*), josta käytetään jatkossa lyhyttä nimeä **kiertojono**[42]. Tällainen jakaa tasavertaisesti aikaa kaikille laskentaa tarvitseville prosesseille: Prosessit sijaitsevat jonossa peräkkäin. Jos aika-annos loppuu yhdel-

[41] Esimerkiksi Linuxin lähdekoodi kun on avointa, niin sieltä voi halutessaan etsiä `task_struc` nimisen C-kielisen tietorakenteen määrittelyn, jossa on aika monta kenttää eri tarkoituksiin. Käytännössä monet Linuxin prosessielementin kentistä ovat osoittimia osa-aluekohtaisiin tietorakenteisiin.

[42] "Kiertojono" on tämän monisteen kirjoittajan omavaltainen suomennos. Alkuperäinen termi "round robin" lienee lainattu jonkinlaisesta urheilukilpailujen alkusarjamenettelystä, jossa kukin saa pelata vuorollaan tai jotain sellaista.



Kuva 0.22: *Prosessien vuorontaminen kiertojonolla (round robin). Prosesseja siirretään kiertojonoon ja sieltä pois sen mukaan, täytyykö niiden jäädä odottelemaan jotakin tapahtumaa (eri jonoon, blocked-tilaan).*

tä, siirrytään aina seuraavaan. Piiri pieni pyörii. Jos taas prosessi blokkautuu ennen aika-annoksen loppua, täytyy se tietenkin ottaa pois tästä suoritusvalmiiden eli ”ready”-tilassa olevien prosessien kiertojonosta ja siirtää eri jonoon, jossa on ulkoista tapahtumaa odottavia prosesseja (yksi tai useampia).

Round robin -menettelyssä ”ohiajot” eivät ole mahdollisia, joten se ei sovellu reaaliaikajärjestelmiin, kuten musiikkiohjelmien suorittamiseen. Myöhemmin, luvussa 0.13 palataan tutkimaan vaihtoehtoisia vuoronnusmenettelyjä.

Prosessin luonti fork():lla ja ohjelman lataaminen exec():llä

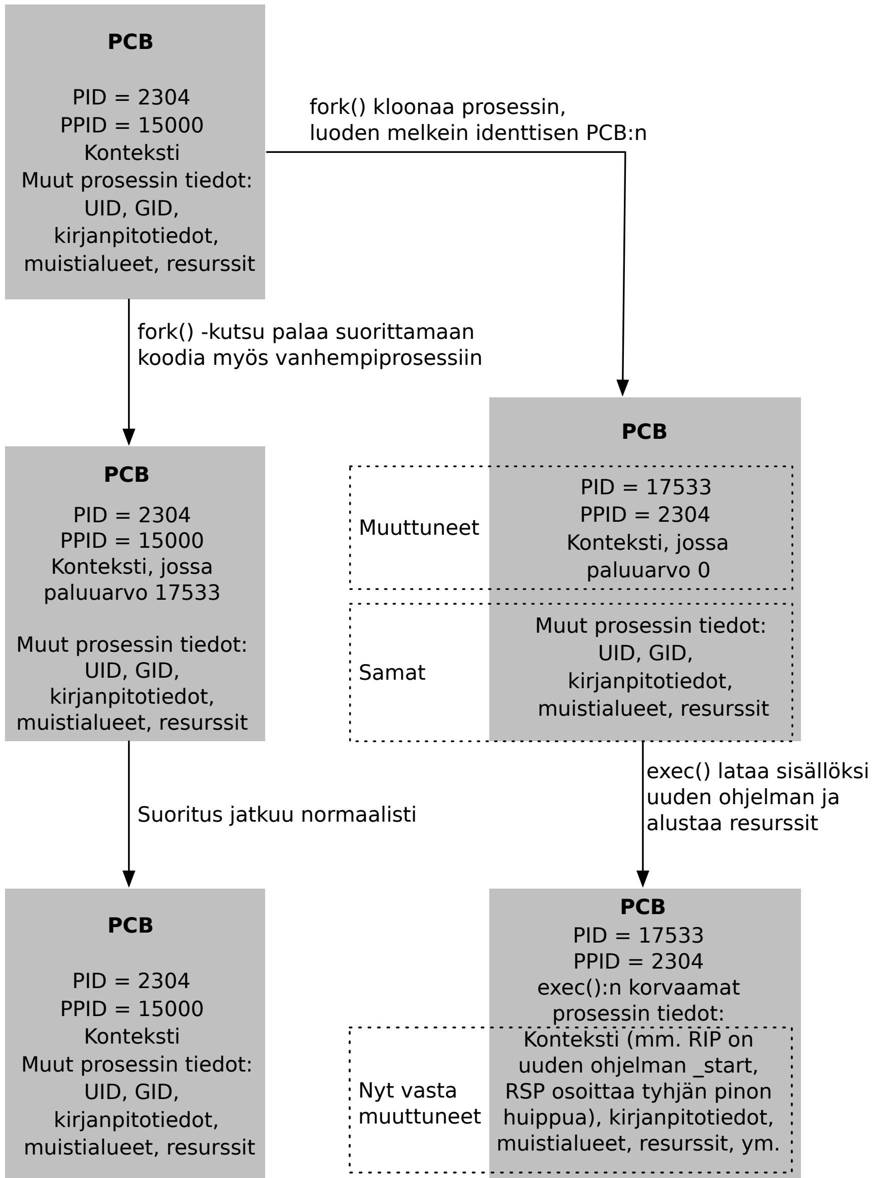
POSIX-standardi määrittelee C-kieliseen käyttöjärjestelmärajapintaan kuuluvaksi aliohjelman nimeltä fork(), jolla tehdään uusi pro-

sessi. Se onkin ainoa tapa, jonka POSIX määrittää prosessin luomiseen. Käytännön järjestelmässä fork() voi olla sellaisenaan suoraan käyttöjärjestelmäkutsu, tai C-kielen peruskirjastossa voi olla jokin kohtalaisen kevyt lisäkerros sen toteuttamiseksi[43].

Nyt, kun tässä vaiheessa kurssia ymmärretään käyttöjärjestelmän kutsurajapinnan laitetoteutus keskeytyksen aiheuttavana konekielikäskynä (nimeltään esim. "syscall", "int", "trap", "svc" tmv.) ja kerroksittaisten kirjastojen olemassaolo laiterajapinnan yläpuolella, voidaan huoletta siirtyä käsittelemään POSIXin määrittelemää C-kielistä rajapintaa, jonka päälle tehdyt ohjelmat ovat lähdekooditasolla siirrettävissä yhteensopivien järjestelmien välillä. Jatkossa termi "käyttöjärjestelmäkutsu" voidaan samaistaa "matalan tason" C-kirjaston kutsuun. Noustaan siis abstraktiotasolla yksi pykälä ylöspäin ja iloitaan siitä, ettei sovellusohjelmissa tarvita assemblerin kirjoittamista.

Kutsuissa on useimmiten mukana parametreja, ja jotkut niistä esiintyvät lähdekoodissa useina eri nimisinä variaatioina, mutta tradition mukaisesti kutsuista ja "kutsuperheistä" käytetään lyhyttä, vakiintunutta nimeä, jossa on perässä tyhjät sulkumerkit ilmaisemassa, että nyt on kyseessä C-aliohjelman kutsu. Parametrit jätetään siis mainitsematta, vaikka tiedetään, että jokaisella kutsulla tietty rajapinnassa määritelty parametrilista onkin. Alkupuolella nähtiin jo write() ja exit() -käyttöjärjestelmäkutsut. Nyt käsitellään fork() ja exec(). Jatkossa tulee lisää samalla menetelmällä nimettyjä kutsuja erilaisiin käyttöjärjestelmän palveluihin liittyen.

[43]Linuxissa fork() toimii, koska monet unix-maiset ja osin POSIX-standardoidut 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 sitten niin iso, että toteutuu perus-unixin fork(). Säikeistä lisää myöhemmin.



Kuva 0.23: *Uuden prosessin luonti fork()-käyttöjärjestelmäkutsulla. Uuden ohjelman lataaminen ja käynnistys edellyttää lisäksi lapsiprosessin sisällön korvaamista käyttöjärjestelmäkutsulla exec().*

Käyttöjärjestelmä luo fork()-kutsua käsitellessään hiukan muutetun kopion nykyisestä prosessista, joka pyysi itselleen ”forkkausta” eli haaroitusta. Kuvan 0.23 ensimmäinen siirtymä ylhäältä alaspäin havainnollistaa tapahtumaa. Uudella prosessilla on oma PID sekä omat PCB-tietonsa. Käyttöjärjestelmä on tässä vaiheessa luonut uuden prosessin, alustanut sille prosessielementin ja kirjoittanut prosessielementin tiedot prosessitauluun[44]. Tämä tietysti epäonnistuu, mikäli järjestelmässä on jo maksimimäärä prosesseja luotuna! PCB:n sisältö on toistaiseksi suurimmalta osin kopio vanhempiprosessin tiedoista:

- PID on uusi.
- Vanhempiprosessin PID eli PPID (”parent PID”) on tietysti uusi.
- Prosessin virtuaalimuistiavaruus on tässä vaiheessa osoitteineen ja sisältöineen identtinen vanhempiprosessin kanssa.
- 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.
- Muutkin tiedot kopioituvat identtisinä; lapsiprosessilla on siis vanhempansa UID ja GID (käyttäjän ja ryhmän ID:t) sekä samat resurssit (ml. tiedostot ja muistialueet kuten koodi, data, pino, dynaamiset alueet).

[44] Tai sitten prosessielementille on varattu tarvittava määrä muistia, ja sen alkuosoite on liitetty prosessilistaan.

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). Toimiva, aukikirjoitettu, ohjelma löytyy kurssin materiaalivarastosta luennoilla esitettyjen esimerkkien joukosta nimellä `minish.c`.

```
while(true) {
  luekomento(komento, argumentit); // "viiteparametrit" saavat
                                   // uuden sisällön päätteeltä
  pid = fork();
  if (pid > 0) {          // onnistui, lapsiprosessin PID saatu!
    status = wait();      // odota, että lapsiprosessi loppuu.
  } else if (pid == -1) {
    ilmoita("fork() epäonnistui!");
    exit(1);
  } else {
    // fork() palautti 0:n, joten tässä ollaan lapsiprosessissa!
    exec(komento, argumentit);
    // Tänne ei saavuta koskaan, jos käynnistäminen onnistui!
    ilmoita("Komentoa ei voitu suorittaa!");
```

```
        exit(1);
      }
    }
```

Ohjelmassa yritetään luoda lapsiprosessi forkkaamalla. 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. Onnistuneen exec()-kutsun jälkeen prosessin aiempi ohjelmakoodi on unohdettu täysin, eli sen jälkeen lähdekoodin myöhempää koodia ei tulla koskaan suorittamaan. On monta syytä, miksi exec() ei välttämättä onnistu. Esim. käyttäjä on pyytänyt komentoa, jota vastaavaa ohjelmatiedostoa ei löydy järjestelmästä, jokin ohjelmatiedoston tarvitsemista dynaamisista kirjastoista ei ole asennettu, muisti on loppu, massamuistijärjestelmän laitevika estää jonkin tarvittavan palasen lataamisen, . . . Ylipäätään, **jokaisen käyttöjärjestelmäkutsun jälkeen on aina tarkistettava paluuarvo kaikkien mahdollisten virhetilanteiden varalta**. Myös uudemmissa oliokielissä alimman tason kirjasto tarkistaa käyttöjärjestelmäkutsujen paluuarvot ja heittää tarvittavat poikkeukset, nimeltään esim. ”IOException” tiedonsiirron epäonnistuessa ja ”OutOfMemoryException”, jos muisti on täynnä ym.

Epäonnistuneen exec()-kutsun jälkeen on ilman muuta lopetettava prosessi – sehän on kloonattu lapsi, joka oli tarkoitus muuttaa erilaiseksi. Jatkaminen ilman exit()-kutsua jättäisi alkuperäisen shellprosessin odottelemaan ja käyttäjän toiminta jatkuisi pyytämättä ja yllättäen syntyneessä ”ali-shellissä”. Riittävän monen epäonnistuvan käynnistyksen jälkeen (esim. käyttöjärjestelmä ei loydä käyttäjän antaman komennon mukaista suoritettavaa ohjelmaa) prosessien maksimimäärä tulisi täyteen, eikä yhtään uutta proses-

sia voisi käynnistää koko järjestelmässä!

Kuva 0.23 etenee ylhäältä alas, näyttäen onnistuneen forkin ja execin vaiheet ja lopputuloksen, jossa käynnissä on kahtena prosessina kaksi eri ohjelmaa, joista toinen on toisen lapsi.

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. Tarpeen mukaan sille voidaan haluta luoda useampia säikeitä.

Huomaa, että yhdenaikaisuus voi täten olla täysin näennäistä, yhdellä prosessoriytimellä tapahtuvaa aikaviipaleiden jakamista. Suoritusta sanotaan **rinnakkaiseksi** (engl. *parallel*), jos samaa tehtävää ratkaisevat säikeet tai prosessit voivat olla samalla ajanhetkellä suorituksessa eri prosessoriytimissä. "Massiivinen rinnakkaislaskenta" (engl. *massively parallel computing*) perustuu isojen laskutehtävien jakamiseen toisistaan riippumattomiin osatehtäviin, jotka voidaan laskea sadoissa rinnakkaisissa säikeissä. Teknisesti kaikkein tehokkaimmaksi tämä saadaan silloin, kun ytimissä voidaan suorittaa synkronisesti samaa koodijonoa datan eri osille, ts. kaikki ytimet noutavat ja suorittavat aina saman käskyn yhtäaikaa ("single instruction stream, multiple data streams", SIMD). Esimerkiksi nykyiset grafiikkaprosessorit sisältävät satoja ytimiä, joista kukin laskee samaa ohjelmaa yhdelle 3D-vektorille (geometriavarjostimessa) tai kuvaruutupisteelle (fragmenttivarjostimessa) kerrallaan.

Tavanomaisissa ohjelmissa 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 useiksi samaan aikaan suorituksessa oleviksi prosesseiksi.

Säikeellä on oma:

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

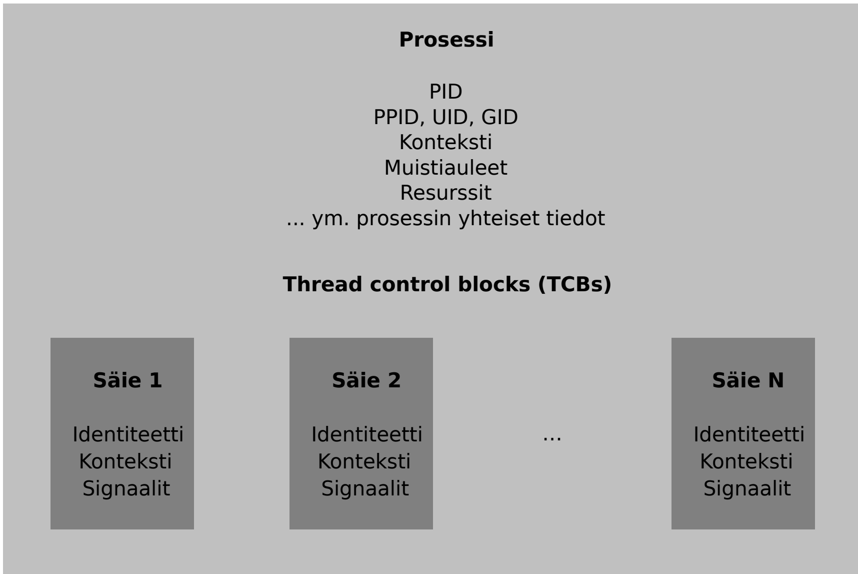
Säie on kuitenkin paljon kevyempi ratkaisu kuin prosessi; sitä sanotaankin joskus **kevyeksi prosessiksi** (engl. *light-weight process*). Kun säikeessä suoritettava koodi tarvitsee prosessin resursseja, ne löytyvät prosessielementistä, joita on prosessia kohti vain yksi. Säikeillä on siis käytössään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

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

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

Riippuen käyttöjärjestelmän monipuolisuudesta, säikeet voidaan toteuttaa joko käyttöjärjestelmän tuella tai ilman:

- ”Kernel-level threads”, KLT; Nimeltään joskus myös ”light-weight process”: Käyttöjärjestelmältä pyydetään säikeistys. Käyttöjärjestelmä näkee niin monta vuoronnettavaa asiaa kuin säikeitä on siltä pyydetty.
  - Moniprosessorijärjestelmissä voi kaikissa prosessoreissa olla yhtäaikaa eri säie. On siis mahdollista tehdä rinnakkaislaskennan kautta prosesseja, jotka toimivat seinäkelloajassa nopeammin kuin yhdellä säikeellä olisi mahdollista.
  - Toimii tietenkin vain käyttöjärjestelmässä, joka on suunniteltu tukemaan säikeitä vuoronnuksessa.



Kuva 0.24: *Voidaan ajatella että säikeet (yksi tai useampia) sisältyvät prosessiin. Prosessi määrittelee koodin ja resurssit; säikeet määrittelevät yhden tai useampia rinnakkaisia suorituskohtia.*

- ”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.
    - Toimii myös käyttöjärjestelmässä, joka ei varsinaisesti ole suunniteltu tukemaan säikeitä.
    - Säikeiden välinen vuorontaminen voidaan tehdä millä tahansa tavalla, joka ei riipu käyttöjärjestelmän vuoronnusmallista.

KLT-toteutuksessa käyttöjärjestelmällä voisi esimerkiksi olla tallessa PCB:n lisäksi ”säie-elementtien” (Thread Control Block, TCB) tiedot siten kuin kuvassa 0.24 on esitetty. TCB:tä voisi tosiaan sa-

noa suomeksi ”säie-elementiksi”, jollaisia sisältyy prosessikokonaisuuden PCB:hen eli prosessielementtiin yksi tai useampia[45].

Sekä prosessien että varsinkin säikeiden käyttö sisältää tilanteita, joissa yhdenaikaisesti suoritettavien ohjelmien tai ohjelmaosioiden tarvitsee jakaa keskenään jotakin tietoa, viestejä tai yhteisiä resursseja. Tämä ”moniohjelmointi” on jonkin verran monimutkaisempaa kuin yhden yksittäisen ohjelman suorittaminen; seuraava luku pureutuu tähän liittyviin erityiskysymyksiin.

Yhteenveto: Prosessi kuvaa muistiin ladatun ja käynnistetyn (konekielisen) sovellusohjelman suorittamista sekä ohjelman hetkellistä tilaa, joka riippuu aiemmista tapahtumista ohjelmassa sekä ympäröivässä järjestelmässä.

Prosessitaulu tai -lista koostuu prosessielementeistä, jotka sisältävät kunkin samaan aikaan suorituksessa olevan ohjelmainstanssin suorittamiseen tarvittavat tiedot. Näitä ovat mm. prosessin ja sen käynnistäneen vanhempiprosessin identiteetit (PID, PPID), konteksti edellisen keskeytyksen sattuessa, käyttäjän ja ryhmän iden-

[45]Kuvassa ja tekstissä on korostettu käsitteellistä yleistystä, että säie kuuluu prosessiin ja että prosessin resurssit ovat yhteiset kaikille säikeille. Esimerkiksi Linuxissa säikeet luodaan samalla clone()-nimisellä käyttöjärjestelmäkutsulla, jonka parametreilla voi määrittää parinkymmenen prosessiin liittyvän osa-alueen suhteen, kopioituvatko ne vanhemmasta vai luodaanko uudet. Linuxissa joka säikeellä on oma PID, eli ne ovat sananmukaisesti “kevyitä tai vähemmän kevyitä prosesseja”, missä keveys tai raskaus riippuu siitä, millä parametreilla clone()’a kutsuttiin. POSIX-yhteensopivuuden saavuttamiseksi säikeet piilotetaan normaaleissa prosessikyselyissä, ja niiden PID:ksi väitetään yhteistä TGID:iä eli ns. “säieryhmän identiteettinumeroa”, joka yksisäikeisen prosessin osalta onkin sama kuin PID. Alustariippumattomien ohjelmien tekemiseksi ei tulisi käyttää suoraan Linuxin clone() -kutsua, vaan POSIXin pthread_create()’a säikeen ja fork()’ia prosessin luomiseksi. Yhteensopivuuskirjasto kutsuu sitten kyllä sopivalla tavoin Linuxin clone()’a, että aikaan saadaan POSIXin lupaaman säikeen tai prosessin näköinen vehje. Tämäkin on yksi hyvä esimerkki siitä, miten standardin mukainen rajapinta on mahdollista toteuttaa ohuena kirjastokerroksena sisäisesti erilaisen rajapinnan päälle.

titeetit (UID, GID), prosessin tila ja prioriteetti, moninaiset resurssit sekä viestit muilta prosesseilta. Käynnissä olevat prosessit muodostavat puurakenteen, koska jokaisen prosessin on joku toinen prosessi käynnistänyt.

Unixmaisissa järjestelmissä uuden prosessin luominen tehdään fork()-käyttöjärjestelmäkutsulla. Tässä vanhempiprosessista luodaan lähes identtinen kopio. Lapsiprosessin sisältö voidaan korvata lataamalla sen paikalle uusi ohjelma exec()-käyttöjärjestelmäkutsulla.

Prosessilla on yksi tai useampia säikeitä. Kullakin säikeellä on oma konteksti, eli prosessorin rekisterien arvot kullakin suorituksen hetkellä. Yhdessä prosessorissa (t. ytimessä) on kerrallaan muuttuvassa tilassa vain yksi säie ja siten myös kyseisen säikeen konteksti. Muiden säikeiden kontekstit ovat jäädytettyinä käyttöjärjestelmän ylläpitämissä tietorakenteissa. Prosessoriajan jakamisesta prosessien (säikeiden) kesken vastaa käyttöjärjestelmän vuorontaja-osio.

Säikeet mahdollistavat yhden prosessin suorittamisen yhdenaikaisesti useasta eri paikasta, mikä tarkoittaa usean eri kontekstin vuorontamista. Säikeiden voidaan ajatella sisältyvän prosessiin: prosessi määrittelee koodin, yhteisen datan ja muut resurssit, ja säikeet määrittelevät yhdenaikaisia suorituskohtia. Kaikkien säikeiden suorituksissa prosessin resurssit ovat saatavilla, ja ne näkyvät kaikille säikeille samanlaisina.

0.9 Yhdenaikaisuus, prosessien kommunikointi ja synkronointi

Avainsanat : prosessien välinen kommunikointi, signaalit, viestit, jaetut muistialueet, putket, postilaatikko, portti, etäaliohjelmakutsu, kriittinen alue, keskinäinen poissulkeminen, lukkiutuminen (deadlock), semafori, tuottaja-kuluttaja

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa antaa konkreettisen koodi- tai pseudokoodiesimerkin, joka voi johtaa lukkiutumistilanteeseen tai kilpa-ajosta aiheutuvaan datan korruptoitumiseen; vastaavasti osaa kertoa, onko annetussa esimerkkikoodissa mahdollisuus jompaan kumpaan ongelmaan [ydin/arvos2]
- muistaa yksinkertaisimmat yhdenaikaisen suorituksen aiheuttamat haasteet (poissulkua, synkronointia edellyttävät tilanteet) ja osaa ratkaista ne käyttäen (POSIX-) semaforeja [edist/arvos3]
- osaa verifioida tuottaja-kuluttaja -ongelman ratkaisemiseksi ehdotetun esimerkkikoodin oikeellisuuden, eli ratkaiseeko se oikeasti ongelman vai ei. [ydin/arvos2]
- osaa selittää (POSIX-tyyppisen) semaforin käsitteenä, (POSIX-yhteensopivan) alustan tietorakenteena sekä sovellusohjelmalle tarjottavana rajapintana; muistaa semaforikäsitteen historian ja pystyy luettelemaan esimerkkisovelluksia [ydin/arvos2]
- osaa nimetä semaforin lisäksi muita tapoja vastaavanvahvuisen synkronoinnin tuottamiseksi. [ydin/arvos2]

Prosessien pitää voida kommunikoida toisilleen. Olisihan esimerkiksi kiva, jos shellistä tai pääteyhteydestä käsin voisin pyytää jotakin toista prosessia suorittamaan lopputoimet ja sulkeutumaan nätisti. Haluaisin siis, että shell-ohjelmasta voisi kommunikoida vähintään yksinkertaisin viestein mille tahansa aiemmin käynnistämälleni prosessille.

Selvästi myös 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*).

Jotkut mekanismit edellyttävät yhteisten resurssien, yksinkertaisimmillaan muistialueiden käyttämistä, mikä taas johtaa tarpeeseen ohjata näiden resurssien oikea-aikaista ja oikeasisältöistä käyttöä. Tähän lukuun onkin siten nivottu sekä prosessien/säikeiden kommunikointiin että niiden yhteistoiminnan synkronointiin liittyvien käyttöjärjestelmän palveluiden esittely.

Tapoja, joilla prosessit voivat kommunikoida keskenään

Mainitsemme ensin nimeltä muutamia IPC-menetelmiä. Ensimmäisistä kerrotaan sitten hivenen täsmällisemmin. Loput jätetään maininnan tasolle:

- **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. *mailbox*) (prosessi laittaa viestin "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 kutsuvalle prosessille; voidaan tehdä myös verkkoyhteyden yli eli voi kutsua vaikka eri tietokoneessa olevaa aliohjelmaa).

Signaalit

Varmaankin yksinkertaisin prosessien kommunikointimuoto ovat signaalit. Ne ovat asynkronisia ilmoituksia, jotka ilmaisevat pro-

sessille 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 aiemmin rekisteröinyt. Signaali voi tulla myös silloin, kun prosessi on "running" -tilassa (ohjelman omien toimenpiteiden aiheuttamassa virhetilanteessa tai silloin, kun signaalin lähettävää prosessia suoritetaan toisessa prosessoriytimessä samaan aikaan).

Sovellusohjelmoijan pitää itse kirjoittaa ohjelmansa signaalinkäsittelijät sekä käyttöjärjestelmäkutsut, joilla pyydetään rekisteröimään käsittelijät. Signaalinkäsittelijä on normaali aliohjelma, jolla on määrätynlainen rajapinta. Jos ohjelma ei rekisteröi signaalikäsittelijöitä, tapahtuu tiettyjen signaalien kohdalla oletuskäsittely – esimerkiksi signaalina annettu pyyntö ohjelman lopettamiseksi voi oletusarvoisesti vain tylysti lopettaa suorituksen aivan kuten käyttöjärjestelmäkutsu exit(). Mikäli ohjelma käsittelee missään vaiheessa tietoja, jotka olisi tarpeen tallentaa tiedostoon ennen lopetusta, on syytä rekisteröidä lopetussignaalille käsittelijä, joka hoitaa tallentamisen! Toinen hyvä esimerkki on ohjelman tekemien väliaikaisten tiedostojen poistaminen signaalinkäsittelijässä, etteivät ne jää turhaan lojumaan ympäriinsä.

Luennoilla nähdään esimerkkikoodi, miten C-kielessä voi määrittää omat käsittelijät. Vapaaehtoisessa demossa tehdään esimerkkien ja Internet-tutoriaalien avulla shell-skripti, joka poimii signaalin suorittaakseen väliaikaisten tiedostojen poiston.

Prosessit voivat lähettää signaaleja toisilleen määrittelemällä kohdeprosessin PID:n sekä signaalin numeron. Signaaleilla on käyttöjärjestelmätoteutuksessa kiinnitetyt numerot; osa on kiinnitet-

ty ohjelman erilaisia lopetusmenettelyjä varten, osa erilaisten virhetilanteiden käsittelemiseksi, ja muutama on jätetty käyttäjän määriteltäväksi (eli ohjelman tekijä päättää, miten ohjelma vastaa näihin signaaleihin, ja ilmoittaa toiminnan sitten käyttöohjeissa; esimerkki tällaisesta on GNU:n versio apuohjelmasta "dd"[46]).

POSIX määrittelee käyttöjärjestelmäkutsun kill() sekä shell-komennon nimeltä "kill", joilla voi lähettää signaalin PID:llä yksilöidylle prosessille. Brutaalista nimestä huolimatta näillä voi lähettää minkä tahansa signaalin, olkoonkin että joskus joutuu viime hädässä komentamaan "kill -9 1234", joka lähettäisi (tässä esimerkissä prosessille, jonka PID on 1234) "tapposignaalin" KILL, jota ohjelma ei pysty itse poimimaan eikä estämään, vaan käyttöjärjestelmä lopettaa prosessin väkivalloin (ilman että prosessi ehtii tehdä mitään lopputoimia kuten tallentaa muuttuneita tietoja!). Täysin jumittuneelle ohjelmalle tämä voi joskus olla viimeinen vaihtoehto saada se loppumaan. Yleensä kannattaa kuitenkin ensin kokeilla "kill -2" (INT), "kill -15" (TERM) tai "kill -3" (QUIT), koska hyvin tehty sovellusohjelma osaa käsitellä nämä ja hoitaa tarvittavat lopputoimet. Näppäinyhdistelmä "Ctrl-C" päätteellä aiheuttaa signaalin INT. Näiden kolmen erilaisen lopetussignaalin nyansseihin ei mennä tässä enempää. Kaikki ovat kuitenkin turvallisempia kuin hätäratkaisu KILL.

Signaali koodataan kokonaisluvulla, joista POSIX kiinnittää tiettyyn merkitykseen luvut 0, 1, 2, 3, 6, 9, 14 ja 15. Järjestelmä voi tukea muitakin signaaleja, joiden numerot ja merkitykset voi listata shell-komennolla "kill -l" (argumenttina siis viiva ja ällä – ei ykkönen).

[46]Tämä "dd" on POSIXin määrittelemä apuohjelma tiedostosisältöjen kopiointiin erilaisia muunnoksia soveltaen. POSIXkin määrää, että "dd":n tulee käsitellä SIGINT-signaali tietyllä tavoin. GNUn toteutus laajentaa toimintaa poimimalla myös SIGUSR1-signaalin.

Viestit

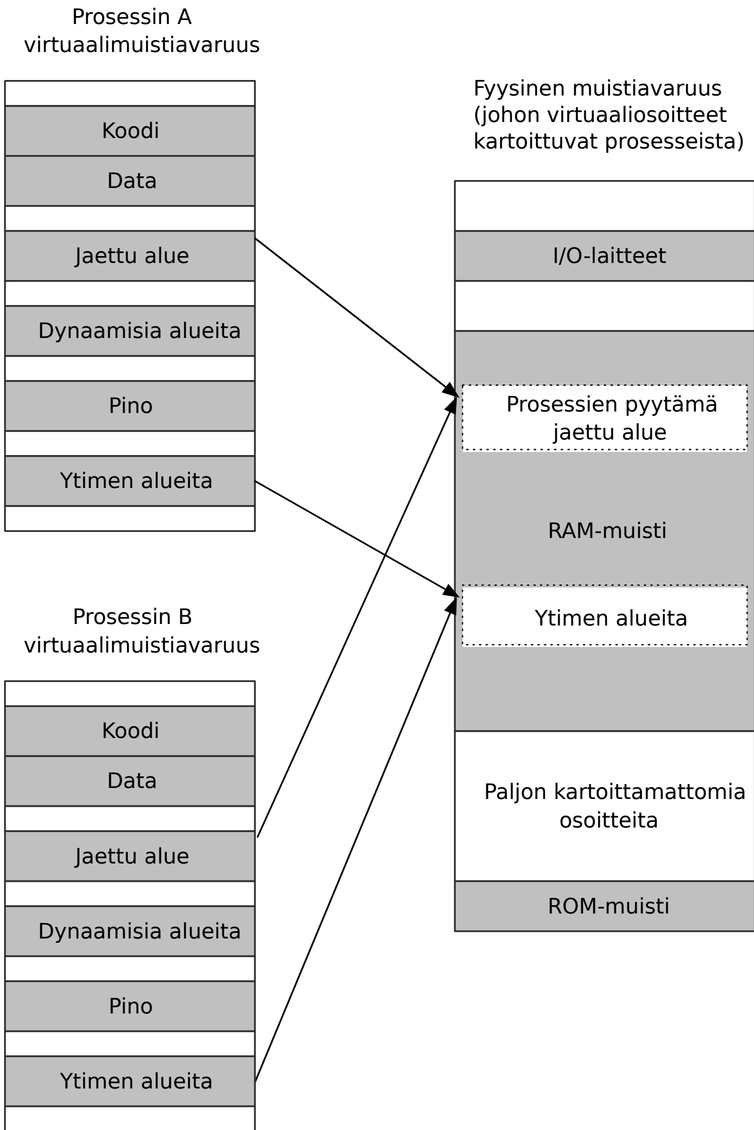
Viestit ovat sovellusohjelmassa määritellyn muotoisia tietorakenteita eli tavujonoja, joita voi lähettää ja vastaanottaa käyttöjärjestelmän hoitamien viestiketjujen kautta. Järjestelmä voi määritellä viesteille jonkin maksimipituuden, mutta muuten niiden sisältämä data on sovellusohjelman päätettävissä. Tällaisiin ketjuihin pääsee käsiksi käyttöjärjestelmäkutsuilla, joiden nimiä voisivat olla esim seuraavat, jotka POSIX määrittelee (tarkemmin ottaen POSIXin ns. XSI-laajennoksen IPC-osio):

- 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 esiteltävällä semaforilla. Viestijonoista on yksi luentoesimerkki, jolla voidaan todeta, että tällaisia välineitä voi luoda mm. Linuxilla kääntyvässä C-ohjelmassa, ja että huolimaton käyttö voi johtaa samanlaisiin ongelmiin kuin muutkin ”lukitukset”. Oikeasti on sitten syytä opiskella standardin määrittelyt sekä jokin tutoriaali viestijonojen oikeelliseen käyttöön, joten tarkempi tutustuminen jätetään myöhemmin elämässä, tarvittaessa, tehtäväksi.

Jaetut muistialueet

Prosessit voivat pyytää käyttöjärjestelmää kartoittamaan fyysisen muistialueen kahden tai useamman prosessin virtuaalimuistin



Kuva 0.25: *Muistialueita voidaan jakaa prosessien välillä niiden omasta pyynnöstä tai oletusarvoisesti (käyttöjärjestelmän alueet sekä dynaamisesti linkitettävät, jaetut kirjastot).*

osaksi. Näin muistialueesta tulee prosessien jakama resurssi, johon ne molemmat voivat kirjoittaa ja josta ne voivat lukea. Tätä havainnollistetaan kuvassa 0.25. 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.

Jaettu muistialue on tehokas tapa kommunikoida mielivaltaisen muotoista dataa. Kun yhteinen fyysisen muistin alue on kartoitettu kahden tai useamman prosessin virtuaalimuistiin, se voi tietysti sijaita aivan eri osoitteessa näiden prosessien omissa virtuaalimuistiavaruuksissa. Sisältö on kuitenkin yhtä ja samaa muistia, joten osoitteiston lisäksi yhteisen muistin käyttö on hyvin samanlaista kuin säikeiden jakamassa yhden prosessin virtuaalimuistiavaruudessa. Luentoesimerkit ja tämän luvun loppuosa esitteleekin jaetun muistin käyttöön liittyviä yleisiä ongelmia ja ratkaisuja tässä hieman yksinkertaisemmassa skenaariossa, jossa koko virtuaalimuistiavaruus on jaettu säikeiden kesken. Samat asiat tulee kuitenkin muistaa myös aina, kun prosessien välillä jaetaan jokin pienempi muistialue[47].

Synkronointi: kilpajuoksu ja poissulku/lukitseminen

Mitä tarkoittaa **synkronointi** (engl. *synchronization*)? Arkikielessä esimerkiksi ”tapahtuma-aikojen ohjaamista siten, että kokonaistehtävä onnistuu”. Myös tietokoneiden prosessien ja säikeiden

[47]POSIXin määrittelemiä käyttöjärjestelmäkutsuja jaetun muistin käyttöön prosessien välillä ovat “shm_open()“ jaetun muistialueen luomiseen tai aiemmin luodun käyttöönottoon. Muisti näyttäytyy POSIXissa tiedostona, joten sen kokoa ja muistiin kartoittumista voi kontrolloida tiedostoja käsittelevien kutsujen avulla – esimerkiksi mmap() tekee varsinaisen kartoittamisen virtuaalimuistiavaruuteen.

yhteydessä tarvitaan synkronointia aina, jos ohjelmien toiminta tai vaikutukset voivat riippua suoritusjärjestyksestä. Yksittäinen suoraviivaista yhden säikeen jälkeä noudattava prosessi, joka elää yksinomaan omien resurssiensa ja muistiavaruutensa sisällä, ei tarvitse synkronointa. Tyypillisempää kuitenkin on, että ohjelmat ja niiden osiot käyttävät yhteisiä resursseja. Heti, jos ohjelma esimerkiksi lukee tai kirjoittaa tiedostoja, on mietittävä, miten muut moniajojärjestelmän prosessit mahdollisesti voivat käyttää näitä samoja tiedostoja.

Seuraava luennoillakin nähtävä esimerkki on ehkä yksinkertaisin esimerkki **kilpajuoksusta** tai **kilpailutilanteesta** (engl. *race condition*) yhteisen resurssin äärelle. Tässä on pseudokoodina olennainen osa esimerkistä (HUOM: katso myös toimivat POSIXin säikeitä käyttävät esimerkit kurssin materiaalirepositoriosta):

```
#define N 100000000
uint64_t summa = 0;

saikeen_koodi() {
  for (int i = 1; i <= N; i++) {
    summa++;
  }
}

main() {
  saie saieA, saieB;

  aloita_saie(&saieA, saikeen_koodi);
  aloita_saie(&saieB, saikeen_koodi);

  odota_saie_valmiiksi(saieA);
  odota_saie_valmiiksi(saieB);

  tulosta("Summa on %d\n", summa);
}
```

Tässä koeputkiesimerkissä tehdään kaksi säiettä, jotka molemmat

lisäävät sata miljoonaa kertaa ykkösen muuttujaan `summa`, joka on molemmille yhteisessä muistissa, eli siinä mielessä 'jaettu resurssi'. Varsin todennäköistä on, että tulos ei kuitenkaan ole 200 000 000, kuten sen haluttaisiin olevan. Sen sijaan ohjelman laskema summa on jotakin pienempää ja jopa satunnaista, eli ohjelman lopputulema on hallitsematon. Tällaisia ohjelmia harvoin halutaan tehdä! Syy käytökseen on nyt mahdollista ymmärtää syvällisesti kurssilla aiemmin kerätyn laitteistoymmärryksen pohjalta. Kurssin esimerkkinä oleva C99-kääntäjä on tuottanut AMD64-prosessorille rivistä `summa++` seuraavan käännöksen, joka tässä nähdään gdb:n disassembly-toiminnon tulosteena:

```
24                  summa++;
   0x00000000004006b7 <+17>:        mov    0x200992(%rip),%rax         # 0x601050
   0x00000000004006be <+24>:        add    $0x1,%rax
   0x00000000004006c2 <+28>:        mov    %rax,0x200987(%rip)         # 0x601050
```

Niinkin viaton rivi kuin `summa++` kääntyy kolmeksi peräkkäiseksi konekielikäskyksi, joista ensimmäinen siirtää muuttujan nykyisen arvon muistista rekisteriin `RAX`, toinen lisää ykkösen, ja kolmas siirtää `RAX`:n arvon takaisin muistiin. Moniajojärjestelmässä *minkä tahansa käskyn suorituksen jälkeen voi tulla keskeytys ja kontekstin vaihto*. Tässä keskeytys voi helposti tulla ennen kuin tulos on tallennettu rekisteristä muistiin. Eli esimerkiksi yksi säie ehtii kasvattaa summaan 10000 kertaa, mutta sen jälkeen suoritukseen pääsee luonnostaan toinen säie, joka tallentaa muuttujaan oman versionsa, jossa on luultavasti paljon pienempi luku. Kyseessä on kilpa-ajo siinä mielessä, että joku aina ehtii ensiksi käyttämään resurssia, olipa toiset valmiita sen kanssa tai eivät.

Sovellus ei voi estää keskeytyksiä, mutta se voi kyllä vaatia itselleen yksinoikeuden suorittaa muiden säikeiden häiritsemättä tietty koodipätkä, jota sanotaan **kriittiseksi alueeksi** (engl. *critical re-*

gion). Koska asian osittain aiheuttaa prosessorin keskeytysominaisuus, ei tällaiseen oikein voi saada apuja muuten kuin käyttöjärjestelmän kautta. Käyttöjärjestelmäkutsun suorituksen aluksihan uudet keskeytykset ovat kiellettyjä, joten kutsussa voidaan kyllä sommitella uudelleen seuraavien säikeiden vuoronnusta ennen kuin prosessorin sallitaan jälleen tehdä suoritussyklin keskeytyskäsittelyvaihe. Perusidea on, että jokainen kriittiselle alueelle pyrkivä säie/prosessi pyytää käyttöjärjestelmältä lupaa edetä. Jos lupa on jo myönnetty jollekin muulle säikeelle, niin kaikki myöhemmät säikeet blokataan jonottamaan (luonnollisesti käyttöjärjestelmän ja mahdollisesti alustakirjaston hallitsemassa jonomaisessa tietorakenteessa, joka sisältää vähintään jonottavien säikeiden/prosessien ID:t), kunnes nykyinen säie saa kriittisen alueensa suorituksen loppuun ja ilmoittaa tästä käyttöjärjestelmälle. Seuraava odotusjonossa ollut säie saa sen jälkeen vastaavan yksinoikeuden, ja pääsee etenemään omalle kriittiselle alueelleen sitten, kun se luonnostaan saa taas suoritusvuoron. Sekä luvan pyytäminen että kriittisen alueen loppumisen ilmoittaminen on sovellusohjelman tekijän vastuulla!

Edellisen, käyttötarkoitukseltaan erityisen järjettömän, ohjelman tapauksessa voitaisiin ajatella, että kriittinen alue ovat nuo kolme konekielistä käskyä eli yksi C-kielinen rivi, `summa++;`. Kriittinen alue voidaan helpoiten suojata menettelyllä nimeltä **keskinäinen poissulku**(engl. *mutual exclusion, "MutEx"*) :

```
#define N 100000000
uint64_t summa = 0;
lukko summalukko;

saikeen_koodi() {
  for (int i = 1; i <= N; i++) {
    lukitse(summalukko);
    summa++;
    avaa(summalukko);
```

```
  }
}

main() {
  saie saieA, saieB;

  aloita_saie(&saieA, saikeen_koodi);
  aloita_saie(&saieB, saikeen_koodi);

  odota_saie_valmiiksi(saieA);
  odota_saie_valmiiksi(saieB);

  tulosta("Summa on %d\n", summa);
}
```

Pseudokoodissa aliohjelmat `lukitse()` ja `avaa()` vastaavat alustakirjaston kutsuja, joiden täytyy jossain vaiheessa johtaa sopiviin käyttöjärjestelmäkutsuihin. Sovellusohjelman osalta ne toimivat siten, että lukitsemisen jälkeen sovellus voi luottaa siihen, että kukaan muu ei pääse vastaavasta koodirivistä eteenpäin ennen kuin omassa koodissa on tehty lukon avaaminen. Tässä mallissa sovellus ei kuitenkaan voi esim. tietää, joutuuko se ensin hetkeksi odottelemaan ennen kuin suoritus jatkuu kriittisen alueen sisälle.

Kuten luentoesimerkissä kokeilemalla nähdään (ks. oikea POSIXin kutsuilla korjattu versio), niin tässä tapauksessa suoritusaika räjähtää täysin käsiin, koska kaikki aika kuluu keskinäisen poissulun tarvitsemiin lukituksiin. Lukituksia kannattaakin oikeasti käyttää vain harvoissa ja valituissa paikoissa, mutta kuitenkin *aina tarvittaessa*. Tästäkin asiasta kannattaa nähdä useampia hyviä unia, ennen seuraavaa ohjelmointiprojektia, jossa käytät useita säikeitä tai prosesseja samojen resurssien käsittelyyn!

POSIX määrittelee yksinkertaiseen lukitukseen kaksi kutsua:

- pthread_mutex_lock()

- pthread_mutex_unlock().

Seuraavassa luvussa käsitellään monipuolisempi synkronointitapa, joka pystyy erityistapauksena hoitamaan myös lukituksen, mutta joka lisäksi pystyy paljon monipuolisempaankin synkronointiin.

Deadlock

Resurssien lukitseminen voi johtaa vaaranpaikkoihin ja vaikeasti havaittaviin toimintahäiriöihin. Yhden resurssin yksittäinen poissulku ei vielä ole oikein helppo rikkoa, kunhan muistaa aina vapauttaa lukon kriittisen alueen lopuksi. Kahdella resurssilla vaara on jo suurempi, koska yksittäinen prosessi voi näyttää aivan oikein tehdyltä, mutta yhdessä toisen, myös itsenäisesti oikein tehdyn, prosessin kanssa saattaa yhteistoiminnasta syntyäkin satunnaisissa tilanteissa esimerkiksi seuraavassa esimerkissä arkihavainnon kautta esiteltävä ongelma.

Kurssimateriaalirepositoriosta löytyy sama esimerkki (siellä tosin "vappusimulaattorina") sekä rikkinäisenä että korjattuna versiona, oikeana C99-koodina POSIX-säikeillä ja POSIXin MutExilla toteutettuna.

Kuvaelma nimeltä "Ruokailevat nörtit"(triviaali johdantoesimerkki):

Essi, Jopi ja Ossi leikkivät kotista. Nörttejä kun ovat, niin heidän leikkinsä kulkee kuin tietokoneen prosessorin toiminta. Essi ja Jopi istuvat pöydässä ja 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. Post(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. Post(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.

Semafori

Semafori (engl. *semaphore*) on käyttöjärjestelmän ja alustakirjaston yhteistyössä käsittelemä tietorakenne, jonka avulla voidaan hallita vuorontamista eli sitä, milloin prosessit/säikeet pääsevät suoritukseen prosessorilaitteelle. Käsittelemme seuraavaksi POSIXin mukaisen semaforin, vaikka muitakin malleja on kirjallisuudessa esitetty.

Semaforin rakenne (vahvasti POSIXia mukaillen)

Yhdessä POSIXin määrittelemässä semaforissa on sisältönä arvo (”value”) ja joukko säikeitä, jotka odottavat semaforilla hallittavan resurssin vapautumista. Arvo on aina ei-negatiivinen kokonaisluku, ja säiejoukko voi olla epätyhjä vain silloin kun semaforin arvo on 0. Käytännössä säikeet ovat tallessa esim. jonomaisessa tietorakenteessa[48]. Esimerkiksi semaforin tilanne voisi käytännössä olla seuraavanlainen:

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

Silloin semafori on lukittuna (Arvo==0), ja kaikki myöhemmät pyytäjät ovat joutuneet blocked-tilaan odottelemaan, että nykyinen lukon hallitsija vapauttaa resurssin. Toinen mahdollinen tilanne:

```
Arvo:  5
Jono:  NULL
```

Tällöin semafori ei ole lukittuna (Arvo>0), joten jonossa ei tietysti ole yhtään säiettä. Semaforin arvosta, joka nyt on 5, tietää, että seuraavat 5 pyytäjää saavat resurssin käyttöönsä ja vasta sen jälkeen semafori lukittuu.

[48]Tarkkaan ottaen POSIX nimenomaisesti puhuu ”joukosta” eikä ota kantaa järjestykseen, jossa odottelevasta joukosta valitaan suoritukseen seuraava

Semaforit pitää voida yksilöidä, koska niistä jokaisella halutaan hallita tiettyä resurssia tai resurssin ominaisuutta. Niitä voi luoda tarpeen mukaan alustakirjaston kutsujen kautta. Semaforien luonnin ja yleisen hallinnan lisäksi alustakirjasto toteuttaa seuraavanlaisen pseudokoodin mukaiset kutsut semaforin soveltamiseksi; niiden nimet voisivat olla esimerkiksi `sem_wait()` ja `sem_post()`, kuten POSIXissa (C-otsikkotiedostossa '`semaphore.h`'), mutta yhtä hyvin jotakin muuta vastaavaa... Alkuperäiset nimet ovat Edsger Dijkstran hollannin kielestä johdetut lyhenteet `P()` ja `V()`. Kutsut ovat sovellusohjelmassa, ja niiden parametrina on annettava yksi tietty, aiemmin luotu, semaforitietorakenne. Alustakirjaston tulee käyttöjärjestelmän avustuksella tehdä seuraavaa ennen kuin sovellusohjelman suoritus saa jatkua siinä olleen vastaavan aliohjelmakutsun perään (pseudokoodi):

```
sem_wait(Sem){
   if (Sem.Arvo > 0)
      Sem.Arvo--;
   else /* eli silloin kun Sem.Arvo == 0 */
      Laita pyytäjäprosessi blocked-tilaan ja tämän semaforin jonoon.
}

sem_post(Sem){
  if (Jono on tyhjä)
      Sem.Arvo++;
  else
      Ota jonosta seuraava odotteleva prosessi suoritukseen.
}
```

Sovellusohjelmaan täytyy kirjoittaa kutsut sopivalla tavoin. Lisäksi semaforien alkuarvot täytyy asettaa tarkoituksenmukaisesti, ennen kuin mikään säie pääsee niitä käyttelemään.

Tuottaja-kuluttaja on ehkä yksinkertaisimpia esimerkkejä, joissa tällaista moniarvoista semaforia voidaan käyttää synkronointiin. Palataan siihen kuitenkin, vasta kun on katsottu, kuinka sinän-

sä yksinkertaisempi keskinäinen poissulkukin saadaan hoitumaan yleispätevällä semaforilla.

Esimerkki: Poissulkeminen (eli MutEx) semaforilla

Edellä esitetty yksinkertainen poissulkulukko voidaan toteuttaa semaforilla, jonka alkuarvoksi asetetaan 1. Katsotaan seuraavaa sovellusohjelmaa:

```
// Alustus ennen kuin varsinaiset tehtävät alkaa:
luo_semafori(semMunMutexi)
aseta_semaforin_arvo(semMunMutexi,1)

// Myöhemmässä vaiheessa tarvittava lukitus:
sem_wait(semMunMutexi) // "atominen käsittely" tälle koodiriville
                       // eli muut käyttäjän prosessit ovat
                       // keskeytettynä kunnes semafori on hoideltu.

/* ... kriittinen alue, yksinoikeus tällä prosessilla ... */

sem_post(semMunMutexi) // jälleen "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/säikeiden tiedossa. Alkutilanteessa semafori on vapaa eli sillä on "yksi kappale resurssia vapaana":

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

PID 77:n koodia suoritetaan, siellä on kutsu sem_wait(semMunMutex
Tapahtuu ohjelmallinen keskeytys, jolloin prosessi PID 77 siirtyy kernel running -tilaan, ja käyttöjärjestelmä pystyy varmistamaan, että muut prosessit eivät häiriköi ennen kuin semaforin käsittely

sem_wait() on tehty yllä olevan pseudokoodin mukaisesti. 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 semMunMutexisemaforilla 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. PID 898:n koodi lähestyisi kriittistä aluetta, jossa sekin kutsuisi sem_wait(semMunMutexi). Jälleen tietenkin tulisi ohjelmallinen keskeytys, prosessi PID 898 menisi kernel running -tilaan, ja alustakirjaston koodista suoritettaisiin kenenkään muun häiritsemättä 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 `sem_post()`-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. Atomisen toimenpiteen

aloittaminen saattaa hetkellisesti keskeyttää kaikkien rinnakkaisten prosessorien suorituksen, koska eihän voida tietää, vaikka toisessa ytimessä olisi ohjelma juuri seuraavaksi haluamassa kajota samaan semaforiin! Tästä syystä usein tehtävät synkronointipyynnöt voivat hidastaa moniydinjärjestelmää merkittävästi verrattuna tilanteeseen, jossa ytimet voivat jatkaa keskeytyksettä ilman muiden ytimien vaatimia tarkistuspisteitä[49].

Useakin prosessi saattaisi halua käsitellä semMunMutexilla suojattua jaettua resurssia. Vuorontaja jakelisi prosesseille aikaa, ja toimenpidepyynnöt tapahtuisivat satunnaisiin aikoihin, nykyprosessorissa hyvinkin nopeaan tahtiin. Semafori kuitenkin on jo lukinnut alueen ensimmäiseksi ehtineen prosessin käyttöön, joten seuraavat pyytäjät joutuvat jonon hännille. Mitään ihmeellistä ei tapahdu – semafori toimii jokaisen pyynnön kohdalla edellä esitetyn pseudokoodin mukaisesti. Avain on atominen toiminta, jossa semaforikäsittely ei keskeydy samassa tai eri prosessoriytimessä meneillään olevien prosessien toimesta. Jossain vaiheessa tilanne voisi 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(sem ensimmäisenä, saa lopulta operaationsa valmiiksi jollakin ajovuorollaan, ja jos se on oikeellisesti ohjelmoitu, niin kriittisen alueen lopussa on kutsu `sem_post(semMunMutexi)`. Jälleen käyttöjärjestelmän avustama alustakirjasto hoitaa tilanteeksi atomisesti:

[49]Moniydinprosessorin prosessoriarkkitehtuurissa täytyy olla saatavilla konekielikäsky, joka pysäyttää muiden ytimien toiminnan siksi aikaa, että käyttöjärjestelmä voi tarkistaa, tarvitaanko jotakin tiedonsiirtoa ytimien välillä ennen kuin rinnakkaiset suoritukset jatkuvat. AMD64:ssä näitä käskyjä on muutamia. Prosessorimanuaalit kertovat totuuden näistä. Ominaisuudet ovat. . . sanotaanko vaikka, että “kohtalaisen monipuoliset”.

```
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 `sem_wait()` -kutsun kautta tullut, ja aiemmat prosessit ovat yksi kerrallaan suorittaneet kriittisen alueensa ja kutsuneet `sem_post()`, niin aivan viimeinen `sem_post()` tapahtuu tietysti seuraavanlaisessa tilanteessa:

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

Silloin viimeisen post()-kutsun toiminta vapauttaa resurssin täysin, ja tilanne on sama kuin aivan esimerkin alussa:

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

Tällä tavoin mikä tahansa määrä yhdenaikaisia yrittäjiä pääsee jonotuskäytännön kautta omalla vuorollaan suorittamaan semaforilla rajattua kriittistä koodialuetta yksinoikeudella.

Laajempi esimerkki: tuottaja-kuluttaja ratkaistuna semaforeilla

Jaetun resurssin käyttö johtaa helposti erilaisiin ongelmatilanteisiin, jotka on jollain tavoin ratkaistava. Käytetään tässä esimerkkinä yksinkertaista, perinteistä ongelmaa, joka voi syntyä käytännön sovelluksissa ja jonka avulla voi testata synkronointimenetelmän toimivuutta.

Tuottaja-kuluttaja -probleemi

Tuottaja-kuluttaja -ongelma (engl. *producer-consumer problem*) syntyy, kun kahden prosessin tai säikeen välillä tarvitaan yksisuuntaisen tietovirran synkronointia. Varmistutaan nyt ensin ajatusesimerkin kautta, että meillä ylipäätään on tässä jokin ongelma ratkaistavana. Tilanne on seuraavanlainen:

- 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. Myös tämä voi olla hidas tai nopea toimenpide, erityisesti kuluttaminen voi olla paljon hitaampaa tai nopeampaa kuin tuottaminen, tai keskinäinen nopeus voi vaihdella datan sisällöstä riippuen.
- Hyötyjä: Tällä tavoin saavutetaan mm. modulaarisuutta ohjelmien tekemiseen, jakeluun ja suorittamiseen. Tuotettua dataa ei välttämättä tarvitse tallentaa pysyvästi, mikä vähentää tallennustilan tarvetta, jos se kulutetaan yhdenaikaisesti. Moniydinjärjestelmässä tuottaminen ja kuluttaminen

voivat myös tapahtua eri prosessoreilla, jolloin kokonaissuoritusaika voi lyhentyä.

- Puolirealistinen esimerkki: yksi prosessi/säie tuottaa kuvasarjaa fysiikkasimuloinnin perusteella (tuottamisen nopeus voi vaihdella esimerkiksi animaatiossa näkyvissä olevien 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). Pakkaamattomia kuvia ei edes haluta tallentaa pysyvästi, vaan talteen halutaan tässä ”kuluttajan” tuottama, paljon pienempään tilaan mahtuva, MP4-video.

- 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.

Jos tuottaja ja kuluttaja vain tekisivät toimenpiteitään ilman keskinäistä synkronointia, videokuvaesimerkin lopputuloksessa olisi oletettavasti puoliksi valmiita ruutuja, jotka ”repeävät” sattumanvaraisesta paikasta: esim. yläosa on uutta, mutta alaosa vielä jotakin vanhaa kuvaa. Jos kuluttaja on paljon nopeampi kuin tuottaja, saattaisi videossa toistua siirtopuskurin kuvat uudelleen ja

uudelleen eli tuloksena olisi jonkinlainen kummallinen ”nykivä hidastuskuva”. Jos taas tuottaja on paljon nopampi, lopputuloksesta jäisi pätkiä pois, koska kuluttaja ei ole ehtinyt niitä käsitellä, kun tuottaja olisi jo tuottanut uusia kuvia puskuriin. Keskinäisen nopeuden vaihdellessa nykivät hidastukset ja pätkien poisjäännit vaihtelisivat sattumanvaraisesti tai animaation sisällön mukaan.

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. → Tarvitaan keskinäinen poissulku vähintään yhteisen tilannetiedon päivitykseen.

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). Jos taas synkronointia tarvitaan saman prosessin säikeiden välille, toki kaikki muisti on jaettua säikeiden kesken, ja osoitteetkin ovat tällöin samat. Säikeetkin tarvitsevat kuitenkin omat indeksinsä, että ne tietävät nykyisen alkionsa paikan rengaspuskurissa.

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 uudenlaisia ongelmia, jotka on ratkaistava, että ohjelma toimisi joka tilanteessa oikeellisesti. Myös ratkaisutapoja on muitakin kuin seuraavaksi esiteltävät semaforit. Yksinkertaisuuden vuoksi Käyttöjärjestelmät -kurssilla käydään läpi vain yksi yksinkertainen ongelmatapaus ja yksi yksinkertainen ratkaisu siihen.

Tuottaja-kuluttaja -probleemin ratkaisu semaforeilla

Tuottaja-kuluttaja -ongelma voidaan ratkaista semaforeilla seuraavaksi esitetyllä tavalla. Tarvitaan kolme semaforia eri merkityksiin, joilla purraan edellä esitettyihin tavoitteisiin:

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

Ohjelmoijan on muistettava hoitaa näiden oikeellinen käyttö. Ennen tuottamista ja kuluttamista sovelluksen on alustettava semaforit seuraavasti:

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

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

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

Mietitäänpä hetki, miksi tässä tarvitaan kolme tällä tavoin alustettua semaforia. MUTEX-semafori on selvä, koska tehtävä edellyttää poissulkua yhteisiä muuttujia ja dataelementtiä käsiteltäessä. EMPTY tarvitaan, jotta tuottajalle voidaan tiedottaa, miten monta tyhjää elementtipaikkaa ('resurssia') sillä on tässä vaiheessa käytettävissä. Vastaavasti FULL tarvitaan, jotta kuluttajalle voidaan tiedottaa, miten monta täytettyä ja toistaiseksi kuluttamatonta elementtipaikkaa ('resurssia') sille on tässä vaiheessa tarjolla. Molempien täytyy pystyä odottamaan, mikäli niille tarkoitettua resurssia ei ole hetkellisesti tarjolla.

Tuottajan idea pseudokoodina:

```
WHILE(1){  // tuotetaan loputtomiin
   tuota()
   sem_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.

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

   sem_post(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
                   // sem_post(FULL) monta kertaa, ja FULL.Arvo voi olla
                   // mitä vaan >= 0 siinä vaiheessa, kun kuluttaja
                   // pääsee apajille.
 }
```

Kuluttajan idea:

```
WHILE(1){
   sem_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 sem_post(FULL)

   sem_wait(MUTEX) // tämä taas selvä jo edellisestä esimerkistä.
   Hae tietoalkio puskurista itselle
   sem_post(MUTEX)

   sem_post(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.
    kuluta()
  }
```

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. Tässä nähtiin 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.

Yhteenveto: Prosesseilla on tarvetta kommunikoida toisilleen esimerkiksi, kun eteen tulee tarve keskeyttää prosessin suoritus, tarjota tietoa muiden prosessien käyttöön tai käyttää toisen prosessin sisältämää palvelua ("etäaliohjelmaa").

Jaettua resurssia käyttävät prosessit eivät saisi päästä tekemään ristiriitaisia luku- ja kirjoitusoperaatioita samanaikaisesti. Sellaista osaa koodista, joka voi aiheuttaa ristiriitoja (eli ns. kriittistä aluetta) saa päästä suorittamaan vain yksi prosessi kerrallaan. Täytyy siis tapahtua prosessien keskinäinen poissulkeminen, MutEx. Hieman monipuolisempi esimerkki kahden prosessin välisen tietovirran synkronoinnista on tuottaja-kuluttaja-ongelma. Nämä ongelmat voidaan ratkaista muun muassa käyttämällä semaforeja, jotka hallinnoivat vuorontamista.

Prosessien yhteistoiminta voi aiheuttaa piilevää, satunnaisesti ilmenevää virhekäyttäytymistä, mikäli synkronointimenettely ei ole loppuun asti suunniteltu (esimerkiksi deadlock). Näiden ongelmien automaattinen havaitseminen ja välttely on jatkuvan tutkimuksen kohde.

0.10 Muistinhallinta

Avainsanat: sivuttava virtuaalimuisti, lokaalisuus, sivu, kehys, työjoukko, sivutaulu, kehystaulu, sivunvaihtokeskeytys (eli sivuvirhe), least-recently-used

Osaamistavoitteet: Esitiedot sekä tämän luvun luettuaan opiskelija:

- osaa selittää, mikä on tietokoneiden muistihierarkia ja siihen liittyvät kompromissit [ydin/arvos1]
- tietää, mikä on lokaalisuusperiaate, miten sitä hyödynnetään muistijärjestelmässä sekä kuinka periaate on yleistettävissä muihin sovelluksiin [ydin/arvos2]
- osaa selittää sivuttavan virtuaalimuistin toimintaperiaatteen ja kuvailla sen toteuttamiseksi tarvittavat käyttöjärjestelmän tietorakenteet sekä ohjelmiston ja laitteiston osat [ydin-/arvos2]
- osaa muuntaa virtuaalimuistiosoitteen fyysiseksi annetun (yksitasoisen) sivutaulun perusteella; ymmärtää sekä osaa selittää jaetun muistialueen toteuttamisen virtuaalisia muistiosoitteita käyttäen [edist/arvos3]
- osaa kuvailla yksityiskohtaisesti sivunvaihtokeskeytyksen kulun LRU-menettelyssä [edist/arvos4]
- tietää cache thrashing -ilmiön ja tunnistaa mahdollisuuden sellaisen olemassaoloon yksinkertaisessa koodiesimerkissä; osaa kuvailla keinoja, joilla ilmiöltä voidaan yrittää välttyä; osaa valita kahdesta konkreettisesta koodiesimerkistä sen, jossa ilmiö on vähemmän paha [edist/arvos4]

Tässä luvussa tarkennetaan, kuinka muistin käyttöä ohjaava käyttöjärjestelmän osio, nimeltään **muistinhallinta** (engl. *memory management*), toimii yhteistyössä tyypillisen nykyaikaisen laitteiston kanssa aiemmissa luvuissa havaittujen tarpeiden toteuttamiseksi.

Virtuaalimuistin ja muistinhallinnan tavoitteet

Kerätään ensin yhteen, mitä havaintoja tähän mennessä on tehty muistin ja sen hallitsemisen osalta. Luvusta 0.2 alkaneen laitteistokuvauksen perusteella lienee käynyt selväksi, että ensinnäkin prosessien ja kirjastojen ohjelmakoodin sekä niiden käsittelemän datan täytyy sijaita konekielen suorittamisen aikana tietokoneen muistissa, koska vain muutama laskennan välitulos mahtuu kerrallaan prosessorin sisäisiin rekistereihin. Laitteisto sisältää toimintaa tehostavia välimuisteja ja muita tekniikoita, joiden toiminta on automaattista ja suurimmalta osin piilossa laitteistorajapinnan eli konekielen takana. Luvussa 0.5 todettiin, että yksittäisen sovellusohjelman kannalta muisti näyttää yhtenäiseltä tallennuspaikkojen sarjalta, jonka jokainen, tyypillisesti yhden tavun kokoinen, paikka on numeroitu peräkkäisellä muistiosoitteella. Osoitteet alkavat nollasta ja mahdollisia osoitteita voi olla niin monta kuin prosessorivalmistajan määrittämän osoiteleveyden bittimäärä sallii. Luvussa 0.5 käytiin myös läpi tyypillinen tapa jakaa tämä prosessin näkemä muisti alueisiin eli segmentteihin (koodi, data, keko, pino), joilla on kullakin oma roolinsa ohjelman suorituksessa. Segmenteille halutaan turvallisuussyistä erilaiset käyttöoikeudet: esimerkiksi data-alueen tai pinon sisällön suorittaminen koodina on hyvä estää teknisesti, samoin kuin koodialueen muuttaminen ohjelman ajon aikana. Prosessit eivät saa vahingossa kirjoittaa eivätkä edes lukea muistipaikkaa, mikäli niille ei ole annettu siihen lupaa.

Prosessin muistiosoitteisto on **virtuaalimuistiavaruus**, johon käyt-

töjärjestelmä sijoittelee ohjelman tarvitsemat segmentit yksilöllisesti jokaisen prosessin lataamisen yhteydessä sekä tarvittaessa ajon aikana tapahtuvien käyttöjärjestelmäkutsujen yhteydessä. Prosessori muuntaa jokaiseen suorittamaansa konekielikäskyyn liittyvät virtuaaliosoitteet automaattisesti fyysisiksi osoitteiksi, jotka vastaavat todellisen tietokonelaitteiston osoitteita eli fyysistä muistiavaruutta. Tämän se voi tehdä käyttöjärjestelmän luoman, prosessikohtaisen, muistikartan perusteella.

Näin hoidettu muistin organisointi näyttää sovellusohjelman näkökulmasta selkeältä, eikä ole vaaraa, että prosessit tyystin vahingossa pääsisivät käsittelemään toistensa muistia. Toisen prosessin fyysiset osoitteet eivät kertakaikkiaan ole saavutettavissa yhtä prosessia suoritettaessa, ellei niitä ole tarkoituksellisesti kartoitettu useamman prosessin virtuaalimuistiavaruuteen. Tarvittaessa prosessit voivat kuitenkin helposti käyttää samaa kohtaa fyysisestä muistista, koska virtuaaliosoitteet voidaan kartoittaa samoihin fyysisiin muistipaikkoihin, kuten luvussa 0.9 nähtiin. Prosessit voivat siis siirtää toisilleen suuria määriä dataa ilman ylimääräisiä kopiointeja: kun yksi prosessi kirjoittaa jaettuun muistialueeseen, on muutos samalla valmis toisenkin prosessin käyttöön. Yhteiset kirjasto-ohjelmistot on taloudellista pitää fyysisessä muistissa vain yhtenä kappaleena ja kartoittaa ne kaikkien niitä käyttävien prosessien virtuaalimuistiavaruuksiin. Käyttöjärjestelmäkutsun suorittaminen saadaan kevyeksi toimenpiteeksi, kun käyttöjärjestelmän muisti kartoitetaan jokaisen prosessin virtuaalimuistiin, vieläpä keskenään identtisiin muistiosoitteisiin.

Luvussa 0.2 puhuttiin muistihierarkiasta. Osoittautuu, että virtuaalimuisti on avain myös massamuistin, kuten kovalevyn, hyödyntämiseen fyysisen muistin ”jatkeena” näppärällä tavalla. Lokaalisuusperiaatteen toteutuminen mahdollistaa nimittäin käyttämättömänä lojuvien tietojen väliaikaisen säilyttämisen massamuistis-

sa, jolloin fyysistä keskusmuistia voidaankin käyttää ”välimuistin” roolissa ja kokonaisjärjestelmän muistikapasiteettia voidaan kasvattaa hitaan, mutta suuren massamuistin puolelle. Sovellusohjelmien ei tarvitse tässäkään tapauksessa nähdä muuta kuin oma virtuaalimuistiavaruutensa.

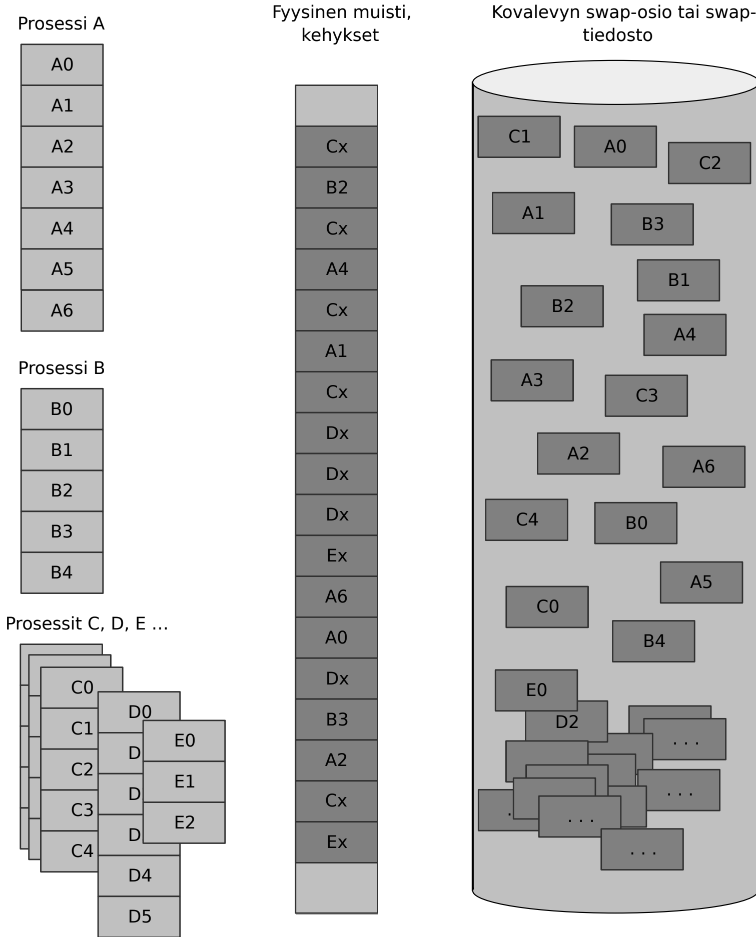
Muistettaneen myös, että nykyiset tavoitteet ja ominaisuudet ovat syntyneet vaiheittain tietotekniikan historian aikana havaittujen puutteiden korjaamiseksi ja laitteistoresurssien käytön helpottamiseksi, kuten luvussa 0.6 kuvailtiin.

Sivuttava virtuaalimuisti

Jotta virtuaalimuistia voidaan käyttää, tarvitaan tietyt ominaisuudet prosessorilta ja sitä ohjaavalta käyttöjärjestelmältä. Nykyisissä prosessoreissa on tyypillisesti erillinen **muistinhallintayksikkö** (engl. *memory management unit, MMU*) välimuisteihin ja virtuaalimuistiin liittyviä toimenpiteitä ja niiden ohjausta varten. Ulospäin näkyvän MMU:n lisäksi muistin käyttöä tehostavia teknisiä lisäjärjestelmiä voi olla muitakin, ja ne ovat roolissa melkein jokaisen konekielikäskyn suorituksen aikana.

Lokaalisuusperiaatetta hyödyntävä sivuttavan virtuaalimuistin perusidea on esitetty kuvassa 0.26: Vasemman laidan laatikot kuvaavat prosessien tarvitsemaa muistitilaa. Keskellä on käytettävissä oleva fyysinen muisti, joka on pienempi kuin prosessien yhteensä tarvitsema muistitila. Oikeassa laidassa on kovalevy, jonka suhteellisen pieneen osioon kaikkien prosessien yhteensä tarvitsema muistitila mahtuu, toisin kuin keskusmuistiin[50]. Kunkin prosessin

[50]Nykyään tyypilliset keskusmuistitkin ovat gigatavuluokkaa, joten kaikki järjestelmät eivät tarvitse kovalevyä. Silloin käynnissä olevien prosessien yhteensä kuluttama muisti on kiinteästi rajoitettu. Sovelluksilla on kuitenkin ollut tapana käyttää aina vain enemmän muistia uudemmissa versioissaan, ja fyysisen muistin kapasiteetin



Kuva 0.26: *Sivuttavan virtuaalimuistin perusidea ja tietorakenteet.*

tarvitsema virtuaalimuisti jakautuu ns. **sivuihin** (engl. *page*), jotka ovat tyypillisesti esim. 4096 tavun mittaisia peräkkäisten osoitteiden pätkiä. Tietokoneen fyysinen muisti puolestaan jaetaan sivun kokoisiin **kehyksiin** (engl. *page frame*). Osoitteiden kartoittaminen virtuaaliosoitteista fyysisiksi tehdään sivukohtaisesti. Sivu kerrallaan voidaan muistia myös tuoda kovalevyltä keskusmuistiin, ikään kuin keskusmuisti olisikin ”suuren suuri välimuisti”. Harvoin käytetty sivu voidaan puolestaan jäädyttää kovalevylle, ikään kuin

loppuessa on aina mahdollista ottaa kovalevy avuksi.

se olisi ”massiivinen kokonaismuisti”. Lisäksi, jos halutaan esim. läppäri täysin virrattomaan horrostilaan, kaikki sivut voidaan jäädyttää kovalevylle laitteiston seuraavaa käynnistystä varten.

Voidaan ajatella, että sivu on “muistinhallinnan perusyksikkö”: Sivun sisältö on aina peräkkäisissä osoitteissa niin fyysisessä muistissa kuin kovalevylläkin. Sivuja siirrellään fyysisten kehysten ja kovalevyn välillä yksiköinä. Muisti kartoitetaan prosessien välillä sivuina, ja myös suojaus- ja lokitiedot ovat sivukohtaisia.

Osoitteenmuunnos virtuaalisesta fyysiseksi

Jokaisen muistiin kohdistuvan toimenpiteen kohdalla (ts. käskyn nouto, käskyn operandien nouto, tulosten tallentaminen muistiin) prosessorin täytyy hoitaa automaattisesti aika montakin asiaa. Ensinnäkin prosessori tekee **osoitteenmuunnoksen** (engl. *address translation*), jossa se muuntaa virtuaalisen muistiosoitteen fyysiseksi osoitteeksi. Se tekee muunnoksen käyttäen prosessoriarkkitehtuurin määräämiä tietorakenteita, jotka käyttöjärjestelmän on luotava ja ilmoitettava MMU:lle silloin, kun jonkun prosessin virtuaalimuistiavaruuden kartta luodaan tai kun sitä tarvitsee muuttaa. Nämäkin tietorakenteet, kuten kaikki, sijaitsevat tietokoneen muistissa.

Käyttöjärjestelmä luo rakenteet keskusmuistiin ja ylläpitää niitä siellä, mutta prosessori nappaa niitä tarpeen mukaan erityisen nopeisiin ns. assosiatiivimuisteihin, nimeltään ”translation look-aside buffer”, **TLB**. Lokaalisuusperiaate toteutuu erittäin vahvasti peräkkäisten konekielikäskyjen tasolla, joten TLB-teknologian avulla muunnokset virtuaalisesta fyysiseksi osoitteeksi tapahtuvat suurimmaksi osaksi yhtä nopeasti kuin laskutoimituksetkin. Myös TLB-muisti on kiinteän kokoinen, joten sen sisältöä täytyy päivittää viittausten siirtyessä kauempaan muistialueeseen. Kun käyt-

töjärjestelmä muuttaa jonkin prosessin muistikarttaa, sen on ilmoitettava prosessorille, että nykyinen TLB-muisti ei ole enää oikeellinen. Seuraavat osoitteenmuunnokset ovat hitaampia, kunnes TLB on taas ajan tasalla. Menettelyn muihin yksityiskohtiin ei ole laitteiston ulkopuolelta edes juurikaan mahdollisuutta vaikuttaa. Tämä on yksi esimerkki lukuisista syistä, joiden vuoksi nykyisten tietokoneiden tarkkoja suoritusaikoja ei oikein pysty ennakoimaan nanosekuntitasolla. Laajemmassa skaalassa aikavaihtelut tietysti keskiarvottuvat.

Prosessorin sivutaulu

Luvussa 0.5 kerrottiin virtuaalimuistiavaruuden jakautumisesta roolitettuihin segmentteihin. Siellä myös viitattiin joihinkin aiempiin prosessoriarkkitehtuureihin, joissa muistin osoittaminen tehtiin ns. segmenttirekisterien avulla. Ohjelman looginen rakenne näkyi siis myös prosessoriarkkitehtuurin suunnittelussa. Nykyäänkin ohjelman alueita sanotaan segmenteiksi, mutta muistin suojaus ja avaruuden kartoittaminen tehdään esimerkiksi x86-64 -arkkitehtuurissa hienojakoisemmin niin sanotun **sivuttamisen** (engl. *paging*) avulla.

Kuvassa 0.27 on esitetty kuvitteellisen prosessorin **sivutaulu** (engl. *page table*), jollainen käyttöjärjestelmän tulee luoda jokaista prosessia varten. Sivutaulun osoite on ilmoitettava prosessorille aina kontekstin vaihdon yhteydessä, eli kun uusi prosessi otetaan suoritusvuoroon. Tässä kuvitteellisessa prosessorissa virtuaalimuistiosoitteet ovat 20-bittisiä ja fyysiset osoitteet ovat 24-bittisiä. Tilanne vastaa esimerkiksi aikakautta, kun 32-bittisessä x86-prosessorissa prosessien muistiosoitteet olivat 32-bittisiä, mutta fyysistä muistia pystyi olemaan tietokoneessa enemmän kuin 32 bitillä osoitettavissa oleva neljä gigatavua. Prosessit olivat rajoitettuja pienempään muistiin (4GB) kuin mitä väylä pystyi osoittamaan pro-

Esimerkki prosessin sivutaulusta fyysisessä muistissa (prosessorin näkökulma).
Leikkiarkkitehtuuri: 20bit virtuaalimuisti, 24bit fyysinen muisti, 4K sivu, 32bit PTE:

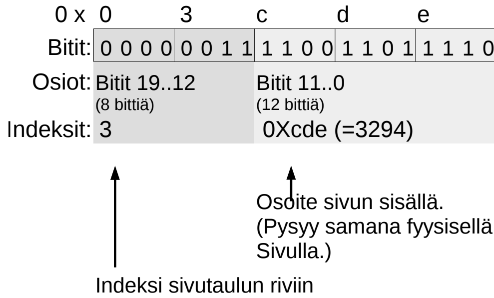
	31–24	23–12	11	10	9	8	7	6	5	4	3	2	1	0		(PTE:n osoite)
rivi/indeksi	käyttöj.	Fyysisen sivun numero	käyttöj.				res.		D	A	E	U	W	P		
0x00		0x000	0	0	0	0	0	0	0	0	0	0	0	0	←	0x027000
0x01		0x345	0	0	1	0	0	0	0	1	1	1	0	1	←	0x027004
0x02		0x88a	0	0	0	0	0	0	0	0	1	1	0	1	←	0x027008
0x03		0x678	0	0	0	0	0	0	0	1	0	1	1	1	←	0x02700c
0x04		0x000	0	0	0	0	0	0	0	0	0	1	1	0	←	0x027010
0x05		0x9ec	0	1	0	0	0	0	1	1	0	1	1	1	←	0x027014
…		…	…													…
0x7f		0x3c0	0	0	0	0	0	0	1	1	0	1	1	1	←	0x0271fc
0x80		0x000	0	0	0	0	0	0	0	0	0	0	0	0	←	0x027200
…		…	…													…
0xfe		0x0e2	0	0	0	0	0	0	1	1	0	0	1	1	←	0x0273f8
0xff		0x0f0	0	0	0	0	0	0	0	1	1	0	0	1	←	0x0273fc

Kuva 0.27: *Leikkiesimerkki prosessin sivutaulusta 20-bittisellä virtuaaliosoiteavaruudella, 24-bittisellä fyysisellä osoiteavaruudella ja 4096 tavun sivukoolla.*

sessorin ulkopuolella (fyysinen muistiavaruus oli teratavuluokkaa). Nyt 64-bittisellä aikakaudella tilanne on jälleen päinvastainen: 64-bittisellä virtuaalimuistiosoitteella voisi saavuttaa eksatavuluokan muistiavaruuden, jollaista määrää fyysistä muistia on toistaiseksi mahdotonta (ja tarpeetonta) ahtaa isoonkaan tietokoneeseen. Fyysinen muistiavaruus onkin nykyään paljon pienempi kuin virtuaalinen[51].

Kuva vaatinee hieman lisää selitystä ja vilkaisua myös kuvaan 0.28, jossa on yksi esimerkki mainitunlaisen leikkiarkkitehtuurin virtuaalimuistiosoitteesta. Osoitteessa on maksimissaan 20 merkitsevää bittiä, ja se jakautuu kahteen osaan. Ensimmäiset 8 bittiä (merkitsevyysnumeroinnissa bitit 12-19) yksilöivät *sivun*, johon viitattu muistipaikka kuuluu. Jälkimmäiset 12 bittiä (merkitsevyysnumeroinnissa bitit 0-11) yksilöivät yhden tavun osoitteen sivun sisällä. Lienee kuvan perusteella selvää, että yhdellä sivulla voi olla 4096

[51]Esimerkiksi AMD64 määrittelee fyysisen muistiosoitteen ehdottomaksi maksimileveydeksi 52 bittiä, joilla voi osoittaa ”vain” noin 4.5 petatavun laajuudelta eri osoitteita (tarkkaan ottaen 4503599627370496 tavua).



Kuva 0.28: *Leikkiesimerkki prosessin virtuaalimuistiosoitteesta 20-bittisellä virtuaaliosoiteavaruudella ja 4096 tavun sivukoolla.*

tavua (sisäiset indeksit 0x0, 0x1, . . . , 0xfff), ja prosessilla voi olla kartoitettuna 256 sivua (indeksit 0x0, 0x1, . . . , 0xff). Prosessi voi siis tällaisessa arkkitehtuurissa nähdä ja käyttää maksimissaan $256 * 4096 = 1048576$ tavua muistia. Osoitteenmuunnoksessa sivun sisäinen osoite pysyy muuttumattomana, mutta virtuaalisen sivunumeron tilalle laitetaan fyysisen *sivukehyksen* numero, jolloin *käytännössä muodostuu fyysinen muistiosoite*, jolla voi väylän kautta osoittaa koneeseen asennettua keskusmuistia. Prosessori määrittää fyysisen sivunumeron *yksinkertaisesti lukemalla sivutaulusta rivin, jonka indeksi on virtuaalimuistiosoitteen alkuosa.* Palataan siis kuvaan 0.27 tutkimaan, miten tässä esimerkkitapauksessa virtuaaliosoite 0x3cde muuttuisi fyysiseksi.

Esimerkin virtuaaliosoitteen alkuosa on 0x03, joten prosessori lukee sivutaulusta rivin, jonka indeksi on 3. Kyseiselle riville on käyttöjärjestelmä kirjoittanut, että prosessin ”sivu 3” tulee kartoittaa fyysisen muistin kehykseen numero 0x678. Prosessori ymmärtää siis käyttää väylän kautta fyysistä muistipaikkaa 0x678cde. Huomaa, että loppuosa pysyi samana, mutta alkuosan tilalle tuli sivutaulussa ilmoitettu fyysinen sivunumero. Koska kaikki käytettävissä oleva fyysinen muisti on jaettu 4096 tavun kokoisiin ke-

hyksiin, vastaa yhdistelmä suoraan yhtä fyysisen osoiteavaruuden peräkkäisjärjestyksessä numeroitua tavua.

Tarkastellaan vielä kuvan 0.27 sivutaulua: Siinä on ensinnäkin 256 riviä (joista osaa ei ole piirretty mukaan, vaan niiden tilalla on ”. . . ”). Varmemmaksi vakuudeksi kuvan oikeaan laitaan on kirjattu fyysiset muistiosoitteet, joihin sivutaulu voisi olla tallennettu. Jokainen sivutaulun rivi eli **sivutaulumerkintä** (engl. *page table entry, PTE*) on 32 bitin (eli 4 tavun) mittainen, joten riveihin mahtuu fyysisen kehyksen indeksin lisäksi muutakin tietoa. Itse asiassa kuva vastaa pitkälti oikean AMD64:n sivutaulun rakennetta tärkeimpine tietoineen[52].

Fyysisen sivun indeksi on biteissä 12–23. Prosessori ei tulkitse bittien 24–31 eikä bittien 8–11 sisältöjä, eikä myöskään itse muuta niitä, joten käyttöjärjestelmätoteutus saa käyttää niitä, mihin se itse haluaa. Esimerkkiin liittyy tietysti jokin kuvitteellinen käyttöjärjestelmä, joka ”käyttäis nyt vaikka bittejä 8–11 johonkin omaan kirjanpitoonsa”, joten niissä on muutakin kuin nollia. Bittejä 24–31 ei tässä esimerkissä kukaan käytä mihinkään, joten niiden kohdalle ei ole selkeyden vuoksi piirretty edes nollia. Bitit 6–7 on kuvitteellisen prosessorin valmistaja merkinnyt ”res.” eli ”reserved”, koska niille voidaan myöhemmässä prosessorimallissa määrittää jokin uusi käyttötarkoitus. Nykyisten käyttöjärjestelmien tulee pitää ne aina nollina. Bitit 0–5 ovat tärkeässä roolissa nykyaikaisen virtuaalimuistin toiminnan kannalta. Ne on koodattu kirjaimilla, joiden merkitykset ovat (AMD64:n manuaalia mukaillen):

[52] Taulun rivimäärä on tässä 256, kun se AMD64:ssä on 512. Rivin pituus on tässä 32 bittiä, kun se AMD64:ssä on luonnollisesti 64. Välimuistien käyttöön liittyvät bitit on jätetty yksinkertaisuuden vuoksi pois. Kiinnostunut lukija löytää AMD64:n manuaalin osasta System Programming Reference luvusta 5 alkuperäisen lähdemateriaalin ja lisää havainnekuvia.

D eli **dirty-bitti**: Prosessori asettaa tämän bitin ykköseksi, kun se suorittaa sivulle kirjoitusoperaation. Käyttöjärjestelmä voi päätellä tästä, että sivun sisältö on muuttunut ”sitten viime näkemän”.

A eli **accessed-bitti**: Prosessori asettaa tämän bitin ykköseksi aina, kun se käyttää sivua (ts. lukee, kirjoittaa tai suorittaa yhdenkään sivun tavuista). Käyttöjärjestelmä voi ajoittain tarkastella tätä bittiä ja päätellä, onko sivu aktiivisessa käytössä. Käyttöjärjestelmä voi siis esim. päivittää jotakin ajassa etenevää käyttölaskuria ja nollata bitin seuraavaa havaintopistettä varten.

E eli **execute-bitti**: Käyttöjärjestelmä kirjaa tähän, saako sivun sisältöä suorittaa koodina, eli saako sieltä noutaa käskyn. Jos tämä on nolla, ja prosessorin pitäisi ladata sivulla oleva osoite IP:hen, prosessori keskeyttää prosessin välittömästi ja siirtää käsittelyn käyttöjärjestelmän suojausvirheen käsittelijään. Tämä estää ohjelmointivirheen vuoksi tapahtuvat hypyt dataksi tarkoitettuihin muistialueisiin.

U eli **user-bitti**: Käyttöjärjestelmä kirjaa tähän, onko sivu tarkoitettu sovellusohjelmille eli käyttäjätilan prosesseille. Jos bitti on 0 sellaisella sivulla, johon käyttäjätilan prosessi osoittaa, prosessori keskeyttää ja siirtää suorituksen käyttöjärjestelmän suojausvirheen käsittelijään. Tämä on siis avain käyttöjärjestelmän koodin ja datan suojaamiseen rikkinäisiltä ja pahantahtoisilta sovellusohjelmilta.

W eli **write-bitti**: Käyttöjärjestelmä kirjaa tähän, onko prosessilla oikeus kirjoittaa sivulle. Jos bitti on 0 kirjoitusoperaation kohteena olevalla sivulla, arvatenkin prosessori keskeyttää ja siirtää suojausvirheen käsittelijään. Tämä on avain

esimerkiksi siihen, ettei loppukäyttäjän mielivaltaiset (ja pahimmillaan pahantahtoiset) syötteet voi koskaan päätyä koodialueelle suoritettavaksi koodiksi.

P eli **present-bitti**: Käyttöjärjestelmä kirjaa tähän, onko sivu tällä hetkellä fyysisessä muistissa. Jos bitti on 0, prosessori tietää, että sivu ei ole saatavilla keskusmuistissa. Silloin se keskeyttää prosessin, ja siirtää suorituksen käyttöjärjestelmälle käsittelemään ns. **sivuvirhettä** (engl. *page fault*). Tämä on avain massamuistin käyttöön keskusmuistia laajemman kapasiteetin saavuttamiseksi. Menettelyä käsitellään erikseen myöhemmässä luvussa.

Monisteen kirjoittaja on yrittänyt asetella kuvan 0.27 bitit ja osoitteet sillä tavoin, että tilanne voisi olla järkevä, ja että siihen syventymällä voisi saada realistisen kuvan muistinhallinnasta. Ainakin seuraavien pitäisi toteutua: Sivuihin, joita prosessi saa suorittaa, ei saa kirjoittaa (W==0 aina kun E==1). Muuttuneita voivat olla vain sivut, joihin voi kirjoittaa (D==1 vain jos W==1). Puolet prosessin muistiavaruudesta on varattu käyttöjärjestelmälle (U==0 rivistä 0x80 alkaen). Alkupuolen loppu on varattu pinokäyttöön, joten sivua 0x7f voi kirjoittaa, mutta ei suorittaa (W==1 ja E==0 virtuaalisivulla 0x7f). Koodialue on sijoitettu muistiavaruuden alkupuolelle (E==1 muutamalla alkupuolen sivulla[53]). Osa sivuista on kartoittamatta (kaikki nollaa), etenkin aivan muistiavaruuden alku. Joitakin sivuja ei ole vielä käytetty edellisen tarkistuspisteen jälkeen (A==0). Osaan kirjoitettavista sivuista ei ole toistaiseksi kirjoitettu (D==0 vaikka W==1). Jotkut sivut eivät ole fyysisessä muistissa (P==0), mikä siis tarkoittaa, että ne ovat kovalevyllä odottelemassa seuraavaa käyttötarvetta.

[53] Pieni ohjelma tässä on kyseessä. . . ehkä “hello world”. . .

Esimerkki virtuaalimuistiosoitteen osista AMD64-prosessorissa, aidossa 64 bitin tilassa, 4K sivukoolla:

0 x	0	0	0	0	0	0	0	0	0	0	4	0	0	9	f	5
Bitit:	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	1 0 0 1	1 1 1 1	0 1 0 1

Osiot:	Bitit 63..48	Bitit 47..39 (9 bittiä)	Bitit 38..30 (9 bittiä)	Bitit 29..21 (9 bittiä)	Bitit 20..12 (9 bittiä)	Bitit 11..0 (12 bittiä)
Indeksit:		0	0	2	0	0X9f5 (=2549)

Osoite sivun sisällä. (Pysyy samana fyysisellä Sivulla.)

Indeksi sivutaulun riviin

Indeksi sivuhakemiston riviin

Indeksi sivuhakemisto-osoitintaulukon riviin

Indeksi neljännen tason sivukartan riviin

Tulee olla samoja kuin bitti 47 nykyisessä arkkitehtuurissa

Kuva 0.29: *Nelitasoinen osoitteenmuunnos AMD64-prosessorissa (alkuperäinen x86-64). Arkkitehtuuri tukee muutamaa isompaa sivukokoa, mutta tässä on esimerkki tyypillisimmästä tapauksesta eli 4096 tavun (=* 2^{12}*) kokoisten sivujen käytöstä.*

Myös käyttöjärjestelmän omat sivut (U==0) voivat olla suojattu vahingossa tapahtuvaa kirjoittamista tai suorittamista vastaan (W==0 tai E==0), eikä niistäkään kaikkia ole kartoitettu mihinkään (kaikki nollaa). Sivutaulu sijaitsee fyysisessä muistissa osoitteessa 0x27000, mikä ilmeisesti sijoittuu käyttöjärjestelmän sivutauluille varattuun alueeseen. Prosesseille allokoidut fyysiset osoitteet ovat reilusti suurempia (>0x300000). Käyttöjärjestelmän sivut voivat osoittaa pienempiinkin osoitteisiin, riippuen käyttötarkoituksesta.

Todellisempi esimerkki: AMD64:n nelitasoinen sivutaulusto

Nykypäivän prosessoreissa virtuaalimuistiavaruus on laaja, ja osoitteenmuunnos voi sivutaulun kokonaistilan säästämiseksi tapahtua monitasoisen taulukkohierarkian kautta. Esimerkiksi x86-64:ssa tapahtuu tyypillisimmässä 64-bittisessä toimintatilassa nelitasoinen osoitteenmuunnos, joka esitetään kuvassa 0.29. 64-bittisessä virtuaalimuistiosoitteessa on nykyisellään (vuonna 2013 julkaistussa

AMD64-manuaalissa) 48 bittiä käytössä, mutta tätä on mahdollista tulevissa prosessorimalleissa laajentaa täyteen 64 bittiin. 9 bitin mittaiset pätkät ovat indeksejä, joilla löytyy aina seuraavan tason taulukon fyysinen muistiosoite ja lopulta alimmalla tasolla sijaitsevasta varsinaisesta sivutaulusta kehyksen osoite, aivan samoin kuin edeltävässä yksitasoisessa leikkiesimerkissä. Kaikki taulukot sijaitsevat keskusmuistissa, ja prosessori toimintansa nopeuttamiseksi lataa niitä myös välimuistiin ja TLB:hen. Taulukoiden ”rivit” ovat 64 bitin mittaisia, ja niihin sisältyy seuraavan tason taulukon fyysisen osoitteen lisäksi mm. suojaustietoa ilmaisevat bitit hyvin pitkälti samoin kuin kuvan 0.27 esimerkissä. Kussakin taulukossa on 512 riviä (koska indeksit ovat 9 bitin mittaisia), joten myös taulukot ovat täten 4096 tavun mittaisia. Tämä on mukavan symmetristä, koska taulukoita voidaan sivuttaa kuten muutakin muistia, ja yksi taulukko mahtuu aina tarkalleen yhteen kehykseen. Itse asiassa arkkitehtuuri vaatii, että jokainen taulukko alkaa 4096:lla jaollisesta osoitteesta, joten ne väistämättä menevät yksi-yhteen sivukehysten kanssa.

Käytännöt vaihtelevat prosessoriarkkitehtuurien välillä, joten useita prosessorimalleja tukevan käyttöjärjestelmän täytyy muuntaa omat sisäiset sivutaulustonsa kutakin prosessorimallia varten erikseen. Konekielellä on ohjelmoitava vähintään käskyt, joilla prosessorin järjestelmärekisteriin kirjoitetaan vuoroon tulevan prosessin sivutaulun (tai taulukoston) alkuosoite. Lisäksi arkkitehtuurikohtaisia konekielikäskyjä saatetaan tarvita ilmoittamaan prosessorille, että nykyisen sivutaulun tietoja on muutettu. Tehokkuussyistä prosessori olettaa ilman erillistä herätettä, että sen TLB on ajan tasalla.

Lokaalisuusperiaate, välimuistit ja heittovaihto

Osoitteenmuunnoksessa selviää fyysinen muistiosoite ja se, onko muistioperaatio ylipäätään mahdollinen. Vasta sen jälkeen voi tapahtua varsinainen luku tai kirjoitus muistiin. Tässä kohtaa voidaan suorituskykyä optimoida prosessorin sisäisillä välimuisteilla ja muistin kokonaiskapasiteettia lisätä ”heittämällä” osa sivuista kovalevylle.

Välimuistit prosessorin sisällä

Jos osoite on sallittu ja tällä hetkellä keskusmuistissa, prosessori tarkistaa seuraavaksi, onko muistipaikan sisältö tällä hetkellä lähimmässä välimuistissa. Jos ei ole, niin sisältö pitää hakea välimuistiin kauempaa. Samalla tuodaan kokonainen pötkö muistia viitatun osoitteen ympäriltä. Lokaalisuusperiaatteeseen luottaen prosessori olettaa, että seuraavat käskyt käsittelevät todennäköisesti läheisiä muistipaikkoja. Pötkö on nimeltään **välimuistirivi** (engl. *cache line*). Välimuistirivin pituus ja välimuistiin kerrallaan mahtuvien rivien määrä vaihtelee eri prosessorityyppien sekä eri tarkoituksiin tehtyjen välimuistien välillä. Kapasiteetti on joka tapauksessa rajallinen. L1-välimuistissa esimerkiksi saattaa olla 64 kappaletta 64 tavun mittaista riviä. Intelin x86-arkkitehtuuriin perustuvissa prosessoreissa myös L2-välimuistin rivit ovat 64-tavuisia. Välimuistissa tulee siis jokaisen muistioperaation jälkeen olemaan kaikki 64 tavua fyysisen muistiosoitteen ympäriltä, alkaen edellisestä 64:llä jaollisesta osoitteesta. Kriittisimmissä nopeusoptimoinneissa voi hyödyntää tätä tietoa sijoitellessaan dataa muistiosoitteisiin[54].

[54]Muista kuitenkin, että todelliset nopeutukset tulevat aina kokonaisalgoritmin suunnittelun eikä pienen mittakaavan näpertelyn kautta! Kurssimme Algoritmit 1 ja Algoritmit 2 johdattelevat aiheeseen. Datan sijoittelua ja konekielikoodia algoritmin sisimmässä silmukassa ei ole mitään syytä miettiä ennen kuin on varma, että (A)

Varsin usein täytyy korvata jokin aiempi välimuistirivi kopioimalla kauemmasta välimuistista tai keskusmuistista. Jos tuo aiempi välimuistirivi oli **likainen** (engl. *dirty*) eli siihen oli kirjoitettu uusia arvoja, jotka eivät vastaa kauemmassa muistissa olevaa alkuperäistä tietoa, niin korvattavaksi valittu välimuistirivi pitää ensin kirjoittaa talteen ja vasta sitten sen tilalle voi tuoda uuden.

Välimuisti sijaitsee prosessoriytimen sisällä. Entäs jos esimerkiksi kirjoittamista varten välimuistiin tarvittava data on tällä hetkellä likaisena toisen ytimen välimuistissa? Tilanne menee entistä hankalammaksi, kun toisen ytimen välimuistista pitää käydä ensin kirjoituttamassa likainen data talteen keskusmuistiin. Vaikka välimuistit ovat pitkälti automaattisia, sovellusohjelmoijan tulee ymmärtää niiden olemassaolo, jotta niistä saa varmasti aina enemmän hyötyä kuin haittaa.

Välimuistin ruuhkautuminen (Cache thrashing)

Kuten havaittiin, jokainen muistiosoitus voi aiheuttaa joko hyvin vähän tai hyvin paljon elektronisia operaatioita ennen kuin konekielikäskyn suoritus saadaan valmiiksi. Onneksi lokaalisuusperiaate yleensä toteutuu, ja ajoittain välttämättömistä hitaista vaihdoista huolimatta kaikkein suurimman osan aikaa välimuistista löytyy juuri seuraavaksi tarvittavat tavut. Lisäksi sovellusohjelmoija ymmärtää viimeistään käyttöjärjestelmiä koskevan kurssin käytyään, että ohjelmat kannattaa suosiolla tehdä sellaisiksi, että luonnollinen lokaalisuus toteutuu niissä mahdollisimman hyvin. Normaalit ohjelmointirakenteet, kuten aliohjelmakutsut ja peräkkäin muistiin ladottujen taulukkoalkioiden tai olioiden läpikäyn-

algoritmi on tehtävän ratkaisemiseen paras mahdollinen, (B) säikeistys ja muut helpommat keinot on jo käytetty, (C) nykyinen suoritusaika ei jostakin syystä ole *riittävä* nykyiseen tarpeeseen ja (D) että pienen mittakaavan näpertelyllä voi oikeasti saada riittäviä vaikutuksia työmäärään nähden.

ti peräkkäisjärjestyksessä, tuottavat luonnostaan välimuisteilla tehostuvaa koodia.

Vahingossa etäisten muistipaikkojen välillä hyppivä koodi voi aiheuttaa ilmiön nimeltä **välimuistin ruuhkautuminen** (engl. *cache thrashing*), jossa suuri osa ajasta kuluu hyötylaskennan sijasta välimuistin päivittämiseen. Esimerkiksi kaksiulotteisen taulukon eli matriisin käyminen läpi indekseillä voi aiheuttaa tämän, mikäli ohjelmoija ei tiedä, tallentaako alustajärjestelmä luvut riveittäin vai sarakkeittain, ja arvaa väärin:

```
for (int i=0; i<m; i++){
  for (int j=0; j<n; j++){
    mat[i][j] = 0.0;      // Kaikki hyvin, jos data makaa
                          // muistissa riveittäin.
  }
}

for (int j=0; j<n; j++){
  for (int i=0; i<m; i++){
    mat[i][j] = 0.0;      // Auts! En kai juuri vaihdattanut
                          // prosessorissa välimuistiriviä!?
  }
}
```

Sikäli kuin edelliset esimerkit olisivat C-kieltä ja käsiteltävä matriisi kohtalaisen kokoinen, ensimmäinen versio toimisi normaalilla tavalla. Välimuistit hyödyttäisivät sen verran, minkä ne nyt normaalissa taulukon läpikäynnissä ylipäätään pystyvät. Jälkimmäinen versio kuitenkin aiheuttaisi käytännössä jokaisen muistiosoituksen kohdalla uuden välimuistirivin hakemisen keskusmuistista ja jonkin edellisen välimuistirivin tallentamisen alta pois (koska koodihan kirjoittaa muistiin).

Esimerkiksi FORTRANissa ja C-kielessä matriisien tallennustapa on nimenomaisesti eri päin. Jos halutaan tehokasta koodia, joka käyttää ajan laskemiseen, eikä turhaan välimuistirivien vaihdatta-

miseen, täytyy vastaavissa tilanteissa varmistaa kielen spesifikaatiosta alkioiden fyysinen tallennusjärjestys[55].

Sovellusohjelmoijan kannattaa muistaa myös, että eri ytimissä toimiviksi tarkoitettujen säikeiden ei kannattaisi turhaan kirjoittaa lähekkäisiin muistipaikkoihin, koska silloin välimuistien hyöty muuttuu haitaksi, kun prosessorien välillä täytyy synkronoida muistin sisältöä usein.

Heittovaihto eli swappaus

Sivuttaminen mahdollistaa muiden hienojen ominaisuuksien lisäksi myös lokaalisuusperiaatteen hyödyntämisen prosessorin välimuistien ulkopuolella. Menettelyn nimi on **heittovaihto** (engl. *swapping*), jossa tavoitellaan ulkoista massamuistia hyödyntäen maksimaalista muistikapasiteettia, kuitenkin siten, että kokonaisuus pysyy riittävän nopeana. Massamuisti on todella hidas, mutta se on halpa ja sinne mahtuu hirmuisen paljon dataa keskusmuistiin verrattuna.

Perusidea on sama kuin prosessorin sisäisissä välimuisteissa, mutta kyseessä on käyttöjärjestelmän muistinhallinnan ohjelmallisesti hoitama toiminto, jossa tiedonsiirto tapahtuu I/O -laitetta eli esim. kovalevyä käyttäen. Tässä tapauksessa myöskään muistin siirtoyksikkönä ei ole välimuistirivi vaan muistinhallinnan muutoinkin käsittelemä sivu.

Kussakin fyysisen muistin kehyksessä säilytetään yhtä sivua prosessien ”näinä aikoina” eniten tarvitsemaa muistialuetta, esim. jo-

[55]Myös on niin, että jos esim. jollekin matemaattiselle algoritmille on hieno, toimiva ja tehokas toteutus julkaistu 1970-luvulla FORTRAN-kielellä, niin sen ”copy-paste-modify” -soveltamisessa tänä päivänä täytyy varmistaa, että silmukat käydään oikeassa järjestyksessä suhteessa prosessorin välimuisteihin! Saattaa vaatia joitakin muutoksia alkuperäiseen.

takin yhtenäistä pätkää koodista tai datasta. Muistissa kulloinkin olevia sivuja sanotaan **työjoukoksi** (engl. *working set*), johon prosessorin muistiviittaukset kohdistuvat aina onnistuneen osoitteenmuunnoksen jälkeen. Sen sijaan ”kauan sitten” tarvitut sivut voivat odotella levyllä ”jäädytettynä”. Suspended -tilassa täysin pysähdyksissä olevien prosessien kaikki sivut voi heittää levylle, koska niitä ei tulla tarvitsemaan ennen kuin prosessi taas jossain vaiheessa tieten tahtoen herätetään käyntiin. Lopputuloksena ohjelmille varatun muistin kokonaismäärää rajoittaa kovalevylle varattu ns. **heittovaihtotila** (engl. *swap space*) eikä fyysinen muisti, joka saattaa olla pienempi.

Muistinhallinnan tietorakenteet käyttöjärjestelmässä

Prosessoriarkkitehtuurin määräämä sivutaulu sisältää vain välttämättömät perustiedot fyysisen muistin käyttöön. Rakenne on myös täysin riippuvainen prosessorimallista. Prosessori ei myöskään tarvitse, eikä siten myöksään tarjoa valmiina, tietorakennetta fyysisen muistin kehysten ylläpitoon. Se kun operoi riittävässä määrin kulloinkin valittuna olevan sivutaulunsa perusteella. Käyttöjärjestelmän muistinhallinta voi kuitenkin itse pitää yllä erilaisia lisätietoja.

Heittovaihdon yksityiskohdat ovat joka tapauksessa käyttöjärjestelmän vastuulla, koska prosessori vain minimaalisesti mahdollistaa tekniikan sivutaulun yksittäisten bittien kautta. Tärkein niistä on tietysti ”present-bitti”, joka kertoo, onko sivu läsnä fyysisessä muistissa. Roolissa ovat myös ”accessed-bitti”, jonka perusteella käyttöjärjestelmä voi arvioida kunkin sivun viimeaikaista käyttöastetta sekä ”dirty-bitti”, joka kertoo, tarvitseeko fyysisen muistin sivun sisältö tallentaa ennen sen korvaamista uudella.

Tässä luvussa irtaannutaan yksinkertaisuuden vuoksi todellisuudesta. Käsitellään yksinkertaista mallia, jossa koko muisti pidetään kovalevyllä ja keskusmuistia käytetään ison välimuistin roolissa. Kehyksillä ja sivuilla on selkeät, edellä kuvaillut, määritelmät. Alkeellinen muistinhallinnan toteutus olisi ilman muuta mahdollinen tälläkin tavoin, mutta todellisissa järjestelmissä tarvitaan varmasti monenlaisia optimointeja riittävän suorituskyvyn saavuttamiseksi[56]. Tässä käsitellään yksitasoisia sivutauluja, vaikka suuret virtuaalimuistiavaruudet edellyttävät käytännössä monitasoisia taulustoja[57].

Käyttöjärjestelmän sivutaulut

Käyttöjärjestelmän täytyy joka tapauksessa tuottaa prosessorin käyttöön esimerkiksi luvun 0.10 mukainen sivutaulu jokaista prosessia varten, joten samat tiedot on luonnollisesti oltava mukana myös käyttöjärjestelmän omissa tiedoissa. Näiden lisäksi käyttöjärjestelmä voi pitää omissa sivutauluissaan lisätietoja, kuten kirjanpitoa prosessin käyttämästä muistista. Tämän kurssin leikkiesimerkissä ajatellaan myös levyosoitteen olevan mukana käyttöjärjestelmän ylläpitämässä prosessikohtaisessa sivutaulussa[58]. Sivutaulun rivillä (yhdessä PTE:ssä) olkoon nyt siis seuraavat tiedot:

- Fyysisen sivun numero, jos sivu on keskusmuistissa: Osoitteenmuunnos tapahtuu tämän tiedon perusteella.

[56]Kiinnostunut lukija löytää runsaasti nettilähteitä esimerkiksi Linuxin ja BSD:n muistinhallinnan toteutuksesta. Niissä on myös lähdekoodi saatavilla, joten koko toteutus on periaatteessa tutkittavissa. Apuna on syytä olla myös aiheesta kertova kirja. Netistä saatavilla on ainakin `https://www.kernel.org/doc/gorman/pdf/understand.pdf`

[57]Esimerkiksi Linuxin sisäinen sivutaulusto on kolmitasoinen.

[58]Näin ainakin vielä vuonna 2016. Jos aikaa on, voi olla, että leikkiesimerkkikin saattaa jossain myöhemmässä vaiheessa lähestyä joiltain osin Linuxin todellista rakennetta esim. edellisen alaviitteen nettilinkin pohjalta.

- Bitit kommunikointiin prosessorin kanssa – siis aiemmin esitetyt D, A, E, U, W ja P.
- Kiinteä sijainti levyllä. Simppelissä perusmallissa jokaisesta sivusta on jatkuvasti levyllä tallessa jäädytetty kopio: Käyttöjärjestelmä tietää tämän perusteella, mistä kohtaa levyä sen pitää lukea sivun sisältö, jos se ei toistaiseksi ollut muistissa.

Kehystaulu

Pelkillä prosessorin sivutauluilla pärjättäisiin hyvin, jos kiinnitettäisiin muistin maksimimäärä samaksi kuin fyysisen muistin kapasiteetti. Heittovaihdon päämäärä on kuitenkin muistikapasiteetin kasvattaminen suuremmaksi kuin fyysiseen muistiin mahtuu kerrallaan sivuja. Pelkkien sivutaulujen tiedot eivät enää riitä siinä vaiheessa, kun fyysinen muisti on täpötäynnä. Uuden sivun lataaminen nimittäin edellyttää silloin jonkin jo käytössä olevan kehyksen sisällön korvaamista. Korvattava kehys täytyy valita jollakin periaatteella, mieluiten siten, ettei kokonaisuuden suorituskyky kärsi liikaa. Lisäksi *kaikkien korvattavaa sivua käyttävien prosessien sivutauluihin* täytyy merkitä, että sivu ei ole keskusmuistissa. Siis niiden P-bitti ja fyysinen osoite täytyy käydä asettamassa nollaksi. Tarvitaan siis jonkinlainen "takaisinkartoitus" (engl. *reverse mapping*) fyysisestä kehyksestä prosessitauluihin. Kaikkien prosessien sivutaulujen läpikäynti olisi mahdollinen, mutta kovin hidas tapa löytää kehykseen kartoitetut virtuaalisivut[59].

Yksinkertaisimmillaan takaisinkartoitus saadaan aikaan, kun käyttöjärjestelmään toteutetaan fyysisiä kehyksiä ylläpitävä **kehys-**

[59]Esimerkiksi alkuperäinen Linux-ydin kävi läpi kaikkien prosessien sivutaulut. Menettelyä tehostettiin myöhemmin suurin piirtein tässä esitetyllä keinolla, mutta nykyisellään Linux käyttää vieläkin tehokkaampaa tapaa.

taulu (engl. *frame table*). Kehystaulussa olkoon esimerkiksi seuraavat tiedot:

- ”takaisinkartoitus”: Mihin prosessiin/prosesseihin tämän kehyksen sisältämä sivu on kartoitettu, ts. missä sivutauluissa on PTE, jonka fyysinen sivunumerokenttä vastaa kehystä. Monesko PTE kussakin prosessissa on kyseessä. Prosessikohtaiseen sivutauluun päästään käsiksi tämän tiedon kautta silloin, kun kehyksen sisällöksi pitää vaihtaa uusi sivu massamuistista lukemalla. Fyysisten sivujen käyttäminen usean prosessin jaettuna muistialueena edellyttää listamaista dynaamista tietorakennetta, koska sivuhan voi olla käytössä usealla prosessilla. Tämän johdantokurssin esimerkeissä ja tenttikysymyksissä ei kuitenkaan (vuonna 2016) ole jaettua muistia, vaan jokainen kehys viittaa takaisin täsmälleen yhteen prosessiin ja sivutaulun riviin.

- ”seinäkelloaika”: *Jonkinlainen* tieto kehyksessä sijaitsevan sivun edellisestä käyttöajankohdasta. Varsinainen seinäkelloaika olisi liian raskas ja hidas formaatti käsitellä, joten kyseessä on jonkinlainen kokonaislukulaskuri, jota käyttöjärjestelmä kasvattaa kellokeskeytyksen yhteydessä ja nollaa silloin, kun prosessori on ilmoittanut sivutaulun kautta, että jotakin osoitetta sivulta on käytetty. Aikalaskuri liittyy tässä esitettävään, ”helposti ymmärrettävään” LRU-menettelyyn; korvattavan sivun valinta on todellisuudessa vaikea suunnittelutehtävä, joka vaikuttaa heittovaihdon tehokkuuteen dramaattisesti ja oletettavasti erilaisin tavoin riippuen siitä, missä järjestyksessä sovellukset sattuvat käyttämään muistia[60].

[60]Kohtuuttoman hidasta olisi esim. käydä prosessin vaihdon yhteydessä läpi kaikkien prosessin sivujen A-bitit aikalaskurien päivittämiseksi. Täytyisi siis esimerkiksi tarkistella A-bittejä vain silloin tällöin ja joidenkin sivujen osalta. Käyttöajankohdat ovat siis käytännössä jonkinlaisia arvioita todelliseen muistinkäyttöön nähden.

Kehystaulun ja sivutaulujen käyttö sivunvaihtokeskeytyksessä

Sivutaulun esittelyn yhteydessä todettiin, että mikäli sivutaulumerkinnän osoittama sivu ei ole keskusmuistissa, prosessori havaitsee tämän rakenteeseen kuuluvan läsnäolobitin (P) avulla, jolloin tapahtuu **sivunvaihtokeskeytys**, toiselta nimeltään "**sivuvirhe**" (engl. *page fault exception*), jonka kautta käyttöjärjestelmä pääsee lataamaan ja tallentamaan sivuja levyltä/levylle.

Sivuvirhe on sinänsä normaali keskeytys: prosessi, jonka koodissa normaali osoitteenmuunnos johtaa sivulle, joka ei olekaan missään fyysisen muistin kehyksessä, keskeytetään, ja kontrolli siirtyy käyttöjärjestelmän koodiin. Käyttöjärjestelmän pitää sitten ladata sivun sisältö fyysisen muistin vapaaseen kehykseen. Tarvittavat tiedot fyysisen muistin tilanteesta löytyvät kehystaulusta ja sen kautta saatavilla olevista prosessikohtaisista sivutauluista. Sivunvaihtokeskeytyksen käsittelyssä käyttöjärjestelmän täytyy huolehtia seuraavista toimenpiteistä:

- Ladattavan sivun sijainti levyllä löytyy suoraan sen prosessin sivutaulusta, joka aiheutti sivuvirheen.
- Jos vapaata kehystä ei ole (eli tietokoneen fyysinen muisti on todellakin täynnä), pitää ensin valita joku käytössä olevista ja vaihtaa (engl. *swap*) sen sisältämä sivu puolestaan levylle jemmaan. Valinta tehdään **sivunkorvausalgoritmilla** (engl. *page replacement algorithm*), joita on monia.
- Tässä esimerkissä käytetään korvausalgoritmia nimeltään **LRU least-recently-used**, eli käyttöjärjestelmän muistinhallintamoduuli heittää käyttöajankohdan mukaan kauimmin käyttämättä olleen sivun levylle ja lataa sen tilalle uuden.

- Jos jokin sivu heitettiin levylle pois fyysisestä muistista, on entiseen sivuun kartoitettujen prosessien sivutauluja myös muutettava vähintäänkin siten, että virtuaalisivun läsnäolobitti (P) nollataan. Fyysisen sivunumeronkin voi nollata, koska se tullee olemaan seuraavalla kerralla eri. Sivun sisällön käyttämiseen tarvitaan seuraavalla kerralla jälleen vastaava sivuvirheen käsittely, johon prosessori päätyy luonnostaan P-bittiä tutkiessaan.

- Myös kehystaulun tiedot on päivitettävä uutta tilannetta vastaavasti, eli tässä esimerkissä kehyksen omistajaprosessi(t) voi(vat) vaihtua ja "seinäkelloaikaa" kuvaava laskuri pitää nollata, koska ladattu sivu on nyt kaikkein uusin.

- Sivunvaihdon pitkään kestävien levyoperaatioiden ajaksi voidaan antaa ajovuoro jollekin toiselle prosessille suoritusvalmiiden listasta ja jättää sivuvirheen aiheuttaja odottelemaan tarvitsemansa sivun latautumista (blocked-tilaan). Näin kokonaisjärjestelmä jatkaa toimintaansa, ja vain sivunvaihtoa tarvitseva prosessi hidastelee. Toki hidastelee myös kovalevyn käyttö muihin tarkoituksiin.

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. Ymmärrettävästi heittovaihto on myös tyypillinen syy järjestelmän "hyytymiseen" tai "hidasteluun" silloin, kun ohjelmia on käynnissä monta yhtäaikaa tai kun pitkään "alapalkissa pienennettynä ollut" ohjelma klikataan taas aktiiviseksi. Pahimmillaan heittovaihto voi välimuistin tavoin ruuhkautua (engl. *"swap thrashing"*), mikä on massamuistin hitauden takia vielä kivuliaampi tilanne kuin välimuistin ruuhkautuminen.

Ylläpitäjä pystyy määrittelemään rajoituksia heittovaihdon tiheydelle ja massamuistista käytettävän heittovaihtotilan määrälle. Järjestelmä saadaan toimimaan tasaisesti maksiminopeudella poistamalla heittovaihto kokonaan käytöstä, mutta uusien ohjelmien käynnistäminen epäonnistuu latausvaiheessa, mikäli niille ei löydy muistista riittävästi tilaa.

Dynaamiset muistinvaraukset (esim. olioiden luonti) voivat kasvattaa olemassaolevan prosessin tarvitsemaa muistitilaa, ja tietysti myös olion luonti epäonnistuu, kun muisti on täynnä. Oliokielissä seuraa jonkinlainen ”OutOfMemoryException” -poikkeus. C-kielessä ei vastaavaa poikkeuksenhallintaa ole, joten sovellusohjelman pitäisi tarkistaa aina jokaisen dynaamisen muistinvarauksen jälkeen, onnistuiko operaatio, ja käsitellä muistin loppuminen tarkoituksenmukaisella tavalla[61].

Leluesimerkki: yksitasoiset sivutaulut ja kehystaulu

Käsitteen ymmärtäminen lienee helpointa jatkamalla aiempaa yksinkertaista ”leluesimerkkiä”. Oletetaan siis aiemman osoitteenmuunnosesimerkin mukaisesti, että muistiosoitteen pituus on 20 bittiä, jossa 8 eniten merkitsevää ovat sivunumero ja 12 vähiten merkitsevää ovat tavun osoite sivun sisällä. Fyysiset sivunumerot ovat 12-bittisiä, tuottaen 24-bittisen fyysisen osoiteavaruuden.

Kuvassa 0.30 esitetyillä rakenteilla periaatteessa pärjättäisiin, paitsi että jaettua muistia ei pystyisi hoitamaan, kun sivukehyksen

[61]Kuinka moni Ohjelmointi 2 -kurssin käynyt opiskelija muuten toteutti harjoitustyöhönsä käsittelyn muistin täyttymiselle? Kuinka moni nimeltämainitsemattomista ulkoistusmaista hankittu halpa koodari toteutti kalliilla rahalla myytävään bisnessovellukseen käsittelyn muistin täyttymiselle? Asiaa kannattaa reflektoida ja kehittää ainakin itsestään sellainen ohjelmantekijä, joka tiedostaa mahdolliset poikkeustilanteet ohjelman suorituksessa ja jopa tekee riittävät ja käyttötarkoitusta vastaavat käsittelykoodit. Ohjelmointi 2 -harkan käyttötarkoitushan tietysti oli läpäistä kyseinen kurssi :).

omistajia on tässä rakennelmassa vain yksi.

Esimerkkiin on poimittu joitakin rivejä kahden eri prosessin (PID 2 ja 7) sivutauluista. Järjestelmässä on toki samaan aikaan suorituksessa muitakin prosesseja, ilmeisesti ainakin PID:t 234 ja 876. Prosessi 2 on ilmeisesti saanut käyttöönsä ainakin kolme sivua virtuaalimuistia, yhteensä siis $3 \times 4096 = 12288$ tavua. Tällä hetkellä näistä kuvassa näkyvistä sivuista on keskusmuistissa kaksi eli 8192 tavua. Kolmas sivu pitäisi hakea massamuistista, jos prosessissa tulisi muistiviittaus puuttuvan sivun osoitteeseen, vaikkapa osoitteeseen 0x03456, jonka alkuosan 0x03 perusteella prosessori havaitsisi sivutaulun riviltä nollatun muistibitin.

Prosessi 7 puolestaan on saanut virtuaalimuistia käyttöönsä ainakin kuvassa näkyvät viisi sivua, yhteensä $5 \times 4096 = 20480$ tavua. Tällä hetkellä keskusmuistissa on näistä kolme sivua. Muut kaksi ovat heittovaihdon päässä massamuistissa. Esimerkiksi prosessin 7 tekemä viittaus muistiosoitteeseen 0x02400 tai 0x04444 aiheuttaisi nyt sivunvaihtokeskeytyksen. Sen sijaan viittaus osoitteeseen 0x03333 tapahtuisi prosessorissa salamannopeasti, kartoittuen fyysisen muistin osoitteeseen 0x436333. Mikäli kyseessä olisi kirjoitusoperaatio, muuttuisi sivutaulun vastaavalla rivillä ”dirty”-bitti.

Kehystaulussa on ”seinäkelloaikana” jonkinlainen historialaskuri. Mikäli kaikki kehykset olisivat täynnä, pitäisi kehystaulusta etsiä se rivi, jossa laskurin arvo on suurin. Näin löytyisi kauimman aikaa sitten käytetty kehys, jonka sisältö pitäisi korvata lukemalla tilalle sivunvaihtokeskeytyksen aiheuttanut sivu.

Tässä esimerkissä esimerkiksi fyysisen sivun 0x335 käytöstä on kulunut jo merkittävä aika, joten se saattaisi joutua korvatuksi. Silloin täytyy mennä tutkimaan omistajaprosessin sivutaulusta, onko tämänhetkinen sivu likainen. Tässä tapauksessa ei näytä olevan,

joten tilalle saa suoraan lukea uuden sivun sisällön. Prosessin 2 tietoihin täytyy käydä merkitsemässä, että aiemmin keskusmuistissa ollut sivu ei enää olekaan siellä (ts. ”muistibitti” pitää käydä laittamassa nollaksi ja tarpeettomaksi käyneen fyysisen sivunumeronkin voi nollata).

Myös fyysisen sivun 0x345 käytöstä on kulunut paljon aikaa, joten sekin saattaisi pian olla pudotusuhan alla, kun heittovaihtoa tarvittaisiin. Jälleen kehystaulun tietojen perusteella päästäisiin näkemään sivutaulun rivi, josta pääteltäisiin sivun likaisuutta. Tässä tapauksessa pääteltäisiin, että kyseistä sivua on muokattu muistissa (”dirty”-bitti on ykkönen). Se pitäisi siis tallentaa massamuistiin kohtaan, joka löytyy jonkinlaisen indeksinumeron 0x00bc perusteella (yksinkertaisimmillaan esimerkiksi 0x00bc000 tavua heittovaihtoa varten varatun levyosion alusta lukien; sivut voisivat sijaita levyllä peräjälkeen, ja niiden alkukohdillahan on tässä neljän kilon sivuja käyttävässä esimerkissä välimatkaa 0x1000 tavua; sivun ensimmäisen tavun osoite löytyy lisäämällä nollia heksalukuna ilmoitetun järjestysnumeron perään, aivan kuten keskusmuistin kehystenkin kohdalla).

Potentiaalinen tenttikysymys on antaa tämän kaltainen leluesimerkki ja esittää siitä seuraavanlaisia (tai muita esimerkkitaulukoiden avulla selvitettävissä olevia) kysymyksiä:

- Mihin fyysiseen muistiosoitteeseen kohdistuisi prosessin 2 tekemä kirjoitus virtuaalimuistiosoitteeseen 0x7f123? (Vastaus: 0x345123)
- Mihin fyysiseen muistiosoitteeseen kohdistuisi prosessin 7 tekemä luku virtuaalimuistiosoitteesta 0x7f123? (Vastaus: 0x4001
- Prosessi 2 suorittaa hyppykäskyn aliohjelmaan muistiosoit-

teessa 0x03200. Tapahtuuko prosessorissa sivunvaihtokeskeytys? (Vastaus: kyllä)

- Prosessi 2 suorittaa hyppykäskyn aliohjelmaan muistiosoitteessa 0x03200. Kello- tai I/O-laitekeskeytyksiä ei ole tullut. Noutaako prosessori seuraavan käskyn kyseisestä osoitteesta? (Vastaus: ei)

- Prosessi 7 suorittaa hyppykäskyn aliohjelmaan muistiosoitteessa 0x03200. Tapahtuuko prosessorissa sivunvaihtokeskeytys? (Vastaus: ei)

- Prosessi 7 suorittaa hyppykäskyn aliohjelmaan muistiosoitteessa 0x03200. Kello- tai I/O-laitekeskeytyksiä ei ole tullut. Noutaako prosessori seuraavan käskyn kyseisestä osoitteesta? (Vastaus: kyllä)

- Prosessissa 7 aiheutuu sivunvaihtokeskeytys ja fyysinen muisti on täynnä; korvausalgoritmi on LRU. Pitääkö fyysisen sivun 0x437 sisältö tällöin tallentaa levylle (Vastaus: ei)

- Fyysinen sivu 0x335 joudutaan korvaamaan uudella sivunvaihtokeskeytyksessä. Onko sen sisältö tallennettava massamuistiin ennen korvaamista? (Vastaus: ei)

Yhteenveto: Tyypillinen tapa toteuttaa prosesseille oma virtuaalimuistiavaruus on sivuttava virtuaalimuisti, jossa muistiavaruus jaetaan ns. sivuihin. Laitteisto (prosessori ja MMU) muuntaa virtuaaliosoitteet fyysisiksi muistiosoitteiksi sivukohtaisesti: Osoitteen alkuosa kartoittuu fyysisen muistin osoitteen alkuosaksi, johon loppuosa liitetään sellaisenaan. Samalla periaatteella hoituu muistialueiden erilaiset suojaukset sekä jakaminen eri prosessien kesken. Käytettävissä oleva fyysinen muisti jaetaan sivun kokoisiin kehyksiin, jollaisessa virtuaalimuistin sivun sisältöä pidetään.

Muistissa säilytettävät sivut muodostavat ns. työjoukon. Siirtämällä työjoukossa hetkellisesti tarpeettomia sivuja talteen massamuistiin saadaan käyttöön enemmän virtuaalimuistia kuin koneeseen on asennettu fyysistä muistia. Tämän ns. heittovaihdon käytännön toimivuus perustuu havaintoon, että tyypilliset ohjelmat tarvitsevat yleensä suhteellisen pitkiä aikoja tiettyä peräkkäisten osoitteiden aluetta ennen kuin niiden tarvitsee siirtyä eri alueelle (lokaalisuusperiaate). Sama ilmiö on myös prosessorilaitteiston välimuistijärjestelmän ja osoitteenmuunnosta nopeuttavan TLB:n tausta-ajatuksena.

```
Prosessin (PID=2) sivutaulun joitakin rivejä:
+----------------------------------------------+
|     virt.  fyys.  muist.  dirty   diskIndex|
|     sivu   sivu   (P)       (D)            |
+----------------------------------------------+
|     0x02   0x335   1         0      0x0010 |
|     0x03   0x000   0         0      0x0022 |
|     0x7f   0x345   1         1      0x00bc |
+----------------------------------------------+

Prosessin (PID=7) sivutaulun joitakin rivejä:
+----------------------------------------------+
|     virt.  fyys.  muist.  dirty   diskIndex|
|     sivu   sivu   (P)       (D)            |
+----------------------------------------------+
|     0x02   0x000   0         0      0x0004 |
|     0x03   0x436   1         0      0x0008 |
|     0x04   0x000   0         0      0x00f2 |
|     0x2e   0x47e   1         1      0x006c |
|     0x7f   0x400   1         1      0x0007 |
+----------------------------------------------+

Järjestelmän kehystaulun joitakin rivejä:
+--------------------------------------------------------+
| fyys.  omistajan  omistajan    aikayksiköt edellisen |
| sivu    PID        PTE#        käyttökerran jälkeen  |
+--------------------------------------------------------+
| 0x335     2           0x02         195                 |
| 0x345     2           0x7f         155                 |
| 0x436     7           0x03         7                   |
| 0x437     234         0x45         8                   |
| 0x47e     7           0x2e         12                  |
| 0x47f     876         0x45         4                   |
+--------------------------------------------------------+
```

Kuva 0.30: *Sivutaulujen ja kehystaulun käyttöä: leluesimerkki ja perinteinen tenttitärppi.*

0.11 Oheislaitteiden ohjaus

Avainsanat: laiteohjain, ajuri, DMA-järjestelmä, laitteistoriippumaton ja laiteriippuva I/O-ohjelmisto, I/O -operaation vaiheet; kovalevyn rakenne: ura, sektori, sylinteri, hakuaika; RAID

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa kuvailla laiteohjelmistojen kerrosmaisen rakenteen ja I/O-pyynnön sekä I/O-keskeytyksen käsittelyn yleisellä tasolla [ydin/arvos1]
- osaa kuvailla tiedon pitkäaikaiseen tallennukseen ja lataamiseen liittyvät vaiheet laitteiston ja ohjelmiston osien toiminnan tasolla [ydin/arvos2]
- tietää DMA:n toimintaperiaatteen hyötyineen ja implikaatioineen [ydin/arvos2]
- tietää RAIDin (vähintään muutamien eri tasojen) toimintaperiaatteet hyötyineen ja implikaatioineen [edist/arvos3]

Laitteiston piirteet ja laiteriippuvan I/O -ohjelmiston tehtävät

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

- Ne on liitetty prosessoriin ja muistiin väylän kautta.
- Laitteiden toimintajakso on erilainen kuin prosessorin – se on paljon hitaampi ja usein jopa satunnainen (esim. käyttäjän klikkailut ja näppäinpainallukset).

- Laitteita on monia erilaisia, moniin eri tarkoituksiin. Niitä voidaan kategorisoida esimerkiksi tiedonsiirtotavan mukaan **merkkilaitteisiin** (engl. *character device*) ja **lohkolaitteisiin** (engl. *block device*); näppäimistö on merkkilaite, koska sieltä saapuu dataa yksi merkki kerrallaan; kovalevy on lohkolaite, koska siellä luonnostaan on peräkkäin paljon dataa, jota on luontevaa lukea määrämittainen peräkkäinen pötkö eli **lohko** (engl. *block*) kerrallaan.

- I/O -laitteet ovat alttiita häiriöille. Mekaaniset komponentit ovat alttiimpia vikaantumaan kuin esim. prosessorin elektroniset komponentit; lisäksi I/O-laitteet ovat usein liitäntäjohdon päässä, mistä aiheutuu mm. johdon yllättävän irtoamisen vaara. Toimivuuden tarkkailu, virheenkorjaus ja selviytymiskeinot vian ilmetessä 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 laiteohjain kommunikoi yhdelle tai useammalle fyysiselle laitteelle. Prosessorilta voi antaa laiteohjaimen kontrollilogiikalle väylän kautta komentoja, jotka ovat tavuiksi 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:lla ja lähetettävä lopullisena ääniohjaimelle. Nykyisille näytönohjaimille toimitetaan ohjauskomentojen lisäksi määrämuotoista dataa 3D-geometrioista ja pintamateriaalien kuvioinneista sekä näytönoh-

jaimen ymmärtämälle konekielelle käännettyjä rinnakkaislaskentaohjelmia, jotka hoitavat varsinaisen piirtämisen.

Laitteiden monimuotoisuuden vuoksi tarvitaan niin sanottu **laiteriippuva ohjelmiston osa**, joka viimekädessä ohjaa fyysisen laitteen ohjainta. Jokaisen laitteen tai laitetyypin käyttämiseksi tarvitaan yleensä **laiteajuri** (engl. *device driver*), joka osaa muuntaa sovellusohjelmoijan ymmärtämässä, yleensä standardoidussa, muodossa annetut laiteohjauskomennot laitteen oman rajapinnan edellyttämään muotoon. Kenties monipuolisin esimerkki ovat nykyisten näytönohjaimien ajurit, jotka muuntavat esimerkiksi OpenGL- tai DirectX-ohjelmointirajapintojen mukaisen grafiikkakoodin laitevalmistajakohtaisiksi laitekomennoiksi. Ajureiden täytyy sisältää jopa kääntäjä, joka hoitaa varjostinkielellä kirjoitettujen koodien eli shaderien kääntämisen GLSL- tai HLSL-kielestä näytönohjaimen rinnakkaisprosessorien ymmärtämälle konekielelle. Pidemmällä aikavälillä joidenkin yksinkertaisempien ja yleisempien laitteiden rajapinnat on standardoitu siinä määrin, että sama ajuri voi ohjata minkä tahansa valmistajan laitetta. Mm. näppäimistöt, hiiret, kovalevyt ja useat USB:n kautta ohjattavat laitetyypit kuuluvat näihin.

Laitekohtainen, mahdollisesti laitevalmistajan toimittama, ajuriohjelmisto siis muodostaa varsinaiset komennot I/O -laitteelle. Muodostaminen on mahdollista käyttäjätilassa (jos ajureita ei haluta turvallisuussyistä suorittaa käyttöjärjestelmätilassa ja hyväksytään IPC:stä aiheutuva suorituskyvyn heikkeneminen). Vähintään komentojen lähettäminen väylän kautta on kuitenkin tehtävä käyttöjärjestelmätilassa, joten käyttöjärjestelmän ytimeen kuuluu kiinteästi kaikki se koodi, jolla yksilöidylle I/O-laitteelle lähetetään ajurin valmistelema laitekomento. Tyypillisesti I/O -laitteelle annettavat komennot ovat yksinkertaisia pyyntöjä kirjoittaa tai lukea jotakin, merkkilaitteissa tavu kerrallaan (suoraan annettuna

arvona), lohkolaitteissa puolestaan isompi pätkä kerrallaan (suoraan muistiosoitteen ja siirrettävän pätkän pituuden perusteella). Prosessoriarkkitehtuurissa voi olla erillinen konekielikäsky laitekomentojen lähetykseen, tai sitten osa fyysisestä muistiavaruudesta voi kartoittua keskusmuistin sijasta laitteiden ohjausportteihin.

Suuremman datamäärän siirron tekee usein väylään liitetty **DMA**-järjestelmä (engl. *Direct memory access*), joka osaa käyttää väylää itsenäisesti tavujonon kopioimiseksi muistin ja I/O-laitteiden välillä. DMA:n ansiosta prosessori vapautuu suorittamaan laskentaa tarvitsevien ohjelmien koodia koko tiedonsiirron ajaksi, joten se nopeuttaa kokonaisuuden toimintaa merkittävästi. Täysin ilmaiseksi DMA ei työtänsä tee, koska väylä on kaikille käyttäjilleen yhteinen ja se lukittuu laitteistotasolla kahden komponentin välille jokaisen siirron yhteydessä. Muut joutuvat odottelemaan, että väylä vapautuu. DMA:n datasiirto aiheuttaa tätä kautta ”kellojaksovarkauksia” (engl. *cycle stealing*), eli prosessori ei välttämättä pääse käyttämään muistia välittömästi, mikäli väylällä on juuri sillä hetkellä siirtymässä jotakin DMA:n autonomisesti hoitamaa liikennettä. Prosessorin välimuistit luonnollisesti auttavat pienentämään ulkoisen väylän ruuhkaa.

Laitteistoriippumattoman I/O -ohjelmiston tehtävät

Korkeammalla abstraktiotasolla tarvitaan ns. **laitteistoriippumaton I/O-ohjelmisto**, jonka tehtävä on luoda yhtenäinen tapa käyttää moninaisia laitteita (*uniform interfacing*). Tällaiselta yhtenäiseltä tavalta vaaditaan seuraavaa:

- tiedostojärjestelmä (file system): datan looginen organisointi tiedostoiksi ja hakemistoiksi, joilla on käyttäjän kannalta

ymmärrettävät nimet

- laitteiden nimeäminen (device naming): millä tavoin käyttäjän prosessi voi tietää fyysisesti asennettuna olevat I/O-laitteet ja päästä niihin käsiksi

- käyttöoikeudet (protection): käyttäjillä on 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 toiminta ei sotkeudu kilpajuoksutilanteiden vuoksi

- 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

- puskurointi (buffering): esim. peräkkäisten näppäinpainallusten tallennus pidemmäksi merkkijonoksi tai yhden yksittäisen tavun lukeminen kovalevyltä, joka antaa kuitenkin aina vähintään sektorin kerrallaan ulos

- yhtenäinen ajuriliitäntä (device driver interface): erilaisten laitteiden valmistajien, tai ainakin ajurien tekijöiden, täytyy tietää millaisen rajapinnan toteuttamista käyttöjärjestelmä vaatii ajuriohjelmistolta.

Kuva 0.31: *I/O -operaatioon osallistuvat kerrokset, niiden väliset rajapinnat, ja operaation suoritusvaiheet ja osapuolet aikajärjestyksessä (1–7).*

I/O -kutsun ja I/O -tapahtuman vaiheet

Käyttöjärjestelmän I/O:ta hoitavan ohjelmakoodin tyypillinen kerrosmainen rakenne on esitetty kuvassa 0.31. Tapahtumien kulku tavallisessa I/O -operaatiossa on ohjelmistokerrosten, prosessorin ja oheislaitteiston näkökulmista seuraavanlainen:

1. Sovellusohjelman kutsu päätyy alustakirjaston kautta vaiheeseen, jossa tarvitaan syöttöä tai tulostusta jotakin käyttöjärjestelmän tarjoamaa I/O -menettelyä käyttäen.

2. Siirtyminen käyttöjärjestelmäkoodin suoritukseen tapahtuu alimmalla kirjastotasolla normaalisti ohjelmoidun keskeytyksen (”syscall”) kautta. (kuvan 1. nuoli)

3. Laitteistoriippumaton I/O -ohjelmisto tulkitsee pyynnön, ja saattaa hyödyntää tiedostojärjestelmän, muistinhallinnan ja joidenkin laitteistoa korkeammalla tasolla kuvailevien abstraktiokerrosten algoritmeja. Viime kädessä tullaan tarvitsemaan laitekohtaisten ajurien algoritmeja ohjauskomennon muun-

tamiseen laitteen ymmärtämään muotoon. Käyttöjärjestelmässä on sovittu, millä tavoin siirtyminen laiteajurin puolelle tapahtuu, ja laiteajurin on tietenkin kunnioitettava tätä ns. **ajurirajapintaa**. (kuvan 2. nuoli)

4. Lopulta käyttöjärjestelmän I/O:ta hoitava ohjelmakoodi antaa I/O -laitteen kontrollilogiikalle ajuriohjelmiston valmisteleman yksinkertaisen käskyn (tyypillisesti lue/kirjoita + lähteen ja kohteen sijaintitiedot) ulkoisen väylän kautta. (kuvan 3. nuoli)

5. Koska fyysisen I/O -toiminnon kestoa ei tiedetä, I/O:ta tarvinneen ohjelman prosessi on syytä laittaa odottelemaan (blocke tilaan) ja päästää jokin ready-tilassa oleva prosessi suoritusvuoroon.

6. I/O -laite tekee autonomisesti fyysisen toimenpiteensä eli lukee tai kirjoittaa (tai skannaa tai tulostaa tai liikuttaa moottoreita tmv.). Samaan aikaan prosessori voi vuorontaa muita prosesseja normaalisti.

7. Valmistuttuaan I/O -laite antaa prosessorille keskeytyksen, jolloin laitekohtainen keskeytyskäsittelijä päätyy suoritukseen. (kuvan 4. nuoli)

8. Laitetapahtuma saattaa jälleen edellyttää laiteriippuvan ajuriohjelmiston toimenpiteitä laitteelta tulevan kommunikaation tulkitsemiseksi. (kuvan 5. nuoli)

9. Ajurin tulee muokata laitteelta tullut kommunikaatio ajurirajapinnan mukaisesti muotoon, jossa käyttöjärjestelmän laiteriippumaton osio pystyy sitä käsittelemään. (kuvan 6. nuoli)

10. käyttöjärjestelmän laitteistoriippumaton I/O -koodi voi (jälleen yhteistyössä käyttöjärjestelmän muiden osioiden kanssa) jatkaa tarvittavilla toimenpiteillä, eli mm. siirrellä odottavan prosessin taas blocked-tilasta suoritusvalmiiksi ja valmistella sille paluuarvon, jossa on tieto operaation onnistumisesta tai mahdollisesta epäonnistumisesta.

11. Suoritusvuoron saadessaan käyttäjän prosessi jatkuu I/O -kutsun aiheuttaneen ohjelmoidun keskeytyksen jälkeisestä konekielikäskystä; koodi on yleensä vielä alustakirjaston, ei sovelluksen koodia. (kuvan 7. nuoli)

12. Muutaman aliohjelmapaluun jälkeen sovellusohjelma jatkuu I/O -operaatiota edustaneen kirjastokutsun lähdekoodiriviltä esimerkiksi tallentamalla kutsun paluuarvon paikalliseen muuttujaan erilaisten virheiden tarkistamista varten.

I/O -operaatio voi herkästi epäonnistua ulkomaailmaan liittyvistä syistä, joten sovellusohjelman on syytä aina tarkistaa operaation onnistuminen.

Edellä kuvattu sovellusohjelmoijan pyytämän I/O -operaation suoritus kulkee ohjelmistokerrosten läpi kahteen suuntaan: Käyttäjän prosessi käyttää käyttöjärjestelmän kutsurajapintaa. Laitteistoriippumaton I/O-ohjelmisto abstrahoi alla olevat laitteet ja delegoi varsinaisen ohjauksen laiteriippuvalle ohjelmistolle ja viime kädessä ajureille, jotka tuntevat itse laitteiston rajapinnan.

Jotkut I/O -tapahtumat voivat tietysti ilmetä ilman, että kukaan varsinaisesti odotti niitä: esimerkiksi lisämonitorin liitin kiinnitetään, verkkoyhteydestä saapuu dataa, näppäintä painetaan, tai johonkin laitteeseen tulee äkillinen toimintahäiriö, josta se haluaa ilmoittaa. Tällöinkin aiheutuu aivan normaali laitekeskeytys. Ku-

van 0.31 esittämistä nuolista silloin tapahtuvat kuitenkin vain 4–7. Laiteohjelmisto käsittelee keskeytyksen aiheuttaneen tilanteen tarkoituksenmukaisella tavalla, ja käyttöjärjestelmä päättää mihin prosessiin suoritus palaa. Yksinkertaisimmillaan esimerkiksi näppäinpainallus tai hiiren liikahdus laitetaan puskuriin odottamaan myöhempää käsittelyä ja palataan saman tien suorittamaan juuri keskeytynyttä prosessia. Toisaalta, jos keskeytys johtui vaikkapa ääniohjaimen puskurin tyhjentymisestä, voi olla syytä vaihtaa prosessiksi audiopalvelin, jonka vastuulla on täyttää puskuri välittömästi uudella äänidatalla, ettei ulostulo ehdi hetkeksikään katketa häiritsevästi.

Kovalevyn rakenne

Tutustutaan kovalevyn rakenteeseen toisaalta yhtenä käytännön esimerkkinä I/O-laitteesta ja toisaalta valmisteluna tiedon tallentamiseen liittyvän järjestelmämoduulin eli tiedostojärjestelmän 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). Pää liikkuu puomin varassa levyn sisä- ja ulkoreunan välillä.

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). Yksittäisten tavujen käsittely levyn pinnassa ei siis ole mahdollista.

- Sektori sisältää tarkistebittejä virheiden havaitsemista ja korjaamista varten[62].
- Fyysisen levyn osoitteet ovat sektorikohtaisia.
- Tyypillisesti ohjelmisto (esim. käyttöjärjestelmän tiedostonhallinta) niputtaa muutamia peräkkäisiä sektoreita lohkoiksi (block).

Käyttöjärjestelmä voi esimerkiksi käyttää 4096 tavun mittaista lohkoa, jolloin se tallentaa ja lataa fyysiseen kovalevyyn aina vähintään 8 sektoria, mikäli sektorin pituus on 512 tavua.

Hakuaikaan (engl. *seek time*) eli siihen, kuinka nopeasti levyltä saadaan luettua jotakin, vaikuttavat:

- lukupään siirtoaika uralta uralle (nopeus riippuu lukupään puomin teknisestä toteutuksesta ja kokonaisaika siitä, kuinka kaukana seuraava ura on edellisestä – tarvittava siirto voi olla millimetrin osien luokkaa tai maksimissaan muutama senttimetri)
- pyörähdysviive, eli kuinka nopeasti uralla oleva sektori pyörähtää lukupään alle (riippuu levyä pyörittävän moottorin nopeudesta; tyypillistä kuluttajatuotteille on 5400 kierrosta minuutissa, jolloin sama kohta kulkee lukupään alta reilun sadasosasekunnin välein)
- siirtoaika sisäiseen puskurimuistiin

[62]Kovalevyssä on oikeasti enemmän sektoreita kuin se ilmoittaa ulkomaailmalle. Se voi automaattisesti poistaa käytöstä vikaantuneita sektoreita ja kopioida virheenkorjausbittien avulla palautetun datan redundanteille varasektoreille. Ohjauslogiikka tarjoaa diagnostiikkakomentoja, joiden avulla jäljellä olevaa käyttöikää voidaan arvioida mm. vikaantuneiden sektoreiden suhteellisen määrän perusteella.

- sisäisen puskurimuistin nopeus ulospäin
- välimuistitus: kovalevy itsessään voi laitteen sisäisesti hyödyntää lokaalisuusperiaatetta (peräkkäisiä sektoreita käytetään ehkä pian uudelleen) ja tarjota datan omasta välimuististaan, ilman tarvetta lukea levyn pintaa jokaisen latauksen yhteydessä.

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

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

Sektorien määrä voi vaihdella levyn eri osien välillä, mikä hieman mutkistaa laskutoimituksia todellisuudessa, ja aiheuttaa ilmiön, että levyn ulkolaidalla data voi olla käsiteltävissä nopeammin kuin sisälaidalla.

Viime aikoina yleistyneet SSD-levyt (solid state drive) eivät sisällä pyöriviä osia vaan puolijohteista valmistettuja muistikomponentteja, joiden sisältö säilyy ilman jatkuvaa sähkövirtaa. Niissä hakuaika on vakiomittainen ja hyvin lyhyt verrattuna pyörivään levyyn. Muutoin SSD näyttää ulospäin hyvin samanlaiselta I/O -laitteelta kuin magneettilevykin (sopii mm. samaan liittimeen ja tarjoaa hyvin samantyyppisen laitekomentorajapinnan).

RAID-järjestelmät

Kovalevyt ovat mekaanisten osiensa vuoksi herkkiä vikaantumaan. SSD:t kestävät töyssyjä paremmin, mutta niiden muistikomponentit kuoleutuvat ajan mittaan luonnostaan. Massamuistin käyttö sisältää aina epävarmuustekijän: data tai sen osia voi milloin ta-

hansa tuhoutua. Tärkeiden tietojen ajoittainen varmuuskopiointi useampaan fyysiseen sijaintiin on erittäin tärkeää.

Mikäli datan täytyy pysyä ehjänä jatkuvasti, myös varmuuskopiointien välillä, voidaan toimintavarmuutta lisätä merkittävästi käyttämällä tietojen reaaliaikaista kahdentamista saman laitteiston sisällä. Samalla voidaan saada lisäystä myös suorituskykyyn.

Alkujaan 1970-luvulla kehitetty tekniikka nimeltä **RAID** (engl. *redundant array of inexpensive/independent disks*) perustuu kahden tai useamman erillisen kovalevyn käyttöön saman tiedon tallentamiseen rinnakkain erilaisin tavoin siten, että kokonaisuuden toimintavarmuus, nopeus tai molemmat ovat paremmat kuin yhtä kovalevyä käytettäessä. Osa erilaisista RAID-tekniikoista on sittemmin standardoitu ja numeroitu niin sanotuiksi ”tasoiksi”:

- RAID 0: Tallennettava datapötkö jaetaan kiinteän mittaisiin ”juoviin” (engl. *stripe*), jotka hajautetaan tasaisesti eri levyille. Dataa ei siis kahdenneta, vaan sen tallennus hajautetaan: esimerkiksi joka toinen pätkä yhdelle ja joka toinen toiselle levylle. Tällä saadaan lisäys kapasiteetin lisäksi suorituskykyyn, koska datan lappaminen väylän kautta on nopeampaa kuin mitä yksittäinen levy pystyy ottamaan vastaan tai lukemaan. Toimintavarmuus *ei* lisäänny, vaan itse asiassa hieman kärsii: yhden levyn hajoaminen jättäisi myös kaikkien muiden hajautuksessa käytettyjen levyjen datat käyttökelvottomiksi (osia puuttuu välistä, kun ne olivat hajonneella levyllä).

- RAID 1: Data ”peilataan”, eli jokaiselle levylle tallennetaan identtinen kopio datasta. Jo kahdella levyllä saadaan suuri toimintavarmuus: Yhden levyn hajotessa ei tarvitse kuin vaihtaa uusi tilalle ja tehdä uusi peilikuvakopio ehjänä säily-

neeltä levyltä. Lukuoperaatiot voidaan hajauttaa, joten niissä saadaan nopeutus samalla tavoin kuin RAID 0:ssa. Kirjoitusoperaatiot täytyy tehdä samanlaisina kahteen kohteeseen, joten kirjoitus on jopa hivenen hitaampaa kuin yhdellä levyllä. Kapasiteettia RAID 1 ei tuo lisää, vaan toinen ja useammat lisälevyt ovat ylimääräisiä (”redundantteja”) apuvälineitä toimintavarmuuden saavuttamiseksi.

- RAID 2: (Ei käytössä enää nykyään, koska hyöty osoittautui muita tasoja pienemmäksi) Data hajautetaan eri levyille bitti kerrallaan ja käytetään lisälevyä virheenkorjaustiedolle (bitittäinen tarkistussumma). Levyt ovat keskenään samanlaisia ja niitä pyöritetään fyysisesti synkronoituna: lukupää on jokaisessa levyssä samassa kohtaa. Kapasiteetti lisääntyy levyjen määrän kasvaessa - vain virheenkorjaustiedon tarvitsema tila on pois käyttökelpoisesta kapasiteetista.

- RAID 3: (Harvoin enää käytössä) Kuten RAID 2, mutta data hajautetaan tavu kerrallaan.

- RAID 4: (Harvoin enää käytössä) Kuten RAID 2, mutta data hajautetaan lohko kerrallaan, eivätkä levyt ole synkronoitu samaan pyörimiskohtaan.

- RAID 5: (Nykyinen korvaaja RAID 2-4:n ideoille) Lohkot hajautetaan eri levyille. Datasta lasketaan virheenkorjaustieto, joka myös hajautetaan eri levyille. Tarvitsee vähintään kolme levyä. Virheenkorjaustiedon perusteella kaikki tiedot on täysin palautettavissa, mikäli kerrallaan rikkoutuu vain yksi levy. Luku- ja kirjoitusoperaatiot nopeutuvat hajautuksen ansiosta ja kapasiteetti kasvaa levyjä lisäämällä.

- RAID 6: Kuten RAID 5, mutta tallentaa myös virheenkorjaustiedon hajautetusti. Lopputuloksena kaksi levyä voi ol-

la hajalla ilman lopullista datan tuhoutumista. Suorituskyky on muuten melkein yhtä hyvä kuin RAID 5:ssä, mutta virheenkorjaustieto on raskaampi laskea.

Nykyään käytössä olevien RAID-tasojen edellyttämä levyjen koordinointi voidaan hoitaa ohjelmistolla, ja käyttöjärjestelmä voi tukea RAIDia tietokoneeseen asennettuja kovalevyjä käyttäen. Tällöin kuitenkin virheenkorjaustieto on laskettava CPU:ssa, mikä ottaa oman aikansa. Tehokkaampi ratkaisu on erillinen RAID-laite, joka hoitaa virheenkorjaustiedon laskennan omassa sisäisessä prosessorissaan. Tällainen ulkoinen laitteisto-RAID näyttäytyy tietokoneen kannalta jotakuinkin yhtenä tavallisena, mutta luultavasti hyvin nopeana ja suurena kovalevynä.

RAIDia tai vastaavaa hajautusta hyödyntävät laajemmat **tallennusjärjestelmät** (engl. *storage system*) ovat hyvä teknologia mm. **verkkolevyiksi** (engl. *network attached storage, NAS*), jossa voi olla teratavuittain virtuaalista kovalevytilaa jaettavissa verkon yli useisiin koneisiin.

Yhteenveto: Syöttö- ja tulostuslaitteiden eli I/O -laitteiden ohjaimet (tai sovittimet) on liitetty prosessoriin ja muistiin väylän kautta. Välissä oleva käyttöjärjestelmän ohjelmisto on jaettu sisäisesti kerroksiin. Korkeamman tason ohjelmisto tarjoaa yhtenäiset nimeämiskäytänteet, yhteisen koon käsiteltäville lohkoille sekä rajapinnan sekä sovelluksille että ajureille. Matalan tason ohjelmisto hoitaa kommunikoinnin laiteohjaimien kanssa. Ohjaimet ohjaavat itse fyysisiä laitteita (optoelektronista, elektromekaanista tmv. ”kalustoa”).

Fyysisen I/O:n kestoa ei tiedetä, joten I/O:ta tarvitseva ohjelma laitetaan odottamaan, kunnes operaatio on valmis. I/O-laite ilmoittaa valmistuneista tai yllättävistä tapahtumista prosessorille

keskeytyspyynnöllä. Kaikki I/O -ohjelmiston kerrokset osallistuvat operaation tekemiseen.

Laitteet ovat alttiita mekaanisille häiriöille ja ulkoisille poikkeustilanteille, joten I/O -operaation onnistumiseen ei saa luottaa sovellusohjelman toimintalogiikassa, vaan virheet on tarkistettava. Kovalevyjen vikaantumista voidaan hallita redundanttiin tallennukseen perustuvaa RAID-järjestelmää käyttämällä.

0.12 Tiedostojärjestelmät

Avainsanat: tiedostojärjestelmä, tiedosto-osoitteet, käyttöoikeudet, i-numero, i-solmu, deskriptoritaulu, tiedostotaulu, (un)mounttau journalointi

Osaamistavoitteet: Luvun luettuaan opiskelija:

- ymmärtää erikoistiedostojen roolin ja "kaikki ilmenee tiedostona -periaatteen"unix-tyyppisissä järjestelmissä; osaa antaa esimerkkejä unixin erikoistiedostoista [edist/arvos3]
- osaa kertoa, miksi naiivisti toteutettu tiedostojärjestelmä voi korruptoitua sähkövirran katketessa ja kuvailla keinot, joilla korruptoitumista voidaan ehkäistä (journalointi) [edist/arvos3]
- osaa selittää yhden konkreettisen tavan organisoida tiedostojärjestelmä levyllä (unix i-nodet) [edist/arvos4]

Tiedostojärjestelmän rajapinta C-kielen tasolla

Tiedostojärjestelmän matalan tason käyttöjärjestelmäkutsuja ovat esimerkiksi seuraavat POSIXin määrittämät kutsut:

```
creat()   - luo tiedoston
open()    - avaa tiedoston ja tarvittaessa lukitsee
read()    - lukee tiedostosta nykyisen tiedosto-osoittimen kohdalta
close()   - "sulkee" tiedoston, ts. vapauttaa lukituksen
mmap()    - kartoittaa tiedoston prosessin virtuaalimuistiin
```

Tarkempi katsaus kutsun mmap() rajapintaan osoittaa, että prosessin muistiin liitettävä jaettu muistialue ilmenee samanlaisena tiedostona kuin esimerkiksi kovalevyllä sijaitseva tekstitiedosto. Tämä on tyypillistä Unixista polveutuvissa järjestelmissä, jollaisen

POSIX-standardikin määrittelee. Erilaisten asioiden samaistaminen tiedostoiksi yhdenmukaistaa mm. niiden nimeämistä, osoitteiden antamista ja operaatioiden kohdistamista niihin. Jaettu muistialue toimii tietysti teknisesti hyvin eri tavoin kuin kovalevylle kirjoitettu tiedosto. Rajapinnan yläpuolella ne käyttäytyvät kuitenkin aika samanlaisella tavalla: molemmat ovat ”datapötköjä”, joilla on alku ja loppu sekä käyttöoikeuksien määrittämä mahdollisuus lukea tai kirjoittaa joko indeksoituja tavuja (”satunnaissaanti”, *random access*) tai esimerkiksi putkien osalta ajan suhteen järjestettyä virtaa (**FIFO**, engl. *first in - first out*.

Matalimman tason POSIX-kutsuissa tiedostoja käsitellään niin sanottujen **deskriptorien** (engl. *file descriptor*) kautta. Ne ovat kokonaislukuna ilmeneviä ”kahvoja”, jotka yksilöivät prosessin käytössä olevan tiedoston (joka voi siis käytännössä olla esimerkiksi kovalevyn tiedosto, jaettu muistialue, syöttö- tai tulostusvirta tai oheislaite).

Käyttöjärjestelmä pitää tiedostoista kirjaa sisäisesti; prosessikohtaiset deskriptorit ovat ”viitetietoa”, jonka perusteella järjestelmä osaa kohdistaa operaatiot haluttuihin kohteisiin. Käyttöjärjestelmäkutsut kuten creat(), open(), shm_open() palauttavat deskriptorin, jos luonti tai yhdistäminen olemassaolevaan kohteeseen onnistuu. Kirjoitus-, luku- ja sulkemisoperaatiot ym. kuten read(), write(), mmap(), close() puolestaan tarvitsevat parametrikseen olemassaolevaan tiedostoon liittyvän deskriptorin. Jos sulkeminen eli close() onnistuu, deskriptorinumero luonnollisesti lakkaa viittaamasta mihinkään käyttökelpoiseen.

Edellä mainitut POSIXin määrittämät kutsut toimivat matalammalla tasolla (lähempänä käyttöjärjestelmäydintä ja laitteistoa) kuin C-kielen alustakirjaston tiedostonkäsittelykutsut. Käytännössä, mikäli se riittää käyttötarkoitukseen, on hyvä käyttää C:n alus-

takirjastoa, koska POSIXin (tai Unixin/Linuxin/vastaavan) matalan tason tiedostototeutusta ei välttämättä löydy muista järjestelmistä, joissa C-kielistä ohjelmaa voitaisiin haluta käyttää.

Kaikkein alustariippumattomin C-sovellus käyttäisi tällaisia ”hieman korkamman tason” kirjastokutsuja, jotka operoivat tiedostojen sijasta ns. **tietovirroilla** (engl. *stream*):

```
fopen()   - avaa virran (yhdistäen sen esimerkiksi tiedostoon)
fread()   - lukee virrasta
fprintf() - kirjoittaa virtaan merkkejä nätisti
fclose()  - sulkee virran (ja tiedoston, johon virta liittyi)
```

Esimerkiksi jaetun muistialueen luonti tai atominen tiedoston lukitseminen (ilman että toinen prosessi pääsee ”varastamaan” tiedoston välissä) ei kuitenkaan onnistu C-alustakirjaston tietovirtakutsujen kautta. Joskus on vain pakko käyttää käyttöjärjestelmästä riippuvia järjestelmäkutsuja, mikä johtaa siihen, että C-koodin alustariippumattomuus edellyttää joidenkin pätkien kirjoittamista kaikille alustoille erikseen esikääntäjää hyödyntäen esimerkiksi seuraavalla tavoin:

```
#ifdef KAANNETAAN_WINDOWSILLE
#include<windows.h>
#elif KAANNETAAN_POSIXILLE
#include<unistd.h>
#else
#error Käännöksen kohdealustaa ei ole määritelty.
#endif
...
```

Suurin osa koodista toimisi samalla tavoin, mutta osa käyttöjärjestelmän palveluista, mm. jaetun muistialueen käsittely, pitäisi ”iffitellä” esikääntäjädirektiiveillä kaikissa kohdissa, joissa niitä tarvitaan. Ohjelma toimisi tietysti vain niissä käyttöjärjestelmissä, joiden mukainen koodi on kirjoitettu mukaan. Laitteistoriippumattomat kielet, kuten Java, perustuvat siihen, että tavukoodia lennosta

konekielelle kääntävä virtuaalikone sekä alustakirjaston käyttöjärjestelmäsidonnaiset kohdat toteutetaan ja käännetään erikseen sitä prosessoriarkkitehtuuria ja käyttöjärjestelmää varten, jossa tavukoodiohjelmien halutaan toimivan[63].

Paluuarvot on tarkastettava kaikissa käyttöjärjestelmäkutsuissa, koska mm. kapasiteetin loppuminen, käyttöoikeuksien puute tai resurssien oleminen varattuna toisen prosessin käyttöön on aina mahdollista. Tiedostojen ja muun I/O:n käytössä tarkistusten rooli korostuu, koska yllätyksiä voi tulla ulkopuolisten tahojen toimesta milloin vain. Tilanne ei ole samalla tapaa ohjelmoijan hallittavissa kuin vaikkapa muistialue, joka on kertaalleen varattu ja saatu käyttöön hamaan tulevaisuuteen asti. Esimerkiksi ulkoisen kovalevyn piuha voi irrota millä hetkellä tahansa, minkä jälkeen tiedoston lukeminen tai kirjoittaminen kertakaikkisesti loppuu aivan yllättäen. Samoin tietoverkot ovat aina jonkin verran epävakaita, joten varsinkin etäällä sijaitsevan verkkolevyn käsittely voi katketa ainakin hetkellisesti milloin vain.

Tiedostojärjestelmän rajapintaa shellissä

Katsotaan muutamia esimerkkejä siihen, miten tiedostoja, hakemistoja ja hakemistopuuhun liitettyjä tiedostojärjestelmiä voi tutkia tekstimuotoisessa shellissä. POSIX määrittelee, että koko tiedostojärjestelmän juuren tulee löytyä nimellä ”/” (eli kauttaviiva).

[63]Tämä JITtaaminen muuten edellyttää sitä, että samaa muistisivua voi sekä kirjoittaa että suorittaa. JITattavat tavukoodikielet eivät siis saa laitteistotason lisäturvaa pahantahtoiseen koodinsuoritukseen, vaan mielivaltaisen koodin injektoinnin estäminen on täysin virtuaalikoneen ja alustakirjaston ohjelmiston vastuulla. Tämä voi jonkin verran sotia suorituskykyä vastaan, koska esimerkiksi taulukoiden maksimikoko olisi syytä tarkistaa jokaisen indeksoidun osoituksen yhteydessä, mikäli taulukkoon on pääsy loppukäyttäjältä. Periaatteessa ohjelmistojen sisäiseen toteutukseen voisi käyttää nopeampia, mutta turvattomampia rakenteita – jolloin vastuu luonnollisesti siirtyy alustakirjaston tekijältä sovellusohjelman tekijälle.

Lisäksi se määrittelee, että olemassa on oltava hakemisto "/tmp" sovellusten tilapäisiä tallenteita varten ja hakemisto "/dev" tiedostoina näkyvien laitteiden hallintaa varten. Laitteista täytyy olla olemassa "/dev/null" (johon ohjattu data yksinkertaisesti unohdetaan), "/dev/tty" (joka edustaa fyysistä tai nykyään emuloitua tekstin syöttö- ja tulostuslaitetta, TTY == TeleTYpewriter) ja "/dev/console" (järjestelmän tekstisyöttö/tulostus, ei tarvitse olla käyttäjän ohjelmien käytettävissä)[64].

Käytännössä datan tallentaminen ja tiedostojen organisointi fyysisiin tallennusvälineisiin voi tapahtua hyvin erilaisin tavoin. Tämä on ymmärrettävää jo tallennusvälineiden erilaisuudesta johtuen (disketti, kovalevy, muistitikku, DVD, keskusmuisti, tietokanta, verkkolevy, pilvipalvelu, ...). Korkeamman tason rajapinnaksi tarkoitettu POSIX-standardi ei ota kantaa siihen, kuinka yksittäisen tiedostojärjestelmän **liittäminen** (engl. *mounting*) kokonaisuuden osaksi tapahtuu tai missä järjestyksessä bitit ja metatiedot ovat. Standardin "rationale" -osio nimenomaan toteaa tämän olevan toteutuksesta riippuvaa. Rajapinta siis määrittelee erittäin yleisen rakenteen, nimeämiskäytännön (muttei montaakaan varsinaista nimeä) ja tavat, joilla nimen takaa löytyvän tiedoston sisältöä luetaan ja kirjoitetaan. Käyttöjärjestelmän koodissa täytyy olla osio, joka tulkitsee juuri tietyn laitteen ja tiedostojärjestelmän sisältönä olevaa dataa.

Esimerkin vuoksi tutkitaan, miten eri tiedostojärjestelmät näkyvät Jyväskylän yliopiston suorakäyttökoneella halava.cc.jyu.fi, jota kurssin esimerkeissä muutenkin on käytetty. Tyypillisesti unixeissa

[64]POSIXissa on pyritty minimoimaan tiedostojen sijainteihin liittyvät sopimukset. Pohdinta löytyy standardin luvusta Rationale for Base Definitions A.10.1; luvussa mm. suositellaan, että uudet sovellukset eivät olettaisi nimen "/tmp" olemassaoloa, vaan käyttäisivät väliaikaistallennuksiin hakemistoa, jonka nimi löytyisi ympäristömuuttujasta TMPDIR.

ja sen perillisissä voi liittää tallennuslaitteita haluamaansa kohtaan hakemistohierarkiaa shell-komennolla `mount`. Komennolla `umount` vastaavasti voi katkaista yhteyden laitteeseen. Liittämisen voi normaalisti tehdä vain pääkäyttäjä/ylläpitäjä[65]. Normaalilla käyttäjällä on lupa tarkistaa shellin kautta, mitä tiedostojärjestelmiä on liitetty mihinkin kohtaan hakemistorakennetta. Tämä onnistuu Linuxissa komennolla `mount` ilman argumentteja:

```
mount                  # näyttää hakemistorakenteeseen
                       # liitetyt tiedostojärjestelmät
```

Apuohjelman tulosteesta ilmenee, että halavaan on Linuxille tyypilliseen tapaan liitetty esimerkiksi ”proc” -tyyppinen tiedostojärjestelmä hakemistonimelle ”/proc” – sen kautta on mahdollista nähdä prosessien tietoja; esimerkiksi prosessin 1234 muistikartan saa nähtäville seuraavalla shell-komennolla (mikä edellyttää tietysti, että käyttäjällä on lukuoikeus prosessin 1234 tietoihin):

```
cat /proc/1234/smaps
```

Käyttäjälle tuo “/proc/1234/smaps“ näyttäytyy luettavana tekstitiedostona. Kun tiedosto luetaan, tiedostojärjestelmä “proc“ kuitenkin käy tutkimassa prosessin sivutaulun sisältöä ja generoi reaaliaikaisen tilannekatsauksen tekstinä. Muihinkin prosessin 1234 tietoihin pääsee käsiksi hakemiston “/proc/1234“ sisällön kautta. Tämä on tyypillinen esimerkki unix-taustaisten järjestelmien tavasta muodostaa yhtenäiseltä näyttävä rajapinta moninaisiin asioihin, joiden käyttötarkoitus on lukeminen, kirjoittaminen tai molemmat.

[65]Henkilökohtaisissa tietokoneissa peruskäyttäjälle on kuitenkin syytä sallia esimerkiksi USB-tikkujen ja DVD-levyjen liittäminen. Tyypillisesti tämä onnistuu graafisen työpöytäohjelmiston avulla, mahdollisesti automaattisesti tallennuslaitetta liitettäessä. Konepellin alla täytyy kuitenkin lopulta tapahtua käyttöjärjestelmäkutsu, joka liittää fyysisellä laitteella olevan tiedostojärjestelmän ja ryhtyy tulkitsemaan sisältöä tiedostorajapinnan mukaisesti.

Unixissa ”kaikki näyttäytyy tiedostona”

Yksi Unixin edistyksellisistä ideoista oli juuri yhtenäisen rajapinnan luominen kaikille laitteiston osille periaatteella, jossa mahdollisimman monella käyttöjärjestelmän hallitsemalla asialla on oma looginen osoitteensa tiedostojärjestelmässä.

Esim. Linuxissa hyvin pitkälti minkä tahansa käynnissä olevan prosessin tiedot ovat, käyttöoikeuksien puitteissa, luettavissa ja myös kirjoitettavissa hakemiston ”/proc/PID” kautta. Hakemistossa “/proc“ on myös tiedostoja, jotka eivät vastaa mitään prosessia, vaan kokonaisjärjestelmää. Saa ilman muuta huvikseen kokeilla esimerkiksi jalavassa kaikkia tiedostoja, joihin on lukuoikeus, esim. “/proc/cpuinfo“, “/proc/meminfo“, “/proc/devices“ jne.

Toinen esimerkki on hakemisto “/sys“, joka on liitetty “sysfs“ -tyyppisenä. Se sisältää tiedostoja, joiden kautta järjestelmänvalvoja voi nähdä kokonaisjärjestelmän tilannetietoja ja muuttaa järjestelmänhallinnan asetuksia, kuten käynnissä olevien prosessien maksimimäärää. Moniin näistäkin on lukuoikeus kaikilla – kirjoitusoikeus kuitenkin toivottavasti vain ylläpitäjillä. Eli ei muuta kuin katselemaan, mitä tietoja voi kaivaa etäkoneemme tämänhetkisistä asetuksista!

Yksi jalavaan/halavaan liitetty tiedostojärjestelmä löytyy hakemistonimellä ”/boot”. Tämä on palvelinkoneeseen liitetty kovalevy[66], jolta käyttöjärjestelmä on käynnistyksen yhteydessä ladattu. Sen tiedostojärjestelmä on ”ext4”, joka on nykyään tyypillinen Linuxeissa käytetty tiedostojärjestelmä datan tallentamiseksi kova-

[66]Kone laiteympäristöineen on isommassa raudassa pyörivä virtuaalikone, mutta ainakin tulosteen perusteella käyttöjärjestelmän näkökulmasta kovalevy näyttää erilliseltä fyysiseltä laitteelta, nimeltään ”/dev/sda”, jonka yksi osio nimeltään ”/dev/sda1” on liitetty nimeen ”/boot” – ainakin näyttää siis aivan samalta kuin esimerkiksi omaan läppäriin kiinnitetty yksittäinen kovalevylaite.

levylle.

Käyttäjien kotihakemistot on liitetty hakemistonimillä "/autohome/nashome1", "/autohome/nashome2" ja "/autohome/nashome3". Niiden tiedostojärjestelmä on "nfs4" ja verkko-osoitteet mallia "nfs4.a

Levyjen kapasiteetista (käytetty ja vapaa tila) saa tietoja shell-komennolla `df`. Mikäli GNU-version haluaa toimivan POSIX-standard mukaisesti, voi asettaa ympäristömuuttujan POSIXLY_CORRECT. Seuraavalla tavalla voi siis kokeilla sekä GNUmaista että POSIX-maista levykapasiteetin tarkistamista halavassa:

```
unset POSIXLY_CORRECT
df
export POSIXLY_CORRECT=1
df
```

Ensimmäinen "df"-komento näyttää kapasiteetit 1024 tavun yksiköissä. Ympäristömuuttujan asettamisen jälkeinen komento näyttää POSIXin määräämissä 512 tavun yksiköissä, mikä GNU-työkalun tekijöiden mielestä on epäloogista[67].

Tiedostojen metatiedot ja konkreettinen hakemistorakenne

Hakemiston sisällön listaavalle apuohjelmalle "ls" voi antaa argumenttina vipuja, joilla tulosteeseen saa nimien lisäksi lisätietoja. Tässä esimerkkisessio halavassa, kevään 2015 kurssin 14. luennon esimerkit sisältävässä hakemistossa:

[67] Ympäristömuuttuja `POSIXLY_CORRECT` vaikuttaa muidenkin apuohjelmien toimintaan kohdissa, joissa GNU-versio toimii oletuksena epästandardisti – yleensä siksi, että tekijät ovat nähneet standardin määritelmät jollain tapaa epäloogisina. Monet määritelmistähän perustuvat yhteensopivuussyistä historiallisiin jäänteisiin, vaikka nykyään olisi fiksumpaa tehdä asiat toisin.

```
[nieminen@halava l14]$ ls
hellofile  hellofile.c  liit  liit.c  tiesototows2.txt  tiesototows.txt
[nieminen@halava l14]$ ls -i
21287487 hellofile    21208941 liit     5271900 tiesototows2.txt
23629082 hellofile.c  23629083 liit.c  21002223 tiesototows.txt
[nieminen@halava l14]$ ls -i -l -a
total 72
23629079 drwx------.  2 nieminen nobody 1024 May  7 15:33 .
10883409 drwx------. 10 nieminen nobody 2048 May  7 14:08 ..
21287487 -rwx--x--x.  1 nieminen nobody 8200 May  7 14:58 hellofile
23629082 -rw-r--r--.  1 nieminen nobody 2439 May  7 14:08 hellofile.c
21208941 -rwx--x--x.  1 nieminen nobody 8020 May  7 15:33 liit
23629083 -rw-r--r--.  1 nieminen nobody  882 May  7 14:08 liit.c
 5271900 -rw-------.  1 nieminen nobody   73 May  7 15:22 tiesototows2.txt
21002223 -rw-------.  1 nieminen nobody   73 May  7 14:59 tiesototows.txt
[nieminen@halava l14]$ ls -i -a ..
10883409 .                                        22693169 l05_helloasm_kommentoit
21048869 ..                                       22604320 l05_helloasm.s
23556450 a.out                                    22693178 l05_hellosys.c
23070220 kaikenlaskija                            22693179 l05_helnolibs.c
14159989 kaikenlaskija.c                          23250643 l07
10478086 l04_c_lahdekoodi_heksoina.txt            23723598 l08
23603826 l04_esimerkki_windowsrivinvaihdot.txt  23735312 l09
23581927 l04_helloworld.c                         20794608 l10
 5290335 l04_helloworld.txt                        8296080 l11
10479039 l04_skripti_heksoina.txt                 23790086 l12
23583442 l04_taukonaytto.sh                       21601592 l13
10479279 l04_tiedoston_metatietoja.txt            23629079 l14
22690502 l05_helloasm.c
```

Tavallisesti “ls“ näyttää vain tiedostojen nimet. Vipuargumentilla “-i“ näkyy myös tiedoston ”sarjanumero”, joka on nimeämiskäytäntöä matalamman tason yksilöinti tiedostolle. Se voi olla hyvin läheisesti kytköksissä tietorakenteisiin, joita käyttöjärjestelmä sisäisesti käyttää tiedostojen organisointiin tallennuslaitteissa. Ii-kirjain viittaa **i-solmuun** (engl. *inode, index node*), jollaista voidaan käyttää perustietorakenteena tiedostojärjestelmässä. Nimi periytyy vanhasta unixista. Vipu “-l“ on jo aiemmasta esimerkistä tuttu. Se pyytää näyttämään mm. käyttöoikeudet, omistajan ja muokkausajan. Vipu “-a“ pyytää näyttämään myös tiedostot, joiden nimi alkaa pisteellä. Näistä “.“ viittaa aina hakemistoon it-

seensä ja "..“ hakemistopuussa yhtä tasoa ylempään hakemistoon. Viimeinen esimerkki ylläolevassa sessiossa listaa nykyisen työhakemiston sijasta yhtä tasoa ylemmän hakemiston tiedostot. Voidaan huomata, että hakemiston "l14“ i-solmun numero on listauksessa sama kuin mikä näkyy nimellä ".“ ensimmäisessä listauksessa. Kyseessähän on sama hakemisto. Samalla periaatteella ensimmäisen listauksen hakemiston "..“ i-solmun numero on sama kuin hakemiston ".“ numero jälkimmäisessä listauksessa.

Johtopäätöksenä voidaan esimerkkien perusteella havaita, että hakemistohierarkia näyttäytyy käyttöjärjestelmässä kahteen suuntaan linkittyvänä puurakenteena, jossa esimerkiksi kokonaisluvuilla yksilöidään kunkin hakemiston vanhempi (ylähakemisto) ja lapset (alihakemistot). Nyt voidaan myös ymmärtää komennon "ls -l“ tulosteesta käyttöoikeussarakkeen jälkeen löytyvä numero: Se näyttää tiedostoon kohdistuvien viittausten eli **linkkien** (engl. *link*) määrän hakemistorakenteessa. Jokaiseen tiedostoon on ainakin yksi linkki (siitä hakemistosta, jossa ne sijaitsevat). Viimeisen linkin poistaminen (engl. *unlinking*) johtaa tiedoston ”unohtumiseen” eli levytilan vapautumisen uuteen käyttöön[68]. Koska alihakemistoista on linkki ylähakemistoon, on esimerkissä luentokohtaiset esimerkkikansiot sisältävällä yläkansiolla linkkilukumäärä suurempi kuin muilla tiedostoilla (tarkkaan ottaen kymmenen: normaali linkki omasta itsestään nimellä ”.”, linkit nimellä ”..” kahdeksasta alihakemistosta sekä yksi linkki omasta ylähakemistostaan.

Linkkejä voi luoda myös hakemistojen välille, joten tosiasiassa puurakenne voi tässä tapauksessa sisältää kulkukelpoisia ”riippusiltoja” etäistenkin haarojen välillä. Linkkejä on kahdenlaisia: **Kova**

[68]Tietoturvan mielessä on muistettava, että fyysinen data ei tuhoudu välittömästi ilman lisätoimenpiteitä, vaan tiedon täydellinen ja varma hävittäminen edellyttää esimerkiksi satunnaisen datan kirjoittamista aiemman sisällön päälle, varmuden vuoksi useita kertoja, ennen kuin tiedoston viimeinen linkki poistetaan.

linkki (engl. *hard link*) on aito viite toiseen sarjanumerolla (tai i-solmun numerolla) yksilöityyn tiedostoon. Käsite on samaistettavissa oliokielen viitteeseen: minkä tahansa samaan tiedostoon viittavaan nimen kautta tehtävä muutos tiedostoon tai sen käyttöoikeuksiin vaikuttaa yhteen ja samaan numerolla yksilöityyn tiedostoon. Kovat linkit toimivat vain saman tiedostojärjestelmän sisällä, esimerkiksi yhdellä kovalevyosiolla. Se toimii sovellusten käyttötarkoituksiin tarpeettoman matalalla tasolla. Kovan linkin käyttö hämärtää tiedoston "sijainnin" käsitteen: data on levyllä yhdessä paikassa, mutta hakemistorakenteen mielessä se näkyy tasaarvoisesti yhtäaikaa jokaisessa sijainnissa, josta siihen on kova linkki. **Symbolinen linkki** (engl. *symbolic link*) puolestaan on oma "tiedosto-olionsa", jolla on oma i-solmu. Symbolinen linkki nimeää viittaamansa tiedoston normaalilla hakemistonimellä, mutta ei sisällä muuta tietoa viitteen kohteesta. Käyttöoikeudet määräytyvät kohdetiedoston oikeuksien mukaan, eikä symbolisella linkillä ole vaikutusta todellisen tiedoston sijaintiin. Viitatun tiedoston poistaminen ei poista linkkiä, mutta poiston jälkeen linkissä oleva osoite ei tietenkään enää ole käyttökelpoinen[69].

Tyypillistä on esimerkiksi linkittää (symbolisella linkillä) POSIXin määräämien komentojen tilalle jotakin monipuolisempia, mahdollisesti eri nimisiä ohjelmia. Esimerkiksi “/bin/sh“ voi olla linkki varsinaiseen ohjelmatiedostoon “/bin/bash“ tai “/usr/bin/vi“ voi olla linkki tiedostoon “/usr/bin/vim“. Suoritettaessa ohjelma voi päätellä käynnistysnimensä perusteella, oletetaanko sen toimivan POSIXmaisesti vai omilla laajennoksilla varustettuna.

[69]Ohjelmointianalogiana symbolisesta linkistä voi siis tulla tietyllä tapaa “dangling pointer“.

Esimerkki tiedoston käyttöoikeuksista POSIXissa:
omistajalla kaikki (rwx), ryhmällä ja muilla luku ja suoritus(rx);
käynnistyy olennaisesti omistajan käyttöoikeuksin (setUID).

				user			group			other		
	setUID	setGID	sticky	r	w	x	r	w	x	r	w	x
bitit	1	0	0	1	1	1	1	0	1	1	0	1
oktaali			4			7			5			5

Kuva 0.32: *Tiedoston käyttöoikeuksien määrittely POSIXin chmod() -kutsulla tai shell-komennolla chmod.*

Käyttöoikeuksien hallintaa tiedostojärjestelmissä

Tiedoston oikeuksien asettaminen tapahtuu POSIX-yhteensopivassa shellissä ja tiedostojärjestelmässä 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
chmod 755 ohojelma     # oikeudet bitteinä, oktaaliluku
```

POSIX määrittää tiedostojen oikeudet yhteensä 12 bitillä tavalla, joka juontaa juurensa vanhaan Unixiin 1970-luvulla. Kuvassa 0.32 on esimerkki oikeusmäärittelystä. Alimmat 9 bittiä tulevat seuraavista:

- Tasot: omistaja / ryhmä / muut (user / group / other) (ugo)
- Oikeudet: luku / kirjoitus / suoritus (read / write / execute) (rwx)

Tasojen ja oikeuksien 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.

Kolme ylintä bittiä, eli nelinumeroisen oktaalikoodin eniten merkitsevä numero, tulevat seuraavista:

- setuid - bitti: jos tämä bitti on asetettu ohjelmatiedostossa, käynnistetyn prosessin omistajaksi tulkitaan (ns. effective user) tiedoston omistaja, eikä se käyttäjä, joka alunperin käynnisti ohjelman. Muutos prosessielementtiin tapahtuu exec() -kutsussa.

- setgid - bitti: jos tämä bitti on asetettu ohjelmatiedostossa, käynnistetyn prosessin omistavaksi ryhmäksi (ns. effective group) tulkitaan tiedoston omistava ryhmä, ei siis käynnistävän käyttäjän oletusryhmä. Muutos prosessielementtiin tapahtuu exec() -kutsussa.

- sticky - bitti: alkuperäisessä ideassa, jos tämä bitti oli asetettu ohjelmatiedostossa, ohjelmaa ei poistettu muistista (tarkemmin heittovaihtotiedostosta) sen suorituksen loputtua, joten uusi instanssi oli nopeampi käynnistää. Nykyvariaatioissa kuitenkin on eri käyttötarkoitukset – mm. Linuxissa ja OS X:ssä, jos tämä bitti on asetettu hakemistolle, rajoittuu sisällön poisto ja uudelleennimeäminen vain kunkin tiedoston omistajalle tai pääkäyttäjälle, riippumatta kunkin tiedoston yksittäisistä oikeusasetuksista. Tämä on järkevää mm. /tmp -hakemistossa, johon kaikki voivat laittaa tiedos-

toja, mutta eivät saa poistaa tai nimetä uudelleen muiden käyttäjien tiedostoja.

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. Tietoturvasta huolehtiva käyttäjä poistaa kotihakemistoltaan rwx -oikeudet kaikilta paitsi itseltään; tämä on myös järkevä oletusarvo uuden käyttäjän kotihakemiston luonnissa. Oktaalilukuna kotihakemiston moodi olisi siis mieluusti 0700.

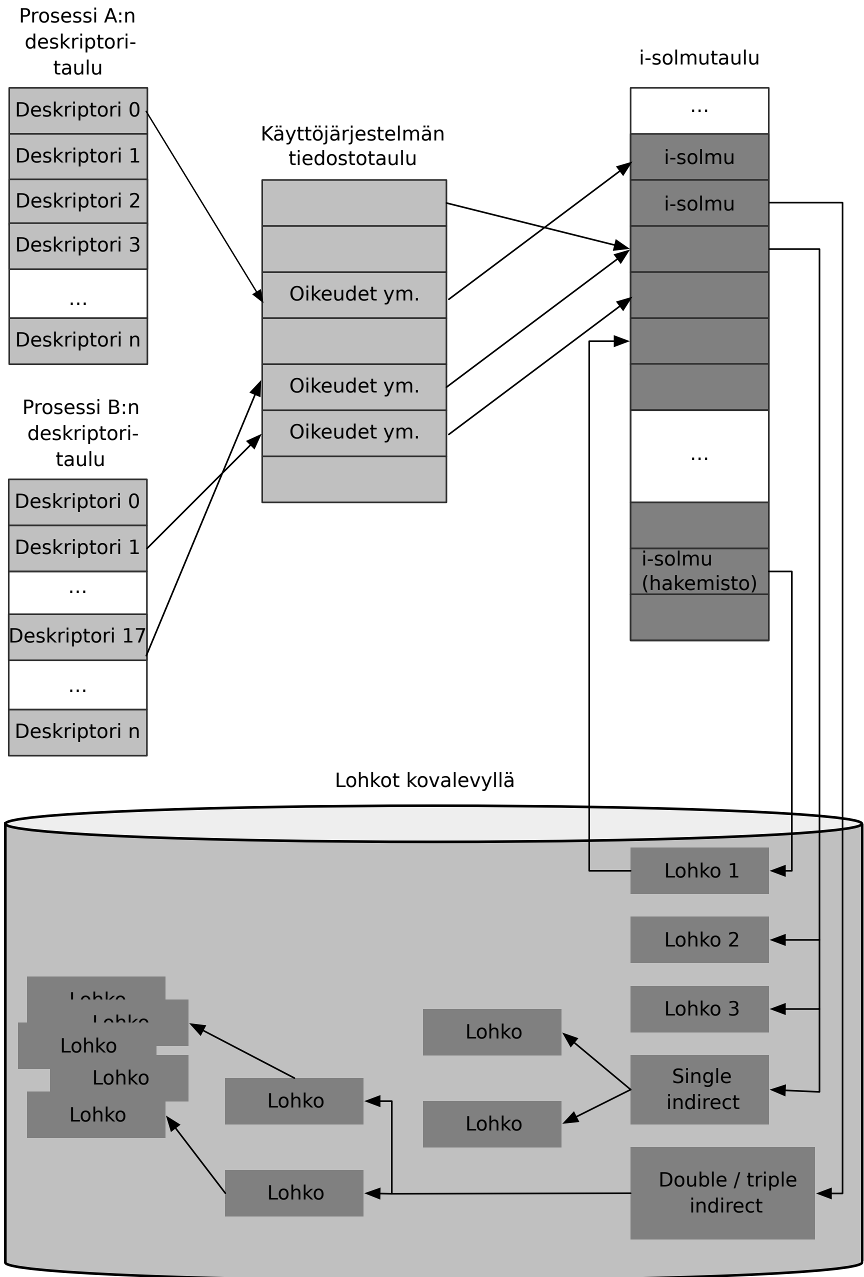
Windowsin graafisella tiedostoselaimella (”Windows Explorer”) voi klikata tiedostoa hiiren oikealla napilla ja säätää oikeuksia Security -välilehdeltä. Oikeuksien määritys voi eri järjestelmissä tapahtua erilaisella tarkkuustasolla, joten tiedostojen siirtäminen voi aiheuttaa suojaustiedon muuttumista tarpeettoman sallituksi tai suojatuksi.

POSIX sallii hienojakoisemmat käyttöoikeusmääritykset, vaikka edellä mainitut ovat minimiedellytys.

Esimerkki tiedostojärjestelmän toteutuksesta: unixin i-solmut

Tiedostojen järkevä käyttö ohjelmissa ja tallennusvälineissä edellyttää joitakin käyttöjärjestelmän sisäisiä tietorakenteita, jotka löytyvät kuvan 0.33 yläpuoliskosta.

Ensinnäkin jokaisella prosessilla on **deskriptoritaulu** osana prosessielementtiä (deskriptoritauluja on siis yhtä monta kuin on prosesseja). Prosessi käyttää deskriptoritaulun indeksejä avaamiensa tiedostojen yksilöintiin käyttöjärjestelmäkutsuissa. Muistele esi-



Kuva 0.33: *Käyttöjärjestelmän tietorakenteita tiedostojärjestelmän toteuttamiseksi.*

merkiksi monisteen alun Hello World -esimerkkiä, joka tulosti Linuxin käyttöjärjestelmäkutsulla deskriptoriin 1, joka oli ohjelman käynnistyessä yhdistetty valmiiksi standardiulostuloon. Vastaavasti Linux avaa standardisyötteen deskriptoriin 0 ja virheulostulon deskriptoriin 2. Muut tiedostot täytyy avata erikseen käyttöjärjestelmäkutsuilla, jotka palauttavat tietenkin deskriptorinumeron, mikäli avaaminen onnistuu.

Korkean tason kielillä, kuten C:llä, ei kannata kovakoodata tietyn käyttöjärjestelmän oletusdeskriptorinumeroita, vaan tulisi käyttää alustakirjaston tietovirtoja, jotka on nimetty C:ssä ”stdin”, ”stdout” ja ”stderr” (ja oliokielissä hyvin vastaavalla tavoin). On hyvä kuitenkin tiedostaa, että sisäisesti nämäkin deskriptorit ovat samanlaisia kokonaislukuja kuin muutkin, ja siten ne ovat hyvin tasa-arvoisessa roolissa muiden tiedostojen kanssa.

Deskriptoritaulussa on avatun / lukitun tiedoston osalta viite käyttöjärjestelmän **tiedostotauluun** (engl. *file table*). Tiedostotauluja tarvitaan vain 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 pyytäessään tiedoston käyttöoikeutta.

Lukemista varten tiedosto voi olla avattuna usealla prosessilla – molemmilla on oma prosessikohtainen tiedosto-osoitin, joten ne voivat lukea tiedostoa eri puolilta. Kirjoittaminen edellyttää tiedoston lukitsemista yhdelle prosessille, jotta data ei mene kilpa-

ajon takia sekaisin. Lukitun tiedoston avaaminen ei tietysti muilta yhdenaikaisilta ohjelmilta onnistu, vaan käyttöjärjestelmäkutsu palauttaa niille virhekoodin. Käyttäjälle kannattanee ilmoittaa tilanteesta, jotta hän osaa tarvittaessa sulkea muut tiedostoa käyttävät ohjelmat ja yrittää uudelleen.

Koska tiedostoihin sisältyy käyttöjärjestelmän tukema ja joidenkin kutsujen osalta atominen lukitusmenettely, niitä voidaan käyttää karkeana keinona prosessien synkronointiin. Sovellusohjelmoijan on oltava tarkka, että alustakutsun dokumentaatio todella lupaa tiedoston lukituksen tapahtuvan atomisesti. Saman prosessin säikeiden lukitus ei tiedostoilla tietenkään onnistu, koska tiedostodeskriptorit, kuten muutkin prosessin tiedot, ovat kaikille säikeille yhteiset. Mikäli tiedoston olemassaolo tulkitaan lukoksi, on syytä varmistaa, että sovellusohjelma käsittelee kaikki mahdolliset lopetussignaalit ja poistaa lopussa omatoimisesti lukkotiedoston. Sikäli kuin edistyneempiä ja tehokkaampia synkronointimekanismeja on mahdollista käyttää, ne ovat tietysti joustavampia kuin tiedostot.

Kuvan 0.33 oikeassa laidassa havainnollistetaan historiallista Unix-tiedostojärjestelmää, jossa tiedoston yksilöi **i-numero** (engl. *i-number*) eli konkreettinen indeksi **i-solmujen** (engl. *i-node*) taulukkoon. Pienet tiedostot (12 lohkoa, so. 48 kilotavua) voidaan kuvata kokonaan yhdellä i-solmulla. Isommille tiedostoille on käytettävä hierarkkista rakennetta. I-solmutaulukko on tiedostojärjestelmän tietorakenne, joka sisältää jokaista tiedostoa (tai tarkemmin i-solmua) kohden seuraavat metatiedot:

- tyyppi eli tavallinen / hakemisto / linkki / erikoistiedosto
- suojaustiedot eli omistava käyttäjä (UID, user ID) ja ryhmä (GID, group ID) sekä luku- / kirjoitus- / suoritusoikeudet 12 bitin jonona.

- aikaleimat (käyttö, tiedoston sisällön muutos, i-solmun muutos)
- koko (*size*) tavuina
- käytettyjen lohkojen lukumäärä (*block count*)
- lohkojen suorat osoitteet tallennuslaitteen pinnassa (*direct blocks*); esim. maksimissaan 12 kpl, pienille tiedostoille, so. esim. 4k-lohkoilla $12 \times 4096 = 49152$ tavuisille, riittävä määrä)
- Tarvittaessa yksinkertainen epäsuora osoite (*single indirect*; viittaa yhteen lisätaulukkoon suorien lohkojen osoitteita; 4k-lohkoilla riittää siis noin 4 megatavun tiedostoille)
- Tarvittaessa kaksinkertainen epäsuora osoite (*double indirect*; viittaa taulukkoon, jonka alkiot viittaavat taulukoihin lohkojen osoitteista; 4k-lohkoilla riittää noin 4 gigatavun tiedostoille)
- Tarvittaessa kolminkertainen epäsuora osoite (*triple indirect*; taulukko, jonka alkiot viittaavat taulukoihin, jonka alkiot viittaavat taulukoihin lohkojen osoitteista; 4k-lohkoilla riittää noin 4 teratavun tiedostoille – isompia ei voi tallentaa vanhanmalliseen unix-järjestelmään, vaan tarvitaan kolmitasoista lohkohierarkiaa edistyneemmät tallennusrakenteet[70])

Tämä alkuperäinen idea i-solmuista on yhä voimissaan. Esimerkiksi Linux käsittelee samanhenkisiä ”v-solmuja”. Erilaisten tiedostojen tyypit Unix-tyyppisessä tiedostojärjestelmässä voivat olla seuraavat:

[70] Tiedostojärjestelmien kapasiteettirajat ovat toki olleet suunnitteluvaiheessa mielivaltaisia, mutta ne perustuvat käytännön realiteetteihin käytettyjen tietorakenteiden muodosta ja laajuudesta.

- tavallinen tiedosto (esim. tekstitiedosto, valokuva, videotiedosto, ... olennaisesti bittijono jossa on lohkojen sisältö peräkkäin)

- hakemisto (hakemistojen idea on järjestellä tiedot hierarkkisesti puuksi). Hakemisto Unixissa on (sisäisesti) aivan tavanomainen tiedosto, jossa on jokaista hakemiston sisältämää tiedostoa kohden seuraavat tiedot:

  - tiedoston nimi
  - i-solmun numero

  Huomaa, että tässä järjestelmässä *tiedoston nimi määräytyy hakemistotiedostoon kirjatun merkkijonon perusteella*! Tiedoston sisältö on lohkoissa lojuva bittijono, jolla on kyllä tietyt i-solmun metatiedot mutta ei nimeä. Nimet löytyvät hakemistotiedostojen riveiltä!

- ”kova linkki” (uusi viite fyysiseen tiedostoon, johon on viite myös muualta; käytännössä nimi, joka ohjaa suoraan i-solmuun, jolla on vähintään yksi toinenkin nimi; useat linkit ovat hyvin normaaleja unixmaisissa tiedostojärjestelmissä)

- ”symbolinen linkki” (myös uusi viite fyysiseen tiedostoon; käytännössä nimi kuitenkin ohjaa hakemisto-osoitteeseen merkkijonona, ei siis i-solmuun)

- erikoistiedosto (vastaa laitetta tai muuta käyttöjärjestelmän tarjoamaa rajapintaa, ei siis tallennusvälineellä olevaa staattista dataa).

Miten hakemistonimiä käytetään, kun prosessi haluaa löytää jonkun tiedoston?

Oletetaan, että prosessin nykyinen työhakemisto on `/home/ktunnus/P` Prosessi haluaa avata tiedoston nimeltä `grillikuvat/2011/nakit.j` Käyttöjärjestelmän toimintoketju on silloin seuraavanlainen:

1. Nykyisestä hakemistosta (/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 edustaa hakemistotyyppistä tiedostoa; tästä uudesta hakemistosta käyttöjärjestelmä etsii taas seuraavaa nimeä (`2011`). (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.

4. Käyttöjärjestelmä tallentaa avauspyynnön mukaiset käyttöoikeudet ja lukitustiedot sisäisiin ajonaikaisiin tietorakenteisiinsa ja palauttaa käyttöjärjestelmäkutsun paluuarvona prosessille prosessikohtaisen deskriptorinumeron.

Esimerkkiä kannattaa makustella mielessään, kunnes on selvää, että samalla periaatteella hakemistopolun osiot ".." ja "." johtavat aivan vastaavasti aina seuraavaan i-solmuun, eivätkä edellytä sen ihmeempiä toimenpiteitä. Ainoa erilainen tapaus, että absoluuttisen polun alussa oleva "/" tarkoittaa, että ensimmäisenä tutkittava i-solmu on koko järjestelmän juurihakemisto, eikä prosessielementissä tällä hetkellä merkitty työhakemisto.

Tämä on siis yksi esimerkki tiedostojärjestelmästä. Muitakin on paljon, ja toteutuksen yksityiskohdista riippuu mm. tiedostojen maksimikoko, nimien maksimipituus ja käytettävissä oleva merkistö, oikeuksien asetuksen tarkkuus ym. seikat. Tarkemmat yksityiskohdat jätetään oman kiinnostuksen ja tulevien projektien tarpeiden varaan. Internetistä löytyy valtavasti hyviä johdantoja sekä tässä esitetyn unix-tiedostojärjestelmän nykyisiin johdannaisiin että muihin tiedostojärjestelmiin.

POSIX ei määritä paljoakaan tiedostojärjestelmän sisäisestä toteutuksesta, mutta rajapinnan osalta se edellyttää suurin piirtein edellä esitetyn näköisen tiedostojärjestelmän. Mm. tiedostoilla tulee olla *jonkinlainen* yksilöllinen ja nimestä riippumaton ”sarjanumero”, olipa se i-solmunumero tai muu. Shellissä on oltava komento “ln“ linkkien tekemiseksi. Matalimman tason C-rajapinta (mm. tiedoston avaaminen) käyttää tiedostodeskriptoria, jonka tyyppi on kokonaisluku “int“. Käyttöoikeudet ja aikaleimatiedot määritetään kuten tässä, ja POSIX-kutsun stat() tulee palauttaa ”mielekkäät arvot” kyseisille kentille.

Huomioita muista tiedostojärjestelmistä

Tiedostojärjestelmiä on monia! Esim. Linuxeissa yleinen ext4 (aiemmat versiot ext3, ext2), Windowsissa tyypillinen ntfs, verkon yli jaettava nfs, yksinkertainen fat, keskusmuistia kovalevyn sijaan käyttävä ramfs. Lisäksi tyypillisiä ovat mm. 9p, afs, hfs, jfs, ... Tiedostojärjestelmät ovat erilaisia, eri tarkoituksiin ja eri aikoina syntyneitä. Niiden mahdollisuudet ja rajoitukset kumpuavat toteutustavasta.

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 palvelinpäässä. Yhteys perustuu määriteltyihin kommunikaatioprotokolliin, joten NFS:llä kytketyillä tietokoneilla voi olla eri käyttöjärjestelmät, kunhan molemmissa on tuki NFS:lle.

Mainittakoon vielä Javan päälle toteutettu HDFS, jota viime aikona suosittu hajautetun datankäsittelyn alusta Hadoop käyttää tiedostojen tallentamiseen. Sen tavoitteena on skaalautuva kapasiteetti ja tiedon sijoittelu sitä käsittelevien laskentaprosessien fyysiseen läheisyyteen klusteriympäristössä.

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

- Nopeus/tehokkuus eri tehtäviin: esimerkiksi tilansäästö levyllä/muistissa, nimen etsintään kuluva aika, tiedoston luontiin, kirjoittamiseen, lukuun, poistoon kuluva aika. Halutaanko säästöjä erityisesti isoille vai pienille tiedostoille.

  → ”Yksi koko ei sovi kaikille” – tiedostojärjestelmä on valittava kokonaisjärjestelmän käyttötarkoituksen mukaan. Esim. paljon pieniä tiedostoja voi toimia paremmin eri järjestelmässä kuin isojen tiedostojen käsittely, magneettilevylle suunniteltu tiedostojärjestelmä ei välttämättä ole omiaan SSD-levyä käytettäessä, ...

- Toimintavarmuus: mitä tapahtuu, jos tulee sähkökatko tai laitevika? Ilman lisävarmistuksia esimerkiksi i-solmuihin perustuva tiedostojärjestelmä menee takuuvarmasti sekaisin, jos laitteesta katkeaa virta (tai liitäntäjohto irtoaa) kesken tiedoston sisältöön liittyvien indeksitaulukoiden päivittämistä. Tiedoston sisältö on saatettu ehtiä tallentaa vain osittain,

mutta ehkä vieläkin pahempaa on, että järjestelmän rakenne menee sekaisin.

→ monissa tiedostojärjestelmissä on toteutettu ”transaktioperiaate” eli **journalointi** (engl. *journaling file system*): Kirjoitusoperaatiot tehdään atominen kirjoituspätkä kerrallaan. Ensin tehdään ”ennakkokirjoitus” yhteen paikkaan, mutta ei vielä siihen kohtaan levyä, mihin on lopulta tarkoitus. Tällaista paikkaa sanotaan esim. journalointilokiksi[71]. 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 jäänyt kesken. Muutossuunnitelma käydään läpi toteutumatta jääneiden varalta aina, kun tiedostojärjestelmä seuraavan kerran kytketään eli mountataan.

→ journalointi ei tietenkään auta totaalisen laitevian tapahtuessa, eli esimerkiksi levyn pinnan naarmuuntuessa, moottorin tai laakeroinnin rikkoutuessa tai SSD-levyn muistikomponenttien koettua lopullisen kuolemansa käyttökelvottomiksi; näistä selviytymiseksi voi varautua vaikkapa aiemmin kuvatun RAID-levyryhmän käytöllä, eikä mikään koskaan korvaa ajoittaista varmuuskopiointia maantieteellisen etäisyyden päähän paikkaan, jonka tuhoutuminen esim. samassa tulipalossa tai maanjäristyksessä on riittävän epätodennäköistä.

Vaikka tiedostojärjestelmää valitessa on helppo sulkea käyttötarkoitukseen soveltumattomat valinnat pois, voi lopullinen valinta olla kompromissi esimerkiksi samalla palvelimella tapahtuvien pien-

[71] Aivan relevantti suomennos voisi olla ”päiväkirja” tai ”muutossuunnitelma”?

ten ja suurten tiedostojen käsittelyn suhteen – jompaa kumpaa on painotettava mm. lohkon koon valinnassa.

Yhteenveto: Tiedostojen nimeäminen ja käyttöoikeudet ovat oma kerroksensa tiedostojen laitetason toteutuksen päällä. Toteutustavat vaihtelevat eri tarkoituksiin kehitettyjen tiedostojärjestelmien välillä, mutta ylöspäin näkyvä rajapinta on yhtenäinen. Tietyn tiedostojärjestelmän käyttö edellyttää konkreettisten tallennustapojen ja -algoritmien toteutusta käyttöjärjestelmän koodissa. Unix-tradition mukaisissa tiedostojärjestelmissä kullakin tiedostolla on i-numero, eli indeksi i-solmujen taulukkoon. Jokaisesta tiedostosta on tietorakenteeseen tallennettu tieto sen tyypistä, omistajasta ja oikeuksista, aikaleimoista, koosta, fyysisten levylohkojen määrästä ja niiden suorista osoitteista levylaitteessa. Isot tiedostot vaativat yksin-, kaksin- tai kolminkertaista epäsuoraa osoitteistamista, jossa kokonainen lohko varataan muiden lohkojen osoitteiden listaamiseen.

0.13 Käyttöjärjestelmän suunnittelusta

Avainsanat: vuoronnusmenettelyt: FCFS (Fist come, first serve), kiertojono, prioriteetti, dynaaminen prioriteetti; reaaliaikajärjestelmä: determinismi, vaste/responsiivisuus, hallittavuus, luotettavuus, vikasietoinen toiminta, pre-emptiivisyys; mikroydin, monoliittinen käyttöjärjestelmä

Osaamistavoitteet: Luvun luettuaan opiskelija:

- osaa luetella ja selittää yksinkertaisia vuoronnusmenetelmiä sekä antaa esimerkkejä tilanteista, joissa kukin niistä voisi olla edullisempi valinta kuin jokin toinen; vastaavasti osaa valita annetuista vaihtoehdoista sopivimman vuoronnusmenetelmän annetussa skenaariossa [ydin/arvos2]
- osaa antaa esimerkkejä vuoronnuksen yleisten periaatteiden ilmenemisestä muissakin yhteyksissä kuin käyttöjärjestelmän prosessivuorontajassa [ydin/arvos2]
- osaa kertoa, mitä tarkoittaa reaaliaikajärjestelmä ja antaa esimerkkejä reaaliaikajärjestelmistä [edist/arvos3]
- tietää, mitä on pre-emptiivinen deadline-vuoronnus ja missä tilanteessa sitä esimerkiksi tarvitaan [edist/arvos3]
- tietää olennaisimmat erot monoliittisen ja mikroydinjärjestelmän välillä [ydin/arvos2]
- osaa kuvailla linux-lähdekoodin uusimman version hakemistorakenteen ylimmällä tasolla sekä perustella, mihin kyseinen hakemistojäsentely perustuu [edist/arvos3]

Tässä luvussa annetaan vielä muutamia esimerkkejä joistakin käyttöjärjestelmän suunnitteluun ja asetusten tekemiseen liittyvistä pohdinnoista ja joistakin mahdollisista toteutustavoista. Nämä ovat esimerkkejä, jotka ovat vuosien varrella valikoituneet kurssilla esitettäviksi perusesimerkeiksi. Todellisuudessa seikkoja on paljon. Tutkimus jatkuu, ja reaalimaailman vaatimukset muuttuvat. Tärkeintä on ymmärtää, että maailma ei ole valmis – muistaen kuitenkin, että pyörä on jo keksitty, eikä sen muotoa kannata yrittää parantaa enää kovin suurilla työpanoksilla.

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 aikaviipaleiden verran. (vaarana alemman prioriteetin nääntyminen)::

```
Prioriteetti 0 [READY0] -> PID 24 -> NULL
Prioriteetti 1 [READY1] -> PID 7 -> PID 1234 -> PID 778 -> NUL
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 ovat hyviä ohjelmille, jotka eivät laske kovin paljon; hintana on, 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 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 ja kohtalaisen monimutkainen algoritmi tätä hallitsemaan.

Esimerkki: 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 syntetisaattoriohjelman 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 ko-

mento korjausliikkeestä, koska muuten kallistus saattaa kärjistyä katastrofaalisesti eikä matkustajille käy hyvin.

- Laskentaohjelmisto tuottaa sääennustetta huomiselle päivälle. Laskennan on ilmeisesti syytä valmistua hyvissä ajoin ennen huomista.

- Tekoäly laskee tietokonepelin simuloidun vastustajan seuraavaa siirtoa. Elämyksellisyys kärsii, jos vastustaja jää seisoskelemaan päättämättömänä paikoilleen.

- Sensoriverkko mittaa merenpinnan korkeutta mittauspoijuilla. Varoitus mahdollisesti liikkeellä olevasta tsunamista on saatava rannikolle siten, että evakuointiin jää aikaa.

Käyttöjärjestelmätoteutuksen kannalta haastavimpia reaaliaikajärjestelmiä ovat ne, joiden aikaskaalat vastaavat vuorontamisen aikaviipaleita tai pahimmillaan yksittäisessä keskeytyskäsittelijässä vietettyä aikaa. Reaaliaikajärjestelmän yleisiä vaatimuksia ovat seuraavat:

- determinismi (determinism); olennaiset toimenpiteet saadaan suoritukseen aina riittävän pian niitä tarvitsevan ilmiön (esim. prosessorin laitekeskeytyksen) 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, ehkä osittain rikkoutuikin, niin ohjausta pitäisi edelleen jatkaa, jottei vauhdissa tule enempää 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. Fyysisesti liikkuvien asioiden ohjaamisessa lyhyet vasteajat myös korreloivat nopean vauhdin ja suurten liike-energioiden kanssa.

Normaali interaktiivinen tietokoneen käyttö ei edellytä ”reaaliaikaisuutta”, koska yleensä ihmiskäyttäjä jaksaa odotella vastetta. 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 sekä vastaamista peliohjaimen kontrolleihin.

Reaaliaikaisessa käyttöjärjestelmässä on tärkeätä olla ominaisuus nimeltä **pre-emptiivisyys** (engl. *preemption/pre-emption*). Pre-

emptioksi voidaan yleisesti sanoa jo sitäkin, kun prosessi ylipäätään voi keskeytyä kesken laskennan (vaikka vain aika-annoksen loppumiseenkin). Reaaliaikajärjestelmissä pre-emption määritelmä on kuitenkin tiukempi: Jotta millisekuntiajoituksiin päästäisiin, on myös meneillään olevan käyttöjärjestelmäkutsun tai keskeytyskäsittelyn voitava keskeytyä pre-emptiivisesti, jos tulee ”se tärkeä keskeytys”. Reaaliaikavaatimus nimittäin tulee usein juuri tietyn laitekomponentin taholta (esim. audiojärjestelmässä äänipiiriltä, teollisuusjärjestelmässä laser-leikkurin anturista, raketissa ohjaussensoreilta). Pre-emptiivinen keskeytyskäsittely edellyttää, että eri syistä aiheutuneita käyttöjärjestelmäkutsuja pitää voida olla käynnissä yhdenaikaisesti, mikä aiheuttaa kirjanpidollisia lisähaasteita ja algoritmien monimutkaisuutta ei-reaaliaikajärjestelmään verrattuna.

Pre-emptiivisyyden kautta voidaan yrittää saada aikaan deadline-perusteinen vuoronnus (engl. *deadline scheduling*), jossa jokainen tiukan takarajan omaava prosessi pääsee vuoronnukseen oman deadlinensa puitteissa. Nopeimmin vastaan tulevan takarajan omaavaa prosessia voidaan priorisoida suhteessa muihin. Sovelluksesta riippuen takarajoja voi olla joko toimenpiteen aloittamisen tai sen loppuun saattamisen suhteen, tai molempien. Lyhyissä aikaskaaloissa nykyistenkin tietokoneiden kellotaajuus muuttuu kiinteäksi rajoitteeksi, joten priorisointi erilaisten tehtävien ja yhtäaikaa vuoronnuksessa olevien prosessien määrän suhteen on välttämätöntä.

Esimerkki: Ajonaikainen moduulijako, mikroydin vs. monoliittinen ydin

Luvussa 0.2 mainittiin lyhyesti mikroydin ja monoliittinen lähestymistapa. Kerrataan ne vielä tässäkin yhteydessä. Riippumatta tavasta, jolla käyttöjärjestelmän lähdekoodi organisoidaan, yksi mer-

kittävä valinta on tehtävä ajonaikaisen organisoinnin suhteen: Toimiiko käyttöjärjestelmä yhtenä kaikkea hallitsevana ”prosessina” muiden prosessien yläpuolella, vai olisiko sitä syytä suorittaa erillisinä prosesseina tai säikeinä, joista jokainen hoitaa tiettyä tehtävää (prosessien vuoronnus yhdellä prosessin vastuulla, muistinhallinta toisen, tiedostojärjestelmät kolmannen vastuulla jne.)

Kokonaan käyttöjärjestelmätilassa suoritettava monoliittinen järjestelmä ei hukkaa prosessorin suoritusaikaa mihinkään ylimääräiseen, joten sellaisen hyvä puoli on suorituskyvyn maksimoituminen. Silloin on kuitenkin suuri vaara, että koko järjestelmä kaatuu tai jumittuu lopullisesti käyttöjärjestelmässä olevan ohjelmointivirheen vuoksi. Perinteisten väärien muistiosoitusten ja ”ikuisten silmukoiden” lisäksi käyttöjärjestelmällä on suuri vaara joutua luvussa 0.9 kuvattuun deadlock-tilanteeseen, koska suuri osa käyttöjärjestelmän sisäisistäkin tehtävistä liittyy resurssien lukitsemisiin.

Toinen ääripää, pääasiassa käyttäjätilassa toimiva mikroydinjärjestelmä, on turvallisempi ja toimintavarmempi, koska palveluprosessin lukkiutumisen tai kaatumisen voi havaita, ja sen voi vaikka pakottaa käynnistymään uudestaan. Tämän voi hoitaa vaikkapa korkealla prioriteetilla toimiva **vahtikoira** (engl. *watch dog*) eli prosessi, joka pääsee korkeimman prioriteettinsa vuoksi aina suoritukseen ja pystyy tarkistamaan muiden prosessien tilanteen.

Lisäksi ajonaikainen jako useampiin prosesseihin helpottaa tehtävien välistä priorisointia ja erilaisten käyttöikeuksien määrittämistä eri palveluita hoitaville prosesseille. Esimerkiksi nettiyhteyttä hallitsevan koodin ei tarvitsisi päästä käsiksi kaikkiin tiedostoihin tai kaikkien prosessien muistialueisiin – siellähän voi olla vaikka ohjelmointivirhe, joka mahdollistaa ulkopuolisen hyökkääjän yritykset kaapata tietokone; tällainen yritys pystyy kaatumaan suojausvirhekeskeytykseen vain, mikäli verkkoyhteyskoodi toimii käyttä-

jätilassa. Mikroydinmallin tavoittelu myös pakottaa suunnittelemaan rajapinnat löyhemmiksi: yksi koodiosa ei pääse käsiksi muiden osien käsittelemiin rakenteisiin muuten kuin erikseen määritellyillä operaatiopyynnöillä, joten ohjelmoijan mahdollinen houkutus kulkea rajapinnan ohi ei pääse teknisestä esteestä johtuen toteutumaan. Olisi myös mukavaa, jos ei tarvitsisi luottaa kaikkia järjestelmän osia jonkin laitevalmistajan toimittaman ajurin koodin armoille, vaan ajuri voisi luotsata ainoastaan sitä laitetta, jonka laiterajapinnan abstrahointiin se on nimenomaisesti tarkoitettu.

Mikroytimessä on tapahduttava vähintäänkin prosessien ja säikeiden vuoronnus, keskeytysten käsittelyt (mukaanlukien järjestelmäkutsujen kutsurajapinta) sekä luvussa 0.9 kuvaillut IPC- ja synkronointimenettelyt (jotta käyttöjärjestelmän palvelut pääsevät pyytämään toisiltaan tarpeellisia palveluita sekä lukitsemaan resursseja). Muut palvelut algoritmeineen ja tietorakenteineen voisivat periaatteessa toimia omissa ”karsinoissaan” käyttäjätilan prosesseina.

Jokainen prosessin vaihto ja jokainen käyttöjärjestelmäkutsun kautta kulkeva IPC-viesti vaatii kirjanpitoa, johon kuluva aika on pois hyötylaskennasta. Kuten luvussa 0.2 todettiin, käytännössä on haettava kompromissi, joka toimii riittävän tehokkaasti, mutta jossa osa palveluista hoidetaan rajoitetussa käyttäjätilassa, missä mm. palvelukohtaisten oikeuksien ja vuoronnusprioriteettien määrittäminen on mahdollista.

Esimerkki: Lähdekoodin moduulijako

Kurssin konkreettisena esimerkkinä olevan Linux-ytimen version 4.5 lähdekoodi jakautuu päätasolla muutamiin alihakemistoihin, joiden merkitykset voidaan tässä vaiheessa ymmärtää. Seuraava

lista on alkuperäisen aakkosjärjestyksen lisäksi jaoteltu kurssin kannalta merkityksellisimpiin. Näihin hakemistoihin sijoitettuja toimintoja käsiteltiin enimmäkseen:

```
Documentation - runsaasti dokumentaatiota tekstitiedostoina
arch          - tiettyihin tietokonearkkitehtuureihin sidottu koodi
block         - lohkolaitteiden käyttö
drivers       - ajurit monenlaisille laitteille
fs            - tiedostojen käsittelyn yleinen rajapinta ja useiden
                konkreettisten tiedostojärjestelmien toteutuksia
include       - otsikkotiedostot, mm. matalimman tason C-kieliset
                käyttöjärjestelmäkutsut, joiden päälle voi tehdä
                Linuxissa toimivan sovellusohjelman (tai mieluummin
                astetta korkeamman tason yhteensopivuuskirjaston)
init          - järjestelmän ylösajo: mitä Linux tekee ensitöikseen
ipc           - prosessien välinen kommunikaatio (IPC), mm. semaforit,
                viestijonot, jaettu muisti
kernel        - "ytimen ydin" eli mm. vuoronnukseen, jäljittämiseen
                ja virheiden käsittelyyn liittyvää koodia
mm            - muistinhallinta
scripts       - Linuxin lähdekoodin konfigurointiin ja kääntämiseen
                tarkoitettuja apuohjelmia toteutettuna skripteinä
                (ainakin Bash-, Python- ja Perl-kielillä)
sound         - audiolaitteiden yleinen rajapinta
```

Seuraaviin liittyvät asiat pääasiassa ohitettiin, mutta toteutukset olennaisiin, muilla kursseilla tarkemmin käsiteltäviin, osa-alueisiin:

```
certs         - kryptattujen avaimien hallintaa (''trusted keyring'';
                ohitetaan tällä kurssilla)
crypto        - kryptografian tarpeet (ohitetaan tällä kurssilla)
net           - tietoverkon käyttö; yleisen TCP/IP -protokollan toteutus
                ja verkkolaitteiden ohjaukseen liittyvää koodia
                (ohitetaan tällä kurssilla)
security      - tietoturvan hallinta (ohitetaan tällä kurssilla)
virt          - ydintason virtualisointi (ohitetaan tällä kurssilla)
```

Edellä lueteltujen lisäksi ytimen päähakemisto sisältää myös seuraavat hakemistot, joiden sinänsä tärkeä rooli on hieman etäämpänä tällä kurssilla käsitellyistä aihepiireistä:

```
firmware      - laitteiden kontrollilogiikalle syötettäviä ohjelmistoja,
                mikäli ovat vapaasti saatavilla; osa ei ole lähdekoodina
                vaan laitevalmistajan valmiina ''binäärimöykkynä''
lib           - vaikuttaisi sisältävän monien "korkeamman tason"
                tehtävien toteutuksia, mm. pakkausalgoritmeja,
                toimintojen testausta, RAIDiin ja
                MPI-rinnakkaislaskentaympäristöön liittyviä koodeja...
samples       - esimerkki- ja testikoodia; ilmeisesti lähtökohdaksi mm.
                ajurikehitykseen
tools         - debuggaus-, testi- ja seurantatyökaluja
usr           - työkalut järjestelmän ylösajovaiheen väliaikaisen
                tiedostojärjestelmän rakentamista varten
```

Huomataan, että Linux-ytimen lähdekoodi on jaettu korkeimmalla tasolla hakemistoihin käyttöjärjestelmän tyypillisten tehtäväkokonaisuuksien mukaisesti. Lisäksi mukana on erikseen ohjelmistokehitykseen liittyviä työkaluja omissa hakemistoissaan.

Yhteenveto: Vuorontamisessa voidaan noudattaa erilaisia periaatteita, joista on valittava tilanteeseen sopivin. Prioriteettijonoilla voidaan suorittaa kiireellisempiä prosesseja useammissa tai pidemmissä aikaikkunoissa kuin alemman prioriteetin prosesseja. Prioriteetteja voidaan muutella dynaamisesti heurististen oletusten perusteella kokonaisuuden tehostamiseksi. Prosessorin lisäksi muitakin resursseja täytyy vuorontaa, esim. kovalevyn luku- ja kirjoitusoperaatioita, joiden nopeuteen (läpivientiin ja vasteaikoihin) vaikuttaa tietojen sijainti pyörivän levyn pinnassa.

Reaaliaikajärjestelmän tulee kyetä toimimaan todellisessa ympäristössä muiden ilmiöiden sanelemassa "reaaliajassa". Vaatimusten täyttäminen riittävällä tasolla on kontekstiin sidottu, ja esimerkiksi

normaali interaktiivinen tietokoneen käyttö ei edellytä reaaliaikaisuutta toisin kuin multimedian toistaminen ilman katkoksia. Reaaliaikajärjestelmän toteutuksen täytyy tukea pre-emptiivisyyttä, eli sitä, että korkeamman prioriteetin toimenpide pystyy keskeyttämään alemman prioriteetin toimenpiteen heti, kun tilanne tulee ajankohtaiseksi.

Käyttöjärjestelmän palvelut voivat toimia osittain käyttäjätilassa modulaarisuuden ja turvallisuuden saavuttamiseksi. Minimalistista käyttöjärjestelmätilassa toimivaa ydintä sanotaan mikroytimeksi ja tämän vastakohtaa, kokonaan käyttöjärjestelmätilassa toimivaa, käyttöjärjestelmää monoliittiseksi.

Suureen osaan ratkaisuja joudutaan hakemaan kompromissi, koska kaikkia tavoitteita ja kaikkia todellisen maailman käyttötilanteita ei voi saada palveltua yhtä hyvin samalla ratkaisulla.

0.14 Shellit ja shell-skriptit

Avainsanat: shell, shell-skripti

Osaamistavoitteet: Tämän luvun sekä kurssin demot 1, 2 ja 6 omaksuttuaan opiskelija:

- tunnistaa ja osaa luetella käyttötarkoituksia, joihin yleisesti käytetään kuorikomentojonoja (shell script, "skripti") [ydin-/arvos1]
- tuntee Bourne Again Shell -kuoren (bash) tärkeimmät (erityisesti POSIX-yhteensopivat) sisäänrakennetut komennot sekä yleisimmät unix-tyyppisiin (etenkin POSIX-yhteensopiviin) järjestelmiin asennetut apuohjelmat (grep, find, ...); osaa käyttää näitä manuaalisivujen perusteella sekä arvioida kriittisesti nettifoorumeilta löytyvien ohjeiden laatua [ydin-/arvos1]
- tuntee syntaksin, jolla perusohjelmointirakenteita (muuttujat, ehdot, silmukat, aliohjelmat) käytetään bash -kuoressa; tuntee myös kuoren käytölle ominaiset “||“ ja “&&“ -syntaksit [ydin/arvos2]
- pystyy lukemaan ja ymmärtämään yksinkertaisia bash -skriptejä muokkaamaan niitä hallitusti sekä tuottamaan itse alkeellisia skriptejä mallin tai ohjeiden perusteella [ydin/arvos2]

Shellin käyttöä ja shell-skriptiohjelmointia käydään läpi käytännön tekemisen kautta kurssin pakollisissa demoissa 1, 2 ja 6, joiden sisältö on tämän kurssin sisältöä. Laitetaan kuitenkin myös monistelehdykän puolelle joitakin yleisiä huomioita asiasta.

Mikä se shell olikaan?

Shelliä eli "käyttöjärjestelmän ydintä ympäröivää kuorta" on käytetty aivan kurssin alusta lähtien demoissa ja esimerkeissä. Luennoilla on myös nähty minimalistinen C-kielinen toteutus shell-ohjelma (minish.c). Shellin tärkein tehtävä on käynnistää ohjelmia käyttäjän pyynnöstä, ja tämän "minish" jopa osaa, vaikkei mitään muuta. Järjestelmästä riippuen shell voi olla myös graafinen, ja tietyssä mielessä työpöytämanageria kuvakkeineen voitaisiin ajatella käyttöjärjestelmän kuorena, jonka kautta käyttäjä voi hallita laitteistoaan. Käsitellään tässä kuitenkin tekstipohjaista shelliä, kuten kurssilla tähänkin asti. Etuja graafiseen nähden ovat mm.

- komentojen toistettavuus ja helppo kommunikoitavuus (komennon teksti on täsmällinen kuvaus operaatiosta, ilman tarvetta selostaa esim. "klikkaa hiiren oikeaa nappia siellä tai tuolla")
- komentojen toiminnallisuuden tarkentaminen argumenttilistalla, ilman tarvetta "klikkailla" toivottua toimintaa sadan valintaruudun kokoisesta ohjausikkunasta.
- skriptaaminen, ts. usein toistuvien toimintosarjojen helppo kerääminen omiksi pienoisapuohjelmiksi
- (opiskelun mielessä) tietokoneen toiminnan ymmärtäminen – tekstimuotoinen shell "piilottaa" tai "kaunistelee" hyvin vähän siitä, miten tietokone ja käyttöjärjestelmä toimii ns. konepellin alla.

Jotkut shellin kautta tehtävistä komennoista vastaavat hyvin läheisesti vastaavien käyttöjärjestelmäkutsujen tai niiden C-kielisten rajapintojen määritelmiä. Joillakin shellin komennoilla on identtinen nimi käyttöjärjestelmäkutsun kanssa, esim. "exit", "exec",

“chmod“. Ytimen lähellä toimivien komentojen lisäksi shellin kautta voi suoraan käynnistää myös korkeammalla tasolla toimivia yleiskäyttöisiä tekstipohjaisia sovelluksia, joista perinteisimmät ja yleisimmin käytetyt ovat vakiintuneet siinä määrin, että esimerkiksi käyttöjärjestelmien rajapintastandardi POSIX edellyttää näiden yleisten apuohjelmien tai vastaavien sisäänrakennettujen shellkomentojen olemassaolon yhteensopivassa järjestelmässä.

Myös monet laajemmat ohjelmistot tehdään siten, että niissä on tekstipohjainen, argumenteilla toimiva koneisto-osa, jolle generoidaan käskyjä erillisenä ohjelmana toteutetun graafisen käyttöliittymän (GUI) kautta. Ohjelman tulostamat vastaukset tietysti luetaan GUIn puolella ja näytetään käyttäjälle sopivasti. Esimerkiksi IDEt toimivat usein näin. Alla on jokin kääntäjä (esim. gcc, javac, ...), linkkeri, make-järjestelmä, versionhallinta (esim. git, mercurial, svn, ...), debuggeri (esim. gdb) ym., joita käskytetään IDEstä käsin. Alla oleva softa vastaa tekstimuotoisiin komentoihin, joita täten voi mm. skriptata. Tekstipohjaisia työkaluja on myös kuvankäsittelyyn (esim. ImageMagick) ja audiotyöskentelyyn (esim. Ecasound). Tällä tavoin ohjelmisto varmasti jakautuu erikseen toimintalogiikkaan ja käyttöliittymään, joiden välille määrittyy rajapinta. Toimintalogiikan (osat) tai käyttöliittymän toteutuksen voi vaihtaa päikseen, kunhan komentojen ja tulosteiden muodot ovat samat[72].

Merkittäviä shellejä ovat olleet mm. Bourne Shell (sh), C Shell (csh), Korn Shell (ksh) ja zsh sekä nykyisin varsin suosittu GNU Bourne Again Shell (bash). Paljon muitakin shellejä on kehitetty.

---

[72]Esimerkiksi demoscene-harrastajat voivat Windows-natiiveja pikkuruisia demoja tehdessä vaihtaa Visual Studion asetuksista linkkeriohjelman tilalle ”crinkler”-nimisen vastineen, joka mm. pakkaa ohjelman automaattisesti ja tekee koko-optimointeja, todennäköisesti rikkoen sovittuja tiedostoformaatteja automaattisesti paikoissa, joissa nykyinen Windows-toteutus ei huomaa tai välitä. Crinkleriä voi ohjata aivan samoilla argumenteilla kuin Visual Studion omaakin linkkeriä.

Pääpiirteissään ne toimivat hyvin samalla tavoin. Syntakseissa ja ominaisuuksissa on eroa, mutta "pienimmän yhteisen nimittäjän" määrittää esimerkiksi POSIX.

Muutamat tämän kurssin esimerkeistä on mukavuussyistä toteutettu Free Software Foundationin GNU bashin laajennetuilla ominaisuuksilla ja GNU:n omilla työkaluohjelmilla, koska GNU-kalusto on asennettu oman yliopistomme suorakäyttökoneisiin. Varsin suuri osa esimerkeistä on kuitenkin pysytellyt POSIXin mukaisissa, kaikkein yleismaailmallisimmissa, shell-komennoissa ja -apuohjelmissa

Shell-skriptit

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, jonka nimi on **skripti** tai tarkemmin **shell-skripti**. Skriptiksi voidaan nimittäin yleisesti sanoa yleispätevämmilläkin tulkattavilla kielillä (Perl, Python ym.) tehtyjä pieniä apuohjelmia, joita ei laajuutensa vuoksi voisi sanoa ihan sovelluksisi.

Miksi esimerkiksi voisi haluta tehdä skriptejä:

- Usein tarvittavat, samanlaisina toistuvat komentosarjat: Esim. tiedostokonversiot, ohjelmistojen julkaisut asiakkaan palvelimille tai varmuuskopioinnit on mukava sijoittaa skriptiin, joka voidaan suorittaa aina tarvittaessa samanlaisena kuin aina ennenkin.
- Ajoitetut tehtävät (esim. varmuuskopiot klo 05:30) voidaan kirjoittaa skriptiin, joka suoritetaan automaattisesti tiettyyn

aikaan (ajoitusapuohjelmalla, luonnollisestikin; POSIX määrittelee tähän komennot “at“ ja “crontab“).

- Konfigurointi: Esim. käyttöjärjestelmän palveluiden ylösajo voi tapahtua skriptillä, jossa määritellään palveluprosesseille järjestelmäkohtaiset argumentit ja käynnistetään vain ne, joita paikallisessa järjestelmässä tarvitaan. Oman pääteyhteyden voi kustomoida skriptillä, joka suoritetaan automaattisesti aina kirjautumisen yhteydessä (esim. kotihakemiston tiedosto /.bash_profile).

- Ohjelmien käynnistäminen, jos ne tarvitsevat vaikkapa joitakin ennakkotarkistuksia, ympäristömuuttujien asettamista, tiettyjen argumenttien lisäämistä käynnistyskomennon perään tai muita valmisteluja. Perusesimerkkinä demossa 6 tutkittiin yliopistomme suorakäyttökoneen gcc -kääntäjän käynnistämistä C99:n roolissa skriptillä nimeltä “c99“.

- Ohjelmistoasennukset ja käännösten esivalmistelut: Skripti voi komentojen ja apuohjelmien avulla tutkia järjestelmän asetuksia, tarkistaa, että kaikki tarvittava on asennettuna, ja tarvittaessa muokata tapaa, jolla asennuksen/käännöksen myöhemmät vaiheet toteutetaan.

- Tekstimuotoisen datan analysointi on mielekästä aloittaa tyypillisillä shell-työkaluilla, joilla datan rakennetta ja sisältöä voi tutkia ennen siirtymistä spesifien analyysityökalujen käyttöön. Osa ensimmäisistä kokeiluista voi päätyä skripteiksi, joilla asiakkaalta saapuva data muokataan automaattisesti omien laskentaohjelmien formaattiin.

- Myös binäärimuotoisen (siis muun kuin tekstin) datan tutkimiseen esim. shellin kautta helposti käytettävät heksavedostyökalut ovat verraton työkalu, mikäli datan alkuperä ja

tarkka tiedostomuoto on tuntematon. Vedoksesta voi päätellä paljon, vaikka muunnokset olisi sitten tehtävä edistyneemmillä työkaluilla.

- Toiminnallisuuksien nopeat testaukset: esim. miten WWW-palvelin vastaa tiettyyn pyyntöön.

Muitakin käyttötarkoituksia voi olla näiden lisäksi, jotka tämän kirjoittajalle tuli ensimmäisenä mieleen. Perus-shellin käyttö on perusteltua, jos ei voida olettaa että hienompia alustoja olisi asennettu koneelle, jossa skriptit tarvitsee ajaa. Esim. bash on saatavilla Linuxin lisäksi monille muillekin käyttöjärjestelmille, mukaanlukien Windowsille. Jos pitäytyy POSIXin osion ”Shell and utilities” määrittämissä ominaisuuksissa ja käyttää vain yleisimpiä apuohjelmia, on siirrettävyys vieläkin varmempi, jopa ilman tarvetta asentaa ylimääräisiä ohjelmia. Monimutkaisempiin sovelluksiin on tietysti suositeltavaa käyttää shellin sijasta jotakin tulkattavaa kieltä, joka on asennettavissa useille alustoille, esim. Perl ja Python ovat suosittuja.

Skriptejä tehdessä on syytä olla huolellinen ja huomioida erityistapaukset ja -tilanteet! Oikeaoppinen skripti toimisi kuin mikä tahansa tekstipohjainen sovellusohjelma, jota voi ohjailla komentoriviargumenteilla. Se mm. tarkistaisi etukäteen, etteivät sen tekemät toimenpiteet tuhoa tietoja vahingossa, ja ilmoittaisi virhetilanteista täsmällisesti. Näinhän tulee ohjelmoidessa toimia aina muutoinkin, mutta shell-skriptien osalta toimitaan kohtalaisen suoraan käyttöjärjestelmän rajapinnan päällä, eikä välissä ole laajaa alustakirjastoa tarjoamassa tuplavarmistuksia ohjelmoijan hölmöilyjä vastaan[73].

---

[73]Mm. tiedostojen atominen lukitseminen varmuudella shell-skriptin omaan käyttöön on aavistuksen verran haastavaa, mutta käyttötarkoituksesta riippuen tietoturvan kannalta välttämätöntä. Laitteiston, käyttöjärjestelmän, ja potentiaalisesti pa-

Yhteenvetoa shell-demoista

Tähän on listattu komennot, joita kurssin käytännön harjoitteissa eli pakollisissa demoissa käytiin läpi ja kokeiltiin omin käsin. Toivottavasti niistä sai käsityksen interaktiivisen shellin ja shell-skriptien mahdollisuuksista. Komennot on jaoteltu kahteen alilukuun, joista ensimmäisessä on POSIXin määräämät ja jälkimmäisessä muut. Järjestys on jotakuinkin sama kuin missä komennot tulivat vastaan demoissa 1-6. Aakkosellinen lista noin 160 yleisestä komennosta löytyy esimerkiksi POSIXin osiosta "Vol 3: Shell and Utilities". Tällä kurssilla katsottiin esimerkin vuoksi lyhyesti noin 30 komentoa.

POSIXin mukaisia komentoja, joita demoissa tehtiin

Demot tehtyäsi olet vähintään kerran käyttänyt omin käsin seuraavia POSIXin määräämiä komentoja, joista kukin voi olla toteutettu erillisenä apuohjelmana tai tehokkuussyistä shell-ohjelmaan sisäänrakennettuna (POSIX-komento "type" muuten kertoo, onko tietty komento sisäänrakennettu[74]):

```
uname -- tulosta yleisiä tietoja järjestelmästä

who   -- tulosta tietoja muista samanaikaisista käyttäjistä

ps    -- tulosta tietoja käynnissä olevista prosesseista

pwd   -- tulosta nykyisen työskentelyhakemiston tiedosto-osoite

ls    -- tulosta lista työskentelyhakemiston tai jonkin muun,
```

hantahtoisten käyttäjien suunnasta kumpuavien ongelmien miettiminen kuitenkin toivottavasti johtaa ohjelmointitaitojen kehittymiseen kaikilla kielillä ja alustoilla. Jatkokurssit Ohjelmistoturvallisuus ja Tietoverkkoturvallisuus on syytä käydä huolella.

[74]Luennollakin olisi näytetty esim. "type echo", mutta oli siinä kohtaa oikosulku, että mikähän komento se taas olikaan... Googlella olisi tietysti parissa lisähetkessä löytynyt, ja POSIXistakin voinut tarkistaa.

```
         argumenttina määritellyn, hakemiston sisältämistä tiedostoista

echo  -- tulosta argumentit välilyönnillä erotettuna

cat   -- tulosta argumenttina annettujen tiedostojen sisällöt
         peräkkäin (ts. ``katenoi'' ne)

man   -- näytä jonkin komennon käyttöohjeet

cd    -- vaihda työskentelyhakemistoa

mkdir -- tee uusi hakemisto

sort  -- lajittele syötteen rivit; tulosta lajitellussa järjestyksessä

cut   -- leikkaa osia syöteriveistä; tulosta vain leikatut osat

uniq  -- tulosta syötteen rivit, paitsi ei niitä, jotka ovat täysin
         sama kuin edellinen rivi

wc    -- laske syötteestä sanat/rivit/merkit ja tulosta lukumäärä

grep  -- tulosta syötteen rivit, jotka vastaavat argumenttina annettua
         RegExp-merkkijonohahmoa

chmod -- muuta tiedoston käyttöoikeuksia eli "moodia"

od    -- tulosta syötteen tavut oletuksena oktaalinumeroina
         ("oktaalivedos" / "octal dump") tai muussa, argumentein
         säädettävässä, formaatissa

file  -- yritä päätellä argumenttina annetun tiedoston tyyppi;
         onko se esim. suoritettava ohjelmatiedosto, tekstitiedosto jne.

true  -- ohjelma, joka "ei tee mitään ja onnistuu siinä". Siis
         olennaisesti suorittaa käyttöjärjestelmäkutsun exit(0).

sed   -- lukee tekstirivejä ja tulostaa ne automaattisesti käsitellyssä
         muodossa (mainittiin demossa; luennolla pieni esimerkki, jossa
         vain muutettiin pilkut välilyönneiksi)

awk   -- käsittelee tekstiä automaattisesti; awkia ohjataan sen omalla,
         tekstinkäsittelyn tarpeisiin suunnitellulla ohjelmointikielellä
         (mainittiin demossa; ei käsitelty kurssilla mainintaa pidemmälle)

vi    -- "visuaalinen tekstieditori", jonka täytyy aina löytyä POSIXin
```

```
          mukaisesta järjestelmästä.

          Käytännössä järjestelmiin on nykyään asennettu avoimen lähdekoodin
          toteutus nimeltä Vim ("Vi IMproved"), joka toteuttaa POSIXin
          määräämät ominaisuudet ja paljon muuta.

          Vim on esitelty erittäin hyvin suomeksi seuraavassa
          kuusikilotavuisessa dokumentissa:
          http://vim.sourceforge.net/6k/features.fi.txt

          Tässä ei tarvinne siis enempää selostella.

curl   -- tulosta palvelimen vastaus argumenttina annetun URL:n
          perusteella (luennolla nähtiin esimerkki sääennusteen
          hakemisesta URLista http://wttr.in/)

c99    -- käännä C99 -standardin mukainen C-kielinen lähdekoodi

export -- julkaise ympäristömuuttuja, joka näkyy myös lapsiprosesseille

locale -- tulosta tietoja "paikallisuus-" eli kieliasetuksista ("locale")

make   -- rakenna ohjelmisto Makefile -tiedostoon kirjoitettuja ohjeita
          tai oletusarvoja noudattaen; tarjoaa perusautomatiikan ohjelman
          osien kääntämiseen ja linkittämiseen; POSIX antaa suosiolla
          järjestelmän päättää monista laajennoksista makeen. Esim. GNUn
          make on hyvin monipuolinen POSIXin perusvaatimuksiin nähden.
```

GNUn ja muiden laajennoksia, joita demoissa käytettiin

Seuraavia GNU-, BSD- ym. laajennoksia käytettiin demoissa mukavuussyistä. Niitä ei välttämättä ole saatavilla standardin mukaisessa POSIX-järjestelmässä ilman lisäasennuksia. Kaikki ohjelmat on kuitenkin mahdollista asentaa POSIXin päälle:

```
whoami  -- tämähän oli ensimmäinen komento koko demoissa.. näyttää
           niin kivalta ja selkokieliseltä ("kukaminäolen" eli
           "whoami")... Sovelluksen manuaalisivu kuitenkin paljastaa,
           että POSIX-yhteensopivasti tämä pitäisi tehdä komennolla
           ``id -un``.

           Tulevissa omissa softissasi kirjoitat tietysti aina ``id
```

```
         -un``, jos haluat olla POSIX-yhteensopiva. Jos rakennat
         GNU-apuohjelmien päälle, ``whoami`` on OK samaan
         tarkoitukseen, mutta  sitten olet kiinni yhdessä tietyssä
         toteutuksessa, joka ei (ainakaan vielä) ole standardoitu.

finger  -- tämä oli jotakuinkin toinen komento, joka tehtiin.. aina
         on kiva 'sormeilemalla' tietää, mitä tietoja kaverista on
         saatavissa. Kuitenkaan tämä toiminnallisuus ei ole esim.
         POSIXissa määritelty. Suorakäyttökoneemme finger-sovellus
         näyttäisi olevan alkujaan kotoisin BSD:stä (Berkeley
         Software Distribution)

less    -- tämä on vain niin näppärä ohjelma omaan tarkoitukseensa.

         POSIX määrittelee, että täytyy löytyä shell-komento
         nimeltä ``more``, joka näyttää syötteensä sivu kerrallaan.

         Mark Nudelman teki teki vuonna 1984 ohjelman nimeltä
         ``less``, joka näyttää syötteensä sivu kerrallaan, mutta
         siten, että päätekäyttäjä voi kelata tulostetta ylös ja
         alaspäin interaktiivisesti.

         Manuaalisivu alustaa ohjelman merkityksen seuraavalla
         tavoin: "less - opposite of more"...

         Wikipedia on tietävinään, että "less" on asennettu
         suurimpaan osaan nykyisistä Unix-maisista
         käyttöjärjestelmistä. Viimeisimpään POSIX-standardiin
         se ei kuitenkaan ole (vielä) päätynyt, eikä välttämättä
         päädykään, koska jo ``more`` tarjoaa minimaalisen
         ratkaisun ongelmaan, eli pitkän tulosteen sivuttamiseen.

top     -- reaaliaikainen seuranta eniten resursseja vaativista
         prosesseista

stat    -- antaa tarkkoja tietoja tiedostosta. POSIX määrää
         C-kielisen rajapinnan järjestelmäkutsulle fstatat()
         mutta ei shell-rajapintaa. GNU:n perustyökaluihin
         stat kuitenkin kuuluu myös shell-komentona.

hexdump -- tulostaa tavuja, esim. tiedostoja, heksavedoksena.
         Kurssin esimerkeissä käytettiin argumenttia "-C"
         eli "canonical format", jonka tulosteessa on nätisti
         tavujen osoitteet syötetiedon alusta laskien, tavut
         heksalukuina sekä selväkieliset merkit tavuista,
         jotka ovat "tulostettava" ASCII-merkkejä eli arvoalueella
```

```
          0x20--0x7f.

objdump -- tulostaa tietoja ELF-tiedoston sisällöstä ihmisen
           ymmärtämässä muodossa; voi tutkia esimerkiksi suoritettavia
           konekielisiä ohjelmia ja C-kielestä käännettyjä
           objektitiedostoja. Mm. Linux-ydin käyttää ELFiä.

nano    -- minimalistinen tekstieditori, joka on helppokäyttöinen,
           mutta jossa ei ole juuri mitään ominaisuuksia. GNU-projektin
           "vapauttama" versio suljetun lähdekoodin pico -editorista.

emacs   -- GNUn tekstieditori, jolla voi tehdä mitä vaan (nettisivun
           mukaan emacs on tarkoitettu mm. projektisuunnitteluun,
           sähköpostiohjelmaksi, debuggerin rajapinnaksi, kalenteriksi).

           Vanhan vitsin mukaan emacs on "hyvä käyttöjärjestelmä, josta
           puuttuu vain tekstieditori". Vitsin kehtaa laukaista, koska
           emacs on tunnetusti ihan hyvä väline myös tekstin editointiin.

           Historiaa emacsin omalta nettisivulta: Alkuperäinen emacs eli
           "Editor MACroS" kirjoitettiin 1976 MIT:n tekoälylaboratoriossa.
           Richard Stallman kirjoitti emacsin uudelleen GNUta varten 1984.
           Vuonna 2016 viimeisin versio on kahdeskymmenesneljäs.

           Rakenteeltaan emacs on olennaisesti LISP-ohjelmointikielen
           tulkki. Jokainen näppäinpainallus laukaisee ohjelman, joka
           voi lisätä merkin tekstitiedostoon kursorin kohdalle, mutta
           voi yhtä hyvin tehdä ihan mitä tahansa, mitä nyt tietokoneella
           ylipäätään pystyy tekemään. Käyttäjä voi itse laajentaa editoria
           mihin suuntaan tahansa ohjelmoimalla toimintoja LISPillä ja
           kytkemällä niitä näppäimiin.

           Netistä löytyy aika paljon erilaisia laajennoksia. 40 vuoden
           kehityksen jälkeen on vaikea keksiä ominaisuutta, jota joku ei
           olisi jo aiemminkin ehtinyt kaivata, toteuttaa ja julkaista
           muiden iloksi.

screen  -- apuohjelma, joka mahdollistaa pääteyhteyden jakamisen useisiin
           "ikkunoihin" sekä yhteyssession irtauttamisen vanhempiprosessista
           siten, että ikkunat jäävät auki, vaikka käyttäjä kirjautuu
           välillä ulos järjestelmästä (tai yhteys katkeaa muista syistä)

wget    -- noutaa tiedoston netin yli (GNUn tuotoksia. POSIX-yhteensopiva
           tapa on käyttää komentoa ‘‘curl‘‘)

gcc     -- GNUn C-kääntäjä, joka GNU/Linux järjestelmässä käynnistyy myös
```

```
            komennolla c99 (demossa 6 katsotaan, mitä komento c99 tarkkaan
            ottaen tekee; osoittautuu, että se on itse asiassa pieni skripti)

gdb      -- GNUn debuggeri

javac    -- Java-kielen kääntäjä

java     -- käynnistää Java-virtuaalikoneen ja suorittaa ensimmäisenä
            argumenttina annetun luokan julkisen luokkametodin
            main(String[] args), sijoittaen luokan nimen jälkeiset
            argumentit parametrinä viitattavaan taulukkoon ‘‘args‘‘.

unzip    -- avaa ZIP-muotoisen pakatun tiedoston (käytettiin tätä kurssin
            demoissa esimerkkikoodien pakkausformaattina, kun ZIP on väelle
            ehkä tutumpi; Unix/Linux-maisempaa olisi käyttää gzip -muotoa,
            ja sen sisällä tar -arkistointiohjelman muotoa; POSIX määrää
            kyllä ZIP/gzip -tyyppisen Lempel-Ziv -pakkauksen tekemiseen ja
            purkamiseen komennot ‘‘compress‘‘ ja ‘‘uncompress‘‘, joita kyllä
            ainakaan tämän monisteen kirjoittaja ei ole koskaan nähnyt
            faktisesti käytettävän missään; sen sijaan tar+gzip -yhdistelmä
            sekä mahdollisesti paremmin pakkaava tar+bzip -yhdistelmä ovat
            varsin universaalissa käytössä. Paketit avataan käytännössä
            esim. komennolla ‘‘tar -xvzf paketti.tgz‘‘ tai putkittamalla
            ‘‘zcat paketti.tgz | untar‘‘).
```

POSIXin mukaisia komentorivisyntakseja

```
|          -- putken luominen peräkkäisten komentojen välille

>          -- ohjaa komennon tulosteen tiedostoon korvaten aiemman
              tiedostosisällön

>>         -- ohjaa komennon tulosteen tiedostoon aiemman
              tiedostosisällön perään

2>         -- ohjaa komennon virhetulosteet ("tietovirta 2" eli stderr)
              tiedostoon

>&2        -- ohjaa komennon normaalin tulosteen eteenpäin virhetulosteena
              (käyttöesimerkki nähtiin demossa 6)

;          -- erottaa komentoja; vastaa rivinvaihtoa, ts. mahdollistaa
```

```
              useat komennot samalla rivillä

&          -- erottaa komentoja; käynnistää edeltävän komennon tausta-ajoksi.
              Shelliin voi antaa heti uusia komentoja, vaikka tausta-ajon
              suoritus olisi kesken.

&&         -- erottaa komentoja; seuraava suoritetaan vain, jos edellinen
              päättyy onnistumista ilmaisevalla virhekoodilla 0.

||         -- erottaa komentoja; seuraava suoritetaan vain, jos edellinen
              päättyy epäonnistumista ilmaisevaan virhekoodiin (muu kuin 0)

#          -- kommenttimerkki; shell ei tulkitse risuaidan ja
              rivinvaihtomerkin välissä olevia merkkejä

#!         -- de facto esim. Linuxissa tiedoston ensimmäisellä rivillä
              voi olla #!, jolloin rivin loppuosa tulkitaan komennoksi ja
              argumenttilistaksi, jolle tiedoston loput rivit syötetään.
              (ei ole sallittu tiukasti POSIX-yhteensopivassa skriptissä!)

$nimi      -- nimetyn muuttujan arvon sijoittaminen komentoriville ennen
              suorittamista; kaikki shellin muuttujat ovat merkkijonoja;
              olemassaoloa ei oletuksena tarkisteta, vaan alustamattoman
              muuttujan kohdalle korvautuu tyhjä merkkijono (ts. ei mitään)

nimi=arvo -- muuttujan arvon asettaminen (syntaksissa ei ole välilyöntejä);
              muuttuja näkyy vain nykyisessä shell-sessiossa, ellei sitä
              erikseen julkaise ympäristöön export -komennolla.

unset nimi -- muuttujan poistaminen kokonaan

$0         -- skriptin nollas argumentti, eli komento, jolla shell on
              käynnistetty; tämä muuttuja on aina automaattisesti olemassa.

$?         -- viimeisimmän komennon palauttama virhekoodi; tämäkin on
              automaattisesti käytettävissä

for        -- for-silmukkarakennetta katsottiin alustavasti demossa 6

case       -- case-valintarakennetta katsottiin alustavasti demossa 6
```

Näiden lisäksi, mitä ehdittiin nähdä, on shellissä mahdollista muutkin normaalit ohjelmointirakenteet, kuten if-lauseet ja aliohjelmat.

Tarkka kuvaus shellin syntaksista ja rakenteista löytyy tietysti POSIXin shell-osuudesta. Standardin teksti on "tikkuista", koska se vain toteaa, miten asian tulee olla. Aiheen opiskelu on varmasti helpompaa tutoriaaleista, joita netistä löytyy paljon. Tutoriaaleissa ja netin foorumeilla yleensä käytetään GNU bashiä, joten standardista täytyy tarvittaessa vilkaista, onko jokin toiminnallisuus yleispätevä kaikissa shelleissä ja käyttöjärjestelmissä vai täytyykö skriptin käyttäjällä olla nimenomaan GNU-työkalusto asennettuna.

GNUn bashin laajennettua syntaksia

Seuraava syntaksi vilahti demossa esimerkkinä interaktiivisen shellin näppäryydestä. Mainittakoon, että se on GNUn bashin laajennos, joka ei varmasti toimi kaikissa muissa shelleissä:

```
hak{1,2,3,4} -- bashin oma syntaksi, joka avautuu komentoriville
                useaksi eri merkkijonoksi (tässä "hak1", "hak2",
                "hak3" ja "hak4").

                Helppo tapa esimerkiksi vaihtaa tiedostopääte tai muu
                osuus tiedostonimestä:

                mv kuva.{JPEG,jpg}

                ... suorittaisi komennon "mv kuva.JPEG kuva.jpg", joka
                vaihtaa tiedoston nimen nätimmäksi ja yhteensopivammaksi
```

Lisätietoa shelleistä

Kurssin pakollisissa demoissa käytiin esimerkkien kautta läpi shellin perusideaa ja kaikkein yleisimpiä komentoja. Vapaaehtoisissa demoissa 5 ja 6 on linkit muutamaan WWW:stä löytyvään tutoriaaliin, ja muitakin hyviä löytyy todella paljon, mikäli aiheen opiskelu alkaa kiinnostaa enemmänkin. Täytyy kuitenkin muistaa, että vähänkään laajempien ohjelmien tekemiseen edistyneemmät

tulkattavat kielet tarjoavat paljon POSIX-shelliä tai bashiakin paremmat mahdollisuudet.

Yhteenveto: Käyttöjärjestelmää komennetaan shellin kautta lähellä käyttöjärjestelmän rajapintoja. Yksi shellin tyypillinen käyttötarkoitus on tietokoneen interaktiivinen käyttö yksittäinen komento kerrallaan. Toinen tyypillinen käyttötarkoitus ovat shellskiptit eli tiedostoon peräkkäin kirjoitetut komennot, jotka shell osaa tulkita ja suorittaa. Shellin ohjelmointi ja toimenpiteiden automatisointi edellyttää jonkinlaista tekstiin perustuvaa komentokieltä, vaikka interaktiivinen käyttöliittymä olisikin graafinen, hiirellä klikkailtava. POSIX määrittelee ominaisuudet ja käytettävissä olevat ohjelmointirakenteet shellille nimeltä “sh“, joka perustuu aikoinaan eri unix-järjestelmissä käytössä olleisiin shelleihin, poimien niiden yhteisiä hyväksi havaittuja piirteitä.

Skriptejä käytetään mm. usein tehtävissä komentosarjoissa, ajoitetuissa tehtävissä, konfiguroimisessa, ohjelmistoasennuksissa sekä ohjelmien käynnistämisessä, mikäli ohjelmat vaatisivat joitakin valmisteluita. Skriptejä tehdessä tulee huomioida erityistapaukset ja -tilanteet huolellisesti, kuten ohjelmoinnissa yleensäkin.

0.15 Epilogi

Tämän kurssin tavoitteena oli saada opiskelija ymmärtämään tietokonelaitteiston ja käyttöjärjestelmän rajapintoja, jotta ohjelmistoja kehittäessä olisi mielikuvamalli siitä, mitä ohjelman suorittamiseen viimekädessä liittyy alustan pohjimmaisilla tasoilla.

Käyttöjärjestelmän merkitystä, periaatteita ja toteutusratkaisuja esiteltiin yksinkertaistettujen teoreettisten yleiskuvausten sekä konkreettisten koodi-, kuori- ja skriptausesimerkkien kautta. Konkreettisina esimerkkiartefakteina käytettin Linuxin ydintä lähdekoodeineen, POSIX-standardia, bash-kuorta, GNU-työkaluja, C-kieltä ja AMD64-prosessoriarkkitehtuuria.

Yhteenveto

Toivottavasti nähtiin ja ymmärrettiin, että vaikka tietokone (edelleenkin, jopa aikojen saatossa syntyneine lisäteknologioineen) on pohjimmiltaan yksinkertainen laite, logiikkaportteihin perustuva bittien siirtäjä, on siihen ja sen käyttöön aikojen saatossa kohdistunut uusia vaatimuksia ja ratkaistavia haasteita. Tuloksena on nykyisellään laaja ja monimutkainenkin 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”. Spesifien julkaisufoorumeiden lisäksi käyttöjärjestelmätutkimuksen tuloksia julkaistaan luonnollisesti myös yleisemmissä ohjelmistoalan lehdissä.

Laajemmin ajateltuna käyttöjärjestelmiin liittyvät myös kaikki yhdenaikaisen ohjelmoinnin eli prosessien ja säikeiden yhteistoimin-

nan kysymykset, kuten kilpa-ajotilanteiden hallinta, viestinvälitys prosessilta toiselle ja säikeiden ajallinen synkronointi.

Kurssin osaamistavoitteissa määriteltiin joitakin tavoitteita, joiden täydellinen saavuttaminen on mahdollista vasta koko materiaalin läpikäynnin jälkeen. Tavoite olisi, että tässä vaiheessa kurssia opiskelija täsmällisempien osaamistavoitteiden lisäksi:

- osaa luetella käyttöjärjestelmän tärkeimmät osajärjestelmät ja tehtävät, joita kunkin osajärjestelmän vastuulla on [ydin-/arvos1]
- tuntee olennaisimman käyttöjärjestelmiin liittyvän terminologian ja sanojen tunnetuimmat variaatiot suomeksi ja englanniksi; kykenee kommunikoimaan aiheeseen liittyvistä erityiskysymyksistä kirjallisesti (ja suullisesti; ei kuitenkaan verifioitavissa kirjallisella tentillä) molemmilla kielillä [ydin/arvos1]
- osaa kuvailla käyttöjärjestelmille tyypilliset abstraktiot (prosessit, resurssit, tiedostot) sekä kerrosmaisen rakenteen abstraktioiden ja fyysisen laitteiston välillä; osaa kertoa syitä abstraktioiden käyttöön laiteohjauksessa ja kykenee yleistämään abstrahointimenettelyn ohjelmistojen tekemiseen muutoinkin [ydin/arvos1]
- ymmärtää standardoinnin päämäärät käyttöjärjestelmien (ja muiden rajapintojen) yhteydessä; osaa kertoa, millaisia asioita POSIX-standardi määrittelee ja toisaalta mihin POSIX ei ota kantaa. [ydin/arvos1]
- tietää ja tunnistaa tyypillisimmät käyttöjärjestelmässä tarvittavat tietorakenteet (tietyt tietueet sekä niistä muodostuvat taulukot, jonot, listat, pinot, hierarkkiset rakenteet) sekä

tietää, mihin tarkoitukseen mitäkin niistä voidaan käyttää. Erityisesti opiskelija osaa tunnistaa ja kuvailla ne käyttöjärjestelmän tietorakenteet, joita käytetään prosessien tilan hallinnassa, sivuttavan virtuaalimuistin hallinnassa, tiedostojärjestelmän toteutuksessa sekä vuoronnuksessa. [ydin/arvos1]

- osaa kuvailla linux-lähdekoodin uusimman version hakemistorakenteen ylimmällä tasolla sekä perustella, mihin kyseinen hakemistojäsentely perustuu [edist/arvos3]

- (monien pääteyhteydellä tehtyjen demojen perusteella) muistaa apuohjelmia ja keinoja, joilla unix-tyyppisen käyttöjärjestelmän hetkellistä tilaa voidaan tarkastella pääteyhteydellä ja/tai tekstimuotoisella kuorella; kykenee etsimään WWW:sta lisätietoa vastaavien tehtävien suorittamiseen ja arvioimaan kriittisesti ohjeiden laatua ja soveltuvuutta tarpeeseen [edist/arvos3]

Meta-taitojen osalta toivottavaa on, että opiskelija myös

- tietää julkaisufoorumeita, joissa käyttöjärjestelmätutkimuksen uusia tieteellisiä tuloksia julkaistaan [ydin/arvos2]

- pystyy seuraamaan käyttöjärjestelmiin liittyvää ammattikeskustelua esimerkiksi lkml-sähköpostilistalla [edist/arvos3]

- osaa hakea oman yliopiston tilaamista tutkimustietokannoista kokotekstiartikkeleita käyttöjärjestelmien osa-alueisiin liittyen [edist/arvos3]

Kurssin sisällön ylittävänä haavetavoitteena oli mainittu, että opiskelija myös halutessaan

- kykenee muodostamaan alustavasti käyttöjärjestelmiin liittyvän kanditason tutkimussuunnitelman tai aihe-ehdotuksen [”ultimaattinen tavoite”, joka ylittää tämän kurssin sisäiset osaamistavoitteet ja parhaimmillaan muodostaa sillan konkreettisen kandidaattitutkielman puolelle]

Mainintoja asioista, jotka tällä kurssilla ohitettiin

Monet asiat käsiteltiin pintapuolisesti, koska kurssin opintopistemäärä ei mahdollista kovin suurta syventymistä. Myöskään emme voi näin suppealla kurssilla esimerkiksi teettää harjoitustyönä omaa käyttöjärjestelmää, kuten joissakin maailman yliopistoissa on tapana. Tarkoitus olikin antaa yleiskuva siitä, mikä oikein on käyttöjärjestelmä, mihin se tarvitaan, ja millaisia osa-alueita sellaisen on vähintään hallittava. Terminologiaa ja käsitteitä esiteltiin luettelonomaisesti, jotta ne olisi tämän jälkeen ”kuultu” ja osattaisiin etsiä lisätietoa itsenäisesti myöhempien opiskelu- ja työtehtävien vaatimista erityisaiheista. Pakollisissa demoissa pyrittiin antamaan käytännön käden taitoja ja lähtökohta omatoimiseen lisäopiskeluun. Vapaaehtoisissa demoissa näitä taitoja pyrittiin vielä lisäämään.

Käsittelemättä jätettiin myös joitakin suositeltuja aihekokonaisuuksia, mm.

- tietoturvaan liittyvät seikat (security / security models) (”policyt”, tietoturvalaitteet, kryptografia, autentikointi) pääasiassa ohitettiin. Meillä on nykyään useita kaikille yhteisiä kursseja (Tietoturva, Ohjelmistoturvallisuus) sekä kokonainen informaatioturvallisuuden maisteriohjelma, joissa turva-asioihin syvennytään perusteellisesti.

- Sulautettujen järjestelmien (embedded systems) erityistarpeita ei juurikaan käsitelty osa-alueiden yhteydessä. Pääasiassa käsiteltiin työasemien ja palvelimien näkökulmaa. Näitä käytäneen läpi tietoliikenteen maisteriohjelman kursseilla. Lisäksi tiedekunnassa on (2016) uusi ”Internet of things” (IoT) -laboratorio, jossa tutkitaan mm. arkipäivän laitteisiin sulautettujen prosessorien laiteläheisiä kysymyksiä ja tietoturvaa.

- Järjestelmän suorituskyvyn analysoinnista (performance evaluation) (tarpeet, menetelmät) ei ollut varsinaisesti puhetta. Aihe läpäisee koko ohjelmistokehityksen alan, joten toivon mukaan suorituskykyyn liittyviä tiedonpalasia löytyy syventäviltä kursseilta, kuten Requirements Engineering, Ohjelmistotestaus, Ohjelmistoarkkitehtuurit. Perusteoria aihepiiriin opetettaneen kanditason kursseilla Algoritmit 1 ja 2.

- vikasietoisen järjestelmän suunnitteluperusteista (fault tolerant systems design) ei ollut erityistä puhetta. Ks. edellisen kohdan jatkokurssit.

- ”Sähköisestä todisteaineistosta” (digital forensics) (kerääminen, analysointi) ei ollut puhetta. Jotkut informaatioturvallisuuden maisteriohjelman kurssit kenties tarjoavat aihepiiriin liittyvää tietoa. Jonkin verran luennoilla (2016) sattumalta sivuttiin esimerkiksi sitä, että käytöstä poistetun kovalevyn tiedot saattavat olla osittain luettavissa, mikäli niiden päälle ei ole tallennettu satunnaisia bittejä jopa muutamaan kertaan.

- moniprosessorijärjestelmän synkronoinnin yksityiskohdat, pilvipalvelut. Tästä aiheesta todennäköisesti lisää mm. kursseilla Modernien moniydinprosessorien ohjelmointi, Introduction to SOA and Cloud Computing, Hajautetut järjestelmät.

- tietoliikenteen yksityiskohdat. Tietoliikenteestä on meillä monta eritasoista kurssia nimikkeillä Tietoliikenne, Tietoliikenneprotokollat 1 ja 2, Sovellusprotokollat.
- käyttöjärjestelmän ylläpito. Tästä on ainakin syventävä kurssi Linux-virtuaalipalvelimen ylläpito.

Mitä seuraavaksi

Toivottavasti kurssi tarjosi tärkeimmän asian, eli peruskeinot käyttöjärjestelmiin ja laitteistoon liittyvien teknisten dokumenttien ja ohjeiden itsenäiseen lukemiseen ja turvallisten kokeilujen tekemiseen, aina sen mukaan, mitä tulevissa kursseissa, töissä tai harrastuksissa tuleekaan vastaan.

Hauskinta on tietysti tehdä myös vähemmän turvallisia kokeiluja, mitä tarkoitusta varten voi asentaa itselleen virtuaalikoneen, jossa voi ajaa turvallisessa ”hiekkalaatikossa” mitä vain.

Mikäli aihepiiri alkoi kiinnostaa tämän kurssin pintaraapaisun jälkeen, Internetistä löytyy runsaasti materiaalia ilmaiseksi, ja ainahan voi myös ostaa paperisia kirjoja.

Harrastelua varten esimerkiksi Cambridgen yliopistolta löytyy tutoriaali oman käyttöjärjestelmäraakileen toteuttamiseksi ARM-"prose käyttävälle Raspberry Pi -halpistietokoneelle [75]. MIT:ltä puolestaan löytyy maisteritason kurssi, jossa tehdään harjoitustyönä käyttöjärjestelmäraakile moniytimiselle 386-prosessorille[76]. Näiden kurssien tekeminen edellyttää Linux-ympäristön ja GNU-työkalujen käyttöä, joka onkin jo tältä johdantokurssiltamme tuttua.

[75] `http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/`

[76] `http://ocw.mit.edu/courses/electrical-engineering-and-computer-scien 6-828-operating-system-engineering-fall-2012/`

Viimeiset sanat

Kiitos mielenkiinnosta tällä kurssilla. Toivottavasti tästä oli hyötyä tulevia haasteita varten. Onnea ja osaamista niiden parissa!

.1 Pullantuoksuinen pehmojohdanto

Vastuuvapauslauseke: Monisteen kirjoittaja nautti aikoinaan suuresti erään kurssimonisteen Pehmojohdanto -nimisestä luvusta, joka johdatteli aihepiiriin arkihavaintojen kautta. Tämäkin luku on erään työkaverin sanoja lainaten johdattelua aiheeseen ”laulun ja leikin keinoin”. Kyseiset keinot eivät johda millään tavalla loppuun asti, kun kyseessä on niinkin täsmälliseen ja matemaattisen tarkkaan alaan kuin informaatioteknologiaan liittyvä johdantokurssi. Kuitenkin eräällä kevään 2014 kurssin opiskelijalla oli vallan hyvä idea siitä, kuinka kurssin tärkeimmät käsitteelliset ongelmat ja osa niiden ratkaisuista voidaan kuvata reaalimaailman analogiana, menemättä ensinkään tietoteknisiin yksityiskohtiin, ykkösten ja nollien maailmaan. Asiaa mietittyään kirjoittaja on lopulta samaa mieltä, joten tässä luvussa käydään läpi kurssin asioita ilman suurempaa mainintaa tietokonelaitteistosta. Tämä on omalla tavallaan hyvin sopivaa, sillä syvälliset ongelmat ratkaisuineen tuleekin pystyä näkemään käsitteellisinä ja tietyistä sovellusalueista erillisinä. Vastaavantyyppinen analogia, ”pikku-ukkolaskin” (engl. *little man computer, LMC*), on maailmalla tunnettu apuväline tietokoneen toimintaperiaatteen opettamisessa, johon viitataan alan pedagogiikkaa käsittelevissä artikkeleissa (ks. esim. [9]). Kirjoittaja pahoittelee, että alkuperäisen idean vastaisesti pehmojohdannossa esimerkkinä on kakkuleipomo eikä pyöräkorjaamo. Oli tehtävä valinta siitä, tuoksuuko johdanto pullalta vai ketjurasvalta. Pulla tuntui pehmeämmältä valinnalta.

Tietokoneen rakenne ja arkkitehtuuri -kurssin käyneet ja Ohjelmointi 1 -kurssista hyvin perillä olevat voivat silmäillä luvun läpi kursorisesti, koska todelliseen sovellukseen siirrytään vasta seuraavissa luvuissa, joissa kaikki käydään läpi uudelleen bittien ja ohjelmoinnin maailmassa. Tämä on tarkoitettu hätäavuksi niille, joilla esitietoja ei ole tai tuntuu että niiden kertaamiseen tarvitaan rau-

talankaa. Rautalanka tarjotaan tosin tällä kertaa kermavaahdon muodossa.

(Vastuuvapauslauseke päättyy)

Kakkulan kylän yksinäinen mestarileipuri

Olipa kerran, kauan, kauan sitten, vaihtoehtoisessa todellisuudessa pieni Kakkulan kylä, jonka asukkaat rakastivat tuoreita leivonnaisia yli kaiken. Eräänä päivänä kylää kohtasi suuren suuri onni: Kakkulaan saapui taitava leipuri, joka halusi omistaa elämänsä kylän asukkaiden leivoksellisten tarpeiden palvelemiselle. Leipuri oli kutsumustyössään niin taitava, ettei moista ollut aiemmin nähty. Hänen operaationsa ”Central Baking Unit (CBU)” -keittiönsä sisällä olivat taianomaisen nopeita. Oli kuin hän olisi leiponut lähes valon nopeudella. Kakut, pullat ja pikkuleivät valmistuivat aina täsmälleen reseptien mukaisesti ja kellontarkasti, tasalaatuisina. Kotileipuritkin saattoivat teettää CBU:ssa leivonnan työläimpiä osavaiheita, kuten kermavaahdon vispausta tai pullataikinan vaivaamista. CBU:n leipuri prosessoi reseptejä taukoamatta, eikä väsynyt, vaikka häneltä olisi tilattu tuhansia litroja kermavaahtoa kerralla. Kylän kesäkauden kohokohdaksi ja turistivetonaulaksi muodostuikin pian suurenmoiset kermavaahtobileet, joista vaahto ei loppunut, kiitos Central Baking Unitin palvelusten. Kylän asukkaat totesivat, että heidän ei enää koskaan tarvitsisi vaivata itseään leipomisen hankalilla ja aikaavievillä työvaiheilla, koska he saattoivat vain viedä reseptinsä CBU:n postilaatikkoon ja nauttia lopputuloksista, kun leivokset kauniina ja tuoksuvina putkahtivat ulos CBU:n lastausovesta jonkin ajan jälkeen.

Vaikkakin CBU:n leipuri oli nopea kuin sähkö, tarkka kuin kellon koneisto ja tehokkaampi kuin tuhat kotileipuria yhteensä, oli hänessä myös valitettavia huonoja puolia. Vilkaisu CBU:n seinien

sisälle paljasti asiasta kiinnostuneille karun totuuden: Leipuri oli niin omistautunut työlleen, ettei hän osannut muuta kuin noudattaa yksinomaan leipomiseen liittyviä yksityiskohtaisia ja yksinkertaisia ohjeita. Niin yksinkertainen hän oli, ettei edes pystynyt päässään muistamaan, mitä oli juuri äsken tehnyt, mistä oli tulossa tai mihin oli menossa. Aina kun uusi resepti putosi CBU:n luukusta, leipuri kävi uuden reseptin läpi kirjain kirjaimelta ja kopioi sen muistivihkonsa puhtaille sivuille. Jos vihkossa ei ollut yhtään puhdasta sivua jäljellä, pyyhki leipuri ensin pois jonkin aiemman reseptin, jonka valmistus oli päättynyt kauan aikaa sitten. Tätä kopiointiakaan hän ei olisi osannut tehdä, ellei hänellä olisi ollut vihkonsa takasivuilla erityiset ohjeet kopiointia ja vihkon sivujen käyttöä varten. Sitten hän alkoi toteuttaa kopioimaansa reseptiä, vihkonsa sivuilta rivi riviltä lukien. Kun resepti päättyi ohjeeseen, jossa leipuria pyydettiin pysähtymään, jäi hän mitääntekemättömänä tuijottelemaan seinää ja odottamaan seuraavan reseptin saapumista.

Suuri ongelma CBU:n työn sankarin kanssa oli, että reseptien piti olla jopa niin yksityiskohtaisia, että tavallisen kylänmiehen ja -naisen oli vaivalloista kirjoittaa niitä. Heidän piti käyttää tarkkaan sovittua kieltä ja toimintaohjeiden joukkoa, jotka sijaitsivat leipomon ulkopuolella ohjekirjassa nimeltä “CBU baking instruction set manual”. Esimerkiksi kylän erikoisuuden, kermavaahdon, valmistaminen vaati seuraavanlaisen monivaiheisen reseptin:

```
RESEPTI tee_2dl_kermavaahtoa

TOIMINTAOHJEET
aloitus:
  Ota    kulho vasempaan käteen
  Ota    kerma-astia oikeaan käteen
  Kaada  2dl oikeasta kädestä vasempaan
  Ota    vispila oikeaan käteen
  Muista 1000 toistokertaa jäljellä
```

```
vispaus:
    Jos    toistokertoja jäljellä 0 niin jatka kohdasta 'vispauksen_lopet
    Vispaa oikean käden esineella vasemmassa olevan esineen sisältöä
    Vahenna luku 1 toistokertojen määrästä
    Jatka  kohdasta 'vispaus'
vispauksen_lopetus:
  Kaada  vasemmasta kädestä tarjoiluastiaan
  Pese   vasemmassa kädessä oleva astia
  Laita  vasemmassa kädessä ollut astia hyllyyn kohtaan 'kulhon_koti'
  Toimita tarjoiluastia asiakkaalle
  Lopeta
```

Jos reseptiin päätyi vahingossa jotakin muuta kuin CBU-manuaalissa sovittuja yksityiskohtaisia käskyjä, leipuri ei voinut ymmärtää vihkossaan olevaa toimintaohjetta, jolloin häneltä meni pasmat aivan sekaisin ja hänen oli pakko lopettaa reseptin toteuttaminen ja mennä paniikissa vihkonsa takasivuille katsomaan ohjeita, mitä tällaisessa hätätilanteessa tulee tehdä. Harmillisen usein asiakas saikin CBU:n lastausovelle valmiiden leivosten sijasta viestin, jossa sanottiin että "Reseptissä oli tunnistamaton toimintaohje; leivonta päättyi virheeseen reseptin seitsemännenkymmenennenkahdeksannen ohjerivin kohdalla". Jopa tavanomaisten pullien leipominen tuntui vaativan kovin paljon kirjoittamista[77]:

```
RESEPTI pullat

.DATA
ainesten_lkm: 8
aines_laatu_0: vesi
aines_maara_0: 5 dl
aines_laatu_1: hiiva
aines_maara_1: 50 g
aines_laatu_2: suola
aines_maara_2: 1 tl
aines_laatu_3: sokeri
aines_maara_3: 2 dl
```

[77]Mukaillen tätä: http://www.kotikokki.net/reseptit/nayta/227752/Ihana%20pullataikina/

```
aines_laatu_4: kardemumma
aines_maara_4: 1 rkl
aines_laatu_5: voi
aines_maara_5: 150 g
aines_laatu_6: vehnajauho
aines_maara_6: 12 dl
aines_laatu_7: kananmuna
aines_maara_7: 1 kpl


.TOIMINTAOHJE

ainesten_mittaaminen:
  Ota    kulho vasempaan kateen
  Muista dataa luetaan rivilta 'aines_laatu_0'
  Muista 'ainesten_lkm' toistokertaa jaljella
aines_toisto:
    Jos    toistojen maara on 0 niin jatka kohdasta 'aines_toiston_lopetu
    Ota    lukurivin mukainen aines oikeaan kateen
    Lisaa  luku 1 rivinumeroon, jolta dataa luetaan
    Kaada  lukurivin mukainen maara oikeasta vasempaan kateen
    Lisaa  luku 1 rivinumeroon, jolta dataa luetaan
    Vahenna luku 1 toistokertojen maarasta
    Jatka  kohdasta 'aines_toisto'
aines_toiston_lopetus:

vaivaaminen:
  Tyhjenna oikea kasi
  Pese     oikea kasi
  Sijoita oikea kasi taikinaan

  Muista  100 toistokertaa jaljella
vaivaus_toisto:
    Jos     toistojen maara on 0 niin jatka kohdasta 'vaivaus_lopetus'
    Purista oikealla kadella
    Pyorayta oikeaa katta
    Vahenna luku 1 toistokertojen maarasta
    Jatka   kohdasta 'vaivaus_toisto'
vaivaus_lopetus:

kohottaminen:
```

```
  Sijoita vasemmasta kadesta poydalle
  Odota   45 minuuttia

pullien_muotoilu:
  Laita  uuni paalle
  Aseta  tavoitelampotilaksi 200 astetta
  Ota    uunipelti oikeaan kateen
  Lisaa  leivinpaperia oikean kaden esineelle
  Sijoita oikeasta kadesta poydalle

  Muista 20 toistokertaa
muotoilu_toisto:
    Jos toistojen maara on 0 niin jatka kohdasta muotoilu_lopetus
    Ota   kahdeskymmenesosa taikinasta oikeaan kateen

pyoritys_toisto:
      Pyorita oikeaa katta poytaa vasten
      Jos oikean kaden alla ei ole pyorea pallo, jatka kohdasta 'pyoritys_

    Sijoita oikeasta kadesta poydalla olevan esineen paalle

    Vahenna luku 1 toistokertojen maarasta
    Jatka kohdasta 'muotoilu_toisto'
muotoilu_lopetus:

paista_pullat:
  Odota  uunin lampotilaksi 200 astetta
  Ota    poydalla oleva esine oikeaan kateen
  Avaa   uuni
  Sijoita oikeasta kadesta esine uuniin
  Sulje  uuni

  Odota  10 minuuttia

  Avaa   uuni
  Ota    uunista esine oikeaan kateen
  Sulje  uuni

  Ota    pussi vasempaan kateen
  Muista 20 toistokertaa
siirto_toisto:
```

```
    Jos     toistojen maara on 0 niin jatka kohdasta 'siirto_toiston_lopet
    Sijoita oikean kaden esineesta sisaltoyksikko vasemman kaden esineesee
    Vahenna luku 1 toistokertojen maarasta
    Jatka  kohdasta 'siirto_toisto'
siirto_toiston_lopetus:

  Toimita vasemman kaden esine asiakkaalle
  Lopeta
```

Niin kovin yksinkertainen leipuri siis oli, että toistuvien työvaiheiden kohdallakin hänen täytyi pitää yhdellä vihkonsa rivillä kirjaa jäljellä olevista toistoista. Joka kierroksella reseptin tekijän oli erikseen kirjattava reseptiin käsky, että leipuri pyyhkii edellisen lukumäärän pois ja laittaa tilalle yhtä pienemmän luvun. Reseptin tekijän oli annettava myös erikseen ohje toistosilmukasta pois siirtymiseksi sitten kun jäljellä ei ollut enää yhtään toistoa (eli silloin kun leipurin vihkon laskuririvillä oli luku 0). Jokaisen toimintaohjeen lopputulema riippui siitä tilasta, johon edelliset ohjeet olivat leipurin saattaneet. Harmillisia olivat esimerkiksi sellaiset virheet, joissa reseptin kirjoittaja unohti ohjeistaa leipuria sijoittamaan oikean kätensä taikinaan ennen kuin vaivaamiseen liittyvä puristelu ja pyöritys tapahtuivat – pullataikinan ainekset jäivät silloin kokonaan sekoittumatta ja reseptin lopputuloksena pussissa oli jotakin hyvin epämääräistä, eikä ollenkaan niitä pullia, joita epäonninen (vai huolimaton?) reseptin kirjoittaja oli toivonut.

Hankaluuksista huolimatta kylän asukkaat ymmärsivät leipurin potentiaalin. Kukaan ei pystyisi leipomaan tehokkaammin, nopeammin, tai suurempia eriä. Kylään perustettiin leipuritieteen yliopisto sommittelemaan ratkaisuja CBU:n hyödyntämisessä havaittuihin ongelmiin. Yksi ensimmäisistä tavoitteista oli reseptien kirjoittamisen helpottaminen

Kääntäjä, kielet ja kirjastot

Leipuri noudatti vain yksinkertaisia ohjeita muistivihkonsa kanssa, mutta onneksi näillä yksinkertaisilla ohjeilla oli paljon ilmaisuvoimaa, kun niitä yhdisteltiin sopivasti. Kyläläiset päätyivät kirjoittamaan CBU:n manuaalista löytyviä käskyjä hyödyntäen ”reseptin reseptin tekemiseksi”. He antoivat tälle reseptireseptille nimeksi RECTRAN, ”RECipe TRANslator” eli kansankielellä reseptikääntäjä. Kääntäjän toimintaperiaate oli seuraavanlainen: Se itsessään oli muodoltaan ihan tavallinen resepti, jonka yksinkertaisen komentojonon leipuri ymmärsi ja pystyi toteuttamaan. Kuitenkaan sen tehtävänä ei ollut valmistaa leivonnaisia vaan uusi CBU:n ohjekirjan mukainen resepti. Jokaisen leivontatyön aluksi leipurille annettiin toimintaohjeeksi tämä kääntäjä-resepti, jonka jälkeen postiluukkuun voitiin työntää yksinkertaisemmalla, tavallisen kyläläisen helpommin ymmärtämällä kielellä kirjoitettu resepti. Esimerkiksi kermavaahdon valmistusohje saatettiin nyt kirjoittaa seuraavasti:

```
Valmistele kulho, jossa 2 dl kermaa
Toista 1000 kertaa:
  vispaa kulhossa
Toimita kulhon sisalto asiakkaalle
```

Leipurille annettiin ensin toimintaohjeeksi RECTRAN-kääntäjä ja sen perään tämä lyhyt RECTRAN-kielellä kirjoitettu resepti. Kääntäjäresepti hoiti vaivalloisten ja yksityiskohtaisten toimintaohjeiden lisäämisen leipurin vihkon tyhjille sivuille. Kun käännösresepti oli ”leivottu” eli leipuri oli RECTRANin toimintaohjeiden mukaisesti käynyt läpi RECTRAN-kielisen reseptin, lopputuloksena leipurin vihkossa oli varsinainen yksinkertaisista toimin-

taohjeista koostuva kermavaahtoresepti, jonka mukaan se saattoi aloittaa itse leivontatyön.

Kylän asukkaat innostuivat erilaisten reseptikielten kehittelystä niin paljon, että 50 vuoden päästä kielten tutkimus kukoisti ja käytössä oli yli 2000 erilaista kieltä erilaisten leivonnaisten ja tarjoilutilaisuuksien erikoistarpeita varten. Eräällä myöhemmällä kielellä kirjoitettuna pullaresepti saattoi näyttää seuraavalta:

```
import Ainekset.Aines as A;
import Leipomo.Leipoja;
import Leipomo.PullanLeivonta;
import Asiakas;
import Peruskirjastot.List;

List<A> ainekset = {
  A("vesi","5 dl"), A("hiiva","50 g"), A("suola","1 tl"),
  A("sokeri","2 dl"), A("kardemumma","1 rkl"),
  A("voi","150 g"), A("vehnajauho", "12 dl"),
  A("kananmuna", "1 kpl")};

Leipoja leipoja = PullanLeivonta.TeeLeipojaAineksille(ainekset);
leipoja.leivo();
Asiakas.toimita(leipoja.tuotokset());
```

Erilaiset kääntäjäreseptit hoitivat leivontaohjeet kyläläisten ymmärtämästä muodosta leipurin ymmärtämään muotoon. Leipurin ylivertaisuutta työssään osoittaa se, että hän hoiti varsinaisen leipomisen lisäksi myös reseptien kääntämisen - toki hän tarvitsi siihen kyläläisten tekemää kääntäjäreseptiä. Itse leipuri ei kuitenkaan ymmärtänyt reseptikieliä. Hän vain kävi läpi yksinkertaisia ohjeitaan yksi kerrallaan, suoraan sanottuna suoritti niitä. Hän luki kielillä kirjoitettuja lappuja kirjain kirjaimelta, koska kääntäjäreseptissä niin ohjeistettiin. Hän raapusteli vihkoonsa kirjaimia, joita kääntäjäreseptin ohjeet sanelivat. Vähän tai ei ollenkaan tiesi leipuri itse niistä nerokkaista keinoista, joilla kyläläiset saivat hänet muokkaamaan korkean tason kielestä niitä yksinkertaisten

toimintaohjeiden sarjoja, joita hänet myöhemmin laitettaisiin rivi riviltä, sivu sivulta, suorittamaan.

Usein käytetyistä työvaiheista oli alkanut muodostua reseptikirjastoja: leipomon yhteydessä oli kirjahylly, jossa oli sivukaupalla valmiiksi muotoiltuja ja valmiiksi CBU:n toimintaohjeiksi käännettyjä reseptejä, joita leipomon asiakkaat saattoivat käyttää omien reseptiensä osana. Kenenkään ei tarvinnut enää erikseen kirjoittaa ohjetta pullataikinan vaivaamiseksi, koska useimmissa ns. ”korkean tason reseptikielissä” oli toiminto valmiina vaikkapa seuraavanlaisten ilmaisujen kautta:

```
...
taikina.vaivaa(100); // vaivaa sadalla puristuksella
taikina.vaivaa(11); // vaivaa yhdellatoista puristuksella
...
```

Reseptikääntäjän suorittaminen muodosti leipurin ymmärtämät ohjeet ja lisäksi liitti mukaan tarvittavia osia kirjastoista. Se, että reseptiä kääntäessään leipuri kävi läpi vihkonsa sivuja kynä suhisten, välillä etsiskellen valmiita reseptinpätkiä kirjastohyllystä, ei enää juurikaan näkynyt leipomon käyttäjille päin. He saattoivat kirjoittaa reseptejä enemmän arkihavaintoa muistuttavilla kielillä. Jos jonkun reseptissä oli selviä kielioppi- tai muita virheitä, niistä saatiin tieto jo siinä vaiheessa, kun reseptiä käännettiin, eikä vasta siinä vaiheessa kun pullat olivat jo uunissa.

Yksinkertaisen leipurin ohjaaminen helpommin ymmärrettävillä korkean abstraktiotason kielillä oli nyt ratkaistu ongelma. . . vai oliko? Kaikissa kielissä oli omat hyvät ja huonot puolensa. Parhaasta reseptikielestä väitellään Kakkulan kylässä kiivaasti vielä tänäkin päivänä. Valveutuneimmat kylänvanhimmat eivät kiistoihin osallistu, koska he tietävät, ettei täydellistä ja kaikkiin tarpeisiin sopivaa reseptikieltä ole edes mahdollista koskaan tehdä. Monet kylässä ovat tänä päivänä innoissaan mm. funktioleivonnasta, jossa

ns. aidosti funktionaaliset reseptit kuvailevat lopputuotteet raaka-aineidensa funktiona ilman minkäänlaista mahdollisuutta sortua perinteisesti dramaattisia seurauksia aiheuttaneisiin virheisiin, jotka johtuisivat leipomon tai leipurin hetkellisestä tilasta – ”muuttuvan tilan” käsitettä kun ei näissä kielissä ole, ellei sitä erikseen mallinna. Funktioreseptien kääntäjät ovat vielä toistaiseksi melko pitkiä ja hitaita suorittaa, mutta niitä kehitetään jatkuvasti paremmiksi leivontatieteilijöiden voimin.

Yhdenaikaisuus: Sama leipuri, useita samanaikaisia asiakkaita

Monella kyläläisellä oli leivonnallisia tarpeita, joten he kerääntyivät reseptiensä kanssa sankoin joukoin CBU:n lähistölle jonottelemaan omaa vuoroaan reseptin pudottamiseksi CBU:n luukkuun. Sisällä leipuri otti vastaan reseptin, leipoi sen loppuun ja jäi odottamaan seuraavan reseptin saapumista. Jotkut reseptit valmistuivat nopeasti, mutta joissakin saattoi kestää kovin kauankin. Esimerkiksi hyydykkeissä ja sorbeteissa oli kiusallisia odotusvaiheita, jolloin vaikutti siltä ettei muutoin niin tehokas leipuri tehnyt mitään muuta kuin odotteli asioiden jäähtymistä jääkaapissa. Sama juttu, kun pullataikina kohosi tai uunissa oli jotakin paistumassa. Oli myös vaikea tietää etukäteen, milloin CBU saisi edellisen leivontatyön valmiiksi. Kyläläiset olivat tyytymättömiä tähän ”ensiksi tulleet palvellaan ensiksi loppuun” -tyyppiseen toimintaan. Lisäksi joskus kyläläisten resepteissä oli toimintavirheitä, joiden takia leipuri jäi vaikkapa vispaamaan ikuisesti, eikä tilannetta voinut korjata muuten kuin sammuttamalla hetkeksi valot, mistä leipuri ymmärsi lopettaa tekemisensä, pyyhkiä vihkonsa sivut tyhjäksi ja palata odottamaan uutta reseptiä.

Onneksi leipomossa tapahtui pian sukupolvenvaihdos. Leipuriksi

tuli edellisen leipurin jälkeläinen, entistä paljon tehokkaampi ja isommalla muistivihkolla varustettu. Huhut kertoivat, että hänellä oli myös neljä kättä sen sijaan että edellisellä oli vain kaksi. Ne, jotka eivät huhuja uskoneet, saattoivat tarkistaa ohjekirjasta, että kyllä aiempien toimintaohjeiden ”ota oikeaan/vasempaan käteen vispilä” lisäksi nyt oli mahdollista kohdistaa toimintoja myös ”alaoikeaan/alavasempaan” käteen. Aiemmat reseptit toimivat yhä hyvin, mutta CBU:n ulkoiseen rajapintaan oli tullut lisää keinoja leipurin käskyttämiseksi.

Muutenkin leipomo oli käynyt läpi remontin: Kaikki laitteet oli nyt varustettu merkkikelloilla, jotka kilahtivat, kun laitteen toiminto oli valmis: Uuni kilahti, kun pullat olivat kullanruskeita. Jääkaappi kilahti, kun hyydyke oli hyytynyt. Lisäksi aina viiden minuutin välein kilahti seinällä oleva ajastinkello. Aina kellon kilahtaessa leipuri keskeytti meneillään olevan toimenpiteensä ja laittoi vihkoonsa ylös tarkoin, mikä kohta reseptistä olisi ollut tarkoitus tehdä juuri seuraavaksi. Myös kaikissa neljässä kädessään olevat asiat hän laittoi muistiin, jotta hän voisi myöhemmin jatkaa meneillään ollutta reseptiä täysin samasta tilanteesta.

Leipurin muistivihkon takasivulle oli kirjattu toimenpide, joka hänen täytyi tehdä aina kunkin kellon kilahtaessa, heti kun aiempi toimintatilanne oli kirjoitettu vihkoon ylös. Nämä takasivun ohjeet vain ohjasivat eteenpäin jollekin toiselle sivulle, jossa oli tarkempia ohjeita nimenomaan uunikellon, jääkaappikellon tai ajastinkellon kilahduksen käsittelyyn. Tätä toimintatilan tallentamista ja kellon kilahduksen käsittelyyn siirtymistä tituleerattiin uusitun CBU:n manuaalissa ”ensimmäisen tason keskeytyskäsittelytoimenpiteeksi” (engl. First level interrupt handling, FLIH).

Uusi leipuri, kuten ensimmäinenkin, oli saapunut kylään CBU-manuaalin ja tyhjän muistivihkon kanssa. Manuaalissa sanottiin,

että valojen syttyessä leipomoon leipuri avaa aina muistivihkon viimeisen sivun ja ryhtyy suorittamaan kyseisen sivun ensimmäiselle riville kirjoitettua toimintaohjetta. Kaikki keskeytysten käsittelyyn liittyvät sekä muutkin leipomon toimintojen koordinointiin liittyvät ohjeet vihkon loppupuoliskossa olivat kyläläisten itsensä kirjoittamia. He kutsuivat kyseisiä ohjeita leipurinohjausjärjestelmäksi ja olivat tyytyväisiä siihen, mitä kaikkea niillä saatiinkaan aikaan. . .

Leipurinohjausjärjestelmän sivut voitiin päivittää ja liittää leipurin vihkon takaosaan aina kun leipomo oli suljettuna. Kun leipomo taas avattiin, leipuri otti esille manuaalissa sovitun viimeisen sivun ja alkoi seurata uudistettuja toimintaohjeita. Kyläläiset olivat tulleet siihen tulokseen, että ensimmäisinä töinään leipurin tulisi pestä kaikki mahdollisesti likaiset työvälineet, laittaa uunit, tiskikoneet ja muut laitteet päälle, tyhjentää vihkonsa sivuilta tilaa uusia reseptejä varten, organisoida reseptikirjastohylly, varmistaa ettei hylly ole mennyt ulkoisista syistä epäjärjestykseen leipomon ollessa suljettuna, ja tehdä muitakin tarvittavia aloitustoimenpiteitä. Sen jälkeen leipuri saisi mennä lepäämään tekemättä mitään, kunnes asiakaskello kilahtaa sen merkiksi, että postiluukussa on uusi resepti. Asiakaskellon kilahtaminen, kuten kaikkien kellojen kilahtaminen, sai leipurin katsomaan vihkonsa takasivulta, miltä sivulta sen piti jatkaa toimenpiteitä kyseisen kellon kilahtaessa. Siellä olisi toimintaohjeita, joiden avulla leipuri osaisi tutkia saapuneen reseptin: löytyykö häneltä kirjastohyllystä kaikki osareseptit, joita tarvitaan, onko resepti valmiiksi ymmärrettävässä leipurikielisessä muodossa, vai täytyisikö ensin suorittaa jonkin kääntäjäreseptin toimenpiteet.

Vihkossa leipuri piti kirjaa aivan kaikesta, koska hänellä ei liiemmin ollut lähimuistia. Tähän mennessä leipurinohjausjärjestelmään oli kertynyt jo aikamoinen kasa kyläläisten kirjoittamia toi-

mintaohjeita, joiden kautta leipurin toivottiin hallitsevan varsinaisten leivontatyöreseptien käsittelyä. Järjestelmäsivut oli sijoitettu aivan omaan osioonsa leipurin muistivihkossa: vihkon puolivälistä alkoi nimittäin erikoinen osuus, joita tässä vaiheessa CBU:n manuaalikin sanoi järjestelmämuistisivuiksi. Nämä sivut leipurin muistivihkosta olivat aluetta, johon mikään asiakkaan resepti ei saanut pyytää leipuria kirjoittamaan suoraan. Loppupuoli muistivihkosta oli yksinomaan keittiön ja leipurin toimintojen organisointia varten, eikä niille sivuille missään vaiheessa sijoitettu leipomon käyttäjien reseptejä eikä muitakaan käyttäjien tietoja. Jos joku asiakkaan resepti edellytti leipuria toteuttamaan järjestelmäsivujen toimintaohjeita, heidän reseptissään täytyi olla aivan erityinen toimintaohje, joka sai leipurin keskeyttämään reseptin normaalin suorituksen ja selaamaan seuraavat toimintaohjeet järjestelmäsivuilta vihkon takaosasta.

Lienee käynyt selväksi, että CBU:n sisätiloissa oli käytössä kaikenlaisia leivontaan liittyviä resursseja, kuten uuni, kulho, vispilä sekä mausteita ja muita raaka-aineita. Yksi mainitsemisen arvoinen leivontaresurssi oli itse leipuri, jonka suoritusaika oli olennainen leipomotuotteiden valmistuksessa.

Asiakkaat tulivat leipomon ovelle reseptinsä kanssa, ja lopputuotteet saatiin ulos, kun kukin asiakkaan reseptin mukainen leivontaprosessi tuli valmiiksi. Jotkut resursseista ja leipomon palveluista olivat luonteeltaan hitaampia kuin toiset. Kylän asukkaat huomasivat, että CBU:n kokonaistuottavuus (engl. throughput) eli aikayksikössä valmiiksi saatujen leivosten määrä saatiin kasvamaan, mikäli esimerkiksi pullataikinan kohoamisen aikana voitiin allokoida leipuri tekemään muita tuottavia tehtäviä, kuten kermavaahdon valmistusta tai toisen pullataikinan vaivaamista.

Uuden sukupolven leipurilla oli kaikki ominaisuudet prosessoin-

tiajan jakamiseen useiden asiakkaiden kesken: Kun aikaa vaativa operaatio, esimerkiksi taikinan nostatus tai pullien paistaminen uunissa alkoi, oli leipuri vapaa ottamaan suoritettavakseen jonkin toisen asiakkaan reseptin, vaikkapa kermavaahdon valmistamisen. Yhden pullataikinan kohotessa oli mahdollista vispata hyvinkin paljon kermavaahtoa…

Suoritusten eriyttäminen nopeutti myös yksittäisten tuotteiden valmistamista, esimerkiksi laskiaispullien: taikinan noustessa ja pullien paistuessa oli mahdollista vispata kermavaahto ja sulattaa marjat pakkasesta hillon tekemistä varten. Leivontatieteen yhteisö puhui yksittäisen reseptin suorittamisen jakamisesta yhdenaikaisiin suoritussäikeisiin (engl. thread of execution). Reseptien piti huomioida tällaiset tarpeet tietyin kielien ja apukirjaston tukemin keinoin, esimerkiksi:

```
...

Osatehtava pullauttaja = PullanLeivonta.TeeLeivontaAineksille(pulla_ainekset);
Osatehtava kermavaahto = VaahdonLeivonta.TeeLeivontaAineksille(kv_ainekset);
Osatehtava hillottaja = HillonLeivonta.TeeLeivontaAineksille(hillo_ainekset);

pullauttaja.aloitaYhdenaikainenLeipominen();
kermavaahto.aloitaYhdenaikainenLeipominen();
hillottaja.aloitaYhdenaikainenLeipominen();

// Tassa kohtaa kolme saman reseptin eri osiota ovat meneillaan
// samanaikaisesti. Paatehtavaa ei voida jatkaa ennen kuin osatehtavat
// ovat kaikki valmiita:

pullanLeipoja.odotaEttaOnValmista();       // Etenee tasta kun pullat tehty.
kermavaahdonLeipoja.odotaEttaOnValmista(); // Etenee tasta kun vaahto tehty.
hillonLeipoja.odotaEttaOnValmista();       // Etenee tasta kun hillo on tehty.

Leivonta laskiaispullanKasaaja = LaskPullaLeivonta.TeeLeivonta();
laskiaispullanKasaaja.yhdistaToisiinsa(pullanLeipoja.tuotokset(),
  kermavaahdonLeipoja.tuotokset(),hillonLeipoja.tuotokset());

Asiakas.toimita(laskiaispullanKasaaja.tuotokset());
```

Edelleen leipuri seurasi yksinkertaisia CBU:n ohjekirjassa julkaistun rajapinnan mukaisia toimintaohjeita yksi pieni toimenpide kerrallaan, mutta muistivihkon takasivuilta löytyvän uuden ja hienon leipurinohjausjärjestelmän resepti mahdollisti yhdenaikaisen suorittamisen: Keskeytyksen tullessa (eli kun jokin leipomon kelloista kilahti), lopetti leipuri nykyisen reseptin suorituksen ja käsitteli kellonkilauksen aiheuttaneen tilanteen. Se saattoi mm. tarvittaessa ottaa uuden asiakkaan reseptin saman tien työjonoonsa, kun asiakaskello kilahti. Leipuri seurasi ohjausjärjestelmän käskyjä, jotka ohjasivat sitä pitämään muistivihkossaan kirjaa kaikista meneillään olevista leivontaprosesseista: Asiakkaille annettiin reseptin vastaanoton yhteydessä vuoronumero, joka yksilöi kunkin leivontaprosessin. Leipuri piti vihkossaan kirjaa meneillään olevista prosesseista. Tai eihän hän oikeastaan varsinaisesti ”tiennyt”, mitä on tekemässä – kunhan vain seurasi orjallisesti ohjausjärjestelmän sivuille kirjoitettua ”järjestelmäreseptiä”. Keskeytettynä olevia leivontaprosesseja voitiin jatkaa myöhemmin, koska kaikki jatkamiseen tarvittavat tiedot oli laitettu ylös muistivihkon sivuille. Leipurinohjausjärjestelmän toimintaohjeet määrittelivät, mikä leivontaprosessi tai -säie otettiin seuraavaksi käsittelyyn, ja leipurin toimintatila voitiin palauttaa kyseisen prosessin aiemmin keskeytyneen tilanteen mukaiseksi.

Ensi alkuun kyläläiset olivat tyytyväisiä menettelyyn, jossa kaikki reseptiluukkuun annetut työt otettiin suoritukseen ja niitä kaikkia leivottiin vuorollaan joko kunnes ajastinkello kilahti tai kunnes niissä tuli odottelua vaativa työvaihe. Kellon kuin kellon kilahtaessa leipuri keskeytti meneillään olevan leivontaprosessin, tallensi sen tilanteen, ja siirtyi seuraamaan leipurinohjausjärjestelmän toimintaohjeita, aivan niin kuin CBU:n ohjekirja lupasi keskeytyskäsittelyä kuvailevassa luvussaan. Kyläläiset olivat sopineet, että yhden prosessin keskeytyessä suoritukseen otettiin järjestyksessä seuraa-

va ja viimeisen jälkeen siirryttiin taas ensimmäiseen. Nimeksi tälle vuoronnusmenettelylle he olivat antaneet kiertojonon (engl. round robin). Leipurin muistivihkossa oli vuoronnettavien prosessien sekä niitä pyytäneiden asiakkaiden identiteettitiedot sekä järjestys jonossa. Vihkossa oli tallessa myös tiedot siitä, mille vuoronumerolle mikäkin vispilä, uuni tai muu resurssi milloinkin oli varattuna. Leipuri ei edelleen tiennyt tuon taivaallista kyläläisten aivoituksista - se vain seurasi sille annettuja yksinkertaisia, muistivihkon sivujen riveille kirjoitettuja ohjeitaan rivi riviltä.

Prioriteetit, Nälkiintyminen, Reaaliaikavaatimukset

Kakkulan kylän pormestarilla oli tulossa suuri edustusjuhla, johon tarvittiin kymmenen täytekakkua mahdollisimman nopeasti. Pormestari laittoi reseptinsä CBU:n käsiteltäväksi, mutta oli varsin tyytymätön siihen, että joutui tavallisten asukkaiden kanssa samaan kiertojonoon, jossa häntä palveltiin tasapuolisesti kaikkien muiden, kymmenien, kyläläisten kanssa. Pormestarin kakkujen valmistus viivästyi, ja juhlat jouduttiin juhlimaan ilman kakkutarjoilua. Ei ollut kenellekään yllätys, että jo seuraavalla viikolla leipurin muistivihkon loppuosassa oleva leipurinohjausjärjestelmä päivitettiin sellaiseen, joka pystyi hallitsemaan prioriteetteja. Jokaiselle leipomon asiakkaan antamalle leivontatyölle määrättiin työn aloituksen yhteydessä prioriteetti. Kylään perustettiin leipurijärjestelmän ylläpitäjän virka, ja viran haltijalle annettiin ainoana kylässä lupa määritellä prioriteetteja ja käyttölupia uuneihin ynnä muihin leipomon resursseihin. Jatkossa pormestarin työt menisivät tarvittaessa edelle kaikista muista. Muutenkin leipomon asiakkaat pystyivät neuvottelemaan keskenään prioriteeteista, jotka ylläpitäjä sitten toimitti leipurin muistivihkoon.

Leipurin vihkon takasivuilla sijaitsevat ohjeet määrittivät nyt, että korkeamman prioriteetin resepteillä oli etuajo-oikeus alemman prioriteetin resepteihin nähden. Jokaista prioriteettitasoa kohden oli oma kiertojononsa leipurin muistivihkossa. Keskeytyskellon soidessa leipuri otti leivottavakseen korkeamman prioriteetin reseptejä useammin kuin matalampien prioriteettien. Kaikki leipurin työt valmistuivat edelleen, mutta enää kiireelliset edustuskakut eivät jääneet valmistumatta ajallaan. Kylän asukkaat olivat jälleen tyytyväisiä. Kuitenkin tavallista asukasta jäi hiljaa harmittamaan se, että ajoittain, sesonkikausina, pormestarin lähipiirin tilaukset veivät kaiken ajan, eivätkä normaalit pullatilaukset meinanneet valmistua koskaan. Kansankielellä he puhuivat omien reseptiensä ”nälkiintymisestä”, mikä ei ollut kaukana heidän omasta tilanteestaan, jos vaikka perhe odotti voileipäkakkua lounaakseen, mutta korkeamman prioriteetin tilaukset estivät päivien ajan leipää tulemasta pöytään asti.

Vuoronnuksessa havaittiin prioriteettien ja niistä silloin tällöin johtuvan nälkiintymisen lisäksi vielä sellainenkin seikka, että tietyt toimenpiteet eivät yksinkertaisesti voineet odottaa yhtään: Esimerkiksi, kun pullat olivat paistuneet uunissa valmiiksi, täytyi tilanne käsitellä mahdollisimman pian, koska muutoin pullat olisivat palaneet. Leivontatieteen piirissä puhuttiin resepteistä, joilla oli reaaliaikavaatimuksia eli tiukkoja takarajoja sille, miten nopeasti mihinkin tilanteeseen täytyi reagoida ja kuinka nopeasti ne täytyi saada hoitumaan loppuun saakka.

Synkronointi, Lukkiintuminen

Prioriteeteista johtuva nälkiintyminen oli ymmärrettävä, joskin harmillinen tilanne. Suurempia ongelmia kylän leivostuotantoon tuli kuitenkin tilanteista, joille kyläläiset antoivat nimen ”kuolettava lukkiintuminen”: Useissa resepteissä tarvittiin yhtäaikaa useampia

resursseja, esimerkiksi uunia ja vispilää. Oli jo kauan sitten havaittu, että kukin resurssi on syytä varata eli lukita yhden leivontaprosessin käyttöön kerrallaan. Muutoinhan uuniin voisi mennä yhtäaikaa eri asiakkaiden pullia ja pitsoja ja ties mitä. Pullat menisivät osittain sekaisin, jolloin olisi mahdollista että kumpikaan kahdesta pulla-asiakkaasta ei saisi kauniita kullankeltaisia pullia uunista ulos – saati sitten se asiakas, joka oli tilannut pitsaa. Leipurinohjausjärjestelmään olikin rakenneltu lukitusmekanismi: resepteissä tuli pyytää esimerkiksi uunin lukitsemista omaan käyttöön ennen kuin uuniin sai laittaa tavaraa. Vastaavasti uuni piti vapauttaa muiden käyttöön sen jälkeen, kun oman reseptin käyttötarve uunille loppui. Puhuttiin leivontaprosessien synkronoinnista, joka oli tarpeen silloin, kun eri leivontaprosessien välillä oli mahdollista tulla kilpajuoksutilanne (eli reseptit ”kiiruhtivat” lähes yhtäaikaa käyttämään resurssia, mutta sisäisesti peräkkäisestä suorituksesta johtuen vain yksi ehti tietysti aina ensimmäisenä). Usean yhdenaikaisen asiakkaan leipomossa reseptit alkoivat näyttää seuraavalta:

```
...
Lukitse(uuni);
uuni.laitaPullatSisaan(omat_pullat);
uuni.odotaPullienValmistuminen();
Asiakas.toimita(uuni.pullat());
Vapauta(uuni);
...
```

Lukkoihin perustuva synkronointi auttoi, mutta hankaluuksia alkoi ilmaantua, kun kylän asukkaiden kirjoittamat reseptit halusivat lukita kerralla useampia keittiön resursseista. Silloin saattoi olla yhtä aikaa suorituksessa kaksi erilaista reseptiä:

```
RESEPTI Jussin pullat:
```

```
  ...
  Lukitse(uuni);
  Lukitse(vispila);
  uuni.laitaPullatSisaan(omat_pullat);
  vispila.vispaa(vaahto);
  uuni.odotaPullienValmistuminen();
  uuni.pullat().laitaVaahtoaSisaan(vaahto);
  Asiakas.toimita(uuni.pullat());
  Vapauta(vispila);
  Vapauta(uuni);
  ...

RESEPTI Paulan pullat:
  ...
  Lukitse(vispila);
  Lukitse(uuni);
  uuni.laitaPullatSisaan(omat_pullat);
  vispila.vispaa(vaahto);
  uuni.odotaPullienValmistuminen();
  uuni.pullat().laitaVaahtoaSisaan(vaahto);
  Asiakas.toimita(uuni.pullat());
  Vapauta(uuni);
  Vapauta(vispila);
  ...
```

Suurimman osan aikaa kaikki saattoi näyttää toimivan oikein hyvin, mutta jos kävikin sattumalta vaikka niin, että leipuri ehti suorittaa Jussin reseptiä siihen asti, että uuni oli lukossa ja juuri siinä kohtaa kilahtikin ajastuskello... Leipuri otti sitten normaalien sääntöjensä mukaisesti suoritukseen Paulan leivontaprosessin, joka lukitsi vispilän. Seuraavaksi Paulan prosessi yritti lukita uunin,

mutta lukitus olikin jo Jussilla... Paulan prosessi joutui odottelemaan, että Jussin prosessi vapauttaisi uunin. Mutta Jussin prosessin seuraava tehtävä oli lukita vispilä, mikä puolestaan olisi edellyttänyt, että Paula ensin vapauttaisi vispilän. Molemmat leivontaprosessit odottivat toistensa operaatioita, ja kumpikaan ei voinut edetä: Paulan prosessi ei voinut edetä ilman uunia eikä Jussin prosessi ilman vispilää. Samaan aikaan kukaan muukaan ei voinut käyttää uunia eikä vispilää. Yhtäkään pullaa ei saatu uunista ulos, kun ei sinne saatu niitä sisäänkään. Tälle viheliäiselle, enemmän tai vähemmän satunnaisesti ilmenevälle ongelmatilanteelle kyläläiset antoivat nimen lukkiutuminen (engl. deadlock). Leivontatieteen piirissä alkoi kuumeinen tutkimus keinoista, joilla kyläläisten resepteillään itse aiheuttamat lukkiutumistilanteet voitaisiin havaita tai ennaltaehkäistä ja kuinka niistä voitaisiin palautua takaisin normaaliin leivontatilanteeseen. Toistaiseksi on lisäksi nähty tarpeelliseksi valistaa reseptien tekijöitä näistä tiettyyn resurssiin kohdistuvista ”kilpajuoksutilanteista” (engl. race condition) ja tekemään reseptinsä siten, että ongelmia ei pääsisi syntymään.

Leipurinhallintajärjestelmäkin alkoi olla jo niin pitkä ja monimutkainen, että siihen alkoi päivitysten yhteydessä tulla virheitä ja joskus jopa itse hallintajärjestelmän sisäiset lukot saivat leipurin tilanteeseen, jossa se jäi ikuisesti odottelemaan, eikä auttanut taas muuta kuin sammuttaa leipomosta valot hetkeksi, että leipuri tajusi nollata tilanteen. Ongelmista raportoitiin leivontajärjestelmän tekijöille ja toivottiin, että järjestelmään saataisiin pian päivitys, jossa jumittumisen aiheuttava virhe olisi korjattu.

Tiedostojärjestelmä, käyttäjänhallinta, etäkäyttö

Leipomossa alettiin tehdä hienompia ja hienompia tuotoksia: Sokerilla voitiin koristella kakkuihin tekstejä ja jopa valokuvia. Marsipaanikakkuihin saatiin teetettyä jopa haluttu kolmiulotteinen muoto. Piparkakkutalot koostettiin elementeistä hierarkkisten mallien perusteella ja ennen niiden valmistusta saatettiin leipuri laittaa tekemään piparkakkutalon rakenteille lujuusanalyysi, jotta raaka-aineita ei menisi hukkaan romahdusherkän talon konkreettiseen valmistukseen. Väki teki toinen toistaan hienompia reseptejä ja leipomon kakuista tuli alati värikkäämpiä ja persoonallisempia. Hienot reseptit ja niihin liittyvät kuvat ja 3D-muodot haluttiin säilyttää leipomossa myöhempää uudelleenkäyttöä varten. Kyläläiset hankkivat keittiöön lisää hyllytilaa luomuksiensa resepteille ja niiden vaatimille lisätiedoille, joita saatettiin käyttää uudelleen ja jakaa naapureiden kanssa. He päivittivät leipurin muistivihkon takaosaan ohjeet, joita noudattamalla leipuri osasi tunnistaa ja löytää tietoja tietynlaisen, selkeän, hyllyosoitteen perusteella. Tätä sanottiin kakkutiedostojärjestelmäksi.

Kakkutiedostojärjestelmän avulla oli helppo tallentaa ja myöhemmin löytää erilaisia leivoksia koskevia tietoja kakkutiedosto-osoitteen perusteella, esimerkiksi ”leipomo/käyttäjät/liisa/marsipaanikakut/-polttarikakut/tuhma01.3d”. Kaikki tietojen siirto kakkutiedostojärjestelmään kulki edelleen CBU:n seinässä olevan laatikon kautta. Leipurinohjausjärjestelmää oli kehitetty siten, että kun kyläläiset laittoivat laatikkoon omia ohjauskomentojaan, he saattoivat luottaa siihen, että kukaan muu ei vahingossa pääse käsiksi heidän omiin resepteihinsä tai niihin liittyviin lisätietoihin. Leipurille antamiinsa komentoihin he liittivät mukaan salasanan, jota kukaan muu ei tiennyt.

Korostaa sopii, että myös tästä kaikesta leipuri oli autuaan tietämätön. Hän vain muistivihkonsa kanssa seurasi rivi riviltä niitä ohjeita, joita hänellä vihkossaan oli. Salasanojen käsittely, kakkutiedostojen tallentaminen hyllykköön... itse leivonta... kaikki se oli kyläläisten suunnittelemaa, leipuritieteen tiedekunta etunenässä. Leipurille päätyessään se kaikki oli muistivihkon takaosaan reseptiksi kirjoitettu. Hänen työnsä oli helppoa: leipomon auetessa hän käänsi muistivihkonsa auki leipurinohjausjärjestelmän viimeiseltä sivulta ja alkoi seurata ohjeita.

Kahden leipurin leipomo; sama ohjevihko, yksi uuni

Vuodet kuluivat, ja leipomossa tapahtui muutamakin sukupolvenvaihdos. Aiemmat leipurit jäivät eläkkeelle ja heidän jälkeläisensä tarttuivat jauhoihin. Jokainen uudistus toi uusia mahdollisuuksia Kakkulan kylän väelle käyttää hyväkseen CBU-leipomoa kaikenlaiseen leipomiseen. Eräs olennainen uudistus oli se, kun leipomoon tulikin töihin uuden sukupolven kaksoset: Leipureita ei ollut enää vain yksi, vaan olikin kaksi, jotka pystyivät kerta kaikkiaan vaivaamaan kahta eri pullataikinaa samaan aikaan, rinnakkaisesti. Omia haasteita CBU:n talon sisällä aiheutti se, että molemmilla identtisillä leipureilla oli kuitenkin vain yksi yhteinen muistivihko. Heillä täytyi siis olla sovittuna aivan tietyt periaatteet siihen, kuinka asiakkaiden ja leivontaprosessien tiedot pidetään järjestyksessä, kun heillä molemmilla oli pääsy samaan vihkoon ja heidän molempien piti tehdä sinne jatkuvasti muutoksia. Kaksoset olivat yhtä neuvottomia kuin vanhempansakin, joten jälleen jäi kylän leivontatieteen osaajien asiaksi rakennella CBU:n muistikirjan takasivuille sellaiset toimintaohjeet, että kaksi leipuria pystyivät yhdessä tuumin prosessoimaan asiakkaiden leivontatarpeita ilman ristiriitoja. Keskinäisiä sopimuksia heillä oli vain muutama, erityisesti

aina tiettyjen kellojen kilahtaessa molemmat lopettivat toimintansa, eivätkä jatkaneet samaan aikaan ennen kuin yhdessä tuumin luetut toimintaohjeet taas sallivat sen. Jotta yhteistä muistivihkoa ei tarvitsisi koko ajan selata sivulta sivulle, oli molemmilla leipurikaksosilla kuitenkin käden ulottuvilla oma, pienempi, muistivihko, joihin he salamannopeasti pystyivät kopioimaan joitakin peräkkäisiä rivejä tai sivuja yhteisestä isommasta vihkostaan. Nimeksi pienille apuvihkoille oli annettu välimuistivihko.

Muutkin kuin Kakkulan kylän asukkaat havaitsivat aikojen saatossa CBU:n edut. Koska tuossa taianomaisessa rinnakkaistodellisuudessa leipureiden geneettinen suunnittelu oli arkipäivää ja suunniteltujen leipureiden kloonaaminen onnistui sarjatuotantona leipuritehtaissa, alkoi monenlaisia leipomoita syntyä lisää leipomoteollisuuden tuotteena – jotkut leipomot oli suunniteltu palvelemaan massiivisia tarjoilutilaisuuksia, jotkut taas yksittäisen henkilön tai kotitalouden pieniä päivittäisiä leivontatarpeita. Massiivisiin tilaisuuksiin tarkoitetut leipomot olivat hienoimpia ja niissä oli muutaman leipurin sijasta kymmeniä tai satoja leipureita, jotka pystyivät suorittamaan toimenpiteitä rinnakkain. Tällaisen massiivisesti rinnakkaisen perheleipomon tehokas hyödyntäminen vaati tietynlaisia reseptejä – erityisen tehokasta oli, jos reseptin osavaiheista suurin osa oli riippumattomia muista osavaiheista, jolloin rinnakkain operoivien leipurien ei tarvinnut pysähtyä odottelemaan toisiltaan tietoja tai lopputuloksia. Leipuritieteen oppikirjoihin oli jäänyt elämään perusesimerkit kaiken aloittaneesta Kakkulan kylästä; rinnakkaisleivonnasta ensimmäisenä esimerkkinä tuli edelleen vastaan tuhannen kermavaahtolitran valmistaminen, josta sata leipuria selviytyi sata kertaa nopeammin kuin yksi – jokainen leipuri kun pystyi vispaamaan toisistaan riippumatta yhtäaikaa kymmenen litraa vaahtoa kukin. Vasta työlään vaiheen jälkeen sata kertaa kymmenen litraa voitiin kaataa lopulta samaan tarjoi-

luastiaan. Kaataminen oli tehtävä peräkkäin yksi kerrallaan, koska lopputulema haluttiin samaan astiaan – kaataminen oli kuitenkin niin paljon nopeampaa kuin vispaaminen, että voitiin sanoa operaation nopeutuvan ainakin käytännössä niin monikertaisesti kuin rinnakkaisia leipureita vain oli käytössä.

Ympäröivä maailma muuttui viestintäyhteyksineen, ja lopulta Kakkulan kylän leipomon käyttäjät eivät välttämättä sijainneet ollenkaan Kakkulan kylässä, vaan ympäri maailman he saattoivat tietoverkon yli käyttää Kakkulan leipomon palveluita. Leipurit olivat autuaan tietämättömiä siitä, tulivatko heidän reseptinsä naapuritalosta vai toiselta puolen planeettaa. He yhä vain toteuttivat yksinkertaisia toimintaohjeitaan, jotka joku muu oli kirjoittanut ja kääntänyt heidän ymmärtämälleen yksinkertaiselle kielelle.

Leipurin ohjekirjan mukaiset toiminnot mahdollistivat kakkujen tekemisen lisäksi paljon muutakin. Koska CBU:n julkaisemat säännöt reseptien kirjoittamiseen muodostivat Turing-täydellisen ohjelmointikielen, päätyivät Kakkulan kylän asukkaat käyttämään leipomoaan myös sääennusteiden tekemiseen, satelliittien kiertoratojen laskemiseen, siltojen rakenteen suunnitteluun, pankkitilien ja verkkomaksujen hallitsemiseen sekä tilapäivitysten ja kissavideoiden jakamiseen kavereiden kesken.

Miten tämä liittyy tietokoneisiin ja käyttöjärjestelmiin?

Edellä kerrottiin ”laulun ja leikin keinoin” monet tämän kurssin perusasioista: Leipuri ja leipomon välineet olivat esimerkkejä joistakin rajallisista resursseista, joita haluttiin käyttää hyväksi ja jakaa eri tahojen kesken. Kyläläisten positiiviset ja negatiiviset kokemukset olivat tavanomaisia resurssien jakamiseen mahdollisesti liittyviä tilanteita. Heidän keksimänsä ratkaisut puolestaan oli-

vat normaaleja toimintamalleja resurssien jakamisessa. Vastaavia ilmiöitä nähdään monessa paikassa, kuten kaupan kassajonoissa, ravintoloiden pöytiintarjoilussa tai pilvenpiirtäjien hisseissä.

Varsin suuri osa kurssilla myöhemmin vastaan tulevasta terminologiasta on löydettävissä tarinasta, poistamalla sanojen leivontaan liittyvät alkuosat. Reseptit olivat tietenkin analogia tietokoneohjelmasta ja leipominen ohjelman suorittamisesta tietokonelaitteistossa. Ero leipurin ymmärtämien yksittäisten toimintaohjeiden ja käännettävien ”korkean tason reseptikielien” välillä yritti vastata eroa tietokoneen prosessorin ymmärtämän niin sanotun konekielen (tai enemmänkin assemblerin) sekä sovelluskehityksessä käytettyjen korkean tason ohjelmointikielten välillä. Erilaiset ohjelmointikielet ja niitä varten tehdyt kääntäjät sekä tulkit tulevat tietotekniikassa jatkuvasti vastaan eri muodoissaan, samoin kuin erilaisia tehtäviä varten tehdyt apuohjelmakirjastot.

Leipurin yksinkertaisuus, ”putkiaivoisuus” ja lähimuistin totaalinen puuttuminen olisivat ihmiselle varsin rampauttavia, eikä sellaista onneksi liiemmin esiinny ihmispopulaatiossa. Satuhahmo vastaa puutteineen kuitenkin läheisesti tietokonetta, joka toteuttaa yksinkertaisten käskyjen sarjaa. Muistivihko sivuineen ja riveineen on analogia tietokoneen muistille, jota tietokone käyttää kaikkeen tekemiseensä. Myös tietokoneen muisti on usein jaettu sivuihin, joilla voi olla erilaiset roolit - osa sivuista sisältää ohjelmakoodia ja osa koodin käsittelemää dataa. Osa sivuista on varattu käyttöjärjestelmän käyttöön. Leipurinohjausjärjestelmä oli tietenkin analogia käyttöjärjestelmästä, jonka kautta tietokonelaitteistoa ohjataan keskitetysti ja kontrolloidusti.

Leipurin vihkon yhdelle sivulle mahtuu vain tietty määrä rivejä. Kullekin riville puolestaan mahtuu vaikkapa vain yksi toimintaohje tai vaihtoehtoisesti yksi reseptin tarvitsema tiedon palanen, kuten

yksittäisen raaka-aineen määrä. Samalla tavoin tietokoneen muistin sivulle mahtuu tietty määrä ohjelman tarvitsemia konekielisiä käskyjä tai ohjelman käsittelemää tietoa. Isommat ohjelmat vaativat enemmän sivuja kuin pienet. Samoin kuin leipurin vihkossa on rajallinen määrä sivuja, on myös tietokoneen muisti rajallinen. Leipuri tarvitseekin avukseen kirjahyllyjä, joista reseptejä ja tietoja voidaan tarvittaessa käydä kopioimassa muistivihkoon; tietokoneessa käytetään kovalevyjä ja muita ns. massamuisteja, joista voidaan tuoda tarvittavia tietoja tai tiedon osasia väliaikaisesti muistisivuille, joita prosessori käsittelee. Samoin kuin leipuri tyhjentää vihkonsa valojen sammuessa, tietokoneenkin muisti tyhjenee virran katketessa laitteesta. (Tai vähintään muisti jää satunnaiseen ja arvaamattomaan tilaan, joten se on joka tapauksessa syytä tyhjentää ennen uutta käyttöä). Massamuisteihin tiedot kuitenkin jäävät talteen, aivan kuten leipurin kirjahyllyihin. Tietojen löytämiseksi niille täytyy antaa ihmisen ymmärtämiä osoitteita, ja yksityisyys vaatii, että jokaiseen tiedostoon liittyy tiedot käyttäjästä, joka sen omistaa sekä siitä, ketkä muut mahdollisesti pääsevät käsiksi tietoihin.

Analogialla on rajansa, mistä syystä koko kurssia ei voidakaan viedä läpi laulun ja leikin keinoin tai leipureista puhuen. Tietokone on ensinnäkin tietyssä mielessä vielä Kakkulan kylän leipuria paljon ”tyhmempi” – se kopioi paikasta toiseen ykkösiä ja nollia eli bittejä, eikä sillä voisi olla käsitystä esimerkiksi uunista, vispilästä tai vispilän pyörittämisestä. Kaikki tietokoneen operaatiot perustuvat bittien siirtämiseen paikasta toiseen. Tyypillisesti kohdepaikan aiempi sisältö pyyhkiytyy yli ja toisaalta lähdepaikan sisältö pysyy muuttumattomana, joten täsmällisempää olisi puhua “kopioinnista, joka korvaa kohteen aiemman sisällön”. Tietokoneella ei siis ole käsitystä esimerkiksi hiirestä, näppäimistöstä, kuvaruudusta, printteristä, nettiyhteydestä, kovalevystä tai käyttäjien oikeuksis-

ta. Se seuraa konekielisiä toimintaohjeitaan ja siirtää bittejä numeroidusta osoitteesta toiseen. Kaikki korkeamman abstraktiotason käsitteet ovat ohjelmointikielellä kirjoitettua, ”kuviteltua/ajateltua” tai virtuaalista mallinnusta, joka on dokumentoitu kunkin laitteen tai järjestelmän käyttöohjeisiin / rajapintaan.

Toisaalta jotkut asiat ovat tietokoneessa helpommin tehtävissä kuin mihin tämän esimerkin leipuri kykeni. Varsinkin muistin käyttö on erilaista ns. virtuaalimuistin ansiosta. Mikäli tätä pehmojohdantoa muistelee jatkossa, on hyvä huomata, että leipurin muistivihko vastaa lähimmin tietokoneen ns. fyysistä muistia, kun taas käyttäjien ohjelmat operoivat ns. virtuaalimuistin kanssa. Yksinkertaisuuden nimissä kuvitteellisen leipurin annettiin nyt käyttää suoraan fyysistä muistivihkoaan kaikkeen toimintaan. Tämän, kuten muidenkin asioiden suhteen, on syytä kahlata tarkoin läpi myös realistisempi, oikeaan laitteistoon perustuva johdantoluku.

Virtuaalimuistin periaatteiden lisäksi kirjoittaja ei löytänyt keinoja, joilla tähän analogiaan olisi ympätty käyttäjän toimenpiteiden odottelua (olisikohan leipuri voinut pyydettäessä jäädä leipomon ovelle odottamaan käyttäjän valintaa tämän päivän kakuntäytteistä?) eikä prosessien välistä kommunikaatiota (mitähän viestejä pullanleivonta voisi lähettää kermavaahdon vispaukselle tai toisin päin. . .). Jotakin lisäyksiä yhdenaikaisuudesta olisi tähän mahtunut – vaikkapa kermavaahtoa tuottava leivontaprosessi, jota täytekakkuja valmistava prosessi kuluttaa. Ehkä suorituspinolle ja aliohjelmakutsullekin olisi jokin analogia löytynyt. Mutta eiköhän tämä tarina jo tällaisenaankin ollut riittävän pitkä ja unettava. Yksityiskohdat tulkoot siis varsinaisissa luvuissa pehmojohdannon jälkeen.

Tässä luvussa nähdyt ”korkean tason reseptikielet” ja leipurin ymmärtämien toimintaohjeiden muoto ovat täysin keksittyjä, vaik-

kakin niissä on varmasti tunnistettavia piirteitä nykyään käytössä olevista ohjelmointikielistä. Esitietona olleen Ohjelmointi 1 -kurssin (tai vastaavan ohjelmointitaidon) perusteella täytyy pystyä seuraamaan kaikkien näiden keksittyjen kielten toimintalogiikkaa. Mikäli se tuntuu vaikealta, täytyy aiemman ohjelmointikurssin asioita kerrata pikapikaa! Oikeassa elämässä kielet ovat tietysti tarkkaan määriteltyjä ja pohtien suunniteltuja. Niiden syntaksia on noudatettava tarkkaan tai kääntäjä ei osaa kääntää ohjelmaa konekielelle. Lisäksi aina on ymmärrettävä myös semantiikka eli se, mitä milläkin ohjelman rivillä oikeastaan tapahtuu. Keksittyjä ”pseudokieliä” käytetään jatkossakin, mutta tällä kurssilla tullaan myös näkemään ja soveltamaan oikeita ohjelmointikieliä, erityisesti C-kieltä, AMD64-prosessorin konekieltä niin sanotun AT&T -murteen mukaisella assemblerilla kirjoitettuna sekä Bourne Again Shell (bash) -skriptikieltä. Kielten opiskelua tukemaan on demoissa käytännön harjoitteita, joiden tekeminen on syytä aloittaa pian.

Jos koet ymmärtäneesi tarinan leipurin toimintaa, tulet varmasti ymmärtämään myös tietokoneen toimintaa. Laulu ja leikki loppuvat nyt tähän. Seuraavassa luvussa saavutaan todellisuuteen.

.2 Koodiliite

Tähän on ladottu vuonna 2016 luennoilla esitellyt ohjelmat sellaisenaan. Pelkästä koodien lukemisesta tuskin on hyötyä verrattuna omatoimiseen kokeilemiseen, johon luentoesimerkeillä pyrittiin kannustamaan.

"Hei maailma" ja "kaikenlaskija"

Hei maailma

(2016/esimerkit/l05/heimaailma.c)

```
#include<stdio.h>
const char *mjono = "Hei maalima!\n";
int main(int argc, char **argv){
  /* Tämä seuraava rivi tulostaa. */
  printf(mjono);
  return 0;
}
```

Argumentit ja ympäristömuuttujat

(2016/esimerkit/l06/argumentit.c)

```
#include<stdio.h>
#include <stdlib.h>

const char *mjono = "Hei maalima!\n";
int main(int argc, char **argv){
  printf("Ympäristömuuttuja: %s\n", getenv("MUN_OMA_ENVI"));
  for (int i=0; i<argc; i++){
    printf("Argumentti nro %d on %s\n",i,argv[i]);
  }
  return 123;
}
```

Konekieli ja Linux-käyttöjärjestelmäkutsu

(2016/esimerkit/l07/helloasm_kommentoitu.s)

```
################################################################
#
# Minimalistinen Hei maailma -sovellus GNU Assemblerilla.
#
################################################################
#
# Tässä demonstroidaan käyttöjärjestelmäkutsua (syscall -käsky
# AMD64 -arkkitehtuurissa). Näin se loppuviimein tapahtuu aina
# jossakin matalimman tason alustakirjaston syövereissä. Tämä on
# nyt täysin kiinni x86-64 -arkkitehtuurissa, Linuxissa ja GNU:n
# assembler-syntaksissa.
#
# Tehdään suoritettavaksi ohjelmaksi seuraavilla GNU-työkalujen
# komennoilla esim. suorakäyttökoneissamme:
#
#   as -o helloasm.o helloasm_kommentoitu.s
#   ld -o helloasm helloasm.o
#
# Ensimmäinen käyttää assembler-kääntäjää komennolla 'as'
# tuottaakseen objektitiedoston helloasm.o
#
# Toinen linkittää objektin helloasm.o suoritettavaksi tiedostoksi
# nimeltä helloasm. Sitä voi sitten ajaa ja debugata luennon tapaa
# seuraavilla komennoilla:
#
#   ./helloasm
#   gdb helloasm
#
# Seuraavalla luennolla viimeistellään koko kuvion kulku eli:
#
#   lähdekoodi -> assembly -> objekti -> executable -> prosessi
#
################################################################


# Julkaistaan myöhemmin määriteltävä symbolinen muistiosoite _star
# "globaalisti", koska sen perusteella linkkeri tunnistaa,
# mistä kohtaa alkaa aloituspisteeksi tarkoitettu ohjelmakoodi.
```

```
        .globl  _start

        # Kirjoitetaan koodisegmenttiin ("text") nyt sekä data että koodi.
        # Laajemmissa ohjelmissa olisi erikseen osiot ainakin koodille (te
        # vakiodatalle (data) ja nollaksi alustettavalle työtilalle (bss).
        # Seuraava ohjerivi kertoo assemblerille, että nyt pitää alkaa
        # tuutata tavaraa koodialueelle:

        .text

.hei_mun_maailmani:             # (Symbolinen nimi muistiosoitteelle)
        .ascii  "Hel"           # Sisältöä merkkeinä. Tässä merkit
        .ascii  "lo world."     # kopioituvat ohjelmakoodiin peräkkäin
        .byte   0xa             # ... ja sekaan voi heittää lukuarvoja
        .ascii  "jeijee!"       # ... kaikki menee vaan koodipötköksi
        .byte   0xa             # ... assembler on aika yksinkertaista!
                                # Jos merkkijonon loppuun haluttaisiin
                                # automaattisesti nollamerkki, voisi
                                # käyttää .ascii:n sijasta .string
                                # kuten luentoesimerkissä oli.

.hei_mun_maailmani_loppu:       # (Symbolinen nimi muistiosoitteelle)

_start:                         # (Symbolinen nimi muistiosoitteelle)

        # Yksi pikkujuttu, joka on tarpeen ainakin liukulukuja käsiteltäes
        # en oo ehtinyt selvittää, vaikuttaako, jos ei käytetä liukulukuja
        and     $0xfffffffffffffff0,%rsp

        # (-> ei vaikuta toimintaan normaaleilla käskyillä, mutta on kyllä
        # erikseen ihan speksattu, että pinon täytyy olla jokaisen aliohje
        # kutsun alkaessa 16 tavun rajalla, eli alimmat 4 bittiä nollia!)

        # Järjestelmäkutsuun mennään x86_64 -prosessoriarkkitehtuurissa
        # käskyllä syscall, joka aiheuttaa prosessorin "kevyen" keskeytyks
        # Toimenpiteet on dokumentoitu esim. AMD64-arkkitehtuurimanuaaliss
        # Eri prosessoreissa voi olla eri niminen ja hieman eri tavoin
        # toimiva käsky ns. ohjelmoidun keskeytyksen aikaansaamiseksi.
        #
        # Sen jälkeen prosessori suorittaa käyttöjärjestelmän koodia, jote
```

```
# on täysin käyttöjärjestelmän toteutuksesta riippuvaa, mitä
# ohjelmoidun keskeytyksen jälkeen alkaa tapahtua, ja mitä on pitä
# tapahtua sovellusohjelman puolella, että toivottu käyttöjärjeste
# palvelu saadaan tarkoitetulla tavoin käyttöön.
#
# Linuxin x86_64 -versio odottaa RAX-rekisterissä tietoa, mikä
# nimenomainen järjestelmäkutsu sen pitäisi tehdä. Dokumentaation
# perusteella se tulostaa kutsulla numero 1, joten Helloworldissä
# sovelluksen on laitettava RAX-rekisteriin ykkönen.
#
# Kyseinen tulostuspalvelu odottaa juuri tietyissä rekistereissä
# olevan tietyt parametrit kutsua varten:
#
# RDI:ssä tulostuksen kohteena olevaa tiedostoa/tietovirtaa kuvaav
# numero. Standardi ulostulovirta on avattu valmiiksi ennen ohjelm
# käynnistymistä numerokoodille 1, joten laitetaan se RDI:hin, kun
# halutaan tulostaa standardiulostuloon. Olennaisesti samalla tavo
# tulostettaisiin myös kovalevylle, mutta ensin pitäisi avata tied
# uudelle numerokoodille - arvatenkin eri käyttöjärjestelmäkutsull
#
# RSI:ssä täytyy olla tulostettavan datan alkuosoite muistissa. Me
# on itse määritelty symbolinen nimi, jonka Assembler-kääntäjä osa
# muuntaa. Itse asiassa muistiosoitteen lopullinen numeroarvo saat
# selvitä vasta siinä vaiheessa kun käyttöjärjestelmä lataa tätä
# ohjelmaa suoritukseen ja yhdistelee ohjelman osioita toisiinsa
# muistiavaruuden eri kohtiin. Onneksi voidaan käyttää symboleita.
#
# RDX:ssä täytyy olla tulostettavan datan määrä tavuina. Onneksi
# voidaan laskea "pituus = loppuosoite - alkuosoite". Tästä tulee
# konkreettinen lukuarvo jo käännösvaiheessa, koska data sijaitsee
# peräkkäisissä muistiosoitteissa, eikä niiden keskinäinen
# suhteellinen sijainti muutu ohjelman lataamisessa tai
# linkittämisessä.
#
# Parametrit kun on laitettu kohdilleen, niin sitten tarvitsee vaa
# pamauttaa prosessori keskeytyskäsittelyyn ja luottaa, että ollaa
# suorituksessa Linux-käyttöjärjestelmän sellaisen version päällä,
# jonka järjestelmäkutsurajapinnassa on samat numerot käytössä kui
# tätä ohjelmaa käännettäessä.
#
# Ja ymmärretään pari asiaa: (1) Konekieli, jota tietokone suoritt
```

```
# on yksinkertaista (2) Assembler on yksinkertaista (3) Alimman ta
# käyttöjärjestelmärajapinta on yksinkertainen (4) Kaikesta tästä
# yksinkertaisuudesta johtuen ei ole juuri mitään järkeä käyttää
# laitetta ilman monitasoista abstraktiota, josta C-kielinen
# Linux-rajapinta on seuraava ylempi taso, POSIX-standardin määrää
# yhteensopivien käyttöjärjestelmien C-kutsurajapinta sitä seuraav
# ja näiden päälle tietenkin löytyy erilaisia ohjelmointikieliä ja
# kirjastoja, jotka edelleen abstrahoivat toimintaa
# yleiskäyttöisemmäksi ja helpommaksi. (5) Johtopäätös:
# Yksinkertainen ei tosiaankaan ole sama kuin helppokäyttöinen.

movq   $1, %rax
movq   $1, %rdi
movq   $.hei_mun_maailmani, %rsi
movq   $.hei_mun_maailmani_loppu - .hei_mun_maailmani, %rdx
syscall

# Seuraava on helppo selittää edellisen sepustuksen jälkeen:
# Linuxin x86_64 -versio lopettaa ohjelman kutsulla numero 60, jok
# odottaa, että virhekoodiksi tarkoitettu luku on rekisterissä RDI

movq $60,%rax
movq $0,%rdi
syscall
```

Ikuinen silmukka; demonstroi aikakatkaisun tarvetta

(2016/esimerkit/l10/kaikenlaskija.c)

```
int main(int argc, char **argv){
  unsigned int a;
  while(1){
    a++;
  }
  return 0;
}
```

Pieni esimerkki kokonaislukuvakioista C-koodissa

(2016/esimerkit/l10/lukuja.c)

```
int main(int argc, char **argv){
  unsigned int a;
  printf("%d \n", 123 + 0x123 + 0123);
  return 0;
}
```

Rekursiivinen aliohjelmakutsu

(2016/esimerkit/l10/rekursio.c)

```
#include<stdio.h>
// Laskee n*(n-1)*(n-2)...*1 rekursiivisesti.
int kertoma(int n){
  if (n==0) return 1;
  return n*kertoma(n-1);
}

int main(int argc, char **argv){
  unsigned int a;
  printf("Luvun 9 kertoma on %d \n", kertoma(9));
  return 0;
}
```

Minimalistinen shell-ohjelma, fork() ja exec()

(2016/esimerkit/l11/minish.c)

```
/* minish.c - example of a minimalistic shell
 * Comments in code deliberately omitted. :-)
 *
 * [ Lainattu Käyttöjärjestelmät -kurssille esimerkkinä shellin
 *   perusideasta ja Unixin fork() ja exec() -kutsujen käytöstä.
 *   Siirretty merkkijonon jäsentäminen omaan aliohjelmaansa, jotta
 *   pääohjelma näyttäisi selkeämmältä.
 *
 *   Alkuperäinen koodi on Tapani Tarvaisen kurssimateriaalista "UNIX
 *   ja shell-ohjelmointi". Tarkempia kommentteja koodissa ei ole,
 *   koska tämä on Tapanin kurssin tehtävämateriaalia ja perinteinen
 *   tenttikysymys myös Käyttöjärjestelmät -kurssilla (...esimerkiksi
 *   kommentoimaton koodi annetaan, ja toiminta on selitettävä tmv.)
```

```
 *   Tärkeä kohta ovat fork() ja execve() -kutsut: mitä ne tekevät,
 *   miten käyttöjärjestelmän tietorakenteet silloin muuttuvat, miten
 *   forkin paluuarvoa tulkitaan sovellusohjelmassa, ja millaiset syyt
 *   voivat johtaa minkäkin kutsun epäonnistumiseen!
 *
 *   Periaate selitetty luentomonisteessa; tarkka totuus rajapinnasta
 *   löytyy esim. POSIXista ja todellinen toteutustapa esim. GCC:n
 *   C-kirjastoista ja Linuxin lähdekoodista. (Vain periaate ja
 *   sovellusrajapinta ovat johdantokurssin asiaa).
 *
 *   Terveisin Paavo 2007-06-05, 2014-04-01, 2015-04-23, 2016-04-27. ]
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
#define MAXLINE 50

/**
 * Suomennetaan kutsu C:n esikääntäjämakrolla :) Palauttaa toden, jos
 * syötteessä on lisää rivejä juuri luetun rivin jälkeen.
 */
#define lue_komentorivi(s, n, f) (fgets(s, n, f))

/**
 * Pilkkoo merkkijonon välilyönneistä. Palauttaa 0, jos eka merkki on
 * '#' ja muutoin 1. (vain esittely koodin alussa; toteutus saa olla
 * missä vain)
 */
int jasenna_komentorivi(char *line, char **args);

/**
 * Pääohjelma. Olennainen ymmärrettävä osuus merkitty erikseen alempana.
 */
int main(int argc, char **argv)
{
   char line[MAXLINE+1], *args[MAXLINE];
```

```
    if (argc>=2 && !freopen(argv[1], "r", stdin)) {
        perror(argv[1]);
        exit(errno);
    }

    /* TÄMÄ WHILE-SILMUKKA YMMÄRRETTÄVÄ KOKONAISUUDESSAAN: */
    while (lue_komentorivi(line, MAXLINE, stdin)) {
        if (!jasenna_komentorivi(line,args)) continue;

        if (!strcmp(line,"exit")) {
            exit(args[1] ? atoi(args[1]) : 0);
        }

        switch (fork()) {
        case 0:
            execvp(args[0], args);
            perror(args[0]);
            exit(errno);
        case -1:
            perror("fork() failed");
            break;
        default:
            wait(NULL);
        }
    }
    exit(0);
}

/* Aiemmin esitellyn aliohjelman toteutus (ei olennainen KJ-kurssilla). *
int jasenna_komentorivi(char *line, char **args){
    char *p, **arg;
    if ((p = strchr(line, '\n'))) {
        *p=0;
    } else {
        fprintf(stderr, "line too long\n");
        exit(1);
    }
    p = line;
    while (*p == ' ')
        ++p;
    if (*p == '#')
```

```
        return 0;
    *(arg=args) = p;
    while ((p = strchr(p, ' '))) {
        while (*p == ' ')
            *p++ = 0;
        if (*p)
            *++arg=p;
    }
    *++arg=NULL;
    return 1;
}
```

Prosesseja, säikeitä ja synkronointia

Signaalinkäsittelijän rekisteröiminen

(2016/esimerkit/l12/sigesim.c)

```
/* Prosessien välistä kommunikointia yksinkertaisimmillaan: Signaalit.
 *
 * Testattu GNU/Linux x86_64 -ympäristössä. Muualla ei ole mitään
 * takuita. Pitäisi kyllä olla ihan standardia Unix -kamaa. (Paitsi
 * ei tavallaan olekaan: POSIXin nykyinen versio määrittelee kyllä
 * tämänkin, mutta suosittelee käyttämään uusissa ohjelmissa
 * luotettavampaa ja monipuolisempaa rajapintaa signaaleille. Luetaan
 * lause kuitenkin niin kuin se onkin, että suositeltu tapa on
 * monimutkaisempi ja perusesimerkkiin vähemmän soveltuva kuin tämä
 * vanha tapa.)
 *
 * Näin voidaan C-kielen avulla määritellä käsittelijäaliohjelmia
 * käyttöjärjestelmän välittämille signaaleille, jotka voivat tulla
 * esimerkiksi muilta prosesseilta tai päätettä näppäilemällä. Tämä
 * ohjelma käsittelee päätenäppäilyt Ctrl-C, Ctrl-Z ja Ctrl-\ itse
 * määräämällään tavalla. Samat signaalit voidaan lähettää unixissa
 * apuohjelmalla kill ("man kill" kertoo tietenkin enemmän.).
 *
 * Suomalaisesta näppäimistöstä on vaikea loitsia "Ctrl-\"
 * -painallusta, jonka SIGABRT -käsittelijä poimisi. Mutta saman asian
 * voit tosiaan tehdä, kun toisessa pääteikkunassa samaan koneeseen
 * komennat::
```

```
 *
 *   kill -6 PID
 *
 * missä PID on käynnissä olevan sigesim-ohjelman prosessitunnus. Sen
 * löydät vaikkapa katsomalla omien prosessiesi listaa::
 *
 *    ps -u `whoami`
 *
 * Ohjelman lopetusta kannattaa pyytää signaaleilla SIGTERM tai SIGINT
 * eli "kill -15 PID" tai "kill -2 PID". Jos ei MIKÄÄN MUU AUTA, niin
 * "kill -9 PID" yleensä lopettaa ohjelman eikä sitä voi estää edes
 * ohjelman määrittelemällä signaalikäsittelijällä. Lopetussignaalit
 * kannattaa käsitellä sovelluksissa, ja tehdä tarvittavat
 * automaattiset tallennukset, resurssien vapauttamiset, ynnä muut
 * elintärkeät operaatiot, ja sitten lopettaa ohjelma
 * tyylikkäästi. Tuo signaali KILL eli "kill -9" on armoton, koska
 * sitä ei voi napata lopputoimien suorittamiseksi! Ohjelman
 * käsittelemät tiedot voivat hukkua / korruptoitua. Siis käytä
 * ensisijaisesti INT ja TERM -signaaleja.
 *
 */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

/* C:ssä pitää esitellä nimet ennen niiden käyttöä. Aliohjelmien
 * toiminnallisuuden määrittely kuitenkin vasta koodin lopussa.
 */
void handle_sigint(int);
void handle_sigtstp(int);
void handle_sigabrt(int);

int main(int argc, char **args)
{
  /* Seuraavilla kutsuilla määritellään omat signaalikäsittelijät
   * (annetaan signal.h:ssa määritellyt kokonaislukutunnukset ja
   * muistiosoitteet itse kirjoitettuihin aliohjelmiin. C-kielessä
   * aliohjelman nimi ilman aktivointia eli sulkumerkkisyntaksia
   * "aliohj()" tarkoittaa aliohjelman alun muistiosoitetta.):
   */
```

```
  signal(SIGINT, handle_sigint);
  signal(SIGTSTP, handle_sigtstp);
  signal(SIGABRT, handle_sigabrt);

  for(;;){
    printf("Tämä ohjelma menee nyt unille...\n");
    pause();
    printf("Heräsi unilta! Ilmeisesti tuli signaali!\n");
  }
  return 1;
}

void handle_sigint(int a)
{
  /* Viritetään käsittelijä varmuuden vuoksi uudelleen, koska joissain
   * toteutuksissa kuulemma signaalikäsittelijä tulkitaan
   * kertakäyttöiseksi ja se on tässä vaiheessa palannut
   * oletuskäsittelijäksi.
   */
  signal(SIGINT, handle_sigint);
  printf("\"Ctrl-C\" signaali poimittu. Mutta ei tehdä mitään\n");
}

void handle_sigtstp(int a)
{
  signal(SIGTSTP, handle_sigtstp);
  printf("\"Ctrl-Z\" signaali poimittu. Mutta ei tehdä mitään\n");
}

void handle_sigabrt(int a)
{
  printf("\"Ctrl-\\\" signaali poimittu. Lopetetaan.\n");
  exit(0);
}
```

Viestijono prosessien välillä

(2016/esimerkit/l12/chattomyself.c)

```
/* Esimerkki viestin välityksestä viestijonon kautta: viestin lähetys
 * ja vastaanotto
```

```
 *
 * Esimerkki myös siitä, miten deadlock-tilanne eli lukkiutuminen
 * esimerkiksi voi tapahtua ohjelmointivirheen takia *joissain
 * tilanteissa* ja taas joissain tilanteissa homma toimii kuten on
 * suunniteltu. Tämä lelu-chatti periaatteessa toimii kahden
 * keskustelijan välillä oikein hyvin, **paitsi** jos jompikumpi
 * jossain vaiheessa sattuu sanomaan esimerkiksi, että "Eipä tässä sen
 * kummempaa kuulu..." tai muuta isolla E-kirjaimella alkavaa...
 *
 *   Älä kokeile esimerkkiä käytännössä yhteiskäytössä olevilla
 *   koneilla ellet ole äärimmäisen vahvasti hajulla siitä, mitä
 *   tapahtuu! (Tai ainakin osaat tarkastaa ja sulkea mahdollisesti
 *   jumiin jääneet ohjelmasi kill -apuohjelman avulla..)
 *
 * Jos kokeilet, niin mieluiten koti-Linuxilla tai joka tapauksessa
 * jossain, missä ihan itse olet järjestelmänvalvoja. Liikutaan
 * alueella, jossa on helppo aiheuttaa itkua ja hammastenkiristelyä
 * kaikille kyseisen tietokoneen käyttäjille...
 *
 * JOS olet turvallisessa paikassa ja haluat kokeilla, niin käynnistä
 * prosessit näillä argumenteilla:
 *
 *   chattomyself 1 2      # Tällä on ensimmäinen puheenvuoro
 *
 *   chattomyself 2 1      # Tällä on toinen ja niin edelleen puheenvuoro
 *
 * Stolen and adapted, with utmost gratitude, from:
 *   http://www.cs.cf.ac.uk/Dave/C/node25.html#SECTION002500000000000000
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MSGSZ   128

/* C:ssä ei ole kunnon syöttöaliohjelmia. Rykäistään oma (eli
 * pollitään netistä jonkun tekele, joka näyttää järkevältä...)
 */
```

```
int my_getline(char* line, int max)
{
  int nch = 0;
  int c;
  max = max - 1; /* leave room for '\0' */

  while((c = getchar()) != EOF)
    {
      if(c == '\n')
        break;

      if(nch < max)
        {
          line[nch] = c;
          nch = nch + 1;
        }
    }

  if(c == EOF && nch == 0)
    return EOF;

  line[nch] = '\0';
  return nch;
}


/*
 * Määritellään, millaisia viestejä oma sovellus lähettelee.
 * Kokonaisluku pitää olla kaikissa viesteissä (se on määritelty
 * käyttöjärjestelmäkutsujen rajapinnassa). Tässä meidän viestiosa on
 * merkkijono, mutta se voisi olla mikä tahansa muukin tietorakenne,
 * kunhan sen pituus tavuina tiedetään.
 */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;


/* Pääohjelma */
```

```
int main(int argc, char **argv)
{
    int msqid;
    key_t key;
    message_buf buf;
    size_t buf_length;
    int i=0;

    long mytype, othertype;

    if (argc != 3){
       fprintf (stderr, "Usage: %s mytype othertype\n", argv[0]);
       return 1;
    }

    mytype = atoi(argv[1]);
    othertype = atoi(argv[2]);


    /* Luodaan viestijono */
    key = 1234;

    if ((msqid = msgget(key, IPC_CREAT | 0666 )) < 0) {
        perror("msgget");
        exit(1);
    }

    /* Onnistui, jos ei virhettä...*/
    fprintf(stderr,"Viestijonomme id = %d\n", msqid);

    while(1){
      if (mytype == 1 || i >= 1){
        printf("Kirjoita viesti: ");
        my_getline(&buf.mtext, MSGSZ);

        if (buf.mtext[0] != 'E') {
          /* Viestin lähetys, lähetetään 'othertype' tyyppinen */
          buf.mtype = othertype;
          buf_length = strlen(buf.mtext) + 1 ;
          if (msgsnd(msqid, &buf, buf_length, IPC_NOWAIT) < 0) {
            perror("msgsnd");
```

```
            exit(1);
          }else{
            printf("Lähetetty: \"%s\"\n", buf.mtext);
          }
          if (strcmp(buf.mtext,"bye") == 0) break;
        }
      }

      printf("Odotan vastausta ...\n");

      /* Vastaanotto, kuunnellaan 'mytype'-tyyppisiä */
      if (msgrcv(msqid, &buf, MSGSZ, mytype, 0) < 0) {
        perror("msgrcv");
        exit(1);
      }

      printf("Vastaanotettu: %s\n", buf.mtext);
      i++;
    }

    if (msgctl(msqid, IPC_RMID, 0) == -1) {
        perror("msgctl");
    }

    exit(0);
}
```

Rinnakkaislaskentaa säikeistämällä

Ohjelma, jonka saisi nopeammaksi hajauttamalla laskennan moneen prosessoriytimeen: (2016/esimerkit/l12/saikeiden_tarve.c)

```
/**
 * Mihin saatettaisiin tarvita säikeitä? Tässä on esimerkki
 * rinnakkaistuvasta tehtävästä, jonka seinäkelloaika saadaan
 * lyhyemmäksi rinnakkaislaskennan avulla.
 *
 * Muita tarpeita? Tausta-ajot, esim. grafiikkaohjelman
 * efektialgoritmit tai nettiselaimen välilehdet tai
 * Download-ominaisuus. Käyttöliittymäsäiettä voisi klikkailla, vaikka
```

```
 * ohjelma tekisi samaan aikaan taustatöitä eri paikoissa koodia ja
 * dataa.
 */

#include<stdint.h>
#include<stdio.h>
#include<inttypes.h>
#include<stdlib.h>

#define TAULUN_KOKO 60000000

/**
 * Täyttää taulukon peräkkäisillä, indeksin mukaisilla luvuilla.
 *
 * Jokainen alkio on riippumaton jokaisesta muusta alkiosta.
 * Tällainen tehtävä rinnakkaistuu helposti ja nopeutuu lähestulkoon
 * verrannollisesti rinnakkaisten prosessorien määrään.
 *
 */
void tayta_perakkaisilla(int64_t *t, int64_t a, int64_t b){
    for(int64_t i = a; i<b; i++){
        t[i] = i;
    }
}

int main(int argc, char **argv){
    int64_t *taulukko = malloc(TAULUN_KOKO*sizeof(int64_t));
    tayta_perakkaisilla(taulukko, 0, TAULUN_KOKO);

    // Köyhän miehen testi.. tokko toimi sinne päinkään (pistokoe):
    for(size_t ind=0;ind<TAULUN_KOKO;ind += (TAULUN_KOKO/7+1)){
        printf("taulukko[%d]==%" PRId64 "\n", (int)ind, taulukko[ind]);
    }
    printf("taulukko[%d]==%" PRId64 "\n", TAULUN_KOKO-1, taulukko[TAULUN_K
    return 0;
}
```

Nopeutettu versio: (2016/esimerkit/l12/saikeet.c)

```
/**
 * Pikku esimerkki POSIX-säikeistä.
```

```
 *
 * Mihin saatettaisiin tarvita säikeitä? Tässä on esimerkki
 * rinnakkaistuvasta tehtävästä, jonka seinäkelloaika saadaan
 * lyhyemmäksi rinnakkaislaskennan avulla.
 *
 * Muita tarpeita? Tausta-ajot, esim. grafiikkaohjelman
 * efektialgoritmit tai nettiselaimen välilehdet tai
 * Download-ominaisuus. Käyttöliittymäsäiettä voisi klikkailla, vaikka
 * ohjelma tekisi samaan aikaan taustatöitä eri paikoissa koodia ja
 * dataa.
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */

#include<stdint.h>
#include<stdio.h>
#include<inttypes.h>
#include<stdlib.h>
#include<pthread.h>

#define TAULUN_KOKO 60000000
#define MAX_SAIKEITA 16

typedef struct {
   int64_t *taulukko;
   int64_t alkuindeksi;
   int64_t loppuind_plusyks;
} saikeen_tiedot;

void tayta_perakkaisilla(int64_t *t, int64_t a, int64_t b){
   for(int64_t i = a; i<b; i++){
      t[i] = i;
   }
}

void * saikeen_koodi(void *tied) {
   int64_t *t = ((saikeen_tiedot*)tied)->taulukko;
   int64_t a = ((saikeen_tiedot*)tied)->alkuindeksi;
```

```
    int64_t b = ((saikeen_tiedot*)tied)->loppuind_plusyks;
    tayta_perakkaisilla(t,a,b);
    return NULL;
}

void tayta_perakkaisilla_saikeissa(int64_t *t, int64_t a, int64_t b, size
    if ((b-a) < nthr) nthr = 1;
    if (nthr >= MAX_SAIKEITA) nthr = MAX_SAIKEITA;
    pthread_t th[MAX_SAIKEITA];
    saikeen_tiedot tied[MAX_SAIKEITA];

    int64_t vali = (b-a)/nthr;
    int64_t jakopiste = a;
    for(int i=0;i<nthr;i++){
        tied[i].taulukko=t;
        tied[i].alkuindeksi=jakopiste;
        jakopiste += vali;
        tied[i].loppuind_plusyks=((jakopiste<=b)?jakopiste:b);
        printf("a=%" PRId64 " b=%" PRId64 "\n",tied[i].alkuindeksi,tied[i]
    }
    for(size_t i=0;i<nthr;i++){
        pthread_create(&th[i], NULL, saikeen_koodi, &tied[i]);
    }
    for(size_t i=0;i<nthr;i++){
        pthread_join(th[i], NULL);
    }
}

int main(int argc, char **argv){
    int64_t *taulukko = malloc(TAULUN_KOKO*sizeof(int64_t));
    //tayta_perakkaisilla(taulukko, 0, TAULUN_KOKO);
    size_t nsaikeita;
    if (argc < 2){
        nsaikeita = 1;
    } else {
        nsaikeita = atoi(argv[1]);
    }

    tayta_perakkaisilla_saikeissa(taulukko, 0, TAULUN_KOKO,nsaikeita);

    // Köyhän miehen testi.. tokko toimi sinne päinkään (pistokoe):
```

```
    for(size_t ind=0;ind<TAULUN_KOKO;ind += (TAULUN_KOKO/7+1)){
        printf("taulukko[%d]==%" PRId64 "\n", (int)ind, taulukko[ind]);
    }
    printf("taulukko[%d]==%" PRId64 "\n", TAULUN_KOKO-1, taulukko[TAULUN_K
    return 0;
}
```

Kilpa-ajotilanne ja datan korruptoituminen

Ongelmallinen koodi, jossa on "data race": (2016/esimerkit/l13/race.c

```
/** Pikku esimerkki yhdenaikaisuuden aiheuttamasta kilpa-ajosta eli
 *  race conditionista. Summaus globaaliin muuttujaan vaatii useita
 *  konekielisiä käskyjä, joiden välissä voi tulla kellokeskeytys!
 *  Korjaus keskinäisesti poissulkevalla (mutual exclusion, mutex)
 *  -lukolla myöhemmässä esimerkissä.
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */

#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <pthread.h>

#define N 100000000
uint64_t summa = 0;

void * saikeen_koodi(void *v) {
    int i;
    for (i = 1; i <= N; i++){
        summa++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
     pthread_t saieA, saieB;
```

```
    pthread_create(&saieA, NULL, saikeen_koodi, NULL);
    pthread_create(&saieB, NULL, saikeen_koodi, NULL);

    pthread_join(saieA, NULL);
    pthread_join(saieB, NULL);

    printf("Summa on %" PRId64 " - pittäis olla %" PRId64 "\n", summa, (u

    return 0;
}
```

Korjattu POSIXin simppelillä MutExilla: (2016/esimerkit/l13/race_

```
/** Pikku esimerkki yhdenaikaisuuden aiheuttamasta kilpa-ajosta eli
 *  race conditionista. Summaus globaaliin muuttujaan vaatii useita
 *  konekielisiä käskyjä, joiden välissä voi tulla kellokeskeytys!
 *
 *  Ongelma on tässä hallittu keskinäisesti poissulkevalla (mutual
 *  exclusion, mutex) lukolla: Lukitus vaatii viimekädessä
 *  käyttöjärjestelmän palveluita, koska sillä halutaan vaikuttaa
 *  vuoronnusjärjestykseen ja säikeiden (tai prosessien)
 *  tilasiirtymiin. Jos lukko on jollain muulla varattuna, uusi
 *  pyytäjä on otettava pois normaalista prosessikierrosta ja
 *  "blokattava" eli laitettava odottamaan lukon vapauttamista.
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */

#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <pthread.h>

#define N 100000000
uint64_t summa = 0;

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

```
/** Nyt on synkronoitu lukolla, mutta suorituskyvyn isku on aika
 *  suuri. Johtopäätös: Älä laske summaa näin, jos mahdollista (mutta
 *  muista lukottaa mikä tahansa jaettua resurssia käsittelevä koodi,
 *  josta voi aiheutua kilpa-ajotilanne!!)
 */
void * saikeen_koodi(void *v) {
    int i;
    for (i = 1; i <= N; i++){
        pthread_mutex_lock(&mymutex);
        summa++;
        pthread_mutex_unlock(&mymutex);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t saieA, saieB;

    pthread_create(&saieA, NULL, saikeen_koodi, NULL);
    pthread_create(&saieB, NULL, saikeen_koodi, NULL);

    pthread_join(saieA, NULL);
    pthread_join(saieB, NULL);

    printf("Summa on %" PRId64 " - pittäis olla %" PRId64 "\n", summa, (ui

    return 0;
}
```

(2016/esimerkit/l13/race_fixed_sem_mutex.c) Korjattu POSIXin semaforilla toteutetulla MutExilla:

```
/** Pikku esimerkki yhdenaikaisuuden aiheuttamasta kilpa-ajosta eli
 *  race conditionista. Summaus globaaliin muuttujaan vaatii useita
 *  konekielisiä käskyjä, joiden välissä voi tulla kellokeskeytys!
 *
 *  Ongelma on tässä hallittu keskinäisesti poissulkevalla (mutual
 *  exclusion, mutex) lukolla, joka on toteutettu semaforilla. Tämä on
 *  "ensimmäinen esimerkki" semaforin käytöstä ja siitä kuinka MutEx
 *  voidaan sellaisella toteuttaa. Semafori on kuitenkin
```

```
 *  yleiskäyttöisempi, ja tässä tapauksessa tarpeettoman "vahva"
 *  menetelmä, koska pthreads-kirjastossa on myös yksinkertaisempi
 *  poissulkuratkaisu: race_fixed_mutex.c
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */

#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <pthread.h>
#include <semaphore.h>

#define N 100000000
uint64_t summa = 0;

sem_t mymutex;

/**
 * Nyt on synkronoitu semaforilla (vertaa pthread -kirjaston mutexiin).
 * Hiukan monimutkaisempi; tähän tarkoitukseen pthreadin mutex parempi.
 */
void * saikeen_koodi(void *v) {
    int i;
    for (i = 1; i <= N; i++){
        sem_wait(&mymutex); // Dijkstran "P()-operaatio"
        summa++;
        sem_post(&mymutex); // Dijkstran "V()-operaatio"
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t saieA, saieB;

    sem_init(&mymutex, 0, 1);

    pthread_create(&saieA, NULL, saikeen_koodi, NULL);
```

```
    pthread_create(&saieB, NULL, saikeen_koodi, NULL);

    pthread_join(saieA, NULL);
    pthread_join(saieB, NULL);

    sem_destroy(&mymutex);

    printf("Summa on %" PRId64 " - pittäis olla %" PRId64 "\n", summa, (ui

    return 0;
}
```

Deadlock -tilanne

(2016/esimerkit/l13/vappu_deadlock.c)

```
/**
 * Pikku esimerkki deadlockista.
 *
 * Esimerkki luennolta vuoden 2015 vapun alla. Vuonna 2016 vasta
 * viikko vapun jälkeen, mutta eiköhän vappu muisteta vielä, niin
 * saadaan tästä vappusimulaattorista irti huvi hyödyn lisäksi.
 *
 *  Tehdään kaksi säiettä, jotka juovat simaa ja syövät munkkia.
 *  Molemmat resurssit ovat yhteisessä pöydässä, joten simasammio ja
 *  munkkilaari täytyy lukita, että yhdistetty vappunautinto onnistuu.
 *
 *  Ikävä kyllä säikeet tekevät tässä lukituksensa ristiriitaisessa
 *  järjestyksessä, jolloin mahdollistuu sattumanvarainen "kuolettava"
 *  lukkiutuminen eli deadlock-tilanne.
 *
 * Tämä esimerkki on triviaali, mutta voit mielessäsi ekstrapoloida
 * todelliseen maailmaan, jossa halutaan käyttää esim. useita jonkin
 * ulkoisen järjestelmän tiedostoja, joihin pitää ehkä tehdä
 * muutoksiakin. Toivottavasti järjestelmään on sovittu jokin
 * "valetiedosto" nimeltä LOCK_EVERYTHING_FOR_TRANSACTION tai
 * vastaava, ja olet lukenut dokumentaatiosta, että siihen tulee
 * hankkia itselle kirjoitusoikeus atomaarisella
 * käyttöjärjestelmäkutsulla ennen kuin mitään muita tiedostoja saisi
 * pahemmin lähteä käyttelemään...
 */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>
#include <pthread.h>

#define VAPPUANNOSTEN_TARVE_LKM 1000

uint64_t simaa_ltr = 1000;
uint64_t munkkeja_kpl = 1000;

pthread_mutex_t mutex_sima = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_munkit = PTHREAD_MUTEX_INITIALIZER;

bool juo_simaa(const char *juoja){
    if (simaa_ltr == 0){
        fprintf(stderr, "Ou nou! Sima on loppu!\n");
        return false;
    }
    fprintf(stdout, "%s: Yksi litra simaa juotu.\n", juoja);
    simaa_ltr--;
    return true;
}

bool ota_munkki(const char *ottaja){
    if (munkkeja_kpl == 0){
        fprintf(stderr, "Ou nou! Munkit on loppu!\n");
        return false;
    }
    fprintf(stdout, "%s: Yksi munkki syöty.\n", ottaja);
    munkkeja_kpl--;
    return true;
}

void * paavo(void *v) {
    const char idstr[] = "Paavo";
    int a = 0;
    for (int i = 1; i <= VAPPUANNOSTEN_TARVE_LKM; i++){
        pthread_mutex_lock(&mutex_sima);
```

```
        pthread_mutex_lock(&mutex_munkit);
        if (juo_simaa(idstr) && ota_munkki(idstr)){
            printf("%s: Nams, olipa hyvää. Olen saanut %d annosta.\n",idst
        } else {
            printf("%s: Hitsi, mulle ei riittänyt! Vappu on peruttu...\n",
        }
        pthread_mutex_unlock(&mutex_sima);
        pthread_mutex_unlock(&mutex_munkit);
    }
    printf("%s: %s Sain yhteensä %d täyttä annosta.\n",
           idstr,(a>0)?"Nams, olipa hyvää.":"",a);
    return NULL;
}

void * tomi(void *v) {
    const char idstr[] = "Tomi";
    int a = 0;
    for (int i = 1; i <= VAPPUANNOSTEN_TARVE_LKM; i++){
        pthread_mutex_lock(&mutex_munkit);
        pthread_mutex_lock(&mutex_sima);
        if (juo_simaa(idstr) && ota_munkki(idstr)){
            printf("%s: Nams, olipa hyvää. Olen saanut %d annosta.\n",idst
        } else {
            printf("%s: Hitsi, mulle ei riittänyt! Vappu on peruttu...\n",
        }
        pthread_mutex_unlock(&mutex_sima);
        pthread_mutex_unlock(&mutex_munkit);
    }
    printf("%s: %s Sain yhteensä %d täyttä annosta.\n",
           idstr,(a>0)?"Nams, olipa hyvää.":"",a);
    return NULL;
}


int main(int argc, char *argv[]) {
    int n;
    pthread_t saieA, saieB;

    if (pthread_create(&saieA, NULL, paavo, NULL) != 0) {
        fprintf(stderr, "Säikeen luonti ei onnistunut.\n");
        exit(1);
```

```
    }
    if (pthread_create(&saieB, NULL, tomi, NULL) != 0) {
        fprintf(stderr, "Säikeen luonti ei onnistunut.\n");
        exit(1);
    }

    if (pthread_join(saieA, NULL) != 0) {
        fprintf(stderr, "Säikeen A odottelussa tuli ongelmia.\n");
    };
    if (pthread_join(saieB, NULL) != 0) {
        fprintf(stderr, "Säikeen B odottelussa tuli ongelmia.\n");
    };

    printf("Simaa jäljellä %" PRId64 "; munkkeja jäljellä %" PRId64 "\n",
           simaa_ltr, munkkeja_kpl);

    return 0;
}
```

(2016/esimerkit/l13/vappu_ei_lukkiudu.c)

```
/**
 * Pikku esimerkki deadlockista.
 *
 *
 * Esimerkki luennolta vuoden 2015 vapun alla. Vuonna 2016 vasta
 * viikko vapun jälkeen, mutta eiköhän vappu muisteta vielä, niin
 * saadaan tästä vappusimulaattorista irti huvi hyödyn lisäksi.
 *
 *  Tehdään kaksi säiettä, jotka juovat simaa ja syövät munkkia.
 *  Molemmat resurssit ovat yhteisessä pöydässä, joten simasammio ja
 *  munkkilaari täytyy lukita, että yhdistetty vappunautinto onnistuu.
 *
 * Tässä esimerkissä vappu ei jumiudu, koska siman lukitseminen
 * tehdään molemmissa säikeissä ensimmäisenä.
 */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
```

```
#include <inttypes.h>
#include <pthread.h>

#define VAPPUANNOSTEN_TARVE_LKM 1000

uint64_t simaa_ltr = 1000;
uint64_t munkkeja_kpl = 1000;

pthread_mutex_t mutex_sima = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_munkit = PTHREAD_MUTEX_INITIALIZER;

bool juo_simaa(const char *juoja){
    if (simaa_ltr == 0){
        fprintf(stderr, "Ou nou! Sima on loppu!\n");
        return false;
    }
    fprintf(stdout, "%s: Yksi litra simaa juotu.\n", juoja);
    simaa_ltr--;
    return true;
}

bool ota_munkki(const char *ottaja){
    if (munkkeja_kpl == 0){
        fprintf(stderr, "Ou nou! Munkit on loppu!\n");
        return false;
    }
    fprintf(stdout, "%s: Yksi munkki syöty.\n", ottaja);
    munkkeja_kpl--;
    return true;
}

void * paavo(void *v) {
    const char idstr[] = "Paavo";
    int a = 0;
    for (int i = 1; i <= VAPPUANNOSTEN_TARVE_LKM; i++){
        pthread_mutex_lock(&mutex_sima);
        pthread_mutex_lock(&mutex_munkit);
        if (juo_simaa(idstr) && ota_munkki(idstr)){
            printf("%s: Nams, olipa hyvää. Olen saanut %d annosta.\n",idst
        } else {
            printf("%s: Hitsi, mulle ei riittänyt! Vappu on peruttu...\n",
```

```
        }
        pthread_mutex_unlock(&mutex_sima);
        pthread_mutex_unlock(&mutex_munkit);
    }
    printf("%s: %s Sain yhteensä %d täyttä annosta.\n",
          idstr,(a>0)?"Nams, olipa hyvää.":"",a);
    return NULL;
}

void * tomi(void *v) {
    const char idstr[] = "Tomi";
    int a = 0;
    for (int i = 1; i <= VAPPUANNOSTEN_TARVE_LKM; i++){
        pthread_mutex_lock(&mutex_sima);
        pthread_mutex_lock(&mutex_munkit);
        if (juo_simaa(idstr) && ota_munkki(idstr)){
            printf("%s: Nams, olipa hyvää. Olen saanut %d annosta.\n",idst
        } else {
            printf("%s: Hitsi, mulle ei riittänyt! Vappu on peruttu...\n",
        }
        pthread_mutex_unlock(&mutex_sima);
        pthread_mutex_unlock(&mutex_munkit);
    }
    printf("%s: %s Sain yhteensä %d täyttä annosta.\n",
          idstr,(a>0)?"Nams, olipa hyvää.":"",a);
    return NULL;
}


int main(int argc, char *argv[]) {
    int n;
    pthread_t saieA, saieB;

    if (pthread_create(&saieA, NULL, paavo, NULL) != 0) {
        fprintf(stderr, "Säikeen luonti ei onnistunut.\n");
        exit(1);
    }
    if (pthread_create(&saieB, NULL, tomi, NULL) != 0) {
        fprintf(stderr, "Säikeen luonti ei onnistunut.\n");
        exit(1);
    }
```

```
    if (pthread_join(saieA, NULL) != 0) {
        fprintf(stderr, "Säikeen A odottelussa tuli ongelmia.\n");
    };
    if (pthread_join(saieB, NULL) != 0) {
        fprintf(stderr, "Säikeen B odottelussa tuli ongelmia.\n");
    };

    printf("Simaa jäljellä %" PRId64 "; munkkeja jäljellä %" PRId64 "\n",
           simaa_ltr, munkkeja_kpl);

    return 0;
}
```

Tuottaja-kuluttaja -probleemi ja sen ratkaiseminen

Rikkinäinen koodi, josta puuttuu synkronointi: (2016/esimerkit/l14/t

```
/** Pikku esimerkki tuottaja-kuluttaja -tilanteesta ilman
 *  synkronointia. Ei tietenkään toimi toivotulla tavalla, koska
 *  säikeiden vuoronnus on sattumanvaraista. Katso korjattu koodi:
 *  tuottaja_kuluttaja.c
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */
#define _POSIX_C_SOURCE 200809L
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <ctype.h>

#define BUF_SIZE 20

/* Meillä on nyt leluesimerkissä vain merkkejä, mutta vois olla
 * videoframeja tai muuta isompaa. Idea olisi sama silloinkin.
 */
```

```
typedef char data_t;

typedef struct {
    int iluku;
    int ikirjoitus;
    data_t data[BUF_SIZE];
} rengaspuskuri_t;

void pikkuinen_tauko(int nsec){
    struct timespec tim;
    tim.tv_sec = 0;
    tim.tv_nsec = nsec;
    nanosleep(&tim,NULL);
}

void rengas_alusta(rengaspuskuri_t *p){
    p->iluku = 0;
    p->ikirjoitus = 0;
}

void rengas_kirjoita(rengaspuskuri_t *p, data_t merkki){
    p->data[p->ikirjoitus] = merkki;
    p->ikirjoitus = (p->ikirjoitus + 1) % BUF_SIZE;
}

data_t rengas_lue(rengaspuskuri_t *p){
    data_t palautettava = p->data[p->iluku];
    p->iluku = (p->iluku + 1) % BUF_SIZE;
    return palautettava;
}

void* tuottaja(void *arg)
{
    char c;
    static int i=0;
    const char tviittausta[] = "Tviit...";

    rengaspuskuri_t *p = (rengaspuskuri_t*) arg;

    for(int n=0;n<1000;n++){
        rengas_kirjoita(p,tviittausta[i]);
```

```
        i = (i + 1) % 8;
        pikkuinen_tauko(5000+(rand()%1500));
    }

    return NULL;
}

void* kuluttaja(void *arg)
{
    char c;
    rengaspuskuri_t *p = (rengaspuskuri_t*) arg;

    for(int n=0;n<1000;n++){
        c = rengas_lue(p);
        fputc(toupper(c),stdout);
        fflush(stdout);
        pikkuinen_tauko(80000+(rand()%1500));
    }
    return NULL;
}

int main(int argc, char **argv)
{
    rengaspuskuri_t rengas;

    pthread_t t,k;

    rengas_alusta(&rengas);

    pthread_create(&t, NULL, tuottaja, &rengas);
    pthread_create(&k, NULL, kuluttaja, &rengas);

    pthread_join(t, NULL);
    pthread_join(k, NULL);

    return 0;
}
```

Korjattu koodi: (2016/esimerkit/l14/tuottaja_kuluttaja.c)

```
/** Pikku esimerkki tuottaja-kuluttaja -tilanteen ratkaisusta kolmella
```

```
 *  semaforilla (luentomonisteen pseudokoodiesimerkin toteutus
 *  POSIX-säikeillä ja POSIX-semaforeilla).
 *
 *  HUOM: Esimerkkien lyhentämiseksi näissä ei tarkisteta,
 *  onnistuivatko operaatiot. Oikeassa ohjelmassa AINA tarkistettava
 *  esim. onnistuiko säikeen luonti vai ei, ja käsiteltävä
 *  epäonnistuminen!!
 */
#define _POSIX_C_SOURCE 200809L
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <ctype.h>

sem_t mutex;
sem_t available;
sem_t used;

#define BUF_SIZE 20

/* Meillä on nyt leluesimerkissä vain merkkejä, mutta vois olla
 * videoframeja tai muuta isompaa. Idea olisi sama silloinkin.
 */
typedef char data_t;

typedef struct {
    int iluku;
    int ikirjoitus;
    data_t data[BUF_SIZE];
} rengaspuskuri_t;

void pikkuinen_tauko(int nsec){
    struct timespec tim;
    tim.tv_sec = 0;
    tim.tv_nsec = nsec;
    nanosleep(&tim,NULL);
}
```

```
void rengas_alusta(rengaspuskuri_t *p){
   p->iluku = 0;
   p->ikirjoitus = 0;
}

void rengas_kirjoita(rengaspuskuri_t *p, data_t merkki){
   p->data[p->ikirjoitus] = merkki;
   p->ikirjoitus = (p->ikirjoitus + 1) % BUF_SIZE;
}

data_t rengas_lue(rengaspuskuri_t *p){
   data_t palautettava = p->data[p->iluku];
   p->iluku = (p->iluku + 1) % BUF_SIZE;
   return palautettava;
}

void* tuottaja(void *arg)
{
   char c;
   static int i=0;
   const char tviittausta[] = "Tviit...";

   rengaspuskuri_t *p = (rengaspuskuri_t*) arg;

   for(int n=0;n<1000;n++){
      sem_wait(&available);
      sem_wait(&mutex);
      rengas_kirjoita(p,tviittausta[i]);
      sem_post(&mutex);
      sem_post(&used);

      i = (i + 1) % 8;
      pikkuinen_tauko(5000+(rand()%1500));
   }

   return NULL;
}

void* kuluttaja(void *arg)
{
   char c;
```

```
    rengaspuskuri_t *p = (rengaspuskuri_t*) arg;

    for(int n=0;n<1000;n++){
        sem_wait(&used);
        sem_wait(&mutex); // Dijkstran "P()-operaatio"
        c = rengas_lue(p);
        sem_post(&mutex); // Dijkstran "V()-operaatio"
        sem_post(&available);
        fputc(toupper(c),stdout);
        fflush(stdout);
        //pikkuinen_tauko(80000+(rand()%1500));
    }
    return NULL;
}

int main(int argc, char **argv)
{
    rengaspuskuri_t rengas;

    pthread_t t,k;

    sem_init(&mutex, 0, 1);
    sem_init(&available, 0, BUF_SIZE);
    sem_init(&used, 0, 0);

    rengas_alusta(&rengas);

    pthread_create(&t, NULL, tuottaja, &rengas);
    pthread_create(&k, NULL, kuluttaja, &rengas);


    pthread_join(t, NULL);
    pthread_join(k, NULL);

    sem_destroy(&mutex);
    sem_destroy(&available);
    sem_destroy(&used);

    return 0;
}
```

Jaetun muistialueen käyttö prosessien välillä

"Leikkipalvelin", joka tekee jaetun muistialueen, ja poistaa sen loppuessaan muutaman sekunnin päästä: (2016/esimerkit/l15/shm_msg

```
/*
 *  shm_msgserver.c
 *
 *  Illustrates memory mapping and persistency, with POSIX objects.
 *
 *  This process produces a message leaving it in a shared segment.
 *  The segment is mapped in a persistent object meant to be subsequently
 *  opened by a shared memory "client".
 *
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unix
 *
 *  Note from nieminen@jyu.fi: This version of Mij's public domain
 *  example is adapted for our spring 2015 and later 2016 course on OS
 *  fundamentals. Students, please see the original site for more
 *  information!
 *
 */

#define _XOPEN_SOURCE 700 /* Single UNIX Specification, Version x */
/* (See POSIX 2008.1 rationale chapter B.2.2 ) */

#include <stdio.h>

/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>

/* exit() etc */
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

/* for random() stuff and small waiting for the race to be evident. */
#include <stdlib.h>
#include <time.h>
```

```
/* POSIX IPC object name [system dependent] - see
 * http://http://mij.oltrelinux.com/devel/unixprg/#ipc__posix_objects
 *
 * ... On Linux, you may observe (and read/write:)) /dev/shm/foo1423
 * after creation. A realistic file should have proper access rights
 * (only for the user who owns the memory!!! or *at most* group),
 * and a name that is unlikely to clash with others in the system
 * (combination of username and application name perhaps?
 * random/nondeterministic names can't be used because the idea is to
 * create the name in all the applications that need the shared
 * segment; graceful handling of pre-existing file should be
 * implemented - the user may always start another instance of a
 * server!).
 */
#define SHMOBJ_PATH      "/foo1423"

/* maximum length of the content of the message */
#define MAX_MSG_LENGTH   50

/* how many types of messages we recognize (fantasy) */
#define TYPES            8

/* message structure for messages in the shared segment */
struct msg_s {
   int type;
   char content[MAX_MSG_LENGTH];
};


int main(int argc, char *argv[]) {
   int shmfd;
   int shared_seg_size = (1 * sizeof(struct msg_s)); /* want shared segme
   struct msg_s *shared_msg;   /* the shared segment, and head of the mes

   /* Forcible cleanup in case we've had an unclean shutdown */
   /* Note: In a real application, termination signals should be
      caught by a signal handler for cleanup! (And that won't protect
      against a kill -9 from the user! As a user, prefer the TERM
      signal over KILL.)*/
```

```
    if (argc > 1) goto cleanup;

    /* creating the shared memory object -- shm_open() */
    shmfd = shm_open(SHMOBJ_PATH, O_CREAT | O_EXCL | O_RDWR, S_IRWXU);

    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    fprintf(stderr, "Created shared memory object %s\n", SHMOBJ_PATH);

    /* adjusting mapped file size (make room for the whole segment to map)
-- ftruncate() */
    ftruncate(shmfd, shared_seg_size);

    /* requesting the shared segment -- mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | P
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d bytes)


    srandom(time(NULL));
    /* producing a message on the shared segment */
    shared_msg->type = random() % TYPES;
    snprintf(shared_msg->content, MAX_MSG_LENGTH, "My message, type %d, nu

    sleep(30); /* adaptation (nieminen 2015): Leave the server "up"
               * for a period, then clean up and "shutdown".
               */

    /* [uncomment if you wish] requesting the removal of the shm object
-- shm_unlink() */

 cleanup:

    if (shm_unlink(SHMOBJ_PATH) != 0) {
        perror("In shm_unlink()");
        exit(1);
```

```
    }

    return 0;
}
```

“Leikkiasiakas”, joka kytkeytyy leikkipalvelimen tekemään muistialueeseen ja muokkaa sen sisältöä. HUOM: Useat yhdenaikaiset asiakasohjelmat sotkevat muistin sisällön kilpa-ajotilanteen vuoksi, kuten luennolla nähtiin: (2016/esimerkit/l15/shm_msgclient.c)

```
/*
 *  shm_msgclient.c
 *
 *  Illustrates memory mapping and persistency, with POSIX objects.
 *  This process reads and displays a message left it in "memory segment
 *  image", a file been mapped from a memory segment.
 *
 *
 *  Note from nieminen@jyu.fi: This version of Mij's public domain
 *  example is adapted for our spring 2015 course on OS fundamentals.
 *  Please see the original site for more information!
 *
 *
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unix
 *
 */

#define _XOPEN_SOURCE 700 /* Single UNIX Specification, Version x */
/* (See POSIX 2008.1 rationale chapter B.2.2 ) */

#include <stdio.h>
/* exit() etc */
#include <unistd.h>
/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
/* for random() stuff */
#include <stdlib.h>
```

```
#include <time.h>


/* Posix IPC object name [system dependant] - see
http://mij.oltrelinux.com/devel/unixprg/index2.html#ipc__posix_objects */
#define SHMOBJ_PATH        "/foo1423"
/* maximum length of the content of the message */
#define MAX_MSG_LENGTH   50
/* how many types of messages we recognize (fantasy) */
#define TYPES            8

/* message structure for messages in the shared segment */
struct msg_s {
    int type;
    char content[MAX_MSG_LENGTH];
};

/* Pause to illustrate the race more often:*/
void pikkuinen_tauko(int nsec){
    struct timespec tim;
    tim.tv_sec = 0;
    tim.tv_nsec = nsec;
    nanosleep(&tim,NULL);
}

int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = (1 * sizeof(struct msg_s)); /* want shared segme
    struct msg_s *shared_msg;   /* the shared segment, and head of the mes


    /* creating the shared memory object -- shm_open() */
    shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU | S_IRWXG);
    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    printf("Created shared memory object %s\n", SHMOBJ_PATH);

    /* requesting the shared segment -- mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | P
```

```
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    printf("Shared memory segment allocated correctly (%d bytes).\n", shar


    printf("Message type is %d, content is: %s\n", shared_msg->type, share
    /* nieminen 2015: Alter the message, in place (with a bad race
     * with multiple concurrent clients!
     *
     * Exercise for those interested: Implement a mutex around this
     * part. Hint: Since the mutex is between processes, you'll need
     * to learn a bit more about POSIX mutexes than is offered on our
     * intro course. For example, the Stackoverflow has nice questions
     * and answers about this):
     */
    char *alt = shared_msg->content;
    if (*alt != '\0'){
        char rot = alt[0];
        for(int i=0;i<MAX_MSG_LENGTH-1;i++){
            if (alt[i+1]=='\0'){
                alt[i] = rot;
                break;
            }
            alt[i]=alt[i+1];
            pikkuinen_tauko(100000+(rand()%150000));
        }
    }

    return 0;
}
```

Välimuistin ruuhkautuminen

Esimerkki suorituskyvyn romahtamisesta, jos välimuistia ei osata hyödyntää: (2016/esimerkit/l15/cache.c)

```
/**
 * Esimerkki Cache thrashing -ilmiöstä.
```

```
 *
 * Ei ole aivan sama, miten päin asettelee algoritminsa silmukat!
 *
 * Luennoilla katsottakoon, kuinka pitkä aika taulukon summan
 * laskemiseen menee kumpaisellakin eri tavalla. Ero on merkittävä, ja
 * se johtuu siitä, että toisessa aika menee prosessorin
 * välimuistirivien vaihtamiseen laskennan sijaan.
 *
 * Tämäkin esimerkki on keinotekoinen, jotta ilmiön luonne nähdään
 * "koeputkessa". Nämä operaatiothan voitaisiin oikeasti tehdä yhdellä
 * silmukalla for(int i=0;i<n*m;i++){...}, joka vastaa tässäkin olevaa
 * oikeellista tapaa, jossa välimuisti ei ruuhkaudu. [Todellisempi
 * esimerkki olisi esim. kahden matriisin kertolasku, jossa toista
 * matriisia luettaisiin riveittäin ja toista sarakkeittain. Silloin
 * voisi olla edullisinta tallentaa sarakkeittain luettava matriisi
 * alunperinkin transpoosina, koska silloin molempien matriisien
 * muistia voitaisiin lukea riveittäin.]
 *
 * Vastaava ilmiö voi esiintyä, jos esimerkiksi oliokielessä käydään
 * läpi pitkää listaa, jossa peräkkäiset viitatut oliot ovat
 * tallennettuna sikin sokin ympäri kekomuistina käytettävää fyysistä
 * muistialuetta. Alustakirjaston oma kekomuistin hallinta voi
 * esim. yrittää siirrellä peräkkäin tallennettuja olioita lähelle
 * toisiaan. Täydellinen keon hallinta onnistuu esim. C++:lla, jossa
 * on teoriassa helppo korvata olioiden varaus- ja vapautuskoodi
 * omalla. Käytännössä alustakirjastot ovat kuitenkin nykyisellään
 * niin hyviä, että niiden kanssa on vaikea kilpailla omilla
 * virityksillä!
 *
 */
#include <stdio.h>
#include <stdlib.h>

#define N 10000

/* Täyttää (m*n) -matriisin annetulla luvulla. */
void tayta(double *taulukko, int n, int m, double luku){
  for (int i=0;i<m;++i){
    for(int j=0;j<n;++j){
      taulukko[i*n+j] = luku;
    }
```

```
  }
}

/* Läpikäynti, jossa välimuisti auttaa: */
double summa(double *taulukko, int n, int m){
  double s=0.;
  for (int i=0;i<m;i++){
    for(int j=0;j<n;j++){
      s += taulukko[i*n+j];
    }
  }
  return s;
}

/* Läpikäynti, jossa välimuisti ruuhkautuu ja suorituskyky romahtaa: */
double summaB(double *taulukko, int n, int m){
  double s=0.;
  for(int j=0;j<n;j++){
    for (int i=0;i<m;i++){
      s += taulukko[i*n+j];
    }
  }
  return s;
}


int main(int argc, char** argv){
    double *t = malloc(N*N*sizeof(double)); /* dyn. varaus N*N liukuluvul
    if (t==NULL) exit(1);

    tayta(t,N,N,1.0);
    double s = summa(t,N,N);
    printf("Summa on %f\n",s);

    free(t); /* eksplisiittinen vapautus; hyvätapaista tehdä aina näin. *
    return 0;
}
```

Tiedostojen käyttöä C:llä

Tulostaa tiedostoon tekstiä (2016/esimerkit/l16/hellofile.c)

```
/**
 * "Hello file" -sovellus. Tulostaa tiedostoon ennalta arvattavan
 * tekstin.
 */
#define _POSIX_C_SOURCE 200809L
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/**
 * Yrittää luoda uuden tiedoston kirjoittamista varten; aliohjelma
 * palauttaa osoittimen tietovirtaolioon tai NULL, jos tiedosto oli jo
 * olemassa tai sitä ei voinut jostain syystä luoda.
 *
 * Kaiketi aika turvallinen tapa luoda tiedosto kirjoittamista varten:
 * soveltuisi tiedoston käyttämiseen lukituksena, koska open()
 * tapahtuu tässä atomisesti POSIXin määräämänä
 * käyttöjärjestelmäominaisuutena. Vain yksi prosessi/säie saa
 * tiedoston auki, ja muut yhdenaikaiset "kilpajuoksijat"
 * epäonnistuvat standardin mukaisesti.
 *
 * Helpommallakin saisi tehtyä tietovirran helloworldia varten
 * (fopen()), mutta tässä saa samalla demonstroitua
 * tiedostodeskriptoria, joka on käyttöjärjestelmän matalimman tason
 * rajapinta tiedoston käsittelyyn. Se on vaan kokonaisluku (jollainen
 * on nähty itse asiassa kurssin alkupuolen helloasm.s -esimerkin
 * syscallin parametrina).
 *
 * Käyttely on helpompaa C-kirjaston tarjoamalla kuorrutuksella (FILE
 * -objekti), joten täällä luodaan deskriptorin ympärille
 * "tietovirtaolio", joka palautetaan. Kutsuja on vastuussa tiedoston
 * sulkemisesta. Sulkeminen onnistuu kuorruteolion kautta.
 *
 */
FILE *luo_kirjoittamista_varten(const char *polku){
   FILE *res;
```

```
    int d = open(polku, O_CREAT | O_EXCL | O_WRONLY, S_IWUSR);
    if (d < 0) {
        if (errno == EEXIST) {
            fprintf(stderr, "Tiedosto '%s' on jo olemassa. En ylikirjoita!
        } else {
            perror("Ongelma tiedoston luonnissa");
        }
        return NULL;
    } else {
        res = fdopen(d, "w"); /* Deskriptorin kuorrutus C:n tietovirraksi.
        if (res == NULL){
            perror("Ongelma tiedoston paketoinnissa virraksi");
        } else {
            return res;
        }
    }
    return NULL;
}

int main(int argc, char *argv[]){
    FILE *tied;
    if (argc < 2){
        fprintf(stderr, "Ilmoita tiedosto, johon kirjoitan.\n");
        exit(1);
    }
    tied = luo_kirjoittamista_varten(argv[1]);

    if (tied == NULL) {
        fprintf(stderr, "Tiedostoa ei saatu luotua.\n");
        exit(1);
    }

    fprintf(tied,
            "Hei, maailma! ... \n"
            "nyt POSIXin C-rajapinnan kautta avatussa tiedostossa! \n");

    if (fclose(tied) != 0){
        perror("Ongelma tiedoston sulkemisessa");
    }
    return 0;
}
```

Lukee tiedostoista merkkejä ja tulostaa ne päätteelle: (2016/esimerkit/l16/liit.c)

```
/** liit - liittää tiedostoja peräkkäin.
 *
 * Vähän niinkuin cat, mutta karvahattumalli ja suomenkielinen :)
 *
 * Käyttää C:n rajapintaa (ei "sinänsä" käyttöjärjestelmän, mutta
 * POSIX sisältää C99 -rajapinnan, joten fopen() on toki käytettävissä
 * POSIX-järjestelmissä).
 */
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[]){
    if (argc < 2){
        fprintf(stderr, "Ei katenoitavaa.\n");
        fprintf(stderr, "Käyttö: %s TIEDOSTONIMI [TIEDOSTONIMI]\n");
        exit(1);
    }

    for (int i=1; i<argc; i++){
        FILE *virta = fopen(argv[i], "r");
        if (virta == NULL){
            perror("Ei saatu auki");
            fprintf(stderr, "Skipataan %s\n",argv[i]);
            continue;
        }

        while (1){
            int byte = fgetc(virta);
            if (byte==EOF){
                if (feof(virta)){ break; /*OK: tiedoston loppu tavoitettu*/
                else {perror("Lukuvirhe"); continue; /*Muu häikkä*/ }
            }
            if (fputc(byte, stdout) == EOF){
                perror("Kirjoitusvirhe");
            }
        }
        fclose(virta);
    }
```

```
    return 0;
}
```

Kirjallisuutta

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

[2] Remzi H. Arpaci-Dusseau ja Andrea C. Arpaci-Dusseau, 2016. Operating Systems: Three Easy Pieces. Ilmainen on-line oppikirja, saatavilla WWW:ssä osoitteessa `http://pages.cs.wisc.edu/~remzi/OSTEP/` (linkin toimivuus tarkistettu 22.4.2016)

[3] 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 22.4.2014)

[4] AMD64 Architecture Programmer's Manual Vol 1–5 `http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/` (linkin toimivuus tarkistettu 22.4.2014)

[5] GAS assembler syntax. `https://sourceware.org/binutils/docs/as/Syntax.html` (linkin toimivuus tarkistettu 22.4.2014)

[6] Michael Matz, Jan Hubička, Andreas Jaeger, Mark Mitchell (eds.). System V Application Binary Interface – AMD64

Architecture Processor Supplement (Draft Version 0.99.6) October 7, 2013. `http://www.x86-64.org/documentation/abi-0.99.pdf` (linkin toimivuus tarkistettu 7.4.2015)

[7] John R. Levine, 1999. Linkers and Loaders. Morgan-Kauffman. (Vapaasti saatavilla oleva käsikirjoitus: `http://www.iecc.com/linker/`)

[8] Filippo Valsorda: Searchable Linux Syscall Table for x86 and x86_64 `https://filippo.io/linux-syscall-table/` (linkin toimivuus tarkistettu 7.4.2015)

[9] Yurcik, W.; Osborne, H. A crowd of Little Man Computers: visual computer simulator teaching tools, Proceedings of the Winter Simulation Conference, 2001.

[10] Peter J. Denning, 2005. The Locality Principle, Communications of the ACM, July 2005 / Vol. 48, No. 7 Pages 19-24.

[11] Dennis M. Richie; Ken Thompson. The UNIX time-sharing system, Communications of the ACM, July 1974 / Vol. 17, No. 7 Pages 365-375. (Jälkikäteen tehty PDF-vedos saatavilla myös `http://www.cs.berkeley.edu/~brewer/cs262/unix.pdf` - linkin toimivuus tarkistettu 20.4.2015)

Hakemisto

"swap thrashing", 251
16(%rbp), 138

ABI, 141
absolute file name, 65
absoluuttinen tiedostonimi, 65
abstract data structure, 32
abstrakti tietorakenne, 32
accessed-bitti, 238
activation record, 136
adapter, 259
address translation, 233
aikajakojärjestelmiä, 151
aikataulutuksen, 148
ajaa, 71
ajonaikaisella kääntämisellä, 86
ajurirajapintaa, 264
aktivaatiotietue, 136
AL, 93
aliohjelma, 131
Aliohjelmakutsu, 47
aliohjelmat, 114
ALU, Arithmetic Logic Unit, 43
Application Binary Interface, 141
aritmeettislooginen yksikkö, 43
array, 29
assembler, 95
assembleria, 95
assemblerilla, 77, 95
assembly language, 77, 95
assemblyä, 95
atomic operation, 217
atomisesti, 217

base plus offset, 121
base pointer, 137
batch processing, 148
binary digit, bit, 35
bittejä, 35
block, 259
block device, 259
booting, 52
bootstrapping, 52
BP, 137–140, 145
bus, 40
byte, 41

cache line, 242
cache memory, 57
cache thrashing, 244
CALL, 139, 140, 145
calling convention, 141
carry flag, 48
CF, 48
channel, 259
character device, 259
class, 24
code, 120
compile, 71
compiled programming language, 86
context, 180
context switch, 170, 180
control flow, 115
control unit, 43
core, 56
CPU, Central Processing Unit, 40
critical region, 209
cycle stealing, 261
cylinder, 266

data, 120
data structure, 25
dataa, 120
deadline scheduling, 303
deadlock, 213
debugger, 75
debuggeriksi, 75
deskriptorien, 274
deskriptoritaulu, 286
desktop manager, 52
destination, 96
device controller, 259
device driver, 260
digital computer, 35
Direct memory access, 261
dirty, 243
dirty-bitti, 238
disassembler, 118
disassembly, 89, 97
disk scheduling, 299
dispatcher, 186
DIV, 111
DMA, 261
double word, 42
dynaamisesta linkittämisestä, 81
dynaamista muistinvarausta, 121
dynamic linking, 81
dynamic memory allocation, 121
dynamically linked library, DLL, 81

EAX, 93
Ehdollinen hyppykäsky, 47
Ehdoton hyppykäsky, 47
ehtorakenteet, 114
ENTER, 135, 140, 143
enter, 134
entry point, 83
eräajosta, 148
etäaliohjelmakutsu, 201
execute, 46

execute-bitti, 238
execution stack, 107
extends, 26

fairness, 160
fetch, 45
fetch-decode-execute, 45
fetch-execute cycle, 45
FIFO, 274
file descriptor, 274
file system, 70
file table, 288
first in - first out, 274
First-level interrupt handling, 167
flag register, 46
FLAGS, 46, 48–50
FLG, 46
FLIH, 167
FPR0, 92, 93
FPR0-FPR7, 93
FPR7, 92, 93
FR, 46
frame table, 249
function, 131
funktio, 131
fyysisen osoiteavaruuden, 41

general purpose registers, 49
graafisista kuoriohjelmista, 65
graphical shell, 65

Hakuaikaan, 267
hard link, 283
head, 32
heap, 123
heittovaihto, 245
heittovaihtotila, 246
heksaluvuista, 37

i-node, 289
i-number, 289
i-numero, 289
i-solmujen, 289
i-solmuun, 281
I/O-laitteita, 40
ikkunointijärjestelmästä, 52
implements, 26
indirect addressing, 96
inheritance, 26
inline assembly, 99
inode, index node, 281
Input/Output modules, 40
INSTR, 46, 49
instruction, 45
instruction pointer, 45
instruction register, 46
instruction set, 55
integer, 23
integer overflow, 39
Inter-process communication, IP 200
interface, 12
interpreted languages, 87
interrupt, 151
interrupt handling, 167
interrupt request, IRQ, 166

interrupt signal, 166
interrupt vector, 168
IP, 45–47, 49, 50, 83, 121, 126, 127, 133, 139, 140, 145
IR, 46
ISA, instruction set architecture, 55

järjestelmäkäskyillä, 50
järjestelmäkutsurajapintaan, 99
jaetut muistialueet, 201
job, 148
Jokin peräkkäissuoritteinen käsky, 47
Jono, 33
journaling file system, 295
journalointi, 295
just-in-time compilation, JIT, 86
juurisolmu, 34

käännettävä ohjelmointikieli, 86
kääntää, 71
käskykanta, 55
käskykanta-arkkitehtuuri, 55
käskyn, 45
käskyosoitin, 45
käskysymboli, 95
käyttäjälle näkyviä rekisterejä, 54
käyttäjätilassa, 53, 150
käyttöaste, 160
käyttöjärjestelmän kutsurajapinta, 172
käyttöjärjestelmätilaan, 51
kanavat, 259
kantaosoitin, 137
kantarekisterin, 121
kehyksiin, 232
kehystaulu, 249
kekomuistissa, 123
keosta, 24, 144
kernel, 13, 51
kernel mode, 51
kernel stack, 168
Kernel-pino, 168
keskeytykset, 151
Keskeytyskäsittely, 167
keskeytyspulssille, 166
keskeytyspyyntö, 166
keskeytyssignaali, 166
keskeytysvektori, 168
keskinäinen poissulku, 209
keskusyksikkö, 40
kevyeksi prosessiksi, 195
kiertojono, 186
kiertovuorottelumenetelmä, 186
kilpailutilanteesta, 207
kilpajuoksusta, 207
kohde, 96
kohdekoodiksi, 70
Konekieli, 45
Konteksti, 180
kontekstin vaihto, 170, 180

kontrollilogiikka, 259
kontrollin siirtyminen, 115
kontrolliyksikkö, 43
koodi, 120
Kova linkki, 283
kovalevyn vuoronnus, 299
kriittiseksi alueeksi, 208
kuoren, 13, 52
Kuori, 65
kutsumalli, 141
kutsurajapinta, 176
kutsusopimus, 141

lähde, 96
lähdekoodi, 70
läpimenoaika, 161
laiteajuri, 260
laiteohjaimet, 259
laiteriippuva ohjelmiston osa, 260
laitteistoriippumaton I/O-ohjelmisto, 261
laskennan teoria, 56
last, 32
lataaja, 81
LEA, 106
lea 32(%rbp), %rdx, 106
leaf node, 34
LEAVE, 135, 140, 143
lehtisolmuja, 34
light-weight process, 195
liittäminen, 277
likainen, 243
link, 282
linking, 80
linkittäminen, 80
linkkien, 282
lippurekisterissä, 46
list, 32
Lista, 32
little man computer, LMC, 331
loader, 81
local variables, 120
logic gate, 40
logiikkaportteja, 40
lohko, 259
lohkolaitteisiin, 259
lokaalisuusperiaate, 57
LRU, least-recently-used, 250
lukkiutumisesta, 213
luokkaan, 24

machine language, 45
mailbox, 201
mass memory, 58
massamuistia, 58
massively parallel computing, 194
memory, 40
memory address, 23
memory cell, 40
memory hierarchy, 58
memory management, 229
memory management unit, MMU, 231
merkkilaitteisiin, 259
message, 201

method, 24, 131
metodeja, 24
metodi, 131
metodit, 114
micro code, 88
microkernel operating system, 53
mikrokoodia, 88
mikroydinkäyttöjärjestelmäksi, 53
MMX0, 92, 93
MMX7, 92, 93
moniajo, 151
moniohjelmointi, 151
monoliittinen käyttöjärjestelmä, 53
monolithic operating system, 53
motherboard, 40
mounting, 277
MOV, 96, 105
movq, 95
movq (%rdx), %rax, 106
movq 32(%rbp), %rax, 106
movq %rsp,%rbp, 104
muisti, 40
muistihierarkiasta, 58
muistinhallinta, 229
muistiosoite, 23
muistisoluja, 40
MUL, 111
multicore processor, 56
multiprogramming, 151
multitasking, 151
mutual exclusion, "MutEx", 209
MXCSR, 92

nälkiintyminen, 213
native code, 86
NEG, 113
negative flag, 48
network attached storage, NAS, 271
NF, 48
nimetyissä muuttujissa, 121
node, 34
northbridge, 40
nouto, 45
nouto-suoritus -sykli, 44

object, 24
object code, 71
objekti, olio, 132
odotusaika, 161
OF, 48
ohjelmalaskuri, 45
ohjelmallinen keskeytyspyyntö, 172
ohjelmoijalle näkymättömästä rekisteristä, 49
ohjelmoijalle näkyvistä rekistereistä, 49
oktaalilukuja, 38
olioihin, 24
opcode, 55
operaatiokoodin, 55

Operaatiotutkimus, 161
operandeina, 96
operandit, 46
operation code, 55
operations research, OR, 161
osoitin, 29
osoitteenmuunnoksen, 233
overflow, 48

päätteet, 152
page, 232
page fault, 239
page fault exception, 250
page frame, 232
page replacement algorithm, 250
page table, 234
page table entry, PTE, 237
paging, 234
paikallisia muuttujia, 120
Paluu aliohjelmasta, 47
parallel, 194
PC, 45
PC, personal computer, 154
peräkkäisjärjestyksessä, 113
peräkkäistä, 148
physical address space, 41
Pino, 33
pino, 120
pino-osoitin, 107
pinokehystä, 136
pinon huipun osoitin, 107
pipe, 201
poikkeukset, 115
pointer, 29
POP, 138
pop, 107, 110, 118
port, 201
portti, 201
postilaatikko, 201
pre-emptiivisyys, 302
preemption/pre-emption, 302
present-bitti, 239
principle of locality, 57, 152
procedure, 131
process, 152, 165, 180
process control block, PCB, 184
process table, 184
processor, 40
producer-consumer problem, 220
program counter, 45
Program status word, PSW, 46
proseduuri, 131
prosessi, 165, 180
prosessielementtejä, 184
prosessien väliseksi kommunikoinniksi, 200
prosessin, 152
Prosessitaulu, 184
prosessori, 40
prosessoriarkkitehtuuri, 55
protected mode, 53
PUSH, 138
push, 107, 110, 118
putket, 201
Puu, 34

quadword, 42
quantum bit, qubit, 158
queue, 33

R10, 92
R11, 92
R15, 92
R8, 92
R9, 92
race condition, 207
RAID, 269
rajapinnat, 12
rajapinta, 12
RAM-muistin, 42
Random Access Memory, 42
RAX, 92, 93, 208
RBP, 92, 96
rbp, 96
RBX, 92
RCX, 92
RDI, 92
RDX, 92
Reaaliaikajärjestelmä, 300
Read-Only Memory, 42
real mode, 51
real time system, 300
redundant array of inexpensive/independent disks, 269
reference, 24
register indirect addressing, 121
registers, 43
rekisterit, 43
relative file name, 65
remote procedure call, RPC, 201
response time, 161
RET, 140
reverse mapping, 248
RFLAGS, 92, 111, 118
rinnakkaiseksi, 194
RIP, 83, 92, 118, 133
ROL, 113
ROM-muistin, 42
root node, 34
ROR, 113
round robin, 186
RSI, 92
RSP, 92, 96, 107, 108, 118
rsp, 96
run, 71

säikeet, 193
SAL, 113
sanalle, 42
sananpituus, 42
SAR, 113
scheduler, 186
scheduling, 148
sector, 266
seek time, 267
segmentoidusta muistista, 126
Segmenttirekisterit, 127
sektori, 266
Semafori, 214
semaphore, 214
serial processing, 148
shared memory, 201

shared object, "so", 81
shell, 13, 52, 65
shell-skripti, 312
signaalit, 200
signal, 200
sisäänmenopisteeksi, 83
sivuihin, 232
sivunkorvausalgoritmilla, 250
sivunvaihtokeskeytys, 250
sivutaulu, 234
sivutaulumerkintä, 237
sivuttamisen, 234
sivuvirhe, 250
sivuvirhettä, 239
skripti, 312
software stack, 12
solmuiksi, 34
source, 96
source code, 70
southbridge, 40
sovittimet, 259
SP, 107, 121, 126, 127, 135, 137–140, 145
stack, 33
stack frame, 136
stack pointer, 107
starvation, 213
storage system, 271
stream, 275
string literal, 21
stripe, 269
subprogram / subroutine, 131
suhteellinen tiedostonimi, 65
suojattu tila, 53
suoritin, 40
suoritus, 46
suorituspinon, 107
supervisor mode, 51, 150
swap, 250
swap space, 246
swapping, 245
syöttö- ja tulostuslaitteita, 40
sylinteri, 266
symbolic link, 283
Symbolinen linkki, 283
symbolisella konekielellä, 77
symbolista konekieltä, 95
symmetric multiprocessing, 56
synchronization, 206
synkronointi, 206
syscall, 103, 104
system call interface, 99, 172, 176
system instructions, 50
system registers, 50
System V, 141

tail, 32
takaisinkäännös, 97
tallennusjärjestelmät, 271
tasapuolisuus, 160
taulukoiden, 29
tavuina, 39
tavujen, 41
terminal, 152

text, 120
thread, 193
throughput, 161
tiedostojärjestelmä, 70
tiedostotauluun, 288
tietokone, 35
tietorakennetta, 25
tietovirroilla, 275
time-sharing systems, 151
TLB, 233
toistorakenteet, 114
track, 266
tree, 34
tulkattavia kieliä, 87
Tuottaja-kuluttaja -ongelma, 220
tuottavuus, 161
turnaround, 161
työ, 148
työjoukoksi, 246
työpöytä, 52

unlinking, 282
ura, 266
user mode, 53, 150
user visible registers, 54
user-bitti, 238
utilization, 160

välimuisteja, 57
välimuistin ruuhkautuminen, 244
välimuistirivi, 242
väylä, 40
vahtikoira, 304
valvontatilassa, 150
vasteaika, 161
verkkolevyiksi, 271
viestit, 201
viitteitä, 24
viittemuuttujia, 124
virtuaalimuistiavaruus, 229
virtuaaliosoitteita, 126
virtuaalisen osoiteavaruuden, 42
visible registers, 49
vuoronnuksen, 148
vuorontaja, 186

waiting time, 161
watch dog, 304
window manager, 65
windowing system, 52
word, 42
word length, 42
working set, 246
write-bitti, 238

XMM0, 92, 93
XMM15, 92, 93

ydin, 13
ydintilaan, 51
yleiskäyttöisiä rekisterejä, 49
YMM0, 92
YMM15, 92

zero flag, 48
ZF, 48

Kuvat

0.1 Kerrokset ja rajapinnat käyttäjän ja tietokonelaitteiston välillä (yksinkertaistus). 11
0.2 Käyttöjärjestelmien suhde muihin kursseihin. Ohjelmointi 1 on välttämätön esitieto. Muut mainitut auttavat, mutta niiden sisällöstä sovelletaan vain joitain osia. . . 16
0.3 Kiinteänmittaisia taulukoita sisältöineen 30
0.4 Lista, jono ja pino periaatepiirroksina. 60
0.5 Reaalimaailman puurakenne periaatepiirroksena 61
0.6 Puurakenne abstraktina tietorakenteena 61
0.7 Yleiskuva tietokoneen komponenteista: keskusyksikkö, muisti, I/O -laitteet, väylä. 62
0.8 Yksinkertaistettu kuva kuvitteellisesta keskusyksiköstä: kontrolliyksikkö, ALU, rekisterit, ulkoinen väylä ja sisäiset väylät. Kuva mukailee Tietotekniikan perusteet -luentomonistetta [3], jossa määritellään myös käskykanta-arkkitehtuuri vastaavalle pikkuprosessorille. 62
0.9 Yksinkertaistettu kuva modernimman tietokoneen komponenteista: useita rinnakkaisia keskusyksiköitä, keskusmuisti, välimuistit, I/O -laitteet, väylä. 63

0.10 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, 'L'). Jako johtuu x86-sarjan historiallisesta kehityksestä 16-bittisestä 64-bittiseksi ja taaksepäin-yhteensopivuuden säilyttämisestä. 93
0.11 Assembler-rivin rakenne, jossa näkyy osoite, konekielinen tavuesitys ja assembler-käsky GNU Assemblerin mukaisena. 98
0.12 Suorituspino: muistialue, jota voidaan käyttää push- ja pop-käskyillä. SP osoittaa aina pinon "huippuun", joka "kasvaa" muistiosoitteen mielessä alaspäin. 109
0.13 Tyypilliset ohjelman suorituksessa tarvittavat muistialueet: koodi, data, pino, dynaamiset alueet. (periaatekuva, joka täydentyy myöhemmin) 120
0.14 Esimerkki segmenttirekisterien käytöstä (esim. 80286-prosessori): 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. 126
0.15 Virtuaalinen muistiavaruus x86-64:ssä tietyn ABI-sopimuksen mukaan. (XXX: Kuvassa väärät heksat, pitäisi olla 48-bit osoitteet, ei 56-bit.) 128
0.16 Geneerinen esimerkki ohjelman näkemän virtuaaliosoiteavaruuden ja tietokonelaitteiston käsittelemän fyysisen osoiteavaruuden välisestä yhteydestä. 130
0.17 Perinteinen pinokehys, ja kuinka se luodaan: eri vaiheet, osallistuvat ohjelman osat sekä "pseudo-assembler-koodi". 137
0.18 Pinon käyttö x86-64:ssä kuten SVR4 AMD64 supplement sen määrittelee. 142

0.19 Prosessorin suoritussykli: nouto, suoritus, tilan päivittyminen, mahdollinen keskeytyksenhoitoon siirtyminen. 167
0.20 Prosessin tilat (geneerinen esimerkki). 182
0.21 Käyttöjärjestelmän keskeisimmät tietorakenteet: prosessielementti ja sellaisista koostuva prosessitaulukko. . 184
0.22 Prosessien vuorontaminen kiertojonolla (round robin). Prosesseja siirretään kiertojonoon ja sieltä pois sen mukaan, täytyykö niiden jäädä odottelemaan jotakin tapahtumaa (eri jonoon, blocked-tilaan). 187
0.23 Uuden prosessin luonti fork()-käyttöjärjestelmäkutsulla. Uuden ohjelman lataaminen ja käynnistys edellyttää lisäksi lapsiprosessin sisällön korvaamista käyttöjärjestelmäkutsulla exec(). 189
0.24 Voidaan ajatella että säikeet (yksi tai useampia) sisältyvät prosessiin. Prosessi määrittelee koodin ja resurssit; säikeet määrittelevät yhden tai useampia rinnakkaisia suorituskohtia. 196
0.25 Muistialueita voidaan jakaa prosessien välillä niiden omasta pyynnöstä tai oletusarvoisesti (käyttöjärjestelmän alueet sekä dynaamisesti linkitettävät, jaetut kirjastot). 205
0.26 Sivuttavan virtuaalimuistin perusidea ja tietorakenteet. 232
0.27 Leikkiesimerkki prosessin sivutaulusta 20-bittisellä virtuaaliosoiteavaruudella, 24-bittisellä fyysisellä osoiteavaruudella ja 4096 tavun sivukoolla. 235
0.28 Leikkiesimerkki prosessin virtuaalimuistiosoitteesta 20-bittisellä virtuaaliosoiteavaruudella ja 4096 tavun sivukoolla. 236
0.29 Nelitasoinen osoitteenmuunnos AMD64-prosessorissa (alkuperäinen x86-64). Arkkitehtuuri tukee muutamaa isompaa sivukokoa, mutta tässä on esimerkki tyypillisimmästä tapauksesta eli 4096 tavun ($= 2^{12}$) kokoisten sivujen käytöstä. 240

0.30 Sivutaulujen ja kehystaulun käyttöä: leluesimerkki ja perinteinen tenttitärppi. 257
0.31 I/O -operaatioon osallistuvat kerrokset, niiden väliset rajapinnat, ja operaation suoritusvaiheet ja osapuolet aikajärjestyksessä (1–7). 263
0.32 Tiedoston käyttöoikeuksien määrittely POSIXin chmod() -kutsulla tai shell-komennolla chmod. 284
0.33 Käyttöjärjestelmän tietorakenteita tiedostojärjestelmän toteuttamiseksi. 287