Luentomuistiinpanoja

Tässä on luennolla käsiteltyjä asioita, jotka eivät ole muissa materiaaleissa ainakaan aivan samoilla sanoilla esitettynä. Osa lienee vähintään yhtä hyvin varsinaisessa luentomonisteessa.

Sisältö

1 Prosessista

Vedetään tässä jollain tapaa yhteen, mitä tarkoittaa **prosessi**. Laajennetaan säieohjelmointiin myöhemmin.

1.1 Konteksti (context)

Konteksti (*context*) on prosessorin tila, kun se suorittaa ohjelmaa. Prosessorin tila taas tietysti määräytyy sähköjännitteistä komponenteissa, erityisesti rekistereissä (mm. IP, PSW, datarekisterit, osoiterekisterit). Käyttäjän prosessin kontekstin osalta puhumme tietenkin käyttäjän näkemistä rekistereistä, emme systeemirekistereistä, joihin voi vuorovaikuttaa vain prosessorin ollessa käyttöjärjestelmätilassa. Huomioitavaa:

- jokaisella prosessilla on oma kontekstinsa.
- vain yhden prosessin konteksti on muuttuvassa tilassa yksiprosessorijärjestelmässä (kaksiprosessorijärjestelmässä kahden prosessin kontekstit, jne...)
- muiden prosessien kontekstit ovat "jäädytettynä" jemmassa (käyttöjärjestelmä pitää niitä tallessa, kunnes prosessi saa ajovuoron, ja konteksti siirretään rekistereihin).

Kontekstin vaihto (*context switch*) on häilyvä termi. Konteksti on esim. meidän luentomonisteemme mukaan prosessorin tila. Jotkut lähteet (kuulemma) laventavat kontekstin olemaan myös muuta prosessiin liittyvää tietoa.

Sanokaamme mieluummin **prosessin vaihto**, kun puhutaan siitä, että prosesseja vaihdellaan perätysten. Yksi osa tätä on se, että prosessori saa "pureskeltavakseen" eri prosesseja, ja sen "työympäristö" eli laitteiston suorituksen "konteksti" vaihtuu samalla kun prosessista toiseen vaihdetaan. Kontekstin vaihto on siis tämän saivartelun mukaan laitteiston kokema toimintatilan vaihdos. Prosessin vaihto kuitenkin sisältää prosessorin kontekstin vaihdon lisäksi myös käyttöjärjestelmän suorittamaa kirjanpitoa keskusmuistissa ja levyllä sijaitsevissa tietorakenteissa, mikä on siis prosessorin kontekstista (käyttäjän näkemien rekisterien sisällöistä) erillinen asia. Puhhuh.

Prosessin vaihto (*process switch*) on tätä:

- yhden prosessin tilan tallentaminen prosessorista jonnekin muistiin. (FLIH tapahtuu laitteistotasolla, käyttöjärjestelmä hoitaa siitä eteenpäin; ks. lisämateriaalista kohta, joka kertoo keskeytyskäsittelystä)
- jonkun toisen prosessin tilan palauttaminen prosessoriin (käyttöjärjestelmä hoitaa, laitteistotasolla tapahtuu keskeytyskäsittelijästä palaaminen eli "return from interrupt")
  - Prosessi itse ei huomaa vaihtoa
  - Koko prosessorin tila (siten kuin käyttäjän ohjelma voi sen nähdä == "user visible registers") on täysin sama kuin silloin kun prosessi joskus aiemmin keskeytettiin. Toki käyttöjärjestelmän näkemä, käyttäjän prosessilta piilossa oleva osa prosessoria (== "system registers") voi olla muuttunut sen mukaan, mitä käyttäjärjestelmä teki prosessin ollessa keskeytettynä.
  - prosessia ei välttämättä tarvitse vaihtaa joka keskeytyksellä; se riippuu käyttöjärjestelmän vuorontajaan valituista algoritmeista. Yleensä mm. kellokeskeytys moniajojärjestelmässä tarkoittaa nykyisen prosessin aikaviipaleen loppumista, ja ainakin silloin vaihdetaan prosessia. Toisaalta joku I/O, vaikkapa näppäinpainallus,

voidaan lyhyesti kirjata käyttöjärjestelmän sisäiseen puskuriin odottamaan myöhempää käsittelyä, ja keskeytynyttä prosessia voidaan jatkaa ilman mitään vaihdosta aina aika-askeleen loppuun tai muuhun luonnolliseen vaihdokseen saakka.

Prosessin vaihtoon liittyy kontekstin vaihdon lisäksi myös kirjanpitoa muista tilatiedoista. Käyttöjärjestelmä hoitaa tämän kaiken.

1.2 Prosessitaulu

Prosessitaulu on yksi käyttöjärjestelmän ylläpitämä tietorakenne. Melkeinpä kaikki käyttöjärjestelmän osat käyttävät prosessitaulun tietoja, koska siellä on kunkin käynnistetyn ohjelman suorittamiseen liittyvät asiat ja juuri ohjelmien sujuva suorittaminen on loppujen lopuksi koko käyttöjärjestelmän ainoa tehtävä.

Prosessitaulussa on useita ns. **prosessielementtejä**, esim.:

```
PROSESSITAULU:

Process Control Block, PCB (prosessielementti, yksi per prosessi)
PID #1

Process Control Block, PCB (prosessielementti, yksi per prosessi)
PID #2

Process Control Block, PCB (prosessielementti, yksi per prosessi)
PID #3

...
... PCB:itä on monta -- jokaisella käynnistetyllä prosessilla yksi
...

Process Control Block, PCB (prosessielementti, yksi per prosessi)
PID #21189
```

Prosessielementtien maksimimäärä on rajoitettu -- esim. positiivisten, etumerkillisten, 16-bittisten kokonaislukujen määrä, eli 32767 kpl. Enempää ei pystyisi prosesseja luomaan. Esim. tuollainen määrä kuitenkin on jo aika riittävä. Esim. 3000 käyttäjää voisi käyttää yli kymmentä ohjelmaa yhtä aikaa. Ennemmin kuin prosessielementit, loppuu luultavasti prosessoriteho tai muisti. (Demon 3 esimerkkiohjelmissa kokeillaan muistinhallinnan ja prosessitaulun rajoja).

Yhden PCB:n sisältö:

- prosessin yksilöivä tunnus eli prosessi-ID, “PID”. Voi olla toteutuksen kannalta PCB:n indeksi prosessitaulussa.
- konteksti eli prosessorin “user-visible registers” sillä hetkellä kun viimeksi tuli keskeytys, joka johti tämän prosessin vaihtamiseen pois Running-tilasta.
- PPID (parent eli vanhempiprosessin PID)
- voi olla PID:t myös lapsiprosesseista ja sisaruksista

- UID, GID (käyttäjän & ryhmän ID:t; tarvitaan käyttöjärjestelmän vastuulla olevissa tietosuojatarkistuksissa)
- prosessin tila (ready/blocked/jne...) ja prioriteetti
- resurssit
  - tälle prosessille avatut/lukitut tiedostot (voivat olla esim. ns. deskriptoreita, eli indeksejä taulukkoon, jossa on lisää tietoa käsiteltävänä olevista tiedostoista)
  - muistialueet (koodi, data, pino, dynaamiset alueeet)
- viestit muilta prosesseilta, mm.
  - Sanomanvälitysjono
  - Signaalijono
  - putket

1.3 Prosessin tilat

Prosessin toimintatilat on esitetty luentomonisteessa; siellä on myös kuvia. Tässä on lisukkeeksi tekstimuotoinen, ajallisesti etenevä, esimerkki tilanvaihdoksista:

```
alkutilanne:
  PID 24: User Running.
  PID 343: Blocked
  Prosessori suorittaa prosessin 24 koodia.

tapahtuma:
  keskeytys/KJ-kutsu

tilanne:
  PID 24: Kernel Running
  Prosessori suorittaa käyttöjärjestelmän koodia.

  Esim. tuli merkki päätteeltä prosessille PID 343
      -> KJ laittaa merkin tiedoksi prosessille PID 343 ja siirtää sen
         Ready-tilaan (oli Blocked, koska odotti merkkiä)
         ja jatkaa PID 24:n suorittamista.

tilanne:
  PID 24: User Running.
  PID 343: Ready
  Prosessori suorittaa yhä prosessin 24 koodia.

  (343 voi jatkua sitten joskus ja saa sitten odottamansa merkin
  paluuarvona käyttöjärjestelmäkutsulta.)
```

TAI (toinen tapa toteuttaa, tapoja on paljon, ne riippuvat käyttöjärjestelmän toteutusvalinnoista; eri menettelyt soveltuvat paremmin joihinkin tietotekniikan käyttötarkoituksiin ja huonommin joihinkin):

```
alkutilanne (sama kuin edellisen esimerkin alussa):
  PID 24: User Running.
```

```
  PID 343: Blocked
  Prosessori suorittaa prosessin 24 koodia.

tapahtuma:
  keskeytys/KJ-kutsu

tilanne:
  PID 24: Kernel Running
  Prosessori suorittaa käyttöjärjestelmän koodia.

  Esim. tuli merkki päätteeltä prosessille PID 343
      -> KJ laittaa PID 24:n Ready-tilaan
      -> KJ laittaa merkin tiedoksi prosessille PID 343 ja siirtää sen
         Running-tilaan (oli Blocked, koska odotti merkkiä)
         ja jatkaa siis siitä.

tilanne:
  PID 24: Ready
  PID 343: User Running
  Prosessori suorittaa prosessin 343 koodia.
```

1.4 Vuorontamismenettelyt, prioriteetit

Tämäkin on luentomonisteessa selitetty. Tässä jonkinlainen esimerkki. Esim. voisi olla ready-jono (pysähtyneinä, mutta valmiina suoritukseen, kun aikaviipale olisi tarjolla):

```
[READY]   ->   PID 24 -> PID 7 -> PID 1234 -> null
```

Blocked-jono (pysähtyneinä, odottavat esim. I/O:ta):

```
[BLOCKED] ->   PID 9139 -> PID 45 -> PID 343 -> null
```

“Round robin” -menettelyssä valitaan aina jonon kärjestä kauiten odotellut prosessi suoritukseen. “Ohiajot” eivät ole mahdollisia round robinissa, joten se ei sovellu reaaliaikajärjestelmiin, kuten musiikkiohjelmien suorittamiseen. Millisekunninkin katkos audion tuotannossa kuulostaa poksahdukselta; audio-ohjelman pitäisi pystyä “etuilemaan” jonossa, jos sen odottama keskeytys saapuu, tai aina korkeintaan tietyn aikavälin kuluttua. Samoin muiden kriittisten reaaliaikasysteemien, kuten rakettimoottorin tai ydinvoimalan ohjaus mittaritiedon perusteella. Tarvitaan siis “pre-emption”, eli aikaviipaleen keskellä tapahtuva nykyisen prosessin keskeytys ja prosessin vaihto tärkeämpään -- ja jokin toteutus prioriteettien hallintaan.

Esimerkki prioriteeteista, monta Ready-jonoa (pysähtyneinä, mutta valmiina suoritukseen):

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

Esim. suositaan pienemmän prioriteettitason jonoja vuoronnuksessa. Prosesseja voitaisiin myös tietyin perustein siirtää prioriteettitasojen välillä, eli prioriteetit voisivat olla dynaamisia. Joka tapauksessa tulee välttää tilanne, jossa joku prosessi jäisi ikuisesti saamatta yhtään aikaa. Reaaliaikaprioriteetin prosessit otetaan jonojen ohi välittömästi, kun esim. niihin liittyvää I/O:ta havaitaan. Edelleen reaaliaikaohjelmatkin räjähtävät käsiin, jos niitä yritetään suorittaa liian paljon. Reaaliaikaoperaatiolla voi olla esim. välttämätön alkuaika, jota myöhemmin se ei saisi joutua alkamaan, ja sillä voi olla myös välttämätön loppuaika, jota aiemmin sen tulisi saada operaatio valmiiksi. Liian paljon reaaliaikaohjelmia voi aiheuttaa sen, että joku niistä missaa joko aloitus- tai lopetusdeadlinen, vaikka ei-reaaliaikaohjelmille olisi käytettävissä paljonkin prosessoriaikaa.

Jonotusongelmat ovat vaikeita hallita, ja ne edellyttävät kompromissiratkaisuja. Laajemmin mm. jonotussääntöjen kaltaisia kysymyksiä tutkii tieteenala nimeltä operaatiotutkimus (OR, "operations research"). Käyttöjärjestelmän vuorontaminen on yksi sovellusala, jossa jotkut teoreettiset mallit voivat olla samankaltaisia kuin esim. pilvenpiirtäjän hissien ruuhka-aikojen mallinnuksessa käytettävät. Ja toisin päin.

1.5 Prosessin luonti fork():lla

Käyttöjärjestelmäkutsu fork() on ainoa tapa, jolla perus-Unixissa voi kukaan tehdä uuden prosessin.

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

Käyttöjärjestelmä luo fork()-kutsua käsitellessään hiukan muutetun kopion nykyisestä prosessista (joka pyysi forkkausta eli haaroitusta):

```
Uuden PCB:n sisältö:

Nämä tulee uusiksi uudelle haaroitetulle prosessille:
   - prosessin yksilöivä tunnus eli prosessi-ID, "PID"
   - PPID (parent eli vanhempiprosessin ID) := forkin kutsujan PID

Tämä on melkein sama:
   - prosessin konteksti (pysyy samana fork() -kutsun paluuarvoa
     lukuunottamatta!)

Nämä kopioituvat identtisinä:
   - UID, GID (käyttäjän & ryhmän ID:t)

   - resurssit

        + tiedostot

        + muistialueet
          (koodi, data, pino, dynaamiset alueeet)

   - viestit muilta prosesseilta

        + Sanomanvälitysjono
```

```
+ Signaalijono
```

Forkin käyttö on esitelty luentomonisteessa esimerkin ja kuvien kera, ja siitä on saatavilla esimerkkikoodi minish.c kurssin nettisivulta. Demon 3 koodeissa käytetään erityisen paljon fork() -kutsua.

2 Säikeet

Yhdenaikainen suorittaminen on hyvä tapa toteuttaa käyttäjäystävällisiä ja loogisesti hyvin jäsenneltyjä ohjelmia. Sovellusohjelman jako useisiin prosesseihin olisi yksi tapa, mutta se on tehottomampaa, esim. muistinkäytön kannalta raskasta, ja monilta osiltaan tarpeettoman monipuolista. Ratkaisu ovat säikeet, eli yhden prosessin suorittaminen yhdenaikaisesti useasta eri paikasta.

2.1 Yhden säikeen tarpeet

Prosessi on siis "muistiin ladatun ja käynnistetyn 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 edellyttää useampaa eri kontekstia, joita vuoronnetaan sopivasti. Tällaisen nimeksi on muodostunut **säie** (*thread*). Yhdellä prosessilla on yksi säie tai useampia säikeitä.

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

Säikeellä on oma:

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

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

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

Säie mahdollistaa moniajon yhden prosessin sisällä tehokkaammin kuin että olisi lapsiprosesseja, jotka kommunikoisivat keskenään.

Toteutustapoja:

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

KLT-toteutuksessa käyttöjärjestelmällä voisi esimerkiksi olla tallessa PCB:n lisäksi TCB-tiedot (Thread Control Block) seuraavalla tavoin:

PCB(prosessielementti):

```
TCB1:
  - säikeen 1 konteksti
  - säikeen 1 pinoalue
  - säikeen 1 ylläpitotiedot

TCB2:
  - säikeen 2 konteksti
  - säikeen 2 pinoalue
  - säikeen 2 ylläpitotiedot

ja niin edelleen... jokaiselle säikeelle eli suorituskohdalle olisi
oma TCB.

... ja sitten PCB:n muu sisältö, joka on kaikille säikeille yhteinen.
```

Sanoisiko TCB:tä suomeksi sitten "säie-elementiksi", jollaisia sisältyy prosessikokonaisuuden PCB:hen eli prosessielementtiin.

3 Prosessien synkronointi

Käytetään tässä esimerkkinä yksinkertaista tuottaja-kuluttaja -ongelmaa. Se on yksi perinteinen ongelma, joka voi syntyä käytännön sovelluksissa, ja jonka avulla voi testata synkronointimenetelmän toimivuutta:

- Yksi prosessi/säie tuottaa dataa elementti kerrallaan. Tämä voi olla hidas tai nopea toimenpide, ja dataelementin koko voi olla pieni tai suuri.

- Toinen prosessi/säie lukee ja käsittelee (="kuluttaa") tuotettua dataa elementti kerrallaan. Tämä voi olla hidas tai nopea toimenpide, erityisesti se voi olla paljon hitaampaa tai nopeampaa kuin tuottaminen.
- Tällä tavoin saavutetaan mm. modulaarisuutta ohjelmien tekemiseen, jakeluun ja suorittamiseen.
- 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 voi keskeytyä kun elementin kirjoittaminen on puolivalmis, ja myös kuluttaja voi keskeytyä kesken elementin lukemisen.

Tärkeätä olisi savuttaa seuraavat asiat:

- Puskurin täyttyessä pitää pystyä odottamaan, että tilaa vapautuu. Muutoin ei ole mahdollista kirjoittaa uutta tuotosta mihinkään. Tuottajan pitää pystyä odottamaan.
- Jos puolestaan puskurissa ei ole uutta dataa tuotettuna, kuluttajan pitää pystyä odottamaan, että sitä ilmaantuu.
- Puskurin sisällön pitää olla koko ajan järkevä (ei puolivalmista dataa) ja myös täytyy olla järkevät osoittimet eli muistiosoitteet paikkaan, jota kirjoitetaan ja jota luetaan.

Odottaminen on tärkeätä, samoin herääminen odotuksen jälkeen. Nähnet, että prosessin suorituksen kannalta odottaminen tarkoittaa käyttöjärjestelmän hoitamaa tilanvaihtoa Blocked-tilaan.

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.

"Kuva":

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

Reunahuomautuksena todetaan, että virtuaalimuistin hienous tässä kohtaa on, että osoitemielessä voi olla:

```
tALKU != kALKU
```

eli tuottaja ja kuluttaja voivat olla erillisiä prosesseja, jotka näkevät puskurin alkavan jostain kohtaa omaa virtuaalimuistiavaruuttaan, ja niillä on oma indeksi tällä hetkellä käsittelemäänsä elementtiin. MUTTA:

```
viitattu fyysinen muistiosoite fALKU on sama.
```

Muistinhallinnan yhteydessä tutustutaan tarkemmin ns. osoitteenmuodostukseen prosessin virtuaaliosoitteesta todelliseksi, joka menee prosessorista osoiteväylälle.

3.1 Termistöä

Kriittinen alue kohta ohjelmakoodissa, joka voi aiheuttaa ongelmatilanteen moniohjelmoinnissa (eli kun on useita prosesseja/säikeitä)

Poissulku Ohjelmointikeino, jolla estetään muita prosesseja/säikeitä suorittamasta kriittistä aluetta väärään aikaan (eli ennen kuin se on järkevää toimintalogiikan kannalta)

Toimii binäärisellä semaforilla.

Synkronointi Ohjelmointikeino, jolla ohjataan jaetun resurssin käyttöä niin että toiminta on oikeellista.

3.2 Semafori

Semafori on käyttöjärjestelmän käsittelemä rakenne, siis yhdenlainen palvelu, jonka käyttöjärjestelmä tarjoaa rajapintansa kautta. Abstrahoituna se mistä tässä kaikessa on koko ajan kyse: käyttöjärjestelmä virtuaalikoneena ajateltuna tarjoaa rajapinnassaan keinoja allaan olevan tason, eli laitteiston, hallintaan. Semaforien avulla hallitaan osaltaan vuorontamista eli sitä, miten prosessorilaitteelle jaellaan prosessien kontekseja.

Yhdessä semaforissa on **arvo** (*value*) ja prosessien **odotusjono** (*queue*). Esimerkki:

```
Semaforin hetkellinen sisältö:
  Arvo (kokonaisluku):      0
  Jono (prosessi-ID:itä):   PID 213 -> PID 13 -> PID 678 -> NULL
```

Semaforit pitää voida yksilöidä, eli niillä on esimerkiksi nimi tai kokonaislukutunnus. Ne ovat saatavilla ja käytettävissä KJ-kutsujen kautta.

Semaforien yksilöinnin, luomisen ja tuhoamisen lisäksi käyttöjärjestelmä toteuttaa seuraavanlaisen pseudokoodin mukaiset käyttöjärjestlemäkutsut semaforin soveltamiseksi; niiden nimet voisivat olla “wait()” ja “signal()”, mutta yhtä hyvin jotakin muuta vastaavaa... Kutsu tehdään synkronointia tarvitsevasta sovellusohjelmasta, ja sen parametrina tietysti annetaan yhden tietyn semaforin tunniste.

wait(Sem):

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

signal(Sem):

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

3.3 Alustava esimerkki: Poissulkeminen (Mutual exclusion, MUTEX)

Alkutilanne:

```
Olkoon prosessit 77 ja 898, joilla on yhteinen semafori
nimeltään semMunMutexi. Se on alustettu näin:

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

PID 77:n koodia suoritetaan. Siellä on kutsu wait(semMunMutexi), mikä päätyy käyttöjärjestelmäkutsuksi (eli suorittuu ohjelmoitu keskeytys). Silloin prosessi PID 77 menee kernel running -tilaan, ja käyttöjärjestelmän koodi suorittaa semaforin käsittelyn wait() siten kuin edellä on pseudokoodina esitetty. Tässä tapauksessa seuraavaksi tilanne on:

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

käyttöjärjestelmästä palataan PID 77:n koodin suorittamiseen heti wait()-kutsun jälkeiseen käskyyn.

Nyt PID 77:llä on yksinoikeus suorittaa semMunMutexi-semaforilla merkittyä kriittistä aluetta.

Esim. PID 898 tulisi jossain vaiheessa vuoronnetuksi suoritukseen ennen kuin semMunMutexi olisi signaloitu. Sitten PID 898:n koodi lähestyisi kriittistä aluetta ja siihen kirjoitettu wait() aiheuttaisi seuraavan tilanteen:

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

käyttöjärjestelmä siirtäisi prosessin 898 Blocked-tilaan, ja liittäisi sen semMunMutexin jonoon odottamaan signal()-kutsua. Tämä (kuten ylipäätään käyttöjärjestelmäkutsu aina) tapahtuu sovellusohjelmien kannalta “atomisesti” eli mikään käyttäjän prosessi ei pääse suorittumaan ennen kuin käyttöjärjestelmä on tehnyt vaadittavat organisointi- ja kirjanpitotyöt.

Useilla prosesseilla voisi olla erilaisia toimenpiteitä semMunMutexilla suojattuun jaettuun resurssiin. Vuorontaja jakelisi prosesseille aikaa ja kaikki tapahtuisi nykyprosessorissa hemmetin nopeasti. 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(semMunMutexi) ensimmäisenä, saa lopulta operaationsa valmiiksi jollakin ajovuorollaan, ja jos se on oikeellisesti ohjelmoitu, niin kriittisen alueen lopussa on kutsu signal(semMunMutexi). Jälleen käyttöjärjestelmä atomisesti hoitaa tilanteeksi:

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

ja 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 tässä ns. binäärisen semaforin tapauksessa tarkoittaa, että resurssi ei vielä ole vapaa, vaan siellä joku suorittaa kriittistä aluetta, ja mahdollisesti sinne on jo jonoakin päässyt kertymään.

Sitten jos uusia jonottajia ei ole wait() -kutsun kautta tullut, ja aiemmat prosessit ovat yksi kerrallaan suorittaneet kriittisen alueensa ja kutsuneet signal(), niin viimeinen signal() tapahtuu esitilanteessa:

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

Ja signalin jälkeen resurssi vapautuu täysin, sillä tilanne on:

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

Tämä vastaa esimerkin ihan ensimmäistä tilannetta.

Tärkeätä on huomata, että ohjelmoijan on varmistettava, että hänen sovellusohjelmansa kutsuu käyttöjärjestelmän palveluita oikeellisesti:

```
Wait(S)  //  atominen käsittely, käyttäjän prosessit keskeytettynä.

... kriittinen alue ...

Signal(S) //  atominen käsittely
```

3.4 Tuottaja-kuluttaja -probleemin ratkaisu

Tuottaja-kuluttaja -ongelma eli kahden prosessin välinen tietovirran synkronointi voidaan ratkaista semaforeilla seuraavaksi esitetyllä tavalla. Tuottajia voisi olla useampiakin - kuluttajia kuitenkin vain yksi. Toinen perinteinen, erilainen, ongelma-asettelu on ns. “kirjoittajien ja lukijoiden” ongelma. Lisäksi on muita perinteisiä esimerkkiongelmia, mm. “ruokailevat filosofit”, ja todellisessa ohjelmistotyössä jokainen yhdenaikaisuutta hyödyntävä sovellus saattaa tarjota uusia vastaavia tai erilaisia ongelmia, jotka on ratkaistava että ohjelma toimisi joka tilanteessa oikeellisesti. Myös ratkaisutapoja on muitakin kuin semaforit. Yksinkertaisuuden vuoksi Käyttöjärjestelmät -kurssilla käydään läpi vain yksi yksinkertainen ongelmatapaus ja yksi yksinkertainen ratkaisu siihen. Kokonaisuuden kannalta on tärkeätä muistaa:

- Javassa ja joissain muissa edistyneissä työkaluissa synkronointi on tehty helpoksi; mutta aina ei ole mahdollisuutta siihen iloon, että koodaisi näillä edistyneillä työkaluilla.
- Samat ongelma-asettelut ovat taustalla, olipa työkalu miten edistynyt tahansa.
- Jotkut ongelmat voivat olla ratkaisemattomissa, jolloin suunnitelma tai vaatimusmäärittely menee yhdenaikaisuuden osalta uusiksi.

- Käyttöjärjestelmä on avain oikeelliseen synkronointiin, koska se hallitsee vuoronnusta ja siis vain sen avulla saa aikaan atomisen operaation. Älä siis edes mieti synkronointia millään kepulikonstilla. Sen on oltava joko työkalussa (kuten Java) tai alustassa (Windows, Linux, OS-X ...).

Tuottaja-kuluttaja -ongelman ratkaisuun tarvitaan seuraavat kolme semaforia:

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

Ohjelmoijan vastuulla on seuraavaa. Aluksi pitää alustaa semaforit seuraavasti:

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

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

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

Tuottajan idea:

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

  wait(MUTEX)    // poissulku binäärisellä semaforilla; ks. edell. esim

  ...Siirrä tuotettu data puskuriin (vaikkapa megatavu tai muuta hurjaa)

  signal(MUTEX)

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

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

                 // ... jolloin FULLin jono tyhjenee, kuluttajia kun on
                 // vain tasan yksi. Eli vuoronnuksesta
                 // riippuen tuottaja(t) voi ehtiä monta kertaa suorituk-
seen
                 // ennen kuluttajaa, ja silloin se ehtii kutsua
                 // signal(FULL) monta kertaa, ja FULL.Arvo voi olla
```

```
                    // mitä vaan >= 0 siinä vaiheessa, kun kuluttaja
                    // pääsee apajille.
```

Kuluttajan idea:

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

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

  wait(MUTEX)  // tämä taas selvä jo edellisestä esimerkistä.

  Lue data puskurista (vaikkapa se megatavu)

  signal(MUTEX)

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

HUOM 1: Yllä on 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).

HUOM 2: Tässä oli ratkaisu kahteen pulmaan: resurssin johdonmukaiseen käyttöön poissulkemisen (*Mutual exclusion*, "*MutEx*") kautta, ja tasan yhden vastaanottavan prosessin yksisuuntaiseen puskuroituun tietovirtaan eli tuottaja-kuluttaja -tilanteeseen. Todelliset IPC-ongelmat voivat olla tällaisia, mutta ne voivat olla monimutkaisempiakin: voi olla useita "tuottajia", useita "kuluttajia", useita eri puskureita ja useita sovellukseen liittyviä toimintoja. Olet nähnyt tässä yksinkertaisia perusperusteita, joista toivottavasti syntyy jonkinlainen pohja ymmärtää monimutkaisempia tilanteita myöhemmin, jos joskus tarvitsee.

HUOM 3: Olet nähnyt 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". Mutta tähän kohtaan siis vedetään tällä kurssilla rajaus synkronoinnin käsittelyssä.

4 Käyttöjärjestelmän rakenteesta: monoliittinen / microkernel

Käyttöjärjestelmän tehtävä kaikkein yleisimmillän on tarjota kaikkein alimman tasoinen ja vähiten abstrakti virtuaalikonerajapinta, joka abstrahoi fyysisen laitteiston käytön.

Mitä osa-alueita tuon päätehtävän toteuttaminen vaatii? Nyt kun ymmärretään toisaalta laitteiston ominaisuudet ja toisaalta käyttäjän tarpeet, voitaneen sanoa, että käyttöjärjestelmän tehtäviä ovat ainakin prosessinhallinta, muistinhallinta, tiedostonhallinta, I/O -laitteiden hallinta. Nämäkin yläkäsitteet sitten jakautuvat täsmällisempiin alitehtäviin ja -osiin. Iso ohjelma, kuten käyttöjärjestelmä, on syytä kirjoittaa modulaarisesti, eli siten, että yksi tehtävä hoituu mahdollisimman tarkkaan rajatulla ohjelmakoodilla, moduulilla, joka tarjoaa selkeän ja tarkoituksenmukaisen rajapinnan muille moduuleille.

Luentomonisteen alussa puhutaan hieman käyttöjärjestelmien tehtävistä ja rakenteesta. Siellä mainitaan mm. monoliittinen eli kerrosmainen käyttöjärjestelmätoteutus ja sitten microkernel-rakenne. Mielestäni nämä toteutusmahdollisuudet alkavat olla paremmin ymmärrettävissä tässä vaiheessa kurssia, kun ollaan (toivottavasti) päästy kärryille prosessorilaitteistosta, prosessorin kernel/user -toimintatiloista sekä kontekstinvaihdoista ja vielä prosessien välisestä kommunikaatiostakin.

Monoliittinen käyttöjärjestelmätoteutus:

- Koko käyttöjärjestelmä kaikkine palveluineen on koko ajan muistisssa siitä asti kun se on ladattu käynnistyksen yhteydessä.
- Käyttöjärjestelmää voidaan ajatella yhtenä erityisenä prosessina, jolla on konteksti ja muistialueita (tosin siis todellisina eikä virtuaalisina muistiosoitteina)
- Kaikki käyttöjärjestelmän palvelut toimitetaan prosessorin ollessa käyttöjärjestelmätilassa, johon se päätyy aina keskeytyksen tullessa.
- mm. bugi laiteajurissa voi jumittaa koko koneen, jos keskeytykset ovat estettynä; silloinhan vuorontajakoodi ei pääse hallitsemaan prosesseja...
- Käyttöjärjestelmäohjelmiston käsitteellinen ja lähdekoodissa toteutuva modulaarisuus ei ilmene sen suorituksessa: Suojausta moduulien välillä ei ole laitteistotasolla -- periaatteessa kaikilla moduuleilla on (väärin ohjelmoituna) mahdollisuus kajota toistensa datoihin ja sotkea kokonaisuuden toiminta perusteellisesti. Monoliittisen käyttöjärjestelmän toteuttaminen on kurinalaista ja vaativaa. Siis bugeja tulee melkeinpä väistämättä, kun erehtyväiset ihmiset näitä koodaavat.
- Uuden ominaisuuden lisääminen käyttöjärjestelmään, tai jonkun toiminnallisuuden muuttaminen edellyttää koko käyttöjärjestelmäkoodin kääntämistä uudelleen, vanhan binääritiedoston korvaamista uudella, ja tietokoneen uudelleenkäynnistämistä asennuksen jälkeen.

Monoliittisessa käyttöjärjestelmässä on siis tiettyjä hankaluuksia. Kuitenkin nykyään useimmiten näemme käytännössä monoliittisia käyttöjärjestelmiä. Syitä ovat ainakin seuraavat:

- monoliittiset ovat toistaiseksi tehokkaampia (eli prosessoriaikaa jää käyttäjäprosessien suorittamiseen enemmän, kun käyttöjärjestelmäprosessi ei vaadi niin paljon aikaa esim. suojattuun viestinvälitykseen moduuliensa välillä)
- ei ole vielä tehty yleisimmin käytettyjä käyttöjärjestelmiä uudellen alusta alkaen siten, että rakenne olisi erilainen.

Microkernel-rakenne puolestaan olisi seuraava:

- Käyttöjärjestelmän “ytimen ydin” (se joka toimii käyttöjärjestelmätilassa) tosi pieni, siis “micro” “kernel”. Microkernelin tehtävänä olisi esim. lähinnä vuorontaminen, fyysisen muistin varaaminen, keskeytyskäsittelyn aloitus ja minimalistinen viestinvälitys prosessien välillä.
- Laiteajurit, muistinhallinnan algoritmit, tiedostonhallinta ym. lisäosat toimivat käyttäjätilassa eli prosesseina siinä missä mikä tahansa prosessi!
- Käyttäjän prosessi (sovellusohjelma) pyytää palveluita käyttöjärjestelmän prosesseilta. Kaikki tapahtuu prosessorin kannalta käyttäjätilassa, paitsi viestinvälitys ja primitiivisimmät laiteoperaatiot.
- mm. suojatumpi, helpompi saada vakaaksi; modulaarisempi -- eli monet monoliittisen käyttöjärjestelmän ongelmat korjautuisivat.
- edellyttää tehokasta IPC:tä käyttäjätilassa toimivien osien välillä

Tutkimus microkernel-mallista on aktiivista, ja voi olla että tulevaisuudessa ainakin uudet käyttöjärjestelmät voivat hyvinkin olla microkernel-tyyppisiä, jos tehokkuudessa päästään jollain keinoin lähelle monoliittista mallia.

5 Moniprosessorikoneet

Perinteinen Flynnin luokitus, kuva ja pieni selitys löytyy luentomonisteessa...

Nykyään on yleistymässä monen prosessorin ja yhden muistin koneet.

Esim. jalavan Intel Xeon Esim. charran AMD Opteron Esim. Playstation 3:n IBM Cell.

Positiivista:

- prosessit voidaan ajaa myös yhtä aikaa - ei vain vuorotellen.
- Erityisesti säikeet voidaan ajaa yhtä aikaa, eli rinnakkaistuvan tehtävän suoritusaika voi lyhentyä dramaattisesti verrattuna suoritukseen yhdellä prosessorilla. Edellyttää KLT-toteutusta ja sitä, että sovellukset käyttävät säikeitä johdonmukaisesti.

Haasteellista:

- vuorontaminen, kun voi olla yhtä aikaa monta prosessia/säiettä Running-tilassa eikä vain yksi.
- muistin käyttö (yhteinen väylä, yksi keskusmuisti vaikka prosessoreilla on omat välimuistit ja rekisterit, ...)
- synkronointi (miten saadaan käyttöjärjestelmän toimenpiteet atomisiksi, kun keskeytyskäsittely on prosessorikohtainen ja samaan aikaan voi toinen prosessori jatkaa käyttäjätilassa)
- ja varmaan paljon muutakin ...

Oleellista huomata:

- käyttäjän näkökulmasta ei mitään uusia ongelmia moniajossa.

Ehkä pari sanaa tietyistä Flynnin luokituksen luokista... Lyhenteet SIMD, MIMD, jne. on avattu luentomonisteessa. Alla on muutaman sanan laveampi kuvaus joistakin yksittäisistä moniprosessoritoteutuksista vähän eri nimillä:

5.1 SMP

Tänä päivänä tyypillinen prosessointimuoto on SMP, symmetrinen moniprosessointi, jossa tosiaan rinnakkaiset prosessorit ovat kaikin tavoin tasavertaisia. Käyttöjärjestelmän ja käyttäjäprosessien koodia ajetaan niissä molemmissa.

5.2 Master/Slave

Toinen prosessointimuoto on Master/Slave -tyyppinen, jossa pääprosessori voi pyytää palveluita vaikkapa liukulukulaskentaan erikoistuneilta apuprosessoreilta. Käyttöjärjestelmä ja käyttäjien prosessit toimivat pääprosessorissa (tai useammassa, kuten SMP:ssäkin) ja lisäksi apuprosessoreille voi jaella laskentatehtäviä eli niiden käskykannan mukaista koodia rinnakkain suoritettaviksi; kommunikointi tapahtuu edelleen yhden keskusmuistin ja väylän avulla; synkronointi pää- ja apuprosessorien kesken on hallittava jotenkin.

5.3 Klusterit

Lisäksi on pitkään ollut ns. klustereita, eli on monta prosessoria, joilla jokaisella on oma muisti; viestinvälitys tapahtuu lähiverkon tai internetin yli.

6 I/O:sta ja levyistä

Kovalevyn käyttö on hidasta, kuten I/O -laitteiden yleensäkin ottaen! Syitä on useita:

- matka laitteen ja prosessorin välillä on pitkä.
- välissä on laitteen kontrollilogiikka ja väylä komennettavana
- kovalevyn lukupää on siirrettävä
- levyn on pyörähdettävä niin että luettava kohta tulee lukupään kohdalle
- levy voi olla jopa virransäästötilassa täysin pysähdyksissä.

Yhden luku- tai kirjoitusoperaation aikana prosessori ehtii palvella montaa prosessia, jotka eivät tarvitse samaa laitetta. Keskeytykset ja vuoronnus ovat avain tähän.

Olisi tehotonta, jos yhdellä käyttöjärjestelmäkutsulla voisi lukea levyltä vain merkin kerrallaan tai edes sektorin kerrallaan. Siksi on kehitetty **DMA** eli *direct memory access* -toimintamalli, jossa:

- I/O-laitteet voivat lukea/kirjoittaa muistia suoraan.
- I/O -toimenpide alustetaan, jolloin kerrotaan muistiosoite ja esim. kovalevylle levyosoite, josta levyllä täytyy hakea.

- Kun I/O on valmis, laitteelta tulee keskeytys. Muisti on silloin jo päivitetty, jos kyseessä oli lukuoperaatio. Levylle on kirjoitettu, jos oli kirjoitusoperaatio.

6.1 Joitain pointteja kovalevylaitteistoista

Seuraavat mainittiin tai katsottiin tykiltä.

Kovalevy:

- raidat, sektorit ym.

RAID (Redundant Array of Independent Disks)-pakat:

- monta kovalevyä rinnakkain.
- hajautetaan tiedostoja eri fyysisille levyille (esim. peräkkäiset kilotavun pätkät eri levyllä). Tämä nopeuttaa lukua ja kirjoitusta
- peilataan tiedostoja eri levyillä eli säilytetään kopiota datasta (tai mieluummin tallennetaan "pariteettitietoa" varsinaisen tiedon lisäksi). Jos yksi levy kerrallaan on rikki, niin dataa ei menetetä. Katastrofi tapahtuu vasta jos kaksi levyä ehtii olla rikki yhtä aikaa (... mahdollistahan sekin on!).

6.2 I/O -ohjelmistosta

I/O -ohjelmisto on järkevää toteuttaa virtuaalikonehierarkiana (lm. s. 53 kertoo enemmän kuin tämä):

Käyttäjän prosessi

Laiteriippumaton I/O -ohjelmisto

Laiteajuri

Keskeytyskäsittelijä

Laite

Laiteriippumaton I/O -ohjelmisto:

- tiedostojärjestelmä
- hakemistorakenne
- Tiedostoille & laitteille: nimet, oikeudet, lohkokoko
- puskurointi
- virheraportointi
- rajapinnat ylös (käyttäjälle) ja alaspäin (laiteajureille).

6.3 UNIX-tiedostojärjestelmä

Jokaisella tiedostolla on i-numero. Indeksi i-solmutaulukkoon.

I-solmussa:

- tyyppi (tiedosto, hakemisto, erikoistiedosto, linkki)
- uid, gid, aika, oikeudet
- koko
- lohkojen määrä
- suorat fyysisten lohkojen osoitteet 1: 283746 2: 430593 3: 12: 234527
- single indirect -> lisää fyysisiä lohko-osoitteita
- double indirect -> lisää "single indirect"-taulukoita
- triple indirect -> lisää "double indirect"-taulukkotaulukoita.

Tiedosto pitää pystyä:

- luomaan
- poistamaan
- nimeämään
- sijoittamaan hakemistorakenteeseen
- avaamaan lukua/kirjoitusta varten
- lohkoittainen ja puskuroitu (merkki kerrallaan) luku ja kirjoitus
- lukitsemaan synkronointia varten
- ...

Tietysti jokaista tavoitetta varten on käyttöjärjestelmäkutsu.

Sisäisessä toteutuksessa jälleen PCB:t olennaisia. Eli tieto prosessin avatuista tiedostoista on prosessin PCB:ssä. Missä muodossa? Useimmiten kokonaislukuna (t. "kahvana" t. "deskriptorina") Esim. tiedosto voi olla auki useiden prosessien tekemää lukua varten.

Deskriptori on prosessin oma kahva, jolla se voi pyytää tiedostokohtaisia palveluita KJ:ltä.

Käyttöjärjestelmä pitää yllä tiedostotaulua, jossa on enemmän tietoa kuin prosessin tarvitsema deskriptorinumero. Erityisesti yhteys laiteajureihin ja tiedostojärjestelmän sisäiseen (käyttäjältä rajapinnan taakse piilotettuun) toteutukseen.

Tiedostojärjestelmiä on MONTA!

- erilaisia
- eri tarkoituksiin
- eri aikoina syntyneitä
- mahdollisuudet ja rajoitukset

Näkyviä eroja:

- tiedostonimien pituudet, kirjainten/merkkien tulkinta ym.

Syvällisiä eroja:

- toteutusyksityiskohdat, päämäärät:
  - nopeus/tehokkuus (haku, nimen etsintä, tiedoston luonti, kirjoitus, luku, poisto, isot vai pienet tiedostot)
    -> välimuistit, puskurit
    -> "Yksi koko ei sovi kaikille" - tiedostojärjestelmä on valittava kokonaisjärjestelmän käyttötarkoituksen mukaan.
  - toimintavarmuus (mitä tapahtuu, jos tulee sähkökatko tai laitevika)
    -> "transaktioperiaate", "journalointi"
    Kirjoitusoperaatiot tehdään atominen kirjoituspätkä kerrallaan.
    Ensin tehdään kirjoitus yhteen paikkaan, "ennakkokirjoitus" (ei vielä siihen kohtaan levyä, mihin on lopulta tarkoitus) Kirjoitetaan myös tieto, mihin kohtaan on määrä kirjoittaa. Jos kirjoitus ei ehdi jostain syystä toteutua, se voidaan suorittaa alusta lähtien uudelleen käyttämällä ennakkoon tehtyä ja tallennettua suunnitelmaa.

Sivun 64 pseudokoodeissa olennaista (ehkä, mun mielestä):

- levyoperaatiot mahdollista tehdä yksi kerrallaan
- pyyntöjä ehtii luultavasti tulla monta eri prosesseilta eri paikkoihin levyä kohdistuen
- pitää siis olla jonkinlainen odottelukäytäntö (joka ei hukkaa yhtään pyyntöä...)
- Palvelu alustetaan ajurin toimesta -> laite hoitaa siitä eteenpäin -> tulee keskeytys, jolla on käsittelijäkoodi -> laiteajuri hoitaa loppuun joko saman tien tai myöhemmin vuoronnettuna.

Tiedostot ovat avain kaikkeen järkevään informaatioteknologian hyödyntämiseen - ovathan ne ainoa, mikä säilyy silloinkin, kun sähkö on pois päältä. Niinpä niitä täytyy ymmärtää syvällisemmin kuin sillä tasolla, että "tekstidokumentti näkyy paperipinkan muotoisena kuvakkeena" tai "peli lähtee päälle kun klikkaa ikonia". Tavoite on tarjota käyttäjälle hyvin toimiva järjestelmä, mikä tarkoittaa joka hetkellä oikeellisena pysyvää tietoa - se taas edellyttää synkronointia, käytettävissä olevan tilan seuraamista ym. Olipa ohjelmointityökalu mikä tahansa, tiedostot sijaitsevat loppukädessä käyttöjärjestelmän palveluiden takana, joita käytetään jonkinlaisen virtuaalikonehierarkian kautta... jotta käyttäjä näkee tietonsa näppärästi kuvakkeina, teksti- ja grafiikkalaatikoina ikkunoissa tmv. ja voi luottaa siihen, ettei mitään häviä vahingossa.