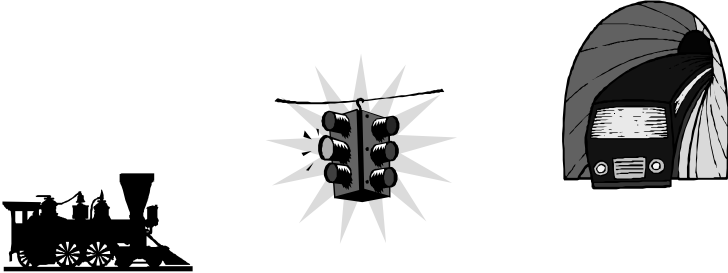
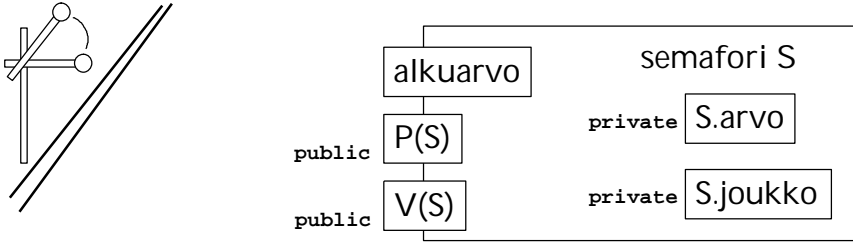


Semaforit



5.6.2006 Liisa Marttinen 2 - 1

Semaforit



```

classDiagram
    class SemaforiS {
        alkuarvo
        private S.arvo
        private S.joukko
        public P(S)
        public V(S)
    }
  
```

- **P()** aka *WAIT()* aka *Down()*
 - | jos kriittinen alue vapaa, lukitse se ja jatka eteenpäin
 - | jos kriittinen alue varattu, odota
- **V()** aka *SIGNAL()* aka *Up()*
 - | jos joku odottamassa, päästä joku odottajista etenemään,
 - | Jos odottajajoukko on tyhjä, vapauta kriittinen alue
- P ja V atomisia: samanaikaisesti vain yksi prosessi voi suorittaa niitä

5.6.2006 Liisa Marttinen 2 - 2

Yleistä semaforeista

- Alkuperäinen idea: Dijkstra (60-luvun puoliväli)
 - | **binäärisemaforit: vain arvot 0 ja 1 käytössä**
- Operaatiot
 - | **Alustus** **sem S=1; sem forks[5] = ([5] 1);**
 - | **P(S)** **(= passeren)**
 - | **V(S)** **(= vrijgeven)**
- S.arvo
 - | **1** $\hat{=}$ vapaa / avoinna / etenemislupa
 - | **0** $\hat{=}$ varattu / suljettu / eteneminen kielletty
- S.joukko
 - | **Etenemislupaa Blocked (Wait) -tilassa odottavat prosessit**
 - | **Toteutus: esim. PCB:t linkitetyssä listassa (\Rightarrow jono)**

5.6.2006

Liisa Marttinen

2 - 3

Yleiset semaforit: S.arvo ~ resurssilaskuri

- **S.arvo > 0**
 - | **vapaiden yksiköiden lkm, etenemislupien määrä**
 - | **kuinka monta prosessia voi tehdä P()-operaation joutumatta Blocked tilaan ja semaforin jonoon**
- **S.arvo = 0**
 - | **ei vapaita**
- **[jotkut toteutukset: S.arvo < 0]**
 - | **jonossa odottavien prosessien lkm**
- **Ei operaatiota**
 - | **jonon käsittelyyn / pituuden kyselyyn!**
 - | **semaforin arvon kyselemiseen!**
 - | **jos tarve tietää, ylläpidä ko. tieto omassa muuttujassa**

5.6.2006

Liisa Marttinen

2 - 4

P/V:n toteutus KJ:n ytimessä

(Andrews Fig 6.5)

- P(S)

```
if (S.arvo > 0)
    S.arvo = S.arvo - 1
else
    aseta prosessin tilaksi Blocked,
    lisää prosessi jonoon S.jono
call Vuorottaja()
```
- V(S)

```
if (isEmpty(S.jono))
    S.arvo = S.arvo + 1
else
    siirrä S.jonon ensimmäinen prosessi
    Ready-jonoon (tilaksi Ready)
call Vuorottaja()
```

5.6.2006

Liisa Marttinen

2 - 5

P/V:n toteutus

(toinen tapa, Stallings Fig. 5.6 Ja 5.7)

- P(S)

```
S.arvo = S.arvo - 1
if (S.arvo < 0)
    aseta prosessin tilaksi Blocked,
    lisää prosessi jonoon S.jono
call Vuorottaja()
```
- V(S)

```
S.arvo = S.arvo + 1
if (S.arvo >= 0)
    siirrä S.jonon ensimmäinen prosessi
    Ready-jonoon
call Vuorottaja()
```
- Miten ohjelmoijan huomioitava tämä toteutusero?

5.6.2006

Liisa Marttinen

2 - 6

Poissulkeminen semaforia käyttäen

```
sem mutex=1;      # vain perinteinen muuttujan nimi

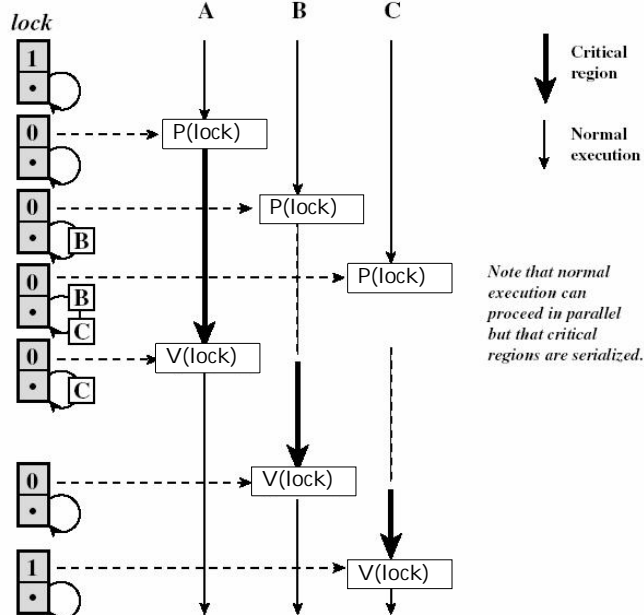
process CS [i=1 to N] { # rinnakkaisuus!
  while (true){
    ei kriittistä koodia;
    P(mutex);      # varaa
    < critical section > # käytä (exclusive!)
    V(mutex);      # vapauta
    ei kriittistä koodia;
  }
}
```

- yksi semafori kullekin erilliselle kriittiselle alueelle
- huomaa oikea alkuarvo

5.6.2006

Liisa Marttinen

2 - 7

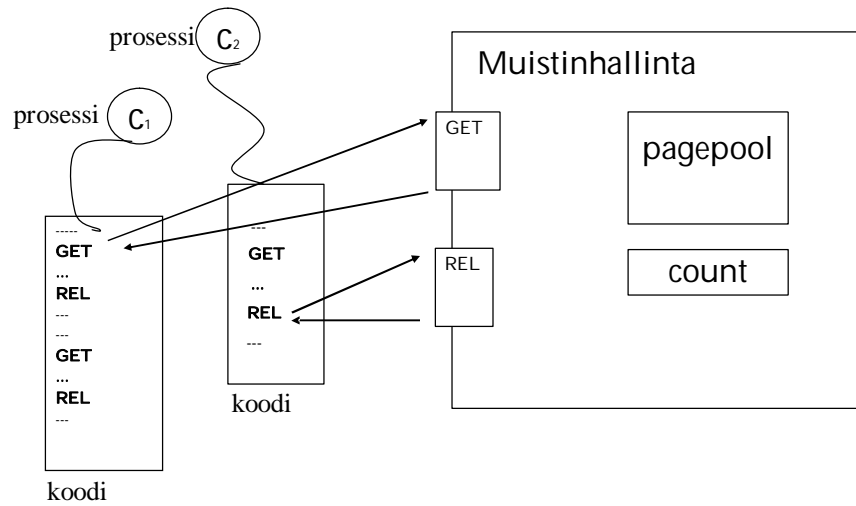


5.6.2006

Liisa Marttinen

2 - 8

Esimerkki: Muistinhallinta (1)



5.6.2006

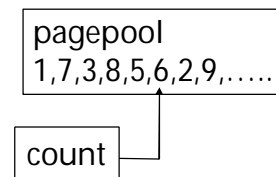
Liisa Marttinen

2 - 9

```
addr pagepool[1:MAX];    # vapaita muistitiloja ; käyttö pinona
int count=MAX;
```

```
addr function GET() {
  addr memaddr;
  memaddr = pagepool[count];
  count = count-1;
  return memaddr ;
}
```

```
procedure REL(addr freepage) {
  count = count+1;
  pagepool[count] = freepage;
}
```



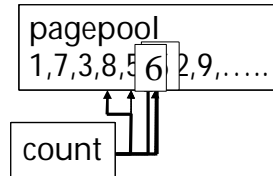
5.6.2006

Liisa Marttinen

2 - 10

• Toimiikos tuo?

```
addr pagepool[1:MAX];  
int count=MAX;
```



```
addr function GET:
```

```
{ memaddr =  
  pagepool[count];  
  count = count-1  
}
```

```
GET==6
```

```
addr function GET:
```

```
{ memaddr =  
  pagepool[count];  
  count = count-1  
}
```

```
GET==6
```

5.6.2006

Liisa Marttinen

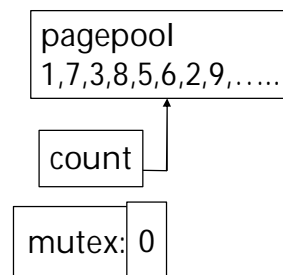
2 - 11

Muistinhallinta (1): poissulkeminen

```
addr pagepool[1:MAX];  
int count=MAX;  
sem mutex=1;
```

```
addr function GET() {  
  addr memaddr;  
  P(mutex);  
  memaddr = pagepool[count];  
  count = count-1;  
  V(mutex);  
  return memaddr;  
}
```

```
procedure REL(addr freepage) {  
  P(mutex);  
  count = count+1;  
  pagepool[count] = freepage;  
  V(mutex);  
}
```

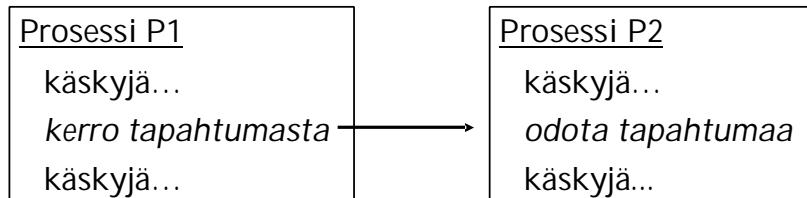


5.6.2006

Liisa Marttinen

2 - 12

Ehtosynkronointi



- Tapahtuma: *"mikä tahansa kiinnostava"*
puskuri_täynnä, io_valmis, laskenta_valmis, kriittinen alue vapaa
- (Looginen) tapahtuma $\hat{=}$ semaforimuuttuja
- Kumpi ehtii ensin synkronointikohtaan?
 $\hat{=}$ tarvitsee paikan missä odottaa (-jonottaa)

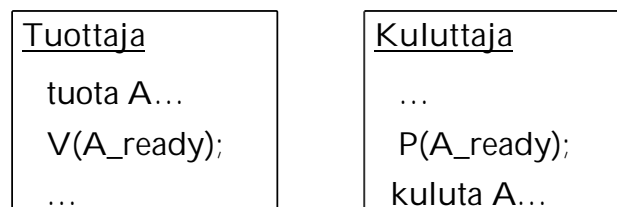
5.6.2006

Liisa Marttinen

2 - 13

Synkronointi semaforia käyttäen

- sem A_Ready = 0;
 - | 0 $\hat{=}$ "ei ole tapahtunut", 1 $\hat{=}$ "on tapahtunut"



- | **Kumpi ehtii ensin?**
- | **Oikea alkuarvo?**

5.6.2006

Liisa Marttinen

2 - 14

Muistinhallinta (2): ehdosynkronointi

```

addr pagepool[1:MAX];
int count=MAX;
sem mutex=1, avail=MAX;

addr function GET() {
  addr memaddr;
  P(avail);
  P(mutex);
  memaddr = pagepool[count];
  count = count-1
  V(mutex);
  return memaddr;
}

procedure REL(addr freepage) {
  P(mutex);
  count = count+1;
  pagepool[count] = freepage;
  V(mutex);
  V(avail);
}

```

pagepool
1,7,3,8,5,6,2,9,.....

count

mutex 0

avail 6

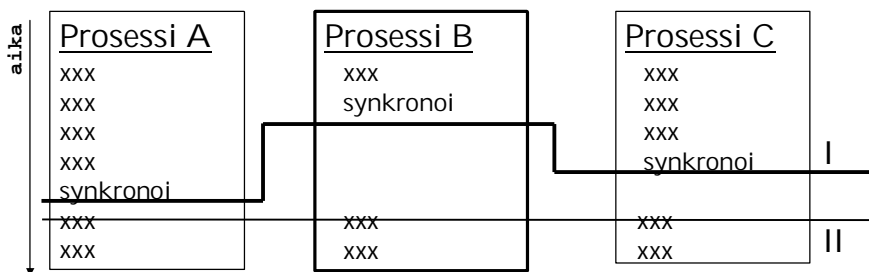
5.6.2006

Liisa Marttinen

2 - 15

Puomisynkronointi (barrier)

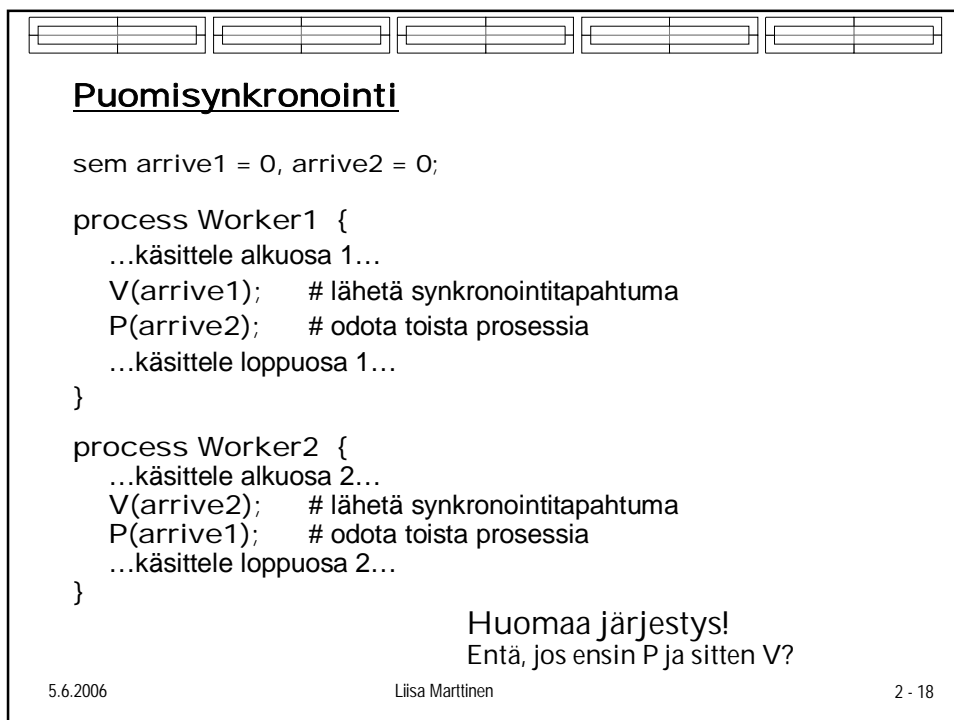
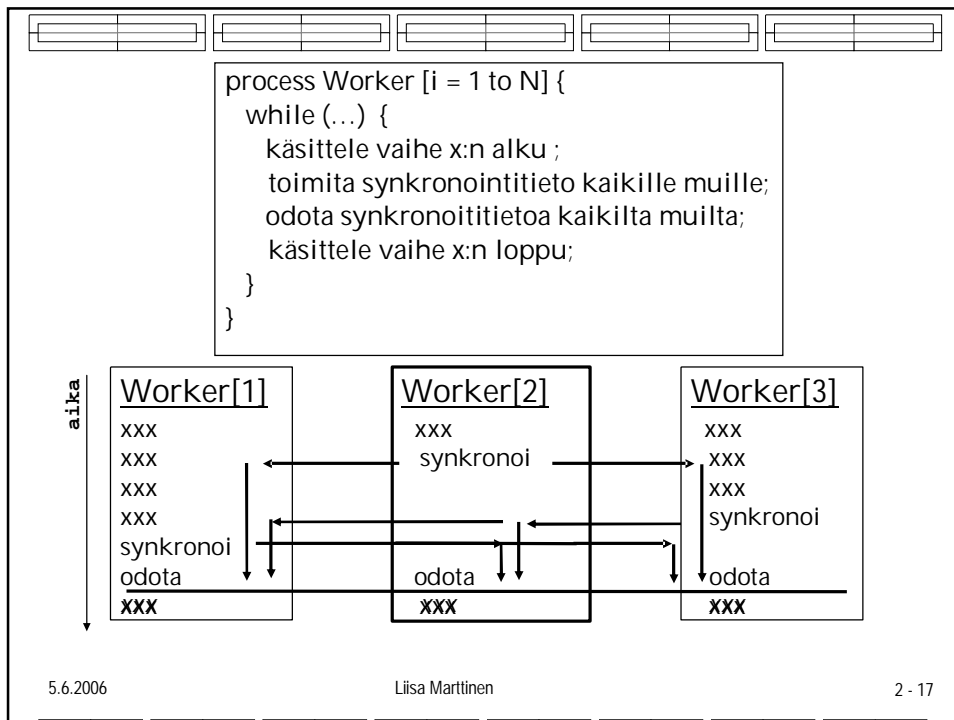
- n prosessia työskentelee yhdessä
- Iteratiivinen käsittely, synkronointi tietyn välivaiheen jälkeen



5.6.2006

Liisa Marttinen

2 - 16



Väärin toteutettu puomisynkronointi!

sem arrive1 = 0, arrive2 = 0;

```
process Worker1 {
  ...käsittele alkuosa 1...
  P(arrive2); # odota toista prosessia
  V(arrive1); # lähetä synkronointitapahtuma
  ...käsittele loppuosa 1...
}
```

```
process Worker2 {
  ...käsittele alkuosa 2...
  P(arrive1); # odota toista prosessia
  V(arrive2); # lähetä synkronointitapahtuma
  ...käsittele loppuosa 2...
}
```

Johtaa lukkiutumiseen: kumpikin
vain odottaa toista!!

5.6.2006

Liisa Marttinen

2 - 19

Entä toimiiko näin? Onko mitään haittaa?

sem arrive1 = 0, arrive2 = 0;

```
process Worker1 {
  ...käsittele alkuosa 1...
  P(arrive2); # odota toista prosessia
  V(arrive1); # lähetä synkronointitapahtuma
  ...käsittele loppuosa 1...
}
```

```
process Worker2 {
  ...käsittele alkuosa 2...
  V(arrive2); # lähetä synkronointitapahtuma
  P(arrive1); # odota toista prosessia
  ...käsittele loppuosa 2...
}
```

5.6.2006

Liisa Marttinen

2 - 20

Puomisynkronointi, yleiset semaforit

```
sem arrive=0, continue[1:N] = ( [N] 0 );
process Worker [i=1 to N] {
  ...käsittele alkuosa i...
  V(arrive);          # synkronointitapahtuma koordinaattorille
  P(continue[i]);    # odota toisia prosesseja
  ...käsittele loppuosa i...
}
process Coordinator {
  while (true) {
    count = 0;
    while (count < N) { # odota, kunnes N synkronointitapahtumaa
      P(arrive);
      count++;
    }
    for [i = 1 to N] V(continue[i]); # synkronointitapahtuma prosesseille
  }
}
```

Entä jos vain yksi continue-semafori?

5.6.2006

Liisa Marttinen

2 - 21

Virheellinen puomisynkronointi! Miksi näin?

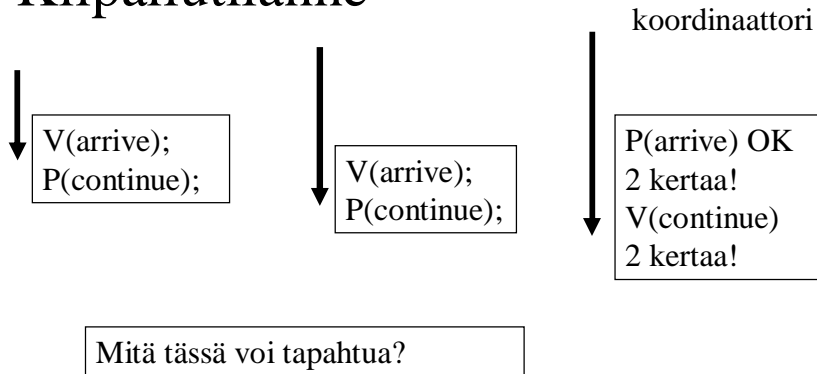
```
sem arrive=0, continue= 0;
process Worker [i=1 to N] {
  ...käsittele alkuosa i...
  V(arrive);          # synkronointitapahtuma koordinaattorille
  P(continue);       # odota toisia prosesseja
  ...käsittele loppuosa i...
}
process Coordinator {
  while (true) {
    count = 0;
    while (count < N) { # odota, kunnes N synkronointitapahtumaa
      P(arrive);
      count++;
    }
    for [i = 1 to N] V(continue); # synkronointitapahtuma prosesseille
  }
}
```

5.6.2006

Liisa Marttinen

2 - 22

Kilpailutilanne



5.6.2006

Liisa Marttinen

2 - 23

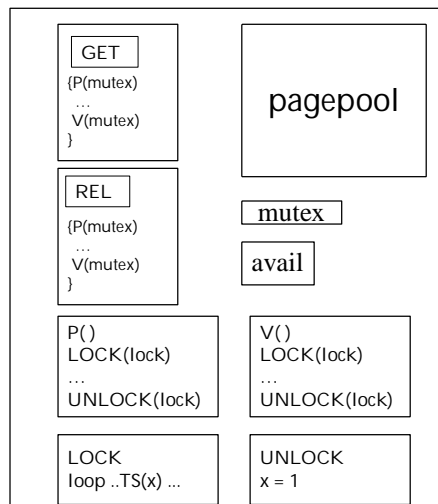
Rinnakkaisuuden hallinta

resurssien varaus
get/rel
muuttujat

hallinnan toteutus
P/V
semafori

P/V:n toteutus
LOCK/UNLOCK
lukkomuuttuja

LOCK/UNLOCK toteutus
test-and-set-käsky
lukkomuuttuja



5.6.2006

Liisa Marttinen

2 - 24

Esimerkkejä semaforin käytöstä

Tuottajat ja kuluttajat
Lukijat ja kirjoittajat
Resurssien hallinta, vuoron antaminen

5.6.2006

Liisa Marttinen

2 - 25

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}

process Consumer[j = 1 to N] {
  while (true) {
    /* fetch result, then consume it */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

buf:

Andrews Fig. 4.3:
Producers and consumers
using semaphores.
(split binary semaphores)

Toimiiko oikein? Mitä pitääkään tarkistaa?

5.6.2006

Liisa Marttinen

2 - 26

Halkaistu binäärisemafori

(Split binary semaphore)

- semaforit empty ja full
 - | **binäärisemaforeja, koska voivat saada vain arvoja 0 ja 1**
 - | **koska aina toisella niistä on arvona 1 ja toisella 0, niin niiden voidaan katsoa olevan yhdestä semaforista halkaistuja osia**
 - | voidaan halkaista myös useampaa osaan: N kappaletta semaforeja, joista aina yhdellä on arvona 1 ja muilla 0
 - | **halkaistu binäärisemafori huolehtii myös keskinäisestä poissulkemisesta:**
 - | vain yksi kerrallaan pääsee suorittamaan sen suojaamaa kriittistä aluetta, jos
 - | yhdellä semaforilla alkuarvona 1, muilla 0
 - | kaikilla prosesseilla koodissaan ensin P-operaatio semaforiinsa
 - | Se prosessi (yksi niistä prosesseista) aloittaa, jonka P-operaation kohteen semaforin arvo on 1.

5.6.2006

Liisa Marttinen

2 - 27

Sem empty = 1, full = 0;

Onko aloitus kunnossa?

Tuottaja i

.....
.....

P(empty);

buf=data;

V(full);

.....

Kuluttaja j

.....
.....

P(full)

result = buf;

V(empty)

.....

Säilyykö oikea järjestys (tahdistus)?

- saa tuottaa, vain jos kuluttaja lukenut

- saa kuluttaa, vain jos tuottaja jotain tuottanut

Tuleeko sekaannusta, jos usea tuottaja tai kuluttaja yrittää samaan aikaan käyttää puskuria?

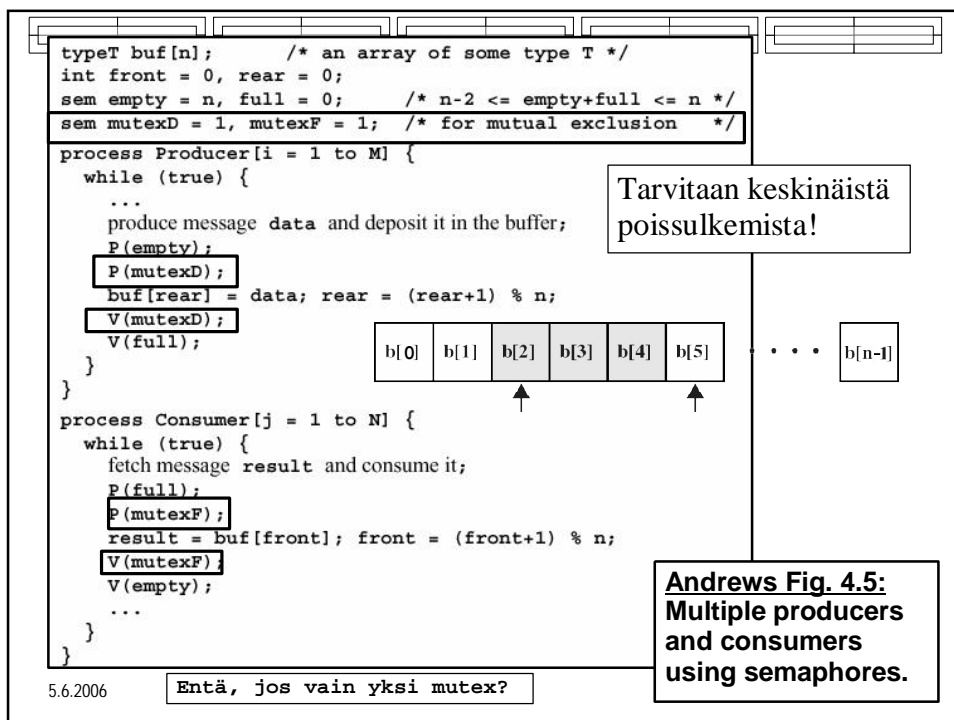
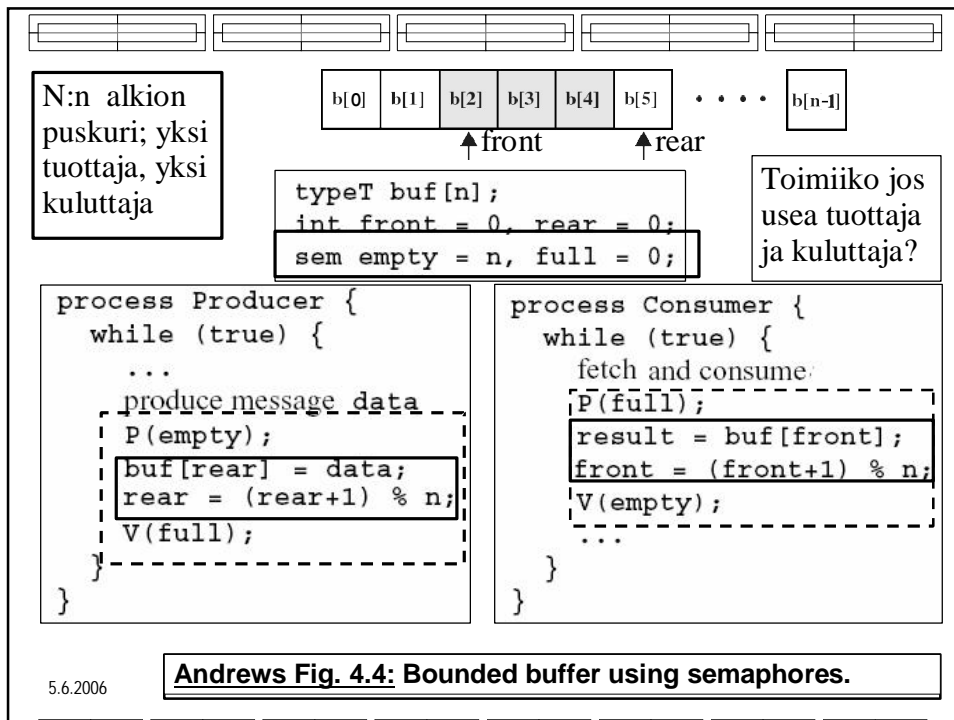
Pääseekö kumpikaan etenemään (lukkiutuminen) ?

Odotetaanko turhaan? Nälkiintyykö jompikumpi?

5.6.2006

Liisa Marttinen

2 - 28



Lukijat ja kirjoittajat

- Yhteinen tietokanta DB
 - | tai joku muu objekti, esim. tiedosto, lista, ...
- Kaksi tietokantaan pääsystä kilpailevaa käyttäjäluokkaa
 - | Lukijat (readers)
 - | lukevat
 - | useita lukijoita voi olla käsittelemässä yhtäaikaan
 - | Kirjoittajat (writers)
 - | lukevat ja muuttavat
 - | vain yksi kerrallaan saa olla muuttamassa tietokantaa
- Luokka aktiivinen vain, jos toinen luokka passiivinen

5.6.2006

Liisa Marttinen

2 - 31

Ratkaisu 1: R/W-poissulkemisongelmana

- Yksinkertaistettu ongelma, yksink. ratkaisu
 - | **Poissulkeminen kaikille muille** => semafori rw
 - | Turhan rajoittava
- Salli lukijoiden toimia rinnakkain
 - | **Luokan poissulkeminen** => semafori rw
 - | **Ensimmäinen lukija varaa tietokannan lukijaluokalle, viimeinen lukija vapauttaa DB:n**
 - | **Kuka on ensimmäinen / viimeinen?** => laskuri nr
 - | **“testaa, varaa / vapauta”** => atominen: < ... >

5.6.2006

Liisa Marttinen

2 - 32

Tehokkaampi ratkaisu 1

- lukijoita voidaan tarkastella yhtenä joukkona
 - | **kun yksi on päässyt lukemaan, muut voivat tulla vapaasti perässä**
 - | vain ensimmäisen lukijan täytyy varmistaa, ettei kukaan kirjoittaja ole kirjoittamassa eli varata kriittinen alue
 - | **kun lukijoita ei enää ole, niin vuoro voidaan luovuttaa kirjoittajille**
 - | viimeinen lukija vapauttaa kriittisen alueen
 - | **tarvitaan laskuri ja sen päivittämistä, jotta tiedetään, mikä prosessi on ensimmäinen ja mikä viimeinen**
 - | `int nr = 0;` # lukijoiden lukumäärä
 - | `nr=nr+1;` # kasvatetaan, kun mennään lukemaan
 - | `nr=nr-1;` # vähennetään, kun poistutaan lukemasta
 - | `if (nr == 1) P(rw);` # ensimmäinen varaa alueen
 - | `if (nr == 0) V(rw);` # viimeinen vapauttaa sen

5.6.2006

Liisa Marttinen

2 - 33

1. lukija varaa tietokannan ja viimeinen lukija vapauttaa sen

```
sem rw = 1;  
int nr = 0;
```

```
sem mutex = 1;
```

```
process Writer [i=1 to N] {  
  while (true) {  
    ...  
    P(rw);  
    write the database;  
    V(rw);  
  }  
}
```

```
process Reader [i=1 to M] {  
  while (true) {  
    ...
```

```
    P(mutex);  
     nr = nr + 1;  
    if (nr == 1) P(rw);  # ensimmäinen lukija  
    V(mutex);  
    read the database;
```

```
    P(mutex);  
     nr = nr - 1;  
    if (nr == 0) V(rw);  # viimeinen  
    V(mutex);
```

```
  }  
}
```

RATKAISUN
REILUUS?

5.6.2006

Liisa Marttinen

2 - 34

Ratkaisu 2: R/W-ehdosynkronointiongelmana

- Odota, kunnes sopiva ehto tulee todeksi
 - | **Toiminnallisuus:** `< await (ehto) lauseet; >`
 - | **Ehdon testaus ja lauseosa atomiseksi**
- Tila
 - | **BAD:** `(nr > 0 and nw > 0) or nw > 1`
 - | **RW:** `(nr == 0 or nw == 0) and nw <= 1`
 - | RW muuttumaton, oltava voimassa aina (invariantti)
 - | nr = number of readers, nw = number of writers
- Ohjelmoi s.e. ehto RW on aina true
 - | **saa lukea** `nw == 0` (ei kukaan kirjoittamassa)
 - | **saa kirjoittaa** `nr==0 and nw==0` (ei muita kirjoittajia eikä lukijoita)

5.6.2006

Liisa Marttinen

2 - 35

```
int nr = 0, nw = 0;
## RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1
process Reader[i = 1 to m] {
  while (true) {
    ...
    <await (nw == 0) nr = nr+1;>
    read the database;
    <nr = nr-1;>
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    <await (nr == 0 and nw == 0) nw = nw+1;>
    write the database;
    <nw = nw-1;>
  }
}
```

Jokaiselle odotukselle oma semafori!

Tässä yksi poissulkeminen ja 2 synkronointiehtoa.

Andrews Fig. 4.11:
A coarse-grained readers/writers solution.

5.6.2006

Liisa Marttinen

- Jaettu binääriarvoinen semafori
 - | Liitä kuhunkin vahtiin (ehtoon) semafori ja laskuri :
sem e, r, w;
 - | Vain yksi semafori kerrallaan 'auki' ($0 \leq (e+r+w) \leq 1$)
 - | alussa: sem e = 1, r = 0, w = 0;
- await (nw == 0) nr = nr + 1
 - | **semafori: r = 0** *waiting place for readers*
 - | **laskuri: dr** *number of delayed readers*
- await (nr == 0 and nw == 0) nw = nw + 1
 - | **semafori: w = 0** *waiting place for writers*
 - | **laskuri: dw** *number of delayed writers*
- poissulkeminen < ... >
 - | **semafori: e = 1** *waiting place for entry*
 - | **laskurit: nr, nw** *numbers of readers and writers*

5.6.2006

Liisa Marttinen

2 - 37

Ratkaisu 2, laajennettu

- Ota huomioon käyttövuorot
 - | Lukijat ensin, tai ...
 - | Vuoro prioriteetin perusteella: Kirjoittajat ensin!
(jos muutokset tärkeitä; nälkiintymisvaara)
- Onko, joku odottamassa vuoroa?
 - | Ei voi kysyä semafori-operaatioilla \bar{O} omat laskurit: dw, dr
- Vuorojen toteutus omassa rutiinissa
SIGNALNEXT
 - | Kriittinen alue vapautuu \bar{O} joku muu saa jatkaa, kuka?
 - | odottava lukija, odottava kirjoittaja, kokonaan uusi tulija

5.6.2006

Liisa Marttinen

2 - 38

```

process Reader {
  while (true) {
    P(e); #varmistaa atomisuuden
    # <await (nw == 0) nr = nr + 1;>
    if (nw > 0){ #joudutaan odottamaan
      dr = dr + 1; V(e); P(r);}
      nr=nr+1;
      SIGNALNEXT;

    read the database;
    # <nr = nr -1;>
    P(e); #varmistaa atomisuuden
    nr=nr-1;
    SIGNALNEXT;
  }
}

```

5.6.2006

Liisa Marttinen

2 - 39

```

process Writer {
  while (true) {
    P(e); #varmistaa atomisuuden
    # <await (nw == 0) nw = nw + 1;>
    if (nr>0 or nw > 0){ #joudutaan odottamaan
      dw = dw + 1; V(e); P(w);}
      nw=nw+1;
      SIGNALNEXT;

    write the database;
    # <nw = nw -1;>
    P(e); #varmistaa atomisuuden
    nw=nw-1;
    SIGNALNEXT;
  }
}

```

5.6.2006

Liisa Marttinen

2 - 40

- **SIGNALNEXT** - vuoron antaminen: Lukijat ensin!

```
if (nw == 0 and dr > 0) {  
    dr = dr -1;  
    V(r);      # herätä odottava lukija, tai  
}  
else if (nr == 0 and nw == 0 and dw > 0) {  
    dw = dw -1;  
    V(w);      # herätä odottava kirjoittaja, tai  
}  
else  
    V(e);      # päästä joku uusi etenemään
```

- Menetelmä: Viestikapulan välitys (Baton passing)

5.6.2006

Liisa Marttinen

2 - 41

Viestikapulan välitys (Baton passing)

- semafori ~ viestikapula;
 - | koska semaforit e, r ja w muodostavat jaetun binäärisemaforin, niin kulloinkin yhdellä niistä on arvo 1 ja muilla arvo 0
- Vain yksi etenee kerrallaan kriittisillä alueilla
 - | pyydettävä etenemislupaa: P(e)
 - | se etenee, joka 'saa' haltuunsa semaforin e
- Muiden odotettava
 - | täysin uusi lukija tai kirjoittaja: P(e)
 - | vuoroaan jonottamaan jääneet lukijat ja kirjoittajat:
Jos etenijän pyyntöön ei voi suostua:
 - | Lukijat: V(e); P(r) (jää odottamaan lukijan vuoroaan)
 - | Kirjoittajat: V(e); P(w) (jää odottamaan kirjoittajan vuoroaan)

5.6.2006

Liisa Marttinen

2 - 42

- Etenijä aktivoi itse "seuraavan viestikapulan haltijan" (kohdassa *SIGNALNEXT*)
 - | **jos vuoroaan odottavia,**
 - | herätä odottaja semaforista: joko $V(r)$ tai $V(w)$
 - | jätä sille poissulkemissemafori valmiiksi kiinni: **älä tee $V(e)$**
 - | **jos ei odottajia,**
 - | jätä viestikapula vapaaksi uusille tulijoille: $V(e)$
- Herätetty välittää aikanaan kapulan seuraavalle ja se seuraavalle ja ...
- Takaa ettei kukaan pääse 'etuilemaan'!
 - | **Ehdot taatusti voimassa, kun jonottaja saa prosessorin**
 - | ***SIGNALNEXT* aktivoi vain yhden prosessin, kun ehto tuli todeksi**
 - | **FCFS**

5.6.2006

Liisa Marttinen

2 - 43

- *SIGNALNEXT* -vuoron antaminen: Lukijat ensin!


```

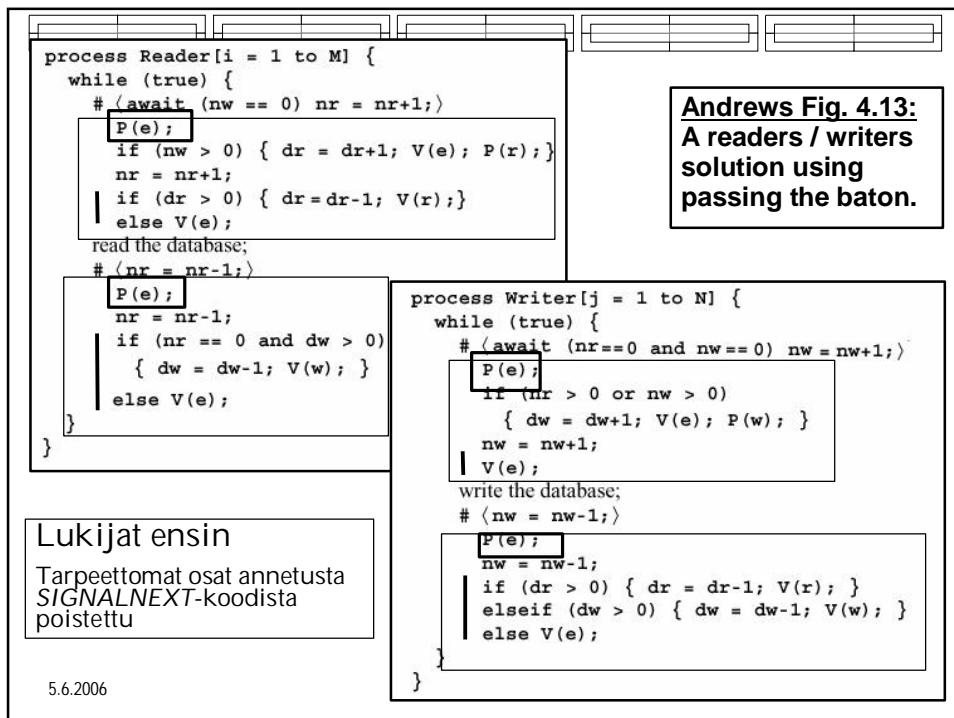
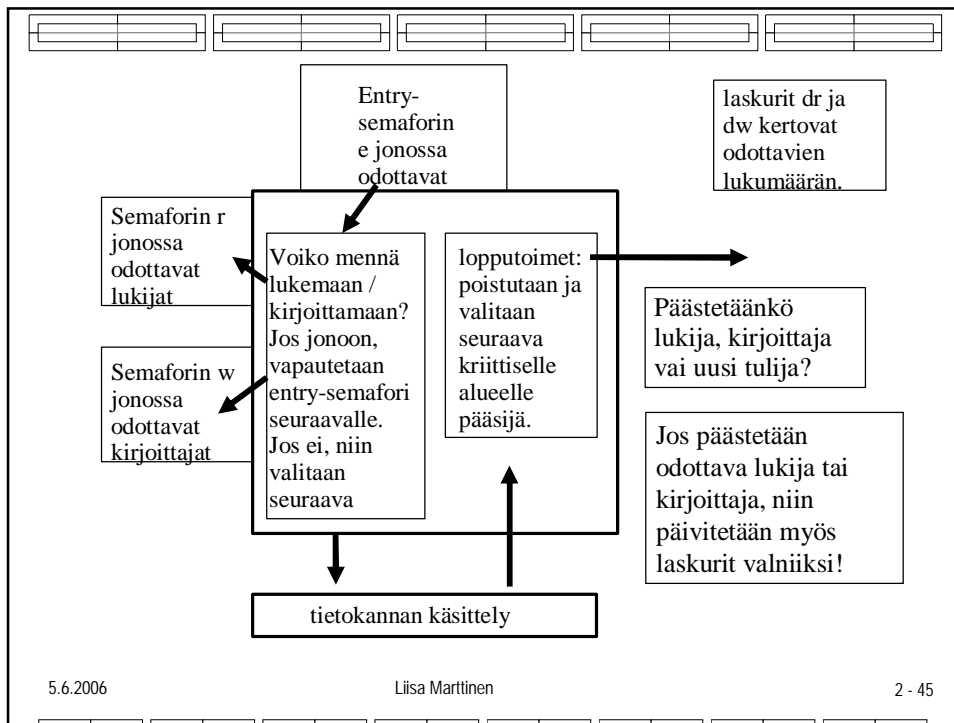
if (nw == 0 and dr > 0) {
  dr = dr -1;
  V(r);      # herätä odottava lukija, tai
}
else if (nr == 0 and nw == 0 and dw > 0) {
  dw = dw -1;
  V(w);      # herätä odottava kirjoittaja, tai
}
else
  V(e);      # päästä joku uusi etenemään
      
```

- Osa ehdoista on jo tiedossa, kun ollaan **lukijassa** tai **kirjoittajassa**
=> voidaan jättää pois eikä tarvitse enää testata!

5.6.2006

Liisa Marttinen

2 - 44



- **SIGNALNEXT** - vuoron antaminen: Kirjoittajat ensin!

- **Reader**: uusi lukija odottamaan, jos kirjoittaja odottamassa
if (nw>0 or dw>0) # DELAY
{ dr=dr+1; V(e); P(r); }

- **Writer**: herätä lukija vain, jos kirjoittaja ei odottamassa

```
if (dw>0) { # SIGNALNEXT
    dw = dw-1; V(w); # herätä kirjoittaja
}
elseif (dr>0) { # herätä lukija
    dr = dr-1; V(r);
}
else V(e); # herätä uusi
```

5.6.2006

Liisa Marttinen

2 - 47

Resurssien hallinta ja vuoronantaminen

- * Resurssi hyvin laajasti ymmärrettynä: mitä tahansa mitä prosessi joutuu odottamaan:

Pääsy kriittiselle alueelle tai tietokantaan
puskuritilaa
muistia
tulostimen käyttö

- * Vuoronantamisessa halutaan tarkemmin määrätä, mikä prosessi pääsee etenemään!

Semafori itse pitää odottajat FIFO-jonossa!
V-operaatio vapauttaa jonon ensimmäisen etenemään!
Lukijat/kirjoittajat: lupa luokkakohtainen

5.6.2006

Liisa Marttinen

2 - 48

- Toimivatko rutiinit oikein
 - | Poissulkeminen?
 - | Ei lukkiutumista (deadlock/livelock)?
 - | Ei tarpeettomia viipeitä?
 - | Lopulta onnistaa?
- sem avail ~ resurssin varaus
 - | yksi kerrallaan; jos ei heti onnistu, niin jonoon odottamaan (FCFS), josta vuorollaan saa resurssin varatuksi
- sem mutex ~ resurssin käyttöönotto
 - | yksi kerrallaan; jos ei heti onnistu, niin FCFS- jonoon odottamaan, josta vuorollaan kirjaa resurssin haltuunsa
- vapautettaessa ensin kirjataan vapaaksi ja sitten annetaan muiden varattavaksi
- Missä järjestyksessä prosessit päästetään jatkamaan?

5.6.2006

Liisa Marttinen

2 - 49

Palvelujärjestys

- Semaforin jonot aina FCFS
 - | Ongelma? Jäljellä 10 sivutilaa, eka haluaa 12, toka 5!
- Voiko semaforiin liittää prioriteetin?
 - | Jonotusjärjestys?
 - | Baton passing- tekniikka: eri luokille omat jonot
- Montako erilaista? Dynaaminen vuorottelu?
 - | Kaikki mahdolliset varattavat sivutilakoot (1... N)
- Ratkaisu: yksityiset semaforit + oma jono
 - | Kullekin prosessille oma semafori, jossa odottaa yksin
 - | Vuoron antamiseen käytettävä tietorakenne (jono) erikseen
 - | alkiossa semafori ja yleensä tietoa, jonka perusteella valitaan
 - | Vuoron antaja valitsee sopivan semaforin vuorojonosta, ja päästää liikkeelle semaforissa odottavan asiakkaan

5.6.2006

Liisa Marttinen

2 - 50

SJN: Lyhyin työ seuraavaksi

- request(time,id): # käyttöaika, prosessin tunnus
P(e);
if (!free) DELAY; # ei ole vapaa => odotus
free = false; # nyt omaan käyttöön!
SIGNAL; #
- release(): # resurssin vapautus
P(e);
free = true;
SIGNAL; # *pienin käyttöaika ensin!*

5.6.2006

Liisa Marttinen

2 - 51

• DELAY:

- | **Odottajan ID ja TIME** (suoritus aika) suoritusajan mukaan järjestettyyn jonoon (**PAIRS**) oikeaan kohtaan
- | **V(e)** eli vapauta kriittinen alue
- | jää odottamaan vuoroasi P(b[ID])
 - | Tässä tarvitaan **kullekin oma semafori**, jotta pystytään 'herättämään' oikea prosessi: **b[n] = ([n] 0)**
 - | **PAIRS-jono määrää järjestyksen**: herätetään aina jonon ensimmäinen prosessi

• SIGNALNEXT:

Request-vaihe

- | vapauta kriittinen alue **V(e)** eli päästä joku uusi Request-vaiheeseen

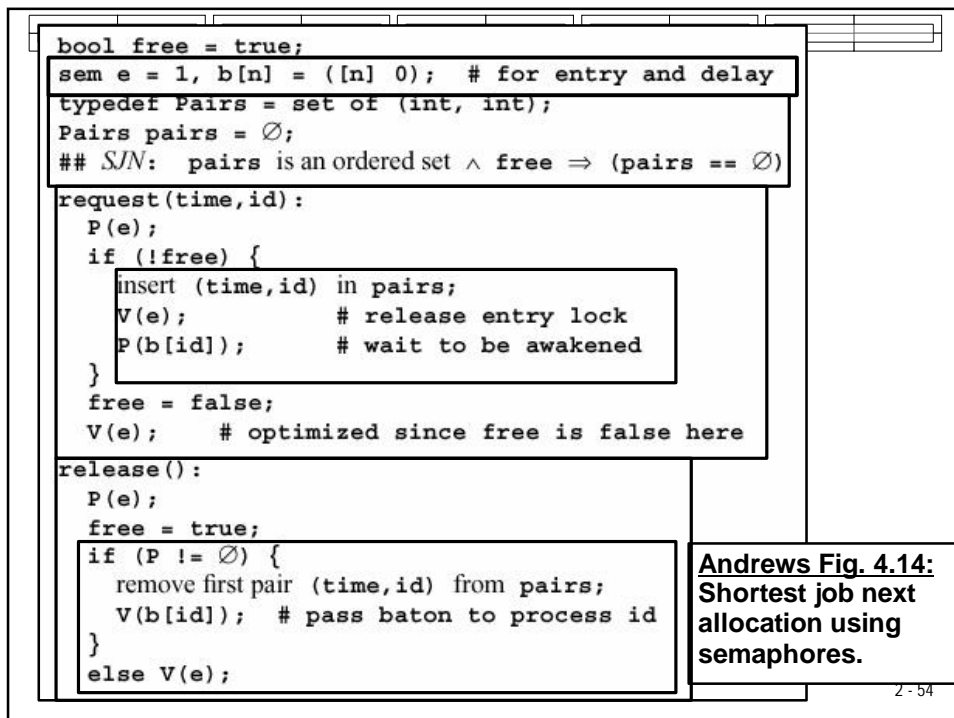
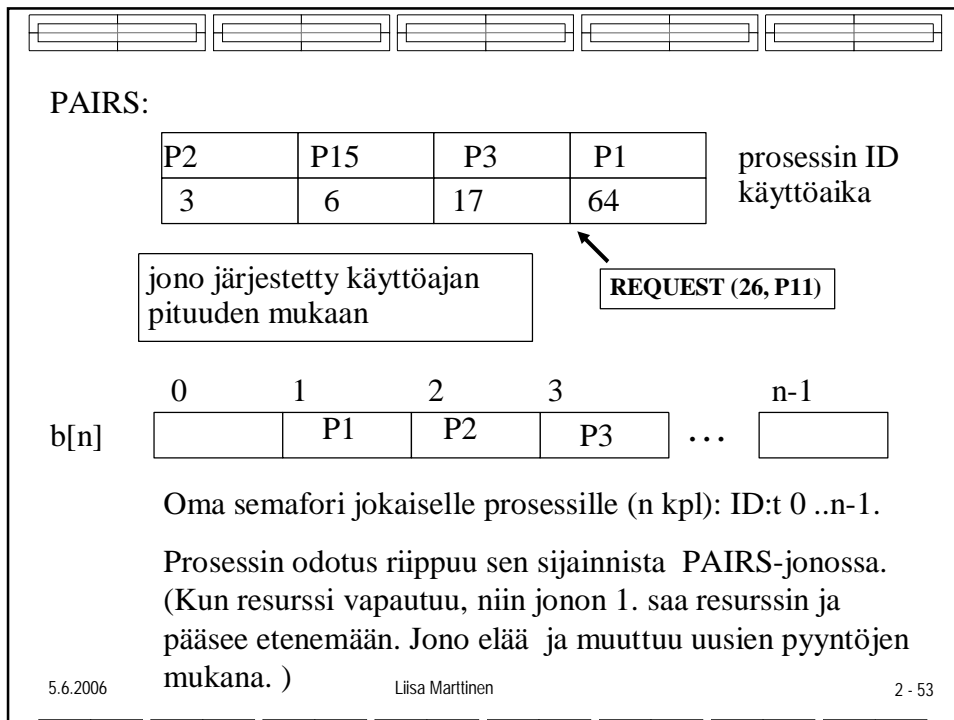
Release-vaiheen lopussa

- | Jos jonossa odottajia, niin ota jonon ensimmäisen alkio pari (**time, ID**) ja herätä prosessi ID: **V(b[ID]);**
- | muuten **V(e)**

5.6.2006

Liisa Marttinen

2 - 52



Entä, jos resurssia enemmän kuin 1 yksikkö?

- amount = montako yksikköä prosessi tarvitsee tai palauttaa
- avail = montako yksikköä on vapaana (~free)
- request:
 - | testattava, onko vapaana tarvittu määrä yksiköitä **amount <= avail**. Jos on, niin varataan, muuten talletetaan myös **amount**
 - | myös tässä voidaan vapauttaa odottavia prosesseja, jos vapaita resursseja on tarpeeksi
- release:
 - | vapautetaan jonosta ensimmäinen prosessi, jonka tarpeet pystytään tyydyttämään

5.6.2006

Liisa Marttinen

2 - 55

POSIX-kirjasto, pthread

```
# include <pthread.h>
| pthread_mutex_init(), _lock(), _trylock(), _unlock(),
| _destroy() _mutexattr_ *(), ...
| pthread_rwlock_init(), _rwlock_rdlock(), _rwlock_tryrdlock(),
| _rwlock_wrlock(), _rwlock_trywrlock(), _rwlock_unlock(),
| _rwlock_destroy(), _rwlockattr_ *(), ...

# include <semaphore.h>
| sem_init(), sem_wait(), sem_trywait(), sem_post(),
| sem_getvalue(), sem_destroy(), ...
```

C Lue man- / opastussivut

C Andrews ch 4.6, 5.5

5.6.2006

Liisa Marttinen

2 - 56

Java 1.5

• java.util.concurrent

- | **Lukkoja, atomisia muuttujia**
- | **Säikeiden synkrointiin on tarjolla neljä luokkaa: Semaphore, CyclicBarrier, CountdownLatch ja Exchanger.**
- | **semaforilta pitää saada lupa lohkon käyttöön kutsumalla acquire()-metodia, johon säie jää odottamaan, mikäli lupaa ei heti saada. Vastaavasti synkroinoitavan osuuden jälkeen lupa pitää palauttaa semaforille release()-metodilla.**

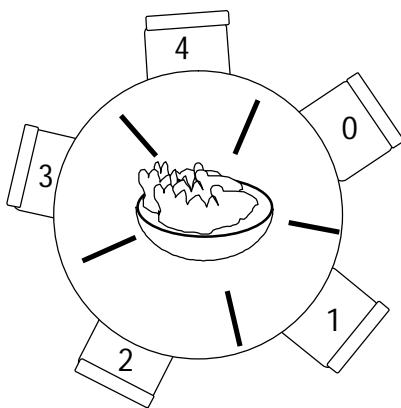
Kurssi: Ohjelmointitekniikka (Java)

5.6.2006

Liisa Marttinen

2 - 57

Aterioivat Filosofit (Dijkstra)



Filosofi:

aattelepa ite
ota kaksi haarukkaa ...
... yksi kummaltakin puolelta
syö, syö, syö spagettia
palauta haarukat

Kuinka varata haarukat ilman

- lukkiutumista
- turhaa odottamista
- nälkiintymistä

s.e. kaikki saavat olla paikalla?

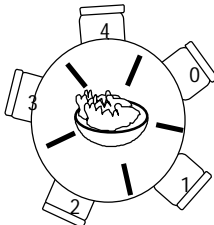
5.6.2006

Liisa Marttinen

2 - 58

Ratkaisu 1: kullekin haarukalle oma semafori

- sem fork[0..4] = (1, 1, 1, 1, 1)



```
process P[i]: repeat
    think()
    P(fork[i])
    P(fork[(i + 1) mod 5])
    eat()
    V(fork[i])
    V(fork[(i + 1) mod 5])
until false
```

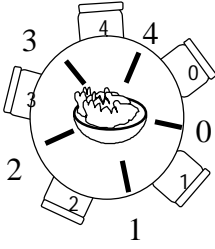
Ei OK!
Miksei?

Voi lukkiutua! Kukin ehtii ottaa yhden haarukan ja jää odottamaan toista eikä kukaan pääse syömään!

5.6.2006 2 - 59

Ratkaisu 2: vain yksi voi yrittää kerrallaan

- sem turn = 1 # säätelee yritysvuoroja



```
process P[i]: repeat
    think()
    P(turn)
    P(fork[i])
    P(fork[(i + 1) mod 5])
    V(turn)
    eat()
    V(fork[i])
    V(fork[(i + 1) mod 5])
until false
```

Ei ihanOK!
Miksei?

Turhaa odotusta!

5.6.2006 2 - 60

Turhaa odotusta!

- Oletetaan, että ensin kaikki ajattelevat ja sitten haluavat syömään esim. järjestyksessä
fil0, fil1, fil2, fil3, fil4
- fil0 saa haarukat ja pääsee syömään, fil1 varaa itselleen varausvuoron ja jää odottamaan haarukkaa
- ja muut sitten vain odottavat, ensin varausvuoroa ja haarukoita
- fil2 joutuu ihan turhaan odottamaan, haarukat olisivat vapaina, mutta niitä ei pääse varaamaan!
- Mutta kukin pääsee varaamaan ja aikanaan syömään => ei ole nälkiintymistä!
 - | **varaussemafori turn pitää varaukset FIFO-jonossa**

5.6.2006

Liisa Marttinen

2 - 61

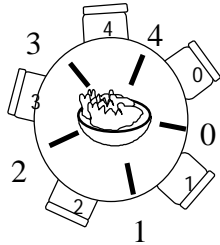
Ratkaisu 3: OK

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
    while (true) {
        P(fork[i]); P(fork[i+1]); # get left fork then right
        eat;
        V(fork[i]); V(fork[i+1]);
        think;
    }
}
process Philosopher[4] {
    while (true) {
        P(fork[0]); P(fork[4]); # get right fork then left
        eat;
        V(fork[0]); V(fork[4]);
        think;
    }
}
```

Andrews Fig. 4.7

5.6.2006

Liisa Marttinen



Entä onko turhaa odotusta?

Korkeintaan 4 filosofia saa haarukan käteensä:

fil0 haarukan 0, fil1 haarukan 1, fil2 haarukan 2 ja joko fil3 tai fil4 haarukan 3 riippuen siitä kumpi ensiksi ehtii.

Haarukka 4 jää aina vapaaksi ja sen saa fil0, joka siis pääsee syömään ja aikanaan vapauttaa hallussaan olevat haarukat ja päästää muut syömään.

Näin estetään lukkiutuminen!

Ratkaisu ei myöskään aiheuta nälkiintymistä. Suosii tosin fil0:aa, mutta vain tässä erikoistilanteessa.

5.6.2006

Liisa Marttinen

2 - 63

Ratkaisu 4: OK, mutta... vrt. 2

```
sem fork[5] = ([5] 1);
sem room = 4;

process Filosoffer[i=0 to 4]
{
  while (true) {
    think();
    P(room);
    P(fork[i]);
    P(fork[(i+1) mod 5]);
    eat();
    V(fork[i]);
    V(fork[(i+1) mod 5]);
    V(room);
  }
}
```

Estää lukkiutumisen!

Ei turhaa odotusta.

Saavatko kaikki olla paikalla?

Stallings Fig. 6.12

5.6.2006

Liisa Marttinen

Ratkaisu 5: OK? Näлкиintyminen?

```
#define N      5          /* number of philosophers */
#define LEFT  (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N   /* number of i's right neighbor */
#define THINKING 0      /* philosopher is thinking */
#define HUNGRY  1       /* philosopher is trying to get forks */
#define EATING  2       /* philosopher is eating */
typedef int semaphore; /* semaphores are a special kind of int */
int state[N];          /* array to keep track of everyone's state */
semaphore mutex = 1;  /* mutual exclusion for critical regions */
semaphore s[N];       /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {      /* repeat forever */
        think();        /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat();          /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks back on table */
    }
}
```

3.6.2000

Elisa Marttinen

Tanenbaum Fig. 2.33

```
void take_forks(int i) /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);      /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i);           /* try to acquire 2 forks */
    up(&mutex);        /* exit critical region */
    down(&s[i]);        /* block if forks were not acquired */
}

void put_forks(i)     /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);      /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);        /* see if left neighbor can now eat */
    test(RIGHT);       /* see if right neighbor can now eat */
    up(&mutex);        /* exit critical region */
}

void test(i)          /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Huom: **down()** = P(), **up()** = V()

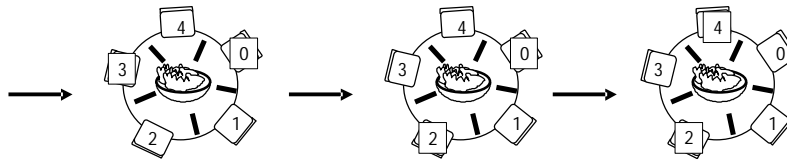
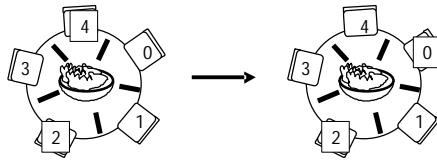
3.6.2000

Elisa Marttinen

Tanenbaum Fig. 2.33

Ratkaisu 5: "Tapettaisko filosofi 1?"

| EATING | HUNGRY |
|--------|--------|
| 4 2 | 0 1 3 |
| 2 0 | 1 3 4 |
| 3 0 | 1 2 4 |
| 0 2 | 1 3 4 |
| 4 2 | 0 1 3 |



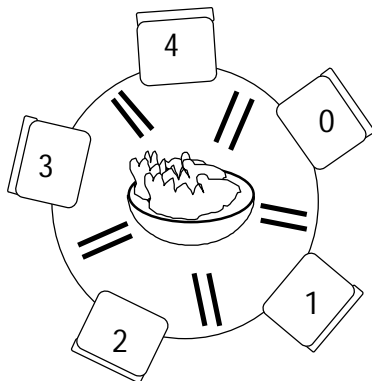
5.6.2006

Liisa Marttinen

2 - 67

Ratkaisu 6: OK, ei yhteisiä resursseja

Ostakaa 5 haarukkaa lisää!



Filosofi[i]:

aatteleppa ite
 ota kaksi haarukkaa
 ...yksi molemmilta puolilta
 syö, syö, syö spagettia
 palauta haarukat

5.6.2006

Liisa Marttinen

2 - 68