Ž Semaforit ja rinnakkaisuuden hallinta

Esimerkkejä semaforin käytöstä:
Tuottajat ja kuluttajat
Lukijat ja kirjoittajat
Resurssien hallinta, vuoron antaminen

Andrews 4.2, 4.4-4.6

Tuottajat ja kuluttajat

Andrews: ss.158-160


```
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)

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.

Sem empty = 1, full = 0;

Onko aloitus kunnossa?

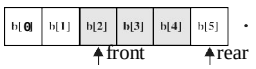
Tuottaja i
.....
.....
P(empty);
buf=data;
V(full);
.....

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?

N:n alkion puskuri; yksi tuottaja, yksi kuluttaja




```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;
```

Toimiiko jos usea tuottaja ja kuluttaja?

```
process Producer {
  while (true) {
    ...
    produce message data
    P(empty);
    buf[rear] = data;
    rear = (rear+1) % n;
    V(full);
  }
}
```

```
process Consumer {
  while (true) {
    fetch and consume
    P(full);
    result = buf[front];
    front = (front+1) % n;
    V(empty);
    ...
  }
}
```

Andrews Fig. 4.4: Bounded buffer using semaphores.

```
typeT buf[n];       /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;   /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Tarvitaan keskinäistä poissulkemista!

Andrews Fig. 4.5: **Multiple producers and consumers using semaphores.**

Entä, jos vain yksi mutex?

Lukijat ja kirjoittajat

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 yhtäaikaa
  - Kirjoittajat (writers)
    - lukevat ja muuttavat
    - vain yksi kerrallaan saa olla muuttamassa tietokantaa
- Luokka aktiivinen vain, jos toinen luokka passiivinen

Ratkaisu 1: R/W poissulkemisongelmana

- Yksinkertaistettu ongelma, yksink. ratkaisu
  - **Poissulkeminen kaikille muille => semafori rw**
  - **Andrews Fig 4.8**
- Salli lukijoiden toimia rinnakkain
  - **Luokan poissulkeminen => semafori rw**
  - **Eka lukija varaa DB:n lukijaluokalle, viimeinen lukija vapauttaa DB:n**
  - **Kuka on eka / viimeinen? => laskuri nr**
  - **"testaa, varaa / vapauta" => atominen: < ... >**
  - **Andrews Fig 4.9 and Fig 4.10**

Andrews: ss. 167-169


```
sem rw = 1

process Lukija {
while (true) {
    ....
    P(rw);
    Lue tietokannasta;
    V(rw);
  }
}
```

```
process Kirjoittaja{
while (true) {
    .....
    P(rw);
    Kirjoita tietokantaan;
    V(rw);
  }
}
```

Vain yksi tietokannan käsittelijä kerrallaan!
- yksi kirjoittaja **OK!**
- yksi lukija **TURHA RAJOITUS!**


```
sem rw = 1;
process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw);    # grab exclusive access lock
    read the database;
    V(rw);    # release the lock
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);    # grab exclusive access lock
    write the database;
    V(rw);    # release the lock
  }
}
```

Andrews Fig. 4.8: **An overconstrained solution.**

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

1. lukija varaa tietokannan ja viimeinen lukija vapauttaa sen

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

```
sem mutex =1;
```

```
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 lukija, muita ei
                          # tällä kertaa ole!
    V(mutex);
  }
}
```



```
int nr = 0;       # number of active readers
sem rw = 1;       # lock for reader/writer exclusion
process Reader[i = 1 to M] {
  while (true) {
    ...
    ⟨ nr = nr+1;
      if (nr == 1) P(rw);  # if first, get lock
    ⟩
    read the database;
    ⟨ nr = nr-1;
      if (nr == 0) V(rw);  # if last, release lock
    ⟩
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

Andrews Fig. 4.9: Outline of readers and writers solution.


```
int nr = 0;       # number of active readers
sem rw = 1;       # lock for access to the database
sem mutexR = 1;   # lock for reader access to nr
process Reader[i = 1 to m] {
  while (true) {
    ...
    P(mutexR);
    nr = nr+1;
    if (nr == 1) P(rw);  # if first, get lock
    V(mutexR);
    read the database;
    P(mutexR);
    nr = nr-1;
    if (nr == 0) V(rw);  # if last, release lock
    V(mutexR);
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

RATKAISUN REILUUS?

Andrews Fig. 4.10: Readers and writers exclusion using semaphores.

Ratkaisu 2: R/W ehtosynkronointiongelmana

- 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
    - 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)


```
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.

- Jaettu binääriarvoinen semafori
  - **Liitä kuhunkin vahtiin (ehtoon) semafori ja laskuri :**
    sem e, r, w;
  - **Vain yksi semafori kerrallaan 'auki' (0 <= (e+r+w) <= 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*

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 ð omat laskurit: dw,dr**
- Vuorojen toteutus rutiinissa *SIGNALNEXT*
  - **Kriittinen alue vapautuu ðjoku muu saa jatkaa, kuka?**
    - odottava lukija, odottava kirjoittaja, kokonaan uusi tulija
  - SIGNALNEXT **on sama kuin kirjassa käytetty** SIGNAL
    - jotta ei sekoittuisi monitorin signal-operaatioon
- Andrews Fig. 4.12

- process Reader {
  while (true) {
  # <await (nw == 0) nr = nr + 1;>

```
P(e);  #varmistaa atomisuuden
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;
```

  }
  }

- process Writer {
  while (true) {
  # <await (nw == 0) nw = nw + 1;>

```
P(e);  #varmistaa atomisuuden
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;
```

  }
  }


```
int nr = 0,    ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,     # controls entry to critical sections
    r = 0,     # used to delay readers
    w = 0;     # used to delay writers
               # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers
```

Andrews Fig. 4.12:
Outline of readers and writers with passing the baton.

```
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0)
        { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      SIGNAL;
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      SIGNAL;
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr== 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0)
        { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      SIGNAL;
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      SIGNAL;
  }
}
```


- *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)

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)

- 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 ...

ð Takaa ettei kukaan pääse 'etuilemaan'!

- **Ehdot taatusti voimassa, kun jonottaja saa prosessorin**
- *SIGNALNEXT* **aktivoi vain yhden prosessin, kun ehto tuli todeksi**
- **FCFS**

- *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!

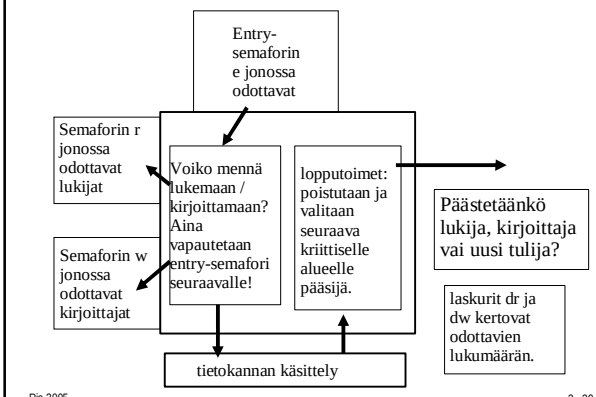

```
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r);}
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r);}
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0)
        { dw = dw-1; V(w); }
      else V(e);
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr==0 and nw==0) nw = nw+1;⟩
      P(e);
      if (dr > 0 or nw > 0)
        { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

Andrews Fig. 4.13:
A readers / writers solution using passing the baton.

Lukijat ensin

Tarpeettomat osat annetusta *SIGNALNEXT*-koodista poistettu



- *SIGNALNEXT* - vuoron antaminen: Kirjoittajat ensin!
- Reader: uusi lukija odottamaan, jos kirjoittaja odottamassa (rivi 5)

```
if (nw>0  or  dw>0)   # DELAY
   { dr=dr+1; V(e);  P(r); }
```

- Writer: herätä lukija vain, jos kirjoittajia ei odottamassa (rivi 13)

```
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
```

Muuta kuvaa 4.13 vastaavasti

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

Resurssien hallinta

- Kilpailua yhteisistä resursseista (objekteista)
  - **Objektit: varaus ja vapautus; odotettava jos ei saa**
  - **Vuorot: kun objekteja vapaa, kuka odottajista saa jatkaa?**
- Sisäinen kirjanpito kustakin erillisestä resurssista (private)
  - **ominaisuudet (esim. muistiosoite, koko, ...)**
  - **vapaa / varattu**
  - **Jos varattu, niin käyttäjä (esim. prosessin ID)**
- Rajapinta, API (public)
  - **pyydä (parametrit)** # parametreina esim. pyytäjä, lohkon koko,
  - **vapauta(parametrit)** # lohkojen määrä, mitkä lohkot ,..

Operaatioiden raaka runko

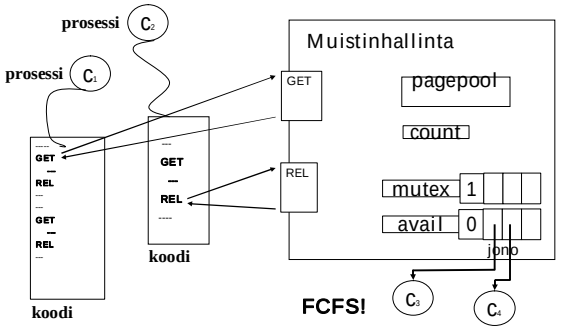
- pyydä(parametrit)

  < **await** (pyyntöön voi suostua)
  anna resurssi, merkitse varatuksi **>**
- vapauta(parametrit)

  < palauta resurssi kirjanpitoon **>**
- Vrt. kriittisen alueen entry protocol, exit protocol

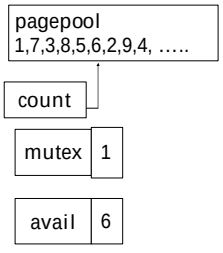
Muistinhallinta (3)

pyynnöt sivu kerrallaan



Poissulkemisongelmana

```
int count=MAX;
sem mutex=1, avail=MAX;
function GET(): returns addr {
  P(avail) ;    // await
  P(mutex);     // anna, merkitse
  get = pagepool[count];
  count  = count-1;
  V(mutex);
}
procedure REL(addr freepage) {
  P(mutex);     // palauta
  count = count+1;
  pagepool[count] = freepage;
  V(mutex);
  V(avail) ; // pyyntöön voi suostua
}
```



- 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?**

VÄÄRIN!

```
int count=MAX;
sem mutex=1, avail=MAX;
function GET(): returns addr {
  P(mutex);
  P(avail) ;
  get = pagepool[count];
  count  = count-1;
  V(mutex);
}

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

pagepool
1,7,3,8,5,6,2,9,…..

count

mutex | 0

avail | 6

Resurssien hallinta, Yleinen ratkaisu (baton passing)

- pyydä(parametrit)
  P(mutex); # poissulkeminen
  if (pyyntöön ei voi suostua) *DELAY*; # odota semaforissa
  anna resurssi;
  SIGNALNEXT;
- vapauta(parametrit)
  P(mutex);
  palauta resurssi;
  SIGNALNEXT;
- *DELAY ~*
  - **V(mutex), P(odotussemafori)**
- *SIGNALNEXT ~*
  - **V(odotussemafori) else V(mutex)**

DELAY:
Älä jätä prosessia
Blocked-tilaan
tärkeä semafori kiinni!

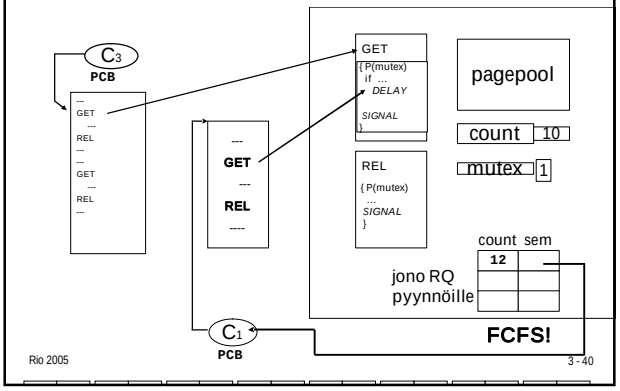
SIGNALNEXT:
Herätä odottaja ja jätä kriittinen alue kiinni (baton passing).
Vapauta semafori , jos ei ole odottajia!

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**

Muistinhallinta (4)

pyynnöt useita sivuja kerralla



RQ

count	sem
2	P1
3	P2
9	P3
2	P4

tail →

Oletetaan:
pagepool on tyhjä ja
P1: GET 2
P2: GET 3
P3: GET 9
P4: GET 2

P0:
REL(1,4,7,12,3,5,8,11,9)

vapauttaa yhdeksän sivutilaa => count = 9

Mitkä prosessit päästetään etenemään?
P1 ja P2? Vai P3? Entä P4?


```
procedure GET (nbr_of_units) {
   P(mutex);
   if  (request can not be satisfied) { # DELAY
       RQ[tail].count = nbr_of_units;
       V(mutex);
       P(RQ[tail].sem);
   }
   take nbr_of_units for this process;
   if (! empty(RQ) and RQ[i].count < count)
      V(RQ[i].sem);    # SIGNALNEXT
   else
      V(mutex);
}
```

vapaana riittävästi muistia seuraavalle / jollekin odottajalle!

```
procedure REL (list_of_units) {
    P(mutex);
    return units into pagepool;
    if (! empty(RQ) and RQ[i].count < count)
        V(RQ[i].sem);     # SIGNALNEXT
    else
        V(mutex);
}
```

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!

- DELAY:
  - **Odottajan ID ja TIME** (suoritusaika) 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:

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

Release-vaiheen lopussa
- Jos jonossa odottajia, niin ota jonon ensimmäisen alkiopari **(time, ID)** ja herätä prosessi ID: **V(b[ID];**
- muuten **V(e)**

PAIRS:

P2	P15	P3	P1	prosessin ID
3	6	17	64	käyttöaika

jono järjestetty käyttöajan pituuden mukaan

REQUEST (26, P11)

	0	1	2	3		n-1
b[n]		P1	P2	P3	…	

Oma semafori jokaiselle prosessille (n kpl): ID:t 0 ..n-1.

Prosessin odotus riippuu sen sijainnista PAIRS-jonossa. (Kun resurssi vapautuu, niin jonon 1. saa resurssin ja pääsee etenemään. Jono elää ja muuttuu uusien pyyntöjen mukana.)


```
bool free = true;
sem e = 1, b[n] = ([n] 0);  # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN:  pairs is an ordered set ∧ free ⇒ (pairs == ∅)
request(time,id):
  P(e);
  if (!free) {
    insert (time,id) in pairs;
    V(e);           # release entry lock
    P(b[id]);       # wait to be awakened
  }
  free = false;
  V(e);    # optimized since free is false here
release():
  P(e);
  free = true;
  if (P != ∅) {
    remove first pair (time,id) from pairs;
    V(b[id]);  # pass baton to process id
  }
  else V(e);
```

Andrews Fig. 4.14: Shortest job next allocation using semaphores.

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

POSIX-kirjasto, pthread

Kurssi: Verkkosovellusten toteuttaminen

include <pthread.h>
- **pthread_mutex_init(), _lock(), _trylock(), _unlock(), _destroy() _mutexatrr_*(), …**
- **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(), …**

- **Lue man- / opastussivut**
- **Andrews ch 4.6, 5.5**

Java 1.5

- java.util.concurrent
  - **Lukkoja, atomisia muuttujia**
  - **Säikeiden synkroinointiin 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)

Kertauskysymyksiä?