OSA I:

Yhteisten muuttujien käyttö

Prosessit samassa koneessa

Sisältöä

❷ Poissulkeminen ja synkronointi

❸ Semaforit ja rinnakkaisuuden hallinta

❹ Lukkiutuminen

❺ Monitorit

❷ Poissulkeminen ja synkronointi

Rinnakkaiset, atomiset operaatiot
Spin Locks
Semaforit
Synkronointi

Andrews 2.1-2.5, 3.1-3.2, 4.1-4.2, 6.3 ja Stallings 5.1,5.3-5.4

Rinnakkaiset, atomiset operaatiot

Andrews 3.1. (s. 94-95)

- **Kriittinen alue, useita käsittelijöitä**
  - **Vain yksi saa edetä alueella kerrallaan**

```
char out, in;
procedure echo {
  input (in, keyboard);
  out = in;
  output (out, display);
 }
process P[i=1 to 2] {
  . . .
  echo();
  . . .
}
```

prosessi P[1]	*prosessi P[2]*
...	*...*
input (in,..);	*...*
	input(in,..);
out = in;	*out = in;*
output (out,..);	
...	*output(out,..);*

Poissulkeminen

- **Ohjelman perusmalli**

```
process CS[i=1 to n] {  # n rinnakkaista prosessia
   while (true) {
      Entry protocol      # varaa
      < kriittinen alue > # käytä (exclusive!)
      Exit protocol       # vapauta
      ei kriittistä koodia;
   }
}
```

- **Ohjelmoi kriittiset alueet lyhyiksi**

Halutut ominaisuudet

Poissulkeminen

- Vain yksi prosessi kerrallaan suorituksessa kriittisellä alueella

Ei lukkiutumisvaraa, ei 'elohiirtä'

- Jos useita sisäänpyrkijöitä, jotain onnistaa

`deadlock`
`livelock`

Ei tarpeettomia viipeitä, ei nälkiintymistä

- Jos alue vapaa, pääsy sallittava

Lopulta onnistaa

- Kukaan ei joudu odottamaan ikuisesti

3 safety properties + liveness property

Spin Locks, Busy Waiting

Andrews 3.2 (s. 97-101)

Lukkomuuttujat, Spin Locks

- **Boolean-muuttuja lock**
  - `lock==true` **kriittinen alue varattu**
  - `lock==false` **kriittinen alue vapaa**
- **Entry protocol**

```
while (lock) ;                  # aktiivinen odotus, "pörrää"
  # check again if (!lock)      # busy loop
lock=true;
```

- **Exit protocol**

```
lock=false;
```

Toteutus symbolisella konekielellä

- **muuttuja L: L==1, jos kriittinen alue vapaa**
  L==0, jos kriittinen alue varattu

	L == 0	
LOOP:	LOAD R1,L	LOOP: LOAD R1,L
	JZER R1,LOOP	JZER R1,LOOP
	LOAD R1,=0	LOAD R1,=0
	STORE R1,L	STORE R1,L
	< kriittinen alue >	< kriittinen alue >
	LOAD R1,=1	LOAD R1,=1
	STORE R1,L	STORE R1,L

OK? Testaus ja asetus atomisesti!

Test-and-Set-käsky

- **Oma konekielen käsky**
  - atominen laitetason suoritus (execute-vaihe)
  - argumentti: lukkomuuttuja
  - boolean paluuarvo
- **TS L**
  - merkitys: < temp=L; L=0; if (temp==1) jump *+2; >
- **Käyttö:**

```
LOOP:   TS    L
        JUMP  LOOP
        .......
```

```
bool TS(bool lock) {
  < bool initial = lock;   /* arvo talteen */
    lock = true;           /* lukitse lukko */
    return initial; >      /* palauta lukon arvo */
  }
```

Andrews s. 99

Test and Set –toiminto
lausekielellä esitettynä

Test-and-Set ja TT-K91?

TS Ri,L < Ri ← mem[L]
mem[L] ← 0
if (Ri==1) jump *+2 >

entry

```
LOOP: TS    R1,L          # L: 1(vapaa), 0(varattu)
      JUMP LOOP
      < kriittinen alue >
```

exit

```
      LOAD  R1,=1
      STORE R1,L          # L ← 1
```

OK? Moniprosessorijärjestelmä? Keskeytys?

Toteutuvatko vaatimukset?

Poissulkeminen ? OK!

- Vain yksi prosessi kerrallaan suorituksessa kriittisellä alueella

Ei lukkiutumisvaraa, ei 'elohiirtä' ? OK!

- Jos useita sisäänpyrkijöitä, jotain onnistaa

Ei turhia viipeitä, ei nälkiintymistä ? OK!

- Jos alue vapaa, pääsy sallittava

Lopulta onnistaa ? EI VÄLTTÄMÄTTÄ

- Kukaan ei joudu odottamaan ikuisesti

Ongelmana suorituskyky, jos monta prosessia kilpailemassa ja koko ajan tutkimassa ja **kirjoittamassa** lukkomuuttujaan

- **Rasittaa muistia ja muistiväylää**
- **Kirjoittaminen häiritsee välimuistin toimintaa**
- **=> Test-and-Test-and-Set = testatan ensin, onko lukko vapaa**

Lukkomuuttuja ja lukko-operaatiot

- **Muuttujan tyyppi lock**
- **Operaatiot**
  - **esittely, alkuarvo**
  - **LOCK(x)**
  - **UNLOCK(x)**

```
lock x=1;          # vapaa

loop
   test=TS(x);     # aseta x = 0
until(test==1); # oliko vapaa?

x=1;
```

[Linux kernel: spinlock_lock(x), spinlock_unlock(x)]
[vrt. kirja CSEnter, CSExit]

Poissulkeminen lukkomuuttujaa käyttäen

```
lock L=1;       # käytä lukitsemaan tiettyä kriittistä aluetta
                # sopimus prosessien kesken!

process CS [i=1 to N] {      # n rinnakkaista!
   while (true){
        ei kriittistä koodia;
        LOCK(L);             # varaa
        < kriittinen alue >  # käytä (exclusive!)
        UNLOCK(L);           # vapauta
        ei kriittistä koodia;
   }
}
```

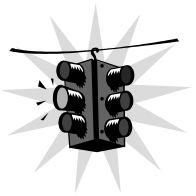
Tulkinta: `L=1` ⇨ **yhdellä lupa edetä**

Aktiivinen vs. passiivinen odotus

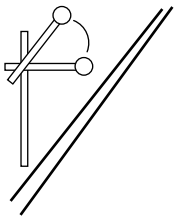
- **busy-wait: turhaa laitteiston käyttöä!**
- **kilpailu**
  - **useita prosesseja (käyttävät yhteistä prosessoria), jotka haluavat päästä kriittiselle alueelle**
- **livelock**
  - **pienen prioriteetin prosessi kriittisellä alueella**
  - **suuren prioriteetin prosessi pörrää LOCK-operaatiossa**
- **vaihtoehto**
  - **jos pääsy ei sallittu, odota Blocked-tilassa**

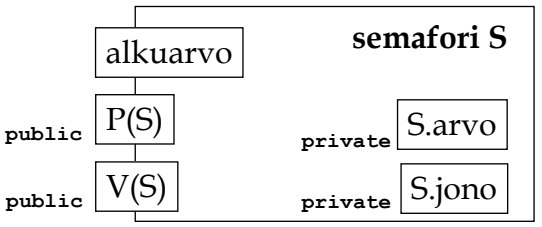
Semaforit

Andrews 4.1 (s. 153-155), 6.3 (s. 276-279)



Semaforit





- **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 odotusjonossa, päästä joku etenemään muuten vapauta kriittinen alue**
- **atomisia**

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 ⇔ vapaa / avoinna / etenemislupa
  - 0 ⇔ varattu / suljettu / eteneminen kielletty
- **S.jono**
  - Etenemislupaa Blocked-tilassa odottavat prosessit
  - Toteutus: PCB:t linkitetyssä listassa (=jono)

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

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()
```


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)**

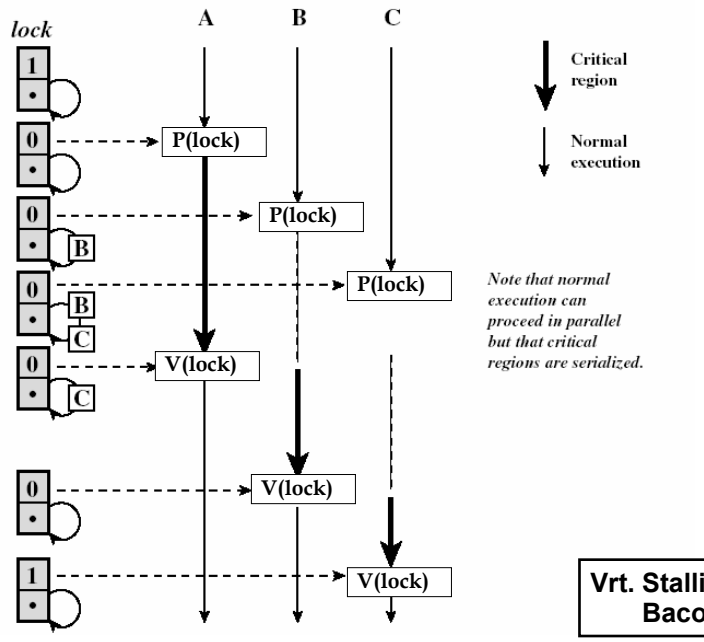
```
  S.arvo = S.arvo + 1
  if  (S.arvo <= 0)
     siirrä S.jonon ensimmäinen prosessi
     Ready-jonoon
call Vuorottaja()
```

- **Miten ohjelmoijan huomioitava tämä toteutusero?**

Poissulkeminen semaforia käyttäen

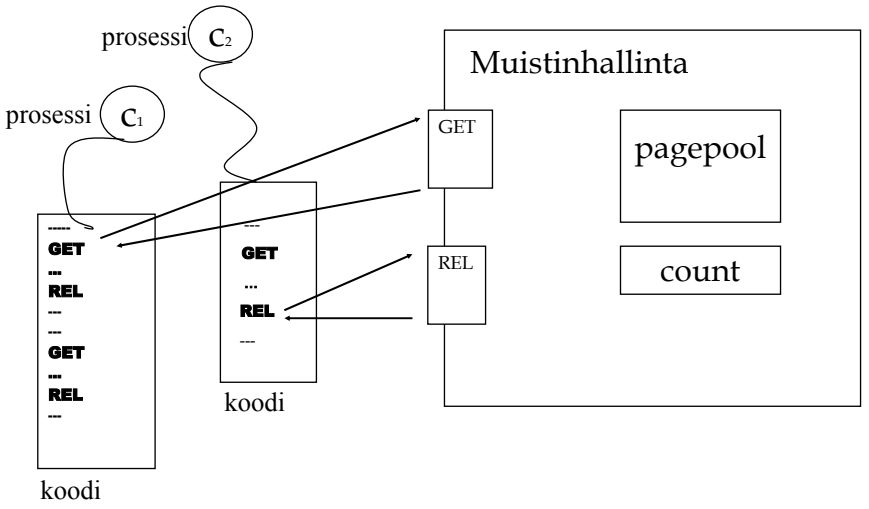
```
sem mutex=1;                    # vain perinteinen muuttujan nimi

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

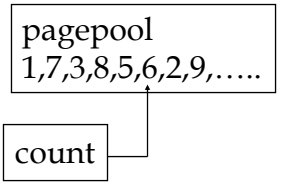


Vrt. Stallings Fig. 5.10
Bacon Fig 9.19

Esimerkki: Muistinhallinta (1)

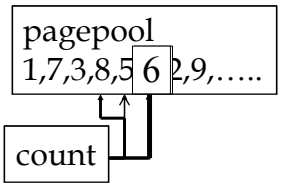



```
addr pagepool[1:MAX];      # käyttö pinona
int count=MAX;

function GET(): returns addr {
    get = pagepool[count];
    count = count-1;
}

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



Toimiikos toi?

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




```
function GET: returns addr
{ get = pagepool[count];
  count = count-1
}
```

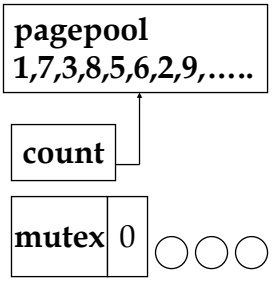
GET==6

```
function GET: returns addr
{ get = pagepool[count];
  count = count-1
}
```

GET==6

Muistinhallinta (1): poissulkeminen

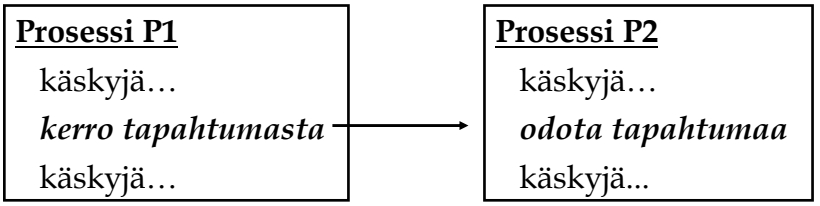
```
addr pagepool[1:MAX];
int count=MAX;
sem mutex=1;

function GET(): returns addr {
  P(mutex);
    get = pagepool[count];
    count = count-1;
  V(mutex);
}

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



Synkronointi

Andrews 4.2 (s. 156-164)

Synkronointi



- **Tapahtuma: *"mikä tahansa kiinnostava"***
  puskuri_täynnä, io_valmis, laskenta valmis, kriittinen alue vapaa
- **(Looginen) tapahtuma ⇔ semaforimuuttuja**
- **Kumpi ehtii ensin synkronointikohtaan?**
  ⇨ tarvitsee paikan missä odottaa (~jonottaa)

Synkronointi semaforia käyttäen

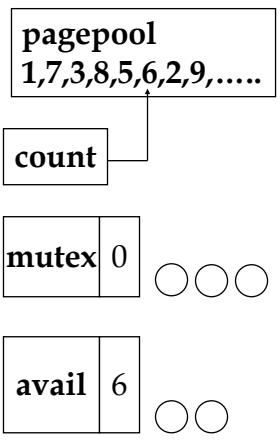
- **sem A_Ready = 0;**
  - **0 ⇔ ”ei ole tapahtunut”, 1 ⇔ ”on tapahtunut”**

Tuottaja	Kuluttaja
tuota A…	…
V(A_ready);	**P(A_ready);**
…	kuluta A…

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

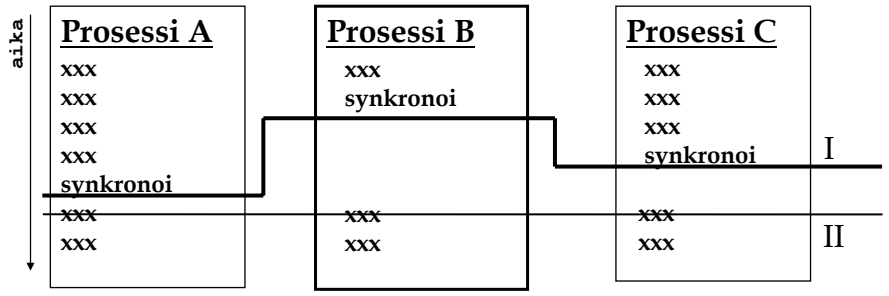
Muistinhallinta (2): synkronointi

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

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

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

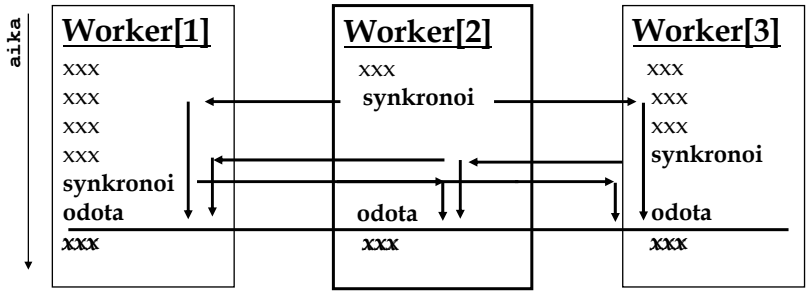


Puomisynkronointi (barrier)

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




```
process Worker [i = 1 to N] {
  while (...) {
    käsittele vaihe x:n alku ;
    toimita synkronoititieto kaikille muille;
    odota synkronoititietoa kaikilta muilta;
    käsittele vaihe x:n loppu;
  }
}
```



Puomisynkronointi

```
sem arrive1 = 0, arrive2 = 0;

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

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

Huomaa järjestys!

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?

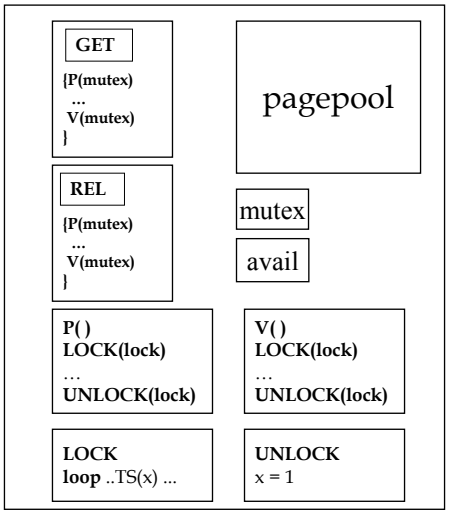
Rinnakkaisuuden hallinta

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



Kertauskysymyksiä?