OSA I:

Yhteisten muuttujien käyttö

Prosessit samassa koneessa

Sisältöä

- Poissulkeminen ja synkronointi
- Semaforit ja rinnakkaisuuden hallinta
- Lukkiutuminen
- Monitorit

V 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

Atomisuus (atomic action)

- alkutila →  → tila muutoksen jälkeen

tilamuutos
välitiloja ei näy

- yksittäinen konekäsky on atominen
- usean konekäskyn vaatima muutos = kriittinen alue (critical section) on tehtävä 'atomiseksi' keskinäisellä poissulkemisella (mutual exclusion)
  - muut prosessit eivät pääse käsittelemään tilamuutoksen muuttujia eli suorittamaan omaa kriittista aluettaan

Kriittinen (koodi)alue (critical section)

int buf[n]; buf Yhteinen muistialue

```
process kertoma (int arvo) {
    int apu = 1,  i =2;
    while (i<= arvo) {
        apu = i*apu;
        i++;
    }
  buf[arvo] = apu;
}
```

```
process laskija( ) {
    int apu, luku, ...;
    while (true) {
        laske jotain;
        luku=f( );  # laske luku
        apu = buf[luku];
        if (apu = = 0) ....
        else
        lasketaan taas;
            ......
    }
{
```


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) {
       ei kriittistä koodia
       Entry protocol          # varaa
       < kriittinen koodialue > # käytä (exclusive!)
       Exit protocol           # vapauta
       ei kriittistä koodia;
   }
}
```

- **Ohjelmoi kriittiset alueet lyhyiksi!**
  - Vain välttämätön

Halutut ominaisuudet

Poissulkeminen
- Vain yksi prosessi kerrallaan suorituksessa kriittisellä alueella

Ei lukkiutumisvaraa, ei 'elohiirtä'
- Jos useita sisäänpyrkijöitä, jotain onnistaa

`deadlock` `livelock`

Ei tarpeettomia viipeitä
- Jos alue vapaa, pääsy sallittava

Lopulta onnistaa, ei 'nälkiintymistä'
- Kukaan ei joudu odottamaan ikuisesti

3 safety properties + liveness property


```
bool in1= false,  in2=false;
process CS1 {
  while (true) {
   <await (!in2) in1 =true;> /*entry*/
    critical section;
    in1= false;     /*exit*/
    noncritical section;
  }
 }
process CS2 {
while (true) {
   <await (!in1) in2 =true;> /*entry*/
    critical section;
    in2= false;    /*exit*/
    noncritical section;
  }
 }
```

Kaksi prosessia

entry protocol!

exit protocol!

Entä jos prosesseja onkin N kappaletta?

Onko varattu (joku käyttää) vai vapaa (ei kukaan käytä)?

Spin Locks, Busy Waiting

Andrews 3.2 (s. 97-103)

Lukkomuuttujat, Spin Locks

- Boolean-muuttuja lock
  - **`Lock== true` kriittinen alue varattu (in1 V in2 V … V inN)**
  - **`lock==false` kriittinen alue vapaa**

- Entry protocol

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

- Exit protocol

```
lock=false;
```



```
bool lock=false;
process CS[i=1 to N] {
  while (true) {
   <await (!lock) lock =true;>  /*entry*/
    critical section;
    lock= false;      /*exit*/
    noncritical section;
   }
  }
```

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; >**
    merkitään varatuksi!
- Käyttö:

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

Jää odottamaan! *Saa edetä!*


```
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 ß mem[L]
mem[L] ß 0
if (Ri==1) jump *+2 **>**

entry

```
LOOP:  TS     R1,L   # L: 1(vapaa), 0(varattu)
       JUMP   LOOP
       < kriittinen alue > # tänne, jos R1 = 1
exit   LOAD   R1,=1
       STORE  R1,L    # L ß 1
```

OK? Moniprosessorijärjestelmä? Keskeytys?

TEST and SET

```
bool lock=false;
process CS[i=1 to N] {
  while (true) {
     while(TS(lock)) skip;
      critical section;
      lock= false;
      noncritical section;
   }
}
```


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 = testataan ensin, onko lukko vapaa**

Test and Test and Set

- kilpailu muistista (lukkomuuttuja) => huono suorituskyky
- TS kirjoittaa aina lukkomuuttujaan => välimuisti muuttuu => muiden prosessorien välimuistit eivät enää ole ajantasalla

```
while (lock) skip;  # pörrätään, kun lukko päällä (lock=true)
while (TS(lock)) {  # yritetään saada lukko itselle
      while (lock) skip;  # pörrätään, jos ei saatu (lock = true)
}
```

- nyt useinmiten vain luetaan arvoa ja harvemmin kirjoitetaan!

TEST and TEST and SET

```
bool lock=false;
process CS[i=1 to N] {
  while (true) {
     while(lock) skip;
     while(TS(lock)) {
          while(lock) skip;
     }
     critical section;
     lock= false;
     noncritical section;
  }
}
```


Lukkomuuttuja ja lukko-operaatiot

- Muuttujan tyyppi lock
- Operaatiot
  - **esittely, alkuarvo** `lock x=1;` # vapaa
  - **LOCK(x)**
    ```
    loop
      test=TS(x);   # aseta x = 0
    until(test==1);   # oliko vapaa?
    ```
  - **UNLOCK(x)** `x=1;`

[Linux kernel: spinlock_lock(x), spinlock_unlock(x)]
[vrt. kirja CSenter, CSexit]

CSenter - CSexit

<await (B) S;> toteutus:

- CSenter;
  while (!B) **{Csexit; Delay; Csenter; }**
  S;
  CSexit;

Ethernetin MAC-protokolla

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 ð** yhdellä lupa edetä

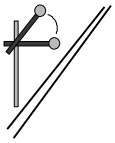
Aktiivinen vs. passiivinen odotus

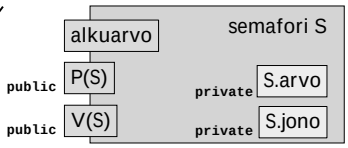
- busy-wait: turhaa laitteiston käyttöä!
- kilpailu
  - **useita prosesseja (käyttävät yhteistä prosessoria), jotka haluavat päästä kriittiselle alueelle**
- livelock
  - **pienen prioriteetin prosessi kriittisellä alueella**
  - **suuren prioriteetin prosessi pörrää LOCK-operaatiossa**
- vaihtoehto
  - **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 ó vapaa / avoinna / etenemislupa**
  - **0 ó 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)

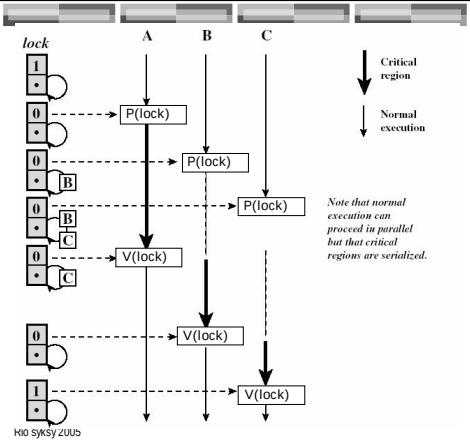
```
 S.arvo = S.arvo + 1
 if (S.arvo >= 0)
     siirrä S.jonon ensimmäinen prosessi
     Ready-jonoon
call Vuorottaja()
```

- Miten ohjelmoijan huomioitava tämä toteutusero?

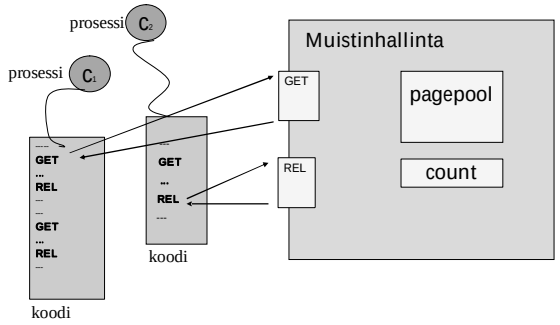
Poissulkeminen semaforia käyttäen

```
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;
   }
}
```

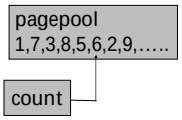
- yksi semafori kullekin erilliselle kriittiselle alueelle
- huomaa oikea alkuarvo



Esimerkki: Muistinhallinta (1)

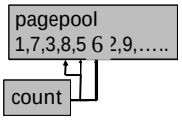



```
addr pagepool[1:MAX];     # vapaita muistitiloja ; käyttö pinona
int count=MAX;

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

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



- Toimiikos tuo?

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

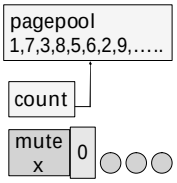


addr function GET:
{ memaddr = pagepool[count];
count = count-1
}
GET==6

addr function GET:
{ memaddr = pagepool[count];
count = count-1
}
GET==6

Muistinhallinta (1): poissulkeminen

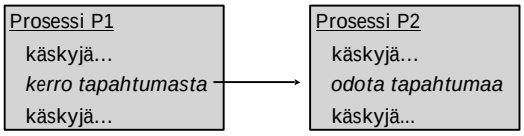
```
addr pagepool[1:MAX];
int count=MAX;
sem mutex=1;

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);
}
```



Ehtosynkronointi

Andrews 4.2 (s. 156-164)

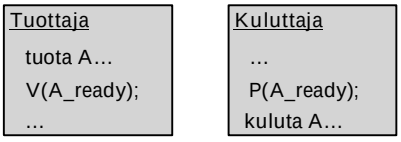
Ehtosynkronointi



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

Synkronointi semaforia käyttäen

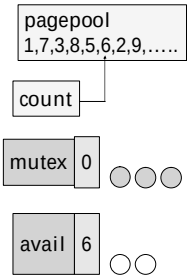
- sem A_Ready = 0;
  - **0 ó "ei ole tapahtunut", 1 ó "on tapahtunut"**



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

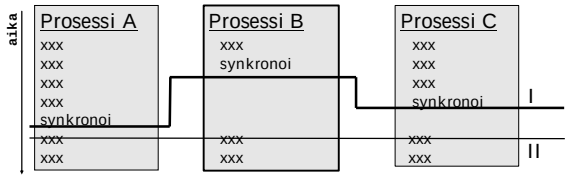
Muistinhallinta (2): synkronointi

```
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);
}
```

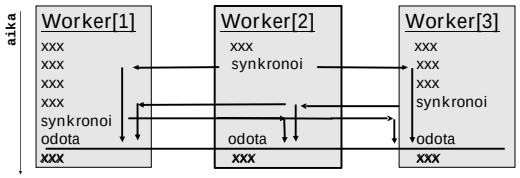


Puomisynkronointi (barrier)

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




```
process Worker [i = 1 to N] {
  while (…) {
    käsittele vaihe x:n alku ;
    toimita synkronointitieto 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!

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!!

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...
}
```


Puomisynkronointi, yleiset semaforit

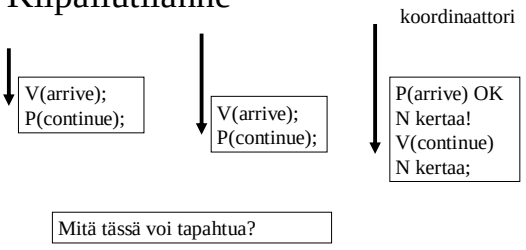
```
sem arrive=0, continue[1:N] = ( [N] 0 );
process Worker [i=1 to N] {
  ...käsittele alkuosa i...
  V(arrive);              # synkronointitapahtuma koordinaattorille
  P(continue[i]);         # odota toisia prosesseja
  ...käsittele loppuosa i...
}
process Coordinator {
  while (true) {
    count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

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
  }
}
```


Kilpailutilanne



Mitä tässä voi tapahtua?

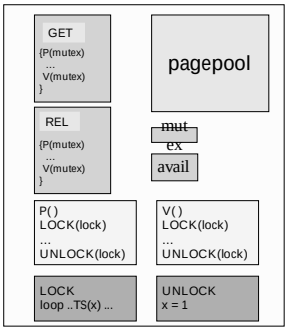
Rinnakkaisuuden hallinta

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?