⑨ Yhteenvetoa

Ongelmakenttä

- **Rinnakkaisuuden tarve**
  - Ympäristö
  - Suunnittelun yksinkertaistaminen
  - Suorituskyky
  - Luotettavuus
- **Kommunikointiin tarvitaan**
  - Yhteisiä muuttujia (data)
  - Kommunikointikanavia
    - one-to-one
    - many-to-many

Ratkottava

- **Poissulkeminen**
  - atominen käyttö
  - ei lukkiumaa
  - ei elohiirtä (livelock)
  - ei nälkiintymistä (suoritus päättyy)

- **Synkronointi**
  - yksittäiset erilliset tapahtumat, tilan muutokset
  - yhden prosessin odotus, ryhmän odotus (puomi)

⇨ oikea vuoronantaminen (esim. ei etuilua)

Mekanismit

- **Yhteinen muisti**
  - yhteiset muuttujat, yhteiset proseduurit
  - lukkomuuttujat
  - semaforit
  - monitorit
- **Hajautettu ympäristö**
  - ei yhteistä muistia
  - sanomanvälitys, send/receive (rinnakkainen toiminta)
  - etäproseduurikutsu (hetkeksi käynnistetty toiminta)
  - rendezvous (rinnakkainen toiminta, synkronointi)

Lukkomuuttujat

- **Atominen konekielen käsky**
  - test-and-set
- **Aktiivinen odotus**
  - Älä käytä, ellet tiedä miksi käytät!

Lukkomuuttujat, Spin Locks

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

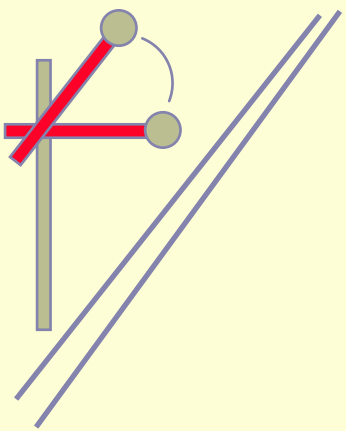
- **Entry protocol**

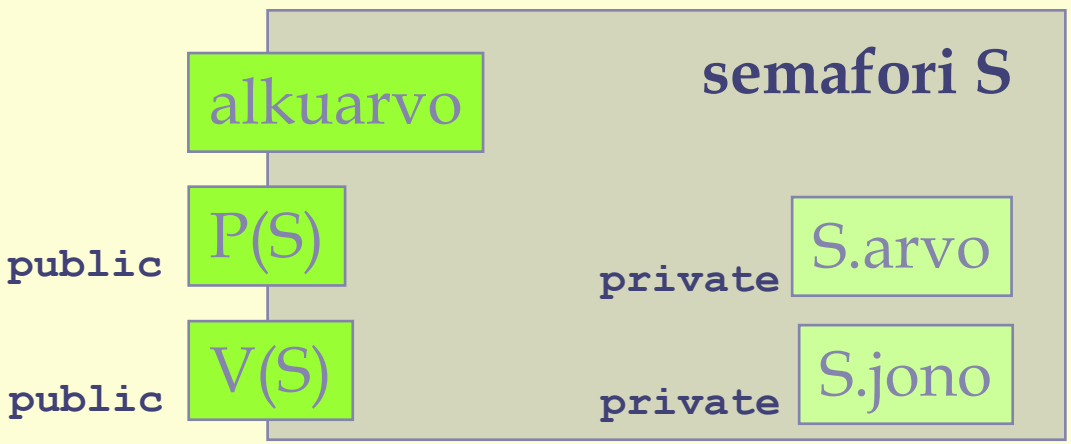
```
while (lock) ;                    # aktiivinen odotus, "pörrää"
  # check again if (!lock)        # busy loop
lock=true;
```

- **Exit protocol**

```
lock=false;
```

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**

Semaforit

- **P(sem) + V(sem): poissulkeminen, synkronointi**
- **Jaettu binäärisemafori** (split binary semaphore)
  - kaksi semaforia (tai useampi), joista vain yksi kerrallaan 1
  - kontrolli hajautettavissa useammalle prosessille
- **Baton passing** (‘viestikapulan välitys’)
  - jätä poissulkemissemafori kiinni, herätä odottaja
    ⇨ suoritettava prosessi vaihtuu, ei salli etuilua
    ⇨ herätetty herättää seuraavan, jne.
- **Oma jono ja yksityiset semaforit**
  - suoritusvuorojen säätely (ei aina FCFS)
  - jonotus jonkun prioriteetin perusteella (tarve järjestää)

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

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?

```
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);
    ...
  }
}
```

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

```
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);
  }
}
```

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

```
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;
      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);
  }
}
```

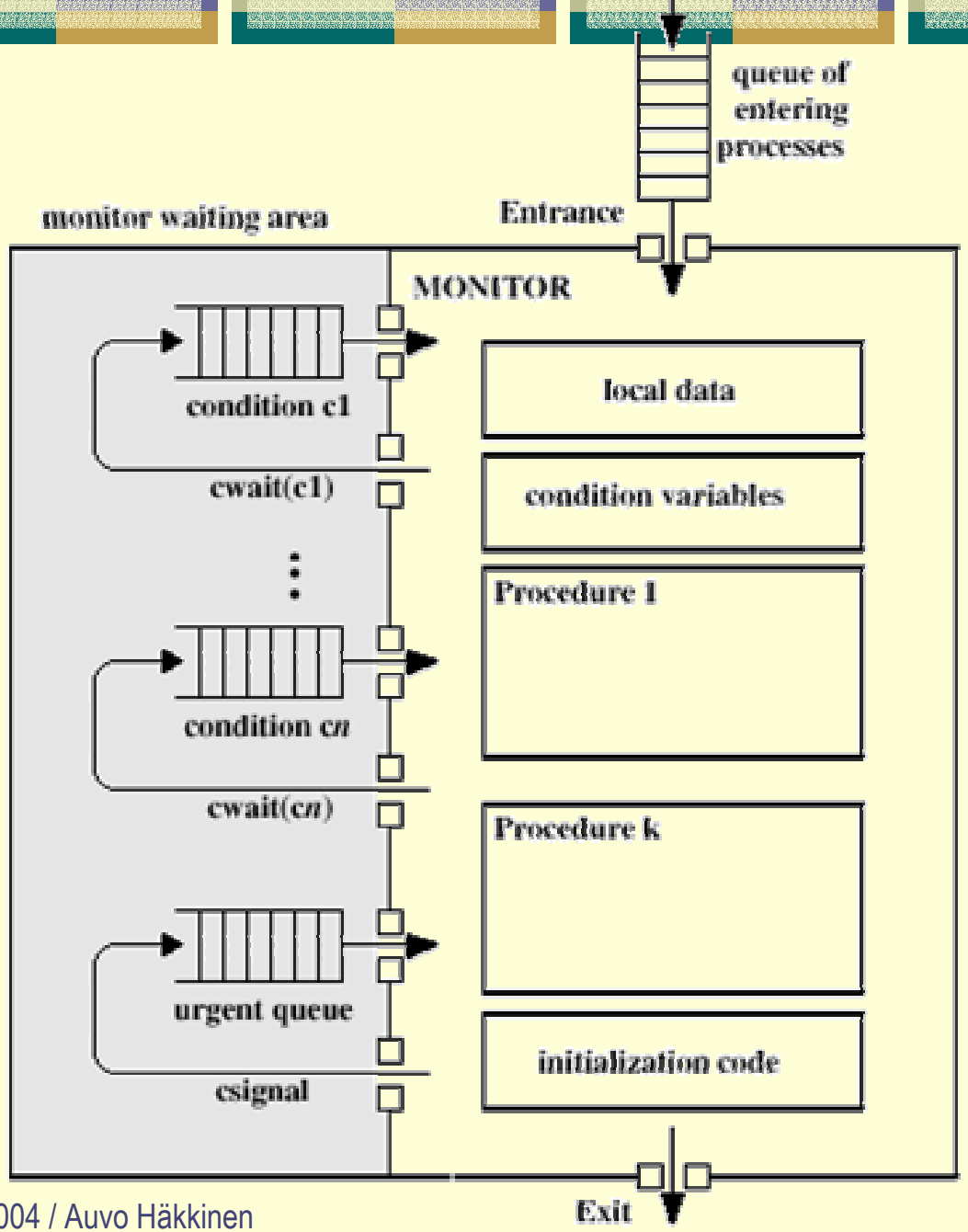
Lukijat ensin

```
bool free = true;
sem e = 1, b[n] = ([n] 0);   # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SIN:  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.

Monitorit

- **wait(ehtomja) + signal(ehtomja): synkronointi**
- **Condition passing**
  - odotuksen syynä ollut ehto jätetään muuttamatta (vaikkei enää ole voimassa), herätä odottaja
    ⇨ suoritettava prosessi vaihtuu, ei salli etuilua
- **Covering condition**
  - odotuksen syynä oleva ehto ei enää voimassa, herätä kaikki odottajat
  - herätetyt tarkistavat vuorollaan saako jatkaa
- **Yksityinen ehtomuuttuja**
  - suoritusvuorojen säätely (ei aina FCFS, tai SJN)



Monitorin muuttujat yhteiskäytössä.

Vain yksi prosessi kerrallaan suorittaa monitorin koodia.

Proseduureissa voi käyttää paikallisia muuttujia,

kullakin prosessilla niistä oma kopio (pinossa).

Stallings Fig. 5.21: Structure of a Monitor.

Ehtomuuttujat ja operaatiot

- **cond cv**
  - ei arvoa - vain jono Blocked prosesseja (paikka odotukselle)
- **wait(cv)**
  - laita prosessi jonoon odottamaan operaatiota s*ignal()*
  - prosessi joutuu aina jonoon!
- **signal(cv)**
  - jos jono tyhjä, ”no operation”, ehtomuuttuja ”ei muista”
  - jos jonossa prosesseja, herätä jonon ensimmäinen
- **empty(cv)**
  - palauta true, jos jono on tyhjä

vrt. semafori!

Lisää operaatioita

- **wait (cv, rank)**
  - odota arvon mukaan kasvavassa järjestyksessä (priority wait)
- **minrank(cv)**
  - palauta jonon ensimmäisen prosessin arvo
- **signal_all(cv)**
  - herätä kaikki ehtomuuttujassa cv odottavat prosessit
  - S&C: while (! empty(cv)) signal(cv);
  - S&W: ei kovin hyvin määritelty miksei?

vrt. semafori!

Prioriteetin mukaan jonotus (Priority Wait)

```
monitor Shortest_Job_Next {

  bool free = true;   ## Invariant SJN:  see text
  cond turn;          #  signaled when resource available

  procedure request(int time) {
    if (free)
      free = false;
    else
      wait(turn, time);
  }

  procedure release() {
    if (empty(turn))
      free = true
    else
      signal(turn);
  }
}
```

Condition passing:

Pidä resurssi varattuna,

anna varattuna seuraavalle prosessille!

⇨ Ei etuilua!

vrt. 4.14

Andrews Fig. 5.6.

”Kattava herätys” (Covering Condition)

```
monitor Timer {

   int tod = 0;    ## invariant CLOCK -- see text
   cond check;     #  signaled when tod has increased

   procedure delay(int interval) {
      int wake_time;
      wake_time = tod + interval;
      while (wake_time > tod) wait(check);
   }

   procedure tick() {
      tod = tod + 1;
      signal_all(check);
   }

}
```

Herätä kaikki odottajat - tarkistakoot itse, onko jatkamislupa edelleen voimassa!

Andrews Fig. 5.7.

Priority Wait

```
monitor Timer {
   int tod = 0;    ## invariant CLOCK -- see text
   cond check;     #  signaled when minrank(check)<=tod
   procedure delay(int interval) {
      int wake_time;
      wake_time = tod + interval;
      if (wake_time > tod) wait(check, wake_time);
   }
   procedure tick() {
      tod = tod+1;
      while (!empty(check) && minrank(check) <= tod)
        signal(check);
   }
}
```

Herätä vain ne, jotka voivat jatkaa!

Andrews Fig. 5.8.

Sanomat

- **send chan(msg) + receive chan(msg)**
- **Palvelut, joissa yksi säie (prosessi)**
  - Ei poissulkemistarvetta
  - Yhtäaikainen odottaminen
    ⇨ pyyntö odottaa (ei prosessi)
- **Palvelut, joissa monta säiettä (prosessia)**
  - Poissulkeminen, synkronointi (semaforit tai monitori)
  - Yhtäaikainen odottaminen
    ⇨ prosessi odottaa
- **send asynkroninen, receive blokkaava**

Kanavat (Andrews)

- **Yhteinen ’postilaatikko’**
  - jono sanomia, FIFO
  - kaikki kanavan sanomat rakenteeltaan samanlaisia
- **chan ch(type$_1$ id$_1$, ..., type$_n$ id$_n$)**
  - ch: kanavan nimi
  - type$_i$ id$_i$: sanoman osien tyypit, ja nimet (saavat puuttua)
- **Esim.**
  - chan input(char);
  - chan disk_access (int cylinder, int block, int count, char* buffer);
  - chan result[n] (int); *# kanavien taulukko*

Operaatiot

- **send kanava(lauseke$_1$, ... , lauseke$_n$)**
  - lähetä sanoma kanavaan
- **receive kanava(muuttuja$_1$, ... , muuttuja$_n$)**
  - vastaanota sanoma kanavasta
- **empty(kanava)**
  - tarkista onko kanava tyhjä sanomista
- **Esim.**
  - send disk_access(cylinder+2, block, count, buf)
  - receive result[i](sum)
  - empty(input)

Ei ota kantaa minkä prosessin kanssa kommunikoi!

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n](int unitID);
process Allocator {
  int avail = MAXUNITS; set units = initial values;
  queue pending;  # initially empty
  int clientID, unitID; op_kind kind;
  declarations of other local variables;
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
       if (avail > 0) {  # honor request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
       } else    # remember request
          insert(pending, clientID);
    } else {    # kind == RELEASE
       if empty(pending) {
          avail++; insert(units, unitid);
       } else {
          remove(pending, clientID);
          send reply[clientID](unitID);
       }
    }
  }
}
```

Resurssin varaus, Palvelija

```
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)
  receive reply[i](unitID);
  # use resource, release
  send request(i,RELEASE,unitID);
   ...
}
```


Andrews Fig. 7.7.

Tiedosto-palvelijat ja asiakkaat

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n](int kind, types of other arguments);
chan open_reply[m](int serverID);  # server id or error
chan access_reply[m](types of results);  # data, error, ...

process File_Server[i = 0 to n-1] {
  string fname; int clientID;
  kind k; variables for other arguments;
  bool more = false;
  variables for local buffer, cache, etc.;
  while (true) {
    receive open(fname, clientID);
    open file fname; if successful then:
    send open_reply[clientID](i); more = true;
    while (more) {
      receive access[i](k, other arguments
      if (k == READ)
        process read request;
      else if (k == WRITE)
        process write request;
      else    # k == CLOSE
        { close the file; more = false; }
      send access_reply[clientID](results
    }
  }
}
```

```
process Client[j = 0 to m-1] {
  int serverID;
  send open("foo", j);
  receive open_reply[j](serverID);
  # use file then close
  send access[serverID](access argument
  receive access_reply[j](results);
    ...
}
```


Andrews Fig. 7.10.

Etäproseduurikutsu, Remote Procedure Call (RPC)

- **Palvelu etäkoneessa, ei yhteistä muistia**
- **Asiakkaat pyytävät palvelua prosedurikutsumekanismilla**
- **Toteutuksen yksityiskohdat KJ:n palvelua**
  - taustalla sanomanvälitys
- **RPC yhdistää monitorin ja synkronisen sanomanvälityksen piirteet**
  - kaksisuuntainen synkroninen kanava yhdellä kutsulla
  - asiakas odottaa

Etäproseduurin moduuli

```
module mname
   op opname(formals)  [returns result]
body
    variable declarations;
    initialization code;
    proc opname(formal identifiers) returns result identifier
        declarations of local variables;
        statements
    end
    local procedures and processes;
end mname
```

julkisten operaatioiden esittely (export)

Kutsu

```
call mname.opname(arguments)
```

Aikapalvelumoduuli

```
module TimeServer
  op get_time() returns int;  # retrieve time of day
  op delay(int interval);     # delay interval ticks
body
  int tod = 0;                # the time of day
  sem m = 1;                  # mutual exclusion semaphore
  sem d[n] = ([n] 0);         # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) {      # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]);    # wait to be awakened
  }
```

Andrews Fig. 8.1.

```
process Clock {
   start hardware timer;
   while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
         remove (waketime, id) from napQ;
         V(d[id]);  # awaken process id
      }
      V(m);
   }
}
end TimeServer
```

Kutsu:

```
time = TimeServer.get_time();
call TimeServer.delay(10);
```

Andrews Fig. 8.1.

RPC / Rendezvous

- **call Mname.opname()**
- **Etäproseduurikutsu**
  - Passiivisten palvelurutiinien etäkäyttö
  - poissulkeminen, synkronointi
- **Rendezvous**
  - Aktiiviset kommunikoivat prosessit
  - Yksi operaatio kerrallaan (kohtaaminen)
  - ei poissulkemista, synkronointi
- **Synkroninen, blokkaava**

Rendezvous moduuli

```
module Mname
    op opname1(formals), opname2(formals);
body
    declarations of shared variables;
    local procedures and processes;
    process pname {
        declarations of local variables;
        while (true) {
            statements;
            in opname1(formals) -> statements;
            [] opname2(formals) -> statements;
            ni
            statements;
        }
    }
end mname
```

julkisten operaatioiden esittely (export)

kohtaamispaikat, jotka toteuttavat operaatiot

Kutsu

```
call Mname.opname(arguments)
```

Yleinen muoto

in $op_1(formals_1)$ and B_1 by e_1 -> S_1;
[ ] ...
[ ] $op_n(formals_n)$ and B_n by e_n -> S_n;
ni

- in op(formals) — operaation nimi ~ kohtaamispaikka
- and B — synkronointilauseke (boolean lauseke)
- [ ] … — muut vartioidut kohtaamispaikat (FCFS)

Ja vuorottamislauseke (by lauseke)

- kohtaamiseen voi syntyä jonoa (synkronointilauseke ei true)
  ⇨ missä järjestyksessä odottavat palvellaan (~prioriteetti)
- oletus: palvele vanhin pyyntö ensin

Bounded Buffer

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  process Buffer {
    typeT buf[n];
    int front = 0, rear = 0, count = 0;
    while (true)
      in deposit(item) and count < n ->
            buf[rear] = item;
            rear = (rear+1) mod n; count = count+1;
      [] fetch(item) and count > 0 ->
            item = buf[front];
            front = (front+1) mod n; count = count-1;
      ni
  }
end BoundedBuffer
```

vrt. Andrews Fig. 5.4

Poissulkeminen? Synkronointi?

Andrews Fig. 8.5.

Shortest_Job-Next allokointi

```
module SJN_Allocator
  op request(int time), release();
body
  process SJN {
    bool free = true;
    while (true)
      in request(time) and free by time -> free = false;
      [] release() -> free = true;
      ni
  }
end SJN_Allocator
```

vrt. Andrews Fig 5.6

Andrews Fig. 8.8.

Klassisia malleja

- **Bounded buffer** (tuottaja – kuluttaja, suodin)
- **Resurssien allokointi** (asiakas – palvelija)
- **Lukijat/kirjoittajat** (luokan vuorot)
- **SJN-skedulointi** (priority wait)
- **Aikaviipaleet ja ajastimet** (vuorottamispalvelu)
- **Nukkuva parturi** (prosessien kohtaaminen
- **Aterioivat filosofit** (lukkiuma)

(hajautettu resurssien jakelu)

Arkkitehtuureja

- liukuhihna, suodin
- tuottaja – kuluttaja
- asiakas – palvelija
- vertaistoimijat (peer-to-peer,P2P)
- monipuolisemmat rakenteet: heartbeat yms.
- ryhmä: keskitetty, rengas, symmetrinen

Evaluoi

- **Oikeellisuus**
  - Suorituspolkujen analyysi
  - Tilamallit
- **Suorituskyky**
  - Yleisrasite
  - Komponentti
  - Kommunikointi / ryhmän kommunikointi
  - Rinnakkaisuusaste

Selvitä aina kuinka järjestelmä käyttäytyy!

END JOB

V(Rio);
signal(Rio);
call Lectures.close(Rio);