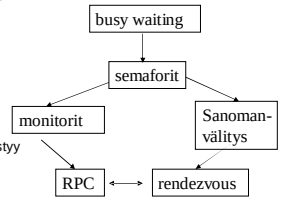
Hajautettu ympäristö

Ei yhteistä muistia
Kommunikointi sanomien välityksellä

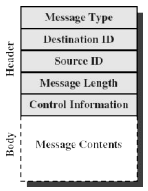
Sisältö, osa II

- Sanomanvälitys
  - **kanavat** (channel), joihin lähetetään dataa (sanomia)
    sanomanvälitysprimitiivejä (**send, receive**)
  - Lähettäminen ja vastaanottaminen myös tahdistaa prosesseja
    - Synkroninen ja asynkroninen
  - Kanava huolehtii itse poissulkemisesta
- Etäperaatio
  (~ monitori + sanomanvälitys)
  - Operaatiokutsu pysäyttää kutsujan odottamaan kutsun tuloksia
  - **Etäproseduurikutsu** (RPC)
    - Luodaan uusi (etä)prosessi
  - **Rendezvous** (Ada-kielessä)
    - Olemassa oleva odottava käynnistyy saatuaan sopivan syötteen



Sanomanvälitys

- Käyttö
  - prosessien välinen tiedonvälitys
  - synkronointi: vuorot / ajoitus
- Käyttäjätason operaatiot
  - lähetys **send (receiver-id, msg)**
  - vastaanotto **receive([sender-id,] msg)**
- Matkassa myös KJ:n lisäämä tieto
  - ainakin: sender-id, receiver-id, sanoma, ...
- Toteutus?
  - pistokkeet: UDP/TCP, ...



Odotussemantiikka

- Send
  - jatka heti, kun KJ kopioinut puskuriinsa ***tai***
  - odota kunnes vastaanottaja saanut sanoman
- Receive
  - odota kunnes KJ kopioinut prosessin muuttujiin ***tai***
  - jos kanava tyhjä, jatka välittömästi
    - jos ei sanomaa ~"no operation"
- *Blocking*, *non-blocking*

Synkronointi

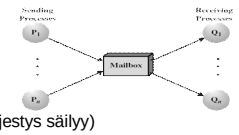
receive \ send	blocking	nonblocking
blocking	synkroninen	asynkroninen
nonblocking	asynkroninen	asynkroninen

Synkroninen
- lähettäjä tietää, että vastaanottaja on saanut sanoman
- toimintojen suoritusjärjestys selvä

Asynkroninen
- toimintojen tarkka suoritusjärjestys epäselvä

oletetaan: asynkroninen tiedonvälitys
send non-blocking, receive blocking

vat



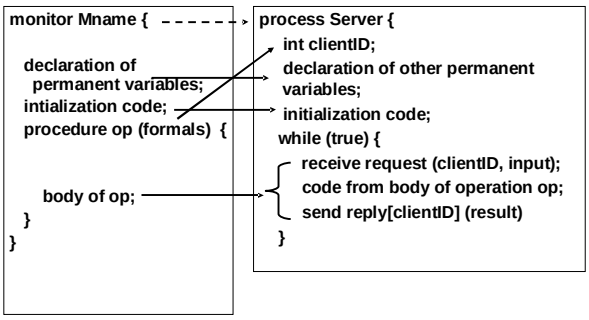
- Yhteinen 'postilaatikko'
  - jono sanomia, **FIFO** (sanomien järjestys säilyy)
  - kaikki kanavan sanomat rakenteeltaan samanlaisia
- chan ch($type_1$ id_1, ..., $type_n$ id_n)
  - ch: kanavan nimi
  - $type_i$ id_i: sanoman osien tyypit, ja nimet (saavat puuttua)
- Esim.
  - chan input(char);
  - 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 (jää odottamaan kunnes saa sanoman)
- **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!

Monitori ó palvelinprosessi



staattinen sidonta, globaalit kanavat

chan request (int clientID, types of input values);
chan reply[n](types of results)

```
process Client[i=0 to n-1] {
      .........
  call Mname.op(arguments);
      .........
}
```

```
process Client[i = 0 to n-1] {
      .........
  send request (i, value args);
  receive reply [i] (result args);
      .......
}
```

Entä jos useita proseduureja? Näinhän monitorilla yleensä on

- asiakkaan kerrottava, mitä operaatiota kutsuu: lisätietona sanomassa

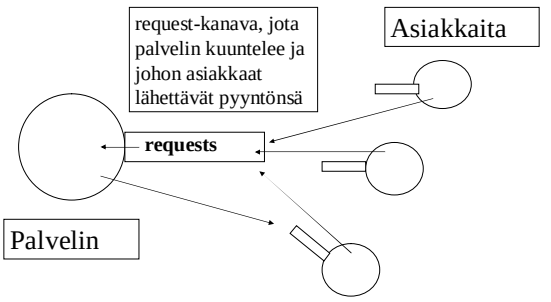
- eri operaatioilla eri parametrit kutsuttaessa ja erilaiset tulosmuodot => eri kutsut eri kanaviin => rinnakkaisuutta palvelimen sisällä eli kuunneltava useaa kanavaa samanaikaisesti.

Asiakkaat ja yhden palvelun palvelija

```
chan request(int clientID, types of input values);
chan reply[n](types of results);
process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {   ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}
process Client[i = 0 to n-1] {
  send request(i, value arguments);     # "call" op
  receive reply[i](result arguments);   # wait for reply
}
```

yhteinen pyyntökanava, yksityiset vastauskanavat

Andrews Fig. 7.4.



Kullakin asiakkaalla on oma kanava reply [i], johon palvelin lähettää vastauksen

Monen palvelun palvelija:

```
process Server {
    int clientID;
   op_kind kind; arg_type args;  res_type results;
    declaration of other  variables; initialization code;
   while (true) {
       receive request (clientID, kind, args);
       if (kind ==op1)  {body of op1; }
       .....
       else if (kind == opn) {body of opn;}
       send reply[clientID] (results);
   }
}
```

```
type op_kind = enum(op1, ... , opn);
type arg_type = union(arg1, ... , argn);
type result_type = union(res1, ..., resn);

chan request (int clientID, op_kind, arg_type);
              chan reply[n] (res_type);

process Client[i=0 to n-1] {
    arg_type myargs; result_type myresults;
    place value arguments in myargs;
    send request(i, opj, myargs);
    receive reply[i](myresults);
    }
```


Asiakkaat ja monen palvelun palvelija

Pyyntökanava

- Asiakkaiden tuntema julkinen kanava
  - käytännössä: IP-osoite ja porttinumero
- Yksi pyyntökanava
  - *sanoma kanavaan ð tulkitse tyyppi, valitse palvelu*

  tai dedikoitu kanava kullekin palvelulle
  - *valitse palvelu ð lähetä sopivaan kanavaan*

Vastauskanava

- Palvelijan tuntema (staattinen)
  - Jokaisella asiakkaalla oma yksityinen
    - kerro oma identiteetti pyyntösanomassa
- Asiakas kertoo pyynnössä (dynaaminen)
  - käytännössä: oma IP-osoite ja porttinumero

Aterioivat filosofit + sihteeri, yksi syöttökanava

```
type opType = enum(REQ, REL);
chan phone(int, opType), reply[5]( );
```

```
process philosopher[i=0 to 4] {
  while (true) {
    think();
    send phone(i, REQ);
    receive reply[i]();
    eat();
    send phone(i, REL);
}
```

Keskitetty resurssien hallinta:

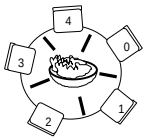
- *"public phone number", many-to-one*
- *"secret phone number", one-to-one*

```
process secretary {
  while (true) {
    receive phone(philo, what);
    switch (what) {
      case REQ: {
        state[philo]=HUNGRY;
        consider_allocation_to(philo);
      }
      case REL: {
        state[philo]=THINKING;
        consider_allocation_to(philo-1);
        consider_allocation_to(philo+1);
      }
    }
}
```



```
type activity = enum(THINKING, HUNGRY, EATING);
activity state[5] = ([5] THINKING);
```

```
procedure consider_allocation_to(int i) {
 if ( state[i] == HUNGRY)
    if (state[i-1]!=EATING AND state[i+1]!=EATING) {
       state[i] = EATING;
       send reply[i]();
    }
}
```



Aterioivat filosofit + sihteeri, dedikoidut kanavat

```
chan request(int), release(int),
      reply[5]( );
sem mutex=1;
```

```
process philosofer[i=0 to 4] {
  while (true) {
    think();
    send request(i);
    receive reply[i]();
    eat();
    send release(I);
}
```

Miten odottaa yhtä aikaa kahdesta eri kanavasta?

```
process secretary {
 co  while (true) {
        receive request(philo);
        P(mutex);
        state[philo]=HUNGRY;
        consider_allocation_to(philo);
        V(mutex);
      }
//  while (true) {
        receive release(philo);
        P(mutex);
        state[philo]=THINKING;
        consider_allocation_to(philo-1);
        consider_allocation_to(philo+1);
        V(mutex);
      }
oc
}
```

Huomaa rinnakkaisuus!

Monitori vs. Palvelija

- proseduurikutsu vs. kanava & sanoma
- poissulkeminen
  - monitori: implisiittisesti, ei semaforeja!
  - palvelija: palvele yksi asiakas kerrallaan, uudet pyyntösanomat jäävät kanavaan
- synkronointi
  - monitori: jos ei saa edetä, wait(ehtomuuttuja)
    - kutsunut prosessi nukahtaa
  - palvelija: jos ei saa edetä, laita sisäiseen jonoon
    - palvelija ei voi nukahtaa!

Monitori vs. Palvelija

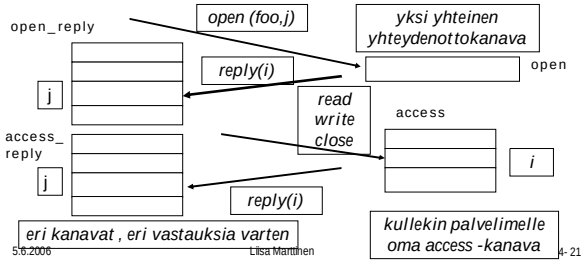
Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	`request` channel and operation kinds
procedure call	`send request(); receive reply`
monitor entry	`receive request()`
procedure return	`send reply()`
`wait` statement	save pending request
`signal` statement	retrieve and process pending request
procedure bodies	arms of case statement on operation kind

Andrews Table 7.1.

Kommunikointi: Jutustelun jatkuvuus / ylläpito

- Useita asiakkaita, useita palvelijoita
- Esim: Tiedostopalvelija
  - yksi ulospäin näkyvä palvelu (tdstojen käyttö)
  - yksi palvelijaprosessi kullekin asiakkaalle
- Sitominen
  - asiakas ó mikä tahansa prosessi
- Tarve
  - prosessi haluaa tehdä sarjan peräkkäisiä tiedosto-operaatioita (open(…), read(), …), sama palvelijaprosessi palvelee

type kind = enum (READ, WRITE, CLOSE);
chan open (string fname; int clientID);
chan open_reply[m] (int serverID); # kuka palvelee
chan access[n](int kind, types of other arguments)
chan access_reply[m](types of results); # dataa, virheilm.




```
process File_Server [i =0 to n-1] {
   string fname; int clientID;
   kind k;   variables fo other arguments;
   bool more = false; local variables;
   while (true) {
     receive open (fname, clientID);
     open file fname; if (open _successful) {
        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 if (k == CLOSE)  {close file; more = false;}
              send access_reply[clientID](results);
} }      }
```

Vertaistoimijat (Interactive Peers)

Esim. Arvojen välittäminen

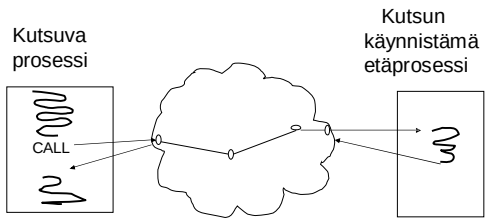
- n prosessia
- kullakin paikallinen arvo v
- etsi globaali min(v), max(v)
- Tiedonvälitys
  - Keskitetty? Symmetrinen? Rengas?
  - Rinnakkaisuus?
  - Tiedonvälityksen tehokkuus?

Synkroninen sanomanvälitys

- synch-send()
  - send() ja receive() "kohtaavat"
- kommunikoivien prosessien synkronointi
  - naapuri tietää naapurinsa tilan
- ei tarvitse välttämättä KJ:n apupuskureita
- rajoittaa rinnakkaisuutta
- **Varo lukkiumaa**
  - algoritmit, jotka toimivat asynkronisen kommunikoinnin yhteydessä eivät ehkä enää toimikaan!

P1	P2
…	…
send	send
receive	receive

Etäproseduurikutsu (RPC)



Etäproseduurikutsu, Remote Procedure Call (RPC)

- Palvelu etäkoneessa, ei yhteistä muistia
- Asiakkaat pyytävät palvelua proseduurikutsumekanismilla
- 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]   julkisten operaatioiden
body                                    esittely (export)
  variable declarations;
  initialization code;
  proc opname(formal identifiers) returns result identifier
    declarations of local variables;
    statements
  end
  local procedures and processes;
end mname
```

Kutsu

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


Poissulkeminen moduulin sisällä?

- Monitorin tapaan implisiittinen poissulkeminen moduulin sisällä?
  - vain yksi aktiivinen prosessi voi suorittaa
- Usea moduulin prosessi voi olla samaan aikaan suorituksessa => poissulkemisesta huolehdittava eksplisiittisesti?
  - **kurssilla oletetaan, että tämä tapa on käytössä**
    - On nykyisin yleisempi!
  - poissulkeminen esim. semaforeilla tai paikallisilla monitoreilla tai rendezvousia käyttäen.

Aikapalvelin TimeServer

```
module TimeServer
  op get_time() returns int;     # kysy aikaa
  op delay (int interval);       # pyydä herätys
body
  int tod = 0;
  sem m =1;   # mutex
  sem d[n] = ([n] 0);  # omat odotussemaforit
  queue of (int waketime, int process_id) napQ;
```

Aikapalvelin, jolta voi
- **kysyä kellonaikaa**
- **pyytää herätystä annetun ajan päästä**


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

proc delay (int interval); {
  int waketime = tod + interval;
  P(m);
  insert (waketime, myid) oikeaan paikkaan
  napQ-jonoon;
  V(m);
  P(d[myid]); # jää odottamaan herätystä
}
```

```
process Clock {  # sisäinen prosessi
   käynnistä ajastin;
   while (true) {
       odota keskeytystä ja käynnistä ajastin uudelleen;
       tod = tod + 1; # taas yksi tikitys lisää
       P(m);
       while (tod >= smallest waketime on napQ) {
           remove (waketime, id)  from napQ;
            V(d[id]);  # herätä prosessi
       }
       V(m);
   }
}
end TimeServer
```

Kutsu:
time = TimeServer.get_time();
call TimeServer.delay(10);

RPC sopii asiakas-palvelija sovelluksiin, muu käyttö hankalaa

- Prosessien välinen kommunikointi puuttuu, se on ohjelmoitava erikseen
- RPC ei tue jatkuvaa kommunikointia
  - kutsut aina erillisiä ja edellisistä riippumattomia
  - ohjelmoitava eksplisiittisesti
- Dynaaminen nimeäminen on mahdollista
  - capability ~ osoitin kommunikointioperaatioon
- Vain mekanismi eri moduuleiden kommunikoiniin
  - Synkronoinnista huolehdittava moduulin sisällä

Arvojen välittäminen: kaksi prosessia vaihtaa arvot keskenään

```
module Exchange[i = 1 to 2]
   op deposit(int);
body
   int othervalue;
   sem ready = 0;   # used for signaling

   proc deposit(other) {    # called by other module
     othervalue = other;    # save other's value
     V(ready);              # let Worker pick it up
   }
   process Worker {
     int myvalue;
     call Exchange[3-i].deposit(myvalue); # send to other
     P(ready);              # wait to receive other's value
     ...
   }
end Exchange
```

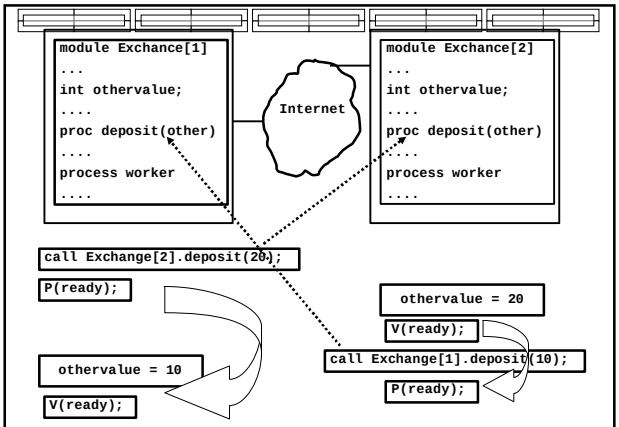
Export-operaatio, jota toinen voi kutsua

Andrews Fig. 8.4.

Oikean indeksin laskemiseen

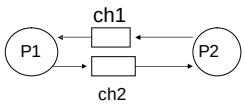
Tuloksena on vähän omituisia ohjelmia!

" Ota talteen minun arvoni!"



Vertaa: Arvojen vaihto sanomavälityksellä:

```
chan ch1 (int),  ch2 (int);
process P1  {            process P2 {
  int value =2;            int value = 1;
  send ch2(value);         send ch1(value);
  receive ch1(value);      receive ch2(value);
}                          }
```



Rendezvous (kohtaaminen)

yhdistää kommunikoinnin ja synkronoinnin,
pyydettyjä operaatioita suoritetan yksi kerrallaan

Rendezvous

- Aktiivinen palvelija
  - suorituksessa oleva prosessi antaa palvelun
  - vrt: RPC: töpö luo/aktivoi prosessin, joka suorittaa palvelun
- Silloin tällöin
  - aktiivinen palvelija ja asiakas *kohtaavat*
  - palvelija suorittaa pyydetyn operaation (asiakas odottaa)
  - ja palvelija jatkaa muita aktiviteetteja (jos on)
- Rendezvous
  - synkronointi ja kommunikointi yhdistetty
  - Palvelija suorittaa operaatioita järjestyksessä, yhden operaation kerrallaan

Vrt. etäproseduuri-kutsu!

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.opname1(arguments);`

Syöttölause (palvelija)

~ Aika ja paikka kohtaamiselle

- in opname(formal identifiers) -> S; ni
  - palvelija odottaa asiakasta, joka kutsuu (call Mname.opname())
  - jonka jälkeen palvelija suorittaa lauseosan S
- opname(...): vahti = kohtaamispaikka
  - Jos asiakas valmiina ("paikalla"), niin suorita lauseosa S
  - Jos asiakas ei paikalla, tarkista muut in-lauseen kohtaamispaikat
- S: vartioidut lauseet (guarded)
  - asiakasprosessi saa jatkaa ("kohtaaminen ohi"), kun lauseosa S on suoritettu

- Yleinen muoto `vahti (guard)`

```
in op1(formals1) and B1  by e1 -> S1;
[] ...
[] opn(formalsn) and Bn  by en -> Sn;
ni
```

Palvelija odottaa, jos mitään in-haaraa ei voida suorittaa (ei ole kutsuttu tai ehto Bi estää)

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

Ja vuorottamislauseke (by lauseke)

- kohtaamiseen voi syntyä jonoa (synkronointilauseke Bi on 'false')
  - ð missä järjestyksessä odottavat palvellaan (~prioriteetti)
- oletus: palvele vanhin pyyntö ensin


```
module BoundedBuffer
  op deposit (typeT), fetch (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
  }
```



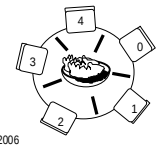
```
process Producer[i=1 to N]
{  typeT item;
   while (true) {
     …
     item = add_this_and_that();
     call BoundedBuffer.deposit(item);
     …
   }
}
```

```
process Consumer[i=1 to M]
{  typeT item;
   while (true) {
     …
     call.BoundedBuffer.fetch(item);
     do_this_and_that(item);
     …
   }
}
```

Ateroivat filosofit, keskitetty ratkaisu

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
           eating[i] = false;
      ni
  }
end Table
```

```
process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    eat;
    call relforks(i);
    think;
  }
}
```



Andrews Fig. 8.6.

Aikapalvelija

```
module TimeServer
  op get_time() returns int;
  op delay(int);
  op tick();      # called by clock interrupt handler
body TimeServer
  process Timer {
    int tod = 0;  # time of day
    while (true)
      in get_time() returns time -> time = tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod = tod+1; restart timer; }
      ni
  }
end TimeServer
```

Andrews Fig. 8.7.

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
```

Kun resurssi vapaa, niin hyväksytään pyyntö, jolla pienin aikatarve (time)

Andrews Fig. 8.8.

Arvojen välittäminen

Asymmetrinen ratkaisu

```
module Exchange[i = 1 to 2]
  op deposit(int);
body
  process Worker {
    int myvalue, othervalue;
    if (i == 1) {   # one process calls
      call Exchange[2].deposit(myvalue);
      in deposit(othervalue) -> skip; ni
    } else {        # the other process receives
      in deposit(othervalue) -> skip; ni
      call Exchange[1].deposit(myvalue);
    }
    ...
  }
end Exchange
```

Parempi kuin RPC:llä, mutta hankalampi kuin sanomanvälityksellä!

Andrews Fig. 8.10.

Varo lukkiumaa!

Ada: Rendezvous ja kohtaamispaikat

```
select
   accept op1(formals1) do
     statements;
or
   ...
or
   accept opn(formalsn) do
     statements;
end select
```


Mitä kaikkea opittiin?

Kurssin sisältö:

1. Rinnakkaisuuden ongelmakenttä
   käsitteitä ja termejä
2. Prosessien tahdistus semaforilla
   semafori, P- ja V-operaatiot
3. Prosessien tahdistus monitorilla
   monitori, signal ja wait
4. Tahdistus ilman yhteistä muistia
   - Sanomanvälitys
   - Etäproseduurikutsu (RPC)
   - Adan rendezvous

Tavoite

- Tunnistaa rinnakkaisuuteen liittyvät ongelmat, tietää peruskeinot rinnakkaisuuden tehokkaaseen toteuttamiseen ja osaa toteuttaa tarvittava prosessien tahdistaminen yleisesti käytettyjen menetelmien avulla.
- Yleisesti, ei mihinkään kieleen tai järjestelmään keskittyen
- Täyttyykö tavoite?

- Harjoitustehtäviä on koottu osoitteeseen:
  http://www.cs.helsinki.fi/u/marttine/nokia/harjoituksia.html
  Vastauksia niihin voi kysellä minulta vaikka sähköpostitse:
  liisa.marttinen@cs.helsinki.fi

Termitesti: osaatko?

Tiedätkö, mitä seuraavat termit tarkoittavat?

rendezvous, binäärisemafori, mutex, nukkuva parturi, kriittinen alue, vuorottamislauseke, lukkiutuminen, kanava, etäproseduurikutsu, invariantti, syöttölause, elolukkiutuminen, monitori, vartioitu lause, vertaistoimija, viestikapulanvälitys, nälkiintyminen, signal and wait, aterioivat filosofit, keskinäinen poissulkeminen, invariantti, ehtosynkronointi, P- ja V-operaatio, atomisuus, lukko, aktiivinen odotus, tuottaja-ja-kuluttaja, rinnakkainen suoritus, sanomanvälitys, test-and-set, puomisynkronointi, halkaistu binäärisemafori, condition passing, kattava herätys, asynkroninen kommunikointi, kohtaamispaikka, rajoitettu puskuri,