❶ Johdantoa

Rinnakkaisuus vs. samanaikaisuus
Laitearkkitehtuureja
Prosessien välinen kommunikointi
Yhteiskäyttöinen muisti
Sovellusten luokittelua
Ohjelmointiparadigmoja, syntaksia

Andrews, Ch 1

Rinnakkaisuus - Samanaikaisuus

- **Peräkkäisyyteen perustuvat sovellukset**
  - yksi kontrollisäie
- **Rinnakkaiset sovellukset**
  - useita kontrollisäikeitä
    - kun yksi odottaa, muut voivat edetä
  - suora tai epäsuora kommunikointi
  - synkronointi, poissulkeminen (atomisuus)
- **Rinnakkaisuus vs. samanaikaisuus**

Miksi rinnakkaisuutta?

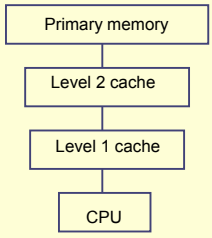
- **Tavoitteita**
  - parantunut suorituskyky
  - parantunut luotettavuus
  - komponenttien yksinkertaisuus
  - sopivuus hajautettuun ympäristöön
- **Hinta**
  - monimutkaisempi suunnitella
  - Käyttäytymisen hallinta vaikeampaa

Laitteistoarkkitehtuureja

SISD
- tavallisia prosessoreja

SIMD
- taulukkoprosessori, vektoriprosessori

Paikallisuus

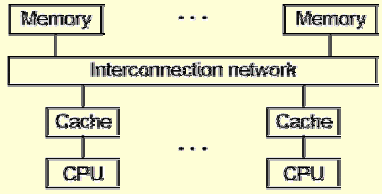


Tavoitteet? Hinta?
Rinnakkaisuus? Samanaikaisuus?

Andrews Fig 1.1

MIMD, tiukasti kytketty

= moniprosessorijärjestelmä, yhteiskäyttöinen muisti

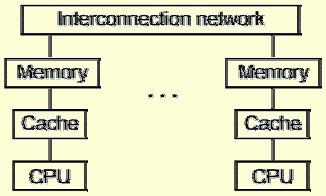


SMP vs. Isäntä ja rengit
Rinnakkaisuus? Samanaikaisuus?
Muistin eheys?

Andrews Fig 1.2

MIMD, löyhästi kytketty

= hajautetut, erilliset koneet



Sanomanvälitys
Rinnakkaisuus? Samanaikaisuus?
Muistin eheys?

Andrews Fig 1.3

Prosessien välinen kommunikointi

Inter Process Communication, IPC

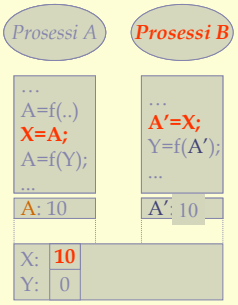
Miksi?

- **Synkronointitarve**
  - Poissulkeminen: vuorot yhteisten resurssien käytössä
  - Synkronointi: odota toisen etenemistä

Kuinka?

- **Talleta paikkaan, josta toinen voi lukea**
  - Muuta yhteiskäyttöisen muuttujan arvoa
    = yhteinen muistialue
    ⇨ Sovellus: poissulkeminen / synkronointi
  - Kirjoita yhteiskäytössä olevaan kommunikointikanavaan
    = sanomanvälitys
    ⇨ KJ: poissulkeminen / synkronointi

❶ Yhteiskäyttöinen muisti



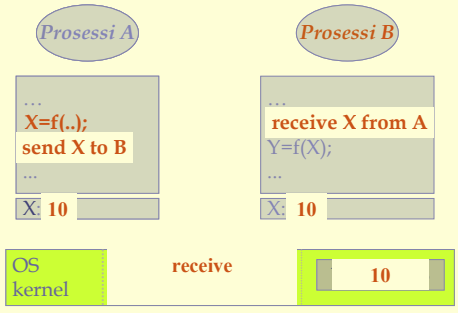
Kommunikointi A⇔B

A ⇨B
A: store into X
B: load from X

B ⇨ A
B: store into Y
A: load from Y

Ongelmakohtia?

❷ Sanomanvälitys



Ongelmakohtia?

Kumpi kommunikointitapa?

Tarkkuus / nopeus (granularity)

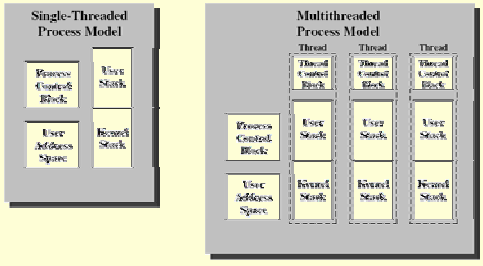
- **Yhteiskäyttöinen muisti**
  - välitön vaikutus
  - käskyn tarkkuudella
- **Sanomanvälitys, dedikoidut prosessorit**
  - LAN: 5-10 ms viiveet
  - proseduurin tarkkuudella
- **Sanomanvälitys, yleiset palvelinkoneet**
  - LAN/WAN: 10 ms – n sekunnin viiveet
  - palvelun tarkkuudella

Luotettavuus

- **Yhteiskäyttöinen muisti**
  - täysin luotettava (tai täysi epäonnistuminen)
- **Sanomanvälitys**
  - Saman koneen sisäisesti
    - sama kuin yhteiskäyttöisellä muistilla (lähes)
  - LAN
    - kommunikointi luotettavaa
    - prosessit voivat epäonnistua toisista riippumatta
  - WAN
    - epäluotettava kommunikointi
    - prosessit voivat epäonnistua toisista riippumatta

Yhteiskäyttöinen muisti

Prosessi ja sen säikeet



Single-Threaded and Multithreaded Process Models

Stallings Fig 4.2

Yhteiskäyttöinen muisti / Säikeet

- **Nopein kommunikointitapa**
- **Yhteiskäyttöinen muisti**
  - oletusarvoisesti saman prosessin säikeille
  - KJ huolehtii viittauksista yhteisiin globaaleihin muuttujiin (kirjanpito PCB:ssä)
- **Kullakin säikeellä oma pino**
  - dynaaminen tilanvaraus aliohjelmia kutsuttaessa
  - kullakin säikeellä omat paikalliset muuttujat
- **Käyttö normaaleilla muistiviitekäskyillä**
  - LOAD, STORE, ...

POSIX-kirjasto, pthread

- **Yli 60 funktiota**
- **Luonti, pysäytys, lopettaminen, ...**
  - pthread_create(), pthread_attr_init()
  - pthread _exit(), pthread_join(), pthread_detach()
  - sched_yield()
- **Synkronointi, poissulkeminen**
  - mutexit, rw-lukot, ehtomuuttujat

JAVA: pakkaus java.lang

- luokka Thread
- säie.start(), säie.stop(), säie.sleep()

⇨ Lue man- / opastussivut
⇨ Andrews ch 4.6, 5.4


```
#include <pthread.h>

// esittele globaalit muuttujat tässä (shared)

int main(int argc, char *argv[]) {
   pthread_t pid, cid;
   printf("Creating two threads\n");
   pthread_create(&pid, NULL, Producer, NULL);
   pthread_create(&cid, NULL, Consumer, NULL);
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);
   printf("Threads joined\n");
}

void *Producer(void *arg) {    // säikeen suoritus alkaa tästä
   // esittele paikalliset muuttujat tässä (private)
   printf("Producer started\n");
}

void *Consumer(void *arg){     // säikeen suoritus alkaa tästä
   printf("Consumer started\n");
}
```

Yhteiskäyttöinen muisti / Prosessit

- Eri vanhemmista peräisin olevat prosessit
- Palvelupyynnöt alueiden luomiseksi ja liittämiseksi prosessin osoiteavaruuteen
  - shmget(), shmctl(), shmat(), shmdt(), ftok()
- Muistiin viittaus normaaleilla käskyillä
- KJ: kirjanpito yhteiskäytössä olevista alueista
- Käyttöoikeudet kuten tdstoilla
  - kuka: u, g, o
  - miten: r, w, x

⇨ Lue man-sivut

RIO-kurssi

```
# esittele yhteiset muuttujat tässä
# esittele yhteiset aliohjelmat tässä

process Prosessi_A(käynnistysparametrit) {
   # esittele paikalliset muuttujat tässä (yksityisessä käytössä)
   # esittele paikalliset aliohjelmat tässä
   lauseet;
}

process Prosessi_B(käynnistysparametrit){
   lauseet;
}
```

Emme ota kantaa siihen kuinka prosessit käynnistetään
Huom: kullakin prosessilla oma pino

Keskeytykset / Vuorottaminen

- Prosessin suoritus voi keskeytyä kahden konekielisen käskyn välissä
  - Prosessori automaattisesti suorittamaan KJ:tä
- Keskeytys voi aiheuttaa prosessin vaihdon...
  - KJ tallettaa rekistereiden arvot PCB:hen
  - KJ lataa rekistereille uudet arvot toisesta PCB:stä

kun prosessi päivittämässä yhteistä data

```
LOAD  R1,A
                LOAD  R1,A
                ADD   R1,=5
                STORE R1, A
ADD   R1,=10
STORE R1, A
```

Atomisuus?

Rinnakkaisten sovellusten luokittelua

1. Moniajojärjestelmät
2. Hajautetut järjestelmät
3. Rinnakkaislaskenta

❶ Moniajojärjestelmät

- Tosistaan riippumattomat prosessit
- Sovellus, jossa useita säikeitä / prosesseja
- Alusta
  - yhteiskäyttöinen muisti, yksi tai useampi CPU
  - vuorottelu
- Esim.
  - Ikkunointiympäristöt
  - Osituskäyttöön perustuvat järjestelmät (esim. palvelimet)
  - Tosiaikaiset ohjausjärjestelmät
- Helppo suunnitella, tehokasta laskentaa
  - kukin prosessi tekee yhtä asiaa

❷ Hajautetut järjestelmät

- Itsenäisten koneiden verkot, sanomanvälitys
  - CPU:t joilla omaa muistia + yhdistävä verkko
  - off-load processing
- Esim.
  - Tiedostopalvelin
  - Tietokanta/transaktiojärjestelmät, esim. lipunvaraus, pankit
  - Web palvelimet Internetissä (… DNS)
  - Globaalit yritysjärjestelmät: Integroidut business-alueet
  - Vikasietoiset järjestelmät, toisinnus

Bacon Fig 1.1: A distributed control system
Bacon Fig 1.2: Multimedia workstations
Bacon Fig 1.3: Airline booking system
Bacon Fig 1.5: A simple distributed operating system

❶ Rinnakkaislaskenta

- Nopeammin, enemmän aikayksikössä
- Esim.
  - Tieteellinen laskenta, mallintaminen (esim. sääennuste)
  - Grafiikka ja kuvankäsittely
  - Laajat kombinatoriset ja optimointiongelmat (esim. talous)
- Rinnakkainen data: SIMD
  - sama temppu, mutta vain osalle dataa
  - Bacon Fig 1.6: Replicated code, partitioned data
- Rinnakkainen laskenta: MIMD
  - eri temput eri prosesseille (eri data)
  - Bacon Fig 1.7: A four step pipeline

Missä luokittelun erot?

- Kommunikointinopeus
  - yhteinen muisti vs. sanomanvälitys
- Yhteistoiminnan luonne
  - asiakas - palvelija
- Toiminnan autonomisuus (ml. virheet.)
  - samassa koneessa vs. erilliset koneet
  - verkko

Ohjelmointiparadigmat

❶ Iteratiivinen rinnakkaisuus
❷ Rekursiivinen rinnakkaisuus
❸ Tuottaja - Kuluttaja
❹ Asiakas - Palvelija
❺ Vuorovaikutteiset kumppanit/ryhmät

❶ Iteratiivinen rinnakkaisuus

- Monta identtistä iteratiivista prosessia
  - sama koodi, toistosilmukka
- Kukin laskee toistossa tuloksen osasta dataa, sitten tulokset yhdistetään
  - yhteinen muisti tai sanomanvälitys
  - esim. puurakenteiset algoritmit
- Monta tapaa järjestää rinnakkaisiksi, jos read set ja write set erillisiä
- Esim.
  - Matriisin kertolasku (s. 13-16)
  - Bacon Fig 1.6: Replicated code, partitioned data.


```
double a[n,n], b[n,n], c[n,n];

for [i = 0 to n-1] {
  for [j = 0 to n-1] {
    # compute inner product of a[i,*] and b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

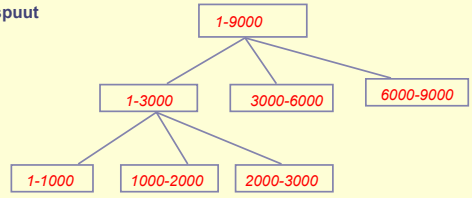
Sequential Matrix Multiplication

```
process row[i = 0 to n-1] { # in parallel
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[ij] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication Using a Process Declaration

Esim.

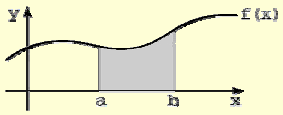
- järjestäminen
- etsiminen
- spekulatiivinen laskenta
- päätöspuut
- pelit



Puurakenteiset algoritmit

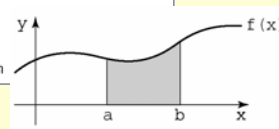
❷ Rekursiivinen rinnakkaisuus

- **Rekursiivinen aliohjelma kutsuu itseään**
  - tilanvaraus pinosta
  - 'Hajoita ja Hallitse'
- **Voidaan tehdä rinnakkain, jos kullakin kutsulla erilliset write -joukot**
  - eivät käytä globaaleja muuttujia
  - kullekin erilliset viite- ja tulosparametrit
- **Esim.**
  - Quicksort
  - Integraalin approksimointi (s. 18-19)




```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

Recursive Procedure for Quadrature Problem

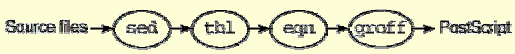



```
co larea = quad(left, mid,fleft,fmid,larea)
// rarea = quad(mid,right,fmid,fright,rarea)
oc
```


❸ Tuottaja - Kuluttaja

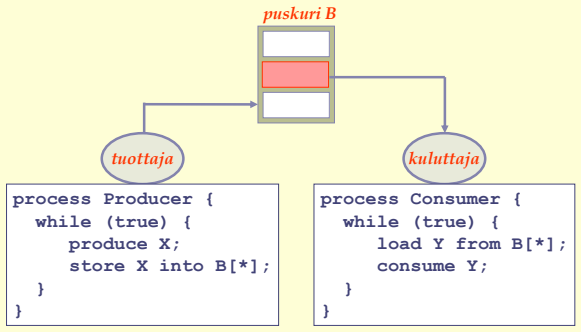
- **Tietoa virtaa vain yhteen suuntaan**
- **Tuottaja laskee ja laittaa tiedon kuluttajan saataville**
- **Kuluttaja käyttää tuotetun tiedon**
  - klassinen rajoitetun puskurin hallintaongelma
- **Esim: Unixin putket**

```
sed -f Script $* | tbl | eqn | groff Macros -
```



Puskurointi?

Andrews Fig 1.5

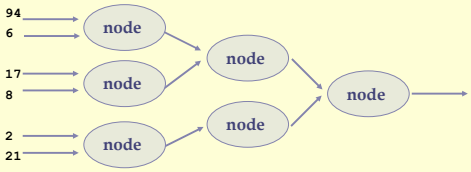


Ongelmia? Huomioitavaa?

Yleistys

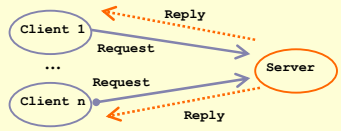
liukuhihna

suodatus



❹ Asiakas – Palvelija

- **Tietoa virtaa molempiin suuntiin**
  - pyyntö-vastaus protokolla
- **Asiakas ja palvelija voivat olla eri koneissa**
  - kutakin pyyntöä palvelemaan oma säie
- **Esim. tdstopalvelin (read, write)**
  - klassinen lukijat / kirjoittajat ongelma

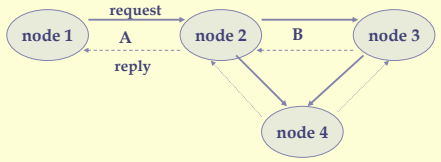


Andrews Fig 1.6


```
process Client [i] {
    ...
    send y to F;
    receive x from F;
    ...
    ...
}
```

```
process Server F {
  while (true) {
    receive msg from *;
    x=f(msg);
    send x to msg.sender;
  }
}
```


Yleistys



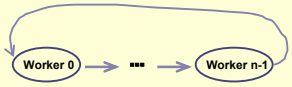
A asiakas: node 1
palvelija: node 2

B asiakas: node 2
palvelija: node 3

Käsite liittyy kommunointiin, ei solmuun

5 Interaktiiviset kumppanit / ryhmät

- **Toimivat palvelijoina sekä asiakkaina**
- *Koordinoija* **tai** *rengas* **tai** *liukuhihna*, **tai kaikki kommunikoivat** *kaikkien kanssa*
- **Sanomanvälitys**
  - send msg to Worker[0]
  - receive msg from Coordinator



Andrews Fig 1.7

Prosessi A
…
vaihe 1
send xa to B
receive xb from B
vertaa
vaihe 2
…

Prosessi B
…
vaihe 1
send xb to A
receive xa from A
vertaa
vaihe 2
…

Asynkroninen vs. synkroninen kommunikointi
Yksisuuntainen operaatio


```
process coordinator {
  double a[n,n];  # source matrix a
  double b[n,n];  # source matrix b
  double c[n,n];  # result matrix c
  initialize a and b;
  for [i = 0 to n-1] {
    send row i of a to worker[i];
    send all of b to worker[i];
  }
  for [i = 0 to n-1]
    receive row i of c from worker[i];
  print the results, which are now in matrix c;
}
```

Esim: hajautettu matriisin kertolasku
(s. 23-26)

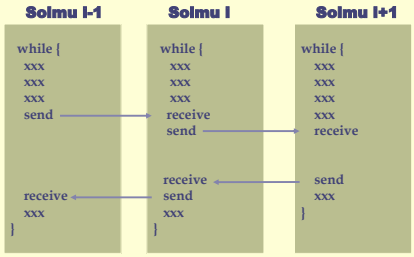
```
process worker[i = 0 to n-1] {
  double a[n];     # row i of matrix a
  double b[n,n];   # all of matrix b
  double c[n];     # row i of matrix c
  receive initial values for vector a and matrix b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j]= c[j]+ a[k]* b[k,j];
  }
  send result vector c to coordinator
}
```

Koordinoija jakaa työt, kokoaa tulokset

```
process worker[i = 0 to n-1] {
  double a[n];        # row i of matrix a
  double b[n];        # one column of matrix b
  double c[n];        # row i of matrix c
  double sum = 0.0;   # storage for inner products
  int nextCol = i;    # next column of results
  receive row i of matrix a and column i of matrix b;
  # compute c[i,i] = a[i,*] × b[*,i]
  for [k = 0 to n-1]
    sum = sum + a[k] * b[k];
  c[nextCol] = sum;
  # circulate columns and compute rest of c[i,*]
  for [j = 1 to n-1] {
    send my column of b to the next worker;
    receive a new column of b from the previous worker;
    sum = 0.0;
    for [k = 0 to n-1]
      sum = sum + a[k] * b[k];
    if (nextCol == 0)
      nextCol = n-1;
    else
      nextCol = nextCol-1;
    c[nextCol] = sum;
  }
  send result vector c to coordinator process;
}
```

Rengas liukuhihnasta

Yleistys: 2-suuntainen liukuhihna

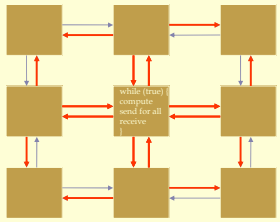


synkronointipisteet:

receive: odota synkronointitietoa

send: lähetä synkronointitieto

Yleistys: Heartbeat



Esim. *Internetin reititystaulut*

algoritmi *vs.* **käyttäytyminen**

Heartbeat

```
process Worker [i=1 to numWorkers]
{
   declaration of local variables;
   initialisation of local variables;
   while (not done) {
       send values to neighbors;
       receive values from neighbors;
       update local values;
   }
}
```

to/from neighbors ⇨ toistosilmukka

Heartbeat, ongelmat

- **Lukkiutumisvaara**
  - synkroninen vs. asynkroninen kommunikointi
- **Käyttäytyminen**
  - Turhaa työtä?
  - "Lyhyin reitti" tiedon välitys
    - mitä jos reitti katoaa?
  - kuormantasaus
    - välitetään tietoa 'vapaasta kapasiteetista'
    - mitä jos verkko ruuhkautunut?

Rinnakkaisuuden ongelmakenttä

Ratkottavien ongelmien aiheita antavat...

- **Rinnakkainen data** → **poissulkeminen**
  - Esim. yhteinen puskuri → **datan ajantasaisuus**
  - read - update - write
- **Rinnakkainen suoritus** → **synkronointi**
  - Esim. tuota - kuluta
- **Resurssien jakelu** → **lukkiutuminen**
  - Joku huutaa lisää
- **Vuorojen jakelu** → **nälkiintyminen**
  - Onneton prioriteetti

Täysin erilliset prosessit	**Kilpailu**	Poissulkeminen Lukkiutuminen Nälkiintyminen
Prosessit epäsuorasti tietoisia toisista	**Yhteistyö** Yhteinen muisti	Poissulkeminen Lukkiutuminen Nälkiintyminen Datan ajantasaisuus
Prosessit suorasti tietoisia toisista	**Yhteistyö** Sanomanvälitys	Lukkiutuminen Nälkiintyminen

- **Suoritusjärjestys / ajoitus**
- **Prosessin lopputulos**

ks. Stallings Tab 5.1

Syntaksimerkinnät

Ohjelmointimerkinnät

Yleiset kurssin algoritmien ohjelmointimerkinnät
Alkujuuret: Pascal, C, C++, Java
Yksikäsitteisesti, ymmärrettävästi

- **Esittelyt**

```
int i, j = 3;
double sum = 0;
real x;
int a[n]          # int a[0:n-1]
int b[1:n]        # array of n integers
double c[n,n] = ([n] ([n] 1.0))
```


- **Peräkkäisesti suoritettavat lauserakenteet**

```
if (ehtolauseke)
   lauseet;
else
   lauseet;

while (ehtolauseke) {
    lauseet;
}

for [määrä1,…, määräN] {
    lauseet;
 }
```

toistojen lukumäärät:

```
i=0 to N
i=1 to N by 2
i=0 to n-1 st i!=x
i=0 to N, j=0 to M
```


- **Rinnakkaisesti suoritettavat lauserakenteet**

```
co lause1;          # N rinnakkaista suoritusta
// ...              # voivat edetä eri CPU:illa
// lauseN;          # vapaassa järjestyksessä
oc
# jatka tästä vasta kun kaikki haarat suoritettu
```

- **Prosessit**

```
process foo {                 # yksi prosessi ...
    lauseet;
}

process bar[i=1 to N]{        # N prosessia
    lauseet;
}
```

Käytämme kurssilla ensisijaisesti prosessimerkintää!

- **Funktiot ja Proseduurit**

```
int addone(int v) {       # palauttaa kokonaisluvun
    return(v+1);
}

main() {                  # proseduuri, ei palauta arvoa
     int n, sum;
     read(n);
     for [i = 1 to n]
           sum = sum + addone(i);
     write ("the final value is", sum);
}
```

Spesifiointimerkinnät

- **Atominen transaktio**

```
state_1              # laskennan tila ennen atomisia operaatioita
< S1; … Sn; >        # lista atomisia operaatioita
state_2              # laskennan tila atomisen osan jälkeen
```

Muut prosessit näkevät joko tilan `state_1` tai `state_2`, mutta eivät koskaan atomisen alueen välituloksia

Esim.
```
x=0; y=10;
< x=x+1; y=y+1; >
```

Muut samoja muuttujia käyttävät prosessit näkevät
`x==0 JA y==10` tai `x==1 JA y==11`

- **< await >**

```
< await (ehtolauseke) lauseet; >
```

- **Synkronointipisteen merkintä**
  - *"odota tässä kunnes ehtolausekkeen arvoksi tulee true"*
  - **Kun suoritus jatkuu kohdasta `lauseet`, `ehto` takuuvarmasti voimassa**
- **Lauseosan suoritus atominen**
  - **Välivaiheet eivät näy muille prosesseille**
- **Esim.** `< await (s>0) s=s-1; >`
- **Toteutus?**
  - **semaforit, monitorit, sanomanvälitys, …**
  - **Eräissä tapauksissa ohjelmoija joutuu vielä tarkistamaan ehdon voimassaolon, ennenkuin voi antaa suorituksen jatkua!**
    (Esim. useita odottajia, jotka pääsevät yhtäaikaa jatkamaan)

Kertauskysymyksiä?