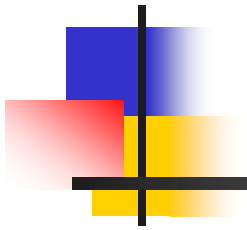
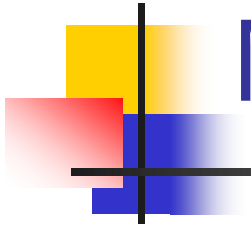


Rinnakkaisohjelmointi

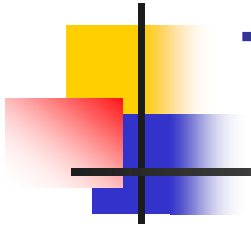
Liisa Marttinen

5.6.2006



Missä rinnakkaisuutta?

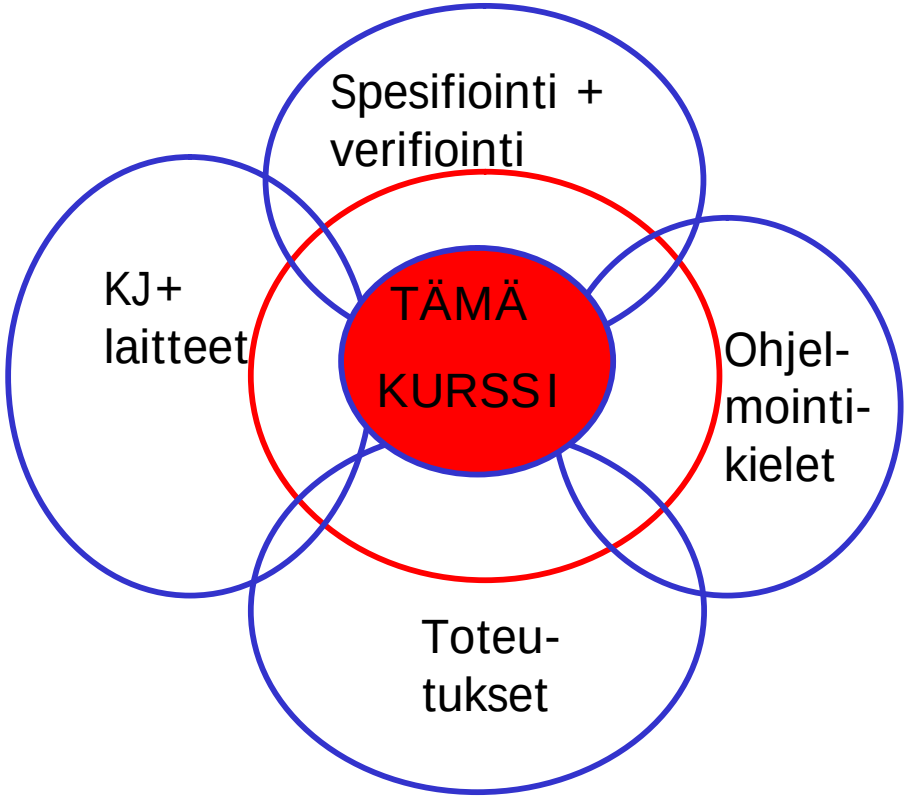
- tietokoneiden käyttöjärjestelmissä
  - I/O-toiminta
- WWW-palvelin
  - palvelee samanaikaisesti useita asiakkaita
- hajautettu laskenta
  - SETI-projekti, sääennusteet, salakirjoituksen murtaminen
- sulautetut järjestelmät
  - esim. autoissa keräämässä ja käsittelemässä tietoja eri paikoista
- simulaatiot
  - reaalimaailman tapahtumien mallintaminen
- multimedia-/peliohjelmat
  - samanaikaisesti käsiteltävä liikkuvaa kuvaa ja ääntä
- komponenttiohjelmointi



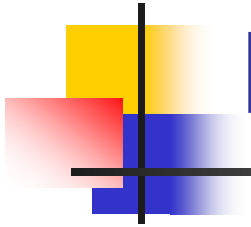
Tavoite

- 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

Eri näkökulmia tarkastella rinnakkaisuutta

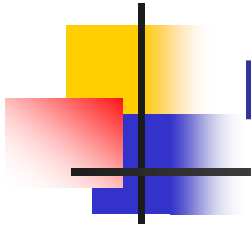


Keskitytään rinnakkaisuuden 'ydinasioihin'.



Kurssin sisältö

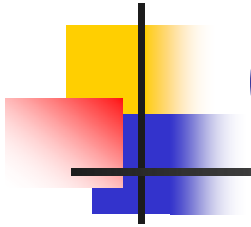
1. Rinnakkaisuuden ongelmakenttä
   käsitteitä ja termejä
2. Prosessien tahdistus semaforilla
   semafori, P- ja V-operaatiot
3. Prosessien tahdistus monitorilla
   monitori, signal ja wait
4. Tahdistus ilman yhteistä muistia
   - Sanomanvälitys
   - Etäproseduurikutsu (RPC)
   - Adan rendezvous



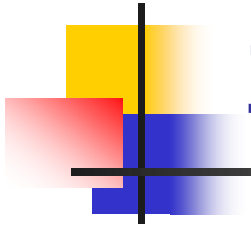
Kirjallisuutta ja muuta materiaalia

- Gregory Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000
- M. Ben-Ari: Principles of concurret and distributed programming, Addison-Wesley, 2. edition 2006
- Gadi Taubefeld: Synchronization Algorithms and Concurrent Programming, Pearson, 2006
- HY:n TKTL:n kurssin Rinnakkaisohjelmointi (-ohjelmistot) luentokalvot
  - Vuosien varrella kehitelleet Teemu Kerola, Timo Alanko, Auvo Häkkinen, Liisa Marttinen

Concurrent programming (computing) (from Wikipedia)



- Concurrent computing is the concurrent (simultaneous) execution of multiple interacting computational tasks. These tasks may be implemented as separate programs, or as a set of processes or threads created by a single program. The tasks may also be executing on a single processor, several processors in close proximity, or distributed across a network.
- Concurrent computing is related to parallel computing, but focuses more on the interactions between tasks. Correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems.
- Pioneers in the field of concurrent computing include Edsger Dijkstra, Per Brinch Hansen, and C. A. R. Hoare



1. Rinnakkaisuuden ongelmakenttä

- Prosessin ajallinen epädeterminististisuus
  - Prosessin tilat ja käskyjen suoritus.
- Atomisuus, kriittinen alue
- Poissulkeminen ja prosessien tahdistus,
- Lukkiutuminen ja nälkiintyminen
- Toiminnan oikeaksi osoittaminen
- Test-And-Set, lukkomuuttuja

Ohjelman suoritus

```
if (x==0)  a=10 else b=5;
```

```
        LOAD R1, x
        J NZER R1, bb
aa      LOAD R2, =10
        STORE R2, a
        JUMP  ohi
bb      LOAD R2, =5
        STORE R2, b
ohi     ……
```

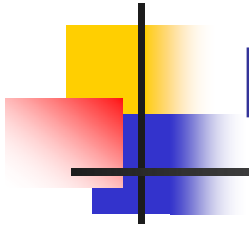
Tietokone osaa suorittaa vain oman konekielensä käskyjä => korkeamman tason ohjelmointikielen lauseet käännetään ensin konekielisiksi.

Konekielisen ohjelman suoritus voi keskeytyä jokaisen konekielisen käskyn jälkeen:

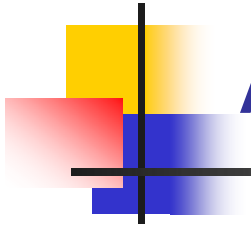
- aikaviipalekeskeytys
- laitteistokeskeytys
- muu KJ:n väliintuloa vaativa tilanne

Prosessin elinkaari: prosessin tilat ja tilasiirtymät

Yksi prosessori, moniprosessorikone, hajautettu järjestelmä



- Yksi prosessori
  - Moniajo eli rinnakkaisuus näennäistä, mutta prosesseja tai säikeitä (thread) suoritetaan limittäin KJ:n määräämässä järjestyksessä.
  - Kommunikointi yhteisen muistin kautta
- Monta prosessoria
  - Aito rinnakkaisuus mahdollinen, kommunikointi yhteisen muistin kautta tai
- Monta tietokonetta
  - Hajautettu järjestelmä, aito rinnakkaisuus, kommunikointi sanomanvälitystä käyttäen



Atominen toiminto (atomic action)

- Käsky tai käskyjakso, joka voidaan suorittaa yhtenä rinnakkaisuuden kannalta jakamattomana toimintona; jota ei voida keskeyttää (*välitilat eivät näy muille*)
  - yksi konekäsky kaikissa koneissa
  - test-and-set–konekäsky
    - Arvon testaaminen ja asetus yhtenä konekäskynä => jakamattomana toimintona
  - Laajempia kokonaisuuksia voidaan toteuttaa kriittisen alueen protokollien avulla
    - Jakamattomuus koskee yleensä vain yhdessä toimivia prosesseja, oikeaan toimintaan ’sitoutuneita’

Kun suoritetaan ohjelmia rinnakkain, lopputulos on epädeterminstinen

Olkoon x==2 ja y ==5

```
co   P:  <x = x + y;>
   || Q: < y = x - y;>
   || R:  <x = x – y;>
oc
```

Rinnakkaisia atomisia (<>) suorituksia

Rinnakkain suoritettuna suoritusjärjestys voi vaihdella (Tässä 6 eri järjestystä).

Jos suoritukset eivät ole atomisia, ne voivat limittyä konekäskyttäin.

P	Q	R
P1: LOAD X	Q1: LOAD X	R1: LOAD X
P2: ADD Y	Q2: SUB Y	R2: SUB Y
P3: STORE X	Q3: STORE Y	R3: STORE X

Erilaisia vaihtoehtoja tulee hyvin, hyvin paljon enemmän!

Kaikki vaihtoehdot eivät ole ’oikean’ toiminnan kannalta hyväksyttyjä.

Esimerkki: pankkiautomaatit

PA P: nosto ==200

.....

LOAD saldo

SUB nosto

STORE saldo

PA Q: nosto =100

.....

LOAD saldo

SUB nosto

STORE saldo

saldo == 300

Vain järjestykset: P1P2P3Q1Q2Q3 ja Q1Q2Q3P1P2P3 antavat pankin kannalta oikean tuloksen saldo ==0, muut saldoksi joko 100 tai 200 STORE-käskyn suoritusjärjestyksestä riippuen

P1P2P3Q1Q2Q3

P1Q1P2P3Q2Q3

P1P2Q1P3Q2Q3

P1Q1Q2P2P3Q3

P1Q1P2Q2P3Q3

P1P2Q1Q2P3Q3

P1Q1Q2Q3P2P3

P1P2Q1Q2Q3P3

P1Q1Q2P2Q3P3

P1Q1P2Q2Q3P3

P1Q1Q2P2Q3P3

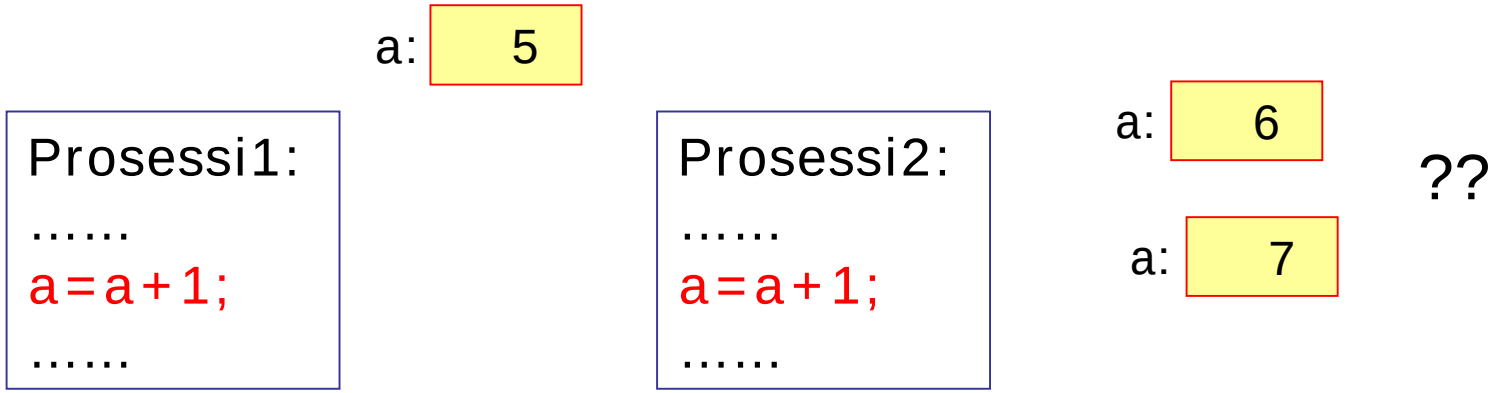
.....

Vastaa-vasti kun ensin on Q1.

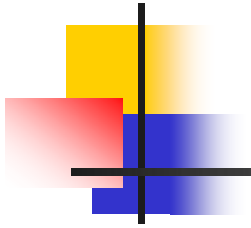
Huom! Tietokantasovellukset lukitsevat datan, rinnakkaisohjelmoinnissa rajoitetaan prosessien suoritusta

Kriittinen alue (critical section)

- Ohjelmakoodin käskyjono, jota vain yksi prosessi kerrallaan voi suorittaa
  - poissulkeminen (mutual exclusion)
  - esim. yhteisen muuttujan päivittäminen



Kriittinen (koodi)alue



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;
                  ......
      {
{
```

Kriiitinen alue pyritään pitämään pienenä, koska se rajoittaa prosessien rinnakkaisuutta

```
while (true)   {
 do something not critical1;
 enter critical section;
   critical section 1;
 exit critical section;
 do  something not critical1
}
```

PROSESSI1:n koodi

```
while (true)   {
 do something not critical2;
 enter critical section;
   critical section 2;
 exit critical section;
 do  something not critical2
}
```

PROSESSI2:n koodi

enter critical section – exit critical section

- varmistavat poissulkemisen
  - vain yksi prosessi pääsee suorittamaan kriittisen alueensa koodia, muut joutuvat odottamaan
  - Seuraava pääsee kriittiselle alueelleen vasta, kun siellä oleva on poistunut (exit critical section)
  - Toteutus riippuu atomisen toiminnon koosta

Decker's algorithm: the solution to the critical section problem for two processes

(atomisia vain yksittäiset konekäskyt (esim. load tai store))

```
boolean enter1=false, enter2=false;
int turn=1;  /* vain lukee enter2- ja turn-muuttujia */
process P1 {
  while (true) {
    noncritical section;
    enter1=true;      /* enter critical section */
    while(enter2)
      if (turn==2) {
         enter1=false;
         while(turn==2) skip; /*odotussilmukka*/
         enter1=true;
      }
critical section;
enter1=false; turn=2  /* exit critical section */
   noncritical section;
 {
{
```

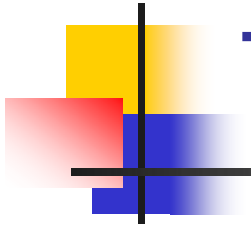
Vastaavasti prosessi P2.

Melkoisen monimutkainen!

Entä jos n prosessia !

Poissulkeminen yksinkertaistuu, jos LOAD + STORE on atominen

Esim. test-and-set, test-and-test-and-set, exchance, ...



Test- and-set ja lukkomuuttuja

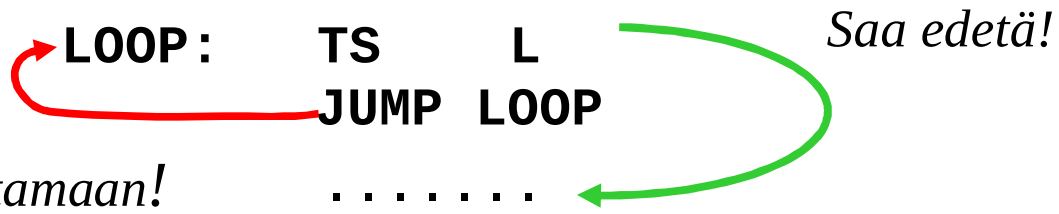
- Luetaan muuttujan arvo ja asetetaan se yhtenä atomisena toimintona
  - Oma konekielinen käsky => atominen laitetasolla

- Nyt voidaan testata muuttuja (”alueen lukon”) tilaa ja varata se itselle:

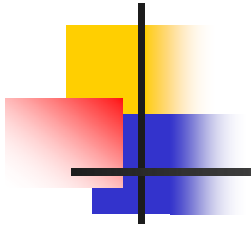
```
bool test-and-set(boolean lock) {
   <bool arvo = lock;
     lock = true;   /*lukko kiinni*/
     return arvo;> /*saiko lukon vai oliko jo
                    varattu jollekin toiselle?*/
   }
```

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:
  - 

Lukkomuuttujat (Spin Locks)



- Boolean-muuttuja lock # *sopimus prosessien kesken!*
  - **`lock== true`** kriittinen alue varattu
    (in1 V in2 V ... V inN)
  - **`lock==false`** kriittinen alue vapaa
- Entry protocol

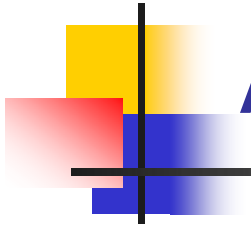
```
while (lock) ;      # aktiivinen odotus, "pörrää"
  # check again if (!lock)  # busy loop
lock=true;
```

- Exit protocol

```
lock=false;
```

Kriittisen alueen hallinta lukkomuuttujalla N:lle prosessille

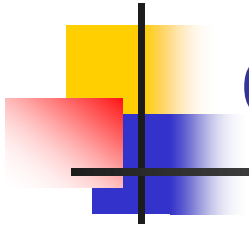
```
bool lock = false; /* lukko auki */
process CriticalSection[i=1 to N] {
    while(true) {
        while (test-and-set (lock)) skip;
        critical section;
        lock = false;
        noncritical section;
    }
}
```



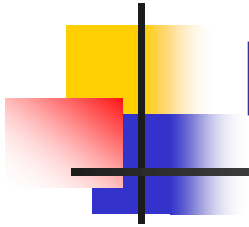
Aktiivinen odotus (busy wait)

- Tuhlaa koneen resursseja
  - Prosessi kuluttaa prosessoriaikaa tai aikaviipaleita odottaessaan
    - Tutkii koko ajan lukkomuuttujaa
    - Korkean prioriteetin prosessi estää alemman prioriteetin prosesseja pääsemästä suoritukseen
- Parempi olisi passiivinen odotus wait-tilassa
  - Ainakin silloin kun prosessorilla on muuta käyttöä
  - Odottava prosessi siirretään wait-tilaan ja sieltä (yleensä) ready-tilaan vasta, kun prosessi todella pääsee etenemään kriittiselle alueelle.
    - => semaforit, monitorit

Rinnakkaisen (jatkuvasti toimivan) ohjelman oikeellisuus (correctness)



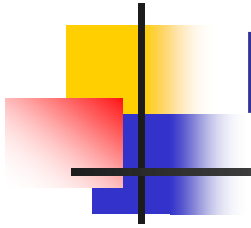
- Turvallisuusominaisuudet (safety properties)
  - Tällaisen ominaisuuden on oltava aina tosi
    - Oikea toiminta, esim. aina kriittisellä alueella on korkeintaan yksi prosessi eli ei koskaan tapahdu sellaista, että usea prosessi suorittaa samanaikaisesti kriittistä koodiaan
    - Ei tapahdu lukkiutumista
- Elävyysominaisuudet (liveness properties)
  - Ominaisuus tulee tulee joskus todeksi
    - Ei tapahdu nälkiintymistä eli jokainen prosessi pääsee joskus kriittiselle alueella
- Tehokkuus
  - Jos pääsisi etenemään (esim. kriittistä aluettaan suorittamassa ei ole muuta prosessia), ei mitään turhaa viivystystä



Lukkiutuminen ja nälkiintyminen

- Lukkiutuminen (deadlock)
  - Mikään prosessi ei pääse etenemään, kaikki odottavat kehässä toisiaan

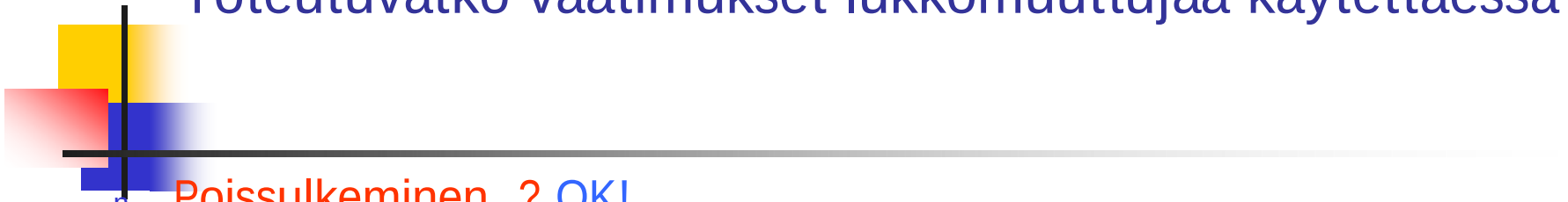
- Nälkiintyminen (starvation)
  - Jokin tai jotkut prosessit eivät lainkaan pääse etenemään. Muut saavat vuoron aina ennen.



Reiluus (fairness)

- Heikko reiluus
  - Jokainen saa joskus vuoron
  - mikään prosessi ei täysin nälkiinny
- Vahva reiluus
  - Kaikki saavat tasapuolisesti vuoroja

Toteutuvatko vaatimukset lukkomuuttujaa käytettäessä?



- 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

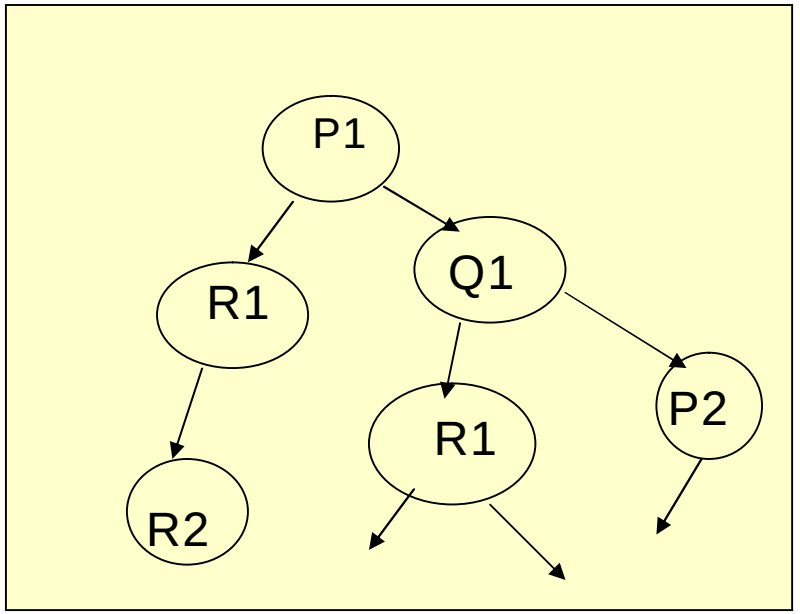
Rinnakkaisen ohjelman oikeellisuuden osoittaminen

- Formaali spesifiointi ja mallintaminen + mallin ohjelmallinen tarkistaminen
  - Tilamalli (state model): kuvataan kaikki mahdolliset suorituspolut tiloina ja kaarina
  - Ongelmana tilaräjähdys
- Deduktiivinen todistus:
  - temporaalilogiikka
    - Invariantti

      lauseke on tosi joka kohdassa ohjelmaa
    - Induktiivitodistus

      tosi alkutilassa =>

      tosi kaikkialla



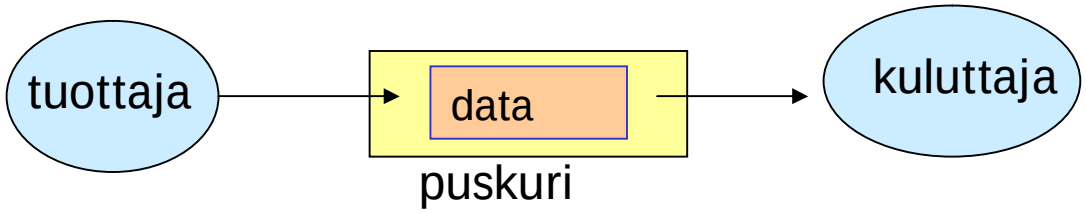
- Specifiointi- ja verifiointivälineitä
  - temporaalilogiikka
  - Spin (model checker)

    pystyvät tarkistamaan nykyisin jo miljardeja tiloja
  - BACI (concurrency simulator)
  - Paljon erilaisia

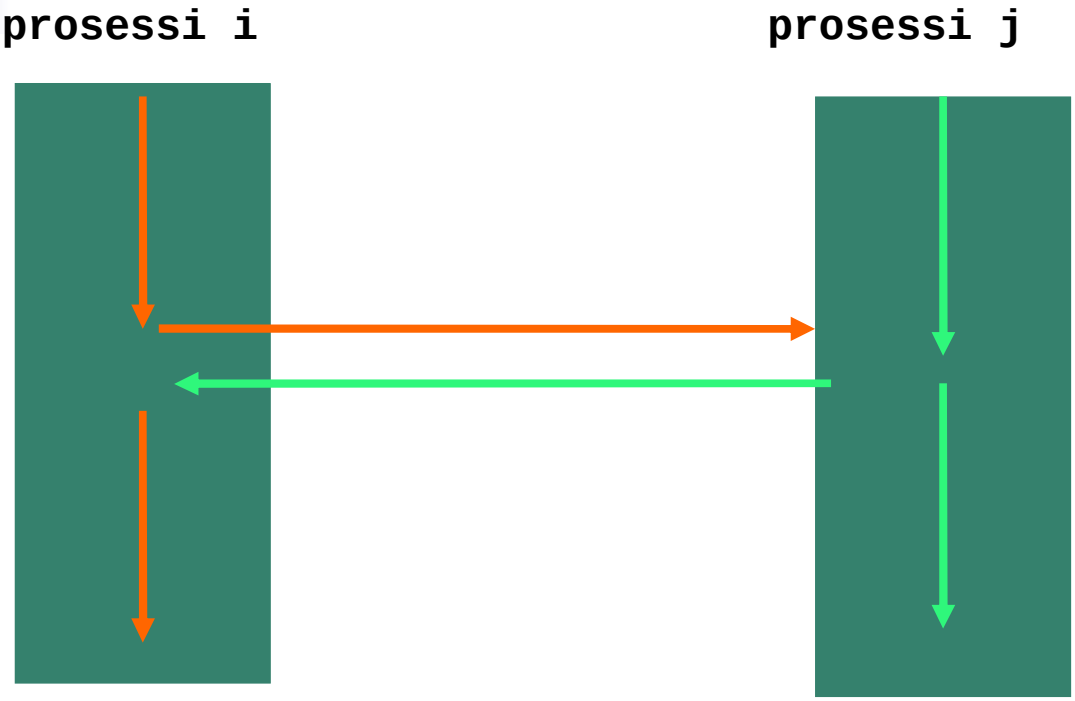
Prosessien tahdistusta eli synkronointia tarvitaan

- Keskinäiseen poissulkemiseen (mutual exclusion), jotta prosessit eivät sotke toistensa toimintaa
  - Vain yksi kerrallaan kriittisellä alueellaan
- Prosessien toiminnan koordinointiin, jotta prosessien yhteistoiminta sujuisi halutulla tavalla
  - Ehtosynkronointi (conditon synchronization)
    - prosessin eteneminen estetään, kunnes tietty ehto toteutuu
      - kuluttaja saa ottaa puskurista vasta kun tuottaja on sinne jotakin tuonut
      - tuottaja saa kirjoittaa uutta vasta kun kuluttaja on entisen lukenut

tuottaja-kuluttaja -malli



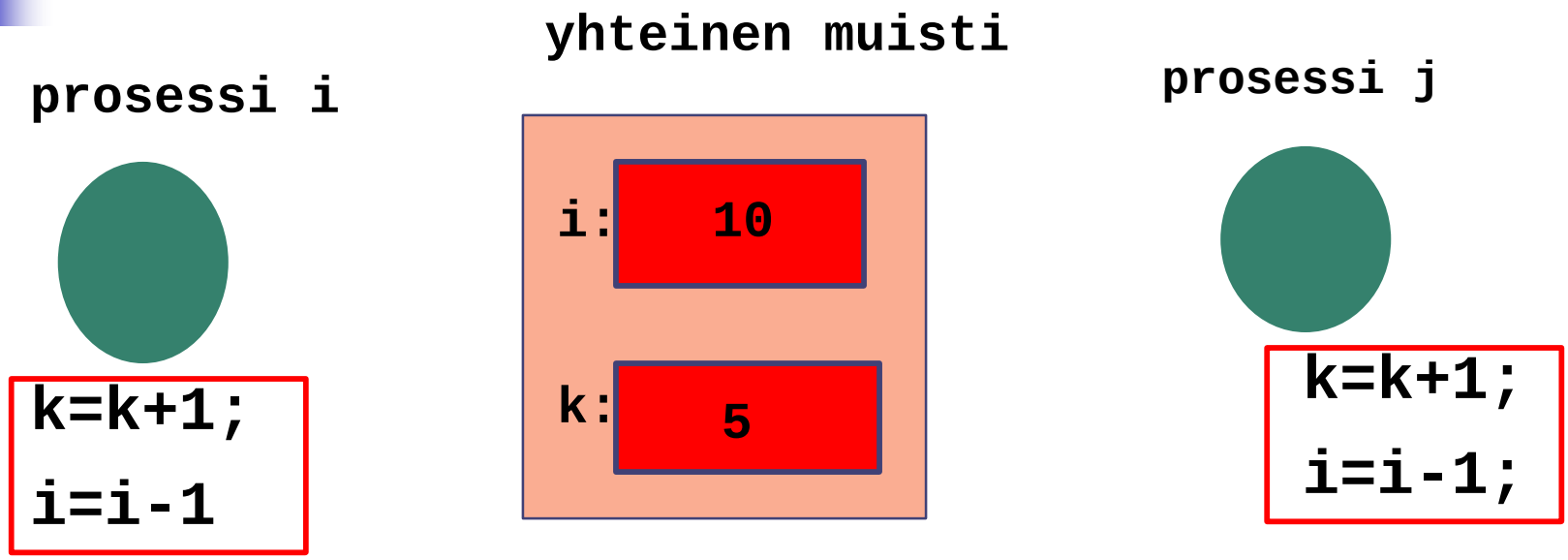
Keskinäinen kommunikointi



kommunikointiin voidaan käyttää
-yhteistä muuttujaa
-sanomanvälitystä

Prosessien ja/tai säikeiden välillä on hyvin monenlaista tarvetta kommunikointiin

Prosessit voivat häiritä toisiaan!

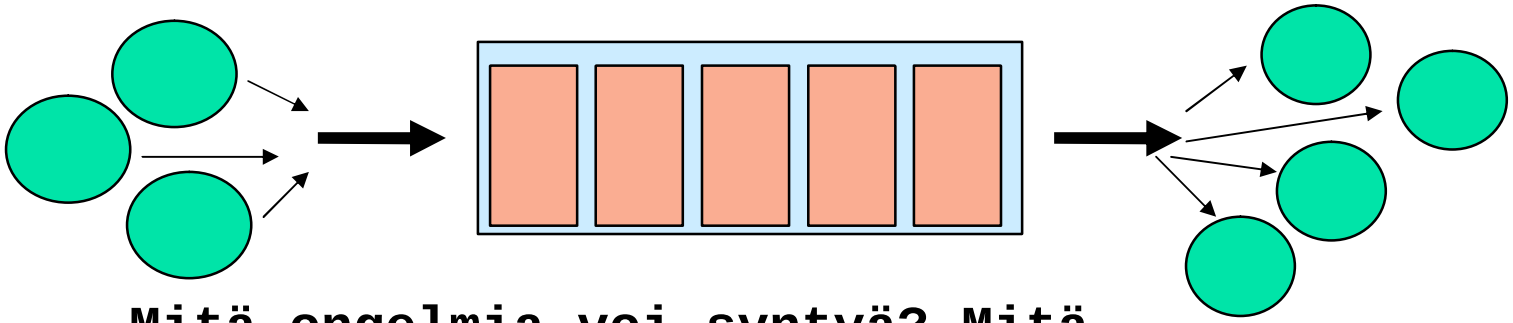


Prosessien tai säikeiden yhteistoiminta pitää varmistaa!

Tuottajat ja kuluttajat: välissä rajallinen puskuri

tuottajat kirjoittavat puskuriin

kuluttajat lukevat puskurista



Mitä ongelmia voi syntyä? Mitä toimintoja tässä pitää synkronoida?

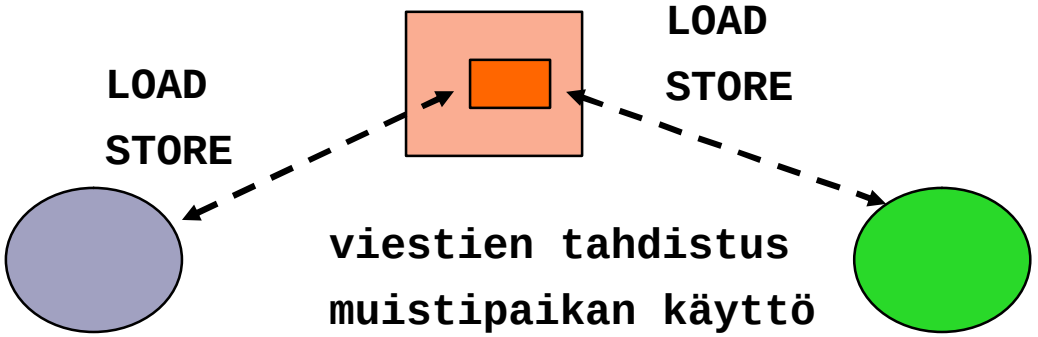
puskurin käsittely

kirjoittamisen ja lukemisen tahdistus:

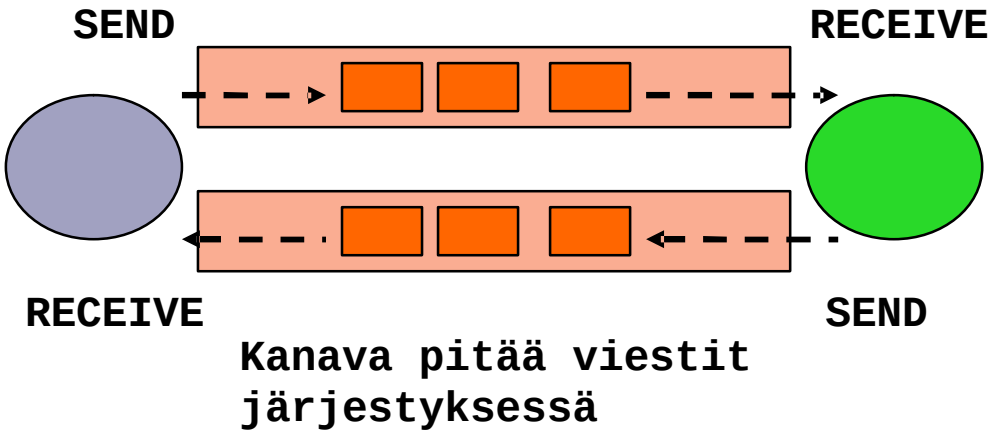
täysi puskuri ⟷ tyhjä puskuri

Kommunikointitavat

Yhteiskäyttöinen muisti



Sanomanvälitys



Kumpi kommunikointitapa?

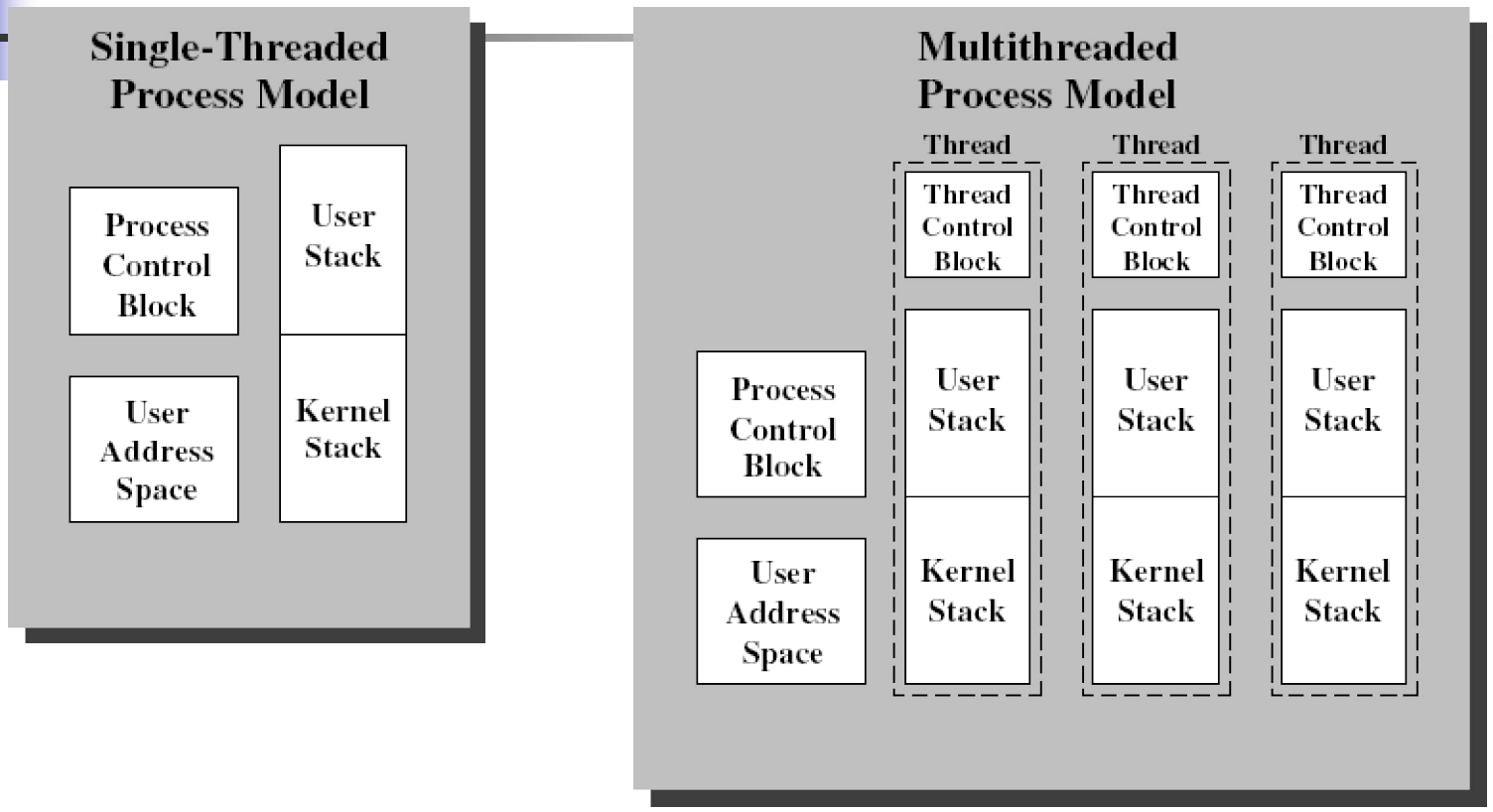
Tarkkuus / nopeus (granularity)

- 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

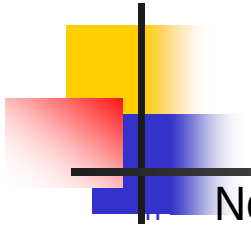
Prosessi ja sen säikeet



Single Threaded and Multithreaded Process Models

Stallings Fig 4.2

Yhteiskäyttöinen muisti / Säikeet

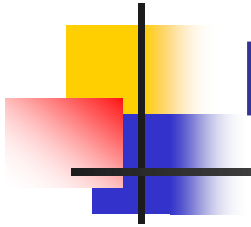


- Nopein kommunikointitapa

- Yhteiskäyttöinen muisti
  - oletusarvoisesti saman prosessin säikeille
  - KJ huolehtii viittauksista yhteisiin globaaleihin muuttujiin (kirjanpito PCB:ssä)

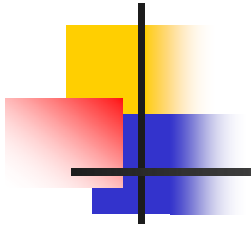
- Kullakin säikeellä oma pino
  - dynaaminen tilanvaraus aliohjelmia kutsuttaessa
  - kullakin säikeellä omat paikalliset muuttujat

- Käyttö normaaleilla muistiviitekäskyillä
  - LOAD, STORE, ...



Prosessien välinen kommunikointi

- putki (pipe) eräänlainen puskuri, joka siirtää tavuvirtaa prosessilta toiselle
  - Unix
- sanomajono (message queue) prosessi voi lähettää sanomia sanomajonojen avulla
  - Esim. Solaris, System V
- jaettu muistialue (shared memory) alue kuuluu useamman prosessin muistiavaruuteen
  - Unixissa yhteinen muistialue kaikkien aluetta jakavien prosessien virtuaalimuistia
- säie (thread)
  - yhteinen muistialue
  - oma kontrolli



POSIX-kirjasto, pthread

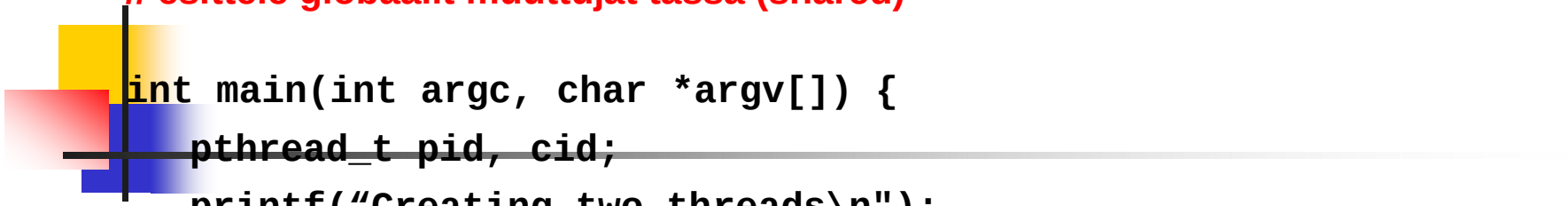
- Yli 60 funktiota

- 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()



```
#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");
}
```

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