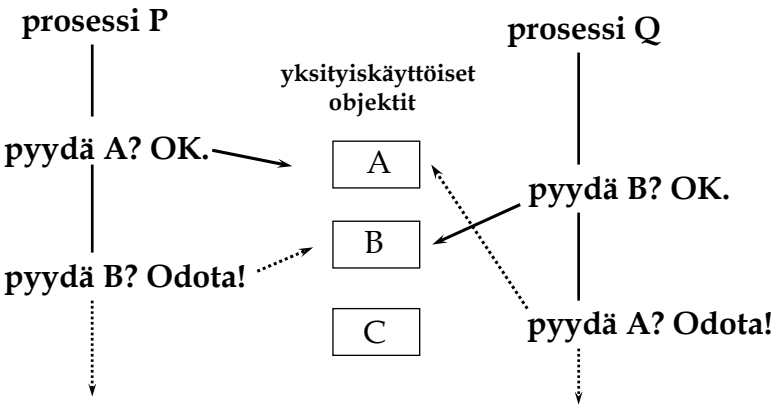
❹ Lukkiutuminen

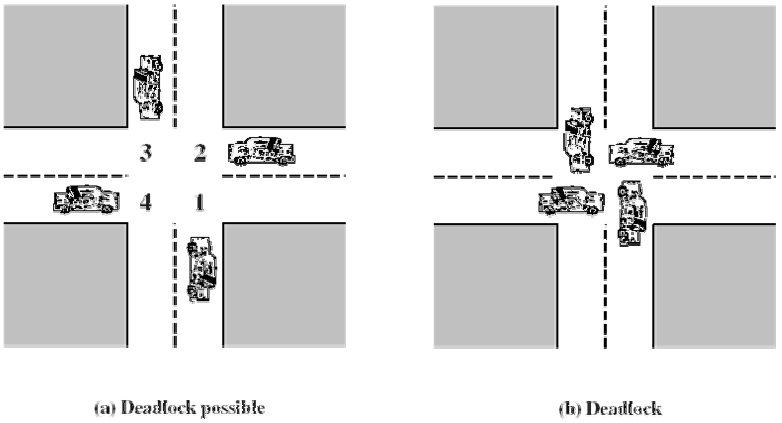
Taustaa
Aterioivat Filosofit
Ennaltaehkäisy
Havaitseminen
Välttely

Andrews 4.3
Stallings 6.1-6.6 (tai mikä tahansa KJ-kirja)

Taustaa



objekti: puskuri, sivu, skanneri, levyajuri, kriittinen vaihe, ...



(a) Deadlock possible (b) Deadlock

Kriittinen vaihe
Suoritusjärjestys, ajoitus

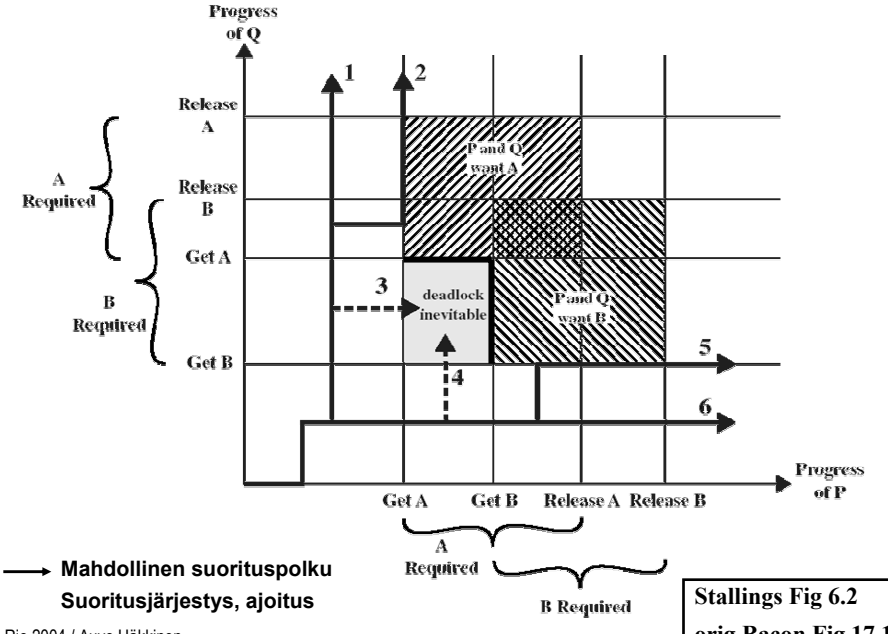
Stallings Fig 6.1

Seuraukset

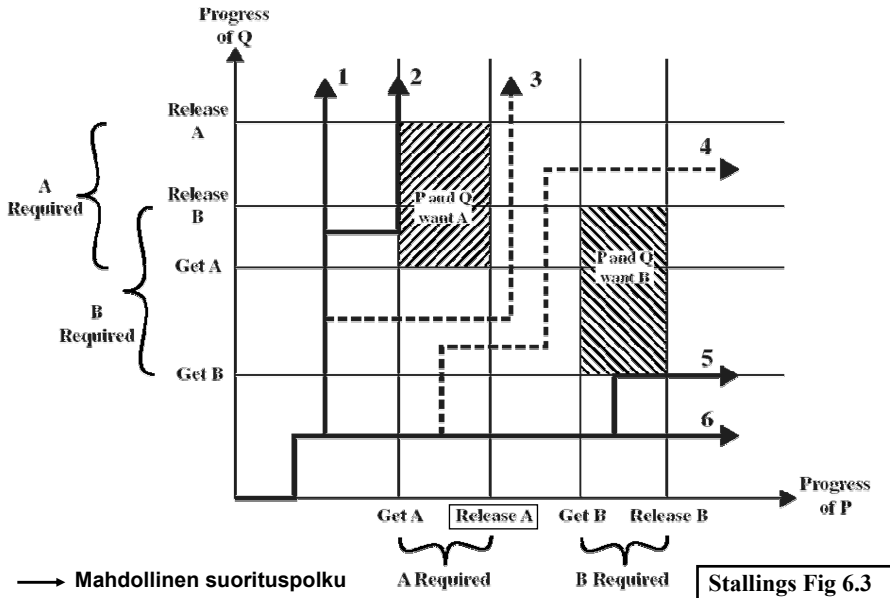
- **Prosessit eivät etene ja ...**
  niiden varaamat resurssit pysyvät varattuina
  - CPU
  - muisti, I/O-laitteet
  - loogiset resurssit (semaforit, kriittiset alueet, ...)
- **Laskenta epäonnistuu**
  - suoritus ei pääty koskaan
  - järjestelmä kaatua köllähtää
  - tilanne ei kenties toistettavissa: suoritusjärjestys

Määritelmiä

- **Lukkiuma** (deadlock)
  - päättymätön odotus BLOCKED-tilassa
- **Livelock**
  - prosessi käyttää prosessoria odottamiseen (spinlock, busy loop)
- **Ping-pong**
  - sinä ensin – eikun sinä ensin - ...
  - kaksi prosessia vuorottelevat tarjoten vuoro toiselle, eivätkä tee mitään hyödyllistä
- **Nälkiintyminen** (starvation)
  - prosessi READY-tilassa - mutta ei silti saa koskaan prosessoria



Stallings Fig 6.2
orig Bacon Fig 17.1



Stallings Fig 6.3
orig Bacon Fig 17.2

Syitä lukkiutumiseen

Kolme staattista toimintapoihin liittyvää syytä

S1 Poissulkemistarve (mutual exclusion)

- resurssilla yksi käyttäjä kerrallaan

S2 Pidä ja odota (hold and wait)

- prosessi pitää saamansa resurssin samalla, kun jää odottamaan lisäresursseja

S3 Kenelläkään ei etuoikeutta (no pre-emption)

- resurssia ei voi ottaa pois väkisin

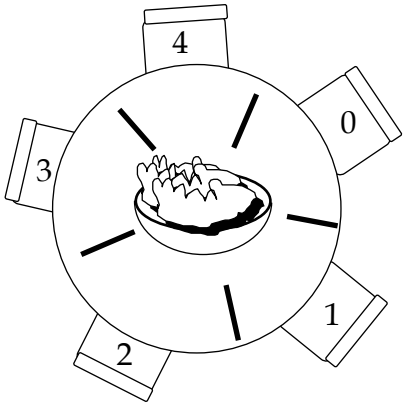
Yksi dynaaminen syy

D1 Odotus kehässä (circular wait)

- on olemassa kehä prosesseista, jotka pitävät itsellään resurssia, jota kehän seuraava prosessi tarvitsee edetäkseen

Aterioivat Filosofit

Aterioivat Filosofit (Dijkstra)



Filosofi:

aattelepa ite
ota kaksi haarukkaa ...
… yksi kummaltakin puolelta
syö, syö, syö spagettia
palauta haarukat

Kuinka varata haarukat ilman
- lukkiutumista
- nälkiintymistä

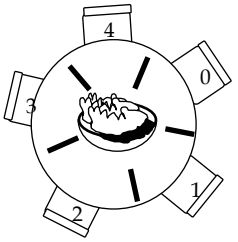
s.e. kaikki saavat olla paikalla?

Ratkaisu 1: kullekin haarukalle oma semafori

- **sem fork[0..4] = (1, 1, 1, 1, 1)**

process P[i]:

```
repeat
        think()
        P(fork[i])
        P(fork[(i + 1) mod 5])
        eat()
        V(fork[i])
        V(fork[(i+1) mod 5])
until false
```



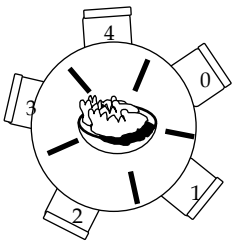
Ei OK! Miksei?

Ratkaisu 2: vain yksi voi yrittää kerrallaan

- **sem turn = 1** # säätelee yritysvuoroja

process P[i]:

```
repeat
        think()
        P(turn)
           P(fork[i])
           P(fork[(i+1) mod 5])
        V(turn)
        eat()
        V(fork[i])
        V(fork[(i+1) mod 5])
until false
```



Ei OK! Miksei?

Ratkaisu 3: OK

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
  while (true) {
    P(fork[i]); P(fork[i+1]); # get left fork then right
    eat;
    V(fork[i]); V(fork[i+1]);
    think;
  }
}
process Philosopher[4] {
  while (true) {
    P(fork[0]); P(fork[4]);   # get right fork then left
    eat;
    V(fork[0]); V(fork[4]);
    think;
  }
}
```

Andrews Fig. 4.7

Ratkaisu 4: OK, mutta... vrt. 2

```
sem fork[5] = ([5] 1);
sem room = 4;

process Philosopfer[i=0 to 4]
{
   while (true) {
       think();
       P(room);
            P(fork[i]);
            P(fork[(i+1) mod 5]);
            eat();
            V(fork[i]);
            V(fork[(i+1) mod 5]);
       V(room);
   }
}
```

Stallings Fig. 6.12

Ratkaisu 5: OK? Nälkiintyminen?

```
#define N               5           /* number of philosophers */
#define LEFT            (i+N–1)%N   /* number of i's left neighbor */
#define RIGHT           (i+1)%N     /* number of i's right neighbor */
#define THINKING        0           /* philosopher is thinking */
#define HUNGRY          1           /* philosopher is trying to get forks */
#define EATING          2           /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Tanenbaum Fig. 2.33

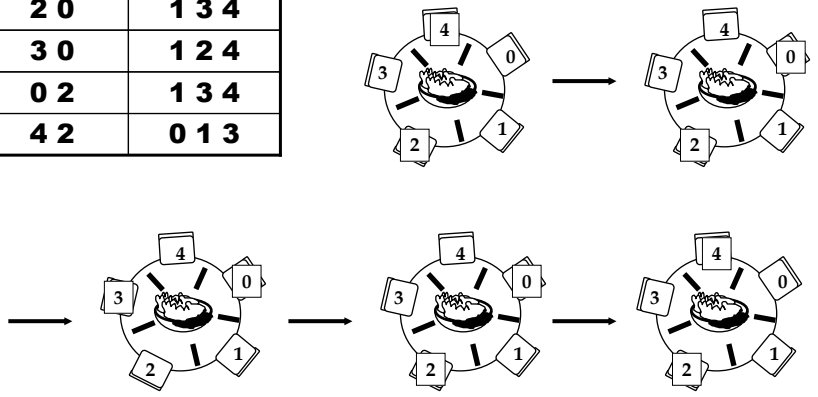
```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Huom: down() = P(), up() = V()

Tanenbaum Fig. 2.33

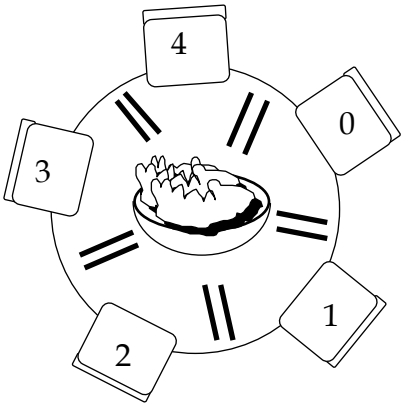
Ratkaisu 5: ”Tapettaisko filosofi 1?”

EATING	HUNGRY
4 2	0 1 3
2 0	1 3 4
3 0	1 2 4
0 2	1 3 4
4 2	0 1 3



Ratkaisu 6: OK, ei yhteisiä resursseja

Ostakaa 5 haarukkaa lisää!



Filosofi[i]:

- aatteleppa ite
- ota kaksi haarukkaa
- …yksi molemmilta puolilta
- syö, syö, syö spagettia
- palauta haarukat

Lukkiuman ennaltaehkäisy

Stallings 6.2

Ennaltaehkäisy

= Poista joku syistä S1, S2, S3 tai D1

<u>Ehkäise S1 (poissulkemistarve)?</u>

= käytä yhteiskäyttöisiä resursseja yksittäiskäyttöisten sijasta, spooling

(= hanki enemmän resursseja)

- **poissulkemistarpeelle omat syynsä, mutta**
  - hienojakoisempi toteutus sallii paremmin rinnakkaisuutta
    ⇨ pienemmät alueet, yhtäaikainen käyttö, enemmän lukkoja
  - yleistä esim. tietokantojen yhteydessä, lyhyet transaktiot

Ehkäise S2 (pidä ja odota)?

= pyydä kaikki resurssit yhdellä kertaa
= odota, että saat kaikki kerralla

- **tehotonta**
  - pitkät odotusajat: varaa nyt - käytä paljon paljon myöhemmin
  - varautuminen pahimpaan: varaa resursseja, joita ei ehkä tarvitakaan
- **vaikea / mahdoton toteuttaa?**
  - tiedettävä etukäteen mitä resursseja tarvitaan kaikilla mahdollisilla suorituspoluilla

Ehkäise S3 (ei etuoikeuksia)?

= jos varaaminen ei onnistu, vapauta jo varatut resurssit, peruuta edelliseen tilanteeseen

- kokeile alustavasti varausta, tai
- tarkistuspisteet
- manageri?

- **OK, jos järjestelmä suunniteltu tämä mielessä**
  - käytännöllinen, jos tilatiedon talletus helppoa ja lukkiutumisriski aidosti olemassa
  - normaalia käytäntöä transaktioiden käsittelyssä

Ehkäise D1 (odottaminen kehässä)?

= numeroi resurssit lineaarisesti,
varaa numerojärjestyksessä

- etukäteistietoa resurssitarpeesta,
  varauduttu pahimpaan tapaukseen

  pessimistinen
- TAI
  varaa sitä mukaa kun tarve (järjestyksessä),
  tarvittaessa palaa aiempaan tilaan

  optimistinen

Ratkaisu 5: Ehkäise S2 (pidä ja odota) = varaa kaksi haarukkaa yhtäaikaa

process P[i]:

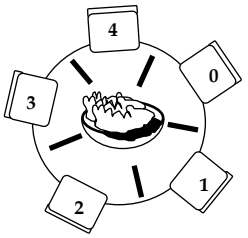
```
repeat
      think( )
      take_ forks(i, (i+1) mod 5)
      eat( )
      put_forks( i, (i+1) mod 5)
until false
```

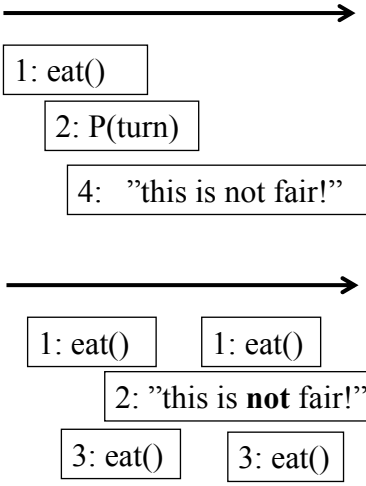
take & put:
- resurssin hallinta

tehtävät:
- vuorojen antaminen
- lukkiutuman ehkäisy
- nälkiintymisen ehkäisy

Vaarallisten 'kohtien' metsästys

```
P(turn)
   P(fork[i])
   P(fork[(i+1) mod 5])
V(turn)
```





Lukkiuman havaitseminen

Stallings 6.4

Lukkiuman havaitseminen

- **Havaitseminen vaikeaa**
  - muodostaako ryhmä prosesseja lukkiuman,
  - vai odottavatko ne ulkoista tapahtumaa?
- **Jotta voisi toipua, pystyttävä havaitsemaan**
  - toipuminen:
    - peruutus edeltävään tilanteeseen
    - tapa yksi tai useampi lukkiumaan kuuluva prosessi
- **KJ tarvitsee tietoa resurssien allokoinnista**
  - sellaisessa muodossa, että voi havaita lukkiuman!
  - tutki aika-ajoin onko lukkiumaa

Resurssien (objektien) allokointi

- **Prosessit** P_i $i=1..m$
- **Resurssit** R_j $j=1..n$
  - yhteensä $R = (R_1, ..., R_n)$
  - vapaana $V = (V_1, ..., V_n)$
- **Allokointimatriisi** $A [P_i, R_j]$
  - montako resurssia Rj allokoitu prosessille Pi
- **Pyyntömatriisi** $Q [P_i, R_j]$
  - montako resurssia Rj prosessi Pi pyytänyt

Esim.: Alkutilanne

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

Prosessilla 2 on resurssit 1 ja 2, ja se haluaa resurssit 3 ja 5.

Kenellä on resurssi 4?

Mitkä resurssit ovat vapaana?

Onko tässä lukkiuma vai ei?

Stallings Fig. 6.9

DDA: Deadlock Detection Algorithm (Dijkstra)

DDA① *[Poista prosessit, joille ei ole allokoitu resursseja]*
Merkitse käsitellyiksi kaikki tyhjät rivit matriisissa A.

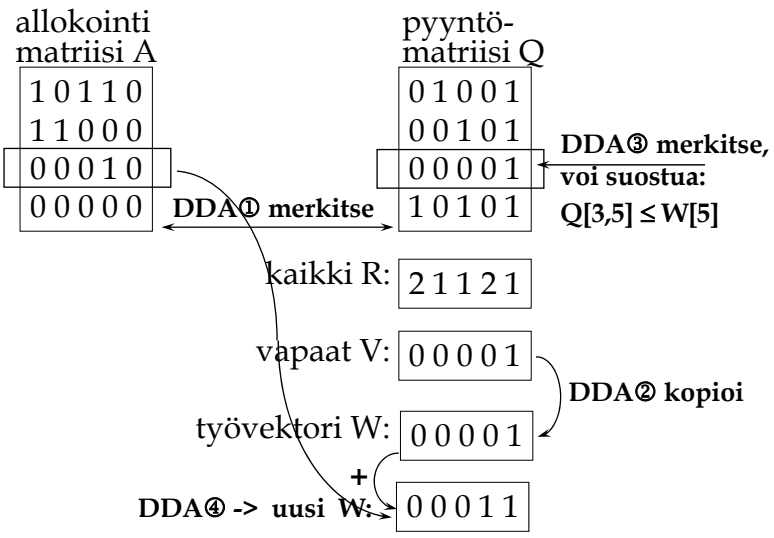
DDA② *[Alusta vapaiden resurssien laskurit]*
Alusta työvektori $W = V$

DDA③ *[Etsi prosessi, jonka maksimipyyntöön voi suostua]*
Etsi merkitsemätön rivi Pi siten, että
$Q[Pi,Rj] \leq W[Rj] \qquad j = 1..n$
Jos ei löydy, algoritmi on päättynyt.

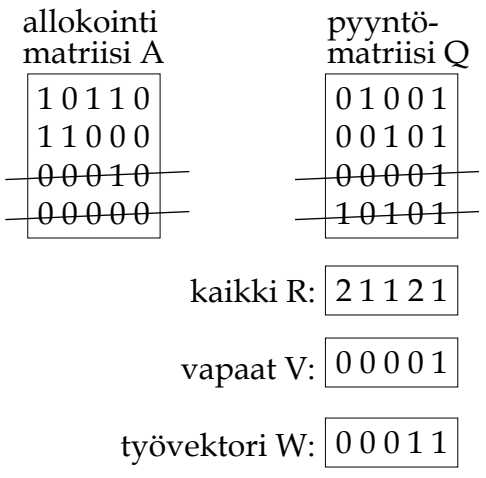
DDA④. *[Oleta, että prosessi suoritettu, ja vapauta varaukset]*
Aseta $W = W+A[Pi]$ ts.. $W[Rj] = W[Rj]+A[Pi,Rj] \quad j = 1..n$
Merkitse rivi Pi käsitellyksi ja palaa askeleeseen DL3.

Kun algoritmi päättyy, merkitsemättömät rivit osoittavat lukkiumaan kuuluvat prosessit. Miksi?

Esim.: Vaihe 1



Esim.: Vaihe 2



DDA③ ei prosessia Pi s.e. Q[Pi,Rj] ≤ W[Rj] ∀j.

Algoritmi päättyy
→ prosessit 1 ja 2 lukkiutuneet

Mitä sitten?

Bacon Fig. 17.9: Graphs…

Lukkiuman välttely

Stallings 6.3

Pankkiirin algoritmi

- **Alkujaan yhdelle resurssille (raha)**
- **Miksi ”Pankkiirin”?**
  - varmistettava, että pankki ei koskaan uloslainaa niin paljon, ettei se pysty tyydyttämään kaikkien talletusasiakkaiden maksimitarpeita
  - (Huom: pankilla myös omaa pääomaa)
- **Varmista ensin, allokoi vasta sitten**

Pankkiirin algoritmi useille resursseille (Dijkstra)

- **Kerää tilatietoa**
  - Prosesseille jo allokoiduista resursseista
  - Resursseista, joita kukin prosessi voisi edelleen pyytää
- **Varmista pyynnön vaikutus etukäteen**
- **Allokoi vasta, kun varma ettei johda lukkiumaan pahimmassakaan tapauksessa**
  - Tarkista, että löytyy ainakin yksi vuorottelujärjestys, jossa kaikki prosessit voivat suoriutua loppuun
  - vaikka muut pyytäisivät maksimitarpeensa

`"worst case analysis"`

- **Allokointi- ja pyyntömatriisit kuten edellä**

LISÄKSI

- **Resurssien maksimipyyntö C[Pi,Rj]**
  - montako resurssia Rj prosessi Pi voisi maksimissaan pyytää
  - prosessien kerrottava etukäteen!
- **Mahdollinen uusi allokointimatriisi A'[Pi,Rj]**
  - montako resurssia Rj prosessilla Pi olisi, jos sille annettaisiin sen maksimitarpeen mukaan
- **Mahdollinen uusi pyyntömatriisi Q'[Pi,Rj]**
  - montako resurssia Rj prosessi Pi voisi vielä tämän jälkeen pyytää
  - Q' = C - A'

Alkuperäisiä ei voi vielä muuttaa!

Mieti allokoinnin seurauksia *etukäteen*:

- **Oleta, että allokointi tehdään**
- **Laske uusi *mahdollinen* allokointimatriisi A'**
- **Laske uusi *mahdollinen* pyyntömatriisi Q'**
  - pahimmassa tapauksessa,
    ts. tilanteessa jossa kaikki pyytävät oman maksimimääränsä
- **Sovella lukkiuman havaitsemisalgoritmia DDA matriiseille A' ja Q'**
- **Jos ei johda lukkiumaan, suostu pyyntöön muuten älä suostu pyyntöön**
  - jätä prosessi odottamaan
  - kun joku vapauttaa, tarkista uudelleen

Allokointimatriisi A

	R1	R2	R3	R4	R5
P1	0	1	0	0	0
P2	1	1	0	0	0
P3	0	0	1	0	1
P4	0	0	1	1	0

Pyyntömatriisi Q

R1	R2	R3	R4	R5
1	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	0	1

Maksimipyyntö C

R1	R2	R3	R4	R5
2	1	0	1	0
1	1	0	0	1
1	0	1	1	1
0	2	1	1	1

Resurssivektori R

R1	R2	R3	R4	R5
2	3	2	1	2

Vapaana V

R1	R2	R3	R4	R5
1	1	0	0	1

Voiko P1:n pyyntöön suostua?
Voisiko lukkiutua?

Jos P1:n pyyntöön suostuttaisiin niin...

Allokointimatriisi A'

	R1	R2	R3	R4	R5
P1	1	1	0	0	0
P2	1	1	0	0	0
P3	0	0	1	0	1
P4	0	0	1	1	0

Pyyntömatriisi Q'

R1	R2	R3	R4	R5
1	0	0	1	0
0	0	0	0	1
1	0	0	1	0
0	2	0	0	1

Maksimipyyntö C

R1	R2	R3	R4	R5
2	1	0	1	0
1	1	0	0	1
1	0	1	1	1
0	2	1	1	1

Resurssivektori R

2	3	2	1	2

Vapaana V

R1	R2	R3	R4	R5
1	1	0	0	1

Vapaana V'

R1	R2	R3	R4	R5
0	1	0	0	1

Työvektori W

R1	R2	R3	R4	R5
0	1	0	0	1

Työvektori W

1	2	0	0	1

DDA④ merkkaa rivi 2

Työvektori W

1	2	1	1	1

DDA④ merkkaa rivi 4

Työvektori W

2	3	1	1	1

DDA④ merkkaa rivi 1

Työvektori W

2	3	2	1	2

DDA④ merkkaa rivi 3

⇨ Voi suostua!

Mitä, jos ei voi?

Ongelmia

- **Yleisrasite**
  - tutki jokaisella pyynnöllä...
  - 20 prosessia ja 100 resurssia?
- **Prosessin tiedettävä maksimitarve**
  - etukäteen
    - viksu arvaus? pahin tapaus?
  - dynaamisesti
    - vieläkin rasittavampaa
- **Aina ei löydy varmaa allokointijärjestystä**
  - ei-turvallinen tila ei aina johda lukkiumaa – voiko ottaa riskin!

Kertauskysymyksiä?