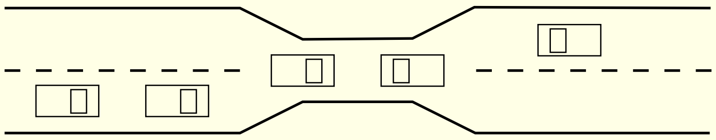
Tupikud

Ülevaade

- Süsteemi mudel
- Tupiku tingimused
- Tupikute käsitlemise meetodid
- Tupikute ennetamine
- Tupikute vältimine
- Tupikute avastamine
- Tupikust taastumine
- Kombineeritud lähenemine

Tupikute probleem

- Tupik — hulk blokeerunud protsesse, millest igaüks ootab mingi ressursi taga, mida kasutab mõni teine neist protsessidest
- Näide: autod kitsal sillal



- Näide: 4 ristmikku
- Näide: kaks semafori A ja B, algväärtusega 1

P_i	P_j
wait(A);	wait(B);
wait(B);	wait(A);

Süsteemi mudel

- Ressursitüübid $R_1, R_2, \ldots, R_m$
- Näiteks protsessoriaeg, mäluruum, välisseadmed
- Igast ressursist R_i on W_i instantsi
- Iga protsess kasutab mingit ressurssi niisuguses järjestuses:
  - hõivamine (*request*)
  - kasutamine
  - vabastamine (*release*)

Tupiku tingimused

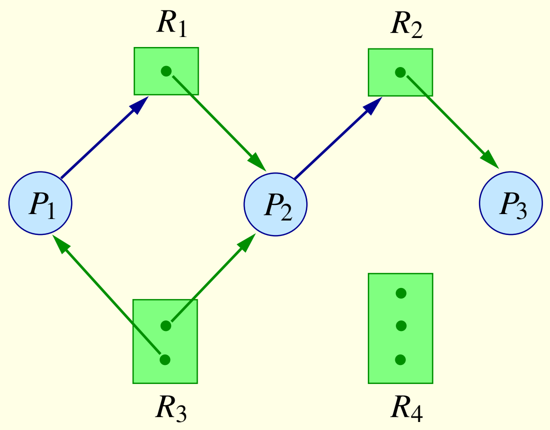
Tupiku tekkimiseks peavad kõik järgnevad tingimused olema rahuldatud:

- Vastastikune välistamine — ressurssi saab korraga kasutada ainult üks protsess
- Hoidmine ja ootamine — protsess hoiab vähemalt ühte ressurssi enda käes ja ootab järgmise ressursi hõivamisel
- Pole väljatõrjumist — ressursse vabastatakse ainult vabatahtlikult (pärast kasutamise lõpetamist), neid ei saa jõuga käest ära võtta
- Tsükliline ootamine — leidub hulk protsesse $P_0, P_1, \ldots, P_n$, nii et P_i ootab P_{i+1} taga ($0 \le i \le n-1$) ja P_n ootab P_0 taga

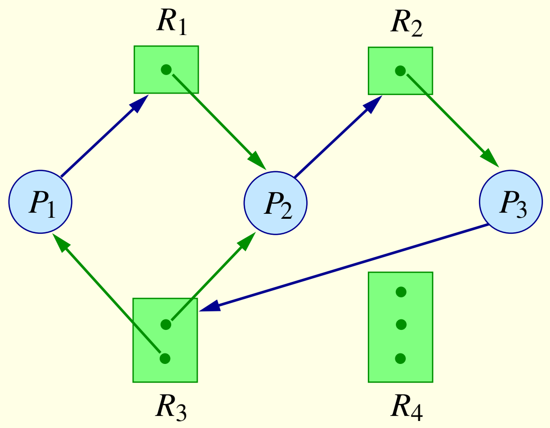
Ressursigraaf

- Protsessidele eraldatud ressursside esitamiseks
- Tippude hulk V ja servade hulk E
- V jaguneb kahete tüüpi tippudeks:
  - $P = \{P_1, \ldots, P_n\}$ — süsteemi kõigi protsesside hulk
  - $R = \{R_1, \ldots, R_m\}$ — süsteemi kõigi ressursitüüpide hulk
- Soovi serv (*request edge*) — suunatud serv $P_i \rightarrow R_j$
- Kasutusserv (*assignment edge*) — suunatud serv $R_j \rightarrow P_i$

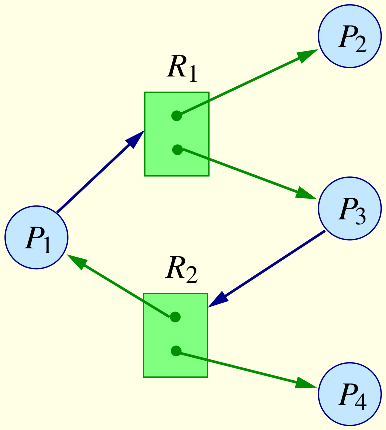
Ressursigraaf



Tupikuga ressursigraaf



Tsükliga tupikuta ressursigraaf



Ressursigraafid ja tupikud

- Graafis pole tsükleid $\Longrightarrow$ pole ka tupikuid
- Graafis on tsükkel $\Longrightarrow$
  - Kui iga ressurssi on ainult üks, siis tupik
  - Kui mõnda ressurssi on mitu instantsi, siis on tupiku võimalus

Tupikute käsitlemine

- Tupikute ennetamine ja vältimine — garanteerime, et süsteem ei satu kunagi tupikusse
- Tupikute avastamine — lubame süsteemi tupikusse sattuda, avastatud tupikud likvideerime
- Tupikute eiramine — ignoreerime probleemi ja teeme näo, et tupikuid kunagi ei juhtugi
  - ”Jaanalinnu algoritm” (Ingl. *The Ostrich Algorithm*)
  - Omab mõtet, kui tupikute tekkimise tõenäosus on suhteliselt väike ja/või nende vältimine on liiga kulukas
  - Kasutusel enamuse operatsioonisüsteemide poolt, näiteks UNIX, Windows

Tupikute ennetamine (1)

- Ingl. *deadlock prevention*
- Põhiidee: piirame võimalusi ressursipäringute tegemiseks, nii et vähemalt üks neljast tupiku tingimusest oleks alati väär
- Vastastikune välistamine — pole vajalik jagatavate ressursside puhul, siiski kohustuslik mittejagatavate ressursside puhul
- Hoidmine ja ootamine — garanteerime, et kui protsess küsib mingit ressurssi, siis ta ei omaks teisi ressursse
  - Laseme rakendusel kõik kasutatavad ressursid käivitamise alguses ära küsida või lubame ainult siis uusi ressursse võtta, kui vanad on vabastatud
  - Ebaefektiivne ressursikasutus, võimalik näljutamine

Tupikute ennetamine (2)

- Pole väljatõrjumist — kui protsess hoiab ressursse ja küsib uusi, mida kohe ei saa, siis anname vanad ressursid kah seni vabaks ja ta ootab nüüd ka nende järgi
  - Alternatiiv — kui küsitav ressurss pole vaba, aga selle on hõivanud mõni parajasti ootel olev protsess, võtame selle talt ära
- Tsükliline ootamine — kasutame ressursside täielikku järjestust ja laseme igal protsessil võtta juurde ainult sellist tüüpi ressursse, mille number pole väiksem ühestki olemasolevast

Tupikute vältimine

- Ingl. *deadlock avoidance*
- Süsteem vajab lisainfot selle kohta, kuipalju ja missuguseid ressursse protsessid tulevikus kasutada tahavad
- Lihtsaim ja kasulikeim mudel käsib igal protsessil deklareerida maksimaalsed vajadused ressursside kohta
- Tupikute vältimise algoritm uurib dünaamliselt ressursitabelit, et ei saaks tekkida tsüklilise ootamise situatsiooni
- Ressursside hõivatuse oleku defineerivad vabade ja hõivatud ressursside arvud ja protsesside maksimaalsed nõudmised neile

Ohutud olekud (1)

- Kui protsess nõuab ressurssi, siis süsteem peab otsustama, kas selle ressursi eraldamise tulemusena oleks süsteemi olek ohutu
- Olekut loetakse ohutuks, kui leidub ohutu järjestus protsessidest
- Protsesside järjestus $P_1, P_2, \ldots, P_n$ on ohutu, kui iga P_i nõudmised saab rahuldada selleks hetkeks vabade ning protsesside P_j $(j < i)$ poolt kasutatavate ressurssidega
  - Kui protsessi P_i ressursivajadused pole koheselt rahuldatavad, siis P_i ootab, kuni ressursse kinnihoidvad protsessid P_j on lõpetanud
  - Kui protsessid P_j on lõpetanud, saab P_i vajaminevad ressursid oma käsutusse ning vabastab need peale kasutamist
  - Seejärel saab protsess P_{i+1} talle vajaminevad ressursid jne.

Ohutud olekud (2)

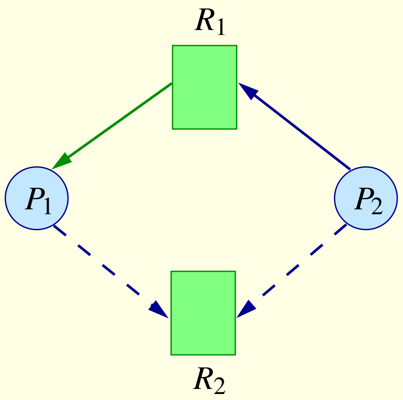
- Kui süsteem on ohutus olekus $\Longrightarrow$ pole tupikuid
- Kui süsteem on ohtlikus (so. ei ole ohutus) olekus $\Longrightarrow$ tupikud võimalikud (aga mitte kõik ohtlikud olekud ei ole tupikud)
- Tupikute vältimiseks garanteerime, et süsteem ei satuks ohtlikku olekusse

Ressursigraafi algoritm

- Kui igast ressursist on üks eksemplar, saab ohtlike olekute kindlakstegemiseks kasutada modifitseeritud ressursigraafi
- Nõudmise serv $P_i \rightarrow R_j$ näitab, et protsess P_i võib nõuda ressurssi R_j (tähistame punktiiriga)
- Nõudmise serv muutub soovi servaks, kui protsess reaalselt nõuab seda ressurssi
- Ressursi vabastamisel läheb serv tagasi nõudmise servaks
- Nõudmised peavad ette teada olema
- Kui graafis leidub tsükkel, siis on tegemist ohtliku olekuga

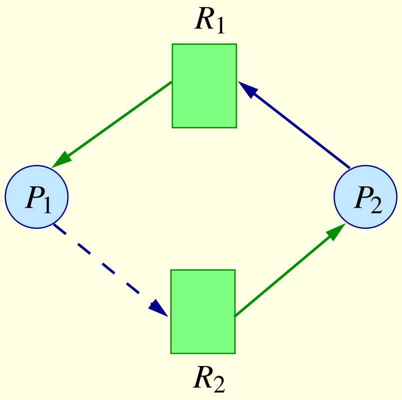
Ressursigraafi algoritmi näide

- Ohutus olekus oleva süsteemi ressursigraaf



Ressursigraafi algoritmi näide

- Ressursi R_2 andmisel protsessile P_2 muutub olek ohtlikuks



Pankuri algoritm

- Igast ressursist palju instantse
- Iga protsess peab ette ära ütlema maksimaalse kasutuse
- Ressursi küsimisel võib protsess ootele jääda
- Ressursid tuleb lõpliku aja jooksul tagastada

Pankuri algoritmi andmestruktuurid

- Olgu n protsesside arv ja m ressursitüüpide arv
- $Available$ — vektor pikkusega m. $Available[j] = k$ tähendab, et ressurssi R_j on saadavalt k tükki
- Max — $n \times m$ maatriks. $Max[i, j] = k$ tähendab, et P_i tohib küsida max. k instantsi ressursist R_j
- $Allocation$ — $n \times m$ maatriks. Näitab, kui palju ressursse hetkel protsesside poolt kasutusel on
- $Need$ — $n \times m$ maatriks. Näitab, kui palju ressursse veel on vaja töö lõpetamiseks

$$Need[i, j] = Max[i, j] - Allocation[i, j]$$

Pankuri algoritm — oleku ohutuse leidmine

1. Olgu $Work$ ja $Finish$ vektorid pikkusega m ja n; initsialiseerime:
   $Work = Available$
   $Finish[i] = false \quad (i = 1 \ldots n)$
2. Leiame sellise i, mille korral:
   $Finish[i] = false$
   $Need_i \leq Work$
   Kui sellist i ei leidu, mine sammule 4
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Mine sammule 2
4. Kui iga i korral $Finish[i] == true$, siis on süsteem ohutus olekus

Pankuri algoritm — ressursi küsimine

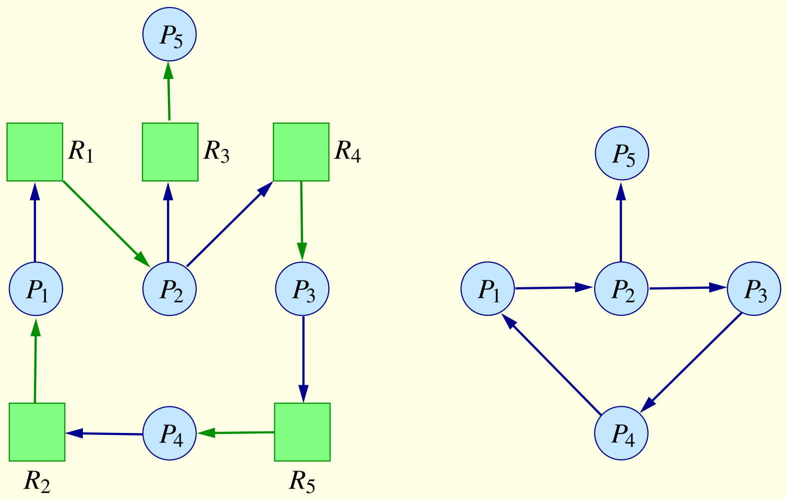
$Request_i$ — järgmisena küsitavate ressursside vektor

1. Kui $Request_i \leq Need$, siis mine sammule 2; vastasel korral viga, kuna protsess on ületanud oma maksimaalsed ressursinõuded
2. Kui $Request_i \leq Available$, siis mine sammule 3; vastasel korral peab protsess P_i ootama, kuna ressursid ei ole vabad
3. Simuleerime nõudmise täitmist:
   $Available = Available - Request_i$
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Need_i - Request_i$
   - Kui saadav olek on ohutu, siis anname protsessile need ressursid
   - Vastasel korral peab protsess ootama ning muutujate vanad väärtused taastatakse

Tupikute avastamine (1)

- Laseme süsteemi tupikusse, avastame selle ja kõrvaldame tupiku
- Kui igast ressursitüübist on üks eksemplar, siis kasutame ootamisgraafi
- Ootamisgraaf on ressursigraafi modifikatsioon, kust
  - ressursitipud on eemaldatud
  - ja vastavad servad on kokku viidud.
- Aegajalt otsime sellest graafist tsükleid (operatsiooni keerukus n^2)

Ressursigraaf ja ootamisgraaf



Tupikute avastamine (2)

Mitme ressursiinstantsi korral kasutame modifitseeritud pankuri algoritmi

1. Olgu *Work* ja *Finish* vektorid pikkusega m ja n; initsialiseerime:

   $Work = Available$

   $Finish[i] = (Allocation_i == 0) \qquad (i = 1 \ldots n)$

2. Leiame sellise i, mille korral:

   $Finish[i] = false$

   $Request_i \leq Work$

   Kui sellist i ei leidu, mine sammule 4

3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   Mine sammule 2

4. Kui $Finish[i] == false$, siis on protsess P_i tupikus

Tupikute avastamine (3)

- See, millal ja kui tihti tupikute avastamise protseduuri kasutada, sõltub
  - Tupikute tekkimise tõenäosusest
  - Tupikutest mõjutatavate protsesside arvust
- Liiga sagedane kasutamine on ebaefektiivne
- Liiga harv kasutamine teeb keeruliseks tupikut "põhjustava" protsessi leidmise

Tupikutest taastumine

- Protsesside termineerimine
  - Tapame kõik tupikusse sattunud protsessid
    - väga kulukas
  - Tapame tupikus protsesse ükshaaval, kuni tupik on kõrvaldatud
    - samuti suhteliselt kulukas
    - millises järjekorras (prioriteedid, arvutusaeg, ressursside olemasolu/vajadus, . . .)
- Ressursi väljatõrjumine
  - Valime ohvri — so. tupikus oleva protsessi, mille katkestamine on kõige "odavam"
  - Kerime tagasi — so. pöördume tagasi mõnda ohutusse olekusse ja taaskäivitame protsessi sealt
  - Näljutamine — võib juhtuda, et alati valime sama ohvri

Kombineeritud meetodid

- Grupeerime ressursid erinevatesse, hierarhiliselt järjestatud, klassidesse; näit.
  - Sekundaarne mälu
  - Protsessiressursid (IO sedmed, failid, . . .)
  - Põhimälu, . . .
- Klasside vahel kasutame tsüklilise ootamise ennetamist
- Iga klassi sees kasutame seal kõige rohkem sobivat meetodi (so. kas ennetamist, vältimist, avastamist)