Operativni sistemi

- Konkurentnost: uzajamno blokiranje i gladovanje -

Prof. dr Dragan Stojanović

Katedra za računarstvo
Univerzitet u Nišu, Elektronski fakultet

Literatura

- *Operating Systems: Internals and Design Principles*, edition, W. Stallings, Pearson Education Inc., 7th – 2012, (5th -2005, 6th - 2008, 8th – 2014 , 9th – 2017)
  - http://williamstallings.com/OperatingSystems/
  - http://williamstallings.com/OperatingSystems/OS9e-Student/

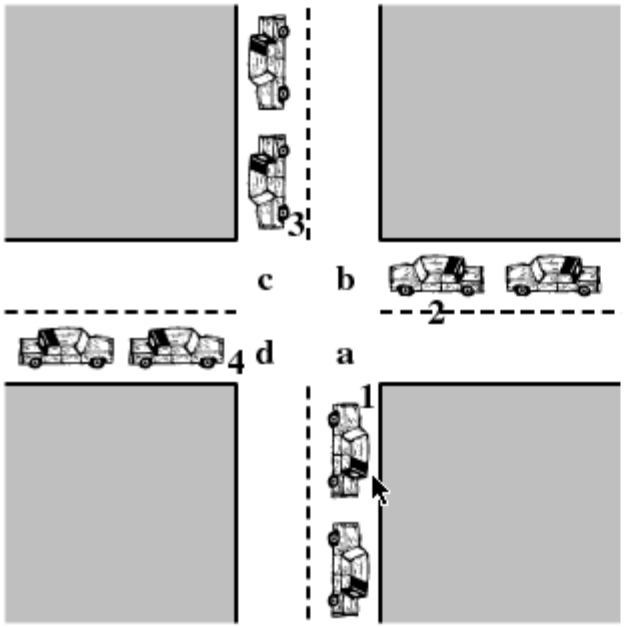
- Poglavlje 6: Konkurentnost: uzajamno blokiranje i gladovanje
- Dodatak A: Teme o konkurentnosti

Uzajamno blokiranje

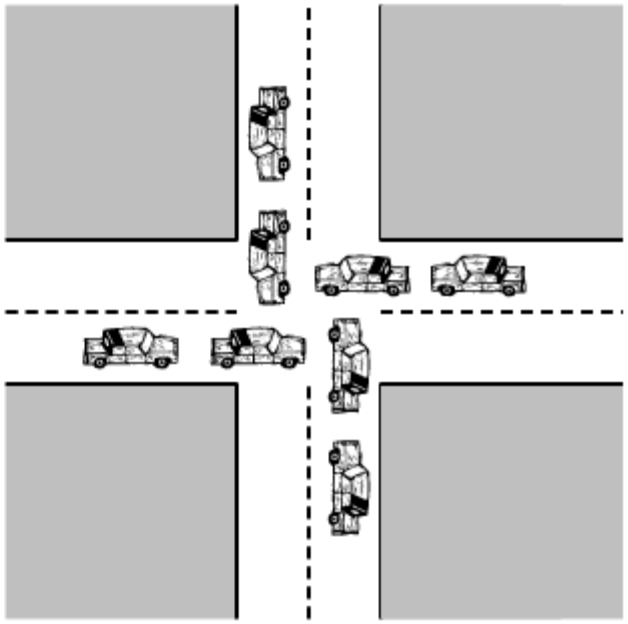
- Šta je uzajamno blokiranje?
  - **Trajno** uzajamno blokiranje skupa procesa koji se nadmeću za resurse sistema ili međusobno komuniciraju
- Skup procesa je uzajamno blokiran kada je svaki proces iz ovog skupa blokiran čekajući na događaj koji može aktivirati samo neki od procesa iz tog skupa
  - Uključuje procese koji se nadmeću za resurse iz istog skupa
- Ne postoji efikasno rešenje uzajamnog blokiranja

Ilustracija uzajamnog blokiranja

4 kvadranta raskrsnice su resursi koje treba kontrolisati



Potencijalno uzajamno blokiranje



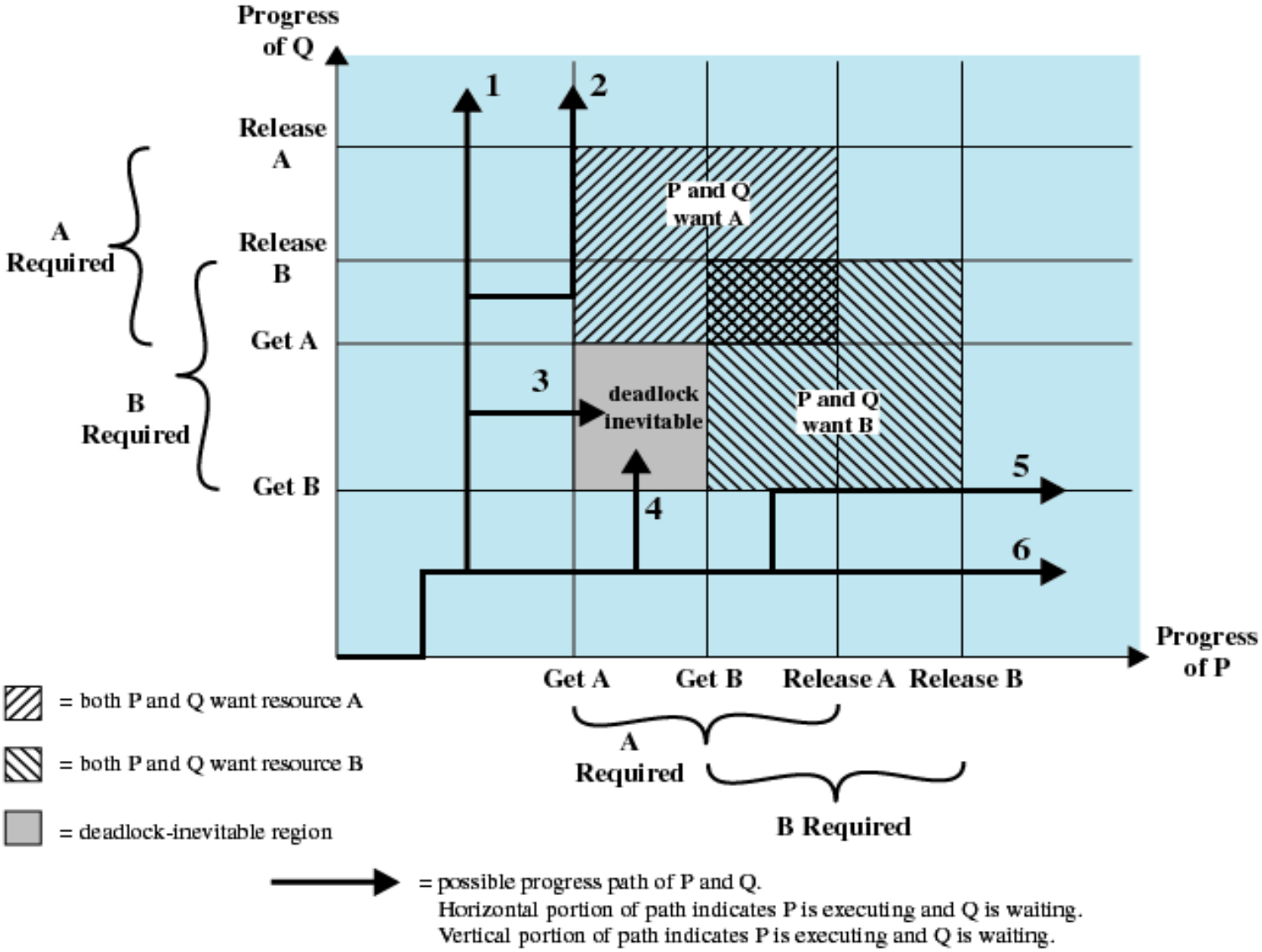
Nastanak uzajamnog blokiranja

Primer uzajamnog blokiranja

- Dva procesa P i Q zahtevaju isključiv pristup resursima A i B za određeni period vremena
- Dijagram zajedničkog izvršavanja (progresa)

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Diagram zajedničkog napretka



Alternativni primer

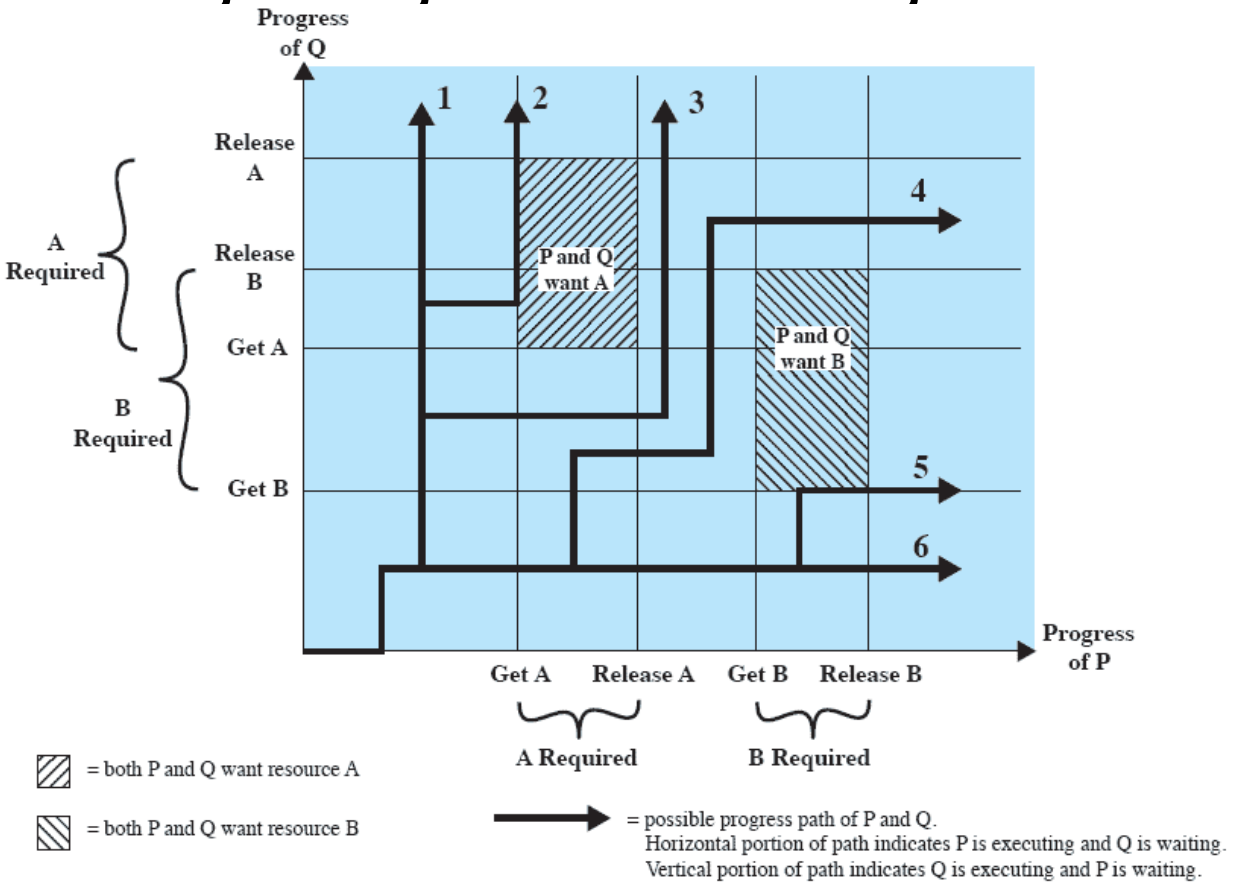
- Pretpostavimo da procesu P ne trebaju oba resursa istovremeno - ne nastaje uzajamno blokiranje

Process P

• • •
Get A
• • •
Release A
• • •
Get B
• • •
Release B
• • •

Process Q

• • •
Get B
• • •
Get A
• • •
Release B
• • •
Release A
• • •



Konkurentnost: uzajamno blokiranje i gladovanje

Kategorije resursa

- Pod resursom ćemo podrazumevati bilo koji objekat koji se može dodeliti procesima
- Postoje dve kategorije resursa:
  - Ponovo upotrebljivi resursi
    - Mogu istovremeno biti korišćeni od strane samo jednog procesa i nakon njegovo korišćenja upotrebljivi su za drugi proces
  - Potrošni resursi
    - To su resursi koji se mogu kreirati (proizvesti) i uništiti (potrošiti)

Ponovo upotrebljivi resursi

- Procesi zauzimaju instance resursa, koriste ih i kasnije ih oslobađaju kako bi ih koristili drugi procesi
- Primer ponovo upotrebljivih resursa:
  - Procesori
  - Glavna i sekundarna memorija
  - U/I uređaji
  - U/I kanali
  - Strukture podataka, poput datoteka, baza podataka, semafora
- Uzajamno blokiranje nastaje kad svaki proces drži jedan resurs i zahteva drugi

Primer uzajamnog blokiranja na ponovo upotrebljivim resursima (1)

- Dva procesa P i Q se nadmeću za ekskluzivan pristup datoteci D i uređaju trake T
- Uzajamno blokiranje će se javiti ako svaki proces drži jedan resurs i zahteva drugi, npr. ako se isprepletе izvršavanje ovih procesa na sledeći način p0 p1 q0 q1 p2 q2

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Primer uzajamnog blokiranja na ponovo upotrebljivim resursima (2)

- Dva procesa P1 i P2 se nadmeću za dodelu raspoloživog memorijskog prostora od 200 KB
  - Nastane sledeća sekvenca događaja

P1
. . .
Request 80 Kbytes;
. . .
Request 60 Kbytes;

P2
. . .
Request 70 Kbytes;
. . .
Request 80 Kbytes;

- Uzajamno blokiranje će se javiti ukoliko oba procesa u izvršavanju stignu do drugog zahteva

Potrošni resursi

- Kada proces potrošač zahteva takav resurs i dobije ga, resurs prestaje da postoji
- Ne postoji ograničenje u broju takvih resursa koje proces može napraviti
- Primeri potrošnih resursa:
  - Prekidi
  - Signali
  - Poruke
  - Informacije u U/I baferima
- Uzajamno blokiranje može nastati ako je operacija *Receive* poruke blokirajuća
- Retke kombinacije događaja mogu prouzrokovati uzajamno blokiranje

Primer uzajamnog blokiranja

- Dva procesa **P1** i **P2** komuniciraju razmenjujući poruke
- Uzajamno blokiranje se može desiti ako je funkcija **`Receive`** blokirajuća

P1

```
...
Receive(P2);
...
Send(P2,M1);
```

P2

```
...
Receive(P1);
...
Send(P1, M2);
```

Model sistema

- Sistem se sastoji od određenog broja resursa koji se dodeljuju procesima
- Resursi su razvrstani u više tipova
  - Npr. štampači, memorija,...
- Svaki tip ima više identičnih instanci
  - Ako proces zahteva neki tip resursa, OS mu može dodeliti bilo koju instancu tog tipa resursa
- Proces može zahtevati proizvoljan broj resursa da bi obavio zadatak
  - Broj zahtevanih resursa ne sme prelaziti broj raspoloživih resursa

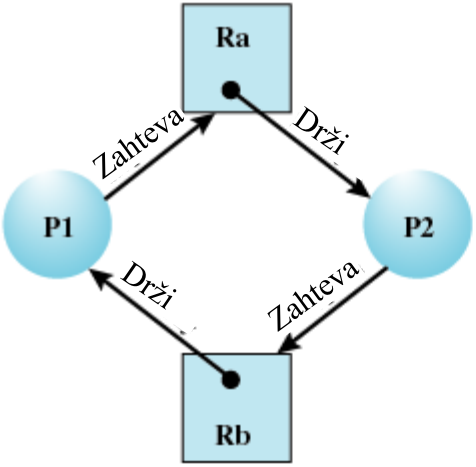
Graf dodele resursa

- Graf dodele resursa je orijentisani graf G=(V,E) kojim se modelira dodela resursa
- Skup čvorova V se deli na dva podskupa:
  - Čvorovi procesa - Skup procesa u sistemu P=(P1,P2,...,Pn)
  - Čvorovi resursa - Skup tipova resursa R=(R1,R2,...,Rm)
- Skup grana E se deli na dva podskupa:
  - Grane zahteva (zahteva, traži) - Skup grana zahteva –par (Pi,Rj)
  - Grane dodele (drži) - Skup grana dodele – to je uređeni par (Rj,Pi)
- Grana zahteva (Pi,Rj) modelira pojavu da proces Pi zahteva instancu resursa Rj
- Grana dodele (Rj,Pi) modelira pojavu da je instanca resursa tipa Rj dodeljena procesu Pi, tj. da proces Pi drži resurs tipa Rj

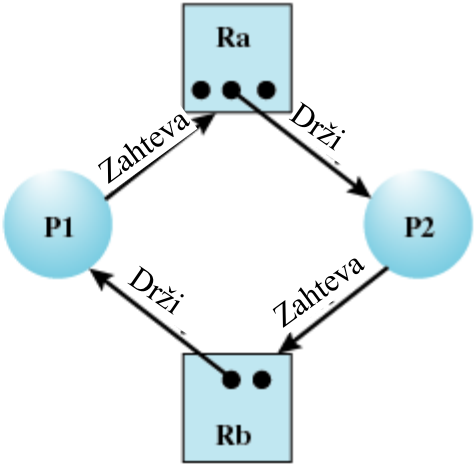
Graf dodele resursa (2)

(a) Proces zahteva resurs

(b) Proces drži resurs



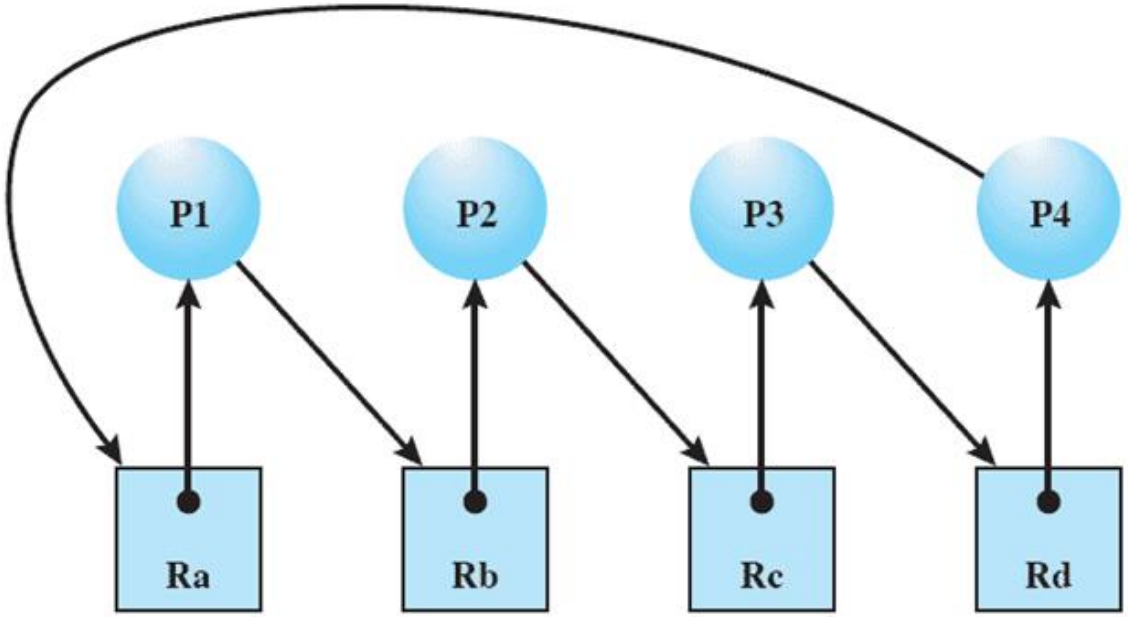
(c) Kružno čekanje



(d) Bez uzajamnog blokiranja

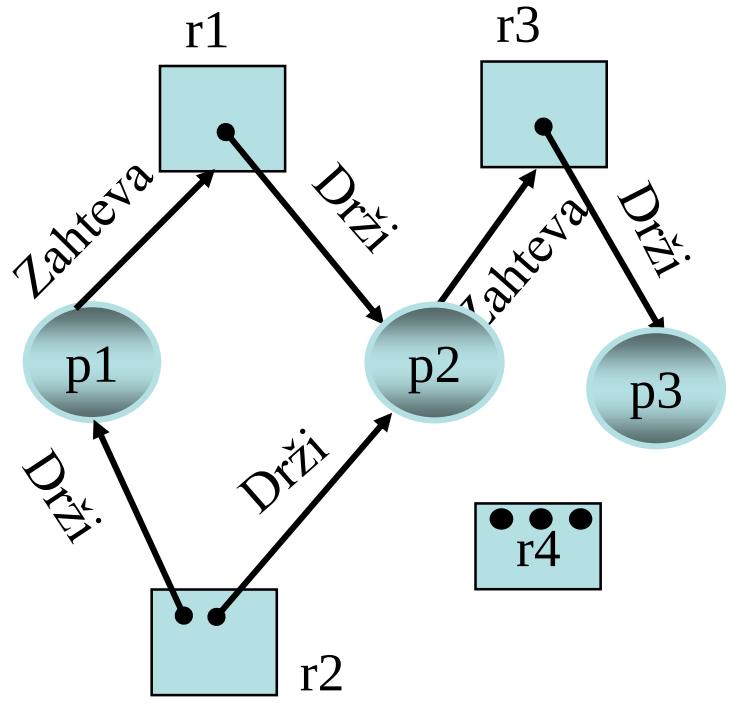
Graf dodele resursa (3)

- Graf dodele resursa za primer automobila na raskrsnici



Primer grafa dodele resursa

- Procesi P=(p1,p2,p3)
- Resursi R=(r1,r2,r3,r4)
- Instance resursa
  - r1 ima 1 instancu
  - r2 ima 2 instance
  - r3 ima 1 instancu
  - r4 ima 3 instance
- Stanje:
  - p1 drži 1 instancu resursa tipa r2 i zahteva 1 instancu resursa tipa r1
  - p2 drži r1 i r2 i zahteva r3
  - p3 drži r3



Nema uzajamnog blokiranja

Modeliranje uzajamnog blokiranja pomoću grafa dodele resursa

- Ako graf dodele resursa nema cikluse, tada nijedan proces u sistemu nije u uzajamnom blokiranju
- Ako graf dodele resursa ima cikluse, tada može postojati uzajamno blokiranje
- Ako svaki tip resursa ima po 1 instancu:
  - Ciklus u grafu dodele resursa je potreban i dovoljan uslov za postojanje uzajamnog blokiranja
- Ako svaki tip resursa ima više instanci:
  - Tada postojanje ciklusa u grafu dodele resursa je potreban, ali nije dovoljan uslov za pojavu uzajamnog blokiranja

Uslovi za uzajamno blokiranje

- Uslovi za moguće uzajamno blokiranje (potrebni)
  1. Uzajamno isključivanje (*Mutual exclusion*)
     - U datom trenutku samo jedan proces može koristiti resurs
  2. Drži i čekaj (*Hold-and-wait*)
     - Proces drži dodeljene resurse dok čeka dodeljivanje ostalih resursa
  3. Bez prekidanja (*No preemption*)
     - Nijedan resurs se ne može nasilno oduzeti procesu koji ga drži
- Uzajamno blokiranje nastaje kad su ispunjena prethodna tri uslova, i:
  4. Kružno čekanje (*Circular wait*)
     - Postoji zatvoreni „lanac" procesa, takav da svaki proces drži bar jedan resurs koji je potreban sledećem procesu u „lancu"
     - Ukoliko ima više instanci resursa ne mora da dodje do uzajamnog blokiranja

Potrebni i dovoljni uslovi za pojavu uzajamnog blokiranja

- Ako su prisutna samo prva 3 uslova postoji mogućnost za pojavu uzajamnog blokiranja –
  - Ovi uslovi se odnose na politiku dodele
- Da bi nastalo uzajamno blokiranje potrebno je da istovremeno budu prisutna sva 4 uslova
  - Kružno čekanje je okolnost koja zavisi od sekvence dodele i oslobađanja resursa od strane uključenih procesa

Metode za upravljanje uzajamnim blokiranjem

- Sprečavanje uzajamnog blokiranja
  - Onemogućiti nastanak nekog od 4 uslova za pojavu uzajamnog blokiranja
- Izbegavanje uzajamnog blokiranja
  - Donositi odgovarajuće dinamičke odluke o dodeli resursa u skladu sa trenutnim stanjem dodele resursa
- Otkrivanje uzajamnog blokiranja i oporavak
  - Udovoljiti zahtevu za resursom kada je to moguće, ali periodično proveravati da li je uzajamno blokiranje prisutno i tada oporaviti sistem od uzajamnog blokiranja

Sprečavanje uzajamnog blokiranja

- OS se projektuje tako da isključi mogućnost pojave uzajamnog blokiranja
- Dve klase metoda:
  - Indirektne metode sprečavanja uzajamnog blokiranja
    - Sprečavanje pojave jednog od 3 potrebna uslova
  - Direktne metode sprečavanja uzajamnog blokiranja
    - Sprečavanje pojave kružnog čekanja
- Ove metode vode neefikasnom korišćenju resursa i neefikasnom izvršenju procesa

Sprečavanje uslova "Uzajamno isključivanje"

- Ako nema resursa koji se dodeljuju ekskluzivno samo jednom procesu, nema uzajamnog blokiranja
- To nije moguće – uzajamno isključivanje se mora zadržati za nedeljive resurse
- Primer nedeljivog resursa: printer
- Primer deljivog resursa: *read-only* datoteka

Sprečavanje uslova "Drži i čekaj"

- Treba garantovati da kad proces zahteva neki resurs, ne sme držati nijedan drugi resurs
- Može se koristiti pristup kojim će se zahtevati od svakog procesa da pre izvršenja zatraži sve potrebne resurse
  - Proces se blokira dok mu se svi traženi resursi dodele
  - Procesi mogu dugo čekati udovoljenje svim zahtevima za resursima
  - Resursi dodeljeni procesu mogu ostati dugo neiskorišćeni iako bi mogao da ih koristi neki drugi proces
  - Proces bi morao da zna sve svoje zahteve za resursima
- Alternativa je da se koristi pristup po kome proces mora osloboditi sve resurse pre nego što zahteva dodatni resurs
- Oba pristupa imaju dva nedostatka:
  - Iskorišćenje resursa je malo kako je prikazano kod prvog pristupa
  - Gladovanje procesa – proces koji zahteva popularne resurse može beskonačno dugo čekati

Sprečavanje uslova "Bez prekidanja"

- **Pristup 1:** Ako proces drži neke resurse i zahteva drugi resurs koji mu se ne može odmah dodeliti (tj. proces mora da čeka), tada se svi resursi koje proces drži "otpuštaju" (oslobađaju) i proces će ih ponovo zahtevati sa dodatnim resursima
- **Pristup 2:** Kada proces P zahteva resurse koji su dodeljeni nekom drugom procesu Q koji takođe čeka na dodelu resursa oduzimaju se resursi od procesa Q koji čeka i dodeljujuju se procesu P. Problem ako su procesi istog prioriteta.
- Kad god proces mora da oslobodi resurs koji je već koristio, stanje tog resursa se mora pamtiti za kasniji nastavak procesa
- Stoga, ovaj protokol je praktičan samo za resurse čije se stanje može lako pamtiti i restaurirati, kao što je procesor

Sprečavanje uslova "Kružno čekanje"

- Strategija za sprečavanje kružnog čekanja
  - Definisati **striktno linearno uređenje O() za sve tipove resursa**
  - Svakom tipu resursa se dodeljuje jedinstven ceo broj tako da se resursi mogu urediti. Npr:
    - R1: jedinica trake O(R1)=2
    - R2: jedinica diska O(R2)=4
    - R3: štampači O(R3)=7
  - **Proces može zahtevati resurse samo u rastućem redosledu njihovog uređenja**
    - Proces inicijalno traži određeni broj instanci nekog tipa resursa, Ri
    - Ako proces zahteva više instanci istog tipa resursa mora ih tražiti jednim zahtevom
    - Ako je procesu dodeljen resurs Ri, može zahtevati instance resursa Rj ako i samo ako je O(Rj)>O(Ri)
- Sprečavanje kružnog čekanja može biti neefikasno, usporava procese i nepotrebno odbija pristup resursima

Izbegavanje uzajamnog blokiranja

- Pažljivom dodelom resursa može se izbeći uzajamno blokiranje
  - Kada proces traži resurse, sistem proverava da li je bezbedno ili nebezbedno dodeliti tražene resurse
  - Ako je bezbedno, sistem može doneti odluku o dodeli
  - Ako nije bezbedno, sistem odbija dodelu resursa
- Pitanje je da li postoji algoritam koji omogućava sistemu da uvek napravi pravi izbor i da se na taj način uvek može izbeći uzajamno blokiranje
  - Potrebne su unapred informacije o tome kako će procesi zahtevati resurse
- Ova metoda se bazira na konceptu bezbednih stanja (*safe states*)

Stanje sistema

- **Stanje sistema** oslikava trenutnu dodelu resursa procesima
  - R=(R1,R2,…,Rm)- (*Resource*) ukupan broj svakog resursa u sistemu
  - V=(V1,V2,…,Vm)- (*Available*) trenutno raspoloživi resursi u sistemu
  - C=(Cij), i=1,n, j=1,m – (*Claim*) max zahtevi procesa za resursima
  - A=(Aij), i=1,n, j=1,m – (*Allocation*) trenutno dodeljeni resursi procesima (vrste predstavljaju procese, a kolone resurse)
- Važi sledeće:

1. Svi resursi su ili raspoloživi ili dodeljeni
   $Rj = Vj + \sum Ai \mid i=1,n$ za svako j=1,m
2. Nijedan proces ne može zahtevati više od ukupne količine resursa u sistemu
   $Cij \leq Rj$, za svako i, j
3. Nijednom procesu nije dodeljeno više resursa od njegovih maksimalnih zahteva
   $Aij \leq Cij$, za svako i, j

Bezbedno stanje

- Stanje je **bezbedno** ako sistem može dodeliti resurse svakom procesu (do max broja resursa) u nekom redosledu i da pri tom izbegne uzajamno blokiranje
- Sistem je u bezbednom stanju samo ako postoji **bezbedna sekvenca procesa**
- Sekvenca procesa <Po,P1,…,Pn> je **bezbedna sekvenca** za tekuće stanje dodele resursa ako za svaki Pi, resursi koje proces Pi može još tražiti mogu biti zadovoljeni trenutno raspoloživim resursima i resursima koje drže svi procesi Pj, j<i, tj. uslov koji se mora zadovoljiti za proces Pi je

  Cij – Aij ≤ Vj , za svako j
- Za svaki proces Pi i za svaki resurs Rj mora biti ispunjen uslov:

  Broj zahtevanih – Broj dodeljenih ≤ Broj raspoloživih resursa

Utvrđivanje bezbednog stanja

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Početno stanje

Procesu P2 su upravo dodeljeni 1 instanca R1 i 1 instanca R3

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – maksimalni zahtevi procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Utvrđivanje bezbednog stanja

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

Proces P2 se izvršava do kraja

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – zahtev procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Utvrđivanje bezbenog stanja

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

Proces P1 se izvršava do kraja

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – zahtev procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Utvrđivanje bezbednog stanja

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

Proces P3 se izvršava do kraja, pa zatim i P4

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – zahtev procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Utvrđivanje nebezbednog stanja

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

Početno stanje

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – zahtev procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Utvrđivanje nebezbednog stanja

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Proces P1 zahteva po jednu instancu resursa R1 i R3
i sistem je u nebezbednom stanju

R=(R1,R2,…,Rm) – ukupna količina svakog resursa u sistemu
V=(V1,V2,…,Vm) - trenutno raspoloživi resursi u sistemu
C=(Cij), za i=1,n, j=1,m – zahtev procesa za resursima
A=(Aij), za i=1,n, j=1,m – trenutno dodeljeni resursi procesima

Bankarov algoritam

- Predložio ga je Dijkstra (1965)
- Ime je dobio po tome što se princip koristi u bankama pri dodeli raspoloživog novca klijentima kako se ne bi došlo u situaciju da više ne mogu biti zadovoljene potrebe klijenata
- Proverava da li dodela resursa vodi u bezbedno stanje
  - Ako DA - vrši se dodela resursa
  - Ako NE – blokirati proces sve dok nije bezbedno dodeliti resurs (uslužiti zahtev)

Bankarov algoritam (pseudo kod)

- Potrebne strukture podataka:
  - R – vektor ukupnog broja svakog resursa u sistemu
  - V – vektor raspoloživih resursa
  - A – matrica resursa trenutno dodeljenih procesima
  - C – matrica maksimalnih zahteva procesa za resursima
    - C[i,*] vektor maksimalnih zahteva procesa Pi
    - C[i,j]=k znači da proces Pi traži maksimalno k instanci resursa Rj
  - Q – vektor dodatnih zahteva procesa Pi za resursima– važi Q= Ci -Ai

Bankarov algoritam (pseudo kod)

1. **Ako** je *A[i,*] + Q[*] > C[i ,*],*
   tada greška "process traži više od maksimuma"
   inače preći na korak 2
2. **Ako** je *Q[*] > V [*]*
   tada traženi resursi nisu raspoloživi i *Pi* mora da se blokira
   Inače preći na korak 3
3. Simulira se dodela zahtevanih resursa *Q[i ,*]* procesu *Pi* i menja stanje dodele resursa:
   V [i,]= V [i,*] – Q[*] // raspoloživi resursi*
   A[i,] = A[i,*] + Q[*] // dodeljeni resursi*
4. Proverava se da li je novo stanje bezbedno (koristi se algoritam ispitivanja bezbednosti sa sledećeg slajda)
5. **Ako** jeste,
   tada se procesu *Pi* stvarno dodeljuju traženi resursi;
   inače proces *Pi* se blokira i čeka da mu se zahtevani resursi *Q[*]* dodele, a staro stanje se restaurira;

Algoritam ispitivanja bezbednosti (pseudo kod)

- **Algoritam ispitivanja bezbednosti** proverava da li je sistem u bezbednom stanju

1. Inicijalizacija
   TrenutnoRaspolozivi = V
   Rest = {svi procesi}
2. Ponavljati korake 3 i 4
3. Naći *Pi u Rest* za koji važi
   C[i ,] – A[i,*] ≤ Trenutno Raspolozivi*
4. **Ako** takav *Pi* postoji **tada**
   Trenutno Raspolozivi = Trenutno Raspolozivi + A[i,]*
   Rest = Rest - {Pi}
   inače preći na korak 5
5. **Ako** je *Rest = {}* **tada** je sistem u bezbednom stanju
   inače sistem je u nebezbednom stanju

Logika izbegavanja uzajamnog blokiranja – Bankarov algoritam

```
struct state {
      int resource[m];
      int available[m];
      int claim[n][m];
      int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                           /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Logika izbegavanja uzajamnog blokiranja - Bankarov algoritam (2)

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Otkrivanje i oporavak

- Sistem ne pokušava da spreči pojavljivanje uzajamnog blokiranja
  - Strategije sprečavanja i izbegavanja uzajamnog blokiranja ograničavaju pristup resursima i nameću ograničenja procesima
- Umesto toga, sistem dozvoljava da se uzajamno blokiranje pojavi, a onda uključuje mehanizme da otkrije uzajamno blokiranje i da oporavi sistem, tj. da raskine uzajamno blokiranje
- Provera uzajamnog blokiranja može se vršiti:
  1. Pri svakom zahtevu za resursima, ili
  2. Povremeno – zavisno od toga kolika je verovatnoća pojave uzajamnog blokiranja
- Prva varijanta je bolja u tome što se ranije otkriva uzajamno blokiranje i algoritam je jednostavan, ali česte provere troše značajno procesorsko vreme

Algoritam otkrivanja uzajamnog blokiranja

Resursi sistema - R

$(R1,R2,R3,\ldots,Rm)$

Raspoloživi resursi - V

$(V1,V2,V3,\ldots,Vm)$

Matrica dodele

$$\begin{vmatrix} A11 & A12 & A13 & \ldots & A1m \\ A21 & A22 & A23 & \ldots & A2m \\ . & . & . & . & \\ . & . & . & . & \\ . & . & . & . & \\ An1 & An2 & An3 & \ldots & Anm \end{vmatrix}$$

Matrica dodatnih zahteva

$$\begin{vmatrix} Q11 & Q12 & Q13 & \ldots & Q1m \\ Q21 & Q22 & Q23 & \ldots & Q2m \\ . & . & . & . & \\ . & . & . & . & \\ . & . & . & . & \\ Qn1 & Qn2 & Qn3 & \ldots & Qnm \end{vmatrix}$$

R– vektor tipova resursa koji postoje u sistemu; R[j] broj instanci tipa resursa Rj

V – vektor raspoloživih resursa; v[j] broj raspoloživih instanci tipa resursa rj

A – matrica tekuće dodele resursa; vrste procesi, kolone tipovi resursa

Q – matrica dodatnih zahteva za resursima Q = C - A

Algoritam otkrivanja uzajamnog blokiranja

- Algoritam označava sve procese koji nisu uzajamno blokirani
  - Inicijalno svi procesi su neoznačeni

1. Označiti svaki proces čija vrsta u matrici dodele A sadrži sve nule
2. Inicijalizovati privremeni vektor W koji je jednak vektoru raspoloživih resursa V
3. U matrici zahteva Q naći neoznačeni proces Pi koji zadovoljava uslov Qi ≤ W , k=1,m . Ako takav proces ne postoji, tada završiti algoritam
4. Ako takav proces postoji, označiti proces Pi i vektoru W dodati i-tu vrstu matrice A (Wk = Wk+Aik , k=1,m) , i vratiti se na korak 3

- Uzajamno blokiranje postoji ako i samo ako postoje neoznačeni procesi po završetku algoritma

Algoritam otkrivanja uzajamnog blokiranja

- U koraku 3 traži se proces koji može da se izvršava – njegovim zahtevima za resursima sistem može odgovoriti.
- Takav proces se izvršava i nakon toga oslobađa resurse koje drži.
- Resursi se vraćaju u raspoložive, dodaju se u W (korak 4) - Proces se označava kao obrađen.
- Uzajamno blokiranje postoji ako i samo ako postoje neoznačeni procesi po završetku algoritma

Otkrivanje uzajamnog blokiranja

- Primer otkrivanja uzajamnog blokiranja

	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
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
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

1. Označava se P4 jer nema alociranih resursa
2. Postavlja se W=(0 0 0 0 1)
3. Zahtev Procesa P3 je ≤ od W, označava se P3 i postavlja W =W+(0 0 0 1 0) = (0 0 0 1 1)
4. Nema neoznačenih procesa čiji su zahtevi Qi ≤ W pa se algoritam završava

Pošto su P1 i P2 neoznačeni, uzajamno su blokirani

Oporavak iz uzajamnog blokiranja

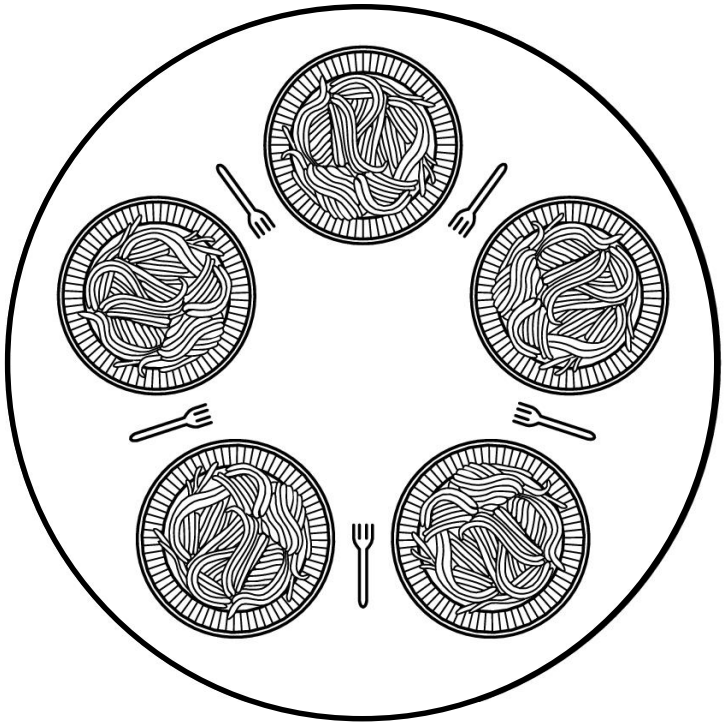
1. Prekinuti sve procese koji su uzajamno blokirani
2. Vratiti sve uzajamno blokirane procese do neke prethodno definisane kontrolne tačke i tada restartovati sve procese
   - Ovo zahteva postojanje *rollback* i *restart* mehanizama ugrađenih u OS
3. Sukcesivno prekidati procese koji su uzajamno blokirani sve dok postoji uzajamno blokiranje
   - Redosled selektovanja procesa koji će biti prekinuti se zasniva na kriterijumu minimalnih troškova. Bira se proces koji ima:
     - najmanje korišćeno procesorsko vreme, najmanju količinu generisanog izlaza, najduže procenjeno preostalo vreme, najmanje alociranih resursa, najmanji prioritet, itd.
   - Nakon svakog prekidanja treba proveriti da li još uvek postoji uzajamno blokiranje algoritmom otkrivanja uzajamnog blokiranja
4. Selektivno oduzeti resurse procesima sve dok postoji uzajamno blokiranje (korišćenjem algoritma detekcije posle svakog oduzimanja).
   - Proces kome su oduzeti resursi se mora vratiti do tačke izvršenja pre dodele resursa

Strategije rešavanja uzajamnog blokiranja – rezime

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates on-line handling	•Inherent preemption losses

Večera filozofa *Dining Philosophers* (1)

- Klasičan sinhronizacioni problem
- Pet filozofa jede (*eat*) i razmišlja (*think*)
- Da bi jeli potrebne su im dve viljuške (*forks*) - za stolom je samo 5 viljuški
- Algoritam mora zadovoljiti uzajamno isključivanje (dva filozofa ne mogu koristiti istovremeno istu viljušku) pri tom izbegavajući uzajamno blokiranje i gladovanje



Rešenje problema večere filozofa

- Prvo rešenje korišćenjem semafora
  - Ukoliko svi procesi izvrše `wait(fork[i])` dolazi do uzajamnog blokiranja

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true)
     {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
     }
```

Rešenje problema večere filozofa

- Drugo rešenje uvodi brojački semafor koji dozvoljava da samo 4 filiozofa mogu da uđu u trpezariju

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Rešenje korišćenjem monitora

- Koristi se vektor sa 5 uslovnih promenljivih **ForkReady**
  - Koriste se da bi omogućili filozofima da čekaju na raspoloživu viljušku
- Koristiti se vektor *boolean* koji registruje status viljuške
  - ***true*** viljuška slobodna
  - ***false*** viljuška zauzeta

Rešenje korišćenjem monitora

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

Rešenje korišćenjem monitora (2)

```
void philosopher[k=0 to 4]              /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);      /* client releases forks via the monitor */
   }
}
```

UNIX mehanizmi konkurentosti

- UNIX obezbeđuje različite mehanizme za sinhronizaciju i komunikaciju procesa (pogledati detalje u praktikumu):
  - *Semaphore*
  - *Signals*
  - *Pipe*
  - *Message*
  - *Shared memory*

Linux Kernel mehanizmi konkurentosti

- Linux uključuje sve mehanizme koji postoje u UNIX-u plus:
  - *Atomic operations*
  - *Spinlocks (Basic, Reader-Writer)*
  - *Semaphores*
    - Semafori na korisničkom niovu odgovaraju UNIX SVR4 semaforima
    - Semafori na nivou kernela samo za kernel procese: binarni, brojački i *reader-writer* semafori
  - *Barriers*
    - Da bi se obezbedio redosled instrukcija koje pristupaju memoriji, definišu se memorijske barijere

Solaris Thread sinhronizacione primitive

- Solaris dopunjuje mehanizme konkurentnosti u UNIX SVR4
  - *Mutual exclusion (mutex) locks*
  - *Semaphores*
  - *Multiple readers, single writer (readers/writer) locks*
  - *Condition variables*
- Ovi mehanizmi su dostupni nitima kernela i nitima korisničkih procesa

Windows mehanizmi konkurentosti

- Windows obezbeđuje sinhronizaciju između niti kao deo svoje objektne arhitekture.
  - Pogledati detalje u praktikumu (2. deo)
- Sinhronizacini objekti koriste *Wait* funkciju
  - *WaitForSingleObject* blokira nit ukoliko kriterijum na kome se čeka nije ispunjen
- Sinhronizacioni objekti:
  - *Event, Mutex, Semaphore, Waitable timer, Process, Thread, ...*
- Objekti kritične sekcije - *critical sections*
- *Slim reader-writer locks, Condition variables.*

Domaći zadatak

- Poglavlje **6 Konkrurentnost: uzajamno blokiranje i gladovanje**
  - 6.13 Ključni pojmovi, kontrolna pitanja i problemi
- Animacija
  - Deadlocks (Resource Allocation Graph)

https://apps.uttyler.edu/Rainwater/COSC3355/Animations/deadlock.htm