Operativni sistemi 2

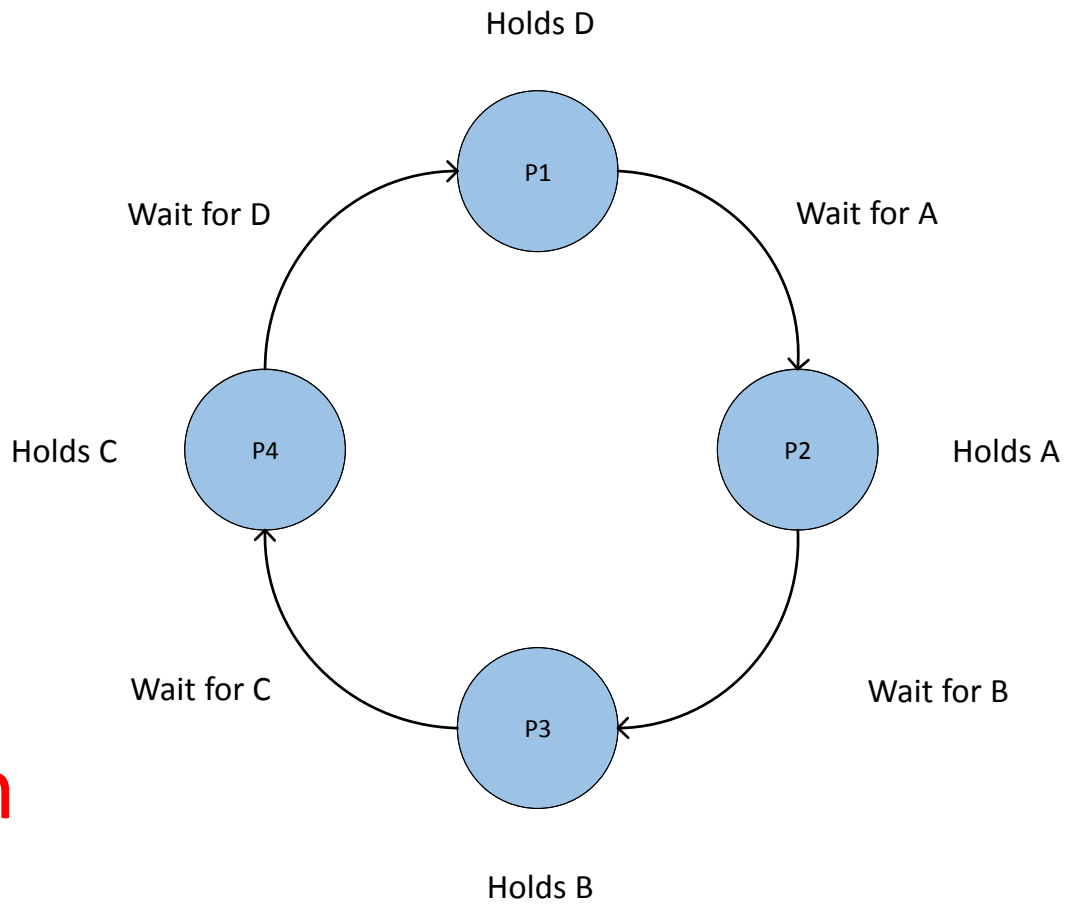
Vežbe 3
Upravljanje resursima

Lista predmeta:
IR smer - 13e113os2@lists.etf.rs
SI Smer - 13s113os2@lists.etf.rs

Deadlock

No preemption

Circular wait



Mutual exclusion

Hold and wait

Zadatak – Januar 2007.

Tri uporedna procesa, A, B i C zauzimaju i oslobađaju dva nedeljiva resursa X i Y po sledećem redosledu:

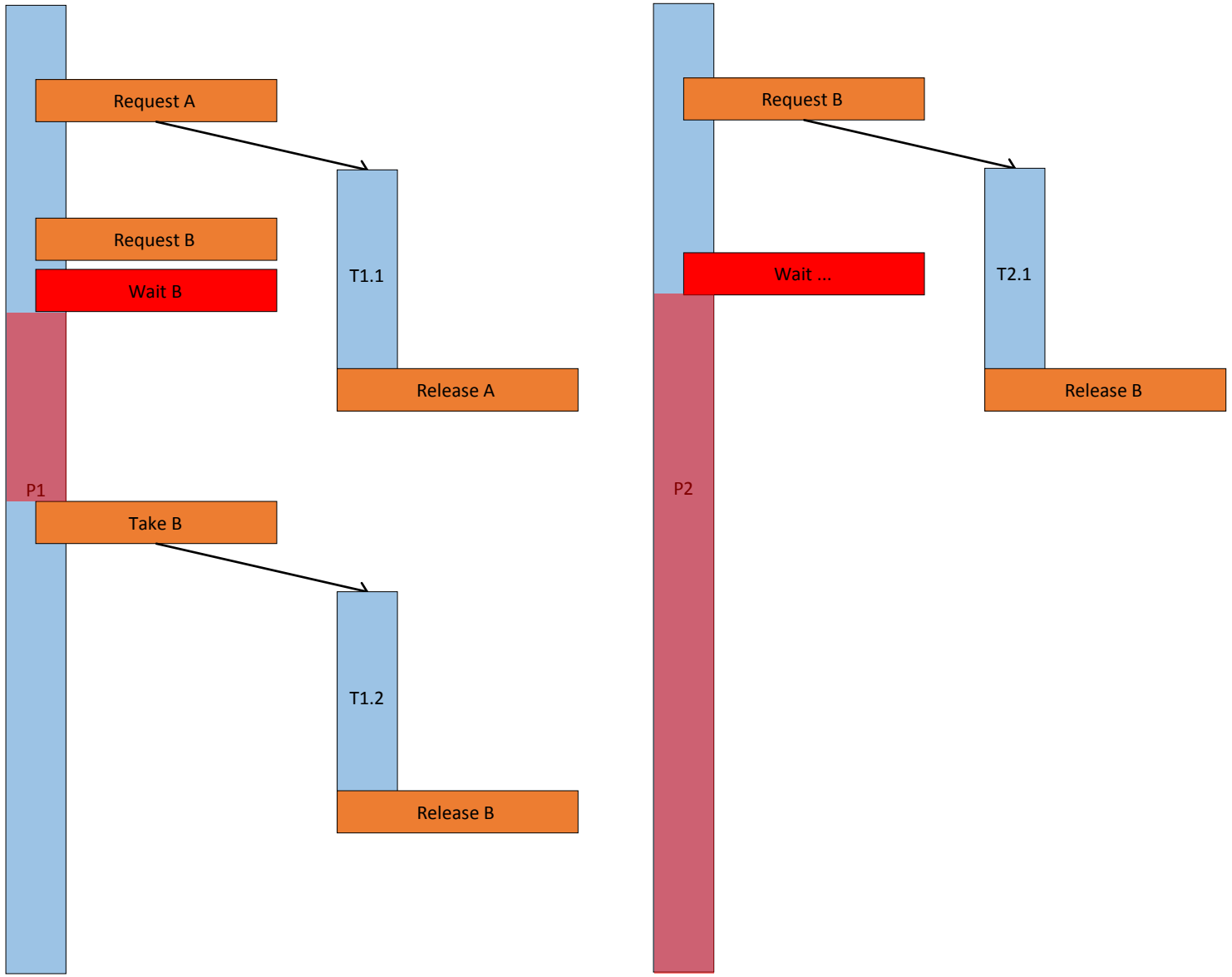
A: request(X), release(X), request(Y), release(Y)
B: request(Y), release(Y), request(X), release(X)
C: request(X), request(Y), release(Y), release(X)

Da li ova tri procesa mogu da uđu u stanje mrtve blokade (*deadlock*)? Ako mogu, dati scenario po kome dolaze u ovo stanje i nacrtati graf zauzeća resursa u tom stanju. Ako ne mogu, precizno objasniti (dokazati) zašto ne mogu.

Ne mogu, jer nije ispunjen neophodan uslov „cirkularno držanje i čekanje“ (*circular wait*). Naime, jedino proces C može doći u stanje „držanja i čekanja“ (*hold and wait*), koje je neophodno za nastanak mrtve blokade, kada zauzme resurs X, a eventualno čeka na resurs Y. Da bi postojala mrtva blokada, taj resurs Y mora da drži neki drugi proces (A ili B) koji u isto vreme čeka na neki drugi resurs. Međutim, ni proces A ni proces B ne traže ni jedan drugi resurs u periodu u kome eventualno drže resurs Y, već mogu da nastave svoje izvršavanje i oslobode resurs Y.

Zadatak – Oktobar 2007.

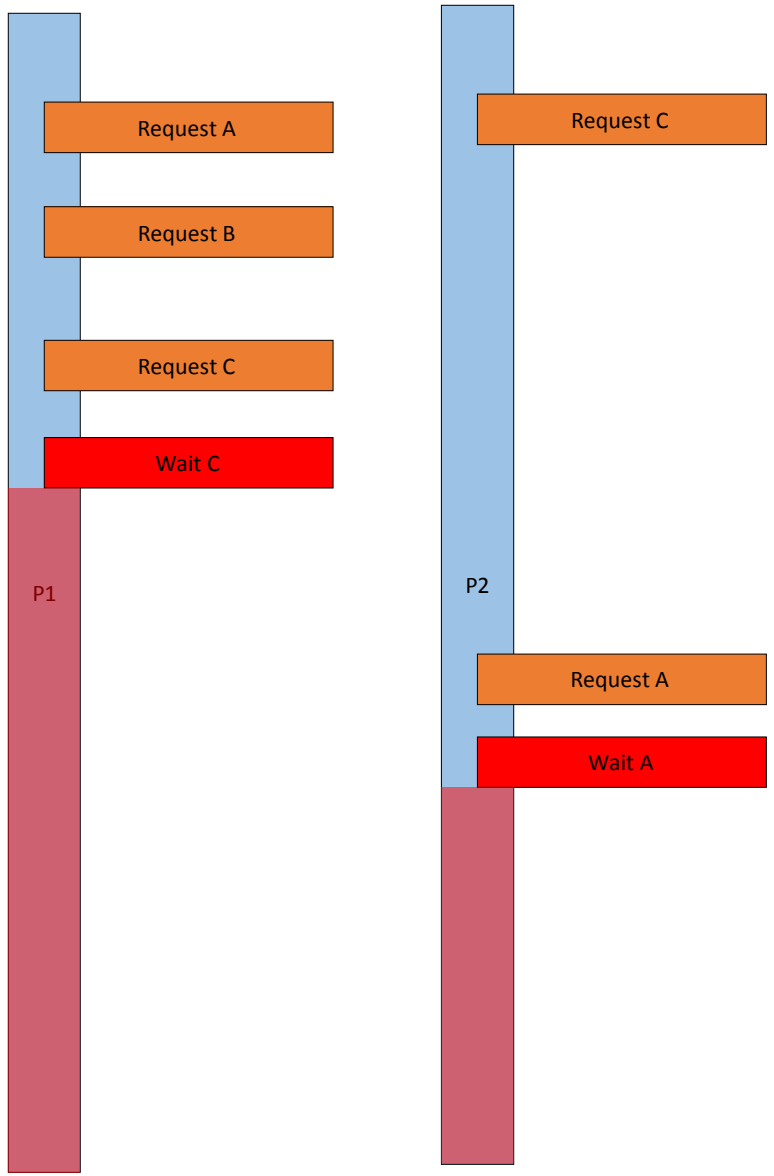
U nekom sistemu primenjuje se sledeći pristup sprečavanju mrtve blokade (*deadlock*). Proces zauzima željeni resurs eksplicitno, odgovarajućim sistemskim pozivom u kome se zadaje i operacija nad tim resursom. Operacija se potom radi asinhrono, u smislu da proces nastavlja svoje izvršavanje uporedo i nezavisno od toka operacije nad resursom. Kada se operacija završi, resurs se oslobađa implicitno (sam sistem ga oslobađa, a ne proces nekom eksplicitnom operacijom). Pre nego što se resurs oslobodi, drugi procesi ga ne mogu zauzeti, već bivaju suspendovani ako ga zatraže. Ovo je ujedno i jedini način zauzimanja i korišćenja resursa od strane procesa. Resursi su interno numerisani prirodnim brojevima, tako da je svakom resursu pridružen jedinstveni prirodan broj. Precizno i u potpunosti objasniti postupak koji sistem treba da sprovodi u sistemskom pozivu zauzimanja resursa da bi sprečio mrtvu blokadu. Posebno objasniti kada i kako sistem treba da suspenduje i deblokira suspendovani proces.

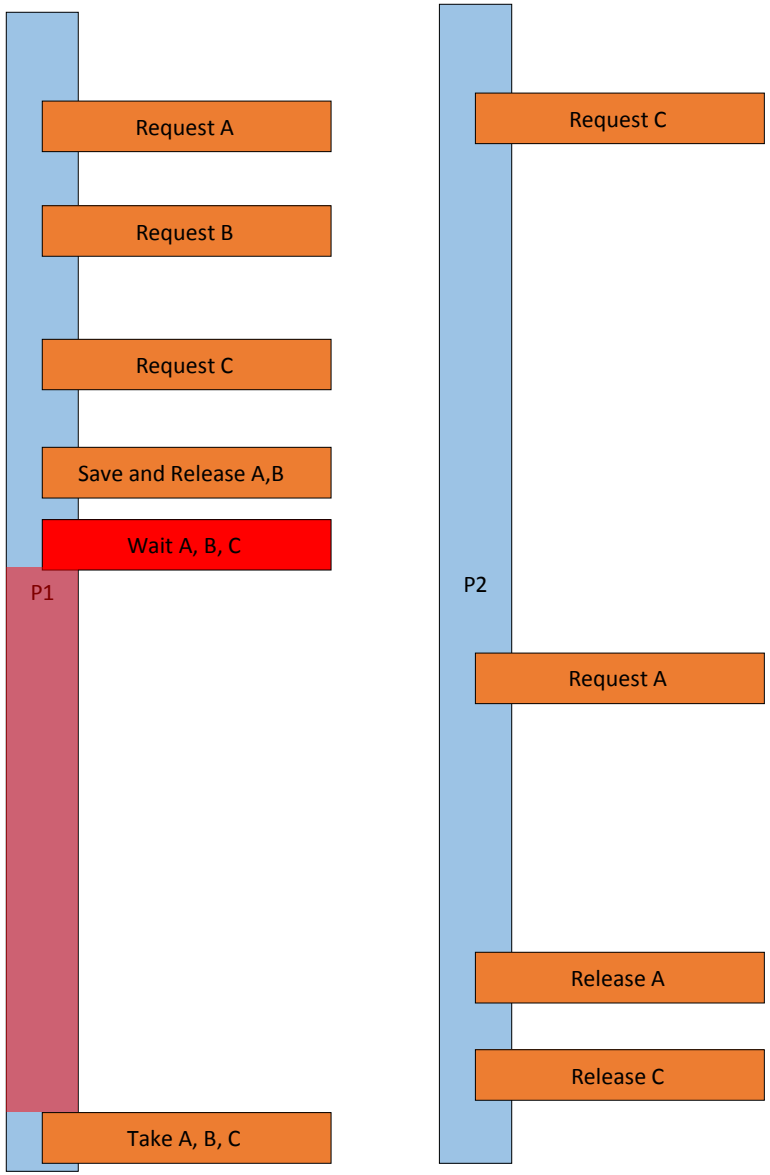

Request A
Request B
Wait B
T1.1
Release A
P1
Take B
T1.2
Release B
Request B
Wait ...
T2.1
Release B
P2

U opisanom sistemu nije moguće mrtvo blokiranje jer nije zadovoljen uslov „drži i čeka“. Naime, iz činjenice da će resursi po završetku operacije biti implicitno oslobođeni sledi da proces ne može (beskonačno) da čeka na jedan resurs, a da pri tome sve vreme drži zauzet drugi resurs, jer će ovaj drugi u konačnom vremenu (pod pretpostavkom ispravnog funkcionisanja sistema) biti oslobođen. Drugim rečima, kada je neki proces blokiran zato što je resurs koga je zatražio zauzet, ostali resursi koje je dobio ranije mogu biti oslobođeni (čim se tražena operacija završi, dakle u konačnom vremenu). Zbog rečenog, pri rezervisanju resursa nije potrebno primenjivati nikakav dodatni algoritam za sprečavanje mrtve blokade. Dakle, pri zahtevu za resursom dovoljno je proveriti da li je resurs već zauzet. Ako jeste, pozivajući proces blokirati. U nastavku, označiti ga kao zauzetog i pokrenuti traženu operaciju. Kada se operacija na nekom resursu završi, ako ne postoji nijedan proces koji na taj resurs čeka, označiti taj resurs kao slobodan. Ako postoji neki proces koji čeka na taj resurs, deblokirati ga.

Zadatak – Jun 2008.

U nekom sistemu upravljanja deljenim resursima resursi su takvi da je moguće sačuvati kontekst korišćenja (stanje) resursa od strane nekog procesa koji ga je zauzeo, osloboditi taj resurs, proces suspendovati, a kasnije restaurirati taj kontekst (stanje) i nastaviti izvršavanje tog procesa uz zauzimanje i korišćenje tog resursa. Predložiti i precizno opisati postupak kojim se sigurno sprečava mrtva blokada (*deadlock*) u ovom sistemu i dokazati da je mrtva blokada nemoguća. (Obratiti pažnju da se ne traži primena detekcije i rešavanja mrtve blokade, već njeno sprečavanje.) Ukazati na eventualne druge probleme predloženog rešenja.


Request A
Request B
Request C
Wait C
P1
Request C
P2
Request A
Wait A


Request A
Request B
Request C
Save and Release A,B
Wait A, B, C
P1
Take A, B, C
Request C
P2
Request A
Release A
Release C

U trenutku kada neki proces zatraži resurs koji je zauzet, treba sačuvati kontekst korišćenja (stanje) svih resursa koje je on već zauzeo i koristi, osloboditi te resurse i suspendovati proces, tako da on čeka da se oslobodi resurs koji je tražio, ali i svi oni koje je već bio zauzeo i koji su mu preoteti. Kada se oslobode svi ti resursi, treba restaurirati kontekst korišćenja resursa koje je proces već koristio, zauzeti onaj koji traži, i deblokirati taj proces.

Ovakav protokol sigurno sprečava mrtvu blokadu jer nije ispunjen uslov „držanja i čekanja“ (engl. *hold and wait*) koji je neophodan za nastajanje mrtve blokade: proces neće držati zauzet ni jedan resurs ako je došao u situaciju da čeka na novi resurs koga je neko drugi zauzeo.

Problem ovog pristupa je potencijalno izgladnjivanje: proces kome su ovako preoteti resursi može beskonačno čekati jer mu drugi procesi zauzimaju po neki od resursa na koje on čeka da se oslobode.

Zadatak – Oktobar 2010.

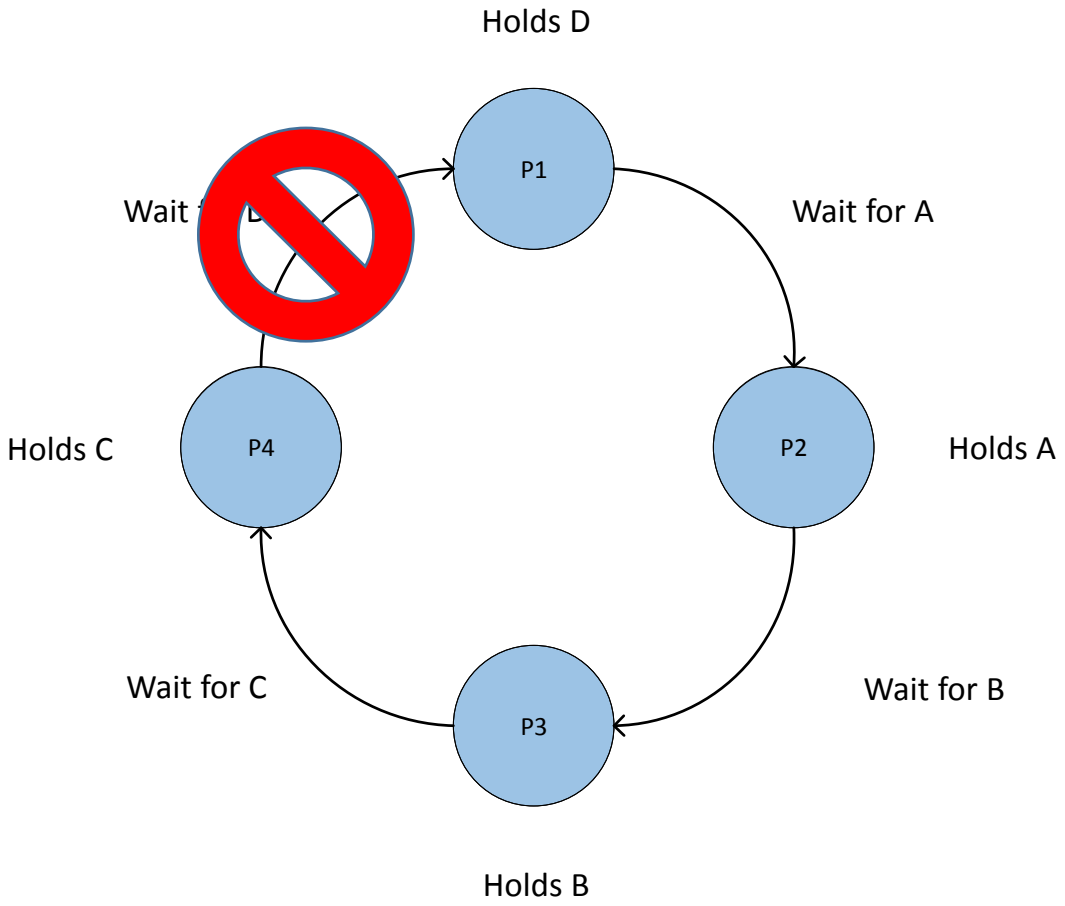
U nekom specijalizovanom sistemu proces se može „poništiti“ (*roll back*) – ugasiti uz poništavanje svih njegovih efekata, i potom pokrenuti ispočetka. U ovom sistemu primenjuje se sledeći algoritam sprečavanja mrtve blokade (*deadlock prevention*). Svakom procesu se, prilikom kreiranja, dodeljuje jedinstveni identifikator tako da se identifikatori dodeljuju procesima po rastućem redosledu vremena kreiranja: kasnije kreirani proces ima veći ID. Kada proces P_i sa identifikatorom i zatraži resurs koga drži proces P_j sa identifikatorom j, onda se postupa na sledeći način:

- ako je $i < j$, onda se P_i blokira i čeka da resurs bude oslobođen;
- ako je $i > j$, onda se P_i poništava i pokreće ponovo.

a)(5) Dokazati da se ovim algoritmom sprečava mrtva blokada.

b)(5) Koji ID treba dodeliti poništenom procesu P_i kada se on ponovo pokrene, da bi ovaj algoritam sprečio izgladnjivanje (*starvation*)? Obrazložiti.

- ako je $i < j$, onda se P_i blokira i čeka da resurs bude oslobođen;
- ako je $i > j$, onda se P_i poništava i pokreće ponovo.



a)(5) Dokaz kontradikcijom. Pretpostavimo da može nastati mrtva blokada, što znači da postoji zatvoren krug procesa P_{i1}, P_{i2}, ..., P_{in} (n>=1) koji su međusobno blokirani. Prema uslovima algoritma, odatle bi sledilo da je: $i_1<i_2<$ ... $<i_n<i_1$, što ne može biti, pa mrtva blokada ne može nastati.

b)(5) Prema uslovima algoritma, ako mlađi proces zatraži resurs koga drži neki stariji proces, mlađi proces se poništava i pokreće ponovo. Kada se poništeni proces ponovo pokrene, ako bi mu se dodelio novi ID koji odgovara vremenu njegovom ponovnog pokretanja, on bi bio još mlađi u sistemu, pa bi trpeo još više poništavanja, što može dovesti do njegovog izgladnjivanja. Zato mu treba dodeliti isti ID koji je imao pri prvom pokretanju. Ako bi on bio dalje ponovo poništavan, vremenom bi taj proces postajao sve stariji i konačno postao najstariji, kada više neće doživeti poništavanje, odnosno neće trpeti izgladnjivanje.

Zadatak – Januar 2016.

U nekom sistemu postoje dva procesa, *P*1 i *P*2, koji koriste resurse *R*1, *R*2 i *R*3 na način dat dole. Pretpostavlja se da odmah nakon operacije zauzeća određenih resursa dati proces vrši operacije sa tim resursom, odnosno da je proces ovako konstruisan da bi resurse držao zauzete najmanje moguće (alocira ih najkasnije što može, pre same operacije). Prestrukturirati ove procese tako da se spreči njihova mrtva blokada (*deadlock*) ukidanjem uslova cirkularnog čekanja (*circular wait*). Prestrukturiranje podrazumeva da proces može da zauzme resurs i ranije nego što je to neophodno za operaciju sa tim resursom, ali ne pre nego što je neophodno za sprečavanje mrtve blokade.

*P*1:	*P*2:
request(*R*1);	*request*(*R*2);
request(*R*2, *R*3);	*request*(*R*1,*R3*);
release(*R*1,*R*2,*R*3);	*release*(*R*1,*R*2,*R*3);

Trivijalno rešenje je da oba procesa zatraže odmah sve resurse. Takvo rešenje je neoptimalno. Treba zauzeti samo one resurse koji su potrebni i dovoljni da se izbgne mrtva blokada.

Mrtva blokada nastaje kada proces P1 zauzme resurs R1 a proces P2 zauzme resurs R2. Nakon toga svaki proces će tražiti resurs koji drugi poseduje i blokiraće se.

Jedan način da se izbegne kružno čekanje je da jedan od procesa traži ranije onaj resurs na koji kružno čeka. Recimo P1 će kružno čekati na resurs R2.

- *P*1: *P*2:

  request(*R*1, R2); *request*(*R*2);
  request(*R*3); *request*(*R*1,*R3*);
  release(*R*1,*R*2,*R*3); *release*(*R*1,*R*2,*R*3);

Zadatak – Februar 2007.

Dat je interfejs monitora koji obezbeđuje neophodnu sinhronizaciju po protokolu *multiple readers – single writer*:

```
class ReadersWriters {
public:
  void startRead();
  void stopRead();
  void startWrite();
  void stopWrite();
//...
};
```

a)(5) Korišćenjem ovog monitora napisati kod tela procesa koji najpre čita vrednost celobrojne deljene promenljive `x`, a onda tu vrednost uvećanu za 1 upisuje u deljenu promenljivu `y`. Pristup ovim deljenim promenljivim treba da bude po protokolu *multiple readers – single writer*, s tim da se garantuje da vrednost promenljive `x` ne bude promenjena sve dok upis u `y` ne bude završen.

```
extern int x, y;
extern ReadersWriters* xGuard;
extern ReadersWriters* yGuard;
xGuard->startRead();
int xTemp = x;
yGuard->startWrite();
y = xTemp + 1;
yGuard->stopWrite();
xGuard->stopRead();
```

b)(5) Ako neki drugi proces radi ovu istu grupu operacija na isti način, s tim da samo zamenjuje uloge `x` i `y` (čita iz `y` i upisuje u `x`), koji problem je moguć? Obrazložiti.

Moguća je mtrva blokada (*deadlock*): prvi proces zaključava `x` na čitanje (*read lock* pomoću `startRead()`), tako da zabranjuje upis u `x`, drugi proces to isto radi za `y`, prvi proces traži ključ za upis u `y` (*write lock* pomoću `startWrite()`) i blokira se, drugi proces to isto radi za `x`, i tako se uzajamno mrtvo blokiraju.

Zadatak - Jun 2006.

Drajver nekog uređaja prima zahteve za operacijama tipa A i B u dva reda čekanja. Prelazak sa obrade zahteva za operacijom A na zahtev za operacijom B i obratno nosi relativno veliki režijski trošak i zbog toga ovaj drajver obrađuje zahteve po sledećem algoritmu: ukoliko je završena obrada zahteva jednog tipa, prihvata se i obrađuje zahtev istog tipa (uzima se iz istog reda); ako takvog zahteva nema, obrađuje se zahtev drugog tipa (iz drugog reda).

a)(5) Koji problem postoji u ovakvom pristupu? Imenovati i precizno opisati problem.

b)(5) Predložiti modifikaciju opisanog algoritma koji rešava ovaj problem.

a)Postoji problem izgladnjivanja (*starvation*): ukoliko pristigne novi zahtev istog tipa kao i onaj koji se trenutno obrađuje, i to se stalno dešava (ili se dešava u jako dugom vremenskom intervalu), zahtevi drugog tipa ne bivaju opsluženi.

b) Postoji mnogo različitih prihvatljivih rešenja. Jedna klasa rešenja podrazumeva upotrebu tehnike starenja (*aging*) koja je prikazivana na predavanjima. Na primer, može se ograničiti broj uzastopno opsluženih zahteva istog tipa ukoliko je sve vreme postojao zahtev drugog tipa itd.

Zadatak – Januar 2008

Data je sledeća pogrešna implementacija protokola više čitalaca – jedan pisac (*multiple readers – single writer*) pomoću klasičnog monitora i uslovne promenljive. Osim favorizovanja čitalaca i izgladnjivanja pisaca, ova implementacija poseduje još jedan značajan problem – moguće je i izgladnjivanje čitalaca. Precizno objasniti kako ovaj problem može da nastane.

```
monitor ReadersWriters
  export startReading, stopReading, startWriting, stopWriting;

  var
    isWriting : boolean;
    readersCount : integer;
    queue : condition;

procedure startWriting;
begin
  if (isWriting or readersCount>0) wait(queue);
  isWriting := true;
end;

procedure stopWriting;
begin
  isWriting := false;
  signal(queue);
end;
```

```
procedure startReading;
begin
  if (isWriting) wait(queue);
  readersCount := readersCount + 1;
end;

procedure stopReading;
begin
  readersCount := readersCount - 1;
  if (readersCount = 0) signal(queue);
end;

begin
  isWriting := false;
  readersCount := 0;
end;
```

Izgladnjivanje čitalaca može da nastane po sledećem scenariju. Neka je u datom intervalu vremena u toku pisanje (pisac je prošao kroz `startWriting`). Tokom ovog intervala, sve dok ovaj pisac piše i ne izvrši `stopWriting`, svi novopridošli čitaoci (označimo ih sa „čekajućim čitaocima“) blokiraju se na uslovnoj promenljivoj `queue` u operaciji `startReading`. Kada pisac završi sa pisanjem i izvrši `stopWriting`, samo jedan od čekajućih čitalaca se deblokira i izlazi iz procedure `startReading`, dok ostali i dalje čekaju. Tokom čitanja tog deblokiranog čitaoca, mogu da stižu novi čitaoci koji odmah prolaze kroz `startReading` (jer je `isWriting=false`), dok ostali čekajući čitaoci, kao i eventualni pisci, ostaju da čekaju potencijalno beskonačno, pošto ni jedan čitalac ne izvrši `signal(queue)` jer je stalno `readersCount>0`. Problem je što nisu svi čekajući čitaoci pušteni da čitaju čim je pisac završio pisanje.

Zadatak – April 2006.

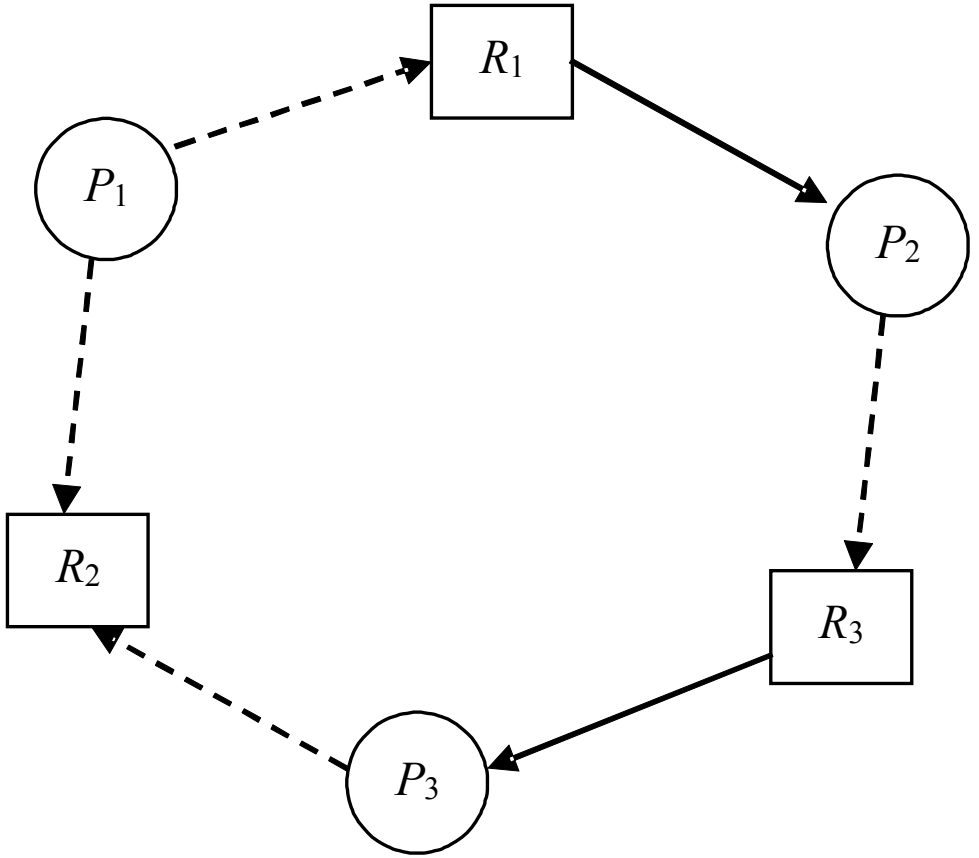
• U nekom sistemu implementiran je algoritam izbegavanja mrtvog blokiranja zasnovan na analizi bezbednih stanja pomoću grafa alokacije. U sistemu postoji samo po jedna instanca resursa R_1, R_2 i R_3, a pokrenuti su procesi P_1, P_2 i P_3. U nekom trenutku sistem se nalazi u sledećem stanju zauzetosti resursa istog tipa:

a)(4) Nacrtati graf alokacije resursa za navedeno stanje.

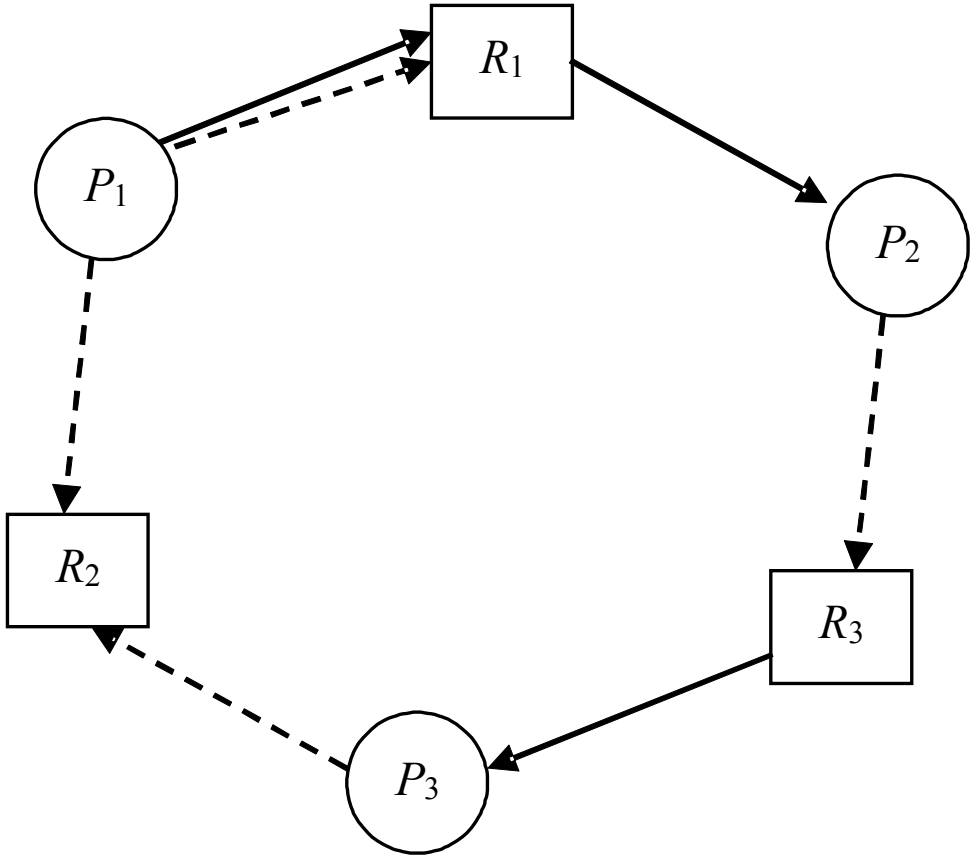
b)(3) Ako u datom stanju proces P_1 zatraži korišćenje resursa R_1, da li je novonastalo stanje sigurno?

c)(3) Ako u datom stanju proces P_1 zatraži korišćenje resursa R_2, da li sistem to treba da mu dozvoli?

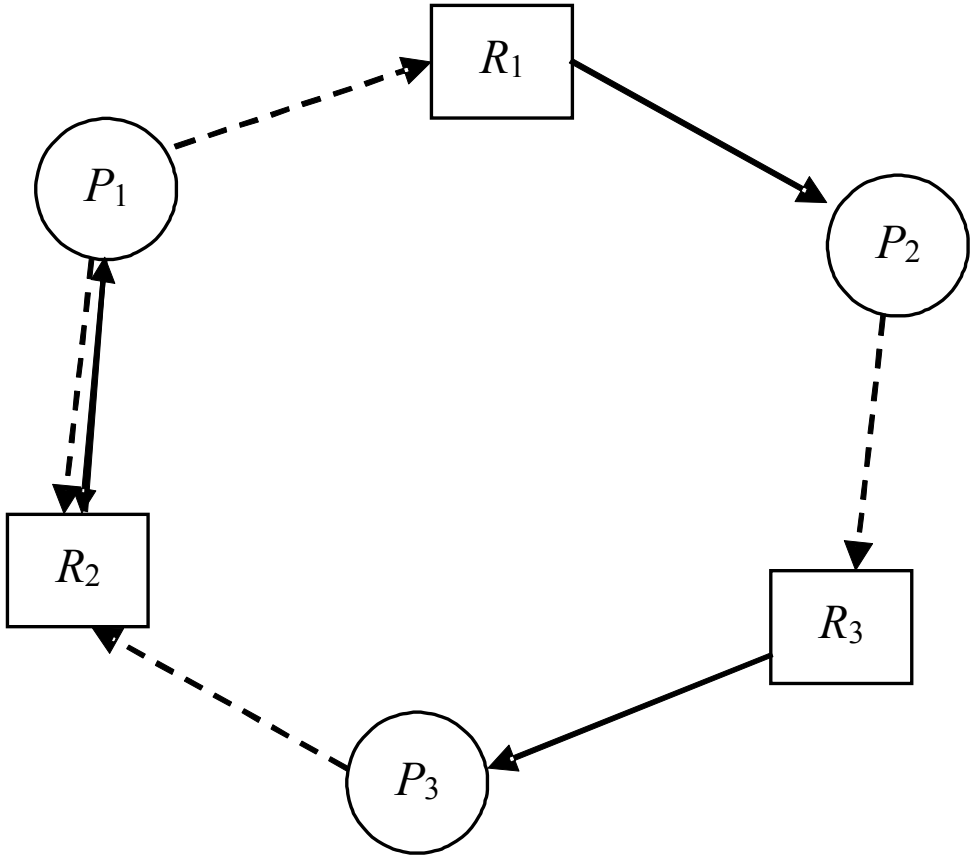
Proces	Zauzeo	Potencijalno traži (najavio korišćenje)
P_1		R_1, R_2
P_2	R_1	R_1, R_3
P_3	R_3	R_2, R_3



P_1 traži R_1



P_1 traži R_2



Zadatak – Septembar 2013.

U nekom sistemu su tri procesa (P1, P2, P3) i tri različite instance resursa (R1, R2, R3). Primenjuje se izbegavanje mrtve blokade praćenjem zauzeća resursa pomoću grafa, a resurs se dodeljuje procesu čim je to moguće i dozvoljeno. Posmatra se sledeća sekvenca operacija:

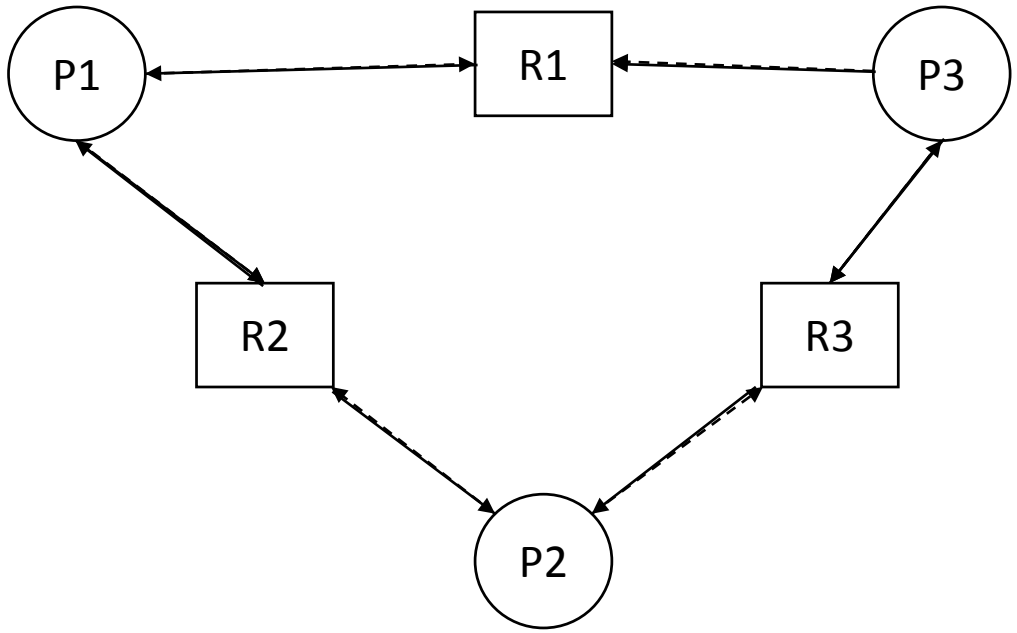
```
P2.request(R3), P1.request(R1), P2.request(R2),
P3.request(R3), P2.release(R3), P1.request(R2),
P3.request(R1), P2.release(R2), P1.release(R1),
 P3.release(R3), P1.release(R2), P3.release(R1)
```

Procesi najavljuju korišćenje onih i samo onih resursa koje zauzimaju u datoj sekvenci.

a)(7) Do kog dela se ova sekvenca može izvršiti baš u datom redosledu, ako se primenjuje izbegavanje mrtve blokade? Nacrtati graf zauzeća resursa u tom trenutku.

b)(3) Nakon koje operacije će proces koji prvi nije dobio resurs odmah kad ga je tražio dobiti taj resurs?

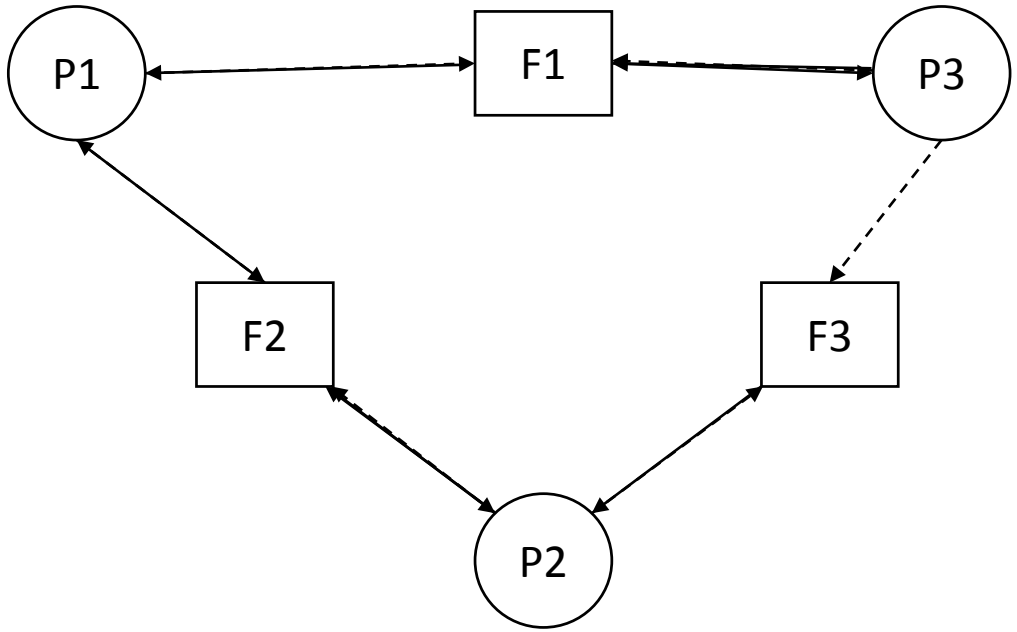
P2.request(R3),
P1.request(R1),
P2.request(R2),
P3.request(R3),
P2.release(R3),
P1.request(R2),
P3.request(R1),
P2.release(R2),
P1.release(R1),
P3.release(R3),
P1.release(R2),
P3.release(R1)



Zadatak – Oktobar 2010.

Na problemu filozofa koji večeraju (*dining philosophers*) demonstrira se mehanizam izbegavanja mrtve blokade (*deadlock avoidance*) zasnovan na grafu alokacije. Svaki filozof traži najpre svoju desnu viljušku, pa kada nju dobije, traži i svoju levu viljušku. Potrebno je prikazati graf alokacije za svako dole navedeno stanje, tim redom. U svakom traženom stanju graf prikazati nakon što je sistem dodelio resurse svima kojima su ih tražili, a kojima se mogu dodeliti resursi. Grane najave posebno naznačiti da bi se jasno razlikovale od ostalih (crtati ih isprekidano, drugom bojom ili slično). Prikazati graf za sledeća stanja:

a)(2) kada svi filozofi razmišljaju, potom
b)(3) nakon što su svi filozofi zatražili svoju desnu viljušku, potom
c)(2) nakon što su svi oni filozofi koji su dobili svoju desnu viljušku, zatražili i svoju levu viljušku, potom
d)(3) nakon što su svi oni koji su dobili obe viljuške završili sa jelom.


P1
F1
P3
F2
F3
P2

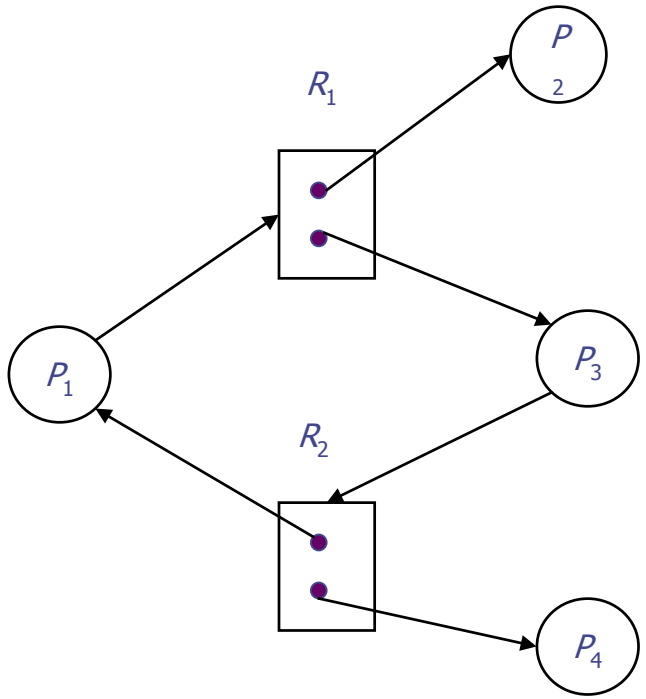
Zadatak – Septembar 2006.

U sistemu postoje četiri procesa, P_1, P_2, P_3 i P_4, i po dve instance dva tipa resursa R_1 i R_2. Odigrao se sledeći scenario: P_4 traži jednu instancu R_2, P_3 traži jednu instancu R_1, P_1 traži jednu instancu R_2, P_2 traži jednu instancu R_1, P_3 traži jednu instancu R_2, P_1 traži jednu instancu R_1.

a)(5) Nacrtati graf alokacije u ovom trenutku (nakon ovog scenarija).

b)(5) Da li u ovom sistemu u datom trenutku (nakon ovog scenarija) postoji mrtva blokada (*deadlock*)? Precizno obrazložiti odgovor.

a)



b) Mrtva blokada ne postoji, jer kada P_4 (koji nije blokiran) oslobodi resurs R_2, P_3 će dobiti taj resurs i nastaviti svoje izvršavanje, čime će osloboditi R_1, pa će i P_1 moći da nastavi izvršavanje. P_2 svakako nije blokiran.

Zadatak – Januar 2006.

b)(5) U nekom sistemu implementiran je bankarev algoritam izbegavanja mrtvog blokiranja zasnovan na analizi bezbednih stanja. U nekom trenutku sistem se nalazi u sledećem stanju zauzetosti resursa istog tipa:

Proces	Zauzeo	Najviše traži
P_1	0	5
P_2	4	11
P_3	3	6
Slobodnih: 4		

Da li sistem treba da dozvoli zauzimanje još jednog resursa od strane procesa P_1 koji ovaj u tom trenutku zahteva? Obrazložiti.

Proces	Zauzeo	Najviše traži
P_1	0	5
P_2	4	11
P_3	3	6
Slobodnih: 4		

Proces	Zauzeo	Najviše traži
P_1	1	5
P_2	4	11
P_3	3	6
Slobodnih: 4		

Da li je stanje bezbedno?

Proces	Zauzeo	Najviše traži
P_1	1	5
P_2	4	11
P_3	3	6
Slobodnih: 3		

Proces	Zauzeo	Najviše traži
P_1	1	5
P_2	4	11
P_3	6	6
Slobodnih: 0		

Sigurna sekvenca: P3

Proces	Zauzeo	Najviše traži
P_1	1	5
P_2	4	11
P_3	0	6
Slobodnih: 6		

Sigurna sekvenca: P3

Proces	Zauzeo	Najviše traži
P_1	5	5
P_2	4	11
P_3	0	6
Slobodnih: 2		

Sigurna sekvenca: P3, P1

Proces	Zauzeo	Najviše traži
P_1	0	5
P_2	4	11
P_3	0	6
Slobodnih: 7		

Sigurna sekvenca: P3, P1

Proces	Zauzeo	Najviše traži
P_1	0	5
P_2	11	11
P_3	0	6
Slobodnih: 0		

Sigurna sekvenca: P3, P1, P2

Zadatak – Septembar 2009.

Kada se primenjuje bankarev algoritam za izbegavanje mrtve blokade (*deadlock*), kada dati proces zahteva resurse vektorom zahteva (N_{R1}, N_{R2}, ..., N_{Rk},), ako bi alokacija traženih resursa odvela sistem u nebezbedno stanje, resursi se ne alociraju, već se dati proces suspenduje (blokira). Precizno objasniti kada i pod kojim uslovom se taj proces deblokira i šta se tada radi?

Kada se dealocira neki broj bilo kog od resursa koje je dati proces P tražio u zahtevu (N_{R1}, N_{R2}, ..., N_{Rk}), tj. kada se dealocira neki broj instanci resursa R_i za koji je $N_{Ri}>0$, ispituje se da li bi tada dati zahtev suspendovanog procesa P odveo sistem u nebezbedno stanje primenom bankarevog algoritma. Ako bi, P se i dalje ostavlja suspendovan. Ako ne bi, proces P se deblokira, a njegov zahtev zadovoljava alokacijom traženih resursa.

Zadatak – Oktobar 2006.

U sistemu postoje četiri procesa, P_1, P_2, P_3 i P_4, i tri tipa resursa *A*, *B* i *C*. U nekom trenutku sistem se nalazi u sledećem stanju zauzeća resursa:

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	4	3	3
P_2	0	2	0	3	5	0			
P_3	2	1	1	4	2	1			
P_4	0	3	1	3	4	5			

U sistemu se primenjuje bankarev algoritam izbegavanja mrtvog blokiranja. Da li sistem treba da dozvoli zauzeće još 2 instance resursa A od strane procesa P_4? Precizno obrazložiti odgovor, uz navođenje svih koraka primene bankarevog algoritma.

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	4	3	3
P_2	0	2	0	3	5	0			
P_3	2	1	1	4	2	1			
P_4	0	3	1	3	4	5			

Da li je stanje bezbedno?

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	2	3	3
P_2	0	2	0	3	5	0			
P_3	2	1	1	4	2	1			
P_4	2	3	1	3	4	5			

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	0	2	3
P_2	0	2	0	3	5	0			
P_3	4	2	1	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	4	4	4
P_2	0	2	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	1	1	4
P_2	3	5	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3, P2

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	1	2	0	5	7	3	4	6	4
P_2	0	0	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3, P2

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	5	7	3	5	7	3	0	1	1
P_2	0	0	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3, P2, P1

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	0	0	0	5	7	3	5	8	4
P_2	0	0	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	2	3	1	3	4	5			

Sigurna sekvenca: P3, P2, P1

	Allocations			*Max*			*Available*		
	A	B	C	A	B	C	A	B	C
P_1	0	0	0	5	7	3	4	7	0
P_2	0	0	0	3	5	0			
P_3	0	0	0	4	2	1			
P_4	3	4	5	3	4	5			

Sigurna sekvenca: P3, P2, P1,P4

Zadatak – Oktobar 2009.

U nekom sistemu primenjuje se bankarev algoritam izbegavanja mrtve blokade (*deadlock*). Strukture podataka koje se vode u sistemu i neke operacije nad njima definisane su na sledeći način:

```
class ResourceAllocationVector {
public:
  ResourceAllocationVector (); // Default constructor, null allocation
  ResourceAllocationVector(const ResourceAllocationVector&);
  ResourceAllocationVector& operator=(const ResourceAllocationVector&);
  //...
};
typedef ResourceAllocationVector RAV;

int operator<=(const RAV& v1, const RAV& v2);  // Is v1<=v2?
int operator<(const RAV& v1, const RAV& v2);  // Is v1<v2?
RAV operator+(const RAV& v1, const RAV& v2); // Returns v1+v2
RAV operator-(const RAV& v1, const RAV& v2); // Returns v1-v2
void copy(RAV from[],int numOfElements,RAV to[]); // Copies 'from' to 'to'
```

- `const int NumOfProcesses; // Number of processes`

```
RAV allocation[NumOfProcesses]; // Process-Resource allocation matrix
RAV maxNeed[NumOfProcesses]; // Process-Max resource need matrix
RAV available; // Current resource availability
```

Potrebno je realizovati sledeću operaciju bankarevog algoritma koja treba da proveri da li je tekuće stanje bezbedno i vrati 1 ako jeste, odnosno 0 ako nije:

```
int isSafeState ();
```

```
int isSafeState () {
  RAV allocTemp[NumOfProcesses];
  copy(allocation,NumOfProcesses,allocTemp);
  RAV availTemp = available;
  int completed[NumOfProcesses];
  for (int i=0; i<NumOfProcesses; i++) completed[i]=0;
  while (1) {
    int found = 0, allCompleted = 1;
    for (int i=0; i<NumOfProcesses; i++) {
      if (completed[i]) continue;
      allCompleted = 0;
      if (maxNeed[i]<=allocTemp[i]+availTemp) {
        found = 1;
        completed[i]=1;
        availTemp = availTemp+allocTemp[i];
      }
    }
    if (allCompleted) return 1;
    if (!found) return 0;
  }
}
```

Zadatak – Oktobar 2008.

U nekom sistemu mrtva blokada (*deadlock*) sprečava se zabranom kružnog čekanja tako što su resursi numerisani celim brojevima 0..RN-1, pa svaki proces sme da traži resurse samo u rastućem poretku ove numeracije. U strukturi PCB postoji sledeći niz `allocatedResources`:

```
struct PCB {
  //...
  int allocatedResources[RN];
};
PCB* running; // The running process
```

Ovaj niz je niz Bulovih promenljivih, tako da element *i* ukazuje da je dati proces alocirao resurs sa numeracijom *i*. Pristup do resursa obezbeđuje klasa `Resource`:

```
class DeadlockPreventionException {
public:
  DeadlockPreventionException (int resourceID);
  //...
};
```

```
class Resource {
public:
  Resource (int resourceID) : myID(resourceID), sem(1) {}
  void alloc () throw (DeadlockPreventionException);
  void dealloc ();
private:
  int myID;
  Semaphore sem;  // For resource allocation (mutual
exclusion)
};

void Resource::dealloc () {
  running->allocatedResources[myID]=0;
  sem.signal();
}
```

Napisati kod operacije `alloc()` koji treba da baci izuzetak datog tipa u slučaju prekršaja navedenog pravila sprečavanja mrtve blokade i obavi ostale potrebne režijske radnje alokacije resursa.

```
void Resource::alloc () throw
            DeadlockPreventionException) {
  for (int i=RN-1; i>=myID; i--)
    if (running->allocatedResources[i])
      throw DeadlockPreventionException(myID);
  sem.wait();
  running->allocatedResources[myID]=1;
}
```