Elektrotehnički fakultet Beograd

Operativni sistemi 2 (SI3OS2)

Rešeni zadaci sa prethodnih kolokvijuma

zaključno sa septembrom 2013.

Sadržaj:

1. (Novembar 2006) Raspoređivanje procesa

U nekom *preemptive time-sharing* operativnom sistemu, u jednom trenutku označenom sa 0 aktivna su sledeća tri procesa sa sledećim vrednostima preostalog vremena naleta CPU izvršavanja (*CPU burst*) i nalaze se u redu spremnih navedenom redosledu svoje aktivacije:
P1 - 3, P2 - 4, P3 – 2.
U trenutku 4 aktivira se još i proces P4 čiji je nalet izvršavanja 2 jedinice vremena. On se smešta na kraj reda spremnih odmah iza procesa koji je u tom istom trenutku upravo izgubio procesor zbog isteka vremenskog kvantuma. Vremenski kvantum koji se dodeljuje procesima za izvršavanje iznosi 1 jedinicu vremena.
Napisati kojim redosledom će se izvršavati ovi procesi za dati algoritam raspoređivanja. U odgovoru navesti samo sekvencu oznaka procesa (npr.: P1, P2, P3, P4 itd.), pri čemu svaki element u sekvenci označava izvršavanje navedenog procesa u trajanju jednog vremenskog kvantuma.

a) *Round-Robin*

Odgovor: P1, P2, P3, P1, P2, P3, P1, P4, P2, P4, P2

b) *Shortest Job First*

Odgovor: P3, P3, P1, P1, P1, P4, P4, P2, P2, P2, P2

2. (Novembar 2007) Raspoređivanje procesa

U nekom *preemptive time-sharing* operativnom sistemu, u red spremnih dolaze procesi sledećih karakteristika (aktivacija znači dolazak u red spremnih iz stanja suspenzije):

Proces	Trenutak aktivacije	Trajanje narednog naleta izvršavanja (*CPU burst*)
P1	2	3
P2	0	6
P3	1	6
P4	4	2

Proces koji se aktivira smešta se na kraj reda spremnih neposredno *ispred* procesa koji je u tom istom trenutku upravo izgubio procesor zbog isteka vremenskog kvantuma. Vremenski kvantum koji se dodeljuje procesima za izvršavanje iznosi 1 jedinicu vremena.
Napisati kojim redosledom će se izvršavati ovi procesi za dati algoritam raspoređivanja. U odgovoru navesti samo sekvencu oznaka procesa (npr.: P1, P2, P3, P4 itd.), pri čemu svaki element u sekvenci označava izvršavanje navedenog procesa u trajanju jednog vremenskog kvantuma.

a) *Round-Robin*

Odgovor:

P2, P3, P2, P1, P3, P2, P4, P1, P3, P2, P4, P1, P3, P2, P3, P2, P3

b) *Shortest Job First*

Odgovor:

P2, P2, P1, P1, P1, P4, P4, P2, P2, P2, P2, P3, P3, P3, P3, P3, P3

3. (Novembar 2008) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) sa sledećim parametrima:

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*LowPriority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši prioritet, a LP najniži.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 4, za MP je RR sa vremenskim kvantumom 8, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red HP. Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje.

Posmatra se proces P koji ima sledeću karakteristiku izvršavanja (C*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava čekanje na I/O operaciju):
C16, B, C6, B, C40, B, C2, B, C10
Napisati sekvencu koja označava redove spremnih procesa u koji se redom smešta P tokom svog životnog veka, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Odgovor: HP, MP, LP, HP, MP, HP, MP, LP, HP, HP, MP

4. (Oktobar 2009) Raspoređivanje procesa

U nekom sistemu koristi se aproksimacija SJF algoritma raspoređivanja procesa, uz predviđanje trajanja narednog naleta izvršavanja eksponencijalnim usrednjavanjem sa α=1/2 i pretpostavljenim početnim vrednostima $t_0=\tau_0=4$. Pretpostavljena (predviđena) vrednost trajanja narednog naleta izvršavanja je uvek ceo broj koji se dobija odsecanjem, a ne zaokruživanjem. U prvoj vrsti sledeće tabele date su vrednosti stvarnog trajanja nekoliko prvih naleta izvršavanja posmatranog procesa. U ćelije druge vrste upisati pretpostavljene (predviđene) vrednosti trajanja svakog od tih naleta.

6	8	10	2	10	2	10	8	8	8

Odgovor:

6	8	10	2	10	2	10	8	8	8
4	5	6	8	5	7	4	7	7	7

5. (Oktobar 2010) Raspoređivanje procesa

U sistemu se aktiviraju (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	2	2
B	1	3	2
C	2	0	4
D	3	1	1

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, kao i vreme odziva procesa i ukupno srednje vreme odziva procesa, ako je algoritam raspoređivanja:

a)(5) po prioritetu sa preuzimanjem (*Preemptive Priority Scheduling*)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	2	4	2
B	4	6	3
C	0	8	8
D	8	9	8
Srednje vreme odziva:			5,25

b)(5) najkraći-posao-prvi sa preuzimanjem (*Preemptive SJF*).

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	2	4	2
B	4	6	3
C	0	9	9
D	1	2	1
Srednje vreme odziva:			3,75

6. (Ispit Januar 2011) Raspoređivanje procesa

U nekom *preemptive time-sharing* operativnom sistemu, u jednom trenutku označenom sa 0, aktivna su sledeća tri procesa sa sledećim vrednostima preostalog vremena naleta CPU izvršavanja (*CPU burst*) i nalaze se u redu spremnih u navedenom redosledu svoje aktivacije:
P1 - 4, P2 - 8, P3 – 6.
U trenutku 8 aktivira se još i proces P4 čiji je nalet izvršavanja 4 jedinice vremena. On se smešta na kraj reda spremnih odmah iza procesa koji je u tom istom trenutku upravo izgubio procesor zbog isteka vremenskog kvantuma. Vremenski kvantum koji se dodeljuje procesima za izvršavanje iznosi 2 jedinice vremena.
Napisati kojim redosledom će se izvršavati ovi procesi za dati algoritam raspoređivanja. U odgovoru navesti samo sekvencu oznaka procesa (npr.: P1, P2, P3, P4 itd.), pri čemu svaki element u sekvenci označava izvršavanje navedenog procesa u trajanju jednog vremenskog kvantuma.

a) *Round-Robin*
Odgovor: P1, P2, P3, P1, P2, P3, P4, P2, P3, P4, P2.

b) *Shortest Job First*
Odgovor: P1, P1, P3, P3, P3, P4, P4, P2, P2, P2, P2

7. (Oktobar 2011) Raspoređivanje procesa

U nekom sistemu koristi se Multilevel Feedback-Queue Scheduling (MFQS), uz dodatni mehanizam koji ga približava ponašanju SJF algoritma. Svakom procesu pridružena je procena trajanja narednog naleta izvršavanja τ koja se izračunava eksponencijalnim usrednjavanjem sa $\tau = 1/2$, uz odsecanje na ceo broj.

- Postoje tri reda spremnih procesa: HP (High Priority), MP (Medium Priority) i LP (Low Priority).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje po redovima je sledeće: za HP je Round-Robin (RR) sa vremenskim kvantumom 5, za MP je RR sa vremenskim kvantumom 10, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red prema proceni τ : ako je $\tau<5$, smešta se u HP, ako je $5< \tau < 10$, smešta se u MP, inače se smešta u LP.
- Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje. Procena τ se ažurira kada se završi ceo nalet izvršavanja, odnosno kada se proces blokira.

Posmatra se novoaktivirani proces P sa inicijalnom procenom $\tau = 8$ koji ima sledeću karaktekteristiku narednog izvršavanja (Rn označava jedan nalet izvršavanja u trajanju n jedinica vremena, B označava I/O operaciju):
R2, B, R7, B, R15, B, R1, B, R4
Napisati sekvencu koja označava redove spremnih procesa u kojima redom boravi P, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Rešenje: MP, HP, MP, MP, LP, MP, HP

8. (Septembar 2012) Raspoređivanje procesa

U nekom sistemu klasa `Scheduler`, čiji je interfejs dat dole, implementira raspoređivač spremnih procesa po prioritetu (engl. *priority scheduling*). Implementirati ovu klasu tako da i operacija dodavanja novog spremnog procesa `put()` i operacija uzimanja spremnog procesa koji je na redu za izvršavanje `get()` budu ograničene po vremenu izvršavanja vremenom koje ne zavisi od broja spremnih procesa (kompleksnost $O(1)$). Između premnih procesa sa istim prioritetom raspoređivanje treba da bude *FCFS*. Konstanta `MAXPRI` je maksimalna vrednost prioriteta (prioriteti su u opsegu `0..MAXPRI`, s tim da je 0 najviši prioritet). U slučaju da nema drugih spremnih procesa, treba vratiti proces na koga ukazuje `idle` (to je uvek spreman proces najnižeg prioriteta). U strukturi `PCB` polje `priority` tipa `int` predstavlja prioritet procesa, a polje `next` pokazivač tipa `PCB*` koji služi za ulančavanje struktura PCB u jednostruke liste.

```
const int MAXPRI = …;
extern PCB* idle;

class Scheduler
{
public:
  Scheduler
  ();
  PCB* get ();
  void put (PCB*);
};
```

Rešenje:

```
const int MAXPRI = …;
extern PCB* idle;

class Scheduler {
public:
  Scheduler ();
  PCB* get ();
  void put (PCB*);
private:
  PCB* head[MAXPRI+1];
  PCB* tail[MAXPRI+1];
};

Scheduler::Scheduler () {
  for (int i=0; i<MAXPRI+1; i++)
    head[i]=tail[i]=0;
}

PCB* Scheduler::get () {
  for (int i=0; i<=MAXPRI; i++)
    if (head[i]) {
      PCB* ret = head[i];
      head[i] = head[i]->next;
      if (head[i]==0) tail[i]=0;
      ret->next = 0;
      return ret;
    }
  return idle;
}

void Scheduler::put (PCB* pcb) {
  if (pcb==0) return; // Exception!
  int p = pcb->priority;
  if (p<0 || p>MAXPRI) return; // Exception!
  pcb->next = 0;
  if (tail[p]==0)
    tail[p] = head[p] = pcb;
  else
    tail[p] = tail[p]->next = pcb;
}
```

9. (Oktobar 2012) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS), uz dodatni mehanizam koji ga približava ponašanju SJF algoritma. Svakom procesu pridružena je procena trajanja narednog naleta izvršavanja τ koja se izračunava eksponencijalnim usrednjavanjem sa $\alpha=1/2$, uz odsecanje na ceo broj.

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*Low Priority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 5, za MP je RR sa vremenskim kvantumom 10, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red prema proceni τ: ako je $\tau < 5$, smešta se u HP, ako je $5 < \tau < 10$, smešta se u MP, inače se smešta u LP.
- Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje. Procena τ se ažurira kada se završi ceo nalet izvršavanja, odnosno kada se proces blokira.

Posmatra se novoaktivirani proces P sa inicijalnom procenom $\tau = 12$ koji ima sledeću karaktekteristiku narednog izvršavanja (R*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava I/O operaciju):
R2, B, R7, B, R2, B, R7, B, R7
Napisati sekvencu koja označava redove spremnih procesa u kojima redom boravi P, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Rešenje: LP, MP, MP, HP, MP, HP, MP

10. (Septembar 2013) Raspoređivanje procesa

U nekom sistemu klasa Scheduler, čiji je delimična implementacija data dole, realizuje raspoređivač spremnih procesa po prioritetu (engl. priority scheduling), tako da i operacija dodavanja novog spremnog procesa put() i operacija uzimanja spremnog procesa koji je na redu za izvršavanje get() imaju ograničeno vreme izvršavanja koje ne zavisi od broja spremnih procesa (kompleksnost O(1)). Promenljiva maxPri služi da ubrza pristup do najprioritetnijeg procesa. Između spremnih procesa sa istim prioritetom raspoređivanje je FCFS. Konstanta MAXPRI je maksimalna vrednost prioriteta (prioriteti su u opsegu 0..MAXPRI, s tim da je 0 najniži prioritet). U slučaju da nema drugih spremnih procesa, treba vratiti proces na koga ukazuje idle (to je uvek spreman proces najnižeg prioriteta). U strukturi PCB polje priority tipa int predstavlja prioritet procesa, a polje next pokazivač tipa PCB* koji služi za ulančavanje struktura PCB u jednostruke liste.

Realizovati operaciju `Scheduler::get()`.

```
const int MAXPRI = …;
extern PCB* idle;

class Scheduler
{
public:
  Scheduler ();
  PCB* get ();
  void put (PCB*);
private:
  PCB* head[MAXPRI+1];
  PCB* tail[MAXPRI+1];
  int maxPri;
};

Scheduler::Scheduler () : maxPri(-1)
{
  for (int i=0; i<MAXPRI+1; i++)
    head[i]=tail[i]=0;
}

void Scheduler::put (PCB* pcb) {
  if (pcb==0) return; // Exception!
  int p = pcb->priority;
  if (p<0 || p>MAXPRI) return; // Exception!
  pcb->next = 0;
  if (tail[p]==0)
    tail[p] = head[p] = pcb;
  else
    tail[p] = tail[p]->next = pcb;
  if (p>maxPri) maxPri=p;
 }
```

Rešenje:

```
PCB* Scheduler::get () {
  if  (maxPri==-1)return  idle;
  PCB* ret = head[maxPri];
  head[maxPri] = head[maxPri]->next;
  if (head[maxPri]==0) {
    tail[maxPri]=0;
    while (maxPri>=0 && head[maxPri]==0) maxPri--;
  }
  ret->next = 0;
  return ret;
}
```

11. (ispit 2006) Raspoređivanje procesa

U jezgru nekog operativnog sistema primenjuje se raspoređivanje procesa po prioritetima (*priority scheduling*). Date su deklaracije iz ovog jezgra:

```
typedef unsigned int Priority;

struct PCB {  // Process Control Block
  PCB* next;  // Next PCB in the Ready or a waiting list
  Priority pri; // Current priority of the process
  ...
}

PCB* ready; // Head of the Ready list

void putInReady (PCB*); // Adds the given PCB to the Ready list
PCB* getFromReady(); // Removes and returns the PCB to schedule
```

Operacije `putInReady()` i `getFromReady()` su operacije raspoređivača; operacija `getFromReady()` vadi iz liste spremnih i vraća PCB onog procesa koji je na redu za izvršavanje. Implementirati ove dve operacije.

```
void putInReady (PCB* p) {
  if (p==0) return;
  if (ready==0 || p->pri>ready->pri) {
    p->next = ready;
    ready = p;
    return;
  }
  for (PCB* cur = ready; cur!=0; cur=cur->next)
    if (cur->next==0 || p->pri>cur->next->pri) {
      p->next = cur->next;
      cur->next = p;
      return;
    }
}

PCB* getFromReady() {
  PCB* p = ready;
  if (ready) ready = ready->next;
  return p;
}
```

12. (ispit 2006) Raspoređivanje procesa

U sistemu se aktiviraju (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	6
B	1	2	4
C	2	0	8
D	3	6	2

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, kao i vreme odziva procesa i ukupno srednje vreme odziva procesa, ako je algoritam raspoređivanja:

a)(5) po prioritetu sa preuzimanjem (*Preemptive Priority Scheduling*)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A			
B			
C			
D			
Srednje vreme odziva:			

b)(5) najkraći-posao-prvi bez preuzimanja (*Nonpreemptive SJF*).

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A			
B			
C			
D			
Srednje vreme odziva:			

a)(5)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	4	10	6
B	2	12	10
C	0	18	18
D	18	20	14
Srednje vreme odziva:			12

b)(5)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	14	20	16
B	10	14	12
C	0	8	8
D	8	10	4
Srednje vreme odziva:			10

13. (ispit 2006) Raspoređivanje procesa

U nekom operativnom sistemu bez preuzimanja (*non-preemptive*) primenjuje se aproksimacija SJF (*Shortest Job First*) algoritma raspoređivanja procesa, pri čemu se koristi predviđanje sa eksponencijalnim usrednjavanjem sa α=1/2. U nekom trenutku u sistemu se nalaze tri spremna procesa A, B i C, čije su vremenske karakteristike date u donjoj tabeli:

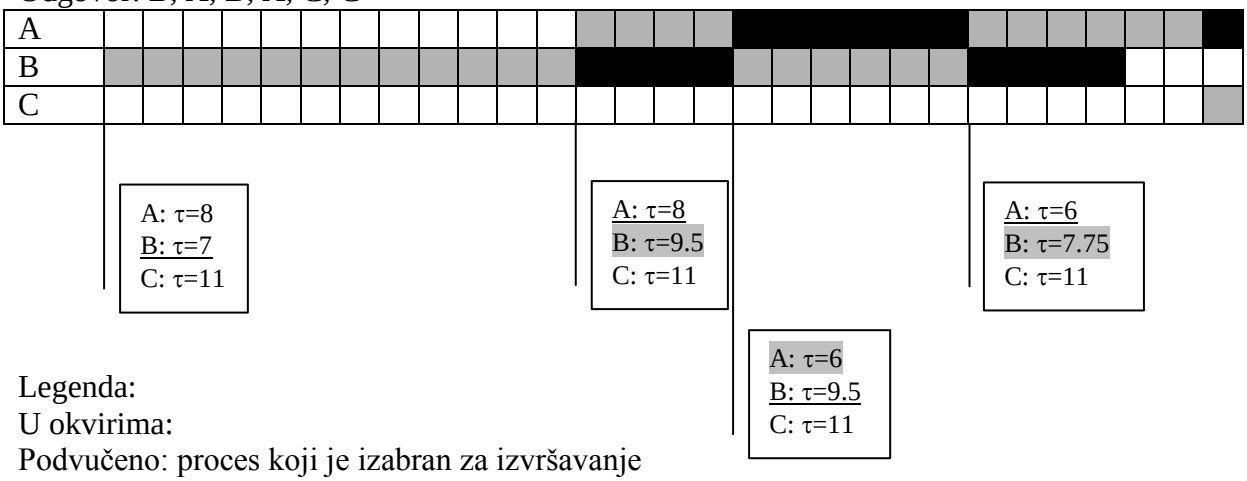
	Parametri predviđanja izvršavanja		Naredna vremena naleta izvršavanja (CPU *burst*) i I/O operacija (I/O *burst*)			
Proces	τ_n	CPU *burst* t_n	CPU *burst* (t_{n+1})	I/O *burst*	CPU *burst* (t_{n+2})	I/O *burst*
A	8	8	4	6	6	4
B	2	12	12	4	6	4
C	12	10	8	4	12	4

Tekuće vrednosti parametara za predviđanje date su u levom delu tabele. Vremena narednih naleta izvršavanja (CPU *burst*) i trajanja I/O operacija (I/O *burst*) data su redom u desnom delu tabele. Na primer, proces A će imati prvi naredni nalet izvršavanja u trajanju od 4 jedinice vremena, pa onda čekati na I/O operaciju koja traje 6 jedinica vremena, pa onda opet imati nalet izvršavanja u trajanju od 6 jedinica vremena, i na kraju čekati na I/O operaciju koja traje 4 jedinice vremena.

Navesti redosled kojim će proces izvršavati CPU nalete ovih procesa. U odgovoru navesti samo redosled kojim će procesor izvršavati CPU nalete procesa, npr. A, B, C, A, B, C, a ispod dati postupak dolaska do odgovora.

Odgovor: _____

Postupak:
Odgovor: B, A, B, A, C, C



Legenda:
U okvirima:
Podvučeno: proces koji je izabran za izvršavanje
Sivo: blokiran proces
U tabeli:
Sivo polje: proces se izvršava
Crno polje: proces blokiran (čeka I/O operaciju)

14. (ispit 2006) Raspoređivanje procesa

U nekom operativnom sistemu primenjuje se raspoređivanje procesa sa prioritetima (*Priority Scheduling*), uz starenje (*aging*) radi sprečavanja izgladnjivanja: svaki put kada se neki spreman proces izabere za izvršavanje, ostali u listi spremnih koji još čekaju dobijaju za jedan viši nivo prioriteta. PCB strukture svih procesa su prealocirane i smeštene u statički alociran i dimenzionisan niz `processes`:

```
typedef unsigned short Priority; // Priority: 0 is the lowest
const Priority maxPri = ...; // Max value of priority
const unsigned int maxProc = ...; // Max number of processes

struct PCB {
  int isUsed;  // Is this PCB used for a process (1) or is a free slot (0)?
  int isReady; // Is this process ready?
  Priority curPri; // Current priority
  Priority defPri; // Default priority (set on creation)
  ...
};

PCB processes[maxProc];
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list"
void putReady (PCB*); // Puts a process to the „ready list"
```

Rešenje:

```
PCB* getReady () {
  Priority mPri = 0;
  PCB* selProc = 0;
  for (unsigned int i=0; i<maxProc; i++) {
    PCB* p = &processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p->curPri>=mPri) {
      mPri = p->curPri;
      selProc = p;
    }
  }
  if (selProc==0) return 0;
  for (unsigned int i=0; i<maxProc; i++) {
    PCB* p = processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p!=selProc && p->curPri<maxPri) p->curPri++;
  }
  return selProc;
}

void putReady (PCB* p) {
  if (p) {
    p->isReady = 1;
    p->curPri=p->defPri;
  }
}
```

15. (ispit 2006) Raspoređivanje procesa

U nekom operativnom sistemu primenjuje se raspoređivanje procesa sa aproksimacijom SJF (*Shortest Job First*) algoritma, uz predviđanje trajanja narednog naleta izvršavanja (*CPU burst*) sa eksponencijalnim usrednjavanjem i $\alpha = ½$. PCB strukture svih procesa su prealocirane i smeštene u statički alociran i dimenzionisan niz `processes`:

```
typedef unsigned int Time; // Execution time in clock ticks

struct PCB {
  int isUsed;  // Is this PCB used for a process (1) or is a free slot (0)?
  int isReady; // Is this process ready?
  Time prediction; // Prediction of CPU burst duration
  Time lastCPUBurst; // Last actual CPU burst duration
  ...
};

PCB processes[maxProc];
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list"
void putReady (PCB*); // Puts a process to the „ready list"
```

Pretpostaviti da je pre poziva funkcije `putReady()` u član `lastCPUBurst` strukture `PCB` datog procesa (koji je upravo izgubio procesor) već upisana vrednost trajanja upravo završenog naleta izvršavanja, kao i da je u članu `prediction` vrednost procene tog naleta; u ovoj funkciji član `prediction` treba ažurirati procenom trajanja narednog naleta izvršavanja.

Rešenje:

```
PCB* getReady () {
  Time minTime = ~0;
  PCB* selProc = 0;
  for (unsigned int i=0; i<maxProc; i++) {
    PCB* p = &processes[i];
    if (!p->isUsed || !p->isReady) continue;
    if (p->prediction<=minTime) {
      minTime = p->prediction;
      selProc = p;
    }
  }
  return selProc;
}

void putReady (PCB* p) {
  if (p) {
    p->isReady = 1;
    p->prediction=(p->prediction+p->lastCPUBurst)/2;
  }
}
```

16. (ispit 2007) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) sa sledećim parametrima:

- Postoji tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*LowPriority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši prioritet, a LP najniži.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 4, za MP je RR sa vremenskim kvantumom 8, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red HP. Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje.

Posmatra se proces P koji ima sledeću karaktekteristiku izvršavanja (R*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava čekanje na I/O operaciju):
R6, B, R8, B, R24, B, R2, B, R6
Napisati sekvencu koja označava redove spremnih procesa u koji se redom smešta P, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, HP, MP, LP, HP, ...

Odgovor: HP, MP, HP, MP, HP, MP, LP, HP, HP, MP

17. (ispit 2007) Raspoređivanje procesa

U nekom trenutku u redu spremnih nalaze se sledeći procesi (u zagradi je dato vreme izvršavanja): P1(5), P2(4), P3(6), P4(1), P5(3), P6(2).
Koliko je srednje vreme čekanja ovih procesa (vreme čekanja je vreme od posmatranog trenutka do započinjanja izvršavanja datog procesa), ako je algoritam raspoređivanja: (Odgovor dati u obliku razlomka.)

a)(5) *Sortest-Job-First*? Odgovor:
(0+1+(1+2)+(1+2+3)+(1+2+3+4)+(1+2+3+4+5))/6 = 35/6

b)(5) *First-Come-First-Served*? Odgovor:
(0+5+(5+4)+(5+4+6)+(5+4+6+1)+(5+4+6+1+3))/6 = 64/6 = 32/3.

18. (ispit 2007) Raspoređivanje procesa

U jednoprocesorskom sistemu aktiviraju se (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	4
B	1	2	6
C	2	2	3
D	3	0	4

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, kao i vreme odziva procesa (definisano kao vreme koje protekne od trenutka aktivacije do trenutka završetka izvršavanja) i ukupno srednje vreme odziva procesa, ako je algoritam raspoređivanja:

a)(5) po prioritetu sa preuzimanjem (*Preemptive Priority Scheduling*)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A			
B			
C			
D			
Srednje vreme odziva:			

b)(5) najkraći-posao-prvi sa preuzimanjem (*Preemptive SJF*).

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A			
B			
C			
D			
Srednje vreme odziva:			

a)(5)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	4	8	4
B	2	12	10
C	12	15	13
D	0	17	17
Srednje vreme odziva:			11

b)(5)

Proces	Trenutak prvog izvršavanja	Trenutak završetka	Vreme odziva
A	7	11	7
B	11	17	15
C	4	7	5
D	0	4	4
Srednje vreme odziva:			7.75

19. (ispit 2007) Raspoređivanje procesa

U nekom operativnom sistemu primenjuje se raspoređivanje procesa po prioritetima (*priority scheduling*). U sistemu postoje sledeći procesi (niža vrednost-viši prioritet):

Proces	Prioritet	Vreme aktivacije	Vreme izvršavanja
A	0	6	2
B	1	2	4
C	2	4	6
D	3	0	8

Koliko je srednje ukupno vreme provedeno u sistemu ovih procesa (*turnaround time*, vreme od aktivacije do završetka izvršavanja procesa) za slučaj:

a)(5) bez preuzimanja (*non-preemptive*);

b)(5) sa preuzimanjem (*preemptive*)?

Odgovor:

a) 10

b) 9

20. (1. februar 2008.) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) sa sledećim parametrima:

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*LowPriority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši prioritet, a LP najniži.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 2, za MP je RR sa vremenskim kvantumom 4, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red HP. Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje.

Posmatra se proces P koji ima sledeću karaktekteristiku izvršavanja (R*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava čekanje na I/O operaciju):
R4, B, R8, B, R20, B, R1, B, R5
Napisati sekvencu koja označava redove spremnih procesa u koji se redom smešta P tokom svog životnog veka, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Odgovor:

HP, MP, HP, MP, LP, HP, MP, LP, HP, HP, MP

21. (ispit 2007) Raspoređivanje procesa

U nekom operativnom sistemu bez preuzimanja (*non-preemptive*) primenjuje se aproksimacija SJF (*Shortest Job First*) algoritma raspoređivanja procesa, pri čemu se koristi predviđanje sa eksponencijalnim usrednjavanjem sa $\alpha = 1/2$. U nekom trenutku u sistemu se nalaze tri spremna procesa A, B i C, čije su vremenske karakteristike date u donjoj tabeli:

Proces	Parametar predviđanja izvršavanja τ	Naredna vremena naleta izvršavanja (CPU *burst*) i I/O operacija (I/O *burst*)			
		CPU *burst* (t_{n+1})	I/O *burst*	CPU *burst* (t_{n+2})	I/O *burst*
A	6	8	4	10	4
B	7	2	2	6	4
C	9	8	4	6	4

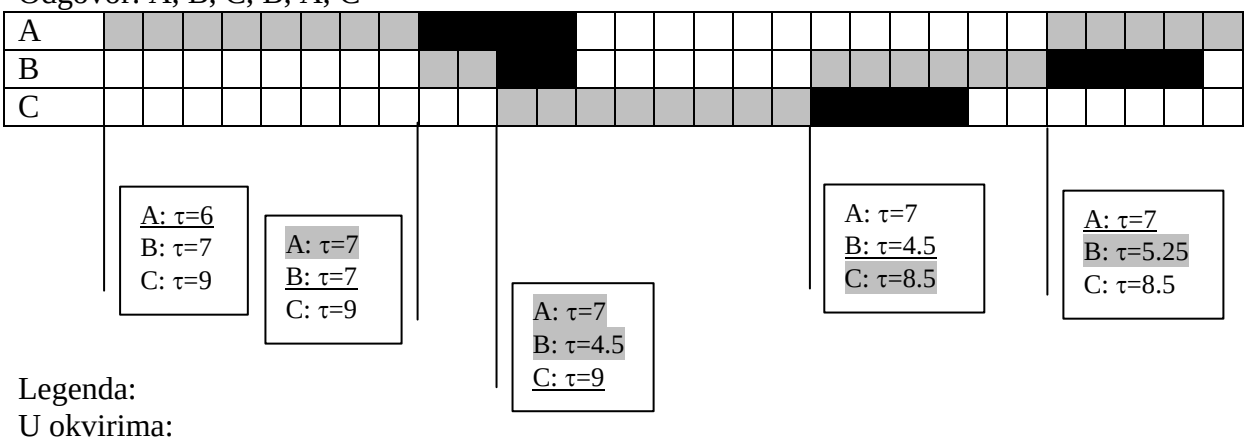
Tekuće vrednosti parametra τ za predviđanje date su u levom delu tabele. Vremena narednih naleta izvršavanja (CPU *burst*) i trajanja I/O operacija (I/O *burst*) data su redom u desnom delu tabele. Na primer, proces A će imati prvi naredni nalet izvršavanja u trajanju od 8 jedinica vremena, pa onda čekati na I/O operaciju koja traje 4 jedinice vremena, pa onda opet imati nalet izvršavanja u trajanju od 10 jedinica vremena, i na kraju čekati na I/O operaciju koja traje 4 jedinice vremena.

Navesti redosled kojim će procesor izvršavati CPU nalete ovih procesa. U odgovoru navesti samo redosled kojim će procesor izvršavati CPU nalete procesa, npr. A, B, C, A, B, C, a ispod dati postupak dolaska do odgovora.

Odgovor: _____

Postupak:

Odgovor: A, B, C, B, A, C



Legenda:
U okvirima:
Podvučeno: proces koji je izabran za izvršavanje
Sivo: blokiran proces
U tabeli:
Sivo polje: proces se izvršava
Crno polje: proces blokiran (čeka I/O operaciju)

22. (1. januar 2008.) Raspoređivanje procesa

U nekom jednoprocesorskom operativnom sistemu kreirani su sledeći procesi (niža vrednost-viši prioritet):

Proces	Prioritet	Vreme aktivacije	Vreme izvršavanja
A	0	6	2
B	1	2	4
C	2	4	6
D	3	0	10

Napisati redosled izvršavanja procesa (npr. „A, B, C, D, A, B, C, D“), za slučaj algoritma raspoređivanja procesa:

a)(5) po prioritetima sa preuzimanjem (*preemptive priority-based scheduling*);

b)(5) SJF sa preuzimanjem (*preemptive SJF*).

Odgovor:

a)(5) D, B, A, (B), C, D

Napomena: Proces B gubi procesor u trenutku kada je završio sa obradom. Ako se pretpostavi da se dato vreme odnosilo na korisnu obradu u tom procesu, onda proces B u tom trenutku još nije završen, ostaju još režijski troškovi da se taj proces ukloni iz sistema u potpunosti. Zato je drugo pojavljivanje B navedeno u zagradama. Pošto se u ovakvim zadacima često pretpostavlja da je to vreme za režijske troškove zanemarljivo malo, pa je i u ovom slučaju moguće reći da će to biti uključeno u zadato vreme izvršavanja. U tom slučaju, izvršavanje procesa B koje je navedeno u zagradi neće postojati.

b)(5) D, B, A, C, D

23. (1. jun 2008.) Raspoređivanje procesa

U nekom jednoprocesorskom operativnom sistemu kreirani su sledeći periodični procesi (niža vrednost označava viši prioritet):

Proces	Prioritet	Perioda	Vreme izvršavanja u svakoj periodi
A	0	6	2
B	1	8	2
C	2	12	4
D	3	24	2

Posmatra se izvršavanje ovih periodičnih procesa od trenutka 0, kada su svi istovremeno pokrenuti, do trenutka 24, uz raspoređivanje po prioritetima sa preuzimanjem (*preemptive priority-based scheduling*).

Napisati redosled izvršavanja delova procesa i dužina tih izvršavanja (npr. „A1, B2, C3, D1, C3, ...“), u toku vremenskog intervala [0, 24]. Zanemaruje se vreme promene konteksta.

Odgovor:

A2, B2, C2, A2, B2, C2, A2, C2, B2, A2, C2, D2

24. (ispit 2009) Raspoređivanje procesa

U nekom sistemu koristi se raspoređivanje procesa po prioritetu (*priority scheduling*), pri čemu se izgladnjivanje sprečava tehnikom starenja (*aging*) na sledeći način. Jezgro periodično, svake druge jedinice vremena, pokreće postupak u kome se svim procesima koji se trenutno nalaze u redu spremnih povećava tekući prioritet za jedan. Na taj način oni procesi koji duže čekaju u redu spremnih vremenom dobijaju sve viši prioritet. Ako neki proces u istom tom trenutku iz reda spremnih treba da dobije procesor, ne menja mu se tekući prioritet jer ne ostaje u redu spremnih; slično, tekući prioritet se ne menja ni procesu koji gubi procesor i dolazi u red spremnih. U ovom sistemu kreirani su sledeći periodični procesi sa datim inicijalnim vrednostima prioriteta prilikom svake periodične aktivacije (niža vrednost označava viši prioritet):

Proces	Prioritet	Perioda	Vreme izvršavanja u svakoj periodi
A	0	6	2
B	4	8	2
C	7	12	4
D	10	24	2

Posmatra se izvršavanje ovih periodičnih procesa od trenutka 0, kada su svi istovremeno aktivirani, do trenutka 24, uz raspoređivanje po prioritetima sa preuzimanjem (*preemptive priority-based scheduling*).

Napisati redosled izvršavanja delova procesa i dužina tih izvršavanja (npr. „A1, B2, C3, D1, C3, ...“), u toku vremenskog intervala [0, 24]. Zanemaruje se vreme promene konteksta i ostalo režijsko vreme.

Odgovor: _____

A2, B2, C2, A2, B2, C2, A2, D2, B2, A2, C4

25. (ispit 2009) Raspoređivanje procesa

U sistemu se aktiviraju (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	2	0	4
B	0	2	3
C	3	3	1
D	1	1	2

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, ako je algoritam raspoređivanja:

a)(5) po prioritetu sa preuzimanjem (*preemptive priority scheduling*)

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A		
B		
C		
D		

b)(5) najkraći-posao-prvi bez preuzimanja (*nonpreemptive SJF*).

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A		
B		
C		
D		

a)(5)

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A	0	9
B	2	5
C	9	10
D	1	6

b)(5)

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A	0	4
B	7	10
C	4	5
D	5	7

26. (ispit 2009) Raspoređivanje procesa

U nekom sistemu primenjuje se aproksimacija SJF algoritma raspoređivanja procesa, sa eksponencijalnim usrednjavanjem pri proceni trajanja sledećeg naleta izvršavanja, i sa težinskim koeficijentom α=1/2. U strukturi PCB, prilikom promene konteksta, kada je posmatranom procesu oduzet procesor, polje `timePredicted` sadrži prethodnu procenu trajanja upravo završenog naleta izvršavanja, a polje `timeElapsed` stvarno trajanje tog izvršavanja.

```
struct PCB {
  ...
  unsigned long timePredicted, timeElapsed;
};
```

a)(5) Realizovati funkciju `predictNextBurst(PCB*)` koja za proces sa datim PCB treba da proceni trajanje sledećeg naleta izvršavanja i upiše ga u polje `timePredicted`.

Rešenje:

b)(5) Za neki proces koji je imao dole prikazani niz stvarnih trajanja naleta izvršavanja, prikazati niz procena tih trajanja (početna pretpostavka je data u tabeli). (Popuniti sva četiri prazna polja u drugoj vrsti tabele.)

`timeElapsed`	10	2	8	8	...
`timePredicted`	6				

a)(5)

```
void predictNextBurst(PCB* pcb) {
  if (pcb==0) return;
  pcb->timePredicted>>=1;
  pcb->timeElapsed>>=1;
  pcb->timePredicted+=pcb->timeElapsed;
}
```

b)(5)

`timeElapsed`	10	2	8	8	...
`timePredicted`	6	8	5	6	7

27. (ispit 2009) Raspoređivanje procesa

U nekom trenutku 0 u redu spremnih procesa nalaze se tri procesa, P1, P2 i P3 u tom redosledu. Karakteristike ova tri procesa date su dole; svaki niz brojeva za dati proces prestavlja niz trajanja pojedinačnih naleta, najpre CPU izvršavanja, pa onda ulazno/izlane operacije na istom uređaju, i tako naizmenično dalje.
P1: 3, 2, 3
P2: 6, 2, 4, 2, 6
P3: 1, 4, 1, 2, 4
Napisati redosled izvršavanja ova tri procesa na procesoru (npr. P1, P2, P3, P1, P2, P3, ...) i izračunati ukupno vreme zadržavanja u sistemu (*turnaround time*, vreme koje protekne od posmatranog trenutka 0 do završetka poslednjeg datog naleta) za svaki proces, ako se primenjuje:
a)(5) FIFO algoritam raspoređivanja procesora.
b)(5) *Round-Robin* algoritam raspoređivanja procesora sa vremenskim kvantumom 4.
(Raspoređivanje I/O uređaja je uvek FIFO.)

Rešenje:

a) Redosled: _____

Vreme zadržavanja P1: _____

Vreme zadržavanja P2: _____

Vreme zadržavanja P3: _____

b) Redosled: _____

Vreme zadržavanja P1: _____

Vreme zadržavanja P2: _____

Vreme zadržavanja P3: _____

a)(5) Redosled: P1, P2, P3, P1, P2, P3, P2, P3
Vreme zadržavanja P1: 13
Vreme zadržavanja P2: 25
Vreme zadržavanja P3: 29
b)(5) Redosled: P1, P2, P3, P1, P2, P3, P2, P3, P2
Vreme zadržavanja P1: 11
Vreme zadržavanja P2: 29
Vreme zadržavanja P3: 23

28. (ispit 2009) Raspoređivanje procesa

U sistemu se aktiviraju (kreiraju ili postaju spremni) procesi sledećih karakteristika (niži broj označava viši prioritet):

Proces	Prioritet	Trenutak aktivacije	Dužina izvršavanja
A	0	4	6
B	1	2	4
C	2	0	8
D	3	6	2

U tabelu upisati u kom trenutku dati proces počinje svoje (prvo) izvršavanje i u kom trenutku se završava, kao i vreme odziva procesa i ukupno srednje vreme odziva procesa, ako je algoritam raspoređivanja:

a)(5) po prioritetu sa preuzimanjem (*Preemptive Priority Scheduling*)

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A	0	9
B	2	5
C	9	10
D	1	6

b)(5) najkraći-posao-prvi bez preuzimanja (*Nonpreemptive SJF*).

Proces	Trenutak početka prvog izvršavanja	Trenutak završetka
A	0	4
B	7	10
C	4	5
D	5	7

29. (ispit 2010) Raspoređivanje procesa

U nekom sistemu primenjuje se raspoređivanje procesa po prioritetima (*priority schceduling*), s tim da se izgladnjivanje (*starvation*) sprečava tako što se svakom procesu ograničava vreme čekanja na procesor. Predložiti i precizno opisati strukture podataka i algoritme kojima se može implementirati ovakvo raspoređivanje.

Odgovor:

Ideja rešenja je ista kao što se sprovodi raspoređivanje zahteva za diskom u Linuxu, uz sprečavanje izgladnjivanja.

Potrebne su dve strukture:
1) Struktura u koju se smeštaju spremni procesi, uređeni po prioritetu. To može biti npr. lista PCB struktura (dvostruko ulančana, radi efikasnosti umetanja i izbacivanja), ili mapa koja omogućava direktno preslikavanje prioriteta u PCB (npr. niz u kome svaki ulaz predstavlja glavu liste PCB struktura sa istim prioritetom).
2) Lista (dvostruko ulančana, zbog efikasnosti umetanja i izbacivanja) PCB struktura uređena hronološki, po vremenu isteka roka čekanja, na primer na sledeći način. Svaki element u listi čuva vreme za koje mu ističe rok, relativno u odnosnu na prethodni element u listi (može biti i 0 kada dva susedna elementa imaju isti rok), dok prvi element u listi čuva vreme do isteka roka relativno u odnosu na tekući trenutak.
Novi spreman proces se ubacuje u obe strukture, u prvu po prioritetu, u drugu se smešta hronološki, prema vremenu koje ograničava njegovo čekanje, kako je opisano. Za izvršavanje se bira proces sa početka druge strukture, ako je njegovo vreme isteklo (tj. ako je u njemu vrednost 0 za vreme). U suprotnom, bira se najprioritetniji proces iz prve strukture (po prioritetu). Na svaku periodu vremena, smanjuje se brojač prvog elementa druge strukture, sve dok ne dođe do 0 (istekao mu je rok čekanja, pa će biti odmah sledeći na redu).

30. (ispit 2010) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) sa sledećim parametrima:

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*LowPriority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši prioritet, a LP najniži.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 2, za MP je RR sa vremenskim kvantumom 4, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red HP. Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje.

Posmatra se proces P koji ima sledeću karaktekteristiku izvršavanja (R*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava čekanje na I/O operaciju):
R3, B, R5, B, R11, B, R1, B, R3
Napisati sekvencu koja označava redove spremnih procesa u koji se redom smešta P, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Odgovor:

HP, MP, HP, MP, HP, MP, LP, HP, HP, MP

31. (ispit 2010) Raspoređivanje procesa

U nekom sistemu primenjuje se SJF (*shortest job first*) raspoređivanje procesa, uz približnu procenu trajanja narednog naleta izvršavanja eksponencijalnim usrednjavanjem. Procenjeno trajanje naleta izvršavanja je uvek ceo broj u opsegu 0..*T*-1. Razmatraju se tri različite implementacije skupa spremnih procesa:
A) Dvostruko ulančana, neuređena lista.
B) Dvostruko ulančana lista uređena po rasrućem redosledu procenjenog trajanja narednog naleta izvršavanja.
C) Niz od *T* ulaza, u svakom ulazu broj *t* je dvostruko ulančana lista (preciznije, glava i rep liste) spremnih procesa sa procenjenim narednim naletom izvršavanja jednakim *t*.
Za svaku od ovih realizacija u donjoj tabeli dati kompleksnost odgovarajuće operacije u funkciji broja spremnih procesa *n*.

Operacija	A	B	C
Dodavanje novog spremnog procesa u skup spremnih			
Uzimanje procesa za izvršavanje (sa njegovim izbacivanjem iz skupa spremnih)			
Dinamička promena procene trajanja narednog naleta izvršavanja procesa koji je u skupu spremnih			

Odgovor:

Operacija	A	B	C
Dodavanje novog spremnog procesa u skup spremnih	$O(1)$	$O(n)$	$O(1)$
Uzimanje procesa za izvršavanje (sa njegovim izbacivanjem iz skupa spremnih)	$O(n)$	$O(1)$	$O(1)$
Dinamička promena procene trajanja narednog naleta izvršavanja procesa koji je u skupu spremnih	$O(1)$	$O(n)$	$O(1)$

32. (ispit 2010) Raspoređivanje procesa

U nekom trenutku u redu spremnih nalaze se redom sledeći procesi:

Proces	Prioritet (niži broj - viši prioritet)	Procenjena dužina narednog naleta izvršavanja
A	1	3
B	3	2
C	0	5
D	2	4

Ako se nadalje ne pojavljuju novi spremni procesi, a svaki od ovih nakon prvog naleta izvršavanja odlazi u stanje suspenzije, navesti redosled kojim će se izvršavati naleti ovih procesa, ukoliko je algoritam raspoređivanja:

a)(3) *Priority Scheduling*: _____

b)(4) SJF: _____

c)(3) FCFS: _____

a)(3) C, A, D, B b)(4) B, A, D, C c)(3) A, B, C, D

33. (ispit 2010) Raspoređivanje procesa

U nekom sistemu primenjuje se raspoređivanje procesa po prioritetima (*priority scheduling*). Opseg prioriteta je 0..*P*-1. Razmatraju se tri različite implementacije skupa spremnih procesa:

A) Dvostruko ulančana, neuređena lista.

B) Dvostruko ulančana lista uređena po prioritetu procesa.

C) Niz od *P* ulaza, u svakom ulazu broj *k* je dvostruko ulančana lista (preciznije, glava i rep liste) spremnih procesa sa tekućim prioritetom jednakim *k*.

Za svaku od ovih realizacija u donjoj tabeli dati kompleksnost odgovarajuće operacije u funkciji broja spremnih procesa *n*.

Operacija	A	B	C
Dodavanje novog spremnog procesa u skup spremnih			
Uzimanje procesa za izvršavanje (sa njegovim izbacivanjem iz skupa spremnih)			
Dinamička promena prioriteta procesa koji je u skupu spremnih			

Odgovor**:**

Operacija	A	B	C
Dodavanje novog spremnog procesa u skup spremnih	$O(1)$	$O(n)$	$O(1)$
Uzimanje procesa za izvršavanje (sa njegovim izbacivanjem iz skupa spremnih)	$O(n)$	$O(1)$	$O(1)$
Dinamička promena prioriteta procesa koji je u skupu spremnih	$O(1)$	$O(n)$	$O(1)$

34. (ispit 2011) Raspoređivanje procesa

U nekom *preemptive time-sharing* operativnom sistemu, u jednom trenutku označenom sa 0, aktivna su sledeća tri procesa sa sledećim vrednostima preostalog vremena naleta CPU izvršavanja (*CPU burst*) i nalaze se u redu spremnih u navedenom redosledu svoje aktivacije:
P1 - 4, P2 - 8, P3 – 6.
U trenutku 8 aktivira se još i proces P4 čiji je nalet izvršavanja 4 jedinice vremena. On se smešta na kraj reda spremnih odmah iza procesa koji je u tom istom trenutku upravo izgubio procesor zbog isteka vremenskog kvantuma. Vremenski kvantum koji se dodeljuje procesima za izvršavanje iznosi 2 jedinice vremena.
Napisati kojim redosledom će se izvršavati ovi procesi za dati algoritam raspoređivanja. U odgovoru navesti samo sekvencu oznaka procesa (npr.: P1, P2, P3, P4 itd.), pri čemu svaki element u sekvenci označava izvršavanje navedenog procesa u trajanju jednog vremenskog kvantuma.

a) *Round-Robin*

Odgovor: P1, P2, P3, P1, P2, P3, P4, P2, P3, P4, P2

b) *Shortest Job First*

Odgovor: P1, P1, P3, P3, P3, P4, P4, P2, P2, P2, P2

35. (ispit 2011) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS), uz dodatni mehanizam koji ga približava ponašanju SJF algoritma. Svakom procesu pridružena je procena trajanja narednog naleta izvršavanja τ koja se izračunava eksponencijalnim usrednjavanjem sa α=1/2, uz odsecanje na ceo broj.

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*Low Priority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 4, za MP je RR sa vremenskim kvantumom 8, a za LP je FCFS.
- Novoaktivirani proces (deblokiran ili kreiran) smešta se u red prema proceni τ: ako je $\tau \leq 4$, smešta se u HP, ako je $4 < \tau \leq 8$, smešta se u MP, inače se smešta u LP.
- Proces kome je istekao vremenski kvantum premešta se u red nižeg prioriteta od onog iz koga je uzet na izvršavanje. Procena τ se ažurira kada se završi ceo nalet izvršavanja, odnosno kada se proces blokira.

Posmatra se novoaktivirani proces P sa inicijalnom procenom τ=6 koji ima sledeću karaktekteristiku narednog izvršavanja (R*n* označava jedan nalet izvršavanja u trajanju *n* jedinica vremena, B označava čekanje na I/O operaciju):
R3, B, R5, B, R13, B, R1, B, R3
Napisati sekvencu koja označava redove spremnih procesa u kojima redom boravi P, tako da za svako smeštanje procesa P u neki red u sekvenci postoji jedan element. Na primer, odgovor može da bude u obliku: HP, MP, LP, HP, ...

Odgovor: MP, HP, MP, HP, MP, LP, MP, HP

36. (ispit 2011) Raspoređivanje procesa

U nekom sistemu koristi se *Multilevel Feedback-Queue Scheduling* (MFQS) sa sledećim parametrima.

- Postoje tri reda spremnih procesa: HP (*High Priority*), MP (*Medium Priority*) i LP (*Low Priority*).
- Globalni algoritam raspoređivanja je po prioritetu, s tim da HP ima najviši, a LP najniži prioritet.
- Raspoređivanje po redovima je sledeće: za HP je *Round-Robin* (RR) sa vremenskim kvantumom 4, za MP je RR sa vremenskim kvantumom 8, a za LP je FCFS.
- Novoaktivirani proces (kreiran ili deblokiran) smešta se na kraj reda HP.
- Proces kome je procesor preotet jer je aktiviran neki prioritetniji proces smešta se na kraj istog reda iz koga je bio uzet na izvršavanje.
- Proces kome je istekao vremenski kvantum premešta se na kraj reda prvog nižeg prioriteta od onog iz koga je bio uzet na izvršavanje.

U strukturi PCB procesa postoji polje u kome se čuva informacija u kom redu se taj proces nalazi ili je poslednji put bio. Ovu informaciju održavaju dole date operacije `getQueue` i `setQueue` klase `Process`. Date su interfejsi klasa koje su na raspolaganju:

```
enum PriorityQueue { LP, MP, HP };
class Process {
public:
  PriorityQueue getQueue ();
  void setQueue (PriorityQueue);
  int getTimeSlice(); // Returns the assigned time slice
  void setTimeSlice(int); // 0 for no time limit
};

class ProcessQueue {
public:
  void put (Process*);
  Process* get(); // Returns 0 if emtpy
};
```

Klasa `Process` apstrahuje proces, odnosno njegov PCB. Klasa `ProcessQueue` implementira jednsotavni FIFO red procesa.

Korišćenjem ovih klasa, u potpunosti realizovati klasu `Scheduler` čiji je interfejs dat, a čije operacije jezgro poziva u odgovarajućim trenucima promene stanja procesa (kreiranje, deblokiranje, istek vremenskog kvantuma, preuzimanje).

```
class Scheduler {
public:
  Scheduler ();
  Process* getProcessToRun(); // Returns process to run, with time slice
set
  void putActivatedProcess (Process*);  // New ready process
  void putPreemptedProcess (Process*);  // Preemted by a new activated proc
  void putExpiredProcess (Process*); // Proc. with expired time slice
};
```

Rešenje:

```
class Scheduler {
public:
  Scheduler () {}
  Process* getProcessToRun(); // Returns process to run, with time slice
set
  void putActivatedProcess (Process*);  // New ready process
  void putPreemptedProcess (Process*);  // Preemted by a new activated proc
  void putExpiredProcess (Process*); // Proc. with expired time slice
private:
  ProcessQueue que[3];
};

Process* Scheduler::getProcessToRun() {
  Process* p = que[HP].get();
  if (p) {
    p->setTimeSlice(4);
    return p;
  }
  p = que[MP].get();
  if (p) {
    p->setTimeSlice(8);
    return p;
  }
  p = que[LP].get();
  if (p) p->setTimeSlice(0);
  return p;
}

void Scheduler::putActivatedProcess (Process* p) {
  if (p==0) return;
  p->setQueue(HP);
  que[HP].put(p);
}

void Scheduler::putPreemtedProcess (Process* p) {
  if (p==0) return;
  PriorityQueue q = p->getQueue();
  que[q].put(p);
}

void Scheduler::putExpiredProcess (Process* p) {
  if (p==0) return;
  PriorityQueue q = p->getQueue();
  if (q>0) q--;
  p->setQueue(q);
  que[q].put(p);
}
```

37. (ispit 2011) Raspoređivanje procesa

U redu spremnih procesa u nekom početnom trenutku nalaze se sledeći procesi sa datim vremenima trajanja narednog naleta izvršavanja (*CPU burst*):
P1 (3), P2 (6), P3 (1), P4 (4)
Raspoređivanje procesa je sa preotimanjem (*preemptive*) i raspodelom vremena (*time sharing*), sa vremenskim kvantumom jednakim 2.
Napisati redosled izvršavanja procesa (za svaku dodelu procesora procesu napisati naziv procesa, npr. P1, P2, P2, P3, P3,P2, ...), ako je algoritam raspoređivanja:

a) Tačni SJF.
Odgovor: P3, P1, P1, P4, P4, P2, P2, P2

a) Aproksimacija SJF sa eksponencijalnim usrednjavanjem (α=1/2, odsecanje na ceo broj) i ako su vrednosti τ u datom početnom stanju za date procese: τ(P1)=4, τ(P2)=3, τ(P3)=2, τ(P4)=1.
Odgovor P4, P4, P3, P2, P2, P2, P1, P1

38. (ispit 2011) Raspoređivanje procesa

U nekom operativnom sistemu primenjuje se raspoređivanje procesa sa prioritetima (*Priority Scheduling*), uz starenje (*aging*) radi sprečavanja izgladnjivanja: svaki put kada se neki spreman proces izabere za izvršavanje, ostali u listi spremnih koji još čekaju dobijaju za jedan viši nivo prioriteta. Red spremnih procesa implementiran je kao jednostruko ulančana lista PCB struktura uređenih po prioritetu:

```
typedef unsigned short Priority; // Priority: 0 is the lowest
const Priority maxPri = ...; // Max value of priority

struct PCB {
  Priority curPri; // Current priority
  Priority defPri; // Default priority (set on creation)
  PCB* next; // Next PCB in ready list
  ...
};

PCB* ready;  //  Head of ready list
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list"
void putReady (PCB*); // Puts a process to the „ready list"
```

Rešenje:

```
PCB* getReady () {
  PCB* selProc = ready;
  if (selProc==0) return 0;
  ready = selProc->next;
  selProc->next = 0;

  for (PCB* p=ready; p!=0; p=p->next)
    if (p->curPri<maxPri) p->curPri++;
  return selProc;
}

void putReady (PCB* p) {
  if (p==0) return;
  p->curPri=p->defPri;
  if (ready==0 || ready->curPri<p->curPri) {
    p->next=ready;
    ready=p;
    return;
  }
  PCB* cur=ready;
  while (cur->next!=0 && cur->next->curPri>=p->curPri) cur=cur->next;
  p->next=cur->next;
  cur->next=p;
}
```

39. (ispit 2011) Raspoređivanje procesa

U nekom operativnom sistemu primenjuje se aproksimacija SJF algoritma raspoređivanja procesa sa eksponencijalnim usrednjavanjem. Red spremnih procesa implementiran je kao jednostruko ulančana lista PCB struktura uređenih po vrednosti procene narednog naleta izvršavanja τ sa α=1/2. U polju `tau` strukture PCB čuva se izračunata vrednost ove procene. Nju treba izračunati prilikom smeštanja PCB procesa u red spremnih, na osnovu vrednosti polja `lastCPUBurst` koje je postavljeno na trajanje poslednjeg naleta izvršavanja kada je sistem procesu oduzeo procesor.

```
typedef unsigned int CPUTime;

struct PCB {
  CPUTime tau, lastCPUBurst;
  PCB* next; // Next PCB in ready list
  ...
};

PCB* ready;  //  Head of ready list
```

Realizovati operacije raspoređivača:

```
PCB* getReady (); // Returns the selected process from the „ready list"
void putReady (PCB*); // Puts a process to the „ready list"
```

Rešenje:

```
PCB* getReady () {
  PCB* selProc = ready;
  if (selProc==0) return 0;
  ready = selProc->next;
  selProc->next = 0;
  return selProc;
}

void putReady (PCB* p) {
  if (p==0) return;
  p->tau=(p->tau+p->lastCPUBurst)/2;
  if (ready==0 || ready->tau>p->tau) {
    p->next=ready;
    ready=p;
    return;
  }
  PCB* cur=ready;
  while (cur->next!=0 && cur->next->tau<=p->tau) cur=cur->next;
  p->next=cur->next;
  cur->next=p;
}
```

1. (Novembar 2006) Komunikacija i sinhronizacija pomoću deljene promenljive

Projektuje se konkurentni klijent/server sistem. Server treba modelovati monitorom. Klijenti su procesi koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (*token*). Kada dobije žeton, klijent započinje aktivnost. Po završetku aktivnosti, klijent vraća žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Prikazati rešenje korišćenjem klasičnih monitora i uslovnih promenljivih. Napisati kod monitora i procesa-klijenta.

Rešenje:

```
monitor server;
export acquireToken, returnToken;

var numOfTokens : integer;
    tokenAvailable : condition;

procedure acquireToken ();
begin
  if (numOfTokens<=0) tokenAvailable.wait;
  numOfTokens := numOfTokens - 1;
end;

procedure returnToken ();
begin
  numOfTokens := numOfTokens + 1;
  tokenAvailable.signal;
end;

begin
  numOfTokens := N;
end; (* server *)


task type client;
begin
  loop
    server.acquireToken;
    do_some_activity;
    server.returnToken;
  end;
end; (* client *)
```

2. (Novembar 2007) Komunikacija i sinhronizacija pomoću deljene promenljive

Korišćenjem klasičnih monitora i uslovnih promenljivih, realizovati monitor koji ima privatnu celobrojnu promenljivu inicijalizovanu na vrednost $N>0$ i dve operacije *wait* i *signal* sa semantikom standardnih brojačkih semafora.

Rešenje:

```
monitor semaphore;
export wait, signal;

var value : integer;
    queue : condition;

procedure wait ();
begin
  if (value=0) queue.wait;
  value := value - 1;
end;

procedure signal ();
begin
  value := value + 1;
  queue.signal;
end;

begin
  value := N;
end; (* semaphore *)
```

3. (Novembar 2008) Komunikacija i sinhronizacija pomoću deljene promenljive

Korišćenjem klasičnih monitora i uslovnih promenljivih, realizovati monitor `forksAgent` koji upravlja viljuškama kao deljenim resursima u problemu filozofa koji večeraju (*dining philosophers*). Filozof postupa tako što odjednom uzima obe svoje viljuške, ako su slobodne:

```
type PhilospherID : integer 0..4;

monitor forksAgent;
  export takeForks, releaseForks;
  procedure takeForks (id : PhilosopherID);
  procedure releaseForks (id : PhilosopherID);
end;

process type Philosopher (myID : PhilosopherID) begin
  loop
    think;
    forksAgent.takeForks(myID);
    eat;
    forksAgent.releaseForks(myID);
  end;
end;
```

Nije potrebno rešavati problem izgladnjivanja koji ovde postoji.

Rešenje:

```
monitor forksAgent;
  exports takeForks, releaseForks;

  var forks : array[PhilosopherID] of 0..2;
      barriers : array[PhilosopherID] of condition;

  procedure takeForks (id : PhilosopherID);
  var left, right : PhilospherID;
  begin
    if id==0 then left:=4 else left:=id-1;
    if id==4 then right:=0 else right:=id+1;
    if forks[id]<2 then
      barriers[id].wait;
    end if
    forks[id]:=0;
    forks[left]:=forks[left]-1;
    forks[right]:=forks[right]-1;
  end;

  procedure releaseForks (id : PhilosopherID);
  var left, right : PhilospherID;
  begin
    if id==0 then left:=4 else left:=id-1;
    if id==4 then right:=0 else right:=id+1;
    forks[id]:=2;
    forks[left]:=forks[left]+1;
    forks[right]:=forks[right]+1;
    if forks[left]==2 barriers[left].signal;
    if forks[right]==2 barriers[right].signal;
  end;

begin
  var i : PhilosopherID;
  for i:=0 to 4 do forks[i]:=2;
end;
```

4. (Oktobar 2009) Komunikacija i sinhronizacija pomoću deljene promenljive

(Malo drugačiji problem filozofa) Iscrpljeni napornim razmišljanjem i stalnim otimanjem za viljuške, dvojica od naših pet filozofa su digli ruke i napustili seansu. Osim toga, bogati mecena je donirao još jednu, šestu viljušku. Ozareni ovim događajima, preostala trojica filozofa su nastavili da učestvuju u seansi za okruglim stolom, oslobođeni brige o viljuškama, pošto sada svaki od njih trojice ima samo svoje sopstvene dve viljuške kojima se služi pri jelu. Ali, avaj, njihovim problemima nije došao kraj! Naime, svaki od njih trojice voli da jede špagete koristeći tačno dva od tri začina: prvi filozof u špagete dodaje so i biber, drugi biber i kečap, a treći so i kečap. Na stolu se nalazi tačno tri posude sa začinima: jedna u kojoj je so, druga u kojoj je biber i treća u kojoj je kečap. Robujući svojim malim ritualima, ni jedan filozof ne može da počne da jede špagete dok ne uzme svoja dva omiljena začina. Dakle, pre nego što počne da jede, svaki filozof uzima dve posude sa svoja dva začina, posluži se i vraća posude na sto.

Korišćenjem klasičnih uslovnih promenljivih, realizovati monitor koji obezbeđuje potrebnu sinhronizaciju i prikazati jedan od tri procesa koji predstavlja ponašanje filozofa. Rešenje ne mora da reši potencijalno izgladnjivanje.

Rešenje:

```
monitor Agent;
  export takeSaltAndPepper,
         putSaltAndPepper,
         takePepperAndKetchup,
         putPepperAndKetchup,
         takeKetchupAndSalt,
         putKetchupAndSalt;
  var
    saltAvailable : boolean;
    pepperAvailable : boolean;
    ketchupAvailable : boolean;
    waitSaltAndPepper : condition;
    waitPepperAndKetchup : condition;
    waitKetchupAndSalt : condition;

  procedure takeSaltAndPepper ();
  begin
    while not (saltAvailable and pepperAvailable) do
      wait(waitSaltAndPepper);
    saltAvailable := false;
    pepperAvailable := false;
  end;

  procedure putSaltAndPepper ();
  begin
    saltAvailable := true;
    pepperAvailable := true;
    signal(waitPepperAndKetchup);
    signal(waitKetchupAndSalt);
  end;

  -- etc.
```

```
begin
  saltAvailable:=true;
  pepperAvailable:=true;
  ketchupAvailable:=true;
end;


process PhilisopherThatLikesSaltAndPepper;
begin
  loop
    think;
    Agent.takeSaltAndPepper();
    spiceItUp;
    Agent.putSaltAndPepper();
    eat;
  end
end;
```

5. (Oktobar 2010) Međuprocesna komunikacija pomoću deljene promenljive

Jedna varijanta uslovne sinhronizacije unutar monitora je sledeća. Svaki monitor ima samo jednu, implicitno definisanu i anonimnu (bez imena) uslovnu promenljivu, tako da se u monitoru mogu pozivati sledeće dve sinhronizacione operacije:

- `wait()`: bezuslovno blokira pozivajući proces i oslobađa ulaz u monitor;
- `notifyAll()`: deblokira sve procese koji su čekali na uslovnoj promenljivoj, s tim da ih pušta da nastavljaju izvršavanje svoje procedure jedan po jedan.

Implementirati ograničeni bafer korišćenjem ove varijante uslovne sinhronizacije.

Rešenje:

```
monitor buffer;
  export append, take;
  var
    buf : array[0..size-1] of integer;
    top, base : 0..size-1;
    numberInBuffer : integer;

  procedure append (i : integer);
  begin
    while numberInBuffer = size do
      wait();
    end while;
    buf[top] := i;
    numberInBuffer := numberInBuffer+1;
    top := (top+1) mod size;
    notifyAll();
  end append;

  procedure take (var i : integer);
  begin
    while numberInBuffer = 0 do
      wait();
    end while;
    i := buf[base];
    base := (base+1) mod size;
    numberInBuffer := numberInBuffer-1;
    notifyAll();
  end take;

begin (* Initialization *)
  numberInBuffer := 0;
  top := 0; base := 0
end;
```

6. (Oktobar 2011) Međuprocesna komunikacija pomoću deljene promenljive

Jedna varijanta uslovne sinhronizacije unutar monitora je sledeća. Svaki monitor ima samo jednu, implicitno definisanu i anonimnu (bez imena) uslovnu promenljivu, tako da se u monitoru mogu pozivati sledeće dve sinhronizacione operacije:

- wait() : bezuslovno blokira pozivajući proces i oslobađa ulaz u monitor;
- notify() : deblokira jedan proces koji čeka na uslovnoj promenljivoj, ako takvog ima.

Projektuje se konkurentni klijent-server sistem. Server treba modelovati monitorom sa opisanom uslovnom promenljivom. Klijenti su procesi koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (token). Kada dobije žeton, klijent započinje aktivnost.
Po završetku aktivnosti, klijent vraća žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Napisati kod monitora i procesa-klijenta.

Rešenje:

```
monitor server;
export acquireToken, returnToken;

var numOfTokens : integer;

procedure acquireToken ();
begin
      if numOfTokens<=0 then wait();
      numOfTokens := numOfTokens - 1;
end;

procedure returnToken ();
begin
      numOfTokens := numOfTokens + 1;
      notify();
end;

begin
      numOfTokens := N;
end; (* server *)

task type client;
begin
      loop
            server.acquireToken;
            do_some_activity;
            server.returnToken;
      end;
end; (* client *)
```

7. (Septembar 2012) Međuprocesna komunikacija pomoću deljene promenljive

Monitor `server`, čiji je interfejs dat dole, služi kao jednoelementni bafer za razmenu podataka između proizvoljno mnogo uporednih procesa proizvođača koji pozivaju operaciju `write` i potrošača koji pozivaju operaciju `read`. Tek kada jedan proizvođač upiše podatak tipa `Data` operacijom `write`, jedan potrošač može da ga pročita operacijom `read`; tek nakon toga neki proizvođač sme da upiše novi podatak, i tako dalje naizmenično. Implementirati monitor `server` sa potrebnom sinhronizacijom korišćenjem standardnih uslovnih promenljivih.

```
type Data = ...;

monitor server;
  export write, read;
  procedure write (data : Data);
  procedure read (var data :
Data);
end;
```

Rešenje:

```
type Data = ...;

monitor server;
  export write, read;

  var d : Data,
      readyToRead, readyToWrite : boolean,
      waitToRead, waitToWrite : condition;

  procedure write (data : Data);
  begin
    while (not readyToWrite) do waitToWrite.wait();
    readyToWrite:=false;
    d:=data;
    readyToRead:=true;
    waitToRead.signal();
  end;

  procedure read (var data : Data);
  begin
    while (not readyToRead) do waitToRead.wait();
    readyToRead:=false;
    data:=d;
    readyToWrite:=true;
    waitToWrite.signal();
  end;

begin
  readyToRead:=false;
  readyToWrite:=true;
end; (* server *)
```

8. (Oktobar 2012) Međuprocesna komunikacija pomoću deljene promenljive

Jedna varijanta uslovne sinhronizacije unutar monitora je sledeća. Svaki monitor ima samo jednu, implicitno definisanu i anonimnu (bez imena) uslovnu promenljivu, tako da se u monitoru mogu pozivati sledeće dve sinhronizacione operacije:

- `wait()`: bezuslovno blokira pozivajući proces i oslobađa ulaz u monitor;
- `notifyAll()`: deblokira *sve* procese koji čekaju na uslovnoj promenljivoj, ako takvih ima (naravno, mešusobno isključenje tih procesa je i dalje obezbeđeno).

Projektuje se konkurentni klijent-server sistem. Server treba modelovati monitorom sa opisanom uslovnom promenljivom. Klijenti su procesi koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (*token*). Kada dobije žeton, klijent započinje aktivnost.
Po završetku aktivnosti, klijent vraća žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Napisati kod monitora i procesa-klijenta.

Rešenje:

```
monitor server;
export acquireToken, returnToken;

var numOfTokens : integer;

procedure  acquireToken  ();
begin
  while numOfTokens<=0 do wait();
  numOfTokens := numOfTokens - 1;
end;

procedure  returnToken  ();
begin
  numOfTokens  :=  numOfTokens  +  1;
  notifyAll();
end;

begin
  numOfTokens  :=  N;
end; (* server *)


task  type  client;
begin
  loop
    server.acquireToken;
    do_some_activity;
    server.returnToken;
  end;
end; (* client *)
```

9. (Septembar 2013) Međuprocesna komunikacija pomoću deljene promenljive

Korišćenjem klasičnih uslovnih promenljivih, napisati kod za monitor `account` koji realizuje bankovni račun sa promenljivom stanja na računu (suma novca trenutno raspoloživa na računu). Monitor ima dve operacije u svom interfejsu:

- `credit(amount:real)`: na račun dodaje dati iznos;
- `debit(amount:real)`: sa računa skida dati iznos, ali tek kada na računu ima dovoljno novca (više ili jednako traženom iznosu).

Rešenje:

```
monitor account;
  export credit, debit;

  var balance : real,
      solvent : condition;

  procedure credit (amount : real);
  begin
    balance := balance + amount;
    solvent.signal();
  end;

  procedure debit (amount : real);
  begin
    while (balance<amount) do solvent.wait();
    balance := balance - amount;
  end;

begin
  balance := 0;
end; (* account *)
```

1. (Novembar 2006) Komunikacija pomoću poruka

Implementirati web servis (web service), na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class CoderProxy {
      public String code(String op1, String op2){…}
      public String code(String op){…}
}
```

Na serveru postoji klasa `Coder` koja pruža uslugu. Klasa `Coder` ima sledeći interfejs:

```
public class Coder {
      public String code(String op1, String op2){…}
      public String code(String op){…}
}
```

Korisnik treba da na instancira objekat klase CoderProxy, na svojoj, klijenstkoj strani, koji će predstavljati posrednik (*Proxy*) do stvarnog objekta klase koder, koja će pružati uslugu. Pretpostaviti da nije dozvoljeno kao parametar proslediti string koji sadrži znak '#'. Takođe pretpostaviti da rezultat metoda `Coder.code(...)` ne sadrži znak '#'. Međuprocesnu komunikaciju realizovati preko priključnica (*Socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Rešenje:

```
public class CoderProxy extends Usluga {

      public CoderProxy (String host, int port){
            super(host, port);
      }

      public String code(String op1, String op2){
            String message = "#code1#" + op1 + "#" + op2 + "#";
            sendMessage(message);
            return receiveMessage();
      }

      public String code(String op){
            String message = "#code2#" + op + "#";
            sendMessage(message);
            return receiveMessage();
      }
}

public class RequestHandler extends Thread{
      protected Coder coder;
      ...
      public RequestHandler(...,Coder k){
            ...
            coder = k;
      }
      protected void processRequest(String request){
            StringTokenizer tokenizer = new StringTokenizer(request, "#");
            String functionName = tokenizer.next();
            ...
            }
```

```
            else if (functionName.equals("code1")){
                  String op1 = tokenizer.next();
                  String op2 = tokenizer.next();
                  String result = coder.code(op1,op2);
                  sendMessage(result);
            }
            else if (functionName.equals("code2")){
                  String op = tokenizer.next();
                  String rezultat = coder.code(op);
                  sendMessage(result);
            }

      }
      ...
}
```

2. (Novembar 2007) Komunikacija pomoću poruka

Korišćenjem koncepta priključnica (*socket*), na jeziku Java implementirati program nad kojim se može pokrenuti serverski demonski proces-osluškivač (*listener daemon*) koji prihvata zahteve za komunikacijom od strane klijenata na portu 1050 i za svaki takav primljeni zahtev sprovodi sledeći postupak:

- odabere jedan trenutno „slobodan“ port iz opsega 1051..1060; ovaj proces vodi evidenciju o „zauzetosti“ portova u ovom opsegu kako je opisano u nastavku;
- ukoliko nema takvog slobodnog porta, klijentu vraća znak ‘0’ i raskida komunikaciju sa klijentom;
- ukoliko nađe takav port, „zauzima“ ga kao novi „kanal“ komunikacije sa klijentom i njegov broj, pretvoren u niz znakova, vraća klijentu;
- kreira novu nit koja će na novozauzetom portu sačekati uspostavljanje komunikacije sa klijentom, zatim vratiti klijentu znak ‘0’ preko tog porta, raskinuti komunikaciju sa klijentom, i konačno „osloboditi“ taj port za ponovno korišćenje.

Rešenje:

```
import java.net.*;
import java.io.*;

private class ChannelServer extends Thread {
  private ServerSocket mySocket;
  private int myPort;

  public ChannelServer (int port) {
    myPort = port;
    mySocket = new ServerSocket(port);
  }

  public void run () {
    try {
      Socket client = mySocket.accept();
      PrintWriter pout = new PrintWriter(client.getOutputStream(),true);
      pout.println("0");
      pout.close();
      client.close();
      MainServer.freePort(myPort);
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
```

```
}

public class MainServer {
  private final static int N = 10;
  private final static int port0 = 1050;
  private static bool[] allocatedPorts = new bool[N];

  private static int getFreePort () {
    for (int i=0; i<N; i++) {
      if (allocatedPorts[i]) continue;
      allocatedPorts[i]=true;
      return port0+i-1;
    }
    return 0;
  }

  public static void freePort (int i) {
    i = i - port0 - 1;
    if (i>=0 && i < N)
      allocatedPorts[i]=false;
  }

  public static void main (String[] args) {
    try {
      ServerSocket sock = new ServerSocket(port0);
      while (true) {
        Socket client = sock.accept();
        int newPort = getFreePort();
        if (newPort > 0)
          new ChannelServer(newPort).start();
        PrintWriter pout = new PrintWriter(client.getOutputStream(),true);
        pout.println(new Integer(newPort).toString());
        pout.close();
        client.close();
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

3. (Novembar 2008) Komunikacija pomoću poruka

Korišćenjem koncepta priključnica (*socket*), na jeziku Java implementirati program nad kojim se može pokrenuti serverski demonski proces-osluškivač (*listener daemon*) koji prihvata zahteve za komunikacijom od strane klijenata na portu 1050 i za svaki takav primljeni zahtev sprovodi sledeći postupak:

- pošalje „signal“ potvrde za uspostavljanjem komunikacije slanjem niza znakova „ack“;
- očekuje novu poruku na istom portu sa sadržajem „ping“;
- ako dobije ovu poruku, ponovo šalje samo „ok“ i raskida vezu sa klijentom;
- ako dobije drugačiji sadržaj poruke, šalje klijentu poruku „repeat“ i ponovo čeka sve dok ne dobije poruku „ping“, kada postupa kao u prethodnom slučaju.

Rešenje:

```
import java.net.*;
import java.io.*;

public class MainServer {
  private final static int port0 = 1050;

  public static void main (String[] args) {
    try {
      ServerSocket sock = new ServerSocket(port0);
      while (true) {
        Socket client = sock.accept();
        PrintWriter pout = new PrintWriter(client.getOutputStream(),true);
        BufferedReader pin = new BufferedReader(new InputStreamReader(
                                                sock.getInputStream()));
        pout.println("ack");
        while (!pin.readLine().equals("ping")) pout.println("repeat");
        pout.println("ok");
        pin.close;
        pout.close();
        client.close();
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

4. (Oktobar 2009) Komunikacija pomoću poruka

Između gradova A i B postoji samo jedna pruga po kojoj vozovi mogu da se kreću u oba smera. Na pruzi se nikada ne smeju naći vozovi koji se istovremeno kreću u različitim smerovima. U svakom trenutku na pruzi može biti i više vozova koji se kreću u istom smeru. Kada voz treba da krene iz jednog od ova dva mesta, mašinovođa je dužan da uspostavi komunikaciju sa dispečerom koji se nalazi u gradu C i dalje sluša njegove komande. Dispečer je osoba koja vodi računa da ne dođe do sudara vozova na ovoj pruzi. Dispečer komunicira sa svim mašinovođama od trenutka kada žele da krenu do trenutka kada stignu na odredište i sprečava sudar tako što određuje kada koji voz može da krene. Dispečer se trudi da postigne maksimalno iskorišćenje pruge, ali pre svega se trudi da prema svima bude fer pa dozvole za polazak daje u istom redosledu u kojem su stigli zahtevi za polaske. Napisati program na programskom jeziku Java koji treba da obavlja posao dispečera. Dispečer očekuje da se svaki mašinovođa javi na port 1050. Prikazati i sekvencu poruka koje jedan mašinovođa razmeni sa dispečerom od trenutka kada uđe u voz do trenutka kada iz njega izađe. Za komunikaciju koristiti priključnice (*socket*) i prosleđivanja poruka (*message passing*), a za sinhronizaciju sinhrone blokove i metode (*synchronized*).

Rešenje:

```
//Beleznica
public class Beleznica {
    private int redni_broj;
    private int opsluzuje_se;
    private int vozAB;
    private int vozBA;

    public Beleznica(){
        redni_broj = 0;
        opsluzuje_se = 0;
        vozAB = 0;
        vozBA = 0;
    }
    public void kreniAB(){
        vozAB++;
        opsluzuje_se++;
    }

    public boolean stigaoAB(){
        return --vozAB == 0;
    }

    public boolean zauzetaAB(){
        return vozAB>0;
    }

    public void kreniBA(){
        vozBA++;
        opsluzuje_se++;
    }

    public boolean stigaoBA(){
        return --vozBA == 0;
    }

    public boolean zauzetaBA(){
        return vozBA>0;
    }
```

```
    public int uzmi_redni_broj(){
        return redni_broj++;
    }

    public boolean dosao_na_red(int i){
        return i == opsluzuje_se;
    }
}

//Server_voz
public class Server_voz extends Thread {
    private Socket s;

    public Server_voz(Socket soc) {
        s = soc;
    }

    public void run(){
    BufferedReader r = null;
    BufferedWriter w = null;
        try {
            r = new BufferedReader(new
InputStreamReader(s.getInputStream()));
            w = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
            String poruka = r.readLine();
            if (r.equals("startA")){
                synchronized(Dispecer.beleznica){
                    int redni_broj = Dispecer.beleznica.uzmi_redni_broj();
                    while(!Dispecer.beleznica.dosao_na_red(redni_broj) &&
Dispecer.beleznica.zauzetaBA()){
                        o.wait();
                    }
                    Dispecer.beleznica.kreniAB();
                    Dispecer.beleznica.notifyAll();
                }
                w.write("kreni");
                w.flush();
                poruka = r.readLine();
                synchronized(Dispecer.beleznica){
                    if (Dispecer.beleznica.stigaoAB() ){
                        Dispecer.beleznica.notifyAll();
                    }
                }
            }else if (r.equals("startB")){
                //analogno prethodnom, AB zameniti sa BA, BA zameniti sa AB
            }else{
                //greska
            }
            r.close();
            w.close();
            s.close();
        } catch (Exception ex) {
            ex.printStackTrace();
            try {
                r.close();
                w.close();
                s.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}

//Dispecer
public class Dispecer {
    public Dispecer() {
    }

    public static Beleznica beleznica = new Beleznica();

    private static ServerSocket s;
    public static void main(String[] args) {
        try {
            s = new ServerSocket(1050);
            while(true){
                Socket cs = s.accept();
                Server_voz w = new Server_voz(cs);
                w.start();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
            try {
                s.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

//masinovodja koji krece iz mesta A
salje poruku "startA"
ceka da primi poruku "kreni"
//putuje
salje poruku "stigao"
```

5. (Oktobar 2010) Međuprocesna komunikacija razmenom poruka

Napisati program na programskom jeziku Java koji udaljenim računarima obezbeđuje operaciju `int fetch_and_increment()` nad jednom deljenom promenljivom. Početna vrednost promenljive je 0. Svaki poziv ove operacije atomično treba da dohvati zatečenu vrednost deljene promenljive i uveća deljenu promenljivu za 1. Prikazati i klasu koja je na raspolaganju korisniku i koja sadrži traženu operaciju. Korisnik jednom instancira objekat ove klase i kasnije više puta poziva traženu operaciju. Za komunikaciju koristiti priključnice (*Socket*) i mehanizam prosleđivanja poruka (*message passing*). Za sinhronizaciju koristiti klasu Semaphore koja obezbjeđuje standardni interfejs brojačkih semafora.

Rešenje:

```
public class Main {
    public Main() {
    }

    public static void main(String[] args) {
        ServerSocket ss;
        try{
            ss = new ServerSocket(5000);
            while(true){
                Socket s = ss.accept();
                (new RequestHandler(s)).start();
            }
        }catch(Exception e){
            //greska
        }
    }
}

public class RequestHandler extends Thread {
    private PrintWriter out;
    private BufferedReader in;
    private Socket s;

    public RequestHandler(Socket s) {
        this.s = s;
        try{
            out = new PrintWriter(s.getOutputStream(),true);
            in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        }catch(Exception e){
            //greska
        }
    }

    public void run(){
        try{
            while(!s.isInputShutdown()){
                String s = in.readLine();
                System.out.println(s);
                if (s.equals("#fetch_and_increment#")){
                    int i = fetch_and_increment();
                    out.println(i);
                }
            }
        }catch(Exception e){
            //greska
        }
```

```
    }

    private static int deljena_promenljiva = 0;
    private static Semaphore mutex = new Semaphore(1);
    private static int fetch_and_increment() {
        int i;
        try {
            mutex.acquire();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            return -1;
        }
        i = deljena_promenljiva++;
        mutex.release();
        return i;
    }

}
Klijent:
public class fetch_and_increment {
    Socket s;
    PrintWriter out;
    BufferedReader in;

    public fetch_and_increment(String host, int port) {
        try {
            s = new Socket(host,port);
            out = new PrintWriter(s.getOutputStream(),true);
            in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        } catch (Exception e) {
            //greska
        }

    }

    public int fetch_and_increment1(){
        try{
            out.println("#fetch_and_increment#");
            return Integer.parseInt(in.readLine());
        }catch (Exception e){
            //greska
        }
        return -1;
    }

}
```

6. (Ispit Januar 2011) Sinhronizacija i komunikacija između procesa

Napisati program na programskom jeziku Java koji obezbeđuje sinhronizaciju prolaska vozila kroz pametnu raskrsnicu. Vozilima je na raskrsnici zabranjeno skretanje, tako da je potrebno obezbediti samo da se kroz raskrsnicu u svakom trenutku vozila kreću u samo jednom od dva moguća pravca. Vozila dolaze sa severa, juga, istoka i zapada. Napisati i odgovarajuće klase koje se koriste u vozilima (svako vozilo ima svoj računar). Pri pisanju koda ne treba rešavati potencijalni problem izgladnjivanja, već samo maksimizovati protok vozila kroz raskrsnicu. Program pametne raskrsnice se pokreće na računaru čija je adresa "pametnaRaskrsnica" i na raspolaganju ima port 6000.

Rešenje:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Komunikator extends Thread {
    private Socket s;
    private PrintWriter out;
    private BufferedReader in;

    public Komunikator(Socket s) {
        this.s = s;
        try {
            out = new PrintWriter(s.getOutputStream(),true);
            in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void run(){
        while(s.isConnected()){
            String zahtev, odgovor;
            try {
                zahtev = in.readLine();
                int i = 0;
                i = Integer.parseInt(zahtev);
                switch (i){
                    case 1:case 3: Main.syn.izStart();
                                   out.println("ok");
                                   odgovor = in.readLine();
                                   Main.syn.izEnd();
                                   out.println("ok");
                                   break;
                    case 2:case 4: Main.syn.sjStart();
                                   out.println("ok");
                                   odgovor = in.readLine();
                                   Main.syn.sjEnd();
                                   out.println("ok");
                                   break;
                }

            } catch (IOException ex) {
                ex.printStackTrace();
            }

        }
```

```
    }
}
//Main.java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Main {

    public Main() {
        iz = 0;
        sj = 0;
    }

    public synchronized void izStart(){
        try {
            while (sj>0)
                wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        iz++;
        return;
    }

    public synchronized void izEnd(){
        iz--;
        if (iz == 0) notifyAll();
        return;
    }

    public synchronized void sjStart(){
        try {
            while (iz>0)
                wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        sj++;
        return;
    }

    public synchronized void sjEnd(){
        sj--;
        if (sj == 0) notifyAll();
        return;
    }

    public static Main syn = new Main();

    private int iz;
    private int sj;

    public static void main(String[] args) {
        ServerSocket ss;
        try {
            ss = new ServerSocket(6000);
            while(true){
                Socket s = ss.accept();
                (new Komunikator(s)).start();
            }

        } catch (IOException ex) {
```

```
            ex.printStackTrace();
        }

    }
}
```

Sinhronizacija automobila je slična sinhronizaciji prikazanoj kod semafora. Vozilo treba da pošalje kod smjera iz kojeg dolazi (1 – sever, 2 – istok, 3 - jug i 4 - zapad) i zatim da saceka odgovor "ok" (time je dobijena dozvola ulaska u raskrsnicu). Kada završi prolazak treba da pošalje signalnu poruku (sadržaj nebitan) i da sačeka odgovor "ok".

7. (Oktobar 2011) Međuprocesna komunikacija razmenom poruka

Na jeziku Java implementirati serverski proces koji predstavlja agenta na aukciji. Ovaj proces treba da „osluškuje“ port 1025 preko koga prima poruku za otvaranje nadmetanja sa početnom
cenom. Zatim, nakon zatvaranja prethodne konekcije, po istom portu počinje da prihvata ponude od ostalih učesnika u nadmetanju, pri čemu pamti trenutno najveću ponudu. U svakoj ponudi učesnik se identifikuje svojom vrednošću priključnice (IP adresa i port računara preko koga prima odgovor). Svaka nova ponuda mora biti veća od prethodne, inače se ponuđaču odmah vraća informacija o odbijanju ponude. U suprotnom se vraća poruka da je ponuda prihvaćena i da se čeka krajnji ishod nadmetanja. U slučaju da u međuvremenu pristigne ponuda sa većom vrednošću, vraća se informacija o odbijanju ponude, a ukoliko među prethodnih 5 ponuda ne stigne ni jedna veća, nadmetanje se zatvara, a procesu koji predstavlja učesnika sa najvišom ponudom šalje se poruka o pobedi. Za sinhronizaciju i komunikaciju koristiti priključnice (sockets) i mehanizam prosleđivanja poruka (message passing).

```
Import java.io.*;
import java.net.*;
import java.util.StringTokenizer;

public class Agent {

static int bestOffer;
static String bestClientHost = null; static int bestClientPort;
static boolean auctionOver = false; static int badOfferCounter = 0;
static BufferedReader in;
static PrintWriter out;

public static void main(String[] args) {
try {
  ServerSocket sock = new ServerSocket(1025);
  Socket clientSocket = sock.accept();
  in = new BufferedReader(new InputStreamReader(
    clientSocket.getInputStream()));
  StringTokenizer st = new StringTokenizer(in.readLine(), "#");
  if (!st.nextToken().equals("StartAuction")) {
  auctionOver = true;
  }
  else {
    bestOffer = Integer.parseInt(st.nextToken());}

  clientSocket.close();

  while (!auctionOver) {
```

```
   clientSocket = sock.accept();

   in = new BufferedReader(new
   InputStreamReader(clientSocket.getInputStream()));
   out = new PrintWriter(clientSocket.getOutputStream(), true); st = new
   StringTokenizer(in.readLine(), "#");
   String clientHost = st.nextToken();
   String clientPort = st.nextToken();
   int newOffer = Integer.parseInt(st.nextToken());

   if (newOffer > bestOffer) {
    if (bestClientHost != null) sendMsgToBestClient("BetterOfferReceived");
    bestOffer = newOffer;
    bestClientHost = clientHost;
    bestClientPort = Integer.parseInt(clientPort);
    out.println("OfferAccepted");
    badOfferCounter = 0;
   } else {
    out.println("OfferRejected"); badOfferCounter++;
   }

   clientSocket.close();

   if (badOfferCounter == 5) {
    if (bestClientHost != null) sendMsgToBestClient("YouWon!"); auctionOver
    = true;
   }
  }
} catch (Exception e) { System.err.println(e);}
}

static void sendMsgToBestClient(String msg) throws UnknownHostException,
IOException {
 Socket clientSocket = new Socket(bestClientHost,bestClientPort);
 PrintWriter oldOut = new PrintWriter(clientSocket.getOutputStream(),true);
 oldOut.println(msg);
 clientSocket.close();
 }
}
```

8. (Septembar 2012) Međuprocesna komunikacija razmenom poruka

Projektuje se konkurentni klijent-server sistem koji koristiti priključnice (engl. sockets) i mehanizam prosleđivanja poruka (engl. message passing). Klijenti su procesi koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (engl. token). Kada dobije žeton, klijent započinje aktivnost. Po završetku aktivnosti, klijent vraća žeton serveru. Server „osluškuje“ port 1033 preko koga prima zahteve i vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Napisati kod procesa-servera i procesa-klijenta. Nije potrebno proveravati uspešnost izvršavanja operacija nad priključnicama.

Rešenje:

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Server {
 static final int N = ...;
 static int count = N;
 static LinkedList<Socket> blockedList = new LinkedList<Socket>();

 public static void main(String[] args) {
  try {
   ServerSocket sock = new ServerSocket(1033);

   while (true) {
    Socket clientSocket = sock.accept();
    BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    String request = in.readLine();

    if (request.equals("GetToken")) {
    if(count>0){
      sendMsgToClient(clientSocket, "Continue");
      count--;
     }
     else blockedList.addLast(clientSocket);
    }else if (request.equals("ReturnToken")) {
    if(blockedList.isEmpty()) count++;
    else   sendMsgToClient(blockedList.poll(), "Continue");
    }
   }
  } catch (Exception e) {
  System.err.println(e);
  }
 }
 static void sendMsgToClient(Socket clientSocket,String msg) throws
UnknownHostException, IOException {
  PrintWriter newOut = new
PrintWriter(clientSocket.getOutputStream(),true);
  newOut.println(msg);
  clientSocket.close();
 }
}
```

```
public class Client {
 public static void main(String[] args) {
 try {
   while (true) {
    Socket srvSocket = new Socket("localhost", 1033);
    sendMsq(srvSocket,"GetToken");
    BufferedReader in = new BufferedReader(new
InputStreamReader(srvSocket.getInputStream()));
    System.out.println(in.readLine());
    srvSocket.close();
    //do Something...
    Thread.sleep(1000);

    srvSocket = new Socket("localhost", 1033);
    sendMsq(srvSocket,"ReturnToken");
    srvSocket.close();
   }
  } catch (Exception e) {
  System.err.println(e);
  }
 }
 private static void sendMsq(Socket srvSocket, String msg) throws
UnknownHostException, IOException {
  PrintWriter out = new PrintWriter(srvSocket.getOutputStream(), true);
  out.println(msg);
  }
}
```

9. (Oktobar 2012) Međuprocesna komunikacija razmenom poruka

Implementirati veb servis, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class NetBuffer {
 public NetBuffer(String host, int
 port) public void put(int d[]);
 public int get();
 public int
getK();
}
```

Servis treba da obezbedi usluge ograničenog kružnog bafera u koji se može atomično umetati K celobrojnih podataka, gde je K konstanta koja je definisana pri kreiranju tog bafera.

`NetBuffer` se povezuje sa odgovarajućim serverom preko koga se vrši razmena podataka. Metode `put` i `get` imaju semantiku primitiva za rad sa ograničenim kružnim baferom, a služe za atomično umetanje K elemenata, odnosno za dohvatanje jednog elementa iz bafera koji se nalazi na serveru. Metoda `getK` dohvata i postavlja konstantu K za koju je udaljeni bafer kreiran. Na raspolaganju je klasa `BoundedBuffer` koja implementira navedeni ograničeni kružni bafer i ima sledeći interfejs:

```
public class BoundedBuffer {
    public BoundedBuffer (int N, int K) ;
    public synchronized void put(int
d[]);
    public synchronized int get(); }
```

Parametar N u konstruktoru predstavlja dimenziju bafera. Poznato je da je K<N, ali nije poznato da li je N deljivo sa K. Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama (kod sa vežbi ne treba prepisivati, nego precizno navesti koja klasa ili koji metod se koriste i/ili menjaju, nasleđuju, ...). Nije potrebno proveravati uspešnost izvršavanja operacija (*try/catch* klauzule).

Rešenje:

```
public class NetBuffer extends Usluga {
 public NetBuffer(String host, int port) {
 super(host, port);
  getK();
 }
 private int K;

 public void put(int d[]) {
  String message = "#put#";
  for (int i = 0; i < K; i++) {
  message += d[i] + "#";
  }
  sendMessage(message);
 receiveMessage();
  }
 public int get() {
  String message = "#get#";
 sendMessage(message);
 return receiveIntMessage();
 }
public int getK() {
  String message = "#getK#";
  sendMessage(message);
  return receiveIntMessage();
```

10. (Septembar 2013) Međuprocesna komunikacija razmenom poruka

Na programskom jeziku Java, koristeći priključnice (*sockets*) i mehanizam prosleđivanja poruka (*message passing*), realizovati sistem koji udaljeno računa zbir zadatih redova matrice. Sistem se sastoji od jednog procesa servera i jednog procesa klijenata. Proces server u svom kontekstu izvršava više niti-radnika (*workers*) koje su realizovane po principu skupa zadataka (*bag-of-tasks*), gde svaka nit-radnik se izvršava na sledeći način:

```
while (true) {
 // dohvati jedan zadatak (indeks reda matrice) iz skupa
 zadatih if (nema preostalih zadataka) break;
 //izvrši zadatak, tj. sumiraj zadati red i upisi
rezultat
}
```

Na ovaj način redovi matrice se sumiraju konkurentno tako što svaka nit-radnik uzima po jedan zadatak i izvršava ga. Klijentski proces na početku šalje poruku za započinjanje (*start*) zajedno sa brojem niti-radnika za izvršavanje, nakon čega sledi poruka za računanje (*calculate*) sa brojevima indeksa redova matrice koje treba sumirati. Serverski proces zatim ove indekse smešta u skup zadataka (*bag-of-tasks*) nad kojim pokreće potreban broj nitiradnika. Po završetku izračunavanja serverski proces vraća rezultat klijentskom procesu. Primer klijentskog procesa dat je u nastavku:

```
PrintWriter out = ...; BufferedReader in = ...;
out.println("#start#4#"); //zapocni operaciju i zadaj br. niti-
radnika
int N = Integer.parseInt(in.readLine()); //dohvati br. redova matrice

String indexes = ""; //postavi indekse redova koji se
sumiraju for (int i = 0; i < N; i++) if(i%2==0)
indexes+=i+"#";
out.println("#calculate#"+indexes); //posalji zahtev

System.out.println("Result :"+in.readLine()); //dohvati rezulat
```

Pretpostaviti da serverski proces „osluškuje“ port 1033 preko koga prima zahteve i da matrica nad kojom se vrše izračunavanja je inicijalizovana i poznatih dimenzija. Za realizaciju skupa zadataka na raspolaganju je klasa `List<Integer>` sa metodama `add()` i `remove(0)` za smeštanje odnosno uzimanje jednog elementa iz liste, koje su predviđene za konkurentno pozivanje (*thread-safe*). Takođe, na raspolaganju je metoda `countTokens()` klase `StringTokenizer` koja vraća broj preostalih prepoznatih tokena u nizu koji se obrađuje. Napisati kompletan kod serverskog procesa i niti-radnika. Nije potrebno proveravati uspešnost izvršavanja operacija (*try/catch* klauzule).

Rešenje:

```
import java.io.*;
import java.net.*;
import java.util.*;

public class Server {
 public static final int N = ...;
 public static final int [][] a ={...};
 public static List<Integer> bagOfTasks = Collections.synchronizedList(new
LinkedList<Integer>());
 public static int result;
 public static Worker[] workers;
 public static int workerNum;
 public static void main(String[] args) {
```

```
   ServerSocket sock = new ServerSocket(1033);
   while (true) {
    Socket clientSocket = sock.accept();
    BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),true);
    while (true) {
     String request = in.readLine();
     StringTokenizer st = new StringTokenizer(request, "#");
     String functionName = st.nextToken();
     if (functionName.equals("start")) {
      workerNum = Integer.parseInt(st.nextToken());
      result=0;
      out.println(""+N);
     } else if (functionName.equals("calculate")){
      for (int i = 0; i <= st.countTokens(); i++)
       bagOfTasks.add(Integer.parseInt(st.nextToken()));

      workers = new Worker[workerNum];
      for (int i = 0; i < workers.length; i++) {
      workers[i]= new Worker();
       workers[i].start();
      }
      synchronized (Server.class) {
       while (workerNum>0){ ; //wait to complete all
       Server.class.wait();
       }
      }
      out.println(""+result);
      break;
     }
    }
   }
 }
}

class Worker extends Thread{
@Override
 public void run() {
  while (!Server.bagOfTasks.isEmpty()) {
   int i = Server.bagOfTasks.remove(0), res=0;
   for (int j = 0; j < Server.N; j++) {
   res+=Server.a[i][j];
   }
   synchronized (Server.class) {
   Server.result+=res;
   }
  }
  synchronized (Server.class) {
  Server.workerNum--;
   Server.class.notifyAll();
  }
 }
}
```

11. (ispit 2006) Sinhronizacija i komunikacija između procesa

Date su klase `Base` i `Derived` na programskom jeziku Java.

```
public class Base {
     public int f(int a, int b) { ... }
     public int f(int a) { ... }
}

public class Derived extends Base {
     public int f(int a, int b) { ... }
}
```

Klasa `Base` ima dve operacije `f` sa preklopljenim imenom (engl. *overloaded*). Klasa `Derived` je izvedena iz klase `Base` i u njoj je redefinisan metod `f(int,int)` (engl. *overrided*). Implementirati klase `BaseProxy` i `DerivedProxy`, koje će se instancirati na klijentskoj strani, a koje će predstavljati posrednike (engl. *proxy*), do stvarnih objekata na serverskoj strani koji izvršavaju stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko priključnica (engl. *socket*) i razmenom poruka (engl. *message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

(3 poena)

```
public class BaseProxy extends Usluga {

      public BaseProxy (String host, int port){
            super(host, port);
      }

      public int f(int a, int b){
            String message =
                              "#f1#" + a +
                              "#" + b + "#";
            sendMessage(message);

            return receiveIntMessage();
      }

      public int f(int a){
            String message =
                              "#f2#" + a + "#";
            sendMessage(message);

            return receiveIntMessage();
      }
}
```

(3 poena)

```
public class DerivedProxy extends BaseProxy {

      public DerivedProxy (String host, int port){
            super(host, port);
      }

      public int f(int a, int b){
            String message =
                              "#f3#" + a + "#" + b + "#";
            sendMessage(message);
```

```
            return receiveIntMessage();
      }
}
```

(4 poena)

```
public class RequestHandler extends Thread{

      protected void processRequest(String request){
            ...
            } else if (functionName.equals("f1") ||
                        functionName.equals("f3")
                       ){
                  ind1 = ind2+1;
                  ind2 = request.indexOf("#",ind1+1);
                  String arg2 = request.substring(ind1,ind2);

                  int a = Integer.parseInt(arg1);
                  int b = Integer.parseInt(arg2);

                  int res =
                              functionName.equals("f1") ?
                              new Base().f(a, b) :
                              new Derived().f(a, b)
                              ;

                  sendMessage("" + res);
            } else if (functionName.equals("f2")){
                  int a = Integer.parseInt(arg1);

                  int res = new Base().f(a);

                  sendMessage("" + res);
            }

      }
      ...
}
```

12. (ispit 2006) Sinnhronizacija i komunikacija između procesa

Implementirati web service, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
Public class MatF{

     Public int racunaj(char op, int operand1, int
operand2){…}

}
```

Metod Matf.racunaj treba operaciju *op*, koja može biti +, - ili *, da primeni na operande operand1 i operand2 i da vrati rezultat. Korisnik treba da na instancira objekat klase MatF, na svojoj, klijenstkoj strani, koji će predstavljati posrednik (*Proxy*), do stvarnog objekta klase MatFServer, koja će vršiti stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko Socketa i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

(4 poena)

```
public class MatF extends Usluga {

     public MatF(String host, int port){
          super(host, port);
     }

     public int racunaj(char op, int operand1, int operand2){
          String message =
                         "#racunaj#" + op + "#" + operand1 +
                         "#" + operand2 + "#";
          sendMessage(message);

          return receiveIntMessage();
     }

}
```

(4 poena)

```
public class RequestHandler extends Thread{
     ...
     ServerMatF sm = new ServerMatF();

     protected void processRequest(String request){
          ...
          } else if (functionName.equals("racunaj")){
               ind1 = ind2+1;
               ind2 = request.indexOf("#",ind1+1);
               String arg2 = request.substring(ind1,ind2);

               ind1 = ind2+1;
               ind2 = request.indexOf("#",ind1+1);
               String arg3 = request.substring(ind1,ind2);
```

```
                char op = arg1.charAt(0);
                int a = Integer.parseInt(arg2);
                int b = Integer.parseInt(arg3);

                int res = sm.racunaj(op, a, b);

                sendMessage("" + res);
            }
        }
        ...
}
```

(2 poena)

```
public class ServerMatF{

    public int racunaj(char op, int operand1, int operand2){
        switch(op){
            case '+' : return operand1 + operand2;
            case '-' : return operand1 - operand2;
            case '*' : return operand1 * operand2;
            default: return operand1 + operand2;
        }
    }
}
```

13. (ispit 2006) Sinhronizacija i komunikacija između procesa

Implementirati veb servis (*Web Service*), sa svim potrebnim delovima i na klijentskoj i na serverskoj strani, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class Service{
      public Data calc(Data d){…}
      public double calc(Data d1, Data d2) {…}
}
```

Metodi `Service.calc` vrše odgovarajuće obrade nad podacima tipa `Data`, a zatim vraćaju odgovarajući rezultat. Korisnik treba da instancira objekat klase `Service` na svojoj, klijenstkoj strani, koji će predstavljati posrednik (*proxy*) do stvarnog objekta na serverskoj strani koji vrši stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama. Interfejs klase `Data` dat je sledećim kodom:

```
public Data{
      public Data(String str){…}
      public String serialize(){…}
      …
}
```

Metod `Data.serialize` služi da napravi string reprezentaciju odgovarajućeg objekta, iz koga se može izvršiti rekonstrukcija tog objekta pomoću konstruktora `Data(String)`. Smatrati da ovaj string neće imati znak `'#'` u sebi.

Videti prethodne ispitne rokove, uz sledeću izmenu: kada je potrebno poslati podatak tipa `Data`, treba poslati string: `d.serialize()`. Kada je potrebno primiti podatak tipa `Data`, onda treba izvršiti konverziju primljene poruke (nastale sa `d.serialize()`) u objekat tipa `Data` pomoću postojećeg konstruktora:

```
message = ...;
return new Data(message);
```

Naravno na odgovarajući način treba primiti poruku koja predstavlja tip `double`, analogno primanju poruke tip `int`.

14. (ispit 2006) Sinhronizacija i komunikacija između procesa

Implementirati web service, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class MatF{
     public int racunaj(int[] a){…}
}
```

Metod `Matf.racunaj` prima niz elemenata `a`, obrađuje ih i vraća rezultat. Korisnik treba da instancira objekat klase `MatF`, na svojoj, klijenstkoj strani, koji će predstavljati posrednik (*engl. Proxy*), do stvarnog objekta klase `MatFServer`, koja će vršiti stvarno izračunavanje. Međuprocesnu komunikaciju realizovati preko Socketa i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Pogledati rešenja prethodnih rokovima. Jedina razlika je u slanju i prijemu poruke kada se šalje niz. To može da se uradi tako što se šalje poruka u formatu `#funcName#ArrayLen#a0#a1#...#`:

```
// Slanje
StringBuffer sb = new StringBuffer("#racunaj#" + a.length + "#");
for (int i = 0; i < a.length; ++ i)
      sb.append("" + a[i] + "#”);
System.out.println(sb.toString());
String message = sb.toString();

// Prijem
StringTokenizer st = new StringTokenizer(message,"#");
String fName = st.nextToken();
int n = Integer.parseInt(st.nextToken());
int[] a = new int [n];
for (int i =0; i < n; ++i)
      a[i] = Integer.parseInt(st.nextToken());
```

15. (ispit 2006) Sinhronizacija i komunikacija između procesa

Implementirati Web Service, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class NetSemaphore {
      public NetSemaphore(String host, int port){…}
      public void create(String name, int initialValue){…}
      public void signalS(String name){…}
      public void waitS(String name){…}
}
```

Servis treba da obezbedi sinhronizaciju niti koje se izvršavaju na udaljenim računarima. `NetSemaphore` se povezuje sa odgovarajućim serverom preko koga se vrši sinhronizacija. Metode `create`, `signalS` i `waitS` imaju semantiku primitiva za rad sa standardnim brojačkim semaforima, a služe da kreiraju, signaliziraju i čekaju na semafor `name` koji se nalazi na serveru. Ako je semafor sa imenom `name` već postoji, onda primitiva `create` nema efekta, već se koristi postojeći semafor. Ako se pozivaju primitive `signalS` i `waitS` na semaforima koji ne postoje, ignorisati te pozive. Pretpostaviti da postoji klasa `Semaphore` koja implementira standardni brojački semafor i ima sledeći interfejs:

```
public class Semaphore {
      public Semaphore(int initialValue){…}
      public void signalS(){…}
      public void waitS(){…}
}
```

Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama (kod sa vežbi ne treba prepisivati, nego npr. reći koja klasa ili koji metod se koriste i/ili menjaju, nasleđuju, ...).

```
public class NetSemaphore extends Usluga {
      public NetSemaphore(String host, int port){ super(host, port); }
      public void create(String name, int initialValue){
            String message = "#create#" + name + "#" + initialValue + "#";
            sendMessage(message);
            receiveMessage();
      }

      public void signalS(String name) {
            String message = "#signal#" + name + "#";
            sendMessage(message);
            receiveMessage();
      }

      public void waitS(String name) {
            String message = "#wait#" + name + "#";
            sendMessage(message);
            receiveMessage();
      }
}
```

Na serverskoj strani u klasi `Server` treba da se dodaju sledeće promenljive:

```
HashMap sems = new HashMap(); // kolekcija semafora
Semaphore mutex = new Semaphore(1); // sinhronizaciona promenljiva
```

`RequestHandler` treba izmeniti na sledeći način:

```
public class RequestHandler extends Thread {
...
      HashMap semaphores;
      Semaphore mutex;
...
      public RequestHandler(Socket clientSocket, HashMap semaphores,
                            Semaphore mutex){
            this.sock = clientSocket;
            this.semaphores = semaphores;
            this.mutex = mutex;
                  ...
      }

      protected void processRequest(String request){
            StringTokenizer st = new StringTokenizer(request,"#");
            String functionName = st.nextToken();
            String semName = st.nextToken();

            if (functionName.equals("create")){
                  mutex.waitS();
                  if (!semaphores.containsKey(semName)){
                      int initialValue = Integer.parseInt(st.nextToken());
                      semaphores.put(semName, new Semaphore(initialValue));
                  }
                  mutex.signalS();

            } else if (functionName.equals("signal") ||
                       functionName.equals("wait")){

                  Semaphore s = null;
                  mutex.waitS();
                  if (semaphores.containsKey(semName))
                        s = (Semaphore) semaphores.get(semName);
                  mutex.signalS();

                  if (s != null) {
                        if (functionName.equals("signal")) s.signalS();
                        else s.waitS();
                  }
            }


            sendMessage("done");
      }
```

16. (ispit 2007) Sinhronizacija i komunikacija između procesa

Implementirati Web Service na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class MutexProxy {
      public MutexProxy(String host, int port, String name){…}
      public ~MutexProxy(){…}
      …

}
```

Klasa treba da obezbedi da u jednom trenutku ne može postojati više od jednog objekta sa istim parametrom `name`. Smatrati da se u sistemu ne pojavljuje više od 10 objekata sa različitim imenima. Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

```
public class MutexProxy extends Usluga {
  private String name;
  public MutexProxy(String host, int port, String name){
      super(host,port);
      this.name = name;
      String message = "#enter#"+name+"#";
      sendMessage(message);
      message = receiveMessage();
  }
  public ~MutexProxy(){
      String message = "#leave#"+name+"#";
      sendMessage("#leave#"+name+"#");
      message = receiveMessage();
  }
}

public class RequestHandler extends Thread{

      protected static String[] Names = new String[10];

      protected void processRequest(String request){
            ...
            } else if (functionName.equals("enter"){
                  boolean postoji;
                  do{
                    int i = 0;
                    int sl = 0;
                    postoji = false;
                    synchronized(Names){
                      while (i<=9){
                        if(Names[i].equals(arg1)) postoji = true;
                        if(Names[i].equals("")) sl=i;
                        i++;
                      }
                      if (postoji) Names.wait();
                        else Names[sl] = arg1;
                    }
                  }while(postoji);
                  sendMessage("confirm");
            } else if (functionName.equals("leave")){
                  int i = 0;
```

```
                synchronized(Names){
                    while (i<=9){
                      if(Names[i].equals(arg1)) {
                            Names[i] = “”;
                            Names.signalall();
                            }
                      i++;
                    }
                }
                sendMessage("confirm");
            }

        }
        ...
}
```

17. (ispit 2007) Sinhronizacija i komunikacija između procesa

Neki operativni sistem podržava koncept *poštanskog sandučeta* (*mailbox*) kao sistemskog resursa koji korisnički proces od sistema dobija sistemskim pozivom:

```
MbxHandle mbx_open (char* symbolicName);
```

Ova operacija vraća „ručku“ kojom se identifikuje sanduče koje je otvoreno; ako operacija nije uspela, vraća se `NULL`. Sistem pokušava da pronađe već kreirano sanduče sa zadatim simboličkim imenom, a ako takvo ne postoji, kreira novo sanduče sa tim simboličkim imenom. Na ovaj način omogućeno je deljenje sandučića između procesa.
Operacije nad sandučićima su:

```
void mbx_send(MbxHandle, char* message);  // Asynchronous send
void mbx_receive(MbxHandle, char* message_buffer);  // Synchronous receive
```

Napisati kod dva procesa koji međusobno komuniciraju posredstvom poštanskog sandučeta. Proces A šalje poruku procesu B sa sadržajem „`?`“. Na svaku ovakvu primljenu poruku, proces B odgovara procesu A porukom u kojoj je jedan novi ceo broj, formatizovan kao niz znakova – decimalnih cifara (maksimalne dužine 10). Kada je poslao „`?`“, proces A čeka da dobije ovaj odgovor, štampa dobijeni niz znakova na standardni izlaz, a onda nastavlja dalje sa slanjem nove poruke „`?`“ i tako ciklično.
Process A:

```
#include ...  // System header for mailbox
#include <stdio.h>

int main () {
  MbxHandle mbxAB = mbx_open("ABMbx");
  MbxHandle mbxBA = mbx_open("BAMbx");
  if ((mbxAB==NULL)||(mbxBA==NULL)) exit(1); // Error
  while (1) {
    char buf[10];
    mbx_send(mbxAB,"?");
    mbx_receive(mbxBA,buf);
    printf(buf);
  }
}
```

Process B:

```
#include ...  // System header for mailbox
#include <stdio.h>

int main () {
  MbxHandle mbx = mbx_open("ABMbx");
  MbxHandle mbx = mbx_open("BAMbx");
  if ((mbxAB==NULL)||(mbxBA==NULL)) exit(1); // Error
  for (unsigned long i=0; i<...; i++) {
    char buf[10];
    mbx_receive(mbxAB,buf);
    if (buf[0]<>'?') continue;
    sprintf(buf,"%d",i);
    mbx_send(mbxBA,buf);
  }
  exit(0);
}
```

18. (ispit 2007) Sinhronizacija i komunikacija između procesa

Na jeziku Java napisati kod za serverski demonski proces koji će obavljati sledeći posao:

- na portu 1024 „osluškivati“ zahteve sa klijenata;
- za svaki novi zahtev sa klijenta na ovom portu, zauzeće jedan novi port u opsegu 1025..1024+*N* koji do sada već nije zauzet na ovaj način (*N* je konstanta);
- kreirati novu nit koja će sa klijentom obavljati komunikaciju preko ovog novog porta;
- javiti klijentu broj ovog novog porta, a zatim nastaviti da „osluškuje“ nove zahteve.

Nakon kreiranja nove niti, klijent prelazi na komunikaciju sa tom niti na dostavljenom portu. Serverska nit uspostavlja ovu komunikaciju na zahtev klijenta i dalje obavlja neki specifičan posao (samo naglasiti mesto gde se taj posao obavlja).

Rešenje:

```
import java.net.*;
import java.io.*;

private class ChannelServer extends Thread {
  private ServerSocket mySocket;

  public ChannelServer (int port) {
    mySocket = new ServerSocket(port);
  }

  public void run () {
    try {
      Socket client = mySocket.accept();
      //... Ovde se radi specifičan posao
      client.close();
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}


public class MainServer {
  private final static int N = ...;
  private static bool[] allocatedPorts = new bool[N];

  private static int getFreePort () {
    for (int i=0; i<N; i++) {
      if (allocatedPorts[i]) continue;
      allocatedPorts[i]=true;
      return 1025+i;
    }
    return -1;
  }

  public static void main (String[] args) {
    try {
      ServerSocket sock = new ServerSocket(1024);
```

```
      while (true) {
        Socket client = sock.accept();
        int newPort = getFreePort();
        if (newPort==-1) break;
        new ChannelServer(newPort).start();
        PrintWriter pout = new PrintWriter(client.getOutputStream(),true);
        pout.println(new Integer(newPort).toString());
        client.close();
      }
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

19. (ispit 2007) Sinhronizacija i komunikacija između procesa

Implementirati veb servis (*Web Service*), sa svim potrebnim delovima i na klijentskoj i na serverskoj strani, na programskom jeziku Java, koji će krajnjem korisniku pružiti sledeći interfejs:

```
public class Service{
      public void sendOP(String s){...}
      public String receiveOP(){...}
}
```

Servis treba da korisniku, koji nema direktan pristup Internetu, omogući komunikaciju sa udaljenim računarom povezanim na Internet. Metod `Service.sendOP` treba da obezbedi da se string *s* pošalje udaljenom računaru. Metod `Service.receiveOP` korisniku omogućava da prihvati string od udaljenog računara. Korisnik (proces ili nit) treba da instancira objekat klase `Service` na svojoj, klijenstkoj strani, koji će predstavljati posrednika (*proxy*) do stvarnog objekta na serverskoj strani koji vrši komunikaciju sa udaljenim računarom. Pri instanciranju objekta klase `Service`, neophodno je proslediti potpunu adresu udaljenog računara. Moguće je kreirati više instanci klase `Service`. Međuprocesnu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Rešenje:

```
public class Service extends Usluga{
      public Service(String host, int port){
            super(host, port); sendMessage("#con#"+Uhost+"#"+Uhost+"#");
      }
      public void sendOP(String s){
            sendMessage("#send#"+s+"#");
      }
      public String receiveOP(){
            sendMessage("#rec#a#");
            return receiveMessage();
      }
}
public class ServerUsluga extends Usluga{
      public ServerUsluga(String host, int port){
            super(host, port);
      }
      public void send(String s){
            sendMessage(s);
      }
      public String receive(){
            return receiveMessage();
      }
}

Izmene u klasi RequestHandler:

protected ServerUsluga u = null;
```

```
protected void processRequest(String request){
      int ind1 = 1, ind2= request.indexOf("#", ind1+1);
      String functionName = request.substring(ind1,ind2);
      ind1 = ind2+1; ind2 = request.indexOf("#",ind1+1);
      String arg1 = request.substring(ind1,ind2);
      ...
      }else if (functionName.equals("con") && (u==null)){
            ind1 = ind2+1; ind2 = request.indexOf("#",ind1+1);
            String arg2 = request.substring(ind1,ind2);
            int p = arg2.parseInt();
            u = new ServerUsluga(arg1,p);
      }else if (functionName.equals("send") && (u!=null)){
            u.send(arg1);
      } else if (functionName.equals("rec") && (u!=null)){
            sendMessage(u.receive());
      }
}
```

20. (ispit 2007) Sinhronizacija i komunikacija između procesa

Potrebno je obezbediti da korisnik može da vrši slanje podataka ka proizvoljnom odredištu na sledeći način. Korisnik navodi adresu odredišta i string koji je potrebno poslati. Slanje počinje tako što se odredištu pošalje broj znakova koji treba da se pošalju. Potom se prije slanja svakog znaka, čeka da se od odredišta dobije dozvola za slanje. Odredište šalje dozvolu u obliku stringa *"r"*.

Korisniku treba pružiti sledeći interfejs:

```
public class Service{
      public void send(String host, int port, String s){…}
}
```

Operacija slanja sa strane korisnika treba da traje što je moguće kraće. Korisniku je dozvoljeno da operaciju poziva neposredno iza prethodnog poziva iste operacije. Kao rešenje, potrebno je napisati kod koji vrši slanje, kao i kod odredišta koje može da prihvata više stringova u isto vreme. Po završenom prijemu odredište treba da ispiše primljeni string na standardni izlaz. Svu komunikaciju realizovati preko priključnica (*socket*) i razmenom poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Rešenje:

```
public class Transmiter extends Thread{  //ista kao klasa Request Handler
      ...
      private String zaSlanje;

      public Transmiter(Socket sock, String s){
            zaSlanje = s;
            ...
      }

      public void run(){
            try{
                  int n = zaSlanje.length();
                  sendMessage("" + n);
                  for(int i = 0; i < n; i++){
                        while (!receiveMessage().equals("r")) ;
                        sendMessage(zaSlanje.charAt(i));
                  }
                  sock.close();
            }catch(Enception e){};
      }

}


public class Service extends Usluga{
      public void send(String host, int port, String s){
            Socket sock = new Socket(host, port);
            Thread t = new Transmiter(sock, s);
            t.start();
      }
}

Koristi se klasa Server.

public class RequestHandler extends Thread{

      ...
```

```
public void run(){
      try{
            String Sduzina = receiveMessage();
            int duzina = Integer.parseInt(Sduzina);
            String cum = "";
            while(sock.isInputShotdown() and (duzina>0)){
                  sendMessage("r");
                  String next = receiveMessage();
                  cum = cum + next;
                  duzina--;
            }
            System.out.println(cum);
      }catch(Exception e){};
}

      ...
}
```

21. (2. januar 2008.) Sinhronizacija i komunikacija između procesa

Korišćenjem koncepta priključnica (*socket*), na jeziku Java implementirati klijentsku i serversku stranu komunikacije po principu poziva udaljene procedure (*remote procedure call*, RPC). Na klijentskoj strani realizovati klasu koja predstavlja *stub* i koja poseduje operaciju add(int,int) koja treba da vrati zbir dva argumenta. Njena implementacija treba da obezbedi serijalizaciju argumenata, komunikaciju sa serverskom stranom, deserijalizaciju rezultata i vraćanje kontrole pozivaocu. Na serverskoj strani implementirati program nad kojim se može pokrenuti demonski proces-osluškivač (*listener daemon*) koji na portu 1050 prihvata zahteve sa klijentske strane inicirane pozivom operacije add od strane klijenata i za svaki takav primljeni zahtev izračunava zbir dva argumenta i vraća rezultat klijentu.

Rešenje:

```
import java.io.*;
import java.net.*;

public class AddStub {
      public int add(int a, int b) {
            String message = "#" + a + "#" + b + "#";
            out.println(message);
            try{
                  message = in.readLine();
                  return Integer.parseInt(message);
            } catch(Exception e){
                  System.exit(1);
            }
            return 0;
      }

      public AddStub(String host, int port){
            try {
                  sock = new Socket(host, port);
            out = new PrintWriter(sock.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                    sock.getInputStream()));
            } catch (Exception e) {
                  System.exit(1);
            }
      }


      protected Socket sock = null;
      protected PrintWriter out = null;
      BufferedReader in = null;

      protected void finalize() throws Throwable{
            super.finalize();
            try{
                  out.close();
                  in.close();
                  sock.close();
            } catch(Exception e){
            }
      }
}
Server:
public class Server{
      public static void main(String[] args){
            ServerSocket server = new ServerSocket(1050);
```

```
            while(true){
                  Socket s = server.accept();
                  PrintWriter out = new PrintWriter(sock.getOutputStream(),
                                                                  true);
                  BufferedReader in = new BufferedReader(new
                              InputStreamReader(sock.getInputStream()));
                  try{
                        String s = in.readLine();
                        StringTokenizer st = new
                              StringTokenizer(request,"#");
                        int a = Integer.paresInt(st.nextToken());
                        int b = Integer.parseInt(st.nextToken());
                  }catch(Exception e){
                  }
                  oup.println(a+b);
                  try{
                        in.close();
                        out.close();
                        socket.close();
                  }catch(Exception e){
                  }
            }
      }
}
```

22. (2. februar 2008.) Sinhronizacija i komunikacija između procesa

Procesi Alarm i Passengers prikazani dole treba da se sinhronizuju na sledeći način. Proces Passengers signalizira ulazak i izlazak putnika iz neke obezbeđene zone. Proces Alarm treba da čeka blokiran sve dok broj putnika koji su trenutno u obezbeđenoj zoni ne pređe neki prag THRESHOLD. Tada treba da se deblokira i uključi alarm pozivom operacije alert. Realizovati monitor Signaller koji obezbeđuje opisanu sinhronizaciju pomoću klasičnih uslovnih promenljivih.

```
process Alarm;
begin
  loop
    Signaller.signal();
    alert;
  end;
end;
```

```
process Passengers;
begin
loop
  Signaller.entry();
        Signaller.exit();
end;
end;
```

Rešenje:

```
monitor Signaller;
  export signal, entry, exit;
  var
    count : integer;
    threshold : condition;

  procedure signal ();
  begin
    while (count<=THRESHOLD) wait(threshold);
  end;

  procedure entry ();
  begin
    count := count + 1;
    if (count>THRESHOLD) signal(threshold);
  end;

  procedure exit ();
  begin
    count := count - 1;
  end;

begin
  count := 0;
end;
```

23. (2. jun 2008.) Sinhronizacija i komunikacija između procesa

Za neki problem postoji rešenje gde se ponavlja sledeći postupak. Poznata je početna vrednost celobrojne promenljive `y`. Na osnovu vrednosti y, računaju se nove vrednosti niza od N celih brojeva `x`, tako da važi `x[i]=Fi(y)` za svako `i` iz intervala `[0..N-1]`, gde su `F0()`, `F1()`,..., `FN-1()` funkcije koje prihvataju jedan ceo broj, i vraćaju rezultat koji je takođe ceo broj. Kada se izračuna novi sadržaj niza x, računa se nova vrednost y tako što se funkciji `F()` prosledi prethodno izračunati niz `x`. Izračunavanje funkcija `Fi()` traje prilično dugo, dok izvržavanje funkcije `F()` traje veoma kratko. Da bi se izračunavanje ubrzalo, ovaj proces je potrebno paralelizovati tako što se na jednom računaru izvršava program koji sadrži funkciju `F()`, promenljivu `y`, niz `x` i dva niza koja pomažu da se dodje do funkcija `Fi()` (i-ti elementi ova dva niza sadrže host i port računara koji poseduje funkciju `Fi()`). Na programskom jeziku Java napisati program za računar koji izvršava funkciju `F()`, kao i ceo kod potreban za pokretanje programa. Takođe objasniti šta radi program za jedan od preostalih N računara na kojima se izvršavaju funkcije `Fi()`. Voditi računa da se vrednosti prihvate u niz što je pre moguće, tj. da se rezultati funkcija `Fi()` ne prihvataju u niz po nekom fiksnom redosledu. Pri rešavanju zadatka, za komunikaciju između programa koristiti priključnice (*sockets*) i mehanizam prosleđivanja poruka (*message passing*). Dozvoljeno je korišćenje koda prikazanog na vežbama.

Rešenje:

Glavni program:

```
public class m {
      private static int n;
      private static String[] hosts;
      private static int[] ports;
      private static Slave[] slaves;
      private static int counter = 0;
      public static Boolean kraj = false;

      public static int y = 1;
      public static int[] x;
      public static Flag flag = new Flag();
      public static Object sinhronizacija = new Object();

      public static void up_counter(){
            synchronized(sinhronizacija){
                  counter++;
                  if (counter == n) sinhronizacija.notifyAll();
            }
      }
      public static void reset_counter(){
            counter = 0;
      }
      public static void main(String[] args) {
            /* dohvatanje pocetnih vrednosti za n, hosts i ports
            .... */
            while (y<100){
                  try{
                        synchronized(sinhronizacija){
                              while (counter < n) sinhronizacija.wait();
                        }
```

```
                }catch(Exception e){
                        kraj = true;
                }
                if (!kraj){
                        y = F(n,x);
                        reset_brojac();
                        flag.change();
                }
        }
        kraj = true;
        flag.change();
    }
    private static int F(int n, int[] x) {
        ....
    }
}
public class Flag {
    protected int flag = 1;

    public synchronized void change() {
        flag = -flag;
        this.notifyAll();
    }
    public synchronized void test_wait(int f) {
        try{
                while ((f != flag) && !m.kraj ) wait();
        }catch(Exception E){
                System.exit(1);
        }
    }
    public int getVal(){
        return flag;
    }
}
```

Za klasu Slave se može iskoristiti klasa usluga:

- dva dodatna polja:

```
protected int flag; - zadnja vrednost flega
protected int i; - redni broj
```

- izmena u kodu:

nova run metoda

```
public void run(){
        flag = m.flag.getVal();
        while(! m.kraj){
                flag = -flag;
                send(m.y);
                m.x[i] = receive();
                m.up_brojac();
                m.flag.test_wait(flag);
        };
        disconect();
}
```

Programi na računarima koji izvršavaju funkcije Fi() kreiraju serversku konekciju, prihvate komunikaciju, i u petlji prihvataju jedan broj, računaju rezultat i šalju rezultat nazad.

24. (ispit 2009) Sinhronizacija i komunikacija između procesa

Koncept poštanskog sandučeta (*mailbox*), koji omogućava komunikaciju sa indirektnim imenovanjem, implementiran je monitorom koji ima sledeći interfejs:

```
type Msg = ...;  // Message type

monitor Mailbox;
 export sendMsg, getMsg;

 procedure sendMsg (msg : Msg);
 function  getMsg (): Msg;

end;
```

Procedura `sendMsg` šalje datu poruku u poštansko sanduče; pozivajući proces nakon slanja poruke odmah nastavlja izvršavanje. Funkcija `getMsg` uzima jednu poruku iz sandučeta; ukoliko je sanduče prazno, pozivajući proces se blokira sve dok poruka ne stigne.

Korišćenjem ovog koncepta napisati opšti oblik procesa koji u sebi ima kritičnu sekciju, uz ulazni i izlazni protokol koji obezbeđuje međusobno isključenje kritične sekcije u odnosu na iste takve procese.

Rešenje:

```
var   mutex : shared Mailbox;

process P;
begin
  ...
  mutex.getMsg;
  <critical section>;
  mutex.sendMsg(new Msg());
  ...
end;

begin
  mutex.sendMsg(new Msg());
end;
```

25. (ispit 2009) Sinhronizacija i komunikacija između procesa

Na programskom jeziku Java implementirati klasu koja sinhronizuje pristup operacijama tipa O1 i O2. Broj operacija tipa O1 koje se smiju izvršavati konkurentno je 10 dok je broj operacija tipa O2 koje se smiju konkurentno izvršavati neograničen, ali nije dozvoljeno da se konkurentno izvršavaju operacije oba tipa. Interfejs klase treba da bude po jedna metoda za svaki tip operacije u koju je umotan poziv odgovarajuće operacije (kao komentar navesti mesto gde se poziva odgovarajuća operacija). Pri rešavanju zadatka nije potrebno rešavati problem izgladnjivanja. Za sinhronizaciju je dozvoljeno koristiti samo jedan objekat i sinhronizovane (*synchronized*) blokove nad tim objektom.

Rešenje:

```
import java.lang.*;

public class OpWrapper {
    private static int val1 = 10;
    private static int val2 = 0;
    private static Object o = new Object();

    private OpWrapper() {
    }

    public static void Op1() {
        try{
        synchronized(o){
            while (val1 <= 0 || val2 > 0){
                o.wait();
            }
            val1--;
        }
        //poziv operacije tipa O1
        synchronized(o){
            val1--;
            o.notifyAll();
        }
        } catch(Exception e){
            System.out.print(e.toString());
        }
    }

    public static void Op2(){
        try{
        synchronized(o){
            while (val1 <10){
                o.wait();
            }
            val2++;
        }
        //poziv operacije tipa O2
        synchronized(o){
            val2--;
            o.notifyAll();
        }
        }catch(Exception e){
            System.out.print(e.toString());
        }
    }
}
```

26. (ispit 2009) Sinhronizacija i komunikacija između procesa

Neki sistem za obaveštavanje funkcioniše tako što svaki korisnik na svom računaru ima instaliran program za prijem poruka. Svaki od tih programa osluškuje port 1024 i preko tog porta prihvata zahteve za uspostavljanje kanala za komunikaciju. Kanal služi za prenos poruka tipa string. Napisati potrebne klase na programskom jeziku Java, koje će obavljati sve opisane funkcije pomenutog programa. Korisniku treba obezbediti metodu za prihvatanje jedne poruke iz proizvoljnog kanala. Ako nema nijedne poruke, nit koja je zatražila poruku se blokira. Ako nijedna nit ne traži poruku, obezbediti da se pamti samo zadnja primljena poruka. Za komunikaciju koristiti koncept priključnica (*socket*), a za sinhronizaciju sinhronizovane blokove i metode (*synchronized*).

Rešenje:

```
public class Channel extends Thread {
    protected Server s;
    protected Socket c;

    public Channel(Server server, Socket socket) {
        s = server;
        c = socket;
    }

    public void run(){
        BufferedReader br = null ;
        try{
            br = new BufferedReader(new
InputStreamReader(c.getInputStream()));
            while(!s.isFinished() && !c.isClosed() ){
                s.setMessage(br.readLine());
            }
        }catch(Exception e){
            e.printStackTrace();
        } finally{
            try {
                br.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
            try {
                c.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}

public class Server extends Thread{
    protected ServerSocket s;
    protected int port;
    protected String msg;
    protected boolean blocked;
    private boolean finished;

    public Server(int port) {
        this.port = port;
        finished = false;
        msg = null;
        blocked = false;
```

```
    }

    public synchronized void setMessage(String message){
        msg = message;
        if (blocked) notifyAll();
    }

    public synchronized String getMessage(){
        blocked = true;
        while ((msg == null) && !isFinished())
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        blocked = false;
        String ret = msg;
        msg = null;
        notifyAll();
        return ret;
    }


    public void finish(){
        finished = true;
        try {
            s.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public boolean isFinished(){
        return finished;
    }

    public void run(){
        try {
            s = new ServerSocket(port);
            while(!isFinished()) {
                try {
                    Socket k = s.accept();
                    Channel c = new Channel(this,k);
                    c.start();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }finally{
            if (!s.isClosed())
                try {s.close();
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
        }
    }

}
```

27. (ispit 2009) Sinhronizacija i komunikacija između procesa

Neki sistem za komunikaciju funkcioniše tako što su korisnici uvezani u niz (svaki osim prvog ima prethodnog i svaki osim poslednjeg ima sledećeg). Poruku koju treba proslediti svim korisnicima treba poslati bilo kojem korisniku u nizu, a odatle poruka dalje propagira ka oba kraja niza. Smatrati da nijedna poruka neće biti poslata prije uspostavljanja svih veza, kao i da neko ko šalje poruku raskida konekciju odmah posle slanja poruke. Pri pokretanju programa, kao prvi argument se prosleđuje adresa prethodnog u nizu na koju treba da se zakači program. Kao drugi argument prosleđuje se informacija da li je to neki od krajnjih korisnika u nizu. Poruka koja se šalje nikada ne sadrži znak #. Napisati program na programskom jeziku Java koji će raditi po opisanom protokolu i koji će obaveštavati jednog korisnika tako što će primljenu poruku štampati na standardni izlaz. Za komunikaciju koristiti priključnice (*sockets*), a za sinhronizaciju sinhrone blokove (*synchronized*).

Rešenje:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

public class Main {
    private static klijent pr = null;
    private static klijent sl = null;
    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        String prethodni = args[1];
        try {
            if (args[2] == "1"){
                pr = new klijent(new Socket(prethodni,1025));
                pr.start();
            }
            ServerSocket ss = new ServerSocket(1025);
            Socket s;
            if (args[2] == "2") {
                s = ss.accept();
                sl = new klijent(s);
            }
            if (pr!=null && sl!=null){
                pr.setDrugaStrana(sl);
                sl.setDrugaStrana(pr);
            }
            String poruka = " ";
            while(poruka != " "){
                s = ss.accept();
                try{
                    BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
                    poruka = br.readLine();
                    System.out.println(poruka);
                    if (pr!=null)pr.send(poruka);
```

```
                    if (sl!=null)sl.send(poruka);
                    br.close();
                    s.close();
                }catch(Exception e){
                    e.printStackTrace();
                }

            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class klijent extends Thread {
    private Socket s;
    public BufferedReader in;
    private BufferedWriter out;
    private klijent drugaStrana;
    /** Creates a new instance of klijent */
    public klijent(Socket soc) throws IOException {
        s = soc;
        drugaStrana = null;
        try {
            in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
            out = new BufferedWriter(new
OutputStreamWriter(s.getOutputStream()));
        } catch(Exception e){
            e.printStackTrace();
        }finally {
        }
    }

    public void setDrugaStrana(klijent ds){
        drugaStrana = ds;
    }

    public void send(String p){
        try {
            out.write(p);
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }

    public void run(){
        try{
            String po = " ";
            while(po != "#"){
                po = in.readLine();
                if (drugaStrana != null) drugaStrana.send(po);
                System.out.println(po);
            }
        }catch(Exception e){
            e.printStackTrace();
```

```
        }finally{
            try {
                out.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
            try {
                in.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }

    }
}
```

28. (ispit 2010) Sinhronizacija i komunikacija između procesa

Na jeziku Java, za udaljene računare obezbediti zajednički kružni bafer u koji se smeštaju celi brojevi. Smatrati da klijenti već imaju implementiran interfejs za umetanje broja u bafer i dohvatanje broja iz bafera. Prilikom kreiranja interfejsa, interfejs uspostavlja vezu sa serverom. Pri svakom umetanju broja, interfejs šalje string koji u počinje slovom 'p' i u nastavku sadrži broj koji se umeće. Kao odgovor se očekuje string "ok". Prilikom dohvatanja, interfejs šalje string "g", nakon čega očekuje odgovor u obliku stringa koji počinje slovom 'o' i u nastavku sadrži dohvaćeni broj. Potrebno je implementirti serversku aplikaciju koja će u pozadini obezbediti funkcionalnost kružnog ograničenog bafera. Za komunikaciju i sinhronizaciju koristiti mehanizam prosleđivanja poruka (*message passing*), kao i sinhronizovne blokove i metode (synchronized).

Rešenje:

```
public class Main {
    public Main() {
        bafer = new int[100];
        prvi = 0;
        zadnji = 0;
        zauzeto = 0;
    }
    private static ServerSocket ss;
    private int[] bafer;
    private int zauzeto;
    private int zadnji;
    private int prvi;

    public synchronized void put(int broj){
        while(zauzeto >= 100){
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        bafer[zadnji] = broj;
        zadnji = (zadnji + 1) % 100;
        notifyAll();
    }

    public synchronized int get(){
        int broj;
        while(zauzeto <= 0){
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        broj = bafer[prvi];
        prvi = (prvi + 1) % 100;
        notifyAll();
        return broj;
    }

    public static void main(String[] args) {
        Socket s;
        try {
            ss = new ServerSocket(4000);
```

```
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
                while (true) {
                    try {
                        s = ss.accept();
                        (new RequestHandler(s)).start();
                    } catch (IOException ex) {
                        ex.printStackTrace();
                    }
                }
            }
        }

        public class RequestHandler extends Thread {
            private Socket s;
            private static Main bafer = new Main();
            public RequestHandler(Socket s) {
                this.s = s;
            }

            public void run(){
                BufferedReader r;
                PrintWriter w;
                try {
                    r           =           new           BufferedReader(new
InputStreamReader(s.getInputStream()));
                    w            =            new            PrintWriter(new
OutputStreamWriter(s.getOutputStream()));

                    while (!s.isInputShutdown()){
                        String zahtev = r.readLine();
                        if (zahtev.charAt(0) == 'g') {
                            w.println("o"+bafer.get());
                        }else if (zahtev.charAt(0) == 'p'){
                            zahtev = zahtev.substring(1);
                            bafer.put(Integer.parseInt(zahtev));
                        }
                    }
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
```

29. (ispit 2010) Sinhronizacija i komunikacija između procesa

Napisati program na programskom jeziku Java koji računa sumu svih brojeva koje klijenti pošalju (u čitavom programu se formira samo jedna suma). Svaki put kada neki klijent pošalje broj, kao odgovor, vraća mu se vrednost sume nakon dodavanja tog broja. Klijenti očekuju server na portu 5000. Kada uspostave komunikaciju, klijenti šalju niz brojeva sa relativno dugačkim vremenskim intervalima između dva uzastopna slanja. Za komunikaciju i sinhronizaciju koristiti priključnice (*socket*) i sinhronizovane metode (*synchronized*).

Rešenje:

```
//main.java
public class Main {
    public Main() {
    }
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(5000);
            while(true){
              Socket s = ss.accept();
              (new Komunikator(s)).start();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

}
//komunikator.java
public class Komunikator extends Thread {

    private static class CelobrojnaSuma{
        private int suma;
        public CelobrojnaSuma(){
            suma = 0;
        }
        public synchronized int dodaj(int v){
            suma += v;
            return suma;
        }
    }

    private static CelobrojnaSuma suma;

    private Socket s;
    private BufferedReader in;
    private PrintWriter out;

    public Komunikator(Socket s) {
        this.s = s;
    }
    public void run(){
        try {
            in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
            out = new PrintWriter(new
OutputStreamWriter(s.getOutputStream()));
            while (!s.isInputShutdown()){
                String s = in.readLine();
                int i = Integer.parseInt(s);
                i = suma.dodaj(i);
                out.print(i);
            }
```

```
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        out.close();
        try {
            s.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

30. (ispit 2010) Sinhronizacija i komunikacija između procesa

Napisati program na programskom jeziku Java koji će pokrenuti N niti koje naizmenično ispisuju "ping" i "pong", ali tako da prvo sve niti ispišu "ping", pa potom sve niti ispišu "pong" i tako dalje naizmenično. Drugim rečima, nije dozvoljeno da bilo koja nit ispiše sledeću reč pre nego sve ostale niti ispišu tekuću reč. Svaka nit treba da ispiše po M reči ("ping" i "pong" su reči). M i N su konstante. Za komunikaciju i sinhronizaciju koristiti priključnice (*socket*) i sinhronizovane blokove (*synchronized*).

Rešenje:

```
public class PingPongNit extends Thread {

    private static int fleg = 1;
    private static Object sin = new Object();
    private static int brojacNitiNaBarijeri = 0;

    private static final int N=10;
    private static final int M=5;

    public static int brojNiti(){
        return N;
    }

    public PingPongNit() {
    }

    public void run(){
        int lokalnaKopijaFlega;
        int i = 0;
        while(i<M){
            synchronized (sin){
                lokalnaKopijaFlega = fleg;
            }

            if (lokalnaKopijaFlega >0){
                System.out.println("ping");
            }else{
                System.out.println("pong");
            }

            synchronized(sin){
                brojacNitiNaBarijeri++;
                while (brojacNitiNaBarijeri<N && lokalnaKopijaFlega==fleg)
                    try {sin.wait();
                    } catch (InterruptedException ex) {
                        ex.printStackTrace();
                    }
                if (lokalnaKopijaFlega == fleg){
                    fleg = -fleg;
                    brojacNitiNaBarijeri = 0;
                    sin.notifyAll();
                }
            }
            i++;
        }
    }
}

public class Main {
```

```
    public Main() {
    }

    private static PingPongNit[] niti;

    public static void main(String[] args) {
        int N = PingPongNit.brojNiti();

        niti =  new PingPongNit[N];

        for(int i=0; i<N; i++){
            niti[i] = new PingPongNit();
            niti[i].start();
        }

        for(int i=0; i<N; i++)
            try {niti[i].join();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
    }

}
```

31. (ispit 2010) Sinhronizacija i komunikacija između procesa

Napisati program na programskom jeziku Java koji će opsluživati zahteve za komunikaciju. Svaki pristigli zahtev se prihvata i opslužuje tako što se pronađe jedan slobodan računar i njegova adresa se pošalje klijentu koji je poslao zahtev. Ako su svi računari zauzeti, umesto adrese vraća se tekst "zauzeto" i raskida komunikaciju. Nakon toga, ako je klijent dobio adresu (string različit od "zauzeto"), klijent uspostavlja vezu sa tim računarom (komunikacija sa tim računarom nije deo zadatka) i po završetku obaveštava server da je završio korišćenje dodeljenog mu računara tako što šalje poruku sa sadržajem "kraj". U suprotnom odmah raskida komunikaciju i završava. Smatrati da se adrese računara kojima se prosleđuje komunikacija prihvataju kao parametri programa iz komandne linije i to jedna adresa, jedan string. Napisati i deo koda koji se izvršava na klijentskom računaru i naznačiti mesto gde treba da se umetne kod za komunikaciju sa dodeljenim računarom. Za komunikaciju i sinhronizaciju koristiti priključnice (*sockets*), mehanizam prosleđivanja poruka (*message passing*) i sinhronizovane blokove i metode (*synchronized*).

Rešenje:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Main {
    public Main() {
    }
    public static rs computers;
    public static void main(String[] args) {
        computers = new rs(args);
        try {
            ServerSocket server = new ServerSocket(1070);
            while(true){
                Socket s = server.accept();
                new worker(s).start();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

public class worker extends Thread {
    private Socket soc;
    public worker(Socket s) {
        soc=s;
    }
    public void run(){
        try {
            BufferedReader r = new BufferedReader(new
InputStreamReader(soc.getInputStream()));
            PrintWriter w = new PrintWriter(new
OutputStreamWriter(soc.getOutputStream()));
            int i = Main.computers.rezervisi();
```

```
            if (i<0){
                  w.println("zauzeto");
                  w.flush();
            }else{
                  w.println(Main.computers.adresa(i));
                  w.flush();
                  String odgovor = r.readLine();
                  Main.computers.oslobodi(i);
            }
            r.close();
            w.close();
            soc.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

public class rs {
    private String[] adrese;
    private boolean[] zauzet;
    public rs(String[] adr) {
        adrese = adr;
        zauzet = new boolean[adrese.length];
        for(int i=0;i<adrese.length;i++) zauzet[i]=false;
    }
    public synchronized int rezervisi(){
        for(int i = 0; i<zauzet.length; i++)
            if(zauzet[i]==false){
                zauzet[i] = true;
                return i;
            }
        return -1;
    }
    public String adresa(int i){
        return adrese[i];
    }
    public synchronized void oslobodi(int i){
        zauzet[i]=false;
    }
}

Segment koda klijenta:
Socket s = new Socket(...); //umjesto tacki treba da stoji adresa servera
BufferedReader r = new BufferedReader(new
InputStreamReader(soc.getInputStream()));
PrintWriter w = new PrintWriter(new
OutputStreamWriter(soc.getOutputStream()));

String adresa = r.readLine();
if (! adresa.equals("zauzeto"){
      //kod koji komunicira sa udaljenim racunarom
}
w.println("kraj");
w.flush();
w.close();
r.close();
s.close();
```

32. (ispit 2010) Sinhronizacija i komunikacija između procesa

Potrebno je na programskom jeziku Java, korišćenjem sinhornizovanih metoda (*synchronized*) implementirati ograničeni kružni bafer veličine N celobrojnih podataka tipa int. Pri umetanju podataka u bafer, umeće se po K podataka, gde je K konstanta koja se definiše pri kreiranju bafera. Podaci se u bafer umeću atomično. Ukoliko u baferu nema dovoljno mesta, pozivajuća nit se blokira, ali tako da se obezbedi fer pristup. To znači da nitima treba obezbediti onaj redosled umetanja elemenata u bafer u kojem su prvi put pokušale da umetnu niz od K podataka. Pri dohvatanju podataka iz bafera, dohvata se jedan po jedan podatak. Ukoliko nema dovoljno podataka, pozivajuću nit blokirati. U slučaju dohvatanja podataka nije potrebno voditi računa o fer redosledu dohvatanja podataka. Zadatak rešiti uz ogranienje da jedine sinhronizovane metode mogu biti metode klase koja predstavlja implementirani bafer. Poznato je da je K<N, ali nije poznato da li je N deljivo sa K.

Rešenje:

```
public class buffer {
    private int data[];
    private int K;
    private int N;
    private int free;   //broj slobodnih mjesta u baferu
    private int tail;   //pozicija sa koje se uzima sledeci podatak
    private int head;   //pozicija na koju se smjesta sledeci podatak
    private int tiket;   //uzimanje rednog broja
    private int sledeci;  //redni broj koji je na redu za umetanje u bafer
                          //kada se pojavi dovoljno slobodnog mjesta
    public buffer(int N, int K) {
        data = new int[N];
        this.N = N;
        free = N;
        this.K = K;
        head = 0;
        tail = 0;
        tiket = 0;
        sledeci = 0;
    }
    public synchronized void put(int d[]) throws InterruptedException{
        int red;
        if (free<K || sledeci != tiket) {
            red = tiket++;
            while(free<K || sledeci!=red){
                wait();
            }
            sledeci++;
        }
        for(int i =0; i<K; i++){
            data[head]=d[i];
            head = (head+1)%N;
        }
        free -= K;
        notifyAll();
    }
    public synchronized int get() throws InterruptedException{
        while (free == N){
            wait();}
        int ret = data[tail];
        tail = (tail+1)%N;
        free++;
        notifyAll();
        return ret;
    }
}
```

33. (ispit 2011) Sinhronizacija i komunikacija između procesa

Napisati program na programskom jeziku Java koji obezbeđuje sinhronizaciju prolaska vozila kroz pametnu raskrsnicu. Vozilima je na raskrsnici zabranjeno skretanje, tako da je potrebno obezbediti samo da se kroz raskrsnicu u svakom trenutku vozila kreću u samo jednom od dva moguća pravca. Vozila dolaze sa severa, juga, istoka i zapada. Napisati i odgovarajuće klase koje se koriste u vozilima (svako vozilo ima svoj računar). Pri pisanju koda ne treba rešavati potencijalni problem izgladnjivanja, već samo maksimizovati protok vozila kroz raskrsnicu. Program pametne raskrsnice se pokreće na računaru čija je adresa "pametnaRaskrsnica" i na raspolaganju ima port 6000.

Rešenje:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Komunikator extends Thread {
    private Socket s;
    private PrintWriter out;
    private BufferedReader in;

    public Komunikator(Socket s) {
        this.s = s;
        try {
            out = new PrintWriter(s.getOutputStream(),true);
            in          =          new          BufferedReader(new
InputStreamReader(s.getInputStream()));
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void run(){
        while(s.isConnected()){
            String zahtev, odgovor;
            try {
                zahtev = in.readLine();
                int i = 0;
                i = Integer.parseInt(zahtev);
                switch (i){
                    case 1:case 3: Main.syn.izStart();
                                   out.println("ok");
                                   odgovor = in.readLine();
                                   Main.syn.izEnd();
                                   out.println("ok");
                                   break;
                    case 2:case 4: Main.syn.sjStart();
                                   out.println("ok");
                                   odgovor = in.readLine();
                                   Main.syn.sjEnd();
                                   out.println("ok");
                                   break;
                }

            } catch (IOException ex) {
                ex.printStackTrace();
            }

        }
```

```
    }
}
//Main.java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Main {

    public Main() {
        iz = 0;
        sj = 0;
    }

    public synchronized void izStart(){
        try {
            while (sj>0)
                wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        iz++;
        return;
    }

    public synchronized void izEnd(){
        iz--;
        if (iz == 0) notifyAll();
        return;
    }

    public synchronized void sjStart(){
        try {
            while (iz>0)
                wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        sj++;
        return;
    }

    public synchronized void sjEnd(){
        sj--;
        if (sj == 0) notifyAll();
        return;
    }

    public static Main syn = new Main();

    private int iz;
    private int sj;

    public static void main(String[] args) {
        ServerSocket ss;
        try {
            ss = new ServerSocket(6000);
            while(true){
                Socket s = ss.accept();
                (new Komunikator(s)).start();
            }

        } catch (IOException ex) {
```

```
                ex.printStackTrace();
            }

        }
    }
```

Sinhronizacija automobila je slična sinhronizaciji prikazanoj kod semafora. Vozilo treba da pošalje kod smjera iz kojeg dolazi (1 – sever, 2 – istok, 3 - jug i 4 - zapad) i zatim da saceka odgovor "ok" (time je dobijena dozvola ulaska u raskrsnicu). Kada zavrsi prolazak treba da pošalje signalnu poruku (sadržaj nebitan) i da sačeka odgovor "ok".

34. (ispit 2011) Sinhronizacija i komunikacija između procesa

Potrebno je implementirati sistem za navođenje aviona. Avion se navodi tako što mu se pred poletanje zadaju adresa i port servera zaduženog za vazdušni prostor u kojem se avion trenutno nalazi. Avionu se zadaje i njegova identifikacija (ceo broj). Avion se povezuje na server nakon čega se serveru predstavi slanjem svoje identifikacije. Posle slanja identifikacije, od servera prihvata poruke koje na početku sadrže tip poruke, zatim dvotačku i nakon toga sadržaj poruke. Poruka može biti jednog od tri moguća tipa. Kada je avion stigao na odredište, poruka je tipa "END" i nema sadržaj. Kada avion prelazi iz jednog vazdužnog prostora u drugi, dobija poruku tipa "NEXT" i kao sadržaj dobija adresu i port (odvojene dvotačkom) servera vazdušnog prostora u koji ulazi. To znači da treba da raskine konekciju sa prethodnim serverom i konektuje se na novi. Čitav ovaj proces se odvija dok avion ne dođe do destinacije. Treći tip poruke je "COMMAND" i u sadržaju nosi komandu koju treba ispisati na standardni izlaz. Za potrebe navođenja aviona, na serverskoj strani je obezbeđena klasa `Navigation` koja sadrži metodu `public string getNextMessage(int id)`. Funkcija prihvata identifikaciju aviona i kada za to dođe vreme, kao rezultat vraća poruku koju treba proslediti avionu. Ova metoda je potpuno bezbedna za pozivanje iz konkurentnih niti (*thread-safe*). Korišćenjem priključnica (*socket*) i mehanizma prosleđivanja poruka (*message passing*), na programskom jeziku Java napisati programe za avion i server.

Rešenje:

```
//server.java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        int port;
        ServerSocket ss;
        port = Integer.parseInt(args[0]);
        try {
            ss = new ServerSocket(port);
            while (true){
                Socket s = ss.accept();
                (new Communicator(s)).start();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }

}
//communicator.java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

public class Communicator extends Thread {
    private Socket s;
    private int id;
```

```
    private BufferedReader in;
    private PrintWriter out;

    private static Navigation = new Navigation();

    public Communicator(Socket s) {
      this.s = s;
      try {
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));
   out = new PrintWriter(new OutputStreamWriter(s.getOutputStream()),true);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void run(){
        try {

            String rec = in.readLine();
            String message;
            id = Integer.parseInt(rec);
            while(s.isConnected()){
                message = getNextMessage(id);
                out.println(message);
                rec = in.readLine();
                while (rec.startsWith("rep") && s.isConnected()) {
                    out.println(message);
                    rec = in.readLine();
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }

}
//avion.java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        String serverAddress = args[0];
        int serverPort, avionId;
        serverPort = Integer.parseInt(args[1]);
        avionId = Integer.parseInt(args[2]);
        System.out.println(serverAddress + serverPort + avionId);
        Socket s = null;
        BufferedReader in = null;
        PrintWriter out = null;
        boolean end = false, connected = false;
        while (!end){
            if (!connected) {
                try {
                    s = new Socket(serverAddress, serverPort);
```

```
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));
   out = new PrintWriter(new OutputStreamWriter(s.getOutputStream()),true);
                    out.println(avionId);
                    connected = true;
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
            if (connected){
                String message;
                try {
                    message = in.readLine();

                    if (message.startsWith("END:")){
                        end = true;
                        in.close();
                        out.close();
                        s.close();
                    } else if (message.startsWith("NEXT:")){
                        try{
                            String newAddress =
                              message.substring(message.indexOf(":")+1);
                            int newPort =
      Integer.parseInt(newAddress.substring(newAddress.indexOf(":")+1));
                            newAddress =
                        newAddress.substring(0,newAddress.indexOf(":"));
                            serverAddress = newAddress;
                            serverPort = newPort;
                            connected = false;
                            in.close();
                            out.close();
                            s.close();
                        }catch(Exception e){
                            e.printStackTrace();
                        }
                    }else if (message.startsWith("COMMAND:")){
                        out.println("ok");

System.out.println(message.substring(message.indexOf(":")+1));
                    } else {
                        System.out.println("Primljena   poruka   nepoznatog
tipa");
                        out.println("rep");
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
}
```

35. (ispit 2011) Sinhronizacija i komunikacija između procesa

Na programskom jeziku Java implementirati klasu koja obezbeđuje mehanizam prosleđivanja poruka sa direktnim simetričnim imenovanjem. Sve operacije treba da budu sinhrone. Smatrati da se poruka sastoji od jednog celog broja, kao i da se klijenti imenuju celim brojevima.

Rešenje:

```
import java.util.HashMap;

public class messageBox {

    protected HashMap data = new HashMap();

    public messageBox() {
    }

    public synchronized void send(int me, int to, int msg){
        data.put(""+me+" "+to, new Integer(msg));
        notifyAll();
        while ( data.containsKey(""+me+" "+to)){
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    public synchronized int receive(int from, int me){
        while ( ! data.containsKey(""+from+" "+me)){
            try {
                wait();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        int ret = ((Integer)data.get(""+from+" "+me)).intValue();
        data.remove(""+from+" "+me);
        notifyAll();
        return ret;
    }

}
```

36. (ispit 2011) Sinhronizacija i komunikacija između procesa

Na programskom jeziku Java implementirati voditelja kviza "Najbrži prsti". Pri kreiranju voditelja zadaju se broj učesnika i port na kojem treba da sačeka da se učesnici uključe u igru. Od tog trenutka režija od voditelja može da zatražiti da svim učesnicima pošalje pitanje i da ih obavesti o ishodu ili da završi kviz. Prilikom slanja pitanja, učesniku se šalje string, nakon čega program koji predstavlja interfejs ka učesniku odgovara sa informacijom da li je učesnik odgovorio ispravno ili ne. Na kraju se od voditelja zahteva da završi kviz, kada svim učesnicima šalje poruku kraj čime se završava kviz. Pošto voditelj nije sposoban da konkurentno komunicira sa svim klijentima, potrebno je implementirati i asistente koji će se izvršavati na serveru i obezbediti opisano ponašanje sistema. Za sinhronizaciju i komunikaciju koristiti priključnice (*sockets*), mehanizam prosleđivanja poruka (*message passing*) i sinhronizovane blokove i metode (*synchronized*).

Rešenje:

```
public class Host {
    public Host(int n, int port) {
        try{

            a = new Assistant[n];
            this.n = n;
            ServerSocket ss = new ServerSocket(port);
            int id = n;
            while (id>0) {
                id--;
                Socket s = ss.accept();
                a[id] = new Assistant(this, s, id);
                a[id].start();
            }
        }catch(Exception e){
            // error
        }
    }

    public synchronized int send(String question){
        int i;
        winner = 0;
        barrier = n;
        for(i = 0; i<n; i++) a[i].send(question);
        try{
            while (barrier > 0) wait();
        }catch(Exception e){
            // error
        }
        return winner;
    }

    public void finish(){
        for(int i = 0; i<n; i++) a[i].send("end");
    }

    public synchronized void answer(int id, boolean tacan){
        if (winner == 0 && tacan) winner = id+1;
        barrier--;
        notifyAll();
    }

    private int winner;
    private int barrier;
```

```
    private Assistant a[];
    private int n;
}
public class Assistant extends Thread{

    public Assistant(Host h, Socket s, int id) {
        this.h = h;
        this.s = s;
        this.id = id;
        try{
            in              =              new              BufferedReader(new
InputStreamReader(s.getInputStream()));
            out              =              new              PrintWriter(new
OutputStreamWriter(s.getOutputStream()));
        }catch(Exception e){
            // error
        }
    }
    public synchronized void send(String question){
        message = question;
        sendReq = 1;
        notifyAll();
    }
    public void run() {
        int rsp;
        while (true){
            synchronized(this){
                try{
                   while (sendReq == 0) wait();
                   sendReq = 0;
                }catch(Exception e){
                    // error
                }
            }
            if (message.equals("Kraj")) {
                //send kraj
                try{
                    out.println("Kraj");
                    out.flush();
                }catch (Exception e){
                    // error
                }
                break;
            }
            try{
                out.println(message);
                out.flush();
                message = in.readLine();
                rsp = Integer.parseInt(message);//receive
                h.answer(id,rsp==1);
            }catch(Exception e){
                // error
            }
        }
    }
    private int sendReq;
    private BufferedReader in;
    private PrintWriter out;
    private Host h;
    private Socket s;
    private int id;
    private String message;
}
```

37. (ispit 2011) Sinhronizacija i komunikacija između procesa

Na programskom jeziku Java implementirati klasu koja uparuje niti. Klasa treba da obezbedi jednu metodu kojoj će nit proslediti svoj ID, i koja će vratiti ID niti sa kojom je pozivajuća nit uparena. Obezbediti da se nit koja je prva izvršila poziv blokira dok se ne pojavi i drugi zahtev, nakon čega se niti uparuju. Redosled uparivanja niti nije bitan (nije neophodno obezbediti da se niti uparuju po redosledu pozivanja tražene metode). Ne sme se desiti da u sistemu postoji više od jednog zahteva, a da su pri tome niti blokirane i da čekaju pristizanje novog zahteva. Za sinhronizaciju koristiti sinhronizovane (*synchronized*) blokove i/ili metode.

Rešenje:

```
public class SyncPlayers {
    int ID1, ID2;
    public SyncPlayers() {
        ID1 = -1;
        ID2 = -1;
    }

    public synchronized int waitPlayer(int ID) throws InterruptedException{
        while (ID1 != -1 && ID2 != -1){
            wait();
        }
        if (ID1 == -1) {
            ID1 = ID;
            notifyAll();
            wait();
            int ret = ID2;
            ID1 = -1;
            ID2 = -1;
            return ret;
        }
        else {
            ID2 = ID;
            int ret = ID1;
            notifyAll();
            return ret;
        }
    }

}
```

38. (ispit 2011) Sinhronizacija i komunikacija između procesa

Dat je interfejs klase `Mailbox` koja apstrahuje „poštansko sanduče“ u indirektnoj razmeni poruka. Operacija `send` asinhrono šalje poruku u sanduče, a operacija `receive` sinhrono uzima jednu poruku iz sandučeta. Poruke se prenose po istom principu kako se prenose argumenti u standardne C funkcije `printf/scanf` (prvi argument je niz znakova koji definiše format preostalih argumenata).

```
class Mailbox {
public:
  void send (char*,...);
  void receive (char*,...);
};
```

Korišćenjem ove klase realizovati funkciju

```
void rendezvous(Mailbox* mbx, int in, int* out);
```

koja implementira randevu svog pozivaoca sa nepoznatim primaocem sa druge strane datog sandučeta, uz slanje argumenta `in` i prijem argumenta `out` indirektno kroz dato sanduče. Radi pojedonstavljenja, pretpostavlja se da je sanduče implementirano tako da nikada ne isporučuje poruku procesu koji je tu poruku poslao.

Rešenje:

```
void rendezvous(Mailbox* mbx, int in, int* out) {
  if (mbx==0) return; // Exception
  mbx->send("%d",in);
  mbx->receive("%d",out);
}
```

1. (Novembar 2006) Upravljanje deljenim resursima

U nekom sistemu semafor, pored standardnih operacija `wait` i `signal`, ima i operaciju `waitNonBlocking`. Ova atomična operacija na semaforu je neblokirajući *wait* i ima sledeće značenje: ako je vrednost semafora veća od 0, ta vrednost se umanjuje za 1 i operacija vraća `true`, a pozivajuća nit nastavlja izvršavanje; inače, pozivajuća nit se ne blokira, vrednost semafora se ne menja, a operacija odmah vraća `false`.

Korišćenjem semafora proširenih ovom operacijom implementirati u potpunosti sledeći algoritam ponašanja filozofa koji večeraju:

```
task type Philosopher
  loop
    think;
    loop
      take_left_fork;
      if can_take_right_fork then
        take_right_fork;
        exit loop;
      else
        release_left_fork;
      end if;
    end;
    eat;
    release_left_fork;
    release_right_fork;
  end;
end;
```

Rešenje ne sme da ima problem utrkivanja (*race condition*) niti problem mrtve blokade (*deadlock*), a problem živog blokiranja (*livelock*) koji je karakterističan za ovaj algoritam ne treba rešavati.

Rešenje:

```
var forks : array [0..4] of semaphore = 1;

task type Philosopher(i:int)
var left, right : 0..4;
begin
  left := i; right:=(i+1) mod 5;
  loop
    think;
    forks[left].wait;
    while not forks[right].waitNonBlocking do
      begin
        forks[left].signal;
        forks[left].wait;
      end;
    eat;
    forks[left].signal;
    forks[right].signal;
  end;
end;
```

2. (Novembar 2006) Upravljanje deljenim resursima

Neki sistem je u stanju prikazanom u tabeli; sva potražnja je za resursima istog tipa. U sistemu se primenjuje tehnika izbegavanja mrtve blokade (*deadlock*).

Proces	Zauzeo	Maksimalno traži
P_0	2	12
P_1	4	10
P_2	2	5
P_3	0	5
P_4	2	4
P_5	1	2
P_6	5	13
Slobodnih: 1		

a)(5) Da li je ovaj sistem u bezbednom ili nebezbednom stanju? Detaljno obrazložiti odgovor.

Odgovor:

Jeste, jer postoji sigurna sekvenca (po bankarevom algoritmu): P_5, P_4, P_2, P_3, P_1, P_0, P_6.

b)(5) Da li sistem u datom stanju treba da dozvoli alokaciju jedinog preostalog slobodnog resursa procesu P_6 ako ga ovaj zatraži? Detaljno obrazložiti odgovor.

Odgovor:

Ne treba, jer bi tada sistem ušao u nebezbedno stanje, za koje ne bi postojala sigurna sekvenca (trivijalno, pošto nema slobodnih resursa, a svi procesi drže zauzeto manje od svoje maksimalne potražnje).

3. (Decembar 2006) Upravljanje deljenim resursima

Data je sledeća implementacija ograničenog bafera pomoću klasičnih brojačkih semafora, čiji je uzor klasično rešenje pomoću uslovnih promenljivih u monitorima:

```
class Data;
const int SIZE = ...;

class BoundedBuffer {
public:
  BoundedBuffer ();
  void  put (Data*);
  Data* get ();

private:
  Data* rep[SIZE];
  int head, tail, size;
  Semaphore mutex, spaceAvailable, itemAvailable;
};

BoundedBuffer::BoundedBuffer ()
    : mutex(1), spaceAvailable(0), itemAvailabe(0),
      head(0), tail(0), size(0) {}

void BoundedBuffer::put (Data* d) {
  mutex.wait();
  while (size==SIZE) {
    mutex.signal();
    spaceAvailable.wait();
    mutex.wait();
  }
  rep[tail++]=d;
  if (tail==SIZE) tail=0;
  size++;
  if (itemAvailabe.value()<0) itemAvailable.signal();
  mutex.signal();
}

Data* BoundedBuffer::get () {
  mutex.wait();
  while (size==0) {
    mutex.signal();
    itemAvailable.wait();
    mutex.wait();
  }
  Data* d = rep[head++];
  if (head==SIZE) head=0;
  size--;
  if (spaceAvailabe.value()<0) spaceAvailable.signal();
  mutex.signal();
}
```

Šta je problem ovog rešenja?

Odgovor:

Problem datog rešenja je što postoji utrkivanje (*race condition*) jer operacije oslobađanja ulaza u monitor (`mutex.signal()`) i blokiranja na semaforu za uslovnu sinhronizaciju (`spaceAvailable.wait()` i `itemAvailable.wait()`) nisu nedeljive. Zbog toga moće doći do sledećeg neregularnog scenarija i gubitka sinhronizacije:
- proizvođač pokušava da smesti element u pun bafer, izvršava prve tri linije koda operacije `put()`, zaključno sa `mutex.signal()`

- dešava se preuzimanje, potrošač izvršava celu operaciju `get()`, uzima jedan element iz bafera, oslobađa mesto, ali pošto se proizvođač još nije blokirao na semaforu `spaceAvailable`, ne izvršava operaciju `signal` na ovom semaforu (jer je njegova vrednost i dalje nula)
- proizvođač nastavlja izvršavanje i blokira se na `spaceAvailable.wait()` bez razloga, potencijalno večno.

4. (Novembar 2007) Upravljanje deljenim resursima

U nekom sistemu semafori su implementirani sa redovima čekanja uređenim po prioritetu procesa, tako da su procesi koji su blokirani na semaforu uređeni po prioritetu njihovog trenutnog izvršavanja. Da li korišćenje ovakvih semafora u svrhu kontrole pristupa deljenom resursu (tj. za međusobno isključenje) obezbeđuje živost (*liveness*)? Ako ne obezbeđuje, navesti koji problem narušava živost i zbog čega.

Odgovor:

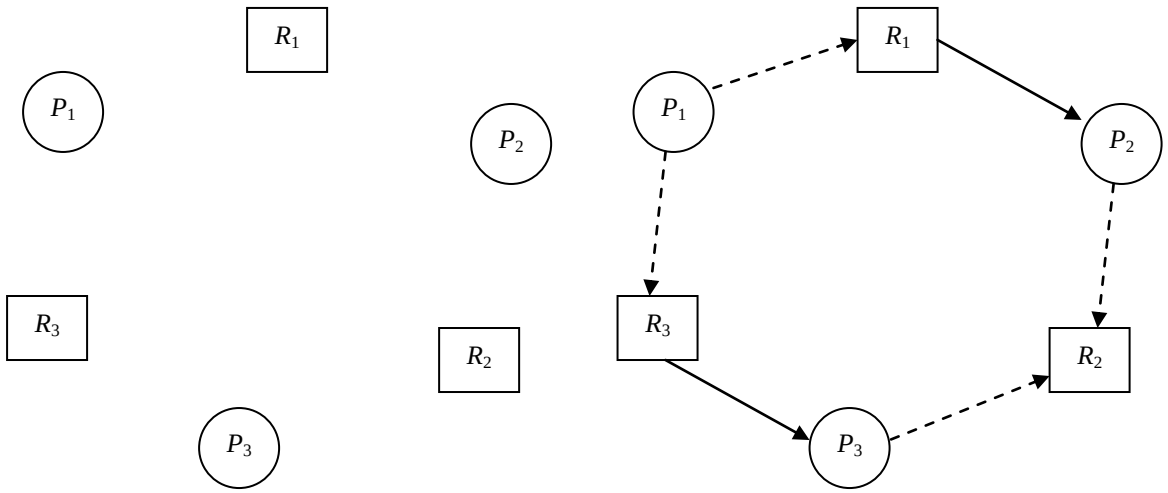
Ne obezbeđuje živost, jer je moguće izgladnjivanje (*starvation*) procesa koji čeka na semaforu (odnosno na pristup deljenom resursu). Proces nižeg prioriteta koji čeka na pristup resursu može da bude pretican od strane procesa višeg prioriteta koji zahtevaju isti resurs i zato pre do resursa dolaze, pošto se u red čekanja na semaforu smeštaju ispred pa se i deblokiraju ranije. Na taj način proces nižeg prioriteta može neograničeno da čeka na pristup resursu. Drugim rečima, ovakav semafor ne obezbeđuje „poštenost“ (*fairness*).

5. (Novembar 2007) Upravljanje deljenim resursima

U nekom sistemu implementiran je algoritam izbegavanja mrtvog blokiranja (*deadlock*) 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:

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

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 bezbedno?

Odgovor i obrazloženje:

Jeste, jer nema petlje u grafu, odnosno postoji sigurna sekvenca, npr. P_3, P_2, P_1.

c)(3) Ako u datom stanju proces P_2 zatraži korišćenje resursa R_2, da li sistem treba da mu ga dodeli?
Odgovor i obrazloženje:

Treba, jer bi novonastalo stanje i dalje bilo bezbedno: ne postoji petlja u grafu, odnosno postoji sigurna sekvenca, npr. P_2, P_3, P_1.

6. (Novembar 2008) Upravljanje deljenim resursima

Posmatra se sledeća implementacija mehanizma optimističkog zaključavanja (bez blokiranja) u cilju postizanja međusobnog isključenja pristupa više uporednih procesa deljenom podatku. Svaki proces najpre atomično pročita i sačuva vrednost deljenog podatka, zatim kreće u izmenu svoje kopije deljenog podatka bez ikakve sinhronizacije sa ostalim takvim procesima, a potom atomično radi sledeću operaciju: proverava identičnost svoje sačuvane kopije početne vrednosti deljenog podatka i njegove tekuće vrednosti i ako su one iste, upisuje svoju izmenjenu verziju, u suprotnom ova operacija vraća neuspeli status. Ukoliko je ova operacija završena neuspešno, proces se „povlači“ u „besposleno“ stanje (engl. *idle*) slučajno vreme, a potom pokušava ceo navedeni postupak ispočetka. Koji problem (ili problemi) postoje u ovakvom pristupu? Precizno obrazložiti kako dati problem/problemi može/mogu da nastane/nastanu.

Odgovor:

Postoji problem potencijalnog izgladnjivanja (*starvation*) koji nastaje kada neki proces mora neograničeno da ponavlja navedeni postupak, pošto se uvek dešava da neki drugi proces učita istu vrednost, ali neposredno pre posmatranog procesa izmeni deljeni podatak, posmatrani proces se povuče i prilikom ponovnog pokušaja dogodi se isto.
Treba primetiti da problem žive blokade (*livelock*) ne postoji, pošto od više procesa koji pročitaju istu početnu vrednost, uvek će jedan, zbog sekvencijalnosti (odnosno atomičnosti) operacije provere i upisa nove vrednosti, uspešno završiti postupak (prvi koji je ušao u tu operaciju, pošto zatiče pročitanu vrednost), dok će ga ostali ponoviti.

7. (Novembar 2008) Upravljanje deljenim resursima

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:

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

U sistemu se primenjuje bankarev algoritam izbegavanja mrtvog blokiranja. Da li sistem treba da dozvoli zauzeće još 2 instance resursa tipa C od strane procesa P_2? Precizno obrazložiti odgovor, uz navođenje svih koraka primene bankarevog algoritma.

Rešenje:
Ako bi sistem dozvolio zauzeće traženih resursa, prešao bi u stanje:

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

Treba ispitati da li je ovo stanje sigurno pronalaženjem sigurne sekvence:

P_1,

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

P_1, P_4,

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

P_1, P_4, P_3,

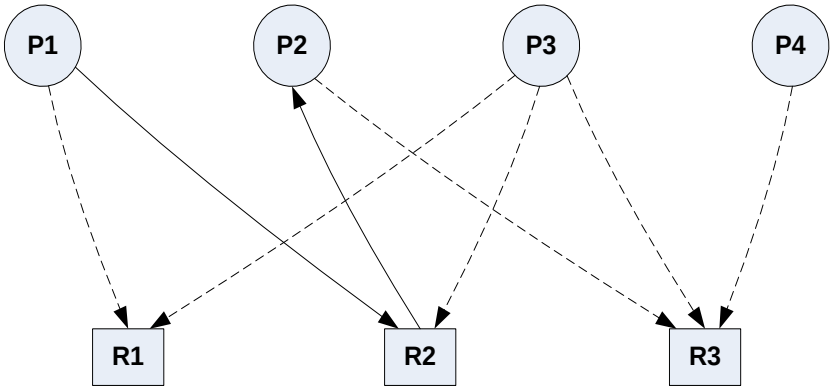
	Allocation			*Max*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
							4	8	5
P_2	1	3	2	5	4	3			

P_1, P_4, P_3, P_2

Pošto je sigurna sekvenca pronađena, odredišno stanje je sigurno, pa sistem može da dozvoli traženo zauzeće resursa.

8. (Oktobar 2009) Upravljanje deljenim resursima

U nekom sistemu primenjuje se mehanizam izbegavanja mrtve blokade (*deadlock*) zasnovan na grafu alokacije. Na slici je prikazan graf alokacije resursa za posmatrano stanje sistema. Precizno navesti koji sve procesi u ovom stanju mogu da zatraže koje pojedinačne resurse i koje od tih zahteva sistem treba da ispuni alokacijom resursa, a koje ne (odgovor datu u vidu niza iskaza oblika „proces - resurs - Da ili Ne“).



Odgovor:

P2 - R3 - Da, P3-R1-Da (prelaz u bezbedno stanje), P3-R2-Ne (već zauzet), P3-R3-Ne (prelaz u nebezbedno stanje), P4-R3-Da (prelaz u bezbedno stanje).

9. (Oktobar 2009) Upravljanje deljenim resursima

Predložite protokol davanja dozvole za operaciju kod protokola više čitalaca - jedan pisac koji ne izgladnjuje ni jedne ni druge u slučaju da postoje i čitaoci i pisci koji čekaju da izvrše svoju operaciju.

Rešenje:

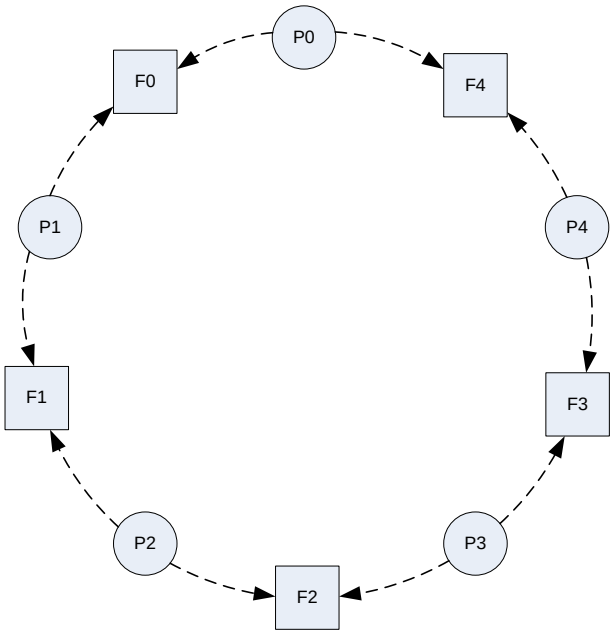
Na primer, puštati naizmenično jednog pisca pa jednog ili nekoliko čitalaca u slučaju da postoje i jedni i drugi koji čekaju, ali ne insistirati na naizmeničnosti ukoliko ne postoje oni koji čekaju.

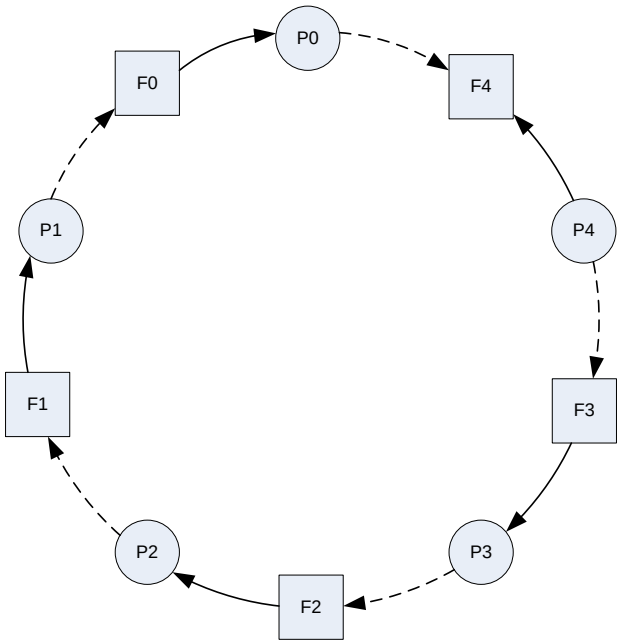
10. (Oktobar 2010) Upravljanje deljenim resursima

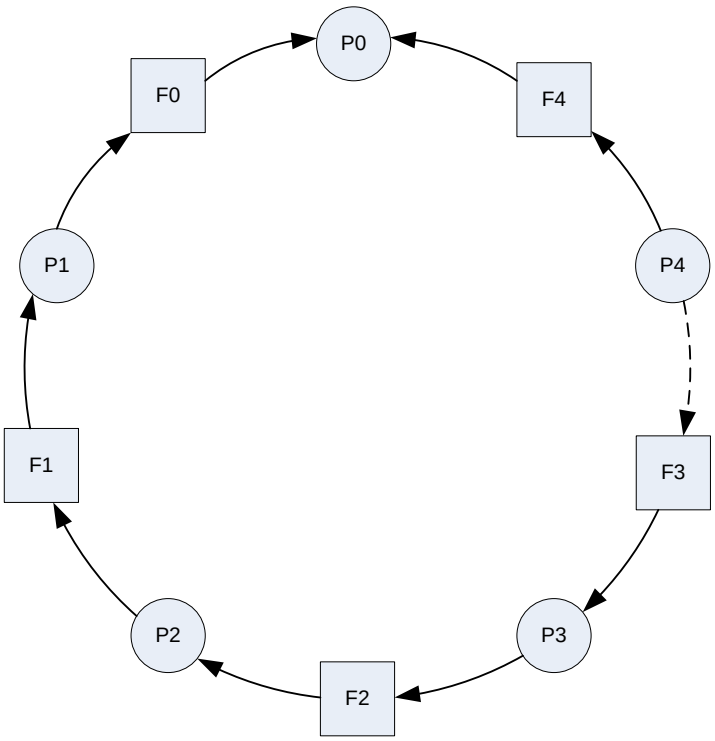
Na problemu filozofa koji večeraju (*dining philosophers*) demonstrira se mehanizam izbegavanja mrtve blokade (*deadlock avoidance*) zasnovan na grafu alokacije. Svaki filozof traži najpre svoju desnu viljušku, pa kada nju dobije, traži i svoju levu viljušku. Potrebno je prikazati graf alokacije za svako 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:

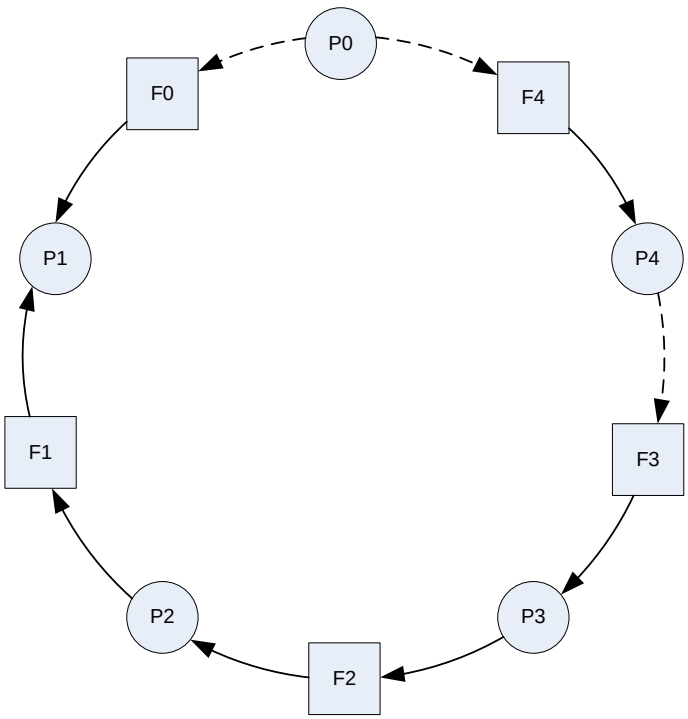
a)(2) kada svi filozofi razmišljaju, potom
b)(3) nakon što su svi filozofi zatražili svoju desnu viljušku, potom
c)(2) nakon što su svi oni filozofi koji su dobili svoju desnu viljušku, zatražili i svoju levu viljušku, potom
d)(3) nakon što su svi oni koji su dobili obe viljuške završili sa jelom.

Rešenje:






P0
F0
F4
P1
P4
F1
F3
P2
P3
F2


P0
F0
F4
P1
P4
F1
F3
P2
P3
F2

11. (Oktobar 2010) Upravljanje deljenim resursima

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.

Odgovor:

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: $i_1 < i_2 < \ldots < i_n < i_1$, 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.

12. (Ispit Januar 2011) Upravljanje deljenim resursima

Na problemu filozofa koji večeraju (*dining philosophers*) demonstrira se mehanizam izbegavanja mrtve blokade (*deadlock avoidance*) zasnovan na bankarevom algoritmu (*banker's algorithm*). Svaki filozof traži najpre svoju levu viljušku, pa kada nju dobije, traži i svoju desnu viljušku. Potrebno je prikazati matricu alokacije i vektor slobodnih resursa za svako dole navedeno stanje, tim redom. U svakom traženom stanju strukture prikazati nakon što je sistem dodelio resurse svima kojima su ih tražili, a kojima se mogu dodeliti resursi. Prikazati date strukture za sledeća stanja:

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

a)(2)

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
Av.	1	1	1	1	1

b)(3)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	0
Av.	0	0	0	0	1

c)(2)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	1
5	0	0	0	0	0
Av.	0	0	0	0	0

d)(3)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	1	0
4	0	0	0	0	0
5	0	0	0	0	1
Av.	0	0	0	0	0

13. (Oktobar 2011) Upravljanje deljenim resursima

U nekom specijalizovanom sistemu proces može biti „poništen“ (rolled back) – ugašen uz poništavanje svih njegovih efekata, i potom pokrenut ispočetka. U ovom sistemu primenjuje se sledeći algoritam sprečavanja mrtve blokade (deadlock prevention). Svakom procesu se, prilikom kreiranja, dodeljuje jedinstven identifikator (ID) 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 j poništava i pokreće ponovo.

a) (5) Dokazati da se ovim algoritmom sprečava mrtva blokada.
b) (5) Koji ID treba dodeliti poništenom procesu P j kada se on ponovo pokrene, da bi ovaj algoritam sprečio izgladnjivanje (starvation)? Obrazložiti.

Odgovor:

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: $i_1 > i_2 > \ldots > i_n > i_1$, što ne može biti, pa mrtva blokada ne može nastati.

b)(5) Prema uslovima algoritma, ako stariji proces zatraži resurs koga drži neki mlađi 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.

14. (Septembar 2012) Upravljanje deljenim resursima

U nekom sistemu su tri procesa (P_1, P_2, P_3) i tri različite instance resursa (R_1, R_2, R_3). Ne primenjuje se izbegavanje mrtve blokade, već se samo prati zauzeće resursa pomoću grafa, a resurs dodeljuje procesu čim je slobodan. Procesi su izvršili sledeće operacije datim redosledom:

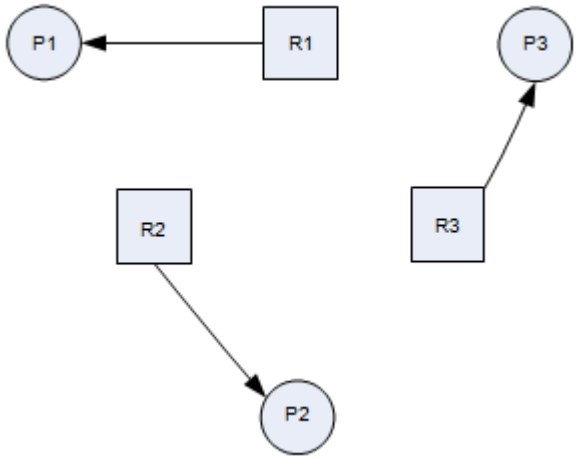
P2.request(R3), P1.request(R1), P2.request(R2), P3.request(R3), P2.release(R3)

a)(5) Nacrtati graf zauzeća resursa nakon ove sekvence.

b)(5) Navesti neki nastavak ove sekvence koji sistem vodi u mrtvu blokadu (*deadlock*).

Rešenje:

a)



b)(5) Na primer: P1.request(R2), P2.request(R3), P3.request(R1)

15. (Novembar 2012) Upravljanje deljenim resursima

U nekom sistemu svaki proces i svaki resurs ima svoj jedinstveni identifikator (tip a `unsigned int`), a zauzeće resursa od strane procesa prati se u matrici `resourceAlloc` u kojoj vrste označavaju procese, a kolone resurse. U toj matrici vrednost -1 u ćeliji (*p*, *r*) označava da je proces sa identifikatorom *p* zauzeo resurs sa identifikatorom *r*, vrednost 1 označava da je proces *p* tražio, ali nije dobio resurs *r* (i čeka da ga dobije), a vrednost 0 označava da proces *p* nije ni zauzeo ni tražio resurs *r*. Implementirati operacije `allocate()` i `release()` koje procesi treba da pozivaju kada traže, odnosno oslobaÿaju resurse.

```
const unsigned MAXPROC = ...; // Maximum number of processes
const unsigned MAXRES = ...;  // Maximum number of resources
extern unsigned numOfProc;    // Actual number of processes
extern unsigned numOfRes;     // Actual number of resources

int resourceAlloc[MAXPROC][MAXRES];

int  allocate  (unsigned  pid,  unsigned
rid);
int release (unsigned pid, unsigned rid);
```

Operacija `allocate()` vraća 1 ako je traženi resurs dodeljen datom procesu, a 0 ako nije. Operacija `release()` dodeljuje osloboÿeni resurs nekom drugom (bilo kom) procesu koji je
čekao na taj resurs i vraća njegov identifikator, ako takav proces postoji; u suprotnom, samo oslobaÿa resurs i vraća -1. Ne primenjuje se nikakav algoritam sprečavanja, izbegavanja, ili
detekcije mrtve blokade.

Rešenje:

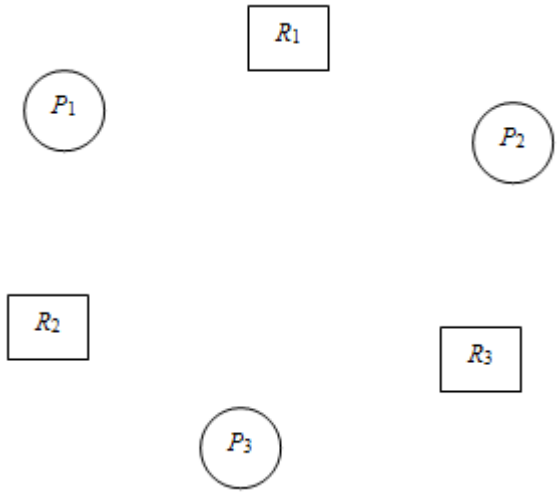
```
int allocate (unsigned int pid, unsigned int rid) {
  if (pid>=numOfProc || rid>=numOfRes) return -1; // Exception!
  // Is this resource free?
  for (unsigned int i=0; i<numOfProc; i++)
  if (i!=pid && resourceAlloc[i][rid]==-1) {
      resourceAlloc[pid][rid]==1;
      return 0;
    }
  // The resource can be allocated: resourceAlloc[pid][rid] = -
  1;
  return 1;
}
// The resource is occupied

int release (unsigned int pid, unsigned int rid) {
  if (pid>=numOfProc || rid>=numOfRes) return -2; // Exception!
  resourceAlloc[pid][rid] = 0;
  // Find a waiting process:
  for (unsigned int i=0; i<numOfProc; i++)
    if (resourceAlloc[i][rid]==1) {
      resourceAlloc[i][rid]==-1;
      return i;
    }
  return -1;
}
```

16. (ispit 2006) Upravljanje deljenim resursima

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 resura istog tipa:

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

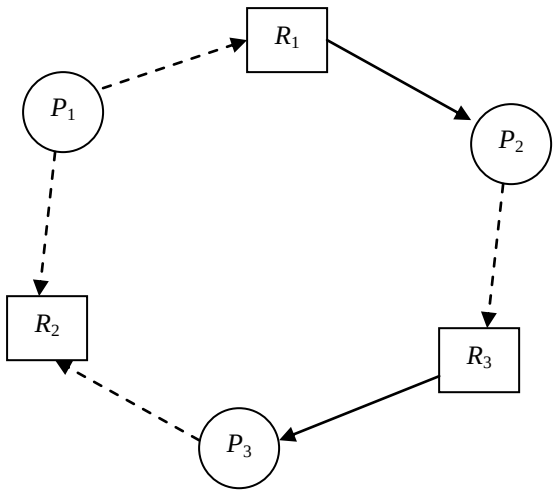
a)(4) Nacrtati graf alokacije resursa za navedeno stanje.



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?

a)(4)



b)(3) Jeste, jer postoji sigurna sekvenca P_3, P_2, P_1.

c)(3) Ne treba, jer bi novonastalo stanje bilo nesigurno (može dovesti do mrtve blokade ako sva tri procesa zatraže nealocirani resurs) – postoji petlja u grafu

17. (ispit 2006) Upravljanje deljenim resursima

a)(5) Jedan proces *A* je zauzeo semafor *S* (izvršivši *wait* na tom semaforu i postavivši ga na 0) i čeka da mu bude dodeljen procesor (kao resurs) da bi nastavio svoje izvršavanje. Drugi proces *B*, koji se izvršava (koristi procesor kao resurs), zadaje operaciju *wait* na semaforu *S*. Koji od neophodnih uslova za nastanak mrtve blokade (*deadlock*) ovde nije ispunjen, pa zbog toga mrtva blokada i ne nastaje? Precizno obrazložiti odgovor.

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 resura 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.

Odgovor:

a)(5) Nije ispunjen uslov „nema preotimanja“ (*no preemption*): kada proces *B* ne može da „zauzme“ semafor kao resurs, operativni sistem mu oduzima procesor i dodeljuje drugom procesu, u ovom slučaju, procesu *A*. Zbog toga takođe nije ispunjen ni uslov „držanje i čekanje“ (*hold and wait*), jer proces koji „drži“ semafor neće više čekati na procesor. Ovo je i primer koji pokazuje da neophodni uslovi za nastanak mrtve blokade (*deadlock*) nisu sasvim nezavisni.

b)(5) Ako bi sistem dozvolio zauzimanje jednog resursa od strane procesa P_1, sistem bi prešao u sledeće stanje:

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

Ovo stanje je bezbedno, jer postoji sigurna sekvenca: P_3, P_1, P_2, pa sistem može da dodeli traženi resurs.

18. (ispit 2006) Upravljanje deljenim resursima

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.

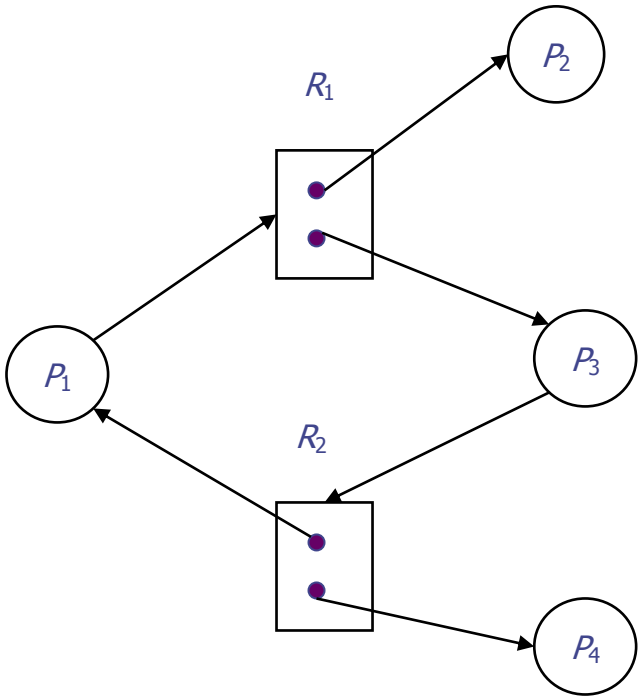
a)(5) Postoji problem izgladnjivanja (*starvation*): ukoliko pristigne novi zahtev istog tipa kao i onaj koji se trenutno obrađuje, i to se stalno dešava (ili se dešava u jako dugom vremenskom intervalu), zahtevi drugog tipa ne bivaju opsluženi.

b)(5) 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.

19. (ispit 2006) Upravljanje deljenim resursima

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.

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_2 moći da nastavi izvršavanje. P_2 svakako nije blokiran.

20. (ispit 2006) Upravljanje deljenim resursima

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:

	Allocation			*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.

Rešenje:

Ako bi sistem dozvolio zauzeće traženih resursa, prešao bi u stanje:

	Allocation			*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			

Treba ispitati da li je ovo stanje sigurno pronalaženjem sigurne sekvence:

P_3,

	Allocation			*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_4	2	3	1	3	4	5			

P_3, P_2,

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

P_3, P_2, P_1,

	Allocation			*Max*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
							5	8	4
P_4	2	3	1	3	4	5			

P_3, P_2, P_1, P_4

Pošto je sigurna sekvenca pronađena, odredišno stanje je sigurno, pa sistem može da dozvoli traženo zauzeće resursa.

21. (ispit 2007) Upravljanje deljenim resursima

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.

Rešenje:

```
extern int x, y;
extern ReadersWriters* xGuard;
extern ReadersWriters* yGuard;
xGuard->startRead();
int xTemp = x;           // Alternativno rešenje: bez ovog reda i...
yGuard->startWrite();
y = xTemp + 1;           // ... ovde: y = x + 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.

Odgovor:

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.

22. (ispit 2007) Upravljanje deljenim resursima

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.

Rešenje:

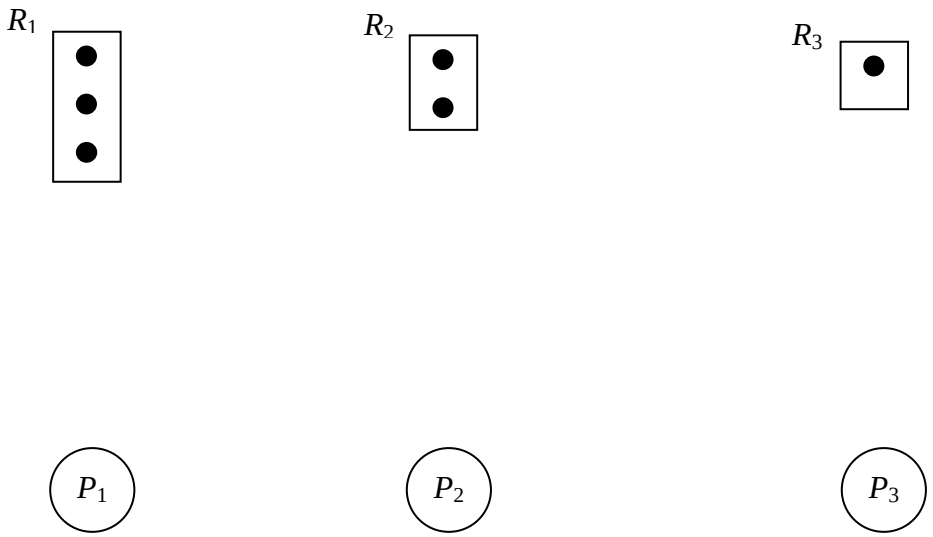
Ne mogu, jer nije ispunjen neophodan uslov „cirkularno držanje i čekanje“ (*circular wait*). Naime, jedino proces C može doći u stanje „držanja i čekanja“ (*hold and wait*), koje je neophodno za nastanak mrtve blokade, kada zauzme resurs X a eventualno čeka na resurs Y. Da bi postojala mrtva blokada, 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.

23. (ispit 2007) Upravljanje deljenim resursima

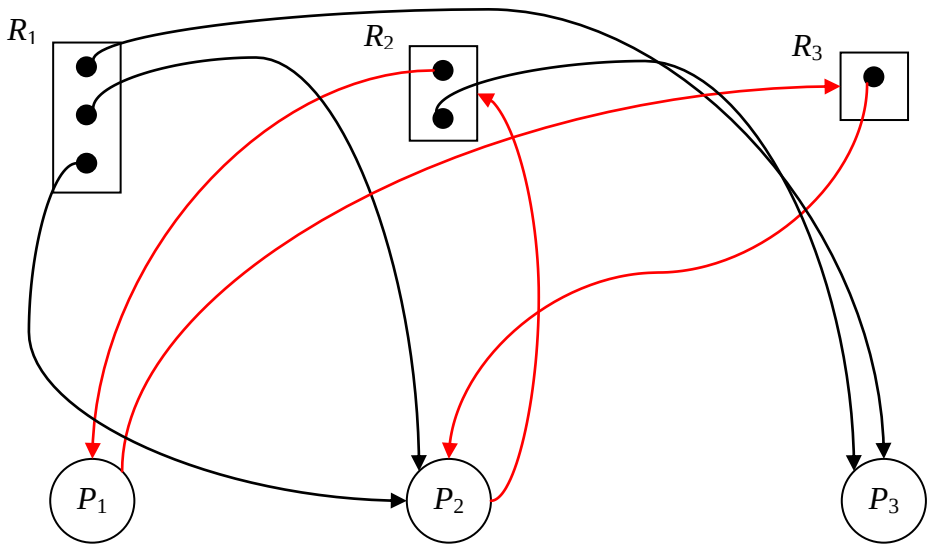
U nekom sistemu postoje resursi tri tipa, sa datim brojem instanci: R_1 – 3, R_2 – 2, R_3 – 1. Nacrtati graf zauzetosti resursa posle sledeće sekvence zauzimanja po jedne instance resursa datog tipa (oznake P_i označavaju procese) i odgovoriti na pitanje da li je sistem tada u mrtvoj blokadi. Precizno obrazložiti odgovor.

P_1-R_2, P_2-R_3, P_3-R_1, P_2-R_1, P_1-R_3, P_2-R_1, P_3-R_2, P_2-R_2

Rešenje:



Odgovor:



Sistem nije u mrtvoj blokadi, iako postoji petlja (označena crvenim granama). Naime, proces P_3 nije blokiran i može da završi svoje izvršavanje, čime će osloboditi svoje resurse, pa i R_2 koji drži. Oslobađanjem jedne instance R_2 proces P_2 može da je zauzme i time se deblokira, a grana imeđu P_2 i R_2 menja smer i postaje $R_2 \rightarrow P_2$. Tada u grafu više nema petlji, jer iz P_2 više ne izlazi ni jedna grana.

24. (ispit 2007) Upravljanje deljenim resursima

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.

Odgovor:

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 (por 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.

25. (ispit 2007) Upravljanje deljenim resursima

Modifikovati klasu `ReadersWriters` za kontrolu pristupa deljenim resursima po principu *multiple readers-single writer* datu na predavanjima, tako da propušta najviše *N* čitalaca (*N*>1). (I dalje treba favorizovati čitaoce, u smislu da treba puštati nove čitaoce ako neki već čitaju.)

```
class ReadersWriters {
public:
  ReadersWriters ();

  void startRead ();
  void stopRead ();
  void startWrite ();
  void stopWrite ();

private:
  Semaphore mutex, wrt, readers;
  int readcount;
};

ReadersWriters::ReadersWriters ()
    : mutex(1), wrt(1), readers(N), readcount(0) {}

void ReadersWriters::startWrite () {
  wrt.wait();
}

void ReadersWriters::stopWrite () {
  wrt.signal();
}

void ReadersWriters::startRead () {
  readers.wait();
  mutex.wait();
  readcount++;
  if (readcount==1) wrt.wait();
  mutex.signal();
}

void ReadersWriters::stopRead () {
  mutex.wait();
  readcount--;
  if (readcount==0) wrt.signal();
  mutex.signal();
  readers.signal();
}
```

26. (3. januar 2008.) Upravljanje deljenim resursima

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;
```

Odgovor:

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.

27. (3. februar 2008.) Upravljanje deljenim resursima

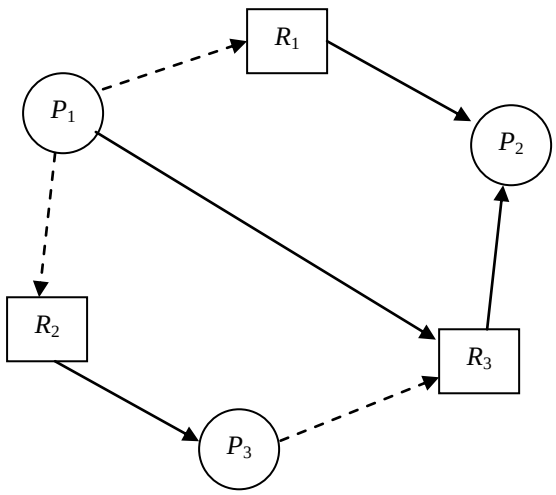
U nekom sistemu implementiran je algoritam izbegavanja mrtvog blokiranja zasnovan na analizi bezbednih stanja pomoću grafa alokacije. U sistemu postoji samo po jedna instanca resursa R_1, R_2 i R_3, a pokrenuti su procesi P_1, P_2 i P_3. U nekom trenutku sistem se nalazi u sledećem stanju zauzetosti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

Proces	Zauzeo	Zatražio (ali čeka, nije dobio)	Potencijalno traži (najavio korišćenje)
P_1		R_3	R_1, R_2
P_2	R_1, R_3		
P_3	R_2		R_3

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 bezbedno?
c)(3) Ako u datom stanju proces P_2 oslobodi resurs R_3, da li sistem treba taj resurs da odmah dodeli procesu P_1 koji ga je već zatražio?

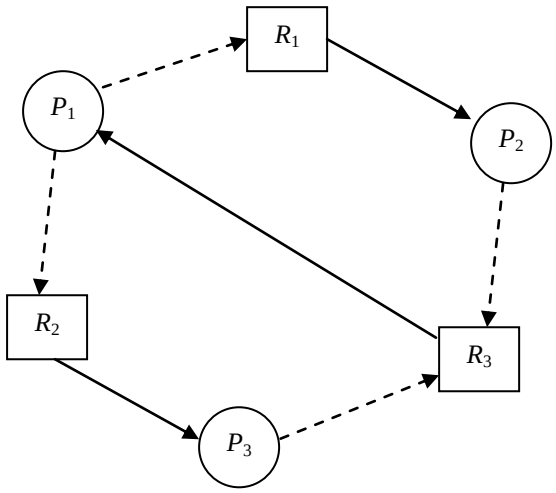
Odgovor:

a)



b) Jeste, jer postoji sigurna sekvenca P_2, P_3, P_1.

c) Ne treba, jer bi novonastalo stanje bilo nebezbedno (može dovesti do mrtve blokade ako potom P_1 zatraži R_1, a P_2 ponovo R_3) – postoji petlja u grafu koji bi tada izgledao ovako:


R_1
P_1
P_2
R_2
R_3
P_3

28.(3. jun 2008.) Upravljanje deljenim resursima

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.

Odgovor:

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.

29. (ispit 2009) Upravljanje deljenim resursima

Za tehniku izbegavanja mrtve blokade navesti koje tvrdnje su tačne, a koje netačne:

a) Ako se sistem trenutno nalazi u nebezbednom stanju, sigurno će ući u mrtvu blokadu.
b) Ako se sistem trenutno nalazi u nebezbednom stanju, sigurno neće ući u mrtvu blokadu.
c) Ako se sistem trenutno nalazi u bezbednom stanju, sigurno može izbeći mrtvu blokadu.
d) Ako se sistem trenutno nalazi u nebezbednom stanju, sigurno može izbeći mrtvu blokadu.

Zaokružiti odgovor:

a) Tačno Netačno
b) Tačno Netačno
c) Tačno Netačno
d) Tačno Netačno

a) Netačno
b) Netačno
c) Tačno
d) Netačno

30. (ispit 2009) Upravljanje deljenim resursima

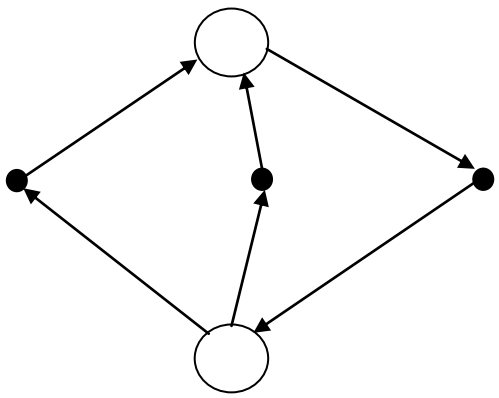
Data su dva procesa, *P* i *Q*, koji koriste deljene resurse *A*, *B* i *C* na sledeći način:

```
process P;                    process Q;
...                           ...
request(A,B);                 request(C);
...                           ...
request(C);                   request(A,B);
...                           ...
release(A);                   release(A,C);
...                           ...
release(B);                   release(B);
...                           ...
release(C);                   end;
...
end;
```

a)(5) Dati jedan scenario izvršavanja u kome ovi procesi ulaze u stanje mrtve blokade (*deadlock*) i nacrtati graf zauzeća resursa u tom stanju.

b)(5) Restrukturirati ove procese tako da se *spreči* mrtva blokada ukidanjem neophodnog uslova „držanje i čekanje“.

a)(5) `P:request(A,B), Q:request(C), P:request(C), Q:request(A,B)`



b)(5)

```
process P;                    process Q;
...                           ...
request(A,B,C);               request(A,B,C);
...                           ...
release(A);                   release(A,C);
...                           ...
release(B);                   release(B);
...                           ...
release(C);                   end;
...
end;
```

31. (ispit 2009) Upravljanje deljenim resursima

Posmatra se optimistički pristup međusobnom isključenju operacija upisa u deljeni podatak od strane konkurentnih procesa (isključenje bez blokiranja i zaključavanja). Ukoliko je ovaj pristup ispravno implementiran, koji od problema mogu da nastupe u opštem slučaju: utrkivanje (*race condition*), živa blokada (*livelock*), mrtva blokada (*deadlock*), izgladnjivanje (*starvation*). Precizno obrazložiti odgovor.

Odgovor:

Samo izgladnjivanje (*starvation*). Utrkivanje (*race condition*) ne može da nastupi pod pretpostavkom da je pristup ispravno implementiran, jer on tada obebzbeđuje međusobno isključenje. Živa blokada (*livelock*) ne može da nastupi, jer nema neograničenog odlaganja: uvek jedan (i samo jedan) proces od proizvoljno mnogo njih koji su uporedo ušli u kritičnu sekciju uspešno iz nje i izlazi; to je onaj koji je prvi izmenio podatak i upisao u njega svoju verziju; ostali moraju da ponove svoju operaciju iznova. Prema tome, nema neograničenog odlaganja svih procesa koji pristupaju resursu, jer jedan po jedan pristupaju resursu u konačnom vremenu. Mrtva blokada (*deadlock*) ne može da nastupi jer nema blokiranja, kao ni neograničenog zadržavanja. Izgladnjivanje (*starvation*) može da nastupi, ukoliko nije sprečeno dodatnim mehanizmom, jer uvek novi procesi mogu da pretiču dati proces i upisuju pre njega, pa posmatrani proces, zbog stalnih konflikata, ne uspeva da uspešno završi svoj pristup neograničeno.

32. (ispit 2009) Upravljanje deljenim resursima

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?

Odgovor:

) 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.

33. (ispit 2010) Upravljanje deljenim resursima

U problemu filozofa koji večeraju (engl. *dining philosophers*), svaki filozof atomično uzima obe svoje viljuške odjednom. U ovom pristupu sigurno nije moguća mrtva blokada (engl. *deadlock*). Precizno objasniti zbog čega.

Odgovor:

Nije zadovoljen neophodan uslov držanja i čekanja (*hold and wait*): svaki proces uzima sve svoje resurse odjednom (obe viljuške odjednom), pa ne postoji situacija kada neki proces drži neki svoj resurs, a čeka da zauzme neki drugi.

34. (ispit 2010) Upravljanje deljenim resursima

U nekom specijalizovanom sistemu proces se može „poništiti“ (*roll back*) – ugasiti uz poništavanje svih njegovih efekata na sistem, i potom pokrenuti ispočetka. U ovom sistemu primenjuje se sledeći algoritam sprečavanja mrtve blokade (*deadlock*). 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 (sa istim ID koji je prethodno imao).

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

b)(5) Dokazati da ovaj algoritam ima problem izgladnjivanja (*starvation*).

Odgovor:

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: $i_1>i_2> \ldots >i_n>i_1$, što ne može biti, pa mrtva blokada ne može nastati.

b)(5) Prema uslovima algoritma, ako stariji proces zatraži resurs koga drži neki mlađi proces, stariji proces se poništava i pokreće ponovo. Kada taj stariji proces ponovo zatraži taj isti resurs, možda je taj resurs zauzeo neki drugi mlađi proces, pa se ovaj stariji ponovo poništava. Vremenom taj proces postaje sve stariji i moguće je da ga novi i sve mlađi procesi izgladnjuju.

35. (ispit 2010) Upravljanje deljenim resursima

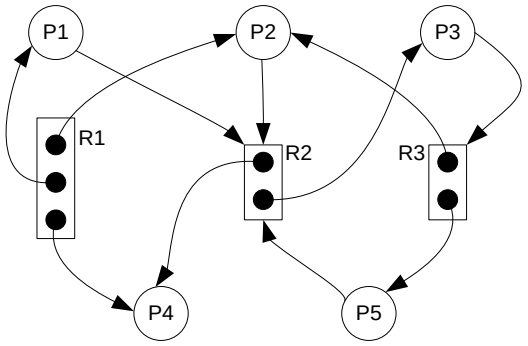
U problemu filozofa koji večeraju (engl. *dining philosophers*), svi filozofi uzimaju atomično najpre jednu, pa onda (u drugoj atomičnoj akciji) drugu viljušku, ne ispuštajući onu prvu dok ne uzmu i drugu, s tim da četvorica od njih uzimaju najpre levu, pa desnu svoju viljušku, a peti to radi obratno, najpre svoju desnu, pa levu. Da li je u ovom sistemu moguće izgladnjivanje ili mrtva blokada (engl. *deadlock*)? Obrazložiti odgovor.

Odgovor:

Nije moguće ni izgladnjivanje ni mrtva blokada. Mrtva blokada nije moguća jer nije moguće ostvariti kružnu zavisnost, pošto je peti filozof raskida. Izgladnjivanje nije moguće jer filozof ne ispušta viljušku koju je već uzeo, dok ne uzme i drugu.

36. (ispit 2010) Upravljanje deljenim resursima

Na slici je prikazan graf zauzeća resursa nekog sistema u nekom trenutku. Da li je ovaj sistem u mrtvoj blokadi (*deadlock*)? Precizno obrazložiti odgovor.



Odgovor:

Dati sistem nije u mrtvoj blokadi. Graf sadrži dve petlje: P2-R2-P3-R3 i P3-R3-P5-R2. Međutim, proces P4 ne čeka ni na jedan resurs, pa može da nastavi izvršavanje i oslobodi resurse koje drži. Tada se oslobođeni resurs R2 može dodeliti nekom od procesa koji čekaju na R2. Ako je to P1, i on će, nakon konačnog vremena, ponovo osloboditi taj R2. Dalje, taj R2 se može dodeliti nekom od P2 ili P5, čime se raskida jedna od dve petlje. Nakon završetka tog procesa, ponovo se oslobađa jedan R2 i raskida se i druga petlja.

37. (ispit 2010) Upravljanje deljenim resursima

Neki programski sistem sastoji se od više procesa, pri čemu svaki od tih procesa koristi najviše po jedan resurs u svakom trenutku, odnosno oslobađa prethodno zauzeti resurs pre nego što zatraži neki novi. Da li ovaj sistem može da uđe u mrtvu blokadu (*deadlock*)? Precizno obrazložiti odgovor.

Odgovor:

Dati sistem ne može da bude u mrtvoj blokadi, jer nije zadovoljen neophodan uslov „držanje i čekanje“. Naime, nikada ne može nastati situacija da jedan proces traži novi resurs (i neograničeno čeka na njega) dok drži neki drugi resurs, zato što je, prema uslovima zadatka, svaki prethodno zauzeti resurs pre toga oslobodio.

38. (ispit 2011) Upravljanje deljenim resursima

Na problemu filozofa koji večeraju (*dining philosophers*) demonstrira se mehanizam izbegavanja mrtve blokade (*deadlock avoidance*) zasnovan na bankarevom algoritmu (*banker's algorithm*). Svaki filozof traži najpre svoju levu viljušku, pa kada nju dobije, traži i svoju desnu viljušku. Potrebno je prikazati matricu alokacije i vektor slobodnih resursa za svako dole navedeno stanje, tim redom. U svakom traženom stanju strukture prikazati nakon što je sistem dodelio resurse svima kojima su ih tražili, a kojima se mogu dodeliti resursi. Prikazati date strukture za sledeća stanja:

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

a)(2)

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
Av.	1	1	1	1	1

b)(3)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	0
Av.	0	0	0	0	1

c)(2)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	1
5	0	0	0	0	0
Av.	0	0	0	0	0

d)(3)

	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	1	0
4	0	0	0	0	0
5	0	0	0	0	1
Av.	0	0	0	0	0

39. (ispit 2011) Upravljanje deljenim resursima

Dat je potprogram koji obavlja transakciju umanjenja sredstava na prvom datom bakovnom računu, pod uslovom da ukupan iznos sredstava na tom i drugom datom računu time ne uđe u minus. Svaki račun identifikovan je jedinstvenim celobrojnim identifikatorom. Operacije čitanja stanja na računu (`read_account`) i upisa novog stanja (`write_account`) su atomične. Inicijalno nije ugrađena nikakva druga sinhronizacija u ovaj potprogram.

```
function trans (c1:long, c2:long, m:real) : boolean
var r1, r2 : real;
begin
  if c1=c2 then return false;
  r1:=read_account(c1);  r2:=read_account(c2);
  if r1+r2>=m then begin
    write_account(c1,r1-m);
    return true;
  end
  else
    return false;
end;
```

a)(5) Precizno objasniti koja neregularna situacija može da nastane pri uporednom izvršavanju ovog potprograma i kako.

b)(5) Svakom računu sa brojem `c` pridružen je po jedan standardni brojački semafor, sa inicijalnom vrednošću 1, kome se pristupa preko `sem(c)`. Pomoću ovih semafora sprečiti nastanak ove neregularne situacije, uz sprečavanje mrtve blokade u sistemu sa velikim, ali konačnim brojem procesa i računa.

Rešenje:

a)(5) Može da dođe do neregularnog stanja u kome ne važi da je ukupna vrednost na oba računa nenegativna utrkivanjem (*race condition*) na sledeći način. Dva uporedna procesa izvršavaju ovaj potprogram skidanja iste sume, npr. 200, sa dva računa `a` i `b`, pri čemu svaki skida sa svog računa (prvi poziva sa `(a,b,200)`, drugi sa `(b,a,200)`). Neka je početno stanje oba računa po 100. Oba procesa uporedo pročitaju vrednosti stanja računa `a` i `b` za koje na početku važi da im je zbir veći ili jednak 200. Zbog ispunjenja oba uslova, oba procesa uporedo izvrše umanjenje, svako svog računa, za po 200. Na kraju, ukupno stanje ova dva računa iznosi -200 (manje od 0).

b)(5)

```
function trans (c1:long, c2:long, m:real) : boolean
var r1, r2 : real;
    b : boolean;
begin
  if c1=c2 then return false;
  if c1<c2 then
    begin sem(c1).wait(); sem(c2).wait(); end
  else
    begin sem(c2).wait(); sem(c1).wait(); end;
  r1:=read_account(c1);  r2:=read_account(c2);
  if r1+r2>=m then begin
    write_account(c1,r1-m);
    b:=true;
  end
  else b:=false;
  sem(c1).signal(); sem(c2).signal();
end;
```

40. (ispit 2011) Upravljanje deljenim resursima

Neki sistem primenjuje Bankarev algoritam izbegavanja mrtve blokade. U nekom trenutku, stanje posmatranog sistema je sledeće:

	Allocation		
	A	*B*	*C*
P1	2	0	1
P2	0	1	1
P3	1	1	0

Max		
A	*B*	*C*
2	3	2
1	3	3
3	2	2

Available		
A	*B*	*C*
2	1	2

Nakon toga, proces *P1* izdaje zahtev za alokacijom resursa (0,1,1), a nakon toga se proces *P2* završava i oslobađa sve resurse koje je zauzimao. Precizno objasniti šta se dešava i dati stanja sistema nakon svakog od ova dva događaja.

Rešenje:

Polazno stanje je bezbedno (jedna sigurna sekvenca je *P3*, *P2*, *P1*):

	Allocation		
	A	*B*	*C*
P1	2	0	1
P2	0	1	1
P3	1	1	0

Max		
A	*B*	*C*
2	3	2
1	3	3
3	2	2

Available		
A	*B*	*C*
2	1	2

Zahtev procesa *P1* za alokaciju resursa (0,1,1) se ne prihvata, jer kada bi se prihvatio, dobijeno stanje bi bilo nebezbedno (nijedan proces ne može da zadovolji maksimalne zahteve):

	Allocation		
	A	*B*	*C*
P1	2	1	2
P2	0	1	1
P3	1	1	0

Max		
A	*B*	*C*
2	3	2
1	3	3
3	2	2

Available		
A	*B*	*C*
2	0	1

Zato se *P1* blokira. Kada se *P2* završi i oslobodi resurse (0,1,1), stanje postaje:

	Allocation		
	A	*B*	*C*
P1	2	0	1
P3	1	1	0

Max		
A	*B*	*C*
2	3	2
3	2	2

Available		
A	*B*	*C*
2	2	3

Tada se procesu *P1* mogu dodeliti resursi na koje čeka (0,1,1), pa se on deblokira i resursi mu se dodeljuju, jer je novo stanje bezbedno i izgleda ovako (sigurna sekvenca je *P3*, *P1*):

	Allocation		
	A	*B*	*C*
P1	2	1	2
P3	1	1	0

Max		
A	*B*	*C*
2	3	2
3	2	2

Available		
A	*B*	*C*
2	1	2

41. (ispit 2011) Upravljanje deljenim resursima

Dato je sledeće rešenje problema filozova koji večeraju (*dining philosophers*).

```
var forks : array 0..4 of semaphore = 1;
task type Philosopher(i:0..4)
var left, right, first, second : 0..4;
begin
  left := i; right:=(i+1) mod 5;
  if (i mod 2 = 0) then begin
    first := left; second:=right;
  end else begin
    first := right; second:=left;
  end;
  loop
    think;
    forks[first].wait;
    forks[second].wait;
    eat;
    forks[first].signal;
    forks[second].signal;
  end;
end;
```

a)(5) Pokazati da u ovom rešenju ne može nastati mrtva blokada (*deadlock*) navođenjem neophodnog uslova nastanka mrtve blokade koji nije zadovoljen.

Odgovor:

Nije zadovoljen neophodan uslov postojanja kružnog čekanja (*circular wait*), jer filozofi uzimaju levu pa desnu, odnosno desnu pa levu viljušku naizmenično.

b)(5) Da li u ovom rešenju postoji izgladnjivanje (*starvation*)?

Odgovor:

Ne postoji.

42. (ispit 2011) Upravljanje deljenim resursima

Na jeziku C implementirati funkciju `isSafeState()` koja se primenjuje u bankarevom algoritmu, a koja ispituje da li je stanje sistema predstavljeno datim strukturama podataka bezbedno ili ne. U nastavku su date definicije potrebnih struktura podataka, kao i pomoćnih operacija koje su na raspolaganju. Pretpostavlja se da su matrice `allocation` i `available` kopije matrica stvarnog stanja alokacije resursa, pa se mogu slobodno menjati u funkciji `isSafeState()` (nije potrebno praviti njihove privremene kopije).

```
const int MAXPROCESSES = ...;
const int MAXRESOURCES = ...;
extern int numOfProcesses, numOfResources;
int allocation[MAXPROCESSES][MAXRESOURCES];
int max[MAXPROCESSES][MAXRESOURCES];
int available[MAXRESOURCES];

int isLessOrEqual (int x[], int y[]) {
  for (int i=0; i<numOfResources; i++) if (x[i]>y[i]) return 0;
  return 1;
}

void add (int x[], int y[], int z[]) {
  for (int i=0; i<numOfResources; i++) z[i]=x[i]+y[i];
}

void sub (int x[], int y[], int z[]) {
  for (int i=0; i<numOfResources; i++) z[i]=x[i]-y[i];
}

int isSafeState();
```

Rešenje:

```
int isSafeState () {
  int allComplete = 0;
  while (!allComplete) {
    int found = 0;
    allComplete = 1;
    for (int i=0; i<numOfProcesses && !found; i++) {
      if (allocation[i][0] == -1) continue; // Process marked as completed
      else allComplete = 0;
      int need[MAXRESOURCES];
      sub(max[i],allocation[i],need);
      if (isLessOrEqual(need,available)) {
        add(available,allocation[i],available);
        allocation[i][0]=-1; // Mark process as completed
        found=1;
      }
    }
    if (!found && !allComplete) return 0;
  }
  return 1;
}
```

43. (ispit 2011) Upravljanje deljenim resursima

Pod pretpostavkom da su semafori implementirani tako da obezbeđuju korektnost (engl. *fairness*), odnosno da ne izgladnjuju procese koji na njima čekaju, dokazati da dole dato rešenje nema problem nastanka mrtve blokade ni izgladnjivanja.

```
var forks : array 0..4 of semaphore = 1;
    deadlockPrevention : semaphore = 4;

task type Philosopher(i:int) begin
  left := i; right:=(i+1) mod 5;
  loop
    think;
    deadlockPrevention.wait;
    forks[left].wait;
    forks[right].wait;
    eat;
    forks[left].signal;
    forks[right].signal;
    deadlockPrevention.signal;
  end;
end;
```

Rešenje:

Mrtva blokada ne može da nastane jer nije ispunjen neophodan uslov cikličnog čekanja (engl. *circular wait*). Dokaz kontradikcijom. Pretpostavimo suprotno, da postoji ciklično čekanje. Ono može nastati u prikazanom rešenju samo ako je svaki filozof zauzeo svoju levu viljušku (prošao `forks[left].wait`), a čeka na svoju desnu (blokiran na `forks[right].wait`). To bi značilo da su svih pet procesa prošli `deadlockPrevention.wait`, što nije moguće, jer je najveća vrednost ovog semafora 4.

Izgladnjivanje ne može da nastane iz sledećeg razloga. Kada se neki proces blokira na semaforu `deadlockPrevention`, on će u konačnom vremenu biti odblokiran, pošto su semafori fer, a nema mrtve blokade. Slično važi i za semafore `forks[left]` i `forks[right]`. Zbog toga će svaki proces-filozof u konačnom vremenu doći do procedure `eat`.

1. (Decembar 2006) Upravljanje memorijom

Jezgro nekog operativnog sistema sprečava *thrashing* praćenjem aproksimacije radnog skupa datog procesa pomoću bita referenciranja. Kad god dati proces izgubi procesor iz bilo kog razloga (uključujući i istek vremenskog kvantuma), jezgro proverava da li su sve stranice koje su referencirane od prethodne takve provere učitane u memoriju. Ako neka nije, pokreće se operacija njenog učitavanja, a na kraju provere se svi biti referenciranja brišu.

U deskriptoru stranice koji koristi procesor najniži bit je bit referenciranja, a bit 1 je bit prisutnosti stranice u memoriji. Tabela preslikavanja stranica deklarisana je na sledeći način:

```
#define pmt_size ...
typedef unsigned long int pg_descr;
typdef pg_descr pmt[pmt_size];
```

Napisati kod koji vrši navedenu proveru i poziva postojeću operaciju `load_pg(pmt, unsigned pg_num)` za svaku referenciranu stranicu koja nije u memoriji za datu tabelu preslikavanja.

Rešenje:

```
#define pmt_size ...
typedef unsigned long int pg_descr;
typdef pg_descr pmt[pmt_size];

void chk_thrashing (pmt pm) {
  static pg_descr one = 1;
  static pg_descr all_ones_and_zero = ~one;
  for (unsigned long p = 0; p<pmt_size; p++) {
    if ((pm[p] & 3) == 1) load_pg(pm,p);
    pm[p] &= all_ones_and_zero;
  }
}
```

2. (Decembar 2006) Upravljanje memorijom

Jezgro nekog operativnog sistema koristi sistem parnjaka (*buddy*) za alokaciju memorije za svoje potrebe. Fizička memorija u kojoj se alocira prostor zauzima 8 okvira (označenih sa 0..7). Inicijalno su svi okviri slobodni, a redom su izvršeni sledeći pozivi/zahtevi:

1A3, 2A1, 3A2, 1F, 4A1, 4F, 2F, 3F

Oznaka *i*A*n* predstavlja zahtev za alokacijom *n* susednih okvira, pri čemu se taj zahtev, odnosno alocirani prostor referiše kao *i*, a oznaka *i*F označava operaciju oslobađanja prostora referisanog sa *i*.

Prikazati stanje zauzetosti okvira posle svake od ovih operacija. Za svaki okvir upisati F*k* ako je slobodan i pripada parnjaku *k* (parnjake označavati redom, tako da okviri koji pripadaju istom parnjaku imaju isti broj *k*, a oni koji pripadaju različitim parnjacima imaju različite oznake), a A*i* ako je zauzet zahtevom za prostor referisanom sa *i*.

Okvir	0	1	2	3	4	5	6	7
Inicijalno	F0	F0	F0	F0	F0	F0	F0	F0
1A3								
2A1								
3A2								
1F								
4A1								
4F								
2F								
3F								

Rešenje:

Okvir	0	1	2	3	4	5	6	7
Inic.	F0	F0	F0	F0	F0	F0	F0	F0
1A3	A1	A1	A1	A1	F0	F0	F0	F0
2A1	A1	A1	A1	A1	A2	F0	F1	F1
3A2	A1	A1	A1	A1	A2	F0	A3	A3
1F	F0	F0	F0	F0	A2	F1	A3	A3
4A1	F0	F0	F0	F0	A2	A4	A3	A3
4F	F0	F0	F0	F0	A2	F1	A3	A3
2F	F0	F0	F0	F0	F1	F1	A3	A3
3F	F0	F0	F0	F0	F0	F0	F0	F0

3. (Decembar 2007) Upravljanje memorijom

Neki operativni sistem primenjuje aproksimaciju LRU algoritma za zamenu stranica korišćenjem dodatnih bita referenciranja (*additional reference-bits algorithm*). Deskriptor stranice u tabeli preslikavanja stranica (*page map table*, PMT), kao i sama PMT koju koristi hardver se iz programa na jeziku C vidi na sledeći način:

```
const int num_of_pages = ...; // number of pages in one address space
typedef unsigned int page_descr;  // page descriptor is one word
typedef page_descr* PMT;   // PMT is an array of page_descr
```

U deskriptoru stranice u PMT najviši bit je bit referenciranja koji postavlja hardver prilikom svakog pristupa stranici. Struktura dodatnih bita referenciranja koju održava operativni sistem za svaki proces izgleda ovako:

```
typedef unsigned int lru_reg;
typedef lru_reg* LRU_regs_table; // LRU table is an array of lru_reg
```

Na raspolaganju je funkcija koja ispituje da li je stranica sa datim deskriptorom trenutno u memoriji ili ne:

```
int is_in_mem(page_descr);
```

Implementirati sledeće dve funkcije:

```
void update_lru_regs (PMT pmt, LRU_regs_table lru);
int get_victim(PMT pmt, LRU_regs_table lru);
```

Funkciju `update_lru_regs()` poziva interna nit kernela koja se aktivira periodično, za svaki aktivni proces i njegovu PMT. Ona treba da ažurira LRU registre, kao i bite referenciranja u PMT. Funkciju `get_victim()` poziva podsistem za zamenu stranica prilikom izbacivanja stranice iz memorije, za proces sa datom PMT i LRU tabelom. Ona treba da izabere stranicu za izbacivanje.

Rešenje:

```
void update_lru_regs (PMT pmt, LRU_regs_table lru)
{
  static int first_call = 1;
  static page_descr pd_mask = 0; // pd_mask == 000...00
  if (first_call) {
    pd_mask = ~pd_mask;  // pd_mask == 111...11
    pd_mask >>= 1;  // pd_mask == 011...11
    first_call = 0;
  }

  for (int i=0; i<num_of_pages; i++) {
    if (!is_in_mem(pmt[i])) continue;
    lru[i]>>1;  // shift right LRU reg
    lru[i] |= (pmt[i] & (~pd_mask)); // set MSB of LRU reg to the ref. bit
    pmt[i] &= pd_mask;  // reset reference bit
  }
}


int get_victim(PMT pmt, LRU_regs_table lru) {
  int victim = -1;
  lru_reg min = ~0;
```

```
  for (int i=0; i<num_of_pages; i++)
    if (!is_in_mem(pmt[i])) continue;
    if (lru[i]<=min) {
      victim = i;
      min = lru[i];
    }
  return victim;
}
```

4. (Decembar 2007) Upravljanje memorijom

Neki operativni sistem podržava memorijski preslikane fajlove (*memory mapped files*). U njemu postoji sistemski poziv

```
int mem_map_file(void* mem_buffer, size_t size, char* filename);
```

koji preslikava deo memorijskog adresnog prostora počev od zadate adrese i zadate veličine u zadati fajl. Ova funkcija vraća 0 ako je operacija uspela, ili kod greške (različit od 0) u slučaju greške. Zadata adresa mora da bude poravnata na početak stranice. Da bi prevodilac na datoj mašini poravnao generisani kod ili statički podatak na početak stranice, može se navesti nestandardna direktiva:

```
#pragma align
```

U fajl se preslikava uvek ceo broj stranica tako što navedena sistemska usluga zaokružuje zadatu veličinu memorijskog segmenta na ceo broj stranica.
Korišćenjem memorijski preslikanih fajlova (a ne standardnog fajl API), napisati program koji iz datog fajla učitava niz celih brojeva i na standardni izlaz ispisuje njihov zbir. Fajl sadrži *N* celih brojeva zapisanih u binarnom formatu, jedan odmah iza drugog. *N* je konstanta definisana u programu.

Rešenje:

```
#include <stdio.h>
int N = ...;
char* filename = ...;

#pragma align
int array[N];
#pragma align

void main () {
  if (mem_map_file(array,N*sizeof(int),filename)==0) {
    int sum = 0;
    for (int i=0; i<N; i++) sum+=array[i];
    printf("Sum: %d\n",sum);
  }
}
```

5. (Decembar 2008) Upravljanje memorijom

Neki operativni sistem primenjuje algoritam davanja nove šanse (*second-chance /clock/ algorithm*) za zamenu stranica. Deskriptor stranice u tabeli preslikavanja stranica (*page map table*, PMT), kao i sama PMT koju koristi hardver se iz programa na jeziku C vidi na sledeći način:

```
typedef unsigned int page_descr;  // page descriptor is one word
typedef page_descr* PMT;   // PMT is an array of page_descr
```

U deskriptoru stranice u PMT najniži bit je bit referenciranja koji postavlja hardver prilikom svakog pristupa stranici.
Kernel održava odgovarajuću strukturu podataka u kojoj vodi evidenciju o zauzetim i slobodnim okvirima fizičke memorije, kao i FIFO red okvira po redosledu učitavanja stranica u njih. Ovoj strukturi pristupa se preko sledećih funkcija koje su na raspolaganju:

```
int get_clock_frame();
void move_clock_hand();
page_descr* get_owner_page(int frame_num);
```

Funkcija `get_clock_frame` vraća broj okvira na koji trenutno ukazuje „kazaljka“ u globalnom algoritmu zamene stranica, dok funkcija `move_clock_hand` kružno pomera tu kazaljku na sledeći zauzet okvir u FIFO redu. Funkcija `get_owner_page` vraća pokazivač na deskriptor stranice u PMT koja zauzima okvir sa zadatim brojem, ukoliko je okvir zauzet, inače vraća 0.
Implementirati sledeću funkciju:

```
int get_victim_frame();
```

koja treba da vrati broj okvira iz koga se izbacuje stranica, a kazaljku ostavi na tom okviru.

Rešenje:

```
int get_victim_frame() {
  while (1) {
    int fm = get_clock_frame();
    page_descr* pd = get_owner_page(fm);
    if (*pd & 1) {  // if referenced,
      *pd &= ~1;  // reset reference bit
      move_clock_hand();  // and give it a new chance
    }
    else return fm;
  }
}
```

6. (Decembar 2008) Upravljanje memorijom

Kernel nekog operativnog sistema koristi tehniku „ploča“ (*slab*) za alokaciju memorije za svoje interne potrebe. U modulu za alokaciju tehnikom ploča implementirana je klasa `Slot` koja apstrahuje jedan pregradak (*slot*) za smeštanje jednog objekta datog tipa unutar ploče, kao i klasa `Slab` koja apstrahuje ploču. Interfejsi ovih klasa izgledaju ovako:

```
class Slot {
public:
  Slab* getOwnerSlab () const;
};

class Slab {
public:
  static Slab* createSlab (int numOfSlots, size_t slotSize);

  Slot* allocateSlot ();
  void  freeSlot(Slot*);

  Slab*  getNext () const;
  void   setNext (Slab*);
};
```

Funkcija `getOwnerSlab` vraća pokazivač na ploču unutar koje je pregradak alociran. Funkcija `createSlab` alocira prostor u memoriji i inicijalizuje jednu ploču za smeštanje `numOfSlots` pregradaka za smeštanje objekata veličine `slotSize`. Funkcija `allocateSlot` zauzima jedan pregradak unutar ploče, ako postoji slobodan pregradak i vraća pokazivač na taj pregradak; ukoliko je ploča sasvim puna, ova funkcija vraća 0. Funkcija `freeSlot` proglašava slobodnim pregradak na koji ukazuje dati pokazivač. Ploče se mogu ulančavati u listu preko pokazivača koji se može pročitati ili postaviti funkcijama `getNext` i `setNext`, respektivno.
Na raspolaganju je i funkcija

```
int getOptimalNumOfSlotsInSlab (size_t slotSize);
```

koja vraća najpogodniji broj pregradaka unutar ploče za datu arhitekturu (veličinu stranice).
Realizovati u potpunosti klasu `Cache` koja apstrahuje keš za smeštanje objekata jednog tipa, zadate veličine. Interfejs ove klase treba da bude sledeći:

```
class Cache {
public:
  Cache (size_t slotSize);

  Slot* allocateSlot ();
  void  freeSlot(Slot*);
};
```

Keš ne mora da dealocira sasvim ispražnjene ploče prilikom dealokacije pregradaka, niti da prioritira delimično popunjene ploče u odnosu na prazne ploče prilikom alokacije pregradaka.

Rešenje:

```
class Cache {
public:
  Cache (size_t slotSize);

  Slot* allocateSlot ();
  void  freeSlot(Slot*);

private:
  Slab *head;
  int numOfSlots;
  size_t slotSize
};

Cache::Cache (size_t slotSz) {
  slotSize = slotSz;
  numOfSlots = getOptimalNumOfSlotsInSlab(slotSize);
  head = Slab::create(numOfSlots,slotSize);
  if (head) head->setNext(0);
}

Slot*  Cache::allocateSlot () {
  for (Slab* cur=head; cur!=0; cur=cur->getNext()) {
    Slot* newSlot = cur->allocateSlot();
    if (newSlot) return newSlot;
  }
  Slab* newSlab = Slab::create(numOfSlots,slotSize);
  if (newSlab==0) return 0; // No more memory!
  newSlab->setNext(head);
  head=newSlab;
  return newSlab->allocateSlot();
}

void Cache::freeSlot (Slot* st) {
  Slab* sb = st->getOwnerSlab();
  if (sb) sb->freeSlot(st);
}
```

7. (Decembar 2009) Upravljanje memorijom

Neki operativni sistem koristi rezervoar (engl. *pool*) slobodnih okvira, uz ponovnu upotrebu istih okvira, kako bi postupak alokacije novog okvira za stranicu koja je tražena učinio efikasnijim. Rezervoar slobodnih okvira predstavljen je klasom `FramePool` na jeziku C++:

```
typedef unsigned int PID;  // Process ID
typedef unsigned long int PgID;   // page number
typedef unsigned long int FID;    // frame number

class FramePool {
public:
  FramePool () : head(0), tail(0) {}
  int  getFrame (PID proc, PgID page, FID& frame);
  void addFrame (PID proc, PgID page);
private:
  struct FPElem {
    PID proc;
    PgID page;
    FID frame;
    FPElem *prev, *next;
  };
  FPElem *head, *tail;
};
```

Evidencija o slobodnim okvirima u rezervoaru vodi se kao dvostruko ulančana lista dinamički alociranih elemenata tipa `FPElem`. U strukturi `FPElem`, u članu `proc` zapisan je ID procesa kojem je okvir zapisan u članu `frame` pripadao, a u članu `page` broj stranice tog procesa koja je izbačena, a koja je bila u tom okviru (i čiji je sadržaj sačuvan). Operacija `getFrame` treba da pronađe slobodan okvir u rezervoaru i da njegov broj vrati kroz izlazni argument `frame`, za dati proces `proc` i datu stranicu `page` koja se traži, i to tako da najpre pokuša da pronađe isti okvir u kome je ta stranica već bila. Ako takav pronađe, ova operacija treba da vrati 1. Ako takav ne pronađe, treba da vrati bilo koji slobodan okvir i da vrati 0. Ako slobodnih okvira u rezervoaru uopšte nema, treba da vrati -1.
Realizovati operaciju `getFrame`.

Rešenje:

```
int FramePool::getFrame (PID proc, PgID page, FID& frame) {
  if (head==0) return -1; // No free frames
  // Try to find the same one to reuse:
  int found = 0;
  for (FPElem* cur=head; cur!=0; cur=cur->next)
    if (cur->proc==proc && cur->page==page) {
      found = 1; break; // Found the same!
    }
  if (found==0) cur=head; // Take the first one
  frame=cur->frame;
  // Now remove the FPElem:
  if (cur->prev) cur->prev->next=cur->next;
  else head=cur->next;
  if (cur->next) cur->next->prev=cur->prev;
  else tail=cur->prev;
  delete cur;
  return ret;
}
```

8. (Decembar 2009) Upravljanje memorijom

Kernel nekog operativnog sistema koristi tehniku „parnjaka“ (*buddy*) za alokaciju memorije za svoje interne potrebe. Najmanji blok koji se može alocirati je veličine 4 KB. U nekom trenutku, slobodni blokovi memorije su sledećih veličina (sve veličine su u KB):
64, 512, 32, 128, 64, 256
U tom stanju zahteva se alokacija dela memorije veličine 6 KB. Napisati veličine slobodnih blokova nakon ove alokacije.

Odgovor: 64, 512, 8, 16, 128, 64, 256

9. (Decembar 2010) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima referenciranja (*additional-reference-bit algorithm*). Registar istorije bita referenciranja ima 4 bita. Posmatra se proces čije su četiri stranice označene sa 0..3 trenutno u operativnoj memoriji. Proces generiše sledeću sekvencu obraćanja stranicama; u ovoj sekvenci, oznaka X predstavlja trenutak kada stiže periodični prekid na koji operativni sistem pomera udesno registre istorije i upisuje u njih bite referenciranja:

`0,1,2,X,3,0,X,1,0,2,X,3,0,X,1,0,X,0,1,3,X,2,3,0,X,0,2,1,0,X,1,0,X,2,3,X`

Prikazati sadržaj registara istorije posle ove sekvence i navesti koja stranica bi bila izabrana za izbacivanje ukoliko se posle ove sekvence traži zamena stranice.

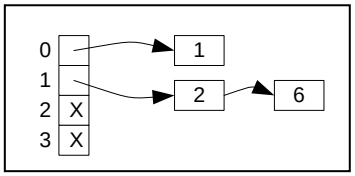
Rešenje:

0	0	1	1	1
1	0	1	1	0
2	1	0	1	1
3	1	0	0	1

Bila bi izbačena stranica 1.

10. (Decembar 2010) Upravljanje memorijom

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 2^{N-1} blokova koji su označeni brojevima $0..2^{N-1}-1$. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici, za primer $N = 4$. Svaki ulaz i ($i = 0..N-1$) prikazanog niza `buddy` sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. Glava liste sadrži broj prvog bloka u slobodnom komadu, a broj narednog bloka u listi je upisan na početku svakog slobodnog bloka u listi (-1 za kraj liste). Deklaracije potrebnih struktura date su dole. Funkcija `block()` vraća adresu početka bloka broj `n`.




```
const int N = ...; // N>1
int buddy[N];
void* block(int n);
```

Realizovati funkciju:

```
void* buddy_alloc(int i);
```

koja alocira komad veličine 2^i susednih blokova ($0 \le i < N$) i vraća pokazivač na alocirani komad, odnosno 0 ako nije u mogućnosti da ga alocira.

Rešenje:

```
void* buddy_alloc (int i) {
  if (i<0 || i>=N) return 0; // Error
  // First, try to find the segment of exact size of 2^i blocks:
  if (buddy[i]>-1) {
    // Found! Remove it from the list buddy[i] and return it:
    int* ret = (int*)block(buddy[i]);
    buddy[i] = *ret;
    return ret;
  }
  // Else, find the first next bigger segment:
  for (int j=i+1; j<N; j++) {
    int seg1 = buddy[j];
    if (seg1>-1) {
      // Found. Divide it into two halves:
      int seg2 = seg1 + (1<<(j-1));
      int* pSeg1 = (int*)block(seg1);
      int* pSeg2 = (int*)block(seg2);
      // Remove it from buddy[j]:
      buddy[j] = *pSeg1;
      // Add two segments to buddy[j-1]:
      *pSeg2 = buddy[j-1];
      *pSeg1 = seg2;
      buddy[j-1] = seg1;
      // Now, try again from the beginning (recursion).
      // You will find it eventually!
      return buddy_alloc(i);
    }
  }
  // Not found, no memory:
  return 0;
}
```

11. (Ispit Januar 2011) Upravljanje memorijom

Neko jezgro za potrebe alokacije svojih struktura koristi sistem ploča (*slab allocator*). Posmatra se realizacija keša za alokaciju objekata neke klase `X` date dole. Kada u kešu više nema slobodnog mesta za alokaciju novog objekta klase `X`, nova ploča traži se od sloja ispod koji predstavlja alokator po sistemu parnjaka (*buddy*) operacijom

```
void* buddy_alloc(int i);
```

koja alocira ploču veličine 2^i susednih stranica i vraća pokazivač na alociranu ploču, odnosno 0 ako nije u mogućnosti da je alocira. Veličina ploče za keš klase `X` se izražava u stepenu dvojke *i* i konfiguracioni je parametar klase `X` (član `slabSizeInPages`). U jednu ploču može da stane *k* objekata klase `X`, gde je *k* takođe konfiguracioni parametar te klase (član `slabSizeInSlots`). Svi prazni pregradci u kešu klase `X` vezuju se u jednostruko ulančanu listu čija je glava u članu `pool`. Novi objekat se alocira prosto u prvom slobodnom pregradku, a prazne ploče ne treba oslobađati. Implementirati operacije `new` i `delete` klase `X`.

```
class X {
public:
  static void* operator new (size_t);
  static void  operator delete (void*);
private:
  static const unsigned int slabSizeInPages, slabSizeInSlots;
  static void* pool;
};
```

Rešenje:

```
class X {
public:
  static void* operator new (size_t);
  static void  operator delete (void*);
private:
  static const unsigned int slabSizeInPages, slabSizeInSlots;
  static void* pool;
};

void* X::operator new (size_t) {
  if (pool==0) {
    pool = buddy_alloc(slabSizeInPages);
    if (pool==0) return 0; // No more memory!
    void** p=(void**)pool;
    for (int i=0; i<slabSizeInSlots; i++,p=(void**)*p)
      *p = (char*)p + sizeof X;
  }
  void* ret = pool;
  pool = *(void**)pool;
  return ret;
}

void X::operator delete (void* p) {
  if (p==0) return;
  *(void**)p = pool;
  pool=p;
}
```

12. (Novembar 2011) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se algoritam „davanja nove šanse“ ili „časovnika“ (*second-chance, clock algorithm*). Primenjuje se lokalna politika zamene stranice unutar istog procesa. U deskriptoru stranice u PMT, koji je tipa `unsigned long int`, najviši bit je bit referenciranja. U posebnom nizu `pagefifo` svaki element odgovara jednoj stranici i sadrži indeks ulaza u tom nizu koji predstavlja sledeću stranicu u kružnoj FIFO strukturi stranica uređenih po redosledu učitavanja.

```
struct PCB {

unsigned int clockHand;

unsigned long* pmt; // Pointer to the PMT

unsigned int* pagefifo; // Pointer to the FIFO page table
};
```

Implementirati funkciju `getVictimPage(PCB*)` koja vraća broj stranice koju treba zameniti po ovom algoritmu zamene, za proces sa datim PCB.

Rešenje:

```
unsigned int getVictimPage (PCB* pcb) {

static const unsigned long mask = ~(~0UL>>1); // 100...0b

if (pcb==0) return -1; // Exception!

while (pcb->pmt[pcb->clockHand] & mask) {
      pcb->pmt[pcb->clockHand] &= ~mask;

      pcb->clockHand = pcb->pagefifo[pcb->clockHand];

}

         unsigned int victim = pcb->clockHand;
         pcb->clockHand = pcb->pagefifo[pcb-
  >clockHand]; return victim;
         }
```

13. (Novembar 2011) Upravljanje memorijom

U jezgru nekog operativnog sistema primenjuje se sistem ploča (*slab allocator*) za alokaciju struktura za potrebe jezgra. Za alokaciju nove ploče kada u kešu više nema slobodnih slotova koristi se niži sloj koji implementira *buddy* alokator.

U nastavku su date su delimične definicije klasa `Cache` i `Slab` i implementacije nekih njihovih operacija. Slotovi za alokaciju su tipa `X`. Struktura `Cache` predstavlja keš i čuva pokazivač `headSlab` na prvu ploču u kešu; ploče u kešu su ulančane u jednostruku listu. Strukura `Slab` predstavlja ploču. U njoj su pokazivač na sledeću ploču istog keša `nextSlab`, kao i pokazivač `freeSlot` na prvi slobodan slot u nizu `slots` svih slotova u ploči. Slobodni slotovi su ulančani u jednostruku listu preko pokazivača koji se smeštaju u sam slot, na njegov početak. Inicijalizaciju jedne ploče vrši dati konstruktor.

```
const int numOfSlotsInSlab = ...;

class Cache {
public:

X* alloc();
private:
friend class Slab;
  Slab* headSlab;
}

class Slab
{
public:
  Slab (Cache* ownerCache);
  static void* operator new (size_t s) { return buddy_alloc(s); }
static void  operator delete (void* p) { buddy_free(p,sizeof(Slab)); }
private:
  friend class
  Cache;
  Slab* nextSlab;
  X* freeSlot;
X slots[numOfSlotsInSlab];
};

Slab::Slab(Cache* c) {
    this->nextSlab=c-
    >headSlab; c-
    >headSlab=this;
    this->freeSlot=&this->slots;
    for (int i=0; i<numOfSlotsInSlab-1; i++)
      *(X**)(&slots[i])=&slots[i+1];
*(X**)(&slots[numOfSlotsInSlab-1])=0;
}
```

Implementirati operaciju `Cache::alloc()` koja treba da alocira jedan slobodan slot `X`. Nije potrebno optimizovati alokaciju tako da se slot traži najpre u delimično popunjenim pločama, pa tek u praznoj, već se može prosto vratiti prvi slobodan slot na koga se naiđe.

Rešenje:

```
X* Cache::alloc() {
  Slab* s=this->headSlab;
  for (; s!=0; s=s->nextSlab)  // Find a slab with a free slot
  if (s->freeSlot) break;
  if (s==0) // No free slot. Allocate a new slab:
    s = new Slab(this);
  if (s==0 || s->freeSlot==0) return 0; // Exception: no free memory
  X* ret = s->freeSlot;
  s->freeSlot=*(X**)s->freeSlot;
  return ret;
}
```

14. (Novembar 2012) Upravljanje memorijom

Neki sistem podržava memorijski preslikane fajlove (*memory mapped files*) i nudi sistemski
poziv `mmapfile` koji preslikava sadržaj fajla zadatog imenom u prvi deo adresnog prostora pozivajućeg procesa koji operativni sistem pronađe kao nealociran, a koji je dovoljno veliki da
se u njega preslika sadržaj veličine datog fajla. Ovaj poziv vraća pokazivač na alocirani segment virtuelne memorije u koji je preslikan dati fajl, odnosno *null* ukoliko poziv nije uspeo.

U binarnom fajlu `log.bin` snimljeni su podaci u sledećem binarnom formatu:

- na početku je snimljen ceo broj *n* (tipa `int`);
- u nastavku je tačno *n* zapisa tipa `DailySales`, u čijem polju `quantity` tipa `double` je upisana količina nekog artikla prodata u jednom danu.

Napisati funkciju koja tehnikom memorijski preslikanog fajla izračunava ukupnu prodatu količinu datog artikla. Smatrati da je ulazni fajl sigurno korektno zapisan u navedenom formatu.

Rešenje:

```
double soldQuantity () {
  void* storage = mmapfile("log.bin");
  if (storage==0) return -1; // Exception!
  int n = *(int*)storage;
  DailySales* log = (DailySales*)((int*)storage+1);
  double sum = 0.0;
  for (int i=0; i<n; i++)
    sum+=log[i].quantity;
  return sum;
}
```

15. (Septembar 2012) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima referenciranja. Svakoj stranici pridružen je 8-bitni registar sa sačuvanim ranijim bitima referenciranja koji se pomeraju udesno. Posmatra se grupa od četiri stranice označene sa 1-4 i sledeća sekvenca njihovog referenciranja (simbol ↓ označava periodični prekid na koji se ažuriraju registri referenciranja):

1, 2, 3, 4, 2, 4, 1, ↓, 2, 1, 3, 1, ↓, 1, 3, 4, 1, ↓, 2, 1, 3, 2, 1, ↓, 4, 2, 4, 2, 4, ↓, 1, 3, 2, 1, 2, 3, ↓

Dati heksadecimalne vrednosti registara za ove četiri stranice nakon ove sekvence. Ako se nakon ove sekvence mora zameniti neka od ovih stranica, koja će to stranica biti?

Rešenje:

1: BCh, 2: ECh, 3: BCh, 4: 54h.

Biće izbačena stranica 4

16. (Septembar 2012) Upravljanje memorijom

U jezgru nekog operativnog sistema primenjuje tehnika izbegavanja pojave *thrashing* praćenjem aproksimacije radnog skupa. Sistem periodično prebrojava bite referenciranja korišćenih stranica svakog procesa i tim brojem aproksimira veličinu radnog skupa tog procesa. U PCB svakog procesa polje `pmt` ukazuje na PMT procesa. PMT je niz od `NumOfPages` deskriptora stranica. Svaki deskriptor je tipa `unsigned int`, a u njemu je najviši
bit bit referenciranja.

Implementirati funkciju

```
unsigned long workingSetSize (PCB* pcb);
```

koja se poziva periodično i koja treba da prebroji postavljene bite referenciranja datog procesa i vrati taj broj.

Rešenje:

```
unsigned long workingSetSize (PCB* pcb) {
  static const unsigned int mask = ~(~0U>>1); // 100...0b
  if (pcb==0) return -1; // Exception!
  unsigned long size = 0;
  for (unsigned long int i=0; i<NumOfPages; i++)
    size += ((pcb->pmt[i] & mask)!=0);
  return size;
}
```

17. (Septembar 2013) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se algoritam časovnika (engl. *clock algorithm*). Svaka učitana stranica nekog procesa opisana je strukturom `PageDescr` čija je definicija data dole. Polje `page` u ovoj strukturi čuva broj stranice koju opisuje struktura, a polje `ref` čuva vrednost bita referenciranja. Ovo polje `ref` ažurira posebna periodična rutina
sistema na osnovu bita referenciranja koje održava hardver i koja nije relevantna ovde. Za svaki proces, ove strukture su ulančane u dvostruko ulančanu kružnu listu, kao podrška algoritmu zamene. Za ovu svrhu služi klasa `PageClock` čija je implementacija data dole; jedan
objekat ove klase implementira listu učitanih stranica i algoritam izbacivanja za jedan proces.

Realizovati operaciju `PageClock::removeVictim()` koja treba da odabere stranicu za izbacivanje, izbaci njenu strukturu iz ulančane liste i vrati broj odabrane stranice. Ukoliko je lista stranica prazna, treba vratiti -1 kao kod greške.

```
typedef unsigned int PageNo;

struct PageDescr {
  PageNo page; // Page
  number int ref; //
  Reference bit PageDecr*
  next;
  PageDescr* prev;

  PageDescr (PageNo pg, PageDescr* nxt, PageDescr* prv) : page(pg),ref(0)
  {
    this->prev=prv;
    if (this->prev) this->prev-
    >next=this; else this->prev=this;
    this->next=nxt;
    if (this->next) this->next-
    >prev=this; else this->next=this;
  }

  void remove () {
  if (this->prev) this->prev->next=this->next;
    if (this->next) this->next->prev=this-
  >prev; }
};

class PageClock
{
public:
  PageClock () : hand(0) {}
void     addPage (PageNo page);  // Adds the loaded page
PageNo removeVictim ();    // Returns and removes the victim page
private:
PageDescr* hand; // Clock hand (cursor)
};

void PageClock::addPage (PageNo pg)
{
  if (hand==0)
  hand = new PageDescr(pg,0,0);
  else
new PageDescr(pg,hand->prev,hand);
}
```

Rešenje:

```
PageNo PageClocl::removeVictim () {
  if (hand==0) return -1; // No pages
  while (hand->ref) {
  hand->ref=0; hand=hand->next;
  }
  PageDescr* victim = hand;
```

```
  PageNo pg = victim->page;
  if (hand->next==hand) hand=0;
  else hand=hand->next;
  victim->remove();
  delete victim;
  return pg;
}
```

18. (ispit 2006) Virtuelna memorija

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima referenciranja (*additional-reference-bit algorithm*). Registar istorije bita referenciranja ima 4 bita. Posmatra se proces čije su četiri stranice označene sa 0..3 trenutno u operativnoj memoriji. Proces generiše sledeću sekvencu obraćanja stranicama; u ovoj sekvenci, oznaka X predstavlja trenutak kada stiže periodični prekid na koji operativni sistem pomera udesno registre istorije i upisuje u njih bite referenciranja:

`0,1,3,X,2,3,0,X,0,2,1,0,X,1,0,X,2,3,X,0,1,2,X,3,0,X,1,0,2,X,3,0,X,1,0,X`

Prikazati sadržaj registara istorije posle ove sekvence i navesti koja stranica bi bila izabrana za izbacivanje ukoliko se posle ove sekvence traži zamena stranice.

Rešenje:

0	1	1	1	1
1	1	0	1	0
2	0	0	1	0
3	0	1	0	1

Bila bi izbačena stranica 2.

19. (ispit 2006) Virtuelna memorija

Za izbor stranice za zamenu u nekom sistemu koristi se algoritam „davanja nove šanse“ ili „časovnika“ (*second-chance, clock algorithm*). Struktura koja čuva bite referenciranja implementirana je kao prosti niz (konstanta `FrameNum` predstavlja broj okvira):

```
const unsigned long int FrameNum = ...;
int refereceBits[FrameNum]; // 0 - not referenced, !=0 - referenced
unsigned long int clockHand;
```

Implementirati funkciju `getVictimFrame()` koja vraća broj okvira čiju stranicu treba zameniti po ovom algoritmu zamene.

Rešenje:

```
const unsigned long int FrameNum = ...;
int refereceBits[FrameNum]; // 0 - not referenced, !=0 - referenced
unsigned long int clockHand;

unsigned long int getVictimFrame () {
  while (referenceBits[clockHand]) {
    referenceBits[clockHand] = 0; // Give a second chance
    clockHand=(clockHand+1)%FrameNum;
  }
  return clockHand;
}
```

20. (ispit 2006) Virtuelna memorija

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima istorije referenciranja. Struktura koja čuva bite referenciranja implementirana je kao niz reči, pri čemu svakoj stranici odgovara jedan bit, a biti su poređani od najnižeg ka najvišem: stranici 0 odgovara bit 0 u reči 0, stranici 1 odgovara bit 1 u reči 0, itd. Veličina jedne reči (tip `int`, u bitima) je definisan konstantom `bitsInWord`. Registri istorije referenciranja implementirani su nizom `referenceHistory`, pri čemu svakoj stranici odgovara jedan element ovog niza veličine jedne reči.

```
const unsigned int bitsInWord = ...;
const unsigned int NumOfPages = ...;  // Number of pages in address space
unsigned int* referenceBits; // 0 - not referenced, 1 - referenced
unsigned int referenceHistory[NumOfPages];
```

Implementirati:

a)(5) funkciju `updateReferenceHistory()` koja se poziva periodično i koja treba da ažurira registre istorije referenciranja bitima referenciranja;

b)(5) funkciju `getVictimPage()` koja vraća redni broj stranice koja je izabrana za zamenu.

Rešenje:

```
const int bitsInWord = ...;
const int NumOfPages = ...;
unsigned int* referenceBits; // 0 - not referenced, 1 - referenced
unsigned int referenceHistory[NumOfPages];
```

a)(5)

```
void updateReferenceHistory () {
  for (int pg=0; pg<NumOfPages; pg++) {
    unsigned int refBitsWordNo = pg/bitsInWord;
    unsigned int refBitsBitNo = pg%bitsInWord;
    unsigned int refBitWord = referenceBits[refBitsWordNo];
    unsigned int refBit = refBitWord>>refBitsBitNo;
    refBit<<=(bitsInWord-1);
    referenceHistory[pg]>>=1;  // Shift right reference history register
    referenceHistory[pg]|=refBit;  // and set its MSB to reference bit
  }
}
```

Napomena: dato rešenje je verovatno jedno od najrazumljivijih, ali ne i najefikasnije. Ostavlja se studentima da pronađu i nešto efikasnija rešenja (u smislu manjeg broja pristupa memoriji).

b)(5)

```
unsigned int getVictimPAge () {
  unsigned int result = 0;
  unsigned int minRefHist = referenceHistory[0];
  for (int pg=1; pg<NumOfPages; pg++)
    if (referenceHistory[pg]<minRefHist) {
      result = pg;
      minRefHist = referenceHistory[pg];
    }
  return result;
}
```

21. (ispit 2006) Virtuelna memorija

U nekom sistemu sa virtuelnom memorijom stranice se označavaju kao „zaprljane“ (*dirty*) ukoliko je izvršena neka operacija upisa u tu stranicu od trenutka njenog učitavanja u memoriju. Kada se ta stranica izabere za zamenu, ukoliko je označena kao „zaprljana“, potrebno je pokrenuti operaciju njenog snimanja na uređaj (disk) koji služi za zamenu stranica.

a)(5) Navesti tehniku koja povećava verovatnoću da stranica izabrana za zamenu nije označena kao „zaprljana“ i time poboljšava performanse sistema jer smanjuje broj operacija snimanja na disk prilikom zamene stranica, odnosno skraćuje prosečno vreme zamene stranica.

Odgovor:

b)(5) Navesti tehniku koja odlaže operaciju upisa na disk, a omogućava da je prilikom stranične greške (*page fault*) najčešće već na raspolaganju slobodan okvir i time poboljšava performanse sistema skraćujući vreme zamene, jer ne postoji potreba za izbacivanjem i snimanjem stranice. Za kada se onda odlaže snimanje zaprljane stranice?

Odgovor:

a)(5) OS u pozadini, tokom rada drugih procesa, obilazi „zaprljane“ stranice i pokreće operaciju njihovog snimanja na disk kad god je uređaj koji za to služi slobodan. Time se povećava verovatnoća da stranica koja je izabrana za zamenu nije označena kao „zaprljana“.

b)(5) Vođenje „bazena“ slobodnih stranica (*page pooling*): kada se traži slobodan okvir, uglavnom se pronalazi takav u bazenu i proces koji je tražio stranicu može odmah da nastavi izvršavanje (nema operacije snimanja zaprljane stranice koja se izbacuje). Da bi se bazen dopunio, OS u pozadini, tokom rada drugih procesa, bira neku stranicu za izbacivanje da bi okvir koji ona zauzima oslobodio i dodao u bazen. U tom trenutku, kada se neki okvir označava slobodnim i smešta u bazen, pokreće se snimanje njegovog sadržaja na disk.

22. (ispit 2006) Virtuelna memorija

U nekom sistemu sa straničnom organizacijom virtuelne memorije primenjuje se tehnika sprečavanja pojave *thrashing* pomoću praćenja radnog skupa stranica. Informacija o radnom skupu, zapravo o njegovoj aproksimaciji, dobija se tako što operativni sistem periodično prepisuje bite referenciranja stranica u PCB strukture procesa i zatim ih briše. PCB strukture su prealocirane u statički dimenzionisani niz `processes`. Date su sledeće deklaracije:

```
const unsigned NumOfVMPages = ...;// Num of pages in virtual address space
unsigned NumOfFrames = ...;// Num of physical frames available for paging
const unsigned MaxNumOfProcs = ...; // Max num of processes

struct PCB {
  int isActive; // Is this process active (1) or is swapped out (0)
  int reference[NumOfVMPages]; // Reference info for all pages
  ...
};

PCB processes[MaxNumOfProcs];
int isThrashing();
```

Implementirati funkciju `isThrashing()` koja treba da vrati 1 ako je po kriterijumu ukupne veličine radnih skupova nastala pojava *thrashing*, a 0 ako nije.

Rešenje:

```
int isThrashing() {
  unsigned long totalWSSize = 0;
  for (unsigned iProc=0; i<MaxNumOfProcs; iProc++)
    if (processes[iProc].isActive)
      for (unsigned iPage=0; iPage<NumOfVMPages; iPage++)
        totalWSSize += processes[iProc].reference[iPage];
  return totalWSSize>NumOfFrames;
}
```

23. (ispit 2007) Upravljanje memorijom

Data je sekvenca referenciranja stranica:
2, 0, 2, 3, 4, 5, 4, 4, 6, 2, 0, 0, 1, 3
Ako je na raspolaganju 4 okvira memorije, koje stranice se redom nalaze u tim okvirima ako je algoritam zamene stranica:

a)(5) LRU. Odgovor: 3, 0, 2, 1

b)(5) FIFO. Odgovor: 1, 3, 2, 0

24. (ispit 2007) Upravljanje memorijom

Neki sistem primenjuje algoritam časovnika (davanja nove šanse, *clock, second-chance*) za izbor stranice za izbacivanje. U donjoj tabeli date su različite situacije u kojima treba odabrati stranicu za zamenu. Date su vrednosti bita referenciranja za sve stranice koje učestvuju u izboru i pozicija „kazaljke“. Kazaljka se pomera u smeru prema višim brojevima stranica. Za svaku od datih situacija navesti koja stranica će biti zamenjena i novo stanje posle izbora stranice za izbacivanje.

	Situacija 1		Situacija 2		Situacija 2	
Stranica	Bit ref.	Kazaljka	Bit ref.	Kazaljka	Bit ref.	Kazaljka
0	1		1		1	
1	1		0		1	
2	0		1		1	
3	0	X	1		1	X
4	1		0		1	
5	0		1	X	1	
6	1		1		1	
7	1		1		1	

Odgovor:

	Situacija 1		Situacija 2		Situacija 2	
Stranica	Bit ref.	Kazaljka	Bit ref.	Kazaljka	Bit ref.	Kazaljka
0						
1						
2						
3						
4						
5						
6						
7						

Zamenjena stranica:

Situacija 1 _____ Situacija 2_____ Situacija 3_____

Odgovor:

	Situacija 1		Situacija 2		Situacija 2	
Stranica	Bit ref.	Kazaljka	Bit ref.	Kazaljka	Bit ref.	Kazaljka
0	1		0		0	
1	1		0		0	
2	0		1	X	0	
3	0		1		0	
4	1	X	0		0	X
5	0		0		0	
6	1		0		0	
7	1		0		0	

Zamenjena stranica:

Situacija 1: 3 Situacija 2: 1 Situacija 3: 3

25. (ispit 2007) Upravljanje memorijom

Neki operativni sistem podržava straničnu organizaciju operativne memorije sa učitavanjem stranica na zahtev (*demand paging*, DP), kao i memorijski preslikane fajlove (*memory-mapped files*, MMF). Sistem obezbeđuje uslugu kojom se proizvoljni fajl sa datim imenom preslikava u virtuelni adresni prostor pozivajućeg procesa počev od zadate adrese. Ukratko, ali precizno objasniti u čemu se sastoji najjednostavniji postupak kreiranja novog procesa nad programom u zadatom .exe fajlu.
Odgovor:
Kreiranje novog procesa svodi se praktično samo na kreiranje potrebnih internih struktura koje opisuju proces i njegov aresni prostor (PCB), pri čemu su sve stranice inicijalno označene kao neučitane. Kontekst treba kreirati nad istim, generičkim kodom koji obavlja sledeće radnje:

- poziv sistemske usluge kojom se dati .exe fajl sa programom preslikava u virtuelni adresni prostor procesa, počev od neke fiksne lokacije u virtuelnom prostoru;
- skok na neku fiksnu lokaciju u virtuelnom adresnom prostoru na kojoj se očekuje početak (prva instrukcija) programa, ukoliko je to propisano tako, ili indirektni skok preko neke takve lokacije u kojoj se očekuje adresa početka programa.

Sve ostalo će obaviti mehanizmi DP i MMF, jer su stranice programa preslikane u fajl sa programskim kodom i statički alociranim podacima, a inicijalno su neučitane, pa će izvršavanje već prve instrukcije generisati *page fault* i učitavanje stranice iz programskog fajla. Dinamičku alokaciju prostora, npr. za podatke ili čak i stek, obaviće sam program tokom svog izvršavanja, pozivajući odgovarajuće sistemske pozive.

26. (ispit 2007) Upravljanje memorijom

Data je sledeća sekvenca referenciranja stranica od strane nekog procesa:
0, 3, 5, 2, 3, 5, 3, 1, 0, 3, 4, 5, 3, 4, 5
Procesu su dodeljena 4 okvira, zamena se vrši lokalno, samo u skupu stranica dodeljenih tom procesu, a inicijalno nije učitana ni jedna stranica ovog procesa. Koliko puta ovaj proces generiše straničnu grešku (*page fault*) ako je algoritam zamene stranica:

a)(5) FIFO? Odgovor: 9

a)(5) LRU? Odgovor: 8

Postupak:

27. (ispit 2007) Upravljanje memorijom

Za alokaciju memorije u jezgru nekog operativnog sistema primenjuje se sistem parnjaka (*buddy*). U nekom trenutku stanje zauzetosti prikazano je na donjoj slici (prikazani su blokovi i njihove veličine izražene u broju stranica; osenčeni blokovi su zauzeti, beli su slobodni). Na isti način prikazati stanje nakon izvršavanja zahteva za alokacijom memorije veličine 1 stranice.

4	2	2	4	4

Rešenje:

4	1	1	2	4	4

28. (4. januar 2008.) Upravljanje memorijom

Data je sledeća sekvenca referenciranja stranica od strane nekog procesa:
2, 5, 7, 4, 5, 7, 5, 3, 2, 5, 6, 7, 5, 6, 7
Procesu su dodeljena 4 okvira, zamena se vrši lokalno, samo u skupu stranica dodeljenih tom procesu, a inicijalno nije učitana ni jedna stranica ovog procesa. Koliko puta ovaj proces generiše straničnu grešku (*page fault*) ako je algoritam zamene stranica:

a)(5) FIFO?

b)(5) LRU?

Odgovor:

a) 9

b)8

29. (4. februar 2008.) Upravljanje memorijom

Zašto, po vašem mišljenju, alokator koji radi po principu ploča (*slab allocator*), slobodan pregradak (*slot*) za alokaciju novog objekta najpre traži u delimično popunjenoj ploči, pa tek ako takvu ne nađe, u potpuno praznoj ploči?

Odgovor:

Da bi forsirao što bolju popunjenost ploča, odnosno „štedeo“ potpuno prazne ploče koje koristi samo ako je zaista neophodno. Potpuno prazne ploče mogu da budu oslobođene iz datog keša i njihove stranice dodeljene drugom kešu kome su potrebne, ili dodeljene za druge potrebe sistema, dok stranice delimično popunjenih ploče to ne mogu. Osim toga, potpuno praznim pločama se i ne pristupa, pa su njihove stranice dobri kandidati za izbacivanje iz memorije, a okviri koje one zauzimaju mogu se dodeljivati drugim stranicama, ukoliko to sistem podržava. Na ovaj način potpuno prazne ploče ostaju takve što je duže moguće, čime se iskorišćenje memorije poboljšava.

30. (4. jun 2008.) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se algoritam „davanja nove šanse“ ili „časovnika“ (*second-chance, clock algorithm*) koji radi lokalno, u opsegu stranica jednog procesa. U deskriptoru stranice (veličine jedne mašinske reči) u tabeli preslikavanja stranica (PMT) najniži bit je bit referenciranja koji hardver postavlja na 1 svaki put kada pristupa stranici, dok vrednost 0 u celom deskriptoru ukazuje da data stranica nije u memoriji. Broj stranica u virtuelnom adresnom prostoru definisan je konstantom `NumOfPages`. Date su sledeće deklaracije:

```
const unsigned long int NumOfPages = ...;
typedef int PMT[NumOfPages];
int getVictimPage(PMT pmt, int* clockHand);
```

Implementirati funkciju `getVictimPage` koja vraća broj stranice koju treba zameniti po ovom algoritmu zamene, za proces sa datom PMT i „kazaljkom“ na koju ukazuje drugi argument. Ukoliko dati proces nema ni jednu stranicu u memoriji, treba vratiti -1.

Rešenje:

```
int getVictimPage (PMT pmt, int* clockHand) {
  static const int mask = 1; // Binary word: 000...001
  int newCH = *clockHead;
  for (int i = 0; i<2; i++) // Do this at most twice:
    do {
      if ((pmt[newCH]!=0) && !(pmt[newCH]&mask)) {// Found the victim
        *clockHead=(newCH+1)%NumOfPages;
        return newCH; };// Adjust *clockHead and return the victim
      if ((pmt[newCH]!=0) && (pmt[newCH]&mask)) // Give a second chance
        pmt[newCH] &= ~mask; // Reset the reference bit
      newCH = (newCH+1)%NumOfPages;  // Go to the next one
    } while (newCH!=*clockHead);
  return -1;
}
```

31. (ispit 2009) Virtuelna memorija

Iste strukture podataka i postupci koji se koriste kod aproksimacije LRU algoritma zamene stranica pomoću dodatnih bita referenciranja mogu se iskoristiti i za sprečavanje pojave *threshing*-a. Precizno objasniti kako.

Odgovor:

Jedan razred registara koji pamte bite referenciranja predstavljaju evidenciju aproksimacije radnog skupa stranica u datoj periodi. Sistem svakako periodično ažurira ove registre kao podršku algoritmu zamene stranica (pomera ove registre i u najsvežiji razred upisuje bite referenciranja). U cilju izbegavanja pojave *thrashing*-a, sistem treba da procesu obezbedi onoliko okvira koliko je potrebno za smeštanje njegovog radnog skupa stranica, a njega čine one stranice čiji su biti referenciranja u najsvežijem razredu ovih registara posavljeni, kao i one dodatne stranice koje su referencirane nakon te protekle periode. Ostale stranice može i da izbaci, oslobađajući okvire za druge procese.

32. (ispit 2009) Virtuelna memorija

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima referenciranja (*additional-reference-bit algorithm*). Registar istorije bita referenciranja ima 4 bita. Posmatra se proces čije su četiri stranice označene sa 0..3 trenutno u operativnoj memoriji. Proces generiše sledeću sekvencu obraćanja stranicama; u ovoj sekvenci, oznaka X predstavlja trenutak kada stiže periodični prekid na koji operativni sistem pomera udesno registre istorije i upisuje u njih bite referenciranja:

`0,1,3,X,2,1,0,X,1,2,1,0,X,2,0,X,2,1,X,0,1,2,3,X,3,1,X,3,0,2,X,3,0,X,1,0,2,X`

Prikazati sadržaj registara istorije posle ove sekvence i navesti koja stranica bi bila izabrana za izbacivanje ukoliko se posle ove sekvence traži zamena stranice.

Rešenje:

0	1	1	1	0
1	1	0	0	1
2	1	0	1	0
3	0	1	1	1

Bila bi izbačena stranica 3.

33. (ispit 2009) Alokacija memorije

Neki sistem koristi tehniku ploča (*slab*) za alokaciju memorije za potrebe jezgra. U datom kešu (*cache*) za objekte jednog tipa, evidencija slobodnih pregradaka (*slot*) vodi se kao jednostruko ulančana lista, pri čemu se pokazivač za ulančavanje upisuju u sam slobodan pregradak koji je zauzimao neki objekat. Date su sledeće deklaracije:

```
struct FreeSlot {
  FreeSlot* next; // Next FreeSlot in the list of free slots of a cache
};

struct Cache* {
  ...
  FreeSlot* freeSlots; // The list of free slots;
};

void free (Cache* cache, void* object);
```

Implementirati operaciju `free()` koja pregradak koji je zauzimao dati objekat u datom kešu proglašava slobodnim.

Rešenje:

```
void free (Cache* cache, void* object) {
  if (cache==0 || object==0) return; // Error
  FreeSlot* fs = (FreeSlot*)object;
  fs->next = cache->freeSlots;
  cache->freeSlots = fs;
}
```

34. (ispit 2009) Upravljanje memorijom

U cilju izbegavanja pojave *thrashing*, neki sistem primenjuje tehniku praćenja aproksimacije radnog skupa pomoću bita referenciranja. Biti referenciranja za sve stranice nekog procesa smešteni su u vektoru mašinskih reči (tip `unsigned int`) veličine `RefBitVectorSize`, na koga ukazuje polje `refBitVector` u PCB strukturi svakog procesa. U ovom vektoru svakoj stranici odgovara po jedan bit čija vrednost 1 znači da je stranici pristupano. Implementirati funkciju:

```
unsigned long int updateRefVector(PCB*);
```

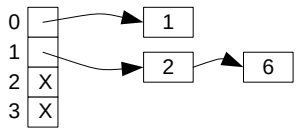
koja se poziva periodično, a koja treba da prebroji stranice kojima se pristupalo u poslednjoj periodi kako bi se odredila veličina aproksimacije radnog skupa za dati proces i taj broj vrati kao rezultat, a potom obriše ceo vektor bita referenciranja.

Rešenje:

```
unsigned long int updateRefVector(PCB* pcb) {
  if (pcb==0 || pcb->refBitVector==0) rerturn 0; // Exception!
  unsigned long int count=0;
  for (int i=0; i<RefBitVectorSize; i++) {
    unsigned int word = pcb->refBitVector[i];
    while (word!=0) {
      count += (word&1); // Mask and count LSB
      word >>= 1; // Shift right
    }
    pcb->refBitVector[i]=0;
  }
  return count;
}
```

35. (ispit 2010) Upravljanje memorijom

Psomatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 8 blokova koji su označeni brojevima 0..7. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici. Svaki ulaz *i* (*i*=0..3) ovog niza sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. U svakom elementu liste je broj prvog bloka u slobodnom komadu.



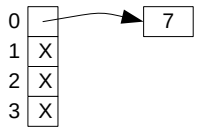
Nacrtati izgled ove strukture nakon što je, za prikazano početno stanje:

a)(5) Izvršena sukcesivno alokacija tri komada memorije veličine 2, 1 i 1 blok, tim redom.

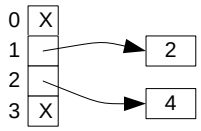
b)(5) Nakon stanja posle alokacije iz tačke a), izvršena sukcesivno dealokacija tri komada memorije koji počinju blokovima broj 2, 6 i 4, tim redom.

Rešenje:

a)(5)

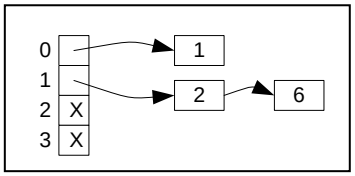


b)(5)



36. (ispit 2010) Upravljanje memorijom

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 2^{N-1} blokova koji su označeni brojevima $0..2^{N-1}-1$. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici, za primer $N = 4$. Svaki ulaz i (i=0..N-1) prikazanog niza `buddy` sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. Glava liste sadrži broj prvog bloka u slobodnom komadu, a broj narednog bloka u listi je upisan na početku svakog slobodnog bloka u listi (-1 za kraj liste). Deklaracije potrebnih struktura date su dole. Funkcija `block()` vraća adresu početka bloka broj `n`.




```
const int N = ...; // N>=1
int buddy[N];
void* block(int n);
```

Realizovati funkciju:

```
void buddy_init ();
```

koja inicijalizuje prikazanu strukturu za početno stanje alokatora u kome je ceo prostor od 2^{N-1} susednih blokova slobodan.

Rešenje:

```
void buddy_init () {
  for (int i=0; i<N-1; i++) buddy[i]=-1;
  buddy[N-1]=0;
  *((int*)block(0))=-1;
}
```

37. (ispit 2010) Upravljanje memorijom

Za izbor stranice za zamenu, neki sistem primenjuje modifikovani algoritam „časovnika“ (*clock*), tj. davanja druge šanse (*second-chance*), s tim da se svakoj stranici umesto proste Bulove vrednosti – bita referenciranja, pridružuje brojač koji se ažurira periodično tako što se inkrementira ukoliko se toj stranici u prethodnoj periodi pristupalo. Struktura koja se koristi da predstavi svaku stranicu u ovom algoritmu izgleda ovako:

```
struct Page {
  PID pid; // Process ID
  unsigned int page; // Page
  Page* next; next in the Clock Algorithm cyclic list
  unsigned int counter; // Counter of references
};

extern Page* clock; // Clock hand
```

Implementirati funkciju:

```
Page* getVictimPage();
```

koja pronalazi stranicu za zamenu.

Rešenje:

```
Page* getVictimPage () {
  while (clock && clock->counter) {
    clock->counter--;
    clock=clock->next;
  }
  return clock;
}
```

38. (ispit 2010) Upravljanje memorijom

Za izbor stranice za zamenu u nekom sistemu koristi se modifikovan algoritam „davanja nove šanse“ ili „časovnika“ (*second-chance, clock algorithm*), tako što se umesto bita referenciranja čuva brojač koji se za svaki okvir periodično uvećava za vrednost bita referenciranja, a okviru na koga naiđe kazaljka daje nova šansa ukoliko je brojač veći od nule, pri čemu se taj brojač tada dekrementira. Struktura koja čuva brojače implementirana je kao prosti niz (konstanta `FrameNum` predstavlja broj okvira):

```
const unsigned int FrameNum = ...;
unsigned int framecounters[FrameNum];
unsigned int clockHand;
```

Implementirati funkciju `getVictimFrame()` koja vraća broj okvira čiju stranicu treba zameniti po ovom algoritmu zamene.

Rešenje:

```
const unsigned int FrameNum = ...;
unsigned int frameCounters[FrameNum];
unsigned int clockHand;

unsigned int getVictimFrame () {
  while (fameCounters[clockHand]) {
    frameCounters[clockHand]--; // Give another chance
    clockHand=(clockHand+1)%FrameNum;
  }
  return clockHand;
}
```

39. (ispit 2010) Upravljanje memorijom

Data je sledeća sekvenca referenciranja stranica:
2, 0, 1, 2, 4, 5, 2, 3, 5, 4, 1, 0, 2, 4, 0, 3, 5, 1, 2, 0
Na raspolaganju je 4 okvira za ove stranice koji su inicijalno prazni. Odrediti za koliko je veći broj straničnih grešaka u slučaju da se koristi LRU algoritam zamene u odnosu na teorijski mimum.

Odgovor: Broj straničnih grešaka je veći za 4.
LRU: 14 straničnih grešaka. OPT: 10 straničnih grešaka.

40. (ispit 2011) Upravljanje memorijom

Neko jezgro za potrebe alokacije svojih struktura koristi sistem ploča (*slab allocator*). Posmatra se realizacija keša za alokaciju objekata neke klase `X` date dole. Kada u kešu više nema slobodnog mesta za alokaciju novog objekta klase `X`, nova ploča traži se od sloja ispod koji predstavlja alokator po sistemu parnjaka (*buddy*) operacijom

```
void* buddy_alloc(int i);
```

koja alocira ploču veličine 2^i susednih stranica i vraća pokazivač na alociranu ploču, odnosno 0 ako nije u mogućnosti da je alocira. Veličina ploče za keš klase `X` se izražava u stepenu dvojke *i* i konfiguracioni je parametar klase `X` (član `slabSizeInPages`). U jednu ploču može da stane *k* objekata klase `X`, gde je *k* takođe konfiguracioni parametar te klase (član `slabSizeInSlots`). Svi prazni pregradci u kešu klase `X` vezuju se u jednostruko ulančanu listu čija je glava u članu `pool`. Novi objekat se alocira prosto u prvom slobodnom pregradku, a prazne ploče ne treba oslobađati. Implementirati operacije `new` i `delete` klase `X`.

```
class X {
public:
  static void* operator new (size_t);
  static void  operator delete (void*);
private:
  static const unsigned int slabSizeInPages, slabSizeInSlots;
  static void* pool;
};
```

Rešenje:

```
class X {
public:
  static void* operator new (size_t);
  static void  operator delete (void*);
private:
  static const unsigned int slabSizeInPages, slabSizeInSlots;
  static void* pool;
};

void* X::operator new (size_t) {
  if (pool==0) {
    pool = buddy_alloc(slabSizeInPages);
    if (pool==0) return 0; // No more memory!
    void** p=(void**)pool;
    for (int i=0; i<slabSizeInSlots; i++,p=(void**)*p)
      *p = (char*)p + sizeof X;
  }
  void* ret = pool;
  pool = *(void**)pool;
  return ret;
}

void X::operator delete (void* p) {
  if (p==0) return;
  *(void**)p = pool;
  pool=p;
}
```

41. (ispit 2011) Upravljanje memorijom

U nekom sistemu primenjuje se tačan LRU algoritam lokalne zamene stranica uz odgovarajuću podršku hardvera. Stranice svakog pojedinačnog procesa koje taj proces koristi organizovane su u LRU stek implementiran kao dvostruko ulančana LRU lista. Na vrh steka (najskorije korišćenu stranicu) ukazuje poseban registar procesora, a brojevi prethodne i sledeće stranice u listi upisani su u deskriptoru stranice u PMT. Dat je isečak PMT za neki proces koji koristi prvih šest svojih stranica. Prikazati ovu tabelu u stanju nakon sledeće sekvence pristupa stranicama: 4, 1, 5.

page	*prev*	*next*
0	3	5
1	4	3
2	*null*	4
3	1	0
4	2	1
5	0	*null*

page	*prev*	*next*
0		
1		
2		
3		
4		
5		

Resenje:

page	*prev*	*next*
0	3	*null*
1	5	4
2	4	3
3	2	0
4	1	2
5	*null*	1

42. (ispit 2011) Upravljanje memorijom

Ako je neki sistem u stanju *thrashing*, kada mu je I/O podsistem izuzetno opterećen zamenom stranica, da li takvu situaciju može da popravi aktiviranje novih procesa za koje se zna da su izrazito *CPU-bound*? Obrazložiti.
Odgovor:

Ne. Naprotiv, može samo da je pogorša, jer novi procesi, čak i ako su *CPU-bound*, povećavaju potražnju za operativnom memorijom, čime još više povećavaju učestanost zamene stranica, a time i dodatno opterećuju I/O podsistem.

43. (ispit 2011) Upravljanje memorijom

Neki sistem koristi *clock* algoritam zamene stranica koji radi u opsegu skupa stranica pojedinačnog procesa (lokalno). Globalna struktura `pageMap` predstavlja niz ulaza, sa po jednim ulazom tipa `int` za svaki okvir fizičke memorije. Okviri koje koristi svaki proces su ulančani u zasebnu cirkularnu listu preko ulaza u nizu `pageMap`. Ovu cirkularnu listu koristi *clock* algoritam. U svakom tom ulazu najviši bit ima vrednost bita referenciranja (*reference bit*) stranice koja se nalazi u tom okviru, dok je neoznačena celobrojna vrednost u preostalim bitima indeks u nizu `pageMap` koji predstavlja sledeći okvir istog procesa u njegovoj cirkularnoj listi. Trenutni položaj „kazaljke“ je u polju `clock` tipa `int` strukture PCB svakog procesa. Implementirati funkciju `getVictim()` koja vraća indeks okvira u `pageMap` čija stranica je izabrana za izbacivanje za proces na čiji PCB ukazuje argument.

```
int getVictim (PCB*);
```

Rešenje:

```
int getVictim (PCB* proc) {
  if (proc==0) return -1; // Exception!
  while (pageMap[proc->clock]<0) {
    pageMap[proc->clock] &= MAXINT; // reset reference bit
    proc->clock = pageMap[proc->clock];
  }
  return proc->clock;
}
```

44. (ispit 2011) Upravljanje memorijom

U tabeli je prikazana struktura stranica nekog procesa uređenih po FIFO redosledu učitavanja. Vrsta *Pg* označava broj stranice, vrsta *Ref* trenutnu vrednost bita referenciranja, a vrsta *Nxt* broj stranice koja je sledeća u FIFO redu. Kurzor, odnosno „skazaljka“ u algoritmu zamene stranica je trenutno na stranici A.

Pg	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ref	1	1	1	0	1	0	1	0	1	0	1	0	1	1	0	1
Nxt	5	C	D	9	7	F	B	1	A	E	2	8	0	4	6	3

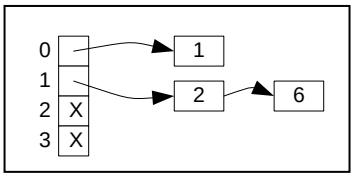
Koje su redom prve tri stranice žrtve za izbacivanje ako je algoritam zamene stranica:

a)(5) FIFO? Odgovor: A, 2, D

b)(5) *Clock*? Odgovor: 7, 5, 3

45. (ispit 2011) Upravljanje memorijom

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 2^{N-1} blokova koji su označeni brojevima $0..2^{N-1}-1$. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici, za primer $N = 4$. Svaki ulaz i ($i = 0..N-1$) prikazanog niza `buddy` sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. Glava liste sadrži broj prvog bloka u slobodnom komadu, a broj prvog bloka narednog slobodnog komada u listi je upisan na početku svakog slobodnog komada (-1 za kraj liste). Deklaracije potrebnih struktura date su dole. Funkcija `block()` vraća pokazivač na početak bloka broj `n`, a funkcija `buddy(n,i)` vraća broj prvog bloka komada-parnjaka datog komada memorije veličine 2^i susednih blokova koji počinje blokom `n` (-1 u slučaju greške, ako blok broj `n` nije regularan početak komada veličine 2^i blokova). Na primer: `buddy(1,0)==0`, `buddy(1,1)==-1`, `buddy(2,1)==0`.




```
const int N = ...; // N>1
int buddy[N];
int* block(int n);
int  buddy(int n, int i);
```

Realizovati funkciju:

```
int buddy_free(int n, int i);
```

koja oslobađa komad veličine 2^i susednih blokova ($0 \le i < N$) koji počinje blokom `n` i vraća 0 u slučaju uspeha, a -1 u slučaju greške.

Rešenje:

```
int buddy_free (int n, int i) {
  if (i<0 || i>=N) return -1; // Error: i out of range
  int* pn=block(n);
  if (pn==0) return -1; // Error: illegal block n
  int nb = buddy(n,i); // Find the buddy of n
  if (nb==-1) return -1; // Error: mismatching n and i

  // First, try to find the buddy of block n in the list buddy[i]:
  int prev=-1, cur=buddy[i];
  while (cur!=-1 && cur!=nb) {
    prev=cur;
    int* pc=block(cur);
    if (pc==0) return -1; // Unexpected error: corrupted structure
    cur=*pc;
  );

  if (cur==-1) { // Not found; just add block n to the list buddy[i]
    *pn=buddy[i];
    buddy[i]=n;
    return 0;
  }

  // Found the buddy. Remove it from buddy[i]
  int* pc=block(cur);
  if (pc==0) return -1; // Unexpected error: corrupted structure
  if (prev!=-1) {
    int* pp = block(prev);
    if (pp==0) return -1; // Unexpected error: corrupted structure
    *pp=*pc;
  } else
    buddy[i]=*pc;
  *pc=-1;

  // Then join n and its buddy nb to one block nm and add it to buddy[i+1]
  int nm=n<nb?n:nb; // nm=min(n,nb)
  if (i<N-1)
    return buddy_free(nm,i+1); // Recursion
  else
    return -1; // Unexpected error: should never occur.
}
```

1. (Decembar 2006) Upravljanje diskovima

a)(5) U red zahteva za diskom pristigli su zahtevi za pristup sledećim cilindrima (po redosledu pristizanja): 17, 54, 53, 62, 12, 73, 24, 26, 18
Kojim redosledom će ovi zahtevi biti opsluženi ako je algoritam SSTF, a glava na početku iznad cilindra 27?

Rešenje: 26, 24, 18, 17, 12, 53, 54, 62, 73

b)(5) Kratko i precizno objasniti zašto je RAID 5 efikasniji u pogledu raspoloživog prostora za podatke u odnosu na RAID 0+1.

Odgovor:

Sa ukupno 2N jednakih diskova, kod RAID 5 prostor za podatke je ukupnog kapaciteta 2N-1 diskova, dok je kod RAID 0+1 taj prostor kapaciteta N diskova.

2. (Decembar 2007) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
56, 27, 89, 124, 64, 25, 45
Prethodno opsluženi zahtev je bio na cilindru 40, a glava se kreće prema blokovima sa nižim brojem. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: 45, 56, 64, 89, 124, 27, 25

b)(5) *Scan*

Odgovor: 27, 25, 45, 56, 64, 89, 124

3. (Decembar 2008) Upravljanje diskovima

Precizno objasniti zbog čega je RAID5 efikasniji od RAID4, odnosno šta je osnovni problem RAID4 zbog kojeg je prevaziđen od strane RAID5?

Odgovor:
Kod RAID4 bit parnosti se uvek smešta na jedan isti, određeni disk. Svaki zahtev za upisom podataka na disk zahteva pristup do diska sa podacima, kao i do diska sa bitima parnosti koje treba ažurirati prilikom svake izmene podataka. Zbog toga taj disk sa bitima parnosti postaje usko grlo, pošto se na njemu gomilaju zahtevi za upis (ostali zahtevi za upis na diskove sa podacima bi se inače mogli paralelizovati). RAID5 ovaj problem rešava tako što su blokovi sa bitima parnosti ciklično raspoređeni po svim diskovima, pa ni jedan nije posebno usko grlo.

4. (Decembar 2009) Upravljanje diskovima

Neki disk drajver sprovodi *C-Look* algoritam opsluživanja zahteva. Jedan zahtev za operaciju sa diskom predstavljen je strukturom `DiskOpReq` u kojoj član `cyl` predstavlja broj cilindra na koji se zahtev odnosi. Zahtevi su smešteni u dvostruko ulančanu listu predstavljenu klasom `DiskQueue`:

```
struct DiskOpReq {
  unsigned int cyl;
  ...
  DiskOpReq *prev, *next;
};

class DiskQueue {
public:
  DiskQueue () : head(0), tail(0), nextToServe(0) {}
  DiskOpReq* getReq ();
  void addReq (DiskOpReq*);
private:
  DiskOpReq *head, *tail, *nextToServe;
};
```

Pokazivač `nextToServe` ukazuje na zahtev u redu koji naredni treba opslužiti. Operacija `getReq` vraća zahtev koji naredni treba opslužiti i izbacuje ga iz reda, a operacija `addReq` stavlja novi zahtev u red. Implementirati ove dve operacije.

Rešenje:

```
DiskOpReq* DiskQueue::getReq () {
  if (nextToServe==0) return 0; // Queue empty
  DiskOpReq* cur = nextToServe;
  nextToServe = nextToServe->next;
  // Remove the pending request:
  if (cur->prev) cur->prev->next=cur->next;
  else head=cur->next;
  if (cur->next) cur->next->prev=cur->prev;
  else tail=cur->prev;
  if (nextToServe==0) nextToServe=head;
  return cur;
}

void DiskQueue::addReq (DiskOpReq* req) {
  if (req==0) return; // Error
  if (head==0) { // Queue empty
    nextToServe=head=tail=req;
    return;
  }
  // Insert the request to keep the list sorted
  for (DiskOpReq* cur=head; cur!=0; cur=cur->next) {
    if (cur->cyl<=req->cyl) continue;
    if (cur->prev) {
      req->prev = cur->prev;
      req->next = cur;
      req->prev->next=req;
      req->next->prev = req;
```

```
    } else {
      // Add to the head:
      req->prev = 0;
      req->next = cur;
      head=req;
      req->next->prev = req;
    }
    return;
  }
  // Add to the tail:
  req->prev = tail;
  req->next = 0;
  req->prev->next=req;
  tail = req;
}
```

5. (Decembar 2010) Upravljanje diskovima

a)(5) U redu zahteva za pristup disku nalaze se zahtevi koji se odnose na sledeće cilindre, redom kojim su postavljeni:
189, 25, 68, 23, 76, 157, 64, 17, 200, 130
Napisati redosled kojim će zahtevi biti opsluženi, ako se primenjuje algoritam *C-LOOK*, glava je trenutno na cilindru broj 100, a zahtevi se opslužuju u hodu glave prema višim cilindrima.

Odgovor: 130, 157, 189, 200, 17, 23, 25, 64, 68, 76

b)(5) Porede se RAID strukture nivoa 2 i nivoa 3 za isti broj fizičkih diskova. Navesti osnovnu prednost svake u odnosu na onu drugu.

Odgovor:

RAID 2 je otporan na otkaz više od jednog diska, jer dodatni ECC biti na redundantnim diskovima mogu da restauriraju otkaz više od jednog bita u pruzi. RAID 3 je otporan samo na otkaz jednog diska, jer je samo bit parnosti redundantan. RAID 3 ima veći efektivan kapacitet za isti broj fizičkih diskova N (kapacitet je $(N\text{-}1)/N$) nego RAID 2 ($N\text{-}k/N$, k je broj ECC diskova i veći je od 1).

6. (Novembar 2012) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
62, 43, 95, 130, 70, 41, 51
Prethodno opsluženi zahtev je bio na cilindru 46, a glava se kreće prema cilindrima sa većim
brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: 43, 41, 51, 62, 70, 95, 130

b)(5) *C-Scan*

Odgovor: 51, 62, 70, 95, 130, 41, 43

7. (Septembar 2013) Upravljanje diskovima

U nekom operativnom sistemu zahtevi za operacije sa diskom rasporeÿuju se po SSF (*Shortest Seek First*) algoritmu koji je implementiran na sledeći način. Svi zahtevi se smeštaju u istu, dvostruko ulančanu i neureÿenu listu. Novi zahtev se uvek smešta na početak liste.

a)(5) Koja je kompleksnost u odnosu na broj zahteva za operacije sa diskom n za operaciju stavljanja novog zahteva u listu i za operaciju izbora zahteva koji će naredni biti opslužen? Precizno obrazložiti.

Odgovor:

Stavljanje novog zahteva: $O(1)$, jer se zahtev stavlja na početak liste (promena nekoliko pokazivača, nezavisno od veličine liste). Izbor zahteva: $O(n)$, linearna pretraga cele liste da se pronaÿe zahtev na minimalnom rastojanju od upravo opsluženog.

b)(5) Predložiti neku implementaciju ovog rasporeÿivanja koja će imati kompleksnost $O(1)$
za obe navedene operacije (u odnosu na broj zahteva za operacije sa diskom n). Može se pretpostaviti da je broj cilindara na disku poznat i ne preveliki (reda do nekoliko hiljada). Precizno obrazložiti.

Odgovor:

Na primer, kreirati niz (statički alociran, veličine jednake broju cilindara na disku C) u kome je svaki element k glava liste zahteva koji se odnose na cilindar k. Novi zahtev koji se
odnosi na cilindar k se stavlja na početak liste u ulazu k. Pretraga za najbližim zahtevom ide
iterativno kroz niz, počev od ulaza koji odgovara cilindru upravo opsluženog zahteva k i proverava najpre susedne ulaze (k-1 i k+1), pa onda sledeće (k-2 i k+2), itd. sve dok ne naiÿe
na neprazan ulaz. Ovaj postupak ima složenost $O(C)$, ali je složenosti $O(1)$ u odnosu na broj zahteva n.

8. (ispit 2006) Upravljanje diskovima

a)(5) Neki *storage* sistem sa više diskova, visoke pouzdanosti, označen je na sledeći način: RAID-5/12+1, pri čemu je kapacitet svakog diska 100GB. Koliki je efektivni kapacitet (za „korisne“ informacije koje koristi fajl sistem) ove strukture diskova?

Odgovor: _____

b)(5) Neki *storage* sistem sa više diskova, visoke pouzdanosti, označen je na sledeći način: RAID-1/2x4x100GB, pri čemu je kapacitet svakog diska 100GB. Koliki je efektivni kapacitet (za „korisne“ informacije koje koristi fajl sistem) ove strukture diskova?

Odgovor: _____

a)(5) 12 * 100GB = 1200GB
b)(5) 4 * 100GB = 400GB

9. (ispit 2006) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
56, 27, 89, 124, 64, 25, 45
Prethodno opsluženi zahtev je bio na cilindru 40. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: _____

b)(5) *Scan*

Odgovor: _____

a)(5) 45, 56, 64, 89, 124, 27, 25
b)(5) 45, 56, 64, 89, 124, 27, 25

10. (ispit 2006) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
56, 37, 89, 124, 64, 35, 45
Prethodno opsluženi zahtev je bio na cilindru 40, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: _____

b)(5) *C-Scan*

Odgovor: _____

a)(5) 37, 35, 45, 56, 64, 89, 124
b)(5) 45, 56, 64, 89, 124, 35, 37

11. (ispit 2006) Virtuelne mašine

a)(5) Smisao virtuelnih mašina za određene programske jezike, kao što su Java VM ili .Net CLR, jeste da korisničke programe pisane na tim jezicima i prevedene na međukod za te virtuelne mašine učine nezavisnim od platforme (računara i operativnog sistema domaćina) i time prenosive na različite platforme. Ali da li je sama takva virtuelna mašina zavisna od platforme? Zašto?

Odgovor:

Virtuelna mašina jeste zavisna od platforme, jer je to program (u binarnom obliku) koji se izvršava od strane mašine-domaćina (program se sastoji od naredbi baš za taj računar) i na operativnom sistemu domaćinu (koristi sistemske pozive tog OS-a).

b)(5) Virtuelna mašina za neki programski jezik (npr. Java VM) se izvršava kao jedan proces na sistemu-domaćinu, a u okviru svog izvršavanja pokreće niti koje su definisane u programu na izvornom jeziku koji VM izvršava. Da li to obavezno znači da ako se neka nit iz programa blokira na nekom sistemskom pozivu (npr. I/O operaciji), da se i ceo proces VM (cela virtuelna mašina) blokira?

Odgovor:

Ne mora da znači. VM može da preslikava (implementira) niti iz programa u niti operativnog sistema-domaćina, ako takve postoje. U tom slučaju, blokiranje niti iz programa znači samo blokrianje jedne niti OS domaćina, ali ne i ostalih niti u okviru istog VM procesa.

12. (ispit 2006) Upravljanje diskovima

a)(5) Šta je osnovni nedostatak RAID 1 u odnosu na RAID 5?

Odgovor:

RAID 1 je jednostavniji za implementaciju, ali ima slabije iskorišćenje (ukupan prostor je duplo veći u odnosu na količinu korisnih informacija, odnos je 2:1), dok je kod RAID 5 taj odnos povoljniji i iznosi n:n-1.

b)(5) Objasniti kako SSTF algoritam raspoređivanja zahteva za pristup disku može da dovede do izgladnjivanja?

Odgovor:

Tako što stalno stižu novi zahtevi koji se odnose na cilindre bliske onome na kome se nalazi glava, dok ostali zahtevi za udaljenim cilindrima ostaju da čekaju.

13. (ispit 2007) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
56, 37, 89, 124, 64, 35, 45
Prethodno opsluženi zahtev bio je na cilindru 50, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) SSTF (*Shortest-Seek-Time-First*)

Odgovor: 45, 37, 35, 56, 64, 89, 124

b)(5) *Scan*

Odgovor: 56, 64, 89, 124, 45, 37, 35

14. (ispit 2007) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
56, 37, 89, 124, 64, 35, 45
Prethodno opsluženi zahtev bio je na cilindru 50, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Look*
Odgovor: 56, 64, 89, 124, 45, 37, 35

b)(5) *C-Look*

Odgovor: 56, 64, 89, 124, 35, 37, 45

15. (ispit 2007) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
58, 39, 91, 126, 66, 37, 47
Prethodno opsluženi zahtev bio je na cilindru 52, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Scan*

Odgovor: 58, 66, 91, 126, 47, 39, 37

b)(5) SSTF (*Shortest-Seek-Time-First*)

Odgovor: 47, 39, 37, 58, 66, 91, 126.

16. (ispit 2007) Upravljanje diskovima

Neki sistem diskova označen je sa RAID5 16x1TB i ima 16 fizičkih diskova. Kolika efektivna količina podataka može da se smesti na ovaj sistem? Obrazložiti.

Odgovor:

15TB. Od 16 diskova po 1TB, jedan se (u RAID5 ciklično) koristi kao redundansa za smeštanje koda za detekciju greške (parnosti).

17. (ispit 2007) Upravljanje diskovima

Za neki sistem masivnog skladištenja podataka, efektivnog kapaciteta od N pojedinačnih diskova (N je mnogo više od jedan), neophodno je obezbediti i visoku pouzdanost i dobre performanse, ali je cena primarni faktor. Za koji sistem biste se radije opredelili, RAID 1+0 ili RAID 5? Obrazložiti.

Odgovor:

RAID 5. I RAID 1+0 i RAID 5 obezbeđuju i povećanu pouzdanost i poboljšane performanse u odnosu na prost sistem sa N pojedinačnih diskova bez RAID, s tim da je RAID 5 jeftiniji jer ima znatno manje diskova (N+1 umesto 2N).

18. (5. januar 2008.) Upravljanje diskovima

Neki sistem diskova ima 12 fizičkih diskova, svaki je veličine 1 TB. Koliki je efektivni kapacitet ovog sistema, ako je on:

a)(2) RAID 0

b)(2) RAID 1

c)(2) RAID 0+1

d)(2) RAID 1+0

e)(2) RAID 5

Odgovor:

a) 12 TB

b) 6 TB

c) 6 TB

d) 6 TB

e) 11 TB

19. (5. februar 2008.) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
46, 27, 79, 114, 54, 25, 35
Prethodno opsluženi zahtev je bio na cilindru 30, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

b)(5) *C-Scan*

Odgovor:

a) 27, 25, 35, 46, 54, 79, 114
b) 35, 46, 54, 79, 114, 25, 27

20. (ispit 2009) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
34, 15, 67, 102, 42, 13, 23
Prethodno opsluženi zahtev je bio na cilindru 18, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: _____

b)(5) *C-Scan*

Odgovor: _____

a)(5) 15, 13, 23, 34, 42, 67, 102 b)(5) 23, 34, 42, 67, 102, 13, 15

21. (ispit 2009) Upravljanje diskovima

a)(5) Šta je osnovni nedostatak RAID 1 u odnosu na RAID 5?

Odgovor:

b)(5) Objasniti kako SSTF algoritam raspoređivanja zahteva za pristup disku može da dovede do izgladnjivanja?

Odgovor:

a)(5) RAID 1 je jednostavniji za implementaciju, ali ima slabije iskorišćenje (ukupan prostor je duplo veći u odnosu na količinu korisnih informacija, odnos je 2:1), dok je kod RAID 5 taj odnos povoljniji i iznosi N:(N-1).

b)(5) Tako što stalno stižu novi zahtevi koji se odnose na cilindre bliske onome na kome se nalazi glava, dok ostali zahtevi za udaljenim cilindrima ostaju da čekaju.

22. (ispit 2009) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
58, 39, 91, 126, 66, 37, 47
Prethodno opsluženi zahtev bio je na cilindru 52, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Scan*

Odgovor: 58, 66, 91, 126, 47, 39, 37

b)(5) SSTF (*Shortest-Seek-Time-First*)

Odgovor: 47, 39, 37, 58, 66, 91, 126

23. (ispit 2009) Upravljanje diskovima

Precizno objasniti zbog čega je RAID 5 uspešniji nego RAID 3 ili RAID 4.

Odgovor:

RAID 3, 4 i 5 imaju istu efikasnost u pogledu iskorišćenja prostora: za N diskova sa efektivnim kapacitetom potrebno je ukupno N+1 diskova. Međutim, kod RAID 3 i 4 se uvek jedan isti disk koristi za smeštanje koda za detekciju greške. Kako se ovi podaci ažuriraju prilikom svakog upisa efektivnih podataka na bilo koji drugi disk, ovaj disk je posebno opterećen jer svaki upis na RAID strukturu uvek ide i na njega. Zato je taj disk usko grlo. Kod RAID 5 blokovi sa kodom za detekciju greške se ravnomerno raspoređuju po svim diskovima, ciklično, pa ni jedan nije usko grlo zbog toga. Zato RAID 5 ima bolje performanse.

24. (ispit 2010) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
59, 40, 92, 127, 67, 38, 48
Prethodno opsluženi zahtev je bio na cilindru 43, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: _____

b)(5) *C-Scan*

Odgovor: _____

a)(5) 40, 38, 48, 59, 67, 92, 127
b)(5) 48, 59, 67, 92, 127, 38, 40

25. (ispit 2010) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
35, 16, 68, 103, 43, 14, 24
Prethodno opsluženi zahtev bio je na cilindru 29, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Look*

Odgovor: 35, 43, 68, 103, 24, 16, 14

b)(5) *C-Look*

Odgovor: 35, 43, 68, 103, 14, 16, 24

26. (ispit 2011) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
21, 2, 54, 89, 29, 1, 10
Prethodno opsluženi zahtev bio je na cilindru 15, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *C-Look*

Odgovor: 21, 29, 54, 89, 1, 2, 10

b)(5) *Look*

Odgovor: 21, 29, 54, 89, 10, 2, 1

27. (ispit 2011) Upravljanje diskovima

a)(5) Neki *storage* sistem sa više diskova, visoke pouzdanosti, označen je na sledeći način: RAID5 (6+1), pri čemu je kapacitet svakog diska 2TB. Koliki je efektivni kapacitet (za „korisne“ informacije koje koristi fajl sistem) ove strukture diskova?

Odgovor: 6·2TB = 12TB

b)(5) Neki *storage* sistem sa više diskova, visoke pouzdanosti, označen je na sledeći način: RAID0+1 (2x8), pri čemu je kapacitet svakog diska 2TB. Koliki je efektivni kapacitet (za „korisne“ informacije koje koristi fajl sistem) ove strukture diskova?

Odgovor: 8·2TB = 16TB

28. (ispit 2011) Upravljanje diskovima

a)(5) Šta je osnovni nedostatak RAID 1 u odnosu na RAID 5?

Odgovor:
RAID 1 je jednostavniji za implementaciju, ali ima slabije iskorišćenje (ukupan prostor je duplo veći u odnosu na količinu korisnih informacija, odnos je 2:1), dok je kod RAID 5 taj odnos povoljniji i iznosi n:n-1.

b)(5) Objasniti kako SSTF algoritam raspoređivanja zahteva za pristup disku može da dovede do izgladnjivanja?

Odgovor:

Tako što stalno stižu novi zahtevi koji se odnose na cilindre bliske onome na kome se nalazi glava, dok ostali zahtevi za udaljenim cilindrima ostaju da čekaju.

29. (ispit 2011) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
61, 41, 93, 128, 68, 39, 49
Prethodno opsluženi zahtev bio je na cilindru 54, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Look.* Odgovor: 61, 68, 93, 128, 49, 41, 39

b)(5) *C-Look.* Odgovor: 61, 68, 93, 128, 39, 41, 49

30. (ispit 2011) Upravljanje diskovima

U redu zahteva za pristup disku nalaze se zahtevi za pristup sledećim cilindrima (po redosledu pristizanja):
47, 28, 80, 115, 55, 26, 36
Prethodno opsluženi zahtev je bio na cilindru 31, a glava se kreće prema cilindrima sa većim brojevima. Napisati redosled opsluživanja ovih zahteva ukoliko je algoritam raspoređivanja:

a)(5) *Shortest-Seek-Time-First*

Odgovor: 28, 26, 36, 47, 55, 80, 115

b)(5) *C-Scan*

Odgovor: 36, 47, 55, 80, 115, 26, 28

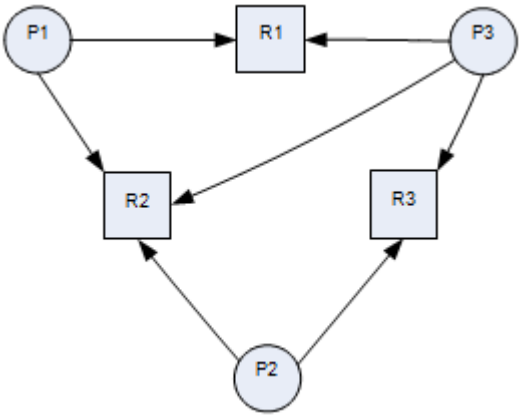
1. (Novembar 2011) Mrtva blokada

Neki sistem primenjuje algoritam izbegavanja mrtve blokade (*deadlock avoidance*) pomoću grafa alokacije. Na slici je dat graf početnog stanja sistema sa tri procesa i tri resursa.

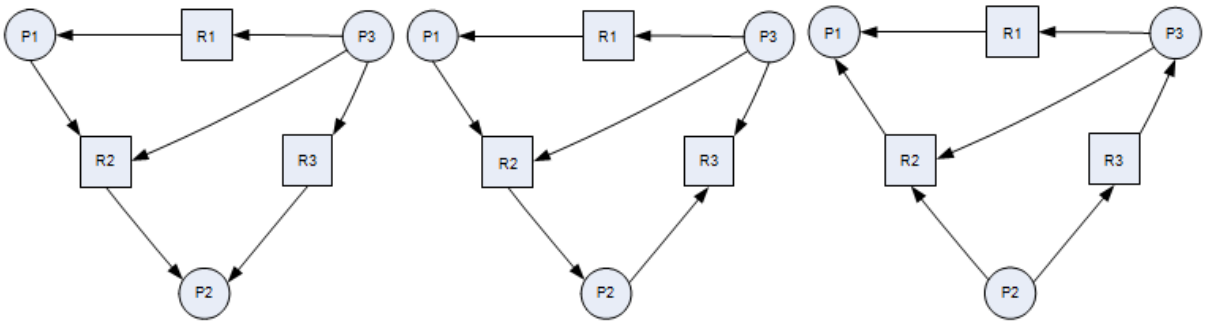
a)(3) Nacrtati graf stanja sistema nakon sledeće sekvence:
P2.request(R3), P1.request(R1), P2.request(R2), P3.request(R3)

b)(3) Nakon sekvence date pod a), P2 oslobađa resurs R3. Nacrtati graf stanja sistema nakon ove operacije.

c)(4) Nakon toga, sekvenca se nastavlja ovako: P1.request(R2), P2.release(R2). Nacrtati graf stanja nakon toga.



Rešenja:



2. (Septembar 2012) Mrtva blokada

U nekom sistemu svaki proces i svaki resurs ima svoj jedinstveni identifikator (tip a `unsigned int`), a zauzeće resursa od strane procesa prati se u matrici `resourceAlloc` u kojoj vrste označavaju procese, a kolone resurse. U toj matrici vrednost 1 u ćeliji (*p*, *r*) označava da je proces sa identifikatorom *p* zauzeo resurs sa identifikatorom *r*, a vrednost 0 označava da proces *p* nije zauzeo resurs *r*. Mrtva blokada sprečava se tako što se procesu dozvoljava zauzimanje resursa samo u opadajućem redosledu vrednosti identifikatora resursa. Zato se, kada proces *p* zatraži resurs *r*, poziva operacija `allocate(p,r)`:

```
const unsigned MAXPROC = ...; // Maximum number of processes
const unsigned MAXRES = ...;  // Maximum number of resources
extern unsigned numOfProc;    // Actual number of processes
extern unsigned numOfRes;     // Actual number of resources

int resourceAlloc[MAXPROC][MAXRES];

int allocate (unsigned pid, unsigned rid);
```

Ova operacija proverava da li se procesu *p* može i sme dodeliti resurs *r* i zauzima ga, ako može. U tom slučaju ova operacija treba da vrati 1. Ako se procesu ne može dodeliti resurs, treba vratiti 0. Implementirati operaciju `allocate`.

Rešenje:

```
int  allocate (int pid, int rid) {
  if (pid>=numOfProcesses || rid>=numOfResources) return 0; // Exception!
  if (resourceAlloc[pid][rid]) return 1; // Already allocated to this proc.
  // Is this resource free?
  for (unsigned i=0; i<numOfProcesses; i++)
  if (resourceAlloc[i][rid]) return 0; // The resource is occupied
  // Deadlock prevention:
  for (i=0; i<rid; i++)
  if (resourceAlloc[pid][i]) return 0; // Cannot allocate
  // The resource can be allocated:
  resourceAlloc[pid][rid] = 1;
  return 1;
}
```

3. (Septembar 2013) Mrtva blokada

U nekom sistemu su tri procesa (P_1, P_2, P_3) i tri različite instance resursa (R_1, R_2, R_3). 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)

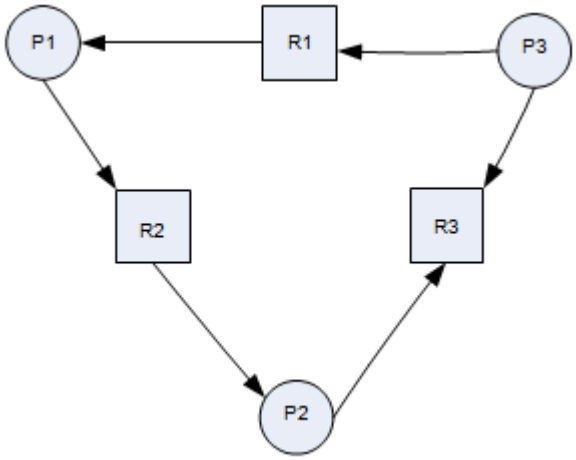
Procesi najavljuju korišćenje onih i samo onih resursa koje zauzimaju u datoj sekvenci.

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?

Rešenje:

a)(7) Sekvenca: P2.request(R3), P1.request(R1), P2.request(R2), P3.request(R3), P2.release(R3), P1.request(R2).



b)(3) Proces P3 će dobiti resurs R3 kada proces P2 oslobodi resurs R2.

1. (Decembar 2007) Korisnički interfejs operativnih sistema

U nekom operativnom sistemu interpreter komandne linije napravljen je tako da bude lako proširiv. On jednostavno učitava niz znakova sa komandne linije, razdvaja reči u njoj, prvu tretira kao naziv programa koga pronalazi na predefinisanom mestu i nad kojim pokreće proces, a ostale prosleđuje tom procesu kao argumente. Precizno objasniti kako za ovakav sistem implementirati drugačiju školjku (*shell*), odnosno podržati drugačiji skup komandi i da li za to treba menjati postojeći interpreter. Posebno prokomentarisati kako se implementira povratak iz nove u staru školjku.

Odgovor:

U zavisnosti od toga da li se predefinisano mesto na kome postojeći interpreter podrazumevano traži programe za komande može menjati dinamički ili ne (tj. da li postojeći interpreter očitava taj parametar prilikom interpretacije svake komande ili samo prilikom inicijalizacije), kao i u zavisnosti od toga da li nova školjka koristi isti stil (format) komandne linije (komanda u jednoj reči, iza nje parametri) ili sasvim drugačiji format, ovo se može realizovati na nekoliko različitih načina. U svakom od njih, nova školjka aktivira se jednostavno pokretanjem odgovarajućeg programa (koji se u postojećoj školjki vidi kao komanda). U zavisnosti od varijante, ona radi sledeće:

- Ukoliko nova školjka koristi isti format komandne linije, a predefinisano mesto na kome se komande (kao programi) traže u fajl sistemu se može izmeniti dinamički, ovaj program može prosto samo da preusmeri to mesto na direktorijum sa komandama nove školjke. Povratak u staru školjku je komanda nove školjke koja vraća ovaj parametar na staru vrednost.
- U suprotnom, ovaj program je novi interpreter komandne linije koji sam učitava i parsira komandnu liniju sa standardnog ulaza, a onda izvršava komande i ispisuje rezultate na standardni izlaz. Povratak na staru školjku je izlaz iz ovog programa (gašenje procesa novog interpretera).

2. (Decembar 2008) Arhitektura operativnih sistema

Od ponuđenih usluga, odnosno funkcionalnosti jednog mikrokernelskog operativnog sistema, koje biste realizovali kao sistemske pozive, a koje kao usluge višeg nivoa – kao sistemske biblioteke ili sistemske programe? (Samo zaokružiti jedno ili drugo.)

Kopiranje fajla	Sistemski poziv	Sistemska biblioteka ili program
Kreiranje/otvaranje fajla	Sistemski poziv	Sistemska biblioteka ili program
Čitanje iz/upis u fajl	Sistemski poziv	Sistemska biblioteka ili program
Ispis niza znakova na uređaj	Sistemski poziv	Sistemska biblioteka ili program
Ispisivanje sadržaja txt fajla na standardni izlaz	Sistemski poziv	Sistemska biblioteka ili program

Odgovor:

Kopiranje fajla		Sistemska biblioteka ili program
Kreiranje/otvaranje fajla	Sistemski poziv	
Čitanje iz/upis u fajl	Sistemski poziv	
Ispis niza znakova na uređaj	Sistemski poziv	
Ispisivanje sadržaja txt fajla na standardni izlaz		Sistemska biblioteka ili program

3. (Decembar 2009) Arhitektura operativnih sistema

U nekom operativnom sistemu koji podržava virtuelnu memoriju, sistemski pozivi realizovani su prostim indirektnim pozivom (tj. dinamičkim vezivanjem) preko tabele pokazivača na funkcije operativnog sistema koje realizuju sistemske pozive i koje se izvršavaju u istom adresnom prostoru pozivajućeg procesa.

a)(5) Objasniti kako je moguće da se isti kod kernela izvršava u različitim adresnim prostorima korisničkih procesa, nad istim strukturama podataka koje su jedinstvene za ceo kernel.

b)(5) Realizovati funkciju iz sistemske biblioteke koja vrši poziv sistemske usluge sa brojem koji je dat kao prvi argument i parametrima složenim u strukturu na koju ukazuje drugi argument.

Rešenje:

a)(5) Deljenjem stranica. Stranice sa kodom i podacima kernela preslikavaju se u fiksni deo virtuelnog adresnog prostora svakog procesa (npr. najniži deo), a okviri sa tim sadržajem se dele između procesa. Ovo obezbeđuje sam kernel prilikom kreiranja procesa.

b)(5)

```
typedef int (*SYS_CALL)(void*);
SYS_CALL sys_call_table[...]; // Table of pointers to sys call functions
const unsigned int SYS_CALL_TABLE_SIZE = ...;

int sys_call (unsigned int id, void* params) {
  if (id>=SYS_CALL_TABLE_SIZE) return -1; // Error
  return sys_call_table[id](params);
}
```

4. (Decembar 2010) Arhitektura operativnih sistema

Neki operativni sistem ima mikrokernelsku arhitekturu. Mikrokernel obezbeđuje samo veoma jednostavan fajl sistem u kome poznaje samo pojam fajla i omogućava pristup do FCB struktura na uređaju, kao i mapiranje sadržaja fajla u memoriju (*memory mapped file*), bez poznavanja pojma strukture direktorijuma. Svim fajlovima na nekoj particiji se pristupa direktno preko identifikatora FCB strukture (`FHANDLE`), a njihovom sadržaju preslikavanjem u adresni prostor procesa. Moguće je samo posebnim flegom u FCB označiti da se sadržaj fajla tumači kao direktorijum, ali kernel ne tumači taj sadržaj fajla. Deklaracije jedinih sistemskih poziva koji su na raspolaganju za pristup fajlovima jesu sledeće:

```
typedef unsigned long FHANDLE; // File (FCB) identifier
int dir_isdir(FHANDLE);  // Is FCB marked as a directory?
void* file_map(FHANDLE); // Map the given file to memory
void file_unmap(void*);  // Unmap file and release memory
```

Funkcija `dir_isdir()` vraća 1 ako je u FCB postavljen fleg koji označava direktorijum, a 0 ako nije. Funkcija `file_map()` preslikava sadržaj datog fajla na prvo slobodno (nealocirano) mesto u virtuelnom adresnom prostoru pozivajućeg procesa i vraća adresu tog mesta u memoriji u slučaju uspeha, a 0 u slučaju greške. Funkcija `file_unmap()` ukida preslikavanje fajla na datoj adresi u memoriji i dealocira taj deo memorije tako da ga proces ne može više koristiti (osim ako ga ponovo ne preslika u neki fajl ili drugačije alocira novim sistemskim pozivom).

Manipulacija hijerarhijskim strukturama direktorijuma podržana je u sistemskim bibliotekama čiji se kod i strukture podataka izvršavaju i nalaze u kontekstu i adresnom prostoru korisničkih procesa. Slično važi i za pojam tekućeg direktorijuma – on je u kontekstu procesa, a ne čuva ga kernel. Na raspolaganju su sledeće sistemske bibliotečne funkcije:

```
FHANDLE dir_curdir();  // Returns current directory
void dir_chdir(FHANDLE);  // Change current directory
FHANDLE dir_find(void* dir, char* fname);  // Find entry in directory
```

Funkcija `dir_curdir()` vraća `FHANDLE` „tekućeg direktorijuma“ pozivajućeg procesa, a `dir_chdir()` postavlja „tekući direktorijum“ na dati `FHANDLE`. Kada se sadržaj nekog fajla koji jeste direktorijum preslika u memoriju na adresu `dir`, onda funkcija `dir_find()` pronalazi u strukturi sadržaja tog fajla (direktorijuma) ulaz sa imenom koje je dato u `fname` i vraća njegov `FHANDLE`. Naziv ulaza se uzima iz prvih znakova na koje ukazuje `fname`, sve dok se ne naiđe na znak `'/'` ili `'\0'`.

Realizovati sistemsku bibliotečnu funkciju:

```
int dir_chdir (char* dirname);
```

koja treba da promeni tekući direktorijum na onaj zadat datom stazom i vrati 0 u slučaju uspeha, -1 u slučaju greške. Staza može početi znakom `'/'` koji označava „koreni“ direktorijum, ili nekim drugim znakom, kada je staza relativna u odnosu na tekući deirektorijum. „Koreni“ direktorijum je uvek u fajlu čiji je `FHANDLE` jednak 0.

Rešenje:

```
int dir_chdir (char* dirname) {
  if (dirname==0) return -1; // Error in argument
  FHANDLE fhandle = dir_curdir();
  if (*dirname=='/')  fhandle = 0, dirname++;
  while (*dirname) {
    void* dir = file_map(fhandle);
    if (dir==0) return -1; // Error in memory mapping the file
    fhandle = dir_find(dir,dirname);
    file_unmap(dir);
    if (fhandle==-1) return -1; // Directory not found
    if (!dir_isdir(fhandle)) return -1; // Not a directory
    while (*dirname!='/' && *dirname!='\0') dirname++;
    if (*dirname=='/') dirname++;
  };
  dir_chdir(fhandle);
  return 0;
}
```

5. (Septembar 2013) Memorijski preslikani fajlovi

U nekom operativnom sistemu podržan je koncept memorijski preslikanih fajlova (engl. *memory mapped files*).
Podsistem za upravljanje fajlovima u svom interfejsu, izmeÿu ostalog, poseduje i funkciju:

```
int fread(FHandle fh, void* buffer, unsigned offset, unsigned size);
```

koja iz sadržaja fajla sa datom ručkom, počev od date pozicije `offset` (u bajtovima, od početnog bajta 0 sadržaja fajla), učitava niz bajtova date dužine `size` u zadati bafer `buffer`.

Svaki logički segment memorije nekog procesa koji se preslikava u fajl opisan je strukturom `MMFSegment` čija je delimična definicija data dole. Polja `pgLow` i `pgHigh` predstavljaju brojeve prve i poslednje stranice u tom segmentu. Polje `fh` je ručka fajla u koji je preslikan dati segment memorije. Ova polja postavlja inicijalizaciona procedura koja se poziva u sistemskom pozivu kojim se traži preslikavanje dela memorije u fajl.

Realizovati funkciju `pgLoad()` koju poziva deo sistema za zamenu stranica kada želi da učita traženu stranicu `pg`, koja pripada datom memorijskom segmentu `mmf`, u već alocirani okvir fizičke memorije `frame`.

Sve pomenute funkcije vraćaju celobrojni status: negativnu vrednost u slučaju greške, 0 u slučaju ispravnog završetka. Pretpostaviti da je sadržaj fajla dovoljno veliki da se u njega preslika ceo dati segment (to obezbeÿuje inicijalizacija preslikavanja).

```
typedef unsigned int PageNo;
typedef ... FHandle;
const unsigned int PAGESIZE = ...; //Page size in bytes
struct MMFSegment {
      PageNo pgLow, pgHigh;
      FHandle fh;
      ...
}
int pgLoad(MMFSegment* mmf, PageNo pg, void* frame);
```

Rešenje:

```
int pgLoad(MMFSegment* mmf, PageNo pg, void* frame) {
if (mmf==0 || pg<mmf->pgLow || pg>pgHigh) return -1; // Exception!
  unsigned int offset = (pg - mmf->pgLow)*PAGEGSIZE;
  return fread(mmf->fh,frame,offset,PAGESIZE);
}
```

1. (Decembar 2006) Sistemski pozivi

U nekom operativnom sistemu za neki računar sa dvoadresnim RISC procesorom i LOAD/STORE arhitekturom sistemski pozivi realizuju se softverskim prekidom koji se izaziva instrukcijom `TRAP`. Adresni deo ove instrukcije nosi redni broj sistemskog poziva koji jedinstveno identifikuje taj poziv. Sistemski poziv u registru R1 očekuje pokazivač na strukturu podataka koja nosi argumente sistemskog poziva, specifične za svaki poziv. Korišćenjem `asm` bloka na jeziku C/C++ napisati bibliotečnu funkciju `create_process()` koja je deo API ovog operativnog sistema i koja obavlja sistemski poziv broj 25h za kreiranje procesa nad zadatim programom. U `asm` bloku treba da budu samo najneophodnije naredbe, ostatak ove funkcije treba da bude realizovan na jeziku C/C++.

Funkcija `create_process()` deklarisana je na sledeći način (rezultat funkcije prenosi se u registru R0):

```
typedef unsigned int PID;
PID create_process (     // Returns: Process ID of the created process
  char* program_file,    // Null-terminated string with program file name
  unsigned int priority, // Default (initial) priority
  char** args            // Program arguments (array of strings)
);
```

Navedeni sistemski poziv vraća ID kreiranog procesa u registru R0, a očekuje argumente u sledećoj strukturi na koju treba da ukazuje R1:

```
struct create_process_struct {
  char* program_file;    // Null-terminated string with program file name
  unsigned int priority; // Default (initial) priority
  char** args;           // Program arguments (array of strings)
};
```

Rešenje:

```
PID create_process (char* p_file, unsigned int pri, char** args) {
  create_process_struct p;
  p.program_file = p_file;
  p.priority = pri;
  p.args = args;
  asm {
    mov r1,sp;
    load r2,#p; // #p is the displacement of p below the top of the stack
    add r1,r2;
    trap 25h;
  }
}
```

2. (Januar 2012) Sistemski pozivi

U nekom operativnom sistemu sistemski poziv se vrši softverskim prekidom. Svakoj grupi srodnih sistemskih usluga odgovara jedan sotfverski prekid. Unutar date grupe, sistemsku uslugu odreÿuje vrednost u registru `R1`. Svaki sistemski poziv u registru `R2` očekuje adresu strukture podataka u kojoj su parametri sistemskog poziva, zavisni od konkretne usluge. Status sistemskog poziva vraća se kroz registar `R0` (0-uspešno, <0 - kod greške).
Grupa usluga koje se pozivaju softverskim prekidom 31h vezane su za upravljanje memorijom. Sistemski poziv broj 21h u toj grupi je zahtev za alokacijom dela virtuelnog prostora pozivajućeg procesa. Svoje parametre ova usluga očekuje u sledećoj strukturi:

```
struct vm_area_desc {
int page; // VM area starting page
  int size; // VM area size in
pages };
```

Proceror je 32-bitni, dvoadresni, RISC, sa *load-store* arhitekturom. Ima registarski fajl sa 32 registra. Svi registri i adrese su 32-bitni. U asemblerskom kodu untar C koda datog kompajlera, može se upotrebljavati identifikator statičke promenljive koji se prevodi u memorijsko direktno adresiranje te promenljive. Povratne vrednosti funkcija se prenose kroz registar `R0`, ukoliko je veličina odgovarajuća.
Implementirati bibliotečnu C funkciju koja poziva opisanu uslugu:

```
int vm_alloc(int starting_page, int size_in_pages);
```

Rešenje:

```
int vm_alloc (int pg, int sz) {
  static vm_area_desc vm;
  static vm_area_desc* ptr=&vm;
  vm.page=pg;
  vm.size=sz;
  asm {
    load r1,#0x21
    load r2,ptr
int      0x31
}
}
```

3. (Septembar 2012) Sistemski pozivi

U nekom operativnom sistemu sistemski poziv se vrši softverskim prekidom. Svakoj grupi srodnih sistemskih usluga odgovara jedan sotfverski prekid. Unutar date grupe, sistemsku uslugu odreÿuje vrednost u registru `R1`. Svaki sistemski poziv u registru `R2` očekuje adresu strukture podataka u kojoj su parametri sistemskog poziva, zavisni od konkretne usluge. Status sistemskog poziva vraća se kroz registar `R0` (0-uspešno, <0 - kod greške).
Grupa usluga koje se pozivaju softverskim prekidom 31h vezane su za upravljanje memorijom. Sistemski poziv broj 21h u toj grupi je zahtev za alokacijom dela virtuelnog prostora pozivajućeg procesa. Svoje parametre ova usluga očekuje u sledećoj strukturi:

```
struct vm_area_desc {
int page; // VM area starting page
  int size; // VM area size in
pages };
```

Procesor je 32-bitni, dvoadresni, RISC, sa *load-store* arhitekturom. Ima registarski fajl sa 32 registra. Svi registri i adrese su 32-bitni. U asemblerskom kodu untar C koda datog kompajlera, može se upotrebljavati identifikator statičke promenljive koji se prevodi u memorijsko direktno adresiranje te promenljive. Povratne vrednosti funkcija se prenose kroz registar `R0`, ukoliko je veličina odgovarajuća.

Data je sledeća implementacija bibliotečne C funkcije koja poziva opisanu uslugu:

```
int vm_alloc (int pg, int sz) {
  static vm_area_desc vm;
  static vm_area_desc*
  ptr=&vm; vm.page=pg;
  vm.size=sz
  ;
  asm {
    load
    r1,#0x21
    load r2,ptr
int       0x31
}
}
```

Ova bibliotečna funkcija statički se linkuje sa pozivajućim programom na uobičajeni način. Prevodilac vrši alokaciju C objekata na uobičajeni način.

a)(5) Da li je ova implementacija bezbedna za korišćenje u konkurentnim procesima? Precizno obrazložiti.

Odgovor:

Jeste. Statički podaci (u ovom slučaju `vm` i `ptr`) se alociraju u prevedenom binarnom fajlu, a on se opet učitava u sopstveni virtuelni adresni prostor svakog procesa. Zbog toga će
svaki proces imati svoje instance ovih podataka koji se koriste u bibliotečnoj funkciji, pa time prenose i u sistemski poziv. Naravno, sistemski pozivi (ovde realizovani prekidima) su inherentno meÿusobno isključivi. Tako procesi neće imati konflikte na ovim strukturama i svaki će vršiti sistemski poziv sa svojim parametrima, bez obzira na konkurentnost.

b)(5) Da li je ova implementacija bezbedna za korišćenje u konkurentnim nitima unutar istog procesa? Precizno obrazložiti.

Odgovor:

Nije. Statički podaci se alociraju kako je gore opisano, pa niti unutar istog procesa dele iste instance ovih podataka. Kako nije obezbeÿeno meÿusobno isključenje koda ove funkcije koja je kritična sekcija, može se dogoditi da jedna nit uÿe u bibliotečnu funkciju i započne

je postavivši vrednosti za `vm` i `ptr`, potom procesor preotme druga nit koja takoўe uўe u ovu funkciju i postavi svoje vrednosti, prepisavši one prethodne. Na taj način nastaje konflikt i prva nit će izvršiti sistemski poziv sa pogrešnim parametrima. Da bi se ovaj problem rešio, potrebno je obezbediti meўusobno isključenje u ovoj funkciji.

4. (5. jun 2008.) Sistemski pozivi

Precizno objasniti potencijalni problem koji postoji ukoliko jezgro operativnog sistema obradu sistemskog poziva vrši korišćenjem steka korisničkog procesa i predložiti rešenje.

Odgovor:

Probem može da nastane zbog prekoračenja opsega memorije koji je dodeljen za stek korisničkog procesa.

Naime, može se desiti da je stek korisničkog procesa trenutno blizu svoje granice. Ukoliko bi se obrada sistemskog poziva nastavila korišćenjem tog prostora, može se, zbog dubine ugnježdavanja poziva potprograma u jezgru, dogoditi da se ta granica prekorači, odnosno da samo jezgro neregularno prepiše sadržaj tuđe emorije, uz nepredvidive posledice na sadržaj memorije i rad celokupnog sistema, ili generisanje izuzetka.

Rešenje je da jezgro svoju obradu radi korišćenjem sospstvenog prostora za stek, a da se odmah na ulasku u sistemski poziv, po čuvanju konteksta izvršavanja pozivajućeg procesa (uključujući i vrednost pokazivača steka), pokazivač steka preusmeri na prostor za stek jezgra koji je inicijalno „prazan“ (pokazivač je na „dnu“). Naravno, pri povratku u kontekst korisničkog procesa, restaurira se kompletan kontekst izvršavanja, pa i vrednost njegovog pokazivača steka. Prostor alociran za stek jezgra treba da bude dovoljno veliki da obezbedi ugnježdavanje potrebne dubine.

1. (Ispit Januar 2011) Operativni sistem Linux

Napisati bash shell script koji od spiska studenata formira listu e-mail adresa onih studenata koji imaju prosek zadat drugim argumentom skripta. Spisak studenata se nalazi u fajlu čiji je naziv zadat prvim argumentom skripta. Svaka linija ulaznog fajla sadrži zapis o jednom studentu u sledećem formatu: <ime><razmak><prezime><razmak><broj indeksa><razmak><prosek>. Indeks je zapisan u formatu <godina>/<broj>, gde su i <godina> i <broj> zadati sa po 4 cifre. Prosek je zaokružen na ceo broj. E-mail adrese se formiraju po sledećem formatu: <prvo slovo prezimena><prvo slovo imena><zadnje dve cifre godine><4 cifre broja indeksa>@student.etf.rs. Navođenje manje ili više argumenata, kao i navođenje fajla koji iz nekog razloga nije moguće pročitati, smatra se greškom. U slučaju greške prekinuti izvršavanje skripta.

Rešenje:

```
#!/bin/bash
if [ $# -eq 2 ]
then
  if [ -f $1 -a -r $1 ]
  then
    grep "$2$" $1 | sed "s_\(.\).* \(.\).* ..\(..\)/\(....\)
.*_\2\1\3\4@student.etf.rs_"
  else
    echo "Ulazni fajl nije moguce otvoriti"
  fi
else
  echo "Neodgovarajuci broj parametara"
fi
```

2. (Ispit Januar 2011) Operativni sistem Linux

Napisati program za operativni sistem Linux koji pri prvom pokretanju učita jedan broj sa standardnog ulaza, a zatim pri sledećem pokretanju ispiše prethodno učitani broj (broj koji je učitan prilikom prethodnog pokretanja). Učitani broj između uzastopnih pokretanja čuvati u memoriji (nije dozvoljeno koristiti bilo kakve dodatne objekte za čuvanje vrednosti). Nakon drugog pokretanja, program treba da vrati sistem u početno stanje, tako da se pri sledećem pokretanju ponaša kao da je pokrenut prvi put. Poznato je da sistem koristi čitavu radnu memoriju računara, kao i da se programu prilikom pokretanja prosleđuje jedinstven ključ (ceo broj) koji u sistemu ne koristi niko drugi.

Rešenje:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main (int argc, char *argv[])
{
  int segment_id;
  int* shared_memory;
  const int shared_segment_size = 4096;
  int key = atoi(argv[1]);
  segment_id = shmget (key, shared_segment_size, IPC_CREAT | IPC_EXCL |
S_IRUSR | S_IWUSR);
  if (segment_id == -1){ //drugo pokretanje
    segment_id = shmget (key, shared_segment_size, S_IRUSR | S_IWUSR);
    if (segment_id == -1) {
      printf("Greska - neuspesno dohvatanje id deljenog segmenta
memorije.\n");
      exit(1);
    }
    shared_memory = (int*) shmat (segment_id, 0, 0);
    printf("%d\n",*shared_memory);
    shmdt (shared_memory);
    shmctl (segment_id, IPC_RMID, 0);
  } else{ //prvo pokretanje
    shared_memory = (int*) shmat (segment_id, NULL, 0);
    scanf("%d", shared_memory);
    shmdt (shared_memory);
  }
  return 0;
}
```

3. (Januar 2012) Operativni sistem Linux

Napisati *shell script* koji generiše e-mail adrese svih studenata koji su predali projektni zadatak. Direktorijum u kome se nalaze projektni zadaci zadat je kao prvi argument pri pozivu, a datoteka u koju se upisuju e-mail adrese zadata je kao drugi argument. Ime datoteke u kojoj se nalazi projektni zadatak svakog studenta zapisana je u formatu:

ime_prezime_brIndeksa.zip - gde znak '_' predstavlja separator izmeÿu polja.

Potrebno je generisati e-mail adresu u sledećem formatu:

<prvo slovo prezimena><prvo slovo imena><broj indeksa>d@student.etf.bg.ac.rs

Pretpostaviti da su imena fajlova u zadatom direktorijumu korektno formatirana. Ukoliko nije
prosleÿen odgovarajući broj parametara, ispisati poruku o grešci. Takoÿe, ako prvi parametar nije direktorijum ili drugi parametar ne predstavlja datoteku u koju se može izvršiti upis, ispisati odgovarajuću poruku o grešci. Na kraju izvršavanja skripta potrebno je ostati u polaznom direktorijumu. Primer imena datoteke u kojoj se nalazi projektni zadatak:

```
pera_zikic_110034.zip
```

Potrebno je generisati e-mail adresu:

```
zp110034d@student.etf.bg.ac.rs
```

Rešenje:

```
#!/bin/bash
if test ! $# -eq 2
then
  echo "Nije prosledjen odgovarajuci broj parametara"
  exit 1
fi
if test ! -f $2 -o ! -w $2
then
  echo "Fajl $2 ne postoji ili nemate dozvolu za upis"
  exit 2
fi
if test ! -d $1
then
  echo "Direktorijum $1 ne postoji"
  exit 3
fi
ls $1 | sed "s/\(.\).*_\(.\).*_\(.*\)\.zip/\2\1\3d@student.etf.bg.ac.rs/"
>$2
```

4. (Januar 2012) Operativni sistem Linux

Na jeziku C/C++, koristeći mehanizam prosleÿivanja poruka operativnog sistema Linux, dati rešenje problema filozofa koji večeraju (*dining philosophers*), pri čemu je data funkcija `philosopher` koja kao argument prima jedinstveni broj (identifikator) filozofa. Svaki filozof pri slanju zahteva identifikuje se ovim brojem. Takoÿe, navedena je struktura poruka koje se razmenjuju, kao i značenje vrednosti svakog polja.

```
#define MESSAGE_Q_KEY 1

struct requestMsg {
 long mtype;  // tip poruke - identifikator filozofa
 char msg[1]; // operacija - vrednost: 1 - request forks, 2 - release
forks
};

void philosopher(int id) {
 int requestMsgQueueId = msgget(MESSAGE_Q_KEY, IPC_CREAT | 0666);
 int responseMsgQueueId = msgget(MESSAGE_Q_KEY + 1, IPC_CREAT |
 0666); size_t len = sizeof(char);

 while (1) {
  //request forks
  struct     requestMsg
  msg; msg.mtype = id;
  msg.msg[0] = (char) 1;
  msgsnd(requestMsgQueueId, &msg, len, 0);
  msgrcv(responseMsgQueueId, &msg, len, id, 0);

  //eat
  sleep(1);

  //release
  forks
  msg.mtype    =
  id;
  msg.msg[0] = (char) 2;
  msgsnd(requestMsgQueueId, &msg, len, 0);

  //think
  sleep(1);
 }
}
```

Napisati implementaciju centralnog procesa koji na početku nad funkcijom `philosopher` kreira potreban broj filozofa predstavljenih procesima, a zatim u vidu konobara (*waiter*) komunicira sa filozofima i obezbeÿuje njihovu sinhronizaciju. Nije potrebno proveravati uspešnost izvršavanja operacije nad sandučićima (*message queue*).

Rešenje:

```
void acquireForksForPhilosopher(int *forks[N], int id, int msgQueueId) {
forks[id] = 0;
 forks[(id + 1) % N] = 0;
 struct requestMsg msg_buf;
 msg_buf.mtype = id + 1;
 msg_buf.msg[0] = 1;
 msgsnd(msgQueueId, &msg_buf, sizeof(char), 0);
}

int main() {
 int requestMsgQueueId = msgget(MESSAGE_Q_KEY, IPC_CREAT | 0666);
 int responseMsgQueueId = msgget(MESSAGE_Q_KEY + 1, IPC_CREAT | 0666);
 size_t len = sizeof(char);
```

```
 //philosophers
 int id;
 for (id = 1; id <= N; id++) { // rezervisana vrednost za mtype=0
 if (fork() == 0) {
   philosopher(id);
  }
 }

 //waiter
 int  *forks[N],  *requests[N];
 for (id = 0; id < N; id++) {
 forks[id] = 1;
  requests[id]  =  0;
 }

 struct  requestMsg  msg_buf;
 while (1) {
  int r = msgrcv(requestMsgQueueId, &msg_buf, len, 0, 0);
  id = (int) msg_buf.mtype - 1;

  if (msg_buf.msg[0] == 1) { //request forks
  if (forks[id] && forks[(id + 1) % N]) {
    acquireForksForPhilosopher(forks,  id,  responseMsgQueueId);
   } else {
    requests[id]  =  1;
   }
  } else { //Release forks
  forks[id] = 1;
   forks[(id + 1) % N] = 1;

   // check neighbors
   int leftNeighbour = id ? id - 1 : N - 1;
   int rightNeighbour = (id + 1) % N;
   if (requests[rightNeighbour] && forks[(rightNeighbour + 1) % N]) {
   requests[rightNeighbour] = 0;
    acquireForksForPhilosopher(forks,  rightNeighbour,
      responseMsgQueueId);
   }
   if (requests[leftNeighbour] && forks[leftNeighbour]) {
   requests[leftNeighbour] = 0;
    acquireForksForPhilosopher(forks,  leftNeighbour,
      responseMsgQueueId);
   }
  }
 }
}
```

5. (Septembar 2012) Operativni sistem Linux

Napisati *shell script* koji treba da za direktorijum zadat kao prvi parametar izbroji datoteke koji se u njemu nalaze, isključujući sve ostalo, kao i sadržaj poddirektorijuma. Ukoliko prosleÿeni parametar nije direktorijum, treba ispisati poruku o grešci. Na kraju izvršavanja skripta potrebno je ostati u polaznom direktorijumu.

Rešenje:

```
#!/bin/bash
if  test  !  -d
$1
then
  echo   "Direktorijum   $1   ne   postoji"
  exit 1
fi
ls -l -a $1 / | grep ^- | wc -l
```

6. (Septembar 2012) Operativni sistem Linux

Na jeziku C/C++, koristeći mehanizam prosleÿivanja poruka kod operativnog sistema Linux,
implementirati serverski proces koji predstavlja agenta na aukciji. Ovaj proces preko unapred
definisanog sandučeta (*message queue*) prima posebnu poruku za otvaranje nadmetanja sa početnom cenom. Zatim, nakon prihvatanja poruke za otvaranje, po istom sandučetu počinje
da prihvata ponude od ostalih učesnika u nadmetanju, pri čemu pamti trenutno najveću ponudu. U svakoj ponudi učesnik se identifikuje svojom jedinstvenom vrednošću
(identifikatorom) na osnovu koga dohvata odgovor agenta putem drugog sandučeta, takoÿe unapred definisanog. Svaka nova ponuda mora biti veća od prethodne, inače se ponuÿaču odmah vraća informacija o odbijanju ponude. U suprotnom se vraća poruka da je ponuda prihvaćena i da se čeka krajnji ishod nadmetanja. U slučaju da u meÿuvremenu pristigne ponuda sa većom vrednošću, vraća se informacija o odbijanju ponude, a ukoliko meÿu prethodnih 5 ponuda ne stigne ni jedna veća, nadmetanje se zatvara, a procesu koji predstavlja
učesnika sa najvišom ponudom šalje se poruka o pobedi. U slučaju pristizanja ponude pre nego što je aukcija otvorena, ponuÿaču treba odmah vratiti poruku o neaktivnoj aukciji. Nije
potrebno proveravati uspešnost izvršavanja operacija nad sandučićima. Takoÿe navesti i strukturu poruka koje se razmenjuju, kao i njihovo značenje.

Rešenje:

```
#include <stdio.h>
#include <stdlib.h>
#include
<sys/msg.h>
#include
<sys/ipc.h>
#include <string.h>

#define   MESSAGE_Q_KEY
1
#define OP_AUCTION 111
```

```
struct requestMsg {
 long  mtype;  //  type  -  id,  (id==OP_AUCTION  ->  open
 auction) char msg[1]; // msq content: price
};

struct responseMsg {
 long mtype; // type - id
 char   msg[128];   //   msq   content:
response
};

size_t  responseMsgLen  =  sizeof  (struct  responseMsg)  -  sizeof  (long
int); size_t requestMsqLen = sizeof(char);

void sendMsgToBidder(long id, char* msg) {
 int  responseMsgQueueId  =  msgget(MESSAGE_Q_KEY  +  1,  IPC_CREAT  |
 0666); struct responseMsg res_msg_buf;
 strcpy(res_msg_buf.msg, msg);
 res_msg_buf.mtype=id;
 msgsnd(responseMsgQueueId, &res_msg_buf, responseMsgLen, 0);
}

int main() {
 char bestOffer=0;
 long bestClientId = 0;
 int auctionOver = 0;
 int badOfferCounter = 0;
 int requestMsgQueueId= msgget(MESSAGE_Q_KEY, IPC_CREAT | 0666);
 struct requestMsg req_msg_buf;

 // opening auction - mtype = OP_AUCTION
 while(1){
  msgrcv(requestMsgQueueId, &req_msg_buf, requestMsqLen, 0, 0);
  if (!(req_msg_buf.mtype == OP_AUCTION))
   sendMsgToBidder(req_msg_buf.mtype,"NotActiveAuction");
  else
   break;
 }
 bestOffer = req_msg_buf.msg[0];

 while (!auctionOver){
  msgrcv(requestMsgQueueId, &req_msg_buf, requestMsqLen, 0, 0);
  int newOffer = req_msg_buf.msg[0];

  if (newOffer > bestOffer) {
      if (bestClientId != 0){
       sendMsgToBidder(bestClientId,"BetterOfferReceived");   }
      bestOffer = newOffer;
      bestClientId = req_msg_buf.mtype;
      sendMsgToBidder(req_msg_buf.mtype,   "OfferAccepted");
      badOfferCounter = 0;
     } else {
   sendMsgToBidder(req_msg_buf.mtype,   "OfferRejected");
   badOfferCounter++; }

     if (badOfferCounter == 5) {
      if (bestClientId != 0) sendMsgToBidder(bestClientId,"YouWon!");
      auctionOver = 1;
     }
 }
 return 0;
}
```

7. (Januar 2013) Operativni sistem Linux

Napisati *shell script* koji dohvata fajlove sa interneta odreÿene ekstenzije u tekući direktorijum i ispisuje njihove nazive na standardnom izlazu. Prvi argument skripte je internet adresa veb stranice u kojoj se nalaze linkovi ka fajlovima koje treba dohvatiti. Drugi argument skripte je ekstenzija fajlova koji se dohvataju. Stranica je tekstualni fajl u HTML formatu. Svaki link ka fajlu predstavlja internet adresu zapisanu na sledeći način:

Naslov.

Primer:

Diplome

U slučaju neodgovarajućeg broja argumenata ili nemogućnosti da se dohvati zadata stranica prijaviti grešku i prekinuti izvršavanje skripte. Nakon završetka izvršavanja skripte potrebno je ostaviti sistem u neizmenjenom stanju.

Program *wget* služi za dohvatanje fajla sa zadate adrese u tekući direktorijum. Njemu se kao parametar prosleÿuje internet adresa fajla (smatrati da adresa ne sadrži razmake). Ovoj naredbi može se zadati opcija *-O* nakon koje sledi ime koje se dodeljuje odredišnom fajlu. Ukoliko se ova opcija ne koristi, ime fajla će biti nepromenjeno, tj. isto kao izvorišno.

Rešenje:

```
#!/bin/bash

if [ $# -lt 2 ];then
    echo "Nedovoljan broj argumenata!"
    exit 1
fi

tmp="tmp.html"
wget "$1" -O $tmp
if [ $? -ne 0 ];then
    echo "Nepostojeci URL"
    exit 2
fi

IFS_old=$IFS
IFS=$'\n'
for i in $(cat $tmp | grep href=\".*\.$2\"\>| sed
's/.*href="\(.*\)">.*/\1/');do
    echo "$i"
    wget "$i"
done
IFS=$IFS_old
rm $tmp
```

8. (Januar 2013) Operativni sistem Linux

Posmatra se sistem od tri procesa koji predstavljaju pušače i jedne klase koja predstavlja agenta. Svaki pušač ciklično zavija cigaretu i puši je. Za zavijanje cigarete potrebna su tri sastojka: duvan, papir i šibica. Jedan pušač ima samo duvan, drugi papir, a treći šibice. Agent
ima neograničene zalihe sva tri sastojka. Agent postavlja na sto dva sastojka izabrana slučajno. Pušač koji poseduje treći potreban sastojak može tada da uzme ova dva, zavije cigaretu i puši. Kada je taj pušač popušio svoju cigaretu, on javlja agentu da može da postavi
nova dva sastojka, a ciklus se potom ponavlja. Primer implementacije navedena tri procesa dat
je u nastavku:

```
int main() {
 key_t   key   =
 ...;
 Agent a(key);

 for(int
 i=0;i<3;i++){
   if(fork()==0){
    while(1){
     switch (i) {
      case 0: a.takePaperAndMatch(); break;
      case    1:    a.takeTobaccoAndMatch();
      break;              case             2:
      a.takeTobaccoAndPaper();        break;
      default: exit(1); break;
     }

     //consum
     e
     sleep(1)
     ;

     a.finishedSmoking();

     //wait
     sleep(1);
    }
   }
 }
 wait(0);
return 0;
 }
```

Koristeći za potrebe meÿuprocesne komunikacije i sinhronizacije isključivo semafore operativnog sistema Linux realizovati na jeziku C++ klasu koji predstavlja agenta. `Agent(key_t key)` prima kao argument vrednost ključa koji jedinstveno odreÿuje skup semafora koji se koristi. Na raspolaganju je funkcija `randNum()` koja nasumično vraća celobrojnu vrednost u opsegu [0,2] različitu od prethodnog poziva. Nije potrebno proveravati uspešnost izvršavanja operacija nad semaforima.

Rešenje:

```
class Agent {
public:
 Agent(key_t key);
 virtual ~Agent();
 void takeTobaccoAndPaper () {atomicOnTwoSems(Paper,Tobacco,-1);}
 void takePaperAndMatch () {atomicOnTwoSems(Paper,Match,-1);}
 void takeTobaccoAndMatch () {atomicOnTwoSems(Match,Tobacco,-1);}
```

```
 void finishedSmoking () {atomicOnTwoSems(randNum(),randNum(),1);}
private:
 int id;
 void atomicOnTwoSems(int first, int second, int op);
 int randNum();
 static const int Paper=0, Match=1, Tobacco=2;
};
```

9. (Septembar 2013) Operativni sistem Linux

Napisati shell script koji treba da za direktorijum zadat kao prvi parametar ispiše imena svih fajlova koji se nalaze u njemu i koji u svom sadržaju imaju niz karaktera zadat kao drugi parametar. Imena fajlova mogu da sadrže razmake. Ukoliko prosleÿeni prvi parametar nije direktorijum ili ukoliko broj parametara nije odgovarajući, treba ispisati poruku o grešci.

Rešenje:

```
#!/bin/bash

if [ $# -ne 2 ]; then
    echo       "Nedovoljan       broj
    parametara"
    exit 1
fi
dir=$1
string=$2
if   [   -d   "$dir"
];then
    IFS_old=$IFS
    IFS=$'\n'
    for     i     in     $(find
    "$dir");do
        if [ -f "$i" ];then
            grep $string "$i" > /dev/null && echo "$i"
        fi
    done
    IFS=$IFS_old
else
    echo      "Prvi      parametar      nije
    direktorijum"
    exit 2
fi
```

10. (Septembar 2013) Operativni sistem Linux

Na jeziku C/C++, koristeći mehanizam deljene memorije kod operativnog sistema Linux, napisati program koji konkurentno množi dve matrice *A* i *B* dimenzija *NxN* i rezultat smešta u
matricu *C*. Svaka od ovih matrica je smeštena u zasebnom, već kreiranom, segmentu deljene
memorije, pri čemu matrice *A* i *B* su inicijalizovane vrednostima. Prvi argument koji se prosleÿuje prilikom poziva programa predstavlja vrednost ključa za dohvatanje odgovarajućih segmenta memorije `key_t key`, gde matricu *A* odreÿuje vrednost `key`, matricu *B* vrednost `key+1`, a *C* vrednost `key+2`. Drugi argument prilikom poziva programa je *N* koje ujedno odreÿuje i veličine segmenata deljene memorije za svaku matricu. Program treba da se izvršava konkurentno tako što će računati svaki element rezultujuće matrice *C* u zasebnom procesu. Nije potrebno proveravati uspešnost izvršavanja operacija nad segmentima deljene
memorije.

Rešenje:

```
#include <stdio.h>
#include   <sys/shm.h>
#include <sys/stat.h>

int main (int argc, const char* argv[])
{
 int shmkey, N, shmsize;
 if ( argc > 2 ) {
     shmkey = atoi( argv[1] );
     N = atoi( argv[2] );
    }
 else return -1;

 shmsize = N*N*sizeof(int);

 int shmid1 = shmget(shmkey,shmsize, IPC_CREAT | S_IRUSR);
 int shmid2 = shmget(shmkey+1,shmsize, IPC_CREAT | S_IRUSR);
 int shmid3 = shmget(shmkey+2,shmsize, IPC_CREAT | S_IRUSR | S_IWUSR);

 for (i = 0; i < N; i++)
  for(j=0;j<N;j++)
   if ( fork() == 0) {

    int * a = (int*)shmat(shmid1,0,0);
    int * b = (int*)shmat(shmid2,0,0);
    int * c = (int*)shmat(shmid3,0,0);

    c[i*N+j]=0;
    for(k=0;k<N;k++)
     c[i*N+j] += a[i*N+k]*b[k*N+j];

    shmdt((void*)a);
    shmdt((void*)b);
    shmdt((void*)c);
    exit(0);
   }

 wait(0);
return   0;
 }
```

11. (ispit 2006) Operativni sistem Linux

a)(3) Koje su metode međuprocesne komunikacije dostupne na Linuxu?

Signali, cevi, liste poruka, semafori i deljena memorija.

b)(7) Kakva je veza između cevi (engl. *pipe*) i sistema fajlova na Linuxu? Opisati funkcionisanje imenovanih i neimenovanih cevi u odnosu na fajl sistem.

Sa korisničke strane, cevi se vide kao virtuelni fajlovi u koji jedan proces upisuje podatke, a drugi proces čita. U slučaju neimenovanih cevi se standardni izlaz jednog procesa i standardni ulaz drugog procesa povezuju sa tim virtuelnim fajlom.

12. (ispit 2006) Operativni sistem Linux

6.1 (3 poena) Kakva je organizacija strukture jezgra koja sadrži procese?

a) Jednostruko-povezana lista

b) Dvostruko-povezana lista

c) Statički niz

d) _____

6.2 (4 poena) Koja je razlik između prostora jezgra („kernel-space“) i prostora korisničkih procesa („user-space“)? Da li je moguća komunikacija između ta dva prostora i kako?

6.3 (3 poena) Kako se zove osnovni element svih UNIX (POSIX) sistema fajlova, koji sadrži informacije o fajlovima i direktorijumima?

a) d-block

b) i-node

c) j-omega

d) f-node

e) _____

----- :(

13. (ispit 2006) Operativni sistem Linux

a)(7) Napisati C++ kod koji će u imenovani cevovod (*pipe*) pod nazivom `Cev` da upiše poruku “Ispit iz OS2”.

Rešenje:

b) (3) Napisati Linux *shell* komandu koja će programu `zad6b` da prosledi podatke iz cevovoda (*pipe*) `Cev`. Program `zad6b` prima podatke preko standardnog ulaza.

a)(7) Cevovod (*Pipe*) je vrsta fajla tako da se u njega podaci upisuju i citaju isto kao iz fajla.

```
#include <stdio.h>
int main() {
  FILE* out = fopen("Cev", "w");
  if (out == NULL) return -1;
  fprintf(out, "Ispit iz OS2\n");
  fclose(out);
  return 0;
}
```

b)(3) `./zad6b < Cev`

14. (ispit 2006) Operativni sistem Linux

Implementirati funkciju `void readFromPipe(char* pipeName, int* n, int a[])`, koja čita brojeve iz imenovanog cevovoda `pipeName` (*engl.* Named pipe), i smešta ih u niz `a`. Broj učitanih elemenata snimiti u `*n`. Proveriti postojanje cevovoda `pipeName`. Pretpostaviti da je niz `a` alociran i da ima dovoljno mesta za sve elemente iz cevi.

```
void readFromPipe(char* pipeName, int* n, int a[]){
  FILE* in = fopen(pipeName, "r");
  if (in == 0) {
    printf("Nema cevi %s!", pipeName);
    exit(0);
  }
  *n = 0;
  while (fscanf(in, "%d", a + *n) > 0) (*n)++;
  fclose(in);
}
```

15. (ispit 2006) Operativni sistem Linux

a)(5) Dati program `fa` čita brojeve sa standardnog ulaza (dokle god ih ima), obrađuje svaki od njih pomoću funkcije `int FA(int)`, i rezultate ispisuje na standardni izlaz. Napisati Linux shell komandu koja će u pozadini pročitati sve brojeve iz cevovoda (engl. *named pipe*) `cev0`, na svakog od njih primeniti funkciju `FA(FA(FA(int)))` i rezultate ispisati u cevovod `cev1`.

```
./fa < cev0 | ./fa | ./fa > cev1 &
```

b)(5) Napisati Linux shell komandu koja će da upiše brojeve 3, 4, 5 i 6 u cevovod `cev0`.

```
echo 3 4 5 6 > cev0
```

16. (ispit 2006) Operativni sistem Linux

a)(4) Kako se zove proces koji jezgro Linuxa automatski pokreće prilikom uključivanja računara? Na kojoj lokaciji u sistemu fajlova se nalazi izvršni fajl tog procesa. U kratkim crtama reći šta taj proces radi na operativnim sistemima GNU/Linux.

c)(6) Tekst fajl `config` u svakom svom redu sadrži ime tačno jednog programa. Napisati program na jeziku C/C++ za Linux, koji će čitati taj fajl i za program iz svakog reda kreirati tačno jedan proces. Zatim startni program treba da se odmah završi, ne čekajući da njegovi potomci završe. Bilo kakva greška treba samo da prekine ovaj program. Pretpostaviti sa se svi potrebni fajlovi nalaze u istom direktorijumu i da imena programa u fajlovima neće imati više od 1000 znakova.

a)(4) Proces koje jezgro Linuxa pokreće automatski prilikom uključivanja sistema se zove „init" proces. Uobičajeno se izvršni fajl tog procesa nalazi na lokaciji „/sbin/init", ali je moguće promeniti lokaciju tog fajla (npr. argumentom jezgra „init=/putanja/do/fajla").

Na operativnim sistemima GNU/Linux funkcija tog procesa je da pokrene sve sistemske servise koji je potrebno pokrenuti. To znači da taj proces pokreće logovanje sistemskih poruka, podešava mrežu, firewall (iptables), pokreće servis za štampanje, pokreće web server, baze podataka, itd. Na kraju, kod grafički orjentisanih distribucija operativnog sistema GNU/Linux, taj proces pokreće i grafičko okruženje X Window System.

Ovaj proces uglavnom ima gore opisanu funkcionalnost. U specifičnim situacijama, kao sto su npr. upravljački računarski sistemi bazirani na Linux jezgru (Embedded sistemi), taj proces može da predstavlja ceo upravljački program.

c)(6)

```
#include <stdio.h>
#include <unistd.h>

int main(){
  FILE *in = fopen("config", "r");

  if (in == NULL) { return -1; }

  char program[1001];
  while (fscanf(in, "%s", program) > 0){
     int PID = fork();

     if (PID == -1) { return -1; }
     else if (PID == 0) execl(program, NULL);
  }

  fclose(in);
  return 0;
}
```

17. (ispit 2006) Operativni sistem Linux

Napisati „Hello World“ modul kernela, koji ispisuje poruku „Hello World, loaded ...“ prilikom učitavanja i poruku „Hello World, unloaded ...“ prilikom isključivanja modula. Definisati svoje ime, prezime i broj indeksa kao autora modula.

------ :(

18. (ispit 2006) Operativni sistem Linux

Napisati modul za Linux jezgro koji kao ulazni parametar prima niz vrednosti tipa `int` i prilikom učitavanja ispisuje zbir elemenata tog niza. Ulazni niz neće imati više od 50 elemenata i rezultujući zbir ne prelazi veličinu tipa `int`.

```
/* ispitniModul.c */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE ("GPL");
MODULE_AUTHOR ("Katedra RTI");
MODULE_DESCRIPTION ("Modul za Junski rok 2006.");



static int niz[50];
static int brNiz;

module_param_array (niz, int, &brNiz, 0000);
MODULE_PARM_DESC (niz, "Niz brojeva");

static int __init ir3os2_init (void) {
    int i;
    int zbir;

    printk (KERN_INFO "IR3OS2 Modul sa ispita\n");
    printk (KERN_INFO "Ucitano je %d elemenata:\n", brNiz);

    zbir = 0;

    for (i=0; i<brNiz; i++) {
        printk (KERN_INFO "\t%d\n", niz[i]);
        zbir += niz[i];
    }

    printk (KERN_INFO "\nZbir elemenata niza je %d.\n", zbir);

    return 0;
}

module_init (ir3os2_init);

static void __exit ir3os2_exit (void) {
    printk (KERN_INFO "Dovidjenja!\n");
}

module_exit (ir3os2_exit);
```

19. (ispit 2006) Operativni sistem Linux

Implementirati funkciju `void mergeSort(int n, int m, int niz[])` koja sortira niz od *n* elemenata u rastućem redosledu na sledeći način. Funkcija uzima po *m* elemenata niza i sortira ih u konkurentnim procesima pomoću algoritma datog funkcijom `int sort(int n, int a[])`, a zatim tako sortirane podnizove spaja (*engl.* merge sort). Na raspolaganju su sledeće funkcije za rad sa imenovanim cevovodima (*engl.* Named Pipes):

```
// Upisuje elemente niza a, duzine n u cevovod pipeName
void sendToPipe(char* pipeName, int n, int a[]);
// Čita niz elemenata a iz cevovoda pipeName i njihov broj broj smešta u *n
void readFromPipe(char* pipeName, int* n, int a[]);
// Vraća ime sledećeg cevovoda koji može da se koristi
char* getNextPipeName();
// Obezbeđuje da funkcija getNextPipeName vraća imena cevovoda od početka
void resetPipeNames();
```

```
int mergeSort(int n, int m, int niz[]){
  // Creating child processes
  int numOfChildren = 0;
  for (int i = 0; i < n; i += m){
      numOfChildren++;
      char* childPipeName = getNextPipeName();
      int len = (i + m < n) ? m : n - i;
      int pid = fork();
      if (pid == 0){
        sort(len, niz + i);
        sendToPipe(childPipeName, len, niz + i);
        exit(0);
      }
  }
  // Collecting sorted subarrays
  int cntr = 0;
  int** b = new pint [numOfChildren];
  int* cntrs = new int [numOfChildren];
  int* sizes = new int [numOfChildren];
  resetPipeNames();
  for(int c = 0; c < numOfChildren; ++c) {
    b[c] = new int[m];
    cntrs[c] = 0;
    readFromPipe(getNextPipeName(), &sizes[c], b[c]);
  }

  // Merging
  while (cntr < n) {
    int arrayInd = -1, min = -1;
    for (int i = 0; i < numOfChildren; ++i)
      if (cntrs[i] < sizes[i] && (arrayInd == -1 || min > b[i][cntrs[i]])){
        min = b[i][cntrs[i]];
        arrayInd = i;
      }
      niz[cntr++] = min;
      cntrs[arrayInd]++;
  }
  // Release temporary space
  for (int i = 0; i < numOfChildren; i++)
    delete b[i];
  delete b;
  delete cntrs;
  delete sizes;
  return 0;
}
```

20. (ispit 2006) Operativni sistem Linux

Implementirati funkciju `void parallelProcess(int n, int a[])` koja prihvata niz od *n* celih brojeva i u konkurentnim procesima obrađuje svaki od elemenata tog niza datom funkcijom `void process(int* x)`. Na raspolaganju su sledeće funkcije za rad sa neimenovanim cevovodima (engl. *pipe*):

```
int pipe(int fd[2]); // Kreira cev (engl. Pipe) i vraća deskriptore: fd[0] // za čitanje i fd[1] za pisanje u kreiranu cev. Funkcija prima kao // argument niz od dva cela broja

int read(int fd, void *Buff, int NumBytes); // čita podatke iz fajla/pipea // sa deskriptorom fd i smešta u bafer Buff velicine NumBytes

int write(int fd, void *Buff, int NumBytes); // upisuje bafer podataka Buff // velicine NumBytes u file/pipe sa deskriptorom fd

int close(int fd); // zatvara descriptor fd
void process(int* x);
int pipe(int fd[2]);
int read(int fd, void *Buff, int NumBytes);
int write(int fd, void *Buff, int NumBytes);
int close(int fd);
```

```
int fileDesc[2];

struct Message{
  int index, value;
};

void parallelProcess(int n, int a[]){
  pipe(fileDesc);

  for (int i = 0; i < n; ++i){
    Message m; m.index = i; m.value=a[i];

    if (fork() == 0){
       process(&m.value);
       write(fileDesc[1], &m, sizeof(Message));
       exit(0);
    }
  }

  for (int i = 0; i < n; ++i) {
    Message msg;
    read(fileDesc[0], &msg, sizeof(Message));
    a[msg.index] = msg.value;
  }

  close(fileDesc[0]);
  close(fileDesc[1]);

}
```

21. (ispit 2007) Operativni sistem Linux

a)(5) Da li Linux prilikom kreiranja procesa sistemskim pozivom `clone()` sa opcijom `CLONE_VM==0` (kloniraj VM, ne deli ga) kopira stranice privatnih regiona zadatka-roditelja da bi formirao iste takve za zadatak-dete? Obrazložiti.

Odgovor:

Ne, jer podržava *copy-on-write* semantiku (objašnjeno na predavanjima).

b)(5) Precizno objasniti zašto se kaže da je algoritam raspoređivanja procesa u Linuxu počev od verzije 2.5 kompleksnosti $O(1)$?

Odgovor:

Zato što vreme izbora procesa za izvršavanje ne zavisi od broja trenutno spremnih procesa. Uvek se uzima prvi proces iz prvog nepraznog reda *active task list* sa najvišim prioritetom. Ovo eventualno uključuje iteriranje kroz redove redom po prioritetima, ali broj ovih iteracija ne zavisi od broja spremnih procesa, već od toga kog prioriteta je najprioritetniji spremni proces.

22. (ispit 2007) Operativni sistem Linux

a)(5) Dat je modul "os2.ko" koji nema parametara. Napisati sekvencu Linux shell komandi koje treba da učitaju pomenuti modul, potom da izlistaju sve trenutno učitane module i na kraju da isključe pomenuti modul.

```
insmod os2.ko
lsmod
rmmod os2
```

b)(5) Navesti dva moguća stanja procesa u Linux operativnom sistemu u kojima se može naći ako je blokiran. Objasniti razliku izmedju njih.

1. TASK_INTERRUPTIBLE,
2. TASK_UNINTERUPTIBLE

Prvi mogu biti probuđeni usled pristizanja nekog signala iako razlog zbog kojeg su blokirani još nije otklonjen.

23. (ispit 2007) Operativni sistem Linux

a)(5) Da li je iz bilo kojeg modula kernela moguće pristupiti korisničkim podacima? Ako nije, zašto, ako jeste, kako? Objasniti kratko i precizno.

Da, korišćenjem funkcija get_user(to, from) i put_user(from, to) čije se deklaracije nalaze u zaglavlju "asm/uaccess.h". get_user(to, from) uzima podatak iz bafera "from" koji je u korisničkom memorijskom prostoru i smešta u bafer "to" koji je lokalni podatak jezgra. Operacija put_user(from, to) radi analogno prethodnoj.

b)(5) Objasniti tehniku kojom se u operativnom sistemu Linux procesima dodeljuje procesorsko vreme u skladu sa njihovom trenutnom interaktivnošću (obavljanjem I/O operacija).

Dinamičko menjanje prioriteta procesa, a samim tim i menjanje intervala procesorskog vremena koje će se procesu dodeliti. Prati se vreme provedenu na čekanju na I/O operacije i u zavisnosti od toga, procesu se dodeljuje prioritet baziran na nice vrednosti +/-5. Veća interaktivnost, veći prioritet (bliže -5), manja interaktivnost, manji prioritet (bliže +5)

24. (ispit 2007) Operativni sistem Linux

a) (5) Precizno objasniti način na koji operativni sistem Linux pruža niz korisnih informacija o procesima.

Operativni sistem Linux u fajl sistem montira virtuelni fajl sistem (Process File System). Montira se u /proc i za svaki fajl kreiran je po jedan direktorijum. Direktorijumi se imenuju po ASCII reprezentaciji process ID. Svaki od tih direktorijum sadrži niz tekstualnih fajlova koji sadrže niz informacija o konkretnom procesu.

b) (5) Da li u operativnom sistemu Linux može doći do izgladnjivanja procesa sa malim prioritetom usled toga što u sistemu stalno postoje spremni procesi sa većim prioritetom? Dati precizno obrazloženje.

Ne. Svaki proces u skladu sa svojim prioritetom ima dodeljen vremenski kvant. Kada neki proces potroši svoj kvant vremena, bez obzira na to da li je spreman i bez obzira na prioritet koji ima, mora sačekati da svi ostali procesi potroše svoj kvant vremena pre nego što opet dobije procesor.

25. (ispit 2007) Operativni sistem Linux

a)(5) Objasniti razliku između procesa i niti u operativnom sistemu Linux.

Linux praktično ne razlikuje direktno procese i niti, već ih generalizuje u zadatak. Razlika nastaje pri kreiranju procesa, odnosno niti sistemskim pozivom *clone()* tako što se specificira koji resursi će se deliti a koji ne. U slučaju kada su svi resursi deljeni, tada imamo dva zadatka koja se izvršavaju u istom adresnom prostoru, ali svaki sa svojim kontekstom – niti. U suprotnom slučaju, kada se nijedan resurs ne deli, dobijaju se zadaci koji se izvršavaju u odvojenim adresnim prostorima – procesi.

b)(5) Precizno objasniti šta su signali, čemu služe i koje su njihove mogućnosti u operativnom sistemu Linux.

Signali su jedan od načina međuprocesne komunikacije. Služe da jedan process signalizira nekom drugom procesu da se desio neki događaj. To je način da procesi budu obavešteni asinhrono o nekom događaju, bez obzira u kojoj fazi izvršavanja se process trenutno nalazi. Dovoljno je da definišu svoju funkciju kojom treba da reaguju na signal. Funkcioniču kao softverski generisani prekidi.

26. (ispit 2007) Operativni sistem Linux

Implementirati funkciju `void parallelProcess(int n, int a[])` koja prihvata niz od *n* celih brojeva i u konkurentnim procesima obrađuje svaki od elemenata tog niza datom funkcijom `int process(int x)`. Rezultate ne treba vraćati u niz, već ih treba ispisati na standardni izlaz. Ispis treba da bude u okviru konkurentnih procesa i to tako da se zadrži početni redosled koji su ulazni parametri imali u ulaznom nizu `a`. Smatrati da `n` nije veće od 100. Na raspolaganju su sledeće funkcije za rad sa neimenovanim cevovodima (engl. *pipe*):

```
      int pipe(int fd[2]); // Kreira cev (engl. Pipe) i vraća deskriptore:
fd[0] // za čitanje i fd[1] za pisanje u kreiranu cev. Funkcija prima kao
// argument niz od dva cela broja

      int read(int fd, void *Buff, int NumBytes); // čita podatke iz
fajla/pipea // sa deskriptorom fd i smešta u bafer Buff velicine NumBytes

      int write(int fd, void *Buff, int NumBytes); // upisuje bafer
podataka Buff // velicine NumBytes u file/pipe sa deskriptorom fd

int close(int fd); // zatvara descriptor fd

Odgovor:

      void process(int* x);
      int pipe(int fd[2]);
      int read(int fd, void *Buff, int NumBytes);
      int write(int fd, void *Buff, int NumBytes);
      int close(int fd);

      int fileDesc[101][2];

      void parallelProcess(int n, int a[]){
        int start, kraj;

        pipe(fileDesc[0]);

        for (int i = 0; i < n; ++i){
          int result;
          int index;
          pipe(fileDesc[i+1]);
          if (fork() == 0){
             result = process(a[i]);
            read(fileDesc[i][0], &index, sizeof(int));
      printf("%d ",result);
            index++;
             write(fileDesc[i+1][1], &index, sizeof(int));
             exit(0);
          }
        }

        write(fileDesc[0][1], &start, sizeof(int));
        read(fileDesc[n][0], &kraj, sizeof(int));

        for (int i = 0; i <= n; ++i) {
          close(fileDesc[i][0]);
          close(fileDesc[i][1]);
        }
        if (kraj != n) exit(1);

}
```

27. (ispit 2007) Operativni sistem Linux

Napisati modul za Linux jezgro. Modul treba da prihvati jedan celobrojni parametar sa nazivom "param1". Pri učitavanju treba da ispiše vrednost tog parametra, dok pri isključivanju treba da ispiše poruku "os2 isključen". Za autora modula navesti svoje ime i prezime.

```
 #include<linux/module.h>
 #include<linux/moduleparam.h>
 #include<linux/kernel.h>
 #include<linux/init.h>
 #include<linux/stat.h>

 MODULE_AUTHOR("ime prezime");
 ststic int param1 = 0;

 module_param(param1,int,0000);

 static int __init os2start(void){
printk(KERN_INFO "%d\n", param1);
return 0;
 }

 module_init(os2start);

 static void __exit os2stop(void){
      printk(KERN_INFO "os2 iskljucen");
 }

 module_exit(os2stop);
```

28. (ispit 2007) Operativni sistem Linux

Implementirati funkciju `void parallelProcess(int n, int a[][])`. Funkcija prihvata jednu matricu celih brojeva dimenzija *n x n* u kojoj su inicijalizovani prvi red i prva kolona. Funkcija treba da popuni matricu tako da za svaki element izvan prve kolone i prvog reda važi a[i][j] = f(a[i-1][j] , a[i][j+1]), gde je f data funkcija koja prihvata dva cela broja i vraća jedan ceo broj. Računanje treba da se obavlja u okviru konkurentnih procesa, tako da za svaki red postoji po jedan proces. Smatrati da `n` nije veće od 100. Za komunikaciju i sinhronizaciju između procesa na raspolaganju su sledeće funkcije za rad sa neimenovanim cevovodima (engl. *pipe*):

```
      int pipe(int fd[2]); // Kreira cev (engl. Pipe) i vraća deskriptore:
fd[0] // za čitanje i fd[1] za pisanje u kreiranu cev. Funkcija prima kao
// argument niz od dva cela broja

      int read(int fd, void *Buff, int NumBytes); // čita podatke iz
fajla/pipea // sa deskriptorom fd i smešta u bafer Buff velicine NumBytes

      int write(int fd, void *Buff, int NumBytes); // upisuje bafer
podataka Buff // velicine NumBytes u file/pipe sa deskriptorom fd

int close(int fd); // zatvara descriptor fd
```

Odgovor:

```
      void process(int* x);
      int pipe(int fd[2]);
      int read(int fd, void *Buff, int NumBytes);
      int write(int fd, void *Buff, int NumBytes);
      int close(int fd);

      typedef struct {
        int i,j;
        int val
      } Item;


      int fileDesc[100][2];
      int return_pipe[2];

      void parallelProcess(int n, int a[][]){
        int upper;
        Item item;

        pipe( fileDesc[0] );
        pipe( return_pipe );

        for (int i = 1; i < n; ++i){
          pipe( fileDesc[i] );
          if (fork() == 0){
           item.i = i;
           item.val = a[i][0];
           for (int j = 1; i < n; ++i){
               if ( i > 1 )
                 read(fileDesc[i-1][0], &upper, sizeof(int));
                 else upper = a[0][j];
             item.j = j;
        item.val = f ( upper , item.val );
        write(return_pipe[1], &item, sizeof(Item));
               if ( i < n-1 )
                 write(fileDesc[i][1], &(item.val), sizeof(int));
```

```
    }
     exit(0);
   }
}
Item item;
for (int i = 0; i < (n-1)*(n-1); ++i){
  read(return_pipe[0], &item, sizeof(Item));
  a[item.i][item.j] = item.val;
}
close( return_pipe[0] );
close( return_pipe[1] );
for (int i = 0; i < n; ++i) {
  close(fileDesc[i][0]);
  close(fileDesc[i][1]);
}
```

29. (ispit 2007) Operativni sistem Linux

Potrebno je za operativni sistem Linux obezbediti rešenje problema sinhronizacije čitalaca i pisaca. Za komunikaciju isključivo koristiti prosleđivanje poruka čiji je interfejs:

int msgget(key_t key, int msgflg);
//Gde je key ključ po kojem se traži red, msgflg, flegovi koji određuju način kreiranja:
666 – vrati identifikator ako red postoji
666 | IPC_CREAT – ako ne postoji kreiraj, ako postoji vrati identifikator postojećeg
666 | IPC_CREAT | IPC_EXCL – kreiraj jedino ako ne postoji.
Funkcija vraća identifikator reda ako je operacija uspešna i -1 ako je neuspešna.

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
//Šalje poruku na koju pokazuje msg_ptr, čija je veličina msg_sz u red sa identifikatorom msqid. msgflg treba da je 0.

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
//Prihvata poruku iz reda sa identifikatorom msqid u bafer na koji ukazuje msg_ptr maksimalne veličine msg_sz. msgtype i msgflg treba da su jednaki nuli. Operacija je blokirajuća.

int msgctl(int msqid, IPC_RMID, 0);
//Oslobađa red sa identifikatorom msqid

Poslednje tri funkcije vraćaju -1 u slučaju greške i neki nenegativan broj u slučaju uspeha.

Rešenje ne sme da ima izgladnjivanje ni čitalaca ni pisaca. Smatrati da je maksimalan broj čitalaca, kao i maksimalan broj pisaca ograničen na 100.

Rešenje:

```
#include<stdio.h>
#include<sys/msg.h>
#define P 1024

typedef struct {
  long int msg_type;
  int odgovor;
}my_msg;

int main(){
  my_msg m;
  int nizR[100], nizW[100]; //citaoci i pisci koji cekaju
  int nR=0, nW=0, nRc=0;          //broj  citalaca,  pisaca,citalaca
koji cekaju
  int nWc=0, pocW=0;                    //broj  pisaca  koji  cekaju,
pozicija pocetka reda u nizu pisaca
  int prijem, status;
  prijem = msgget((key_t)P, 0666 | IPC_CREAT | IPC_EXCL);
  if (prijem < 0) {
    printf("Greska-neuspesno otvaranje inboxa");
    return 1;
  }
  status  =  msgrcv(prijem,  &m,  sizeof(my_msg)-sizeof(long  int),  0,
0);
  while(m.msg_type>1){
    switch(m.msg_type){
     case 2:           //startRead
      if ((nW==0) && (nWc == 0)){
        nR++;
        msgsnd(m.odgovor, &m, sizeof(my_msg)-sizeof(long int), 0);
      }else nizR[nRc++] = m.odgovor;
     break;
```

```
            case 3:           //stopRead
             if ((--nR == 0) && (nWc>0)){//pusti pisca
               nWc--;
               msgsnd(nizW[pocW], &m, sizeof(my_msg)-sizeof(long int), 0);
               pocW = (pocW + 1) % 100;
               nW++;
             }
             break;
            case 4:           //startWrite
             if ((nR==0) && (nW==0)){
               nW++;
               msgsnd(m.odgovor, &m, sizeof(my_msg)-sizeof(long int), 0);
             }else{
               nizW[(pocW+nWc)%100] = m.odgovor;
               nWc++;
             }
             break;
            case 5:           //stopWrite
             nW--;
             while (nRc>0){
               nRc--;
               msgsnd(nizR[nR], &m, sizeof(my_msg)-sizeof(long int), 0);
               nR++;
             }
             if ((nR==0) && (nWc>0)){
               nWc--;
               msgsnd(nizW[pocW], &m, sizeof(my_msg)-sizeof(long int), 0);
               pocW = (pocW + 1) % 100;
               nW++;
             }
             break;
           }
           status = msgrcv(prijem, &m, sizeof(my_msg)-sizeof(long int), 0,
0);
         }
       msgctl(prijem, IPC_RMID, 0);
       return 0;
       }

       typedef struct {
            int s;
            int r;
       }TKanal;

       TKanal startRead(){
         TKanal k;
         int i = 1025, r = 0, s = 0;
         my_msg m;
         while ( (r = msgget((key_t)i,0666 | IPC_CREAT | IPC_EXCL)) < 0)
i++;
         s = msgget((key_t) P,0666 | IPC_CREAT);
         m.msg_type = 2;
         m.odgovor = r;
         if (s>-1) {
           msgsnd(s, &m, sizeof(my_msg)-sizeof(long int), 0);
           msgrcv(r, &m, sizeof(my_msg)-sizeof(long int), 0, 0);
         }
         k.s = s;
         k.r = r;
         return k;
       }

       void stopRead(TKanal k){
```

```
  my_msg m;
  m.msg_type = 3;
  m.odgovor = 0;
  msgsnd(k.s, &m, sizeof(my_msg)-sizeof(long int), 0);
  msgctl(k.r, IPC_RMID, 0);
  return;
}
```

operacije srartWrite i stopWrite su analogne operacijama startRead i stopRead, osim sto se tipovi iz 2 i 3 menjaju u 4 i 5.

primer koriscenja operacija:

```
TKanal k;
k = startRead();
//citanje
stopRead(k);
```

30. (ispit 2007) Operativni sistem Linux

Napisati program za operativni sistem Linux koji treba da ima funkciju servera. Za komunikaciju koristiti prosleđivanje poruka čiji je interfejs:

```
       int msgget(key_t key, int msgflg);
            //Gde je key ključ po kojem se traži red, msgflg, flegovi koji
određuju način    kreiranja:
            666 - vrati identifikator ako red postoji
            666 | IPC_CREAT - ako ne postoji kreiraj, ako postoji vrati
identifikator                 postojećeg
            666 | IPC_CREAT | IPC_EXCL - kreiraj jedino ako ne postoji.
            Funkcija vraća identifikator reda ako je operacija uspešna i -1
ako je neuspešna.

       int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int
msgflg);
            //Šalje poruku na koju pokazuje msg_ptr, čija je veličina
msg_sz u red sa   identifikatorom msqid. msgflg treba da je 0.

       int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int
msgtype, int msgflg);
            //Prihvata poruku iz reda sa identifikatorom msqid u bafer na
koji ukazuje      msg_ptr maksimalne veličine msg_sz. msgtype i msgflg
treba da su jednaki nuli.     Operacija je blokirajuća.

       int msgctl(int msqid, IPC_RMID, 0);
            //Oslobađa red sa identifikatorom msqid

       Poslednje tri funkcije vraćaju -1 u slučaju greške i neki
nenegativan broj u slučaju uspeha.
```

Server treba da osluškuje red poruka sa identifikatorom 1234. Po prijemu zahteva, potrebno je obezbediti da se izmedju klijenta i servera obezbedi dvosmerna veza, tako da u isto vreme mogu da šalju i jedan i drugi (full duplex). Takođe je potrebno napisati i klijent za taj server, kao i definisati sve potrebne podatke. Ni u klijentu, ni u serveru nije potrebno obavljati neki konkretan posao, već samo treba naznačiti mesta gde bi se to obavljalo.

Rešenje:

```
       //server
       #include <sys/msg.h>
       #include <stdio.h>
       #include <stdlib.h>

       typedef struct {
           long int message_type;
           int in,out;
       } my_message;

       int main(int argc, char *argv[])
       {
            my_message m;
            int srv = 1234;
            srv = msgget((key_t)srv,0666 | IPC_CREAT);
            if (srv == -1) {
                  printf("Nije kreiran red\n");
                  return 1;
            }else printf("Server pokrenut na %d \n",srv);

            int i;

            while (1){
```

```
                if (i = msgrcv(srv, &m, 2*sizeof(int), 0, 0) == -1){
                      printf("Neuspeo prijem\n");
                      return 1;
                }
                if (fork()==0){
                      printf("Red    za    prijem    %d,    za    slanje
%d\n",m.in,m.out);
                      //obrada
                      exit(0);
                }
            }
            msgctl(srv, IPC_RMID, 0);
            printf("Kraj rada servera\n");

          return EXIT_SUCCESS;
        }

        //klijent

        #include <sys/msg.h>
        #include <stdio.h>
        #include <stdlib.h>

        typedef struct {
            long int message_type;
            int in,out;
        } my_message;

        int main(int argc, char *argv[])
        {
             my_message m;
             int srv = 1234;
             m.message_type = 5;
             srv = msgget(srv, 0666);
             int i = 0, j = 1;
             while (i < 2){
                   if (i==0){
                         if (m.in = msgget(j, 0666 | IPC_EXCL | IPC_CREAT)
!= -1){
                               i++;
                         }
                   }else if (i==1){
                         if (m.out = msgget(j, 0666 | IPC_EXCL | IPC_CREAT)
!= -1){
                               i++;
                         }
                   }
                   j++;
             }

             if (i = msgsnd(srv, &m, 2*sizeof(int), 0) == -1){
                   printf("Neuspelo slanje\n");
                   return 1;
             }

             //komunikacija sa serverom preko redova in i out
             //kada vise ne trebaju m.in i m.out, osloboditi ih

             printf("Kraj rada klijenta");

             return EXIT_SUCCESS;
        }
```

31. (6. januar 2008.) Operativni sistem Linux

Napisati *shell script* koji treba da iz datoteke, zadate kao prvi parametar, izvuče one zapise o korisnicima čija šifra počinje onim što je zadato kao drugi parametar i koji su iz mesta zadatog trećim parametrom (izdvajaju se samo oni kod kojih su oba uslova zadovoljena). Ulazna datoteka sadrži zapise o korisnicima, i to u svakom redu za tačno jednog korisnika. Pojedinačne informacije u svakom redu su razdvojene tabulatorima. Podaci koji se vode o svakom korisniku su šifra (niz cifara), ime, ime oca, prezime, mesto i broj telefona. Telefon je zapisan u formatu *+<pozivni broj države><blanko znak><pozivni kod grada><blanko znak><broj telefona>*. U izlaznoj datoteci u svakom redu treba da budu informacije o korisnicima, ali u sledećem obliku i redosledu: početno slovo mesta za kojim sledi šifra (bez razmaka između), ime, prvo slovo očevog imena, prezime i broj telefona bez pozivnih brojeva. Prikazati kako se taj skript poziva ako želimo sve korisnike iz datoteke *ulaz.txt* čija šifra počinje sa 4 i koji su iz Beograda.

Primer ulaza:
43 Mika Laza Peric Beograd +381 63 123456

Izlaz za prethodni ulaz:
B43 Mika L Peric 123456

Rešenje:

Sadržaj datoteke `script1` u tekućem direktorijumu:

```
#!/bin/bash
cat   $1|grep   "^$2\([0-9]*\)\(.*\)$3.+\([0-9]*\)\([0-9]*\)   \([0-9]*\)"|
sed "s/\([0-9]*\)\t\(.*\)\t\(.\)\(.*\)\t\(.*\)\t\(.\)\(.*\)\t+\([0-9]*\)
\([0-9]*\) \([0-9]*\)/\6\1 \2 \3 \5 \10/"| cat >izl.txt
```

Primer poziva:

```
source script1 ulaz.txt 4 Beograd
```

32. (6. februar 2008.) Operativni sistem Linux

a)(5) Napisati shell skript za operativni sistem Linux koji treba da u pozadini pokrene sve izvršive fajlove iz zadatog foldera. Folder se zadaje kao argument skripta.

b)(5) Dva procesa u operativnom sistemu Linux imaju potrebu za razmenom velike količine podataka. Koji je najefikasniji način za ovu komunikaciju i zašto? Da li taj način ima i neke nedostatke?

Rešenje:
a)

```
#!/bin/bash
for i in $(ls $1)
do
if test –f $i –a –x $i
then
$i&
fi
done
```

b) Najefikasniji način prenosa podataka je prenos kroz deljenu memoriju. Takvim pristupom izbegava se propagacija kroz niz slojeva operativnog sistema kakva bi bila u drugim načinima komunikacije. Nedostatak deljene memorije je što ne obezbeđuje sinhronizaciju, pa je potrebno upotrebiti neki drugi mehanizam za sinhronizaciju (npr. semafore).

33. (6. jun2008.) Operativni sistem Linux

Precizno objasniti šta su i čemu služe atributi `__init`, `__initdata` i `__exit`.

Odgovor:

Atribut `__init` koristi se za obeležavanje funkcije koja vrši inicijalizaciju. U slučaju modula koji je ugrađen u jezgro, nakon završene inicijalizacije (procesa boot-ovanja), ovako ozbeležena funkcija se odbacuje i memorija koju je zauzimala se oslobađa. Ovakva mogućnost za sada ne postoji za module koji se naknadno učitavaju. Atribut `__initdata` ima istu ulogu kao atributu `__init` osim što se koristi za obeležavanje podataka koji se koriste samo pri inicijalizaciji. Atribut `__exit` koristi se za obeležavanje funkcije koja se poziva pri izbacivanju iz modula. U slučaju modula koji je ugrađen u jezgro, ovako obeležena funkcija se odbacuje. Treba napomenuti da ovi atributi ne definišu namenu neke funkcije kao init ili exit. To je neophodno učiniti odgovarajućim makroima.

34. (7. januar 2008.) Operativni sistem Linux

Implementirati funkciju `void parallelProcess(int n, int k, int a[][])`. Funkcija prihvata matricu *a* celih brojeva dimenzija *n* x *n* u kojoj je inicijalizovano prvih *k* redova. Funkcija treba da popuni matricu tako da za svaki element izvan prvih *k* redova važi `a[i][j] = f(a[i-k],j)`, gde je *f* data funkcija koja prihvata dva cela broja i vraća jedan ceo broj. Računanje je dugotrajan posao i treba ga paralelizovati što je više moguće. Smatrati da *n* nije veće od 100. Za komunikaciju i sinhronizaciju izmeću procesa na raspolaganju su sledeće funkcije za rad sa neimenovanim cevovodima (engl. *pipe*):

```
      int pipe(int fd[2]); // Kreira cev (engl. pipe) i vraća deskriptore:
fd[0] // za čitanje i fd[1] za pisanje u kreiranu cev. Funkcija prima kao
// argument pokazivač na niz od dva cela broja

      int read(int fd, void *buff, int numBytes); // čita podatke iz
fajla/cevi // sa deskriptorom fd i smešta u bafer buff velicine numBytes

      int write(int fd, void *buff, int numBytes); // upisuje bafer
podataka buff // velicine numBytes u file/cev sa deskriptorom fd

int close(int fd); // zatvara cev sa descriptorom fd
```

Rešenje:

```
      const int N = 100;
      void f(int niz[N],int j);
      int pipe(int fd[2]);
      int read(int fd, void *buff, int numBytes);
      int write(int fd, void *buff, int numBytes);
      int close(int fd);

      typedef struct {
        int i,j;
        int val
      } Item;


      int fileDesc[100][2];
      int return_pipe[2];

      void parallelProcess(int n, int k, int a[N][N]){
        Item item;

        pipe( return_pipe );

        for (int i = 0; i < k; ++i){
          pipe( fileDesc[i] );
          if (fork() == 0){
            i += k;

            while (i < n){
              item.i = i;
              //pokrenuti po jedan process za svaki element u redu i
              for(int j = 0; j < n; ++j)
                if (fork() == 0){
                  item.j = j;
                  item.val = f(a[i-k], j);
                  write(fileDesc[i][1], &item, sizeof(Item));
                  exit(0);
```

```
        }
      //sačekati da se svi elementi prikupe nazad i proslediti ih
      for(int j = 0; j < n; ++j){
        read(fileDesc[i][0], &item, sizeof(Item));
        a[i][item.j] = item.val;
        write(return_pipe[1], &item, sizeof(Item));
      }
      //pomaći se za k unaprijed
      i += k;

    }

    exit(0);
  }
}

for (int i = 0; i < (n-k)*n - 1; ++i){
  read(return_pipe[0], &item, sizeof(Item));
  a[item.i][item.j] = item.val;
}
close( return_pipe[0] );
close( return_pipe[1] );
for (int i = 0; i < k; ++i) {
  close(fileDesc[i][0]);
  close(fileDesc[i][1]);
}
```

35. (7. februar 2008.) Operativni sistem Linux

Implementirati funkciju void parallelProcess(int n, int a[][]). Funkcija prihvata matricu *a* celih brojeva dimenzija *n* x *n* u kojoj su inicijalizovani prva kolona i glavna dijagonala. Funkcija treba da popuni dio matrice ispod glavne dijagonale tako da za svaki element koji je potrebno izračunati važi `a[i][j] = f(a[i-1][j-1],a[i-1][j])`, gde je *f* data funkcija koja prihvata dva cela broja i vraća jedan ceo broj. Računanje je dugotrajan posao i treba ga paralelizovati što je više moguće. Smatrati da *n* nije veće od 100. Za komunikaciju i sinhronizaciju između procesa koristiti mehanizam prosleđivanja poruka (engl. *message passing*).

Rešenje:

```
void parallelprocess(int n, int a[100][100]){
 int i = 1000, j = 0, p = 0, k;
 my_msg l,d,r;

 while ( (p = msgget((key_t)i,0666 | IPC_CREAT | IPC_EXCL)) < 0)
i++;
 for(i=1;i<n;i++){
        if (i==(j+1)) { //samo krajnji proces se dijeli
                if (fork() == 0) j++;
        };
        if ((j==0)||(i==j)) {
                //poslati element koji je inicijalizovan
                r.val = a[i][j];
                r.msg_type = i*n+j;
                r.br_kor = 2;
                msgsnd(p, &r, sizeof(my_msg)-sizeof(long), 0);

        }else{
         //prihvatiti oba elementa iznad
         msgrcv(p, &l, sizeof(my_msg)-sizeof(long), (i-1)*n+j-1, 0);
         l.br_kor++;
         if (l.br_kor<3) msgsnd(p,&l,sizeof(my_msg)-sizeof(long),0);
         msgrcv(p, &d, sizeof(my_msg)-sizeof(long), (i-1)*n+j, 0);
         d.br_kor++;
         if (d.br_kor<3) msgsnd(p,&d,sizeof(my_msg)-sizeof(long),0);
         //izracunati
         r.val = f(l.val,d.val);
         r.msg_type = i*n+j;
         r.br_kor = 0;
         //poslati izracunati element
         msgsnd(p, &r, sizeof(my_msg)-sizeof(long), 0);
        }
 }
 if (j>0) exit(0); //ovde je kraj procesa koji su obrađivali kolone
                //nastavlja samo jedan, onaj koji je pokrenuo
                //obradu
 for (i=2;i<n;i++){
    for(j=1;j<i;j++){
      msgrcv(p, &r, sizeof(my_msg)-sizeof(long), i*n+j, 0);
      r.br_kor++;
      if (r.br_kor<3) msgsnd(p, &r, sizeof(my_msg)-sizeof(long), 0);
      a[i][j] = r.val;
    }
 }
 msgctl(p, IPC_RMID, 0);
 return;
}
```

Napomena: Zadatak se mogao rešiti i uz kreiranje više sandučadi, ali je ovde prikazano rešenje prikazalo korišćenje tipa poruke. Kreirano je samo jedno sanduče. Pošto se svaki izračunati element trebao iskoristiti 3 puta (za dva elementa ispod i da se vrednost vrati u matricu u procesu koji nastavlja izvršavanje). Zato je u strukturu uvedeno polje br_kor. Kada polje dostigne vrednost 3 to znači da taj element nije potreban nikome, pa ga nije potrebno posle korišćenja vraćati u sanduče.

36. (7. jun 2008.) Operativni sistem Linux

Posmatra se binarno stablo koje u listovima sadrži cele brojeve. Vrednosti ostalih čvorova u tom stablu dobijaju se kao rezultat poziva funkcije `f()` kojoj se proslede vrednost levog i desnog potomka. Uzimajući u obzir da izračunavanje ove funkcije traje dugo, na programskom jeziku C napisati funkciju `int parallelProcess(struct cvor *koren)` koja izračunava vrednost korena stabla čija se adresa prosleđuje kao jedini argument funkcije. Struktura `cvor` sadrži pokazivače `l` i `r` na potomke, kao i celobrojno polje `val` koje u slučaju listova stabla sadrži vrednost čvora. Smatrati da je dato stablo ispravno, tj da se ne može desiti da u nekom čvoru postoji samo jedan potomak. Obzirom na cenu kreiranja novog procesa, voditi računa da se novi proces kreira samo onda kada je to zaista potrebno (kada će se u okviru procesa pozivati funkcija `f()`). Za komunikaciju između procesa koristiti neimenovane cevovode čiji je interfejs dat u nastavku.

```
       int pipe(int fd[2]); // Kreira cev (engl. Pipe) i vraća deskriptore:
fd[0] // za čitanje i fd[1] za pisanje u kreiranu cev. Funkcija prima kao
// argument niz od dva cela broja

       int read(int fd, void *Buff, int NumBytes); // čita podatke iz
fajla/pipea // sa deskriptorom fd i smešta u bafer Buff velicine NumBytes

       int write(int fd, void *Buff, int NumBytes); // upisuje bafer
podataka Buff // velicine NumBytes u file/pipe sa deskriptorom fd

       int close(int fd); // zatvara descriptor fd
```

Rešenje:

```
int parallelProcess(struct cvor *root){
      int l,r,pd[2];
      if (root->left == null)   //po pretpostavci zadatka, svaki cvor
                              //ili ima oba potomka, ili nema nijednog
            return root->val;
      else if ((root->left->left != null)&&(root->right->left != null)){
            pipe(pd);
            if (fork() == 0) {
                  r = parallelProcess(root->right);
                  write(pd[1],&r,sizeof(int));
                  exit(0);
            }
            l = paralellProcess(root->left);
            read(pd[0],&r, sizeof(int));
            close(pd[0]);
            close(pd[1]);
            return f(l,r);
}else return f(parallelProcess(root->left),parallelProcess(root->right))
}
```

37. (ispit 2009) Operativni sistem Linux

a)(5) Napisati *shell script* koji treba da pređe u direktorijum zadat kao prvi argument, pozove program zadat kao drugi argument prosledivši mu sve ostale argumente kao njegove argumente, ispiše da li se pokrenut program uspešno izvršio i na kraju se vrati u polazni direktorijum. Ako nije prosleđeno dovoljno parametara ispisati odgovarajuću poruku.

Rešenje:

b)(5) Da li se skriptu, napisanom u prethodnoj tački, može proslediti na izvršavanje isti takav skript koji se nalazi u ciljnom direktorijumu različitom od tekućeg, a da po izvršavanju stanje sistema bude kako je očeivano na osnovu postavke zadatka? Precizno i kratko obrazložiti

a)(5)

```
#!/bin/bash
if test $# -ge 2
then
tempdir=$PWD
cd $1
prog=$2
shift
shift
if $prog $*
then
echo "Program uspesno izvrsen"
else
echo "Program nije uspesno izvrsen."
fi
cd $tempdir
else
echo "Nije prosledjeno dovoljno parametara"
fi
```

b)(5) Može se izvršiti traženi poziv. Pred kraj prvog (okružujućeg) poziva sistem će biti vraćen u polazni direktorijum sačuvan na početku skripta. Ugniježđeni skript ne utiče nikakvim bočnim efektom na stanje pozivajućeg skripta, kao što ni okružujući poziv ne utiče na okruženje.

38. (ispit 2009) Operativni sistem Linux

Napisati *shell script* koji prihvata do 9 parametara. Prvi parametar je naziv fajla koji sadrži informacije o programima, od kojih će neke biti potrebno pokrenuti. U svakom redu tog fajla zapisani su sledeći podaci o jednom programu: kontrolni broj (niz cifara), razmak, naziv programa (niz malih slova), razmak, opis (tri mala slova), razmak i na kraju niz parametara koje programu treba proslediti (nizovi znakova različitih od vitičasti zagrada razdvojenih razmacima, svi zajedno ograđeni vitičastim zagradama). Preostali parametri su redni brojevi redova iz kojih treba pokrenuti programe.Ukoliko je broj parametara neodgovarajući, ili ako dodje do bilo koje druge greske, ispisati gresku i nastaviti izvršavanje ukoliko to ima smisla.

Rešenje:

```
#!/bin/bash
if test $# -lt 2 -o $# -gt 9
then
echo "Neodgovarajuci broj parametara"
else
ul=$1
if test -f $ul -a -r $ul
then
shift
redovi=$1
shift
while test $# -gt 0
do
redovi="$redovi\|\\($1\\)"
shift
done
cat -n $ul | grep "^ *\($redovi\)   " | sed 's/\ *[0-9]*\t[0-9]*\ \([a-
   z]*\)\ [a-z]*\ {\(.*\)}/\1 \2 && echo \"Ok\.\" || echo \"Greska pri
   izvrsavanju \1 \2\"/' | /bin/bash
else
echo "Fajl $ul nije dostupan"
fi
fi
```

39. (ispit 2009) Operativni sistem Linux

a)(5) Uporediti dva nacina alokacije prostora na disku za zamenu stranica u pogledu brzine i fleksibilnosti. Precizno i kratko obrazložiti.

Odgovor:
U operativnom sistemu Linux prostor za zamenu stranica na disku može biti fajl u postojećem fajl sistemu ili za to posebno namenjena particija. Zamena stranica pomoću fajla je znatno sporija jer se za pristup stranici mora proći čitav niz sistemskih poziva, koji nisu potrebni u slučaju pristupa posebnoj particiji namenjenoj za zamenu. Međutim, u slučaju neodgovarajuće veličine rezervisane particije, menjanje veličine iste može biti jako složen postupak ukoliko je ta particija sa obe strane okružena drugim korisnim particijama. U slučaju fajla, pod uslovom da postoji dovoljno slobodnog prostora, povećanje veličine prostora za zamenu stranica je trivijalno.

b)(5) Napisati shell komandu koja ispisuje detaljne informacije o svim blokovskim uređajima u koje može da upisuje bilo koji korisnik.

Rešenje:

```
ls -l /dev | grep "^b.......w.*"
```

40. (ispit 2009) Operativni sistem Linux

Napisati *bash shell* skript koji treba da sortira spisak studenata po prezimenu i imenu. Spisak studenata je dat u tekstualnom fajlu čiji se naziv prosleđuje kao prvi parametar skripta. Sortiran spisak ispisati na standardni izlaz. U ulaznom fajlu, svaki student je zapisan u po jednom redu po sledećem formatu: redni broj, broj indeksa, ime, prezime i ocjena. Sva polja su razdvojena razmakom. Na izlazu za svakog studenta zapisati po jedan zapis po sledećem formatu: redni broj, redni broj iz ulaznog fajla, broj indeksa, prezime, ime i ocjena. Smatrati da je ulazni fajl ispravan ukoliko postoji. U slučaju pogrešnog broa parametara, nepostojanja ulaznog fajla ili nemogućnosti čitanja istog ispisati poruku o grešci i prekinuti izvršavanje skripta.

Odgovor:

```
#!/bin/bash
if test $# -eq 1
then
if test -f $1 -a -r $1
then
cat $1 |
sed "s/\([0-9]*\.\)\ \([0-9]*\/[0-9]*\)\ \([a-z,A-Z]*\)\ \([a-z,A-Z]*\)\
\([0-9]*\)/\4 \3 \1 \2 \5/"| sort | grep -n ".*" |
sed "s/\([0-9]*\):\([a-z,A-Z]*\)\ \([a-z,A-Z]*\)\ \([0-9]*\.\)\ \([0-
9]*\/[0-9]*\)\ \([0-9]*\)/\1. \4 \5 \2 \3 \6/"
else
echo "Greska: ulazni fajl ili ne postoji ili mu se ne moze pristupiti";
fi
else
echo "Pogresan broj parametara. $#";
fi
```

41. (ispit 2009) Operativni sistem Linux

Date su funkcije `int f1(int x, int y)` i `int f2(int x, int y)` čija izvršavanja traju približno isto i dugo. Potrebno je izračunati narednih 1000 elemenata nekog reda čijih je prvih 100 elemenata poznato i za koji važi da je $a_{2n} = f1(a_{2n-100},a_{2n-99})$ i $a_{2n+1} = f2(a_{2n-100},a_{2n-99})$ za n>49. Napisati program za Linux koji u paralelnim procesima vrši traženo izračunavanje, tako da se proces paralelizuje u najvećoj mogućoj meri. Za komunikaciju i sinhronizaciju koristiti neimenovane cevi.

Rešenje:

```
typedef struct{
      int ind;
      int val;
} Message;

int a[1100];
int pip[100][2];
int rer[2];

void main(){
      int i, x, y, j = 0;
      Message m;
//.........ucitavanje pocetnih vrednosti podataka
      pipe(ret);
      for(i = 0; i < 100; i++){
            if (i%2 == 0) {pipe(pip[i]); pipe(pip[i+1]);}
            if (fork()){
                  if (i%2) {
                        x = a[i-1];
                        y = a[i];
                        while (j<10){
                              m.val = y = f2(x,y);
                              m.ind = i + 100 * ++j;
                              write(ret[OUT],&m,sizeof(Message));
                              write(pip[i-1][OUT],&m,sizeof(Message));
                              if (j>=10) continue;
                              read(pip[i][IN],&m,sizeof(Message));
                              x = m.val;
                        }
                  }else {
                        x = a[i];
                        y = a[i+1];
                        while (j<10){
                              m.val = x = f2(x,y);
                              m.ind = i + 100 * ++j;
                              write(ret[OUT],&m,sizeof(Message));
                              write(pip[i+1][OUT],&m,sizeof(Message));
                              if (j>=10) continue;
                              read(pip[i][IN],&m,sizeof(Message));
                              y = m.val;
                        }
                  }
                  exit(0);
            }
      }
      for(i = 100; i < 1100; i++){
            read(ret[IN],&m,sizeof(Message))
            a[m.ind] = m.val;
      }
```

```
        close(ret[IN]);
        close(ret[OUT]);
        for(i = 0; i<100; i++){
              close(pip[i][IN]);
              close(pip[i][OUT]);
        }
        //.....podaci spremni za dalje koriscenje
}
```

42. (ispit 2009) Operativni sistem Linux

Dat su sledeća dva niza:

```
#define N ...;
int (*f[N])(int x);
void* x[N];
```

Funkcija f[i]() pri računanju koristi element x[i] i u njemu čuva međurezultat, ali nije poznata veličina polja na koje x[i] pokazuje. Trajanja izvršavanja svih funkcija su približno iste dužine. Potrebno je izračunati narednih 999 elemenata nekog reda čiji je prvi element poznatt i za koji važi da je a_i = f[N-1](f[N-2](...f[1](a_{i-1})...)) za i>1. Napisati program za Linux koji u paralelnim procesima vrši traženo izračunavanje, tako da se proces paralelizuje u najvećoj mogućoj meri, ali da se ne kreira više procesa nego što je potrebno. Za komunikaciju i sinhronizaciju koristiti neimenovane cevi.

Rešenje:

```
#define N ...;
int (*f[N])(int x);
void* x[N];

int a[1000];
int pip[N][2];
int ret[2];

void main(){
      int i, j;
//.........inicijalizacija podataka
      pipe(ret);
      for(i = 0; i < N; i++) pipe(pip[i]);
      write(pipe[0][OUT], a, sizeof(int));
      for(i = 0; i < N; i++)
            if (fork()){
                  for(j = 0; j<999; j++){
                        read(pip[i][IN], a, sizeof(int));
                        a[0] = (*f[i])(a[0]);
                        write(pip[(i+1)%N][OUT], a, sizeof(int));
                        if (i == N-1) write(ret[OUT], a, sizeof(int));
                  }
                  exit(0);
            }
      for(i = 1; i < 1000; i++){
            read(ret[IN],a+i,sizeof(int)
      }
      close(ret[IN]);
      close(ret[OUT]);
      for(i = 0; i<N; i++){
            close(pip[i][IN]);
            close(pip[i][OUT]);
      }
      //.....podaci spremni za dalje koriscenje
}
```

43. (ispit 2009) Operativni sistem Linux

Data je funkcija koja generiše pseudoslučajne brojeve i pri tome koristi jedan statički podatak. Funkcija generiše vrednosti koje imaju normalnu raspodelu i koje zavise i od trenutka poziva funkcije. Deklaracija date funkcije je:

```
int f();
```

Napisati program za Linux koji iz dva paralelna procesa ispisuje: "ping" iz prvog, i "pong" iz drugog. Prvi ispis je iz prvog procesa, ispisuje se a_1 puta "ping", zatim se iz drugog ispisuje a_2 puta "pong", zatim opet iz prvog a_3 puta "ping", pa opet iz drugog a_4 puta "pong" i tako dalje naizmenično, svaki put sa drugim konstantama a_i. Pomenute konstante a_1, a_2, a_3, a_4, a_5, ... moraju biti rezultat uzastopnih poziva funkcije f i to tako da je a_i rezultat i-tog poziva funkcije f. Za komunikaciju i sinhronizaciju koristiti mehanizam prosleđivanja poruka. Dozvoljeno je kreirati samo jedno sanduče. Pretpostaviti da je moguće kreirati bar jedno sanduče sa ključem većim od 1000.

Rešenje:

```
#include<stdio.h>
#include<stdlib.h>
#include<linux/ipc.h>

typedef struct{
  long mtype;
  int val;
} my_msg;

int f();

int main(){
int ai, i = 1000, j, p;
my_msg msg;

while ( (p = msgget((key_t)i,0666 | IPC_CREAT | IPC_EXCL)) < 0) i++;

if (fork() == 0) {
  for (j = 0; j < 10; j++){
    msgrcv(p, (struct msgbuf *)&msg, sizeof(my_msg)-sizeof(int), 1,
0);
    for(i = 0; i<msg.val; i++) printf("pong"); printf("\n");
    msg.mtype = 2;
    msgsnd(p,(struct msgbuf *)&msg,sizeof(my_msg)-
sizeof(int),IPC_NOWAIT);
  }
  exit(0);
}
for (j = 0; j < 10; j++){
  ai = f();
  for(i = 0; i<ai; i++) printf("ping"); printf("\n");
  msg.val = f();
  msg.mtype = 1;
  msgsnd(p,(struct msgbuf *)&msg,sizeof(my_msg)-
sizeof(int),IPC_NOWAIT);

  msgrcv(p, (struct msgbuf *)&msg, sizeof(my_msg)-sizeof(int), 2,
0);
}
msgctl(p, IPC_RMID, 0);
return 0;
}
```

44. (ispit 2009) Operativni sistem Linux

Date su dve funkcije:

```
int f1(int);
int f2(int);
```

Izračunavanje obe funkcije je sličnog, dugog trajanja. Obe funkcije imaju svoje statičke podatke koje koriste pri izračunavanju ali i za čuvanje stanja između uzastopnih izračunavanja. Pomoću ovih funkcija potrebno je izračunati prvih 100 elemenata dva reda. Vrednosti početnih elemenata redova su $a_0 = 1$, $b_0 = 2$. Elementi redova a i b su korelisani preko funkcija f1 i f2 samim načinom računanja: $a_{2i} = f1(a_{2i-1})$, $b_{2i} = f2(b_{2i-1})$, $a_{2i+1} = f2(a_{2i})$ i $b_{2i+1} = f1(b_{2i})$. Zbog korelisanosti redova, da bi se ispravno izračunali svi elementi, nije dozvoljeno početi izračunavanje elementa sa indeksom i bilo kojeg reda pre nego što se izračunaju elementi sa indeksom i-1 u oba reda. Napisati program na programskom jeziku C za operativni sistem Linux koji vrši opisano izračunavanje u optimalnom broju paralelnih procesa. Za komunikaciju i sinhronizaciju koristiti neimenovane cevi.

Rešenje:

```
#define N 100
#define IN 0
#define OUT 1

int f1(int x);
int f2(int x);

int a[100], b[100];
int pip12[2], pip21[2];

void main(){
      int i, j;
  a[0]=1;
  b[0]=2;
      pipe(pip12);
      pipe(pip21);
      if (fork()=0){
            int *x = b, i = 0;
            for(;i<100;i++){
                  x[i+1] = f1(x[i])
            write(pip12[OUT],x+i+1 , sizeof(int));
            if (x==a) x = b;
                  else x = a;
            read(pip21[IN],x+i+1,sizeof(int))
            }
      }else{
            int *x = a, i = 0;
            for(;i<100;i++){
                  x[i+1] = f2(x[i])
            write(pip21[OUT],x+i+1 , sizeof(int));
            if (x==a) x = b;
                  else x = a;
            read(pip12[IN],x+i+1,sizeof(int))
            }
      }
      close(pip12[IN]);
      close(pip12[OUT]);
      close(pip21[i][IN]);
      close(pip21[i][OUT]);

      //.....podaci spremni za dalje koriscenje u nizovima a i b
}
```

45. (ispit 2010) Operativni sistem Linux

Napisati *bash shell* skriptu koja prihvata nazive dve datoteke. Prva datoteka sadrži reči koje se traže u drugoj datoteci. Potrebno je iz druge datoteke izdvojiti sve one redove koji počinju s nekom od reči iz prve datoteke i ispisati ih na standardnom izlazu.

Rešenje:

```
#!/bin/bash
for i in $(<$1)
do
cat $2 | grep "^$i.*"
done
```

46. (ispit 2010) Operativni sistem Linux

Kako je u operativnom sistemu Linux smanjen veličina prostora potrebnog za održavanje logičkih informacija o stranicama virtuelnog adresnog prostora procesa? Šta je urađeno kako bi se performanse što manje degradirale?

Odgovor:

Logičke informacije o grupi kontinualnih, logički povezanih stranica se čuvaju u zapisu tipa `vm_area_struct`. Da bi se za zadatu virtuelnu adresu brže pronašla potrebna informacija, zapisi o grupama stranica su organizovani u balansirano binarno stablo.

47. (ispit 2010) Operativni sistem Linux

Napisati bash shell skript koji kao parametre prihvata nazive programa koje treba da pokrene, tako da izlaz prvog preusmeri kroz neimenovanu cev na ulaz drugog, izlaz drugog na isti način na ulaz trećeg i tako sve do poslednjeg programa u listi, čiji izlaz ostaje standardni izlaz. Ukoliko se uoči da neki od programa iz bilo kog razloga nije moguće izvršiti, ispisati poruku o grešci i prekinuti izvršavanje. Grešku treba prijaviti i u slučaju da skriptu nije prosleđen nijetan argument.

Rešenje:

```
#!/bin/bash
if test $# -eq 0
then
echo "Nije proslijedjen nijedan parametar"
exit 1
fi
while test $# -gt 0
do
if test -e $1 -a -x $1
then
if test -z $KOMANDA
then
KOMANDA="./$1"
else
KOMANDA="$KOMANDA | ./$1"
fi
else
echo "Fajl $1 ne postoji ili nemate dozvolu za izvrsavanje"
exit 2
fi
shift
done
$KOMANDA
```

48. (ispit 2010) Operativni sistem Linux

Napisati *Bash shell script* koji treba da za zadati spisak fajlova prebroji koliko od tih fajlova postoji u tekućem direktorijumu. Spisak fajlova se zadaje kao niz parametara skripta. Izlaz skripta treba da bude u formatu: <broj nadjenih fajlova>/<broj fajlova na spisku>.

Rešenje:

```
#!/bin/bash
ukupno=0
postoji=0
while test $# -ge 1
do
let ukupno++
if test -f $1
then
let postoji++
fi
shift
done
echo "$postoji/$ukupno"
```

49. (ispit 2010) Operativni sistem Linux

a)(5) Izlaz programa "testiranje" nije od značaja, dok standartni izlaz za greške treba proslediti u fajl "greske.txt". Napisati komandu kojom se program "testiranje" pokreće na opisani način.

Rešenje:

b)(5) Kako je u Linux-u ubrzano pokretanje programa? Objasniti kratko i precizno.

Odgovor.
a)(5) `./testiranje >/dev/null 2>greske.txt`
b)(5) Loader u Linux-u ne učitava stranic pri pokretanju programa već ih samo preslikava u adresni prostor procesa. Stranice se učitavaju na zahtev. Na taj način je ubrzano pokretanje jer na početku izvršavanja se učitavaju samo oni delovi programa koji će stvarno biti korišćeni.

50. (ispit 2010) Operativni sistem Linux

Napisati program za operativni sistem Linux koji u N/2 (N je paran broj i to je broj vrsta) paralelnih procesa vrši sledeću obradu nad matricom. Prvo se usrednjavaju vrste sa neparnim indeksom, koristeći sledeću formulu za svaki element a[i,j] = (a[i,j] + a[i-1,j] + a[i+1,j] + a[i,j-1]) / 4, tako da je a[i,j-1] element prethodno izračunat u istoj iteraciji. Kada se računanje za vrste sa neparnim indeksom završi, prelazi se na usrednjavanje vrsta sa parnim indeksom, korišćenjem iste formule kao i za vrste sa neparnim indeksom. Postupak se ponavlja 10 puta. Za sve vrste i kolone koje nedostaju (pri računanju graničnih elemenata), uzeti vrednost 0. Za potrebe komunikcije i sinhronizacije koristiti neimenovane cevi (engl. *pipe*).

Rešenje:

```
const int N = 10;

typedef struct {
  int i, j, v;
} Bafer;

void main(){
  int a[N][N], i,j,k = 10, cevi[N/2][2], povratna[2];
  Bafer b;
//... ucitavanje podataka

  for( i=0; i<N/2; i++) pipe(cevi[i]);
  pipe(povratna);

  for( i=0; i<N/2; i++)
    if (fork() == 0) {
      while (k>0){
        k--;
        for( j=0; j<N; j++)
          a[2*i+1][j] = ( a[2*i+1][j] + a[2*i][j] +
            (i==N/2-1?0:a[2*i+2][j]) + (j>0?a[2*i+1][j-1]:0) ) / 4;

        if (i<N/2-1)
          for( j=0; j<N; j++)
            write(cevi[i+1][1], &a[2*i+1][j], sizeof(int));
        if (i>0)
          for( j=0; j<N; j++)
            read(cevi[i-1][0], &a[2*i-1][j], sizeof(int));

        for( j=0; j<N; j++)
          a[2*i][j]   =   (   a[2*i][j]   +   (i==0?0:a[2*i-1][j])   +
a[2*i+1][j] +
            (j>0?a[2*i][j-1]:0) ) / 4;
        if (i>0)
          for( j=0; j<N; j++)
            write(cevi[i-1][1], &a[2*i-1][j], sizeof(int));
        if (i<N/2-1)
          for( j=0; j<N; j++)
            read(cevi[i+1][0], &a[2*i+1][j], sizeof(int));
      }
      for (j = 0; j < N; j++) {
        b.j = j;
        b.i = 2 * i + 1;
        b.v = a[b.i][j];
        write(povratna[1], &b, sizeof(Bafer));
        b.i = 2 * i;
        b.v = a[b.i][j];
        write(povratna[1], &b, sizeof(Bafer));
      }
```

```
    exit(0);
  }
  for(i = 0; i < N*N; i++){
    read(povratna[0], &b, sizeof(Bafer));
    a[b.i][b.j] = b.v;
  }
  for(i = 0; i < N; i++){
    close(cevi[i][1]);
    close(cevi[i][0]);
  }
  close(povratna[1]);
  close(povratna[0]);
}
```

51. (ispit 2010) Operativni sistem Linux

Na programskom jeziku C napisati program za operativni sistem Linux koji u optimalnom broju paralelnih procesa vrši sledeće izračunavanje. Izračunava se narednih 100 elemenata reda čijih je prvih 6 elemenata unapred dato i za koji su date formule po kojima se računaju ostali elementi: $a_{2i} = f1(a_{2i} - 6)$ i $a_{2i+1} = f2(a_{2i+1} - 6)$. Poznato je da funkcije imaju i statičke podatke (svaka svoje) koji učestvuju u izračunavanju. Paralelizacija mora biti takva da rezultati budu identični kao da su elementi reda računati redom, sekvencijalno unutar jednog procesa. Za komunikaciju koristiti cevi.

Rešenje:

```
        const int N = 100;
        int f1(int x);
        int f2(int x);

        void main(){
          int a[N+6], i,j, povratna[2];
          Bafer b;
        //... ucitavanje prvih 6 elemenata

          pipe(povratna);
          if (fork() == 0) {
            for(i = 6; i < N + 6; i += 2) {
              a[i] = f1(a[i-6]);
              write(povratna[1], a + i, sizeof(int));
            }
            exit(0);
          } else {
            for(i = 7; i < N + 6; i += 2)
              a[i] = f2(a[i-6]);
            for(i = 6; i < N + 6; i += 2)
              read(povratna[0], a + i, sizeof(int));
        //... koriscenje izracunatih podataka
          }
          close(povratna[1]);
          close(povratna[0]);

}
```

52. (ispit 2010) Operativni sistem Linux

Za operativni sistem Linux, napisati program na programskom jeziku C koji treba da iz N paralelnih procesa ispiše redom brojeve od 0 do M*N-1, tako da apsolutna razlika uzastopno ispisanih brojeva iz jednog procesa bude N. Dakle, jedan proces treba da ispiše brojeve 0, N, 2N,...; drugi proces ispisuje brojeve 1, N+1, 2N+1, itd. Za potrebe komunikacije i sinhronizacije između procesa dozvoljeno je koristiti isključivo semafore. Svi alocirani resursi moraju biti oslobođeni.

Rešenje:

```
#include<stdio.h>
#include<sys/sem.h>
#include<stdlib.h>

#define N 5
#define M 5

int main(){
	int i, j, sem_id;
	sem_id = semget((key_t)123, N, 0666 | IPC_CREAT);
	struct sembuf sem;
	for(i=0;i<N;i++){
		if(fork()==0){
			for(j=0;j<M;j++){
				//sem[i].wait();
				sem.sem_num = i;
				sem.sem_op = -1;
				sem.sem_flg = SEM_UNDO;
				semop(sem_id, &sem, 1);

				printf("%d\n",i+j*N);

				//sem[(i+1) % N].signal()
				sem.sem_num = (i+1)%N;
				sem.sem_op = 1;
				sem.sem_flg = SEM_UNDO;
				semop(sem_id, &sem, 1);
			}
			if (i==N-1) {
				semctl(sem_id, 0, IPC_RMID);
			}
			exit(0);
		}
	}
	//sem[0].signal();
	sem.sem_num = 0;
	sem.sem_op = 1;
	sem.sem_flg = SEM_UNDO;
	semop(sem_id, &sem, 1);
}
```

53. (ispit 2010) Operativni sistem Linux

Napisati potprogram na programskom jeziku C za operativni sistem Linux koji će izračunati istu sumu kao i sledeći potprogram (stablo ne mora biti popunjeno na isti način), ali u optimalnom broju paralelnih procesa:

```
int f(cvor *k){
  int r = 0;
  if (k->l != null) {
    k->l->v = hl(k->v);
    k->d->v = hd(k->v);
    r+=k->l->v+k->d->v+f(k->l)+f(k->d);
    return r;
  } else return hl(k->v) + hd(k->v);
}
```

Funkcije hl i hd koriste samo lokalne promenljive, jedini način prenosa podataka iz funkcije je preko povratne vrednosti i izvršavanje ovih funkcija traje približno isto i dugo. Poznato je da se uvek poziva za puno binarno stablo čija je dubina N (puno stablo dubine N ima 2^N-1 čvorova), gde je N celobrojna konstanta. Smatrati da pri pozivu sistemskih funkcija neće doći do greške. Za komunikaciju i sinhronizaciju koristiti neimenovane cevi.

Rešenje:

```
int f(cvor *k){
  int p[2]; s = 0, i, r;
  r = k->v;
  pipe(p);
  if(fork()==0){
    for(i=0;i<N;i++)
      if (fork(0)==0) {
        r = hl(r);
        write(p[1],&r,sizeof(int));
      }else{
        r = hd(r);
        write(p[1],&r,sizeof(int));
      }
    exit(0);
  }
  M=2
  for(i=0;i<N;i++){
    for(j=0;j<M;j++){
      read(p[0],&r, sizeof(int));
      s += r;
    }
    M *= 2;
  }
  close(p[0]);
  close(p[1]);
  return s;
}
```

54. (ispit 2010) Operativni sistem Linux

Neki programer je rešavao sledeći problem. Potrebno je izračunati prvih N elemenata nekog reda koji je zadat sa: $a_0 = 5$ i $a_i = f(a_{i-1})$. Funkcija f u svom radu koristi samo prosleđeni parametar i lokalne promjenljive. Rešenje koje je ponudio programer je sledeće:

```
void main(){
      int zeton = 0;
      int cevi[N+1][2];
      int i;
      int a[N];
      a[0] = 5;
      for (i=0;i<N;i++) pipe(cevi[i]);
      for (i=1;i<N;i++){
            if (fork() == 0){
                  read(cevi[i][IN], &zeton, sizeof(int));
                  a[i] = f(a[i-1]);
                  printf("%d\n",a[i]);
                  write(cevi[i+1][OUT], &zeton, sizeof(int));
                  exit(0);
            }
      }
      write(cevi[1][OUT], &zeton, sizeof(int));
}
```

Dato rešenje ima više nedostataka koji ne utiču na ispravnost ispisa i kao takve ih ne treba uzimati u obzir pri obrazlaganju odgovora na sledeće pitanje. Da li će program ispisati korektne vrijednosti ($a_1, a_2,...,a_{N-1}$)? Odgovor kratko i precizno obrazložiti.

Odgovor:

Ne. Programer je ispravno sinhronizovao računanja, tako da se element a_i uvek računa tek pošto je izračunat element a_{i-1}, ali je previdjeo da se te vrednosti izračunavaju u različitim adresnim prostorima, te da će pri izračunavanju elementa a_i biti korišćena neka slučajna vrijednost iz niza a umesto prethodno izračunate vrednosti a_{i-1}.

55. (ispit 2011) Operativni sistem Linux

Napisati bash shell script koji od spiska studenata formira listu e-mail adresa onih studenata koji imaju prosek zadat drugim argumentom skripta. Spisak studenata se nalazi u fajlu čiji je naziv zadat prvim argumentom skripta. Svaka linija ulaznog fajla sadrži zapis o jednom studentu u sledećem formatu: <ime><razmak><prezime><razmak><broj indeksa><razmak><prosek>. Indeks je zapisan u formatu <godina>/<broj>, gde su i <godina> i <broj> zadati sa po 4 cifre. Prosek je zaokružen na ceo broj. E-mail adrese se formiraju po sledećem formatu: <prvo slovo prezimena><prvo slovo imena><zadnje dve cifre godine><4 cifre broja indeksa>@student.etf.rs. Navođenje manje ili više argumenata, kao i navođenje fajla koji iz nekog razloga nije moguće pročitati, smatra se greškom. U slučaju greške prekinuti izvršavanje skripta.

Rešenje:

```
#!/bin/bash
if [ $# -eq 2 ]
then
  if [ -f $1 -a -r $1 ]
  then
    grep "$2$" $1 | sed "s_\(.\).* \(.\).* ..\(..\)/\(....\)
.*_\2\1\3\4@student.etf.rs_"
  else
    echo "Ulazni fajl nije moguce otvoriti"
  fi
else
  echo "Neodgovarajuci broj parametara"
fi
```

56. (ispit 2011) Operativni sistem Linux

a)(5) Napisati bash shell script koji prihvata jedan fajl koji sadrži nazive foldera u koje redom treba prelaziti i tražiti fajl zadat drugim argumentom. Izvršavanje se završava ako se nađe fajl ili kada se obiđu svi zadati direktorijumi. Izvršavanje se završava i ako se ne uspe preći u neki zadati direktorijum. Nakon završetka pretrage, ukoliko je fajl nađen, ostaviti za tekući direktorijum onaj u kojem je nađen fajl. U suprotnom, vratiti sistem u isto stanje kao neposredno pred poziv.

Rešenje:

a)

```
#!/bin/bash
if [ $# -ne 2 ]
then
      echo "Nekorektan broj prosledjenih argumenata. Za pokretanje treba
obezbediti tacno dva argumenta."
else
      oldWD=$PWD
      for i in $(<$1)
      do
            if cd $i
            then
                  if [ -f $2 ]
                  then
                        echo "Nadjen fajl."
                        return 0
                  fi
            else
                  echo "Neuspesan prelazak u direktorijum $i."
                  cd $oldWD
                  return 1
            fi
      done
      echo "Fajl nije nadjen u zadatim direktorijumima."
      cd $oldWD
      return 2
fi
```

b)(5) Da li je postoje neka ograničenja kojih se pri pozivanju skripta mora pridržavati kako bi skript iz zadatka a) radio ispravno? Kratko i precizno obrazložiti odgovor.

Odgovor:

b) Pored ostalih ograničenja koja su nametnuta samom postavkom zadatka, ovaj skript se mora pozvati korišćenjem komande source. Na taj način će se pozvani skript izvršiti u kontekstu školjke iz koje se poziva, te će tako nakon izvršavanja skripta ostati vidljive sve izmene, uključujući i moguću promenu tekućeg direktorijuma. Ako bi se skript pozvao prostim navođenjem naziva skripta, tada bi se za potrebe izvršavanja skripta kreirao novi proces školjke koji bi se završio završavanjem skripta, dok bi školjka iz koje je skript pozvan ostala neizmenjena. To znači da ne bi moglo doći do promene tekućeg direktorijuma u pozivajućoj školjki.

57. (ispit 2011) Operativni sistem Linux

a)(5) Da li je u operativnom sistemu Linux moguće izgladnjivanje usled raspoređivanja procesa po prioritetu? Kratko i precizno obrazložiti odgovor.

Odgovor:

Izgladnjivanje procesa zbog raspoređivanja po prioritetima u Linux-u nije moguće jer algoritam raspoređivanja zahteva da svaki proces dobije procesor na dodeljeni vremenski kvant pre nego što bilo koji drugi spreman proces dobije novi kvant vremena. To je implementirano tako što se proces kome je istekao dodeljeni kvant vremena prebaci u listu spremnih procesa koji nisu kandidati za dobijanje procesor (expired) i tu zadržava dok se lista spremnih procesa koji jesu kandidati za dobijanje procesora (active) ne isprazni. Nakon toga svi procesi se vraćaju u listu spremnih procesa kandidata za dobijanje procesora i time im se dodeljuje novi kvant vremena.

b)(5) Napisati komandu za bash shell koja će učiniti da se prilikom kasnijih pokretanja programa ti programi traže i u tekućem radnom direktorijumu.
Rešenje:
PATH="$PATH::"

58. (ispit 2011) Operativni sistem Linux

Napisati *bash shell* skript koji vrši pripremu studentskih projekata za automatizovano poređenje. Skript ima dva parametra. Prvi parametar je putanja do direktorijuma u kojem svaki student ima svoj direktorijum. U direktorijumu svakog studenta smešten je projekat i to tako da se svi `.cpp` fajlovi nalaze u direktorijumu `"cpp"`, dok se svi `.h` fajlovi nalaze u direktorijumu `"h"`. Skript treba da u odredišnom direktorijumu, zadatom drugim argumentom, za svakog studenta napravi po jedan odredišni direktorijum studenta istog naziva kao i onaj u izvorišnom direktorijumu, a zatim da u odredišnom direktorijumu studenta formira fajl `"izlaz.cpp"` čiji će sadržaj dobiti spajanjem svih `.cpp` i `.h` fajlova koji su deo projekta studenta koji se posmatra. Pretpostaviti da će skript biti ispravno pozivan, da će struktura izvorišnog direktorijuma biti ispravna, kao i da odredišni direktorijum ne postoji, ali postoje svi ostali koji su u hijerarhiji iznad.
Rešenje:

```
#!/bin/bash

mkdir $2
tmp=$PWD
cd $1

for i in *
do
  cd $tmp
  mkdir $2/$i
  cat $1/$i/h/*.h $1/$i/h/*.H > $2/$i/os.cpp
  cat $1/$i/cpp/*.cpp $1/$i/cpp/*.CPP >> $2/$i/os.cpp
done
```

59. (ispit 2011) Operativni sistem Linux

Napisati *bash shell script* koji treba da proveri da li je sadržaj nekog direktorijuma ispravan. Skript prihvata jedan ili dva parametra. Prvi parametar je naziv fajla koji opisuje potrebni sadržaj direktorijuma (spisak fajlova koji moraju postojati u direktorijumu koji se proverava). Ukoliko postoji i drugi parametar, tada je to putanja do direktorijuma koji treba proveriti. Ukoliko drugog parametra nema, proverava se sadržaj tekućeg direktorijuma. Da bi sadržaj direktorijuma bio korektan, u direktorijumu moraju postojati svi opisani fajlovi i u direktorijumu ne sme postojati nijedan fajl koji nije naveden na spisku fajlova. Smatrati da se u direktorijumu pojavljuju samo fajlovi, kao i da pri pozivanju i izvršavanju skripta neće doći do greške.

Rešenje:

```
#!/bin/bash
if [ $# -ne 1 -a $# -ne 2 ]
then
  echo "Greska: skript ocekuje 1 ili 2 parametra."
else
  if [ $# -eq 1 ]
  then
    DIREKTORIJUM=$PWD
  else
    DIREKTORIJUM=$2
  fi
  SPISAK=$(<$1)
  BROJ_NA_SPISKU=0
  for i in $SPISAK
  do
    let BROJ_NA_SPISKU++
    if [ ! -f $i ]
    then
      echo "Fajl $i se ne nalazi u direktorijumu $DIREKTORIJUM."
      exit 1
    fi
  done
  SADRZAJ=$(ls -a $DIREKTORIJUM)
  BROJ_FAJLOVA=0
  for i in $SADRZAJ
  do
    let BROJ_FAJLOVA++
  done
  let BROJ_FAJLOVA=BROJ_FAJLOVA-2 #zbog . i .. u ispisu ls -a
  if [ $BROJ_FAJLOVA -eq $BROJ_NA_SPISKU ]
  then
    echo "Sadrzaj direktorijuma $DIREKTORIJUM je ispravan."
    exit 0
  else
    echo "U direktorijumu $DIREKTORIJUM postoji vise fajlova nego sto je
potrebno."
    exit 2
  fi

fi
```

60. (ispit 2011) Operativni sistem Linux

a)(5) Kako se u Linux fajl sistemu VFS naziva struktura koja predstavlja FCB?

Odgovor: `inode`.

b)(5) Kako se u Linux fajl sistemu VFS naziva strkuktura koja predstavlja ulaz u tabeli otvorenih fajlova za dati proces?

Odgovor: `file`.

61. (ispit 2011) Operativni sistem Linux

Napisati program za operativni sistem Linux koji pri prvom pokretanju učita jedan broj sa standardnog ulaza, a zatim pri sledećem pokretanju ispiše prethodno učitani broj (broj koji je učitan prilikom prethodnog pokretanja). Učitani broj između uzastopnih pokretanja čuvati u memoriji (nije dozvoljeno koristiti bilo kakve dodatne objekte za čuvanje vrednosti). Nakon drugog pokretanja, program treba da vrati sistem u početno stanje, tako da se pri sledećem pokretanju ponaša kao da je pokrenut prvi put. Poznato je da sistem koristi čitavu radnu memoriju računara, kao i da se programu prilikom pokretanja prosleđuje jedinstven ključ (ceo broj) koji u sistemu ne koristi niko drugi.

Rešenje:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main (int argc, char *argv[])
{
  int segment_id;
  int* shared_memory;
  const int shared_segment_size = 4096;
  int key = atoi(argv[1]);
  segment_id = shmget (key, shared_segment_size, IPC_CREAT | IPC_EXCL |
S_IRUSR | S_IWUSR);
  if (segment_id == -1){ //drugo pokretanje
    segment_id = shmget (key, shared_segment_size, S_IRUSR | S_IWUSR);
    if (segment_id == -1) {
      printf("Greska - neuspesno dohvatanje id deljenog segmenta
memorije.\n");
      exit(1);
    }
    shared_memory = (int*) shmat (segment_id, 0, 0);
    printf("%d\n",*shared_memory);
    shmdt (shared_memory);
    shmctl (segment_id, IPC_RMID, 0);
  } else{ //prvo pokretanje
    shared_memory = (int*) shmat (segment_id, NULL, 0);
    scanf("%d", shared_memory);
    shmdt (shared_memory);
  }
  return 0;
}
```

62. (ispit 2011) Operativni sistem Linux

Napisati funkciju na programskom jeziku C za Linux za pretraživanje veoma velikih neuređenih nizova. Funkcija `void f(int N, int* niz, int M, int* podniz)` treba da u K paralelnih procesa pronađe sva pojavljivanja podniza u zadatom nizu. Dužina niza je N, a podniza M i važi da je N mnogo veće od M i K. Za svako pojavljivanje podniza na standardni izlaz ispisati indeks u nizu od kojeg počinje podniz. Ispisani indeksi treba da budu sortirani po rastućem redosledu. Za sinhronizaciju i komunikaciju koristiti neimenovane cevi.

Rešenje:

```
#define K ...

void f(int N, int* niz, int M, int* podniz){
      int ret[2], i, j, p, l, *indeksi;
      pipe(ret);
      for(i = 0; i < K; i++){
            if (fork()==0) {
                  for(j = i*((N-M+1)/K)+(i<(N-M+1)%K?i:(N-M+1)%K);
             j < (i+1)*((N-M+1)/K)+(i+1<(N-M+1)%K?i+1:(N-M+1)%K); j++){
                        p = 0;
                        while (p < M) {
                              if( niz[j+p] != podniz[p]) break;
                              p++;
                        }
                        if (p==M) write(ret[1], &j, sizeof(int));
                  }
                  j = N;
                  write(ret[1], &j, sizeof(int));
                  close(ret[0]);
                  close(ret[1]);
                  exit(0);
            }
      }
      indeksi = malloc(N*sizeof(int));
      j = K;
      indeksi[0] = -1;
      p = 1;
      while (j>0){
            read(ret[0],&i,sizeof(int));
            if (i==N) j--;
            else{
                  l = p;
                  while (l>0){
                        if (indeksi[--l] > i) indeksi[l+1] = indeksi[l];
                              else {
                                    indeksi[l+1] = i;
                                    break;
                              }
                  }
                  p++;
            }
      }
      for(j = 1; j < p; j++) printf("%d\n",indeksi[j]);
      free(indeksi);
}
```

63. (ispit 2011) Operativni sistem Linux

Na programskom jeziku C implementirati funkciju `put` koja u deljeni bafer stavlja po jedan ceo broj. Bafer se nalazi u deljenoj memoriji. Ključ za dohvatanje već postojećeg i inicijalizovanog segmenta deljene memorije se prosleđuje funkcijama kao prvi parametar. Drugi parametar je ceo broj koji treba umetnuti u deljeni bafer. Smatrati da su na početku bafera zapisani podaci neophodni za implementaciju bafera (uvesti pretpostavku o redosledu potrebnih podataka), nakon čega je smešten sam sadržaj bafera.

Rešenje:

```
typedef struct{
      int capacity;
      int freeSlot;
      int usedSlot;
      key_t sems;
} SBuffer;

void put(key_t key, int data){
  int segmentId, semsId;
  int* bufferData;
  SBuffer *buffer;
  struct sembuf semOp[2];
  segmentId = shmget (key, 0, S_IRUSR | S_IWUSR);
  if (segmentId == -1){
    //error ...
  }
  buffer = (Sbuffer *) shmat (segmentId, 0, 0);
  if (buffer == (Sbuffer *)-1){
    //error ...
  }
  bufferData = (int *) ( (char*)buffer + sizeof(SBuffer) );

  semsId = semget(buffer->sems, 0, S_IRUSR | S_IWUSR);
  if (semsId == -1){
    //error ...
  }

  semOp[0].sem_num = 0;
  semOp[0].sem_op = -1;
  semOp[0].sem_flg = SEM_UNDO;
  semOp[1].sem_num = 2;
  semOp[1].sem_op = -1;
  semOp[1].sem_flg = SEM_UNDO;
  if (semop(semsId, semOp, 2) == -1){
    //error ...
  }
  bufferData[buffer->freeSlot++] = data;
  if (buffer->freeSlot >= buffer->capacity)
    buffer->freeSlot = 0;
  semOp[0].sem_num = 0;
  semOp[0].sem_op = 1;
  semOp[0].sem_flg = SEM_UNDO;
  semOp[1].sem_num = 3;
  semOp[1].sem_op = 1;
  semOp[1].sem_flg = SEM_UNDO;
  if (semop(semsId, semOp, 2) == -1){
    //error ...
  }
  if (shmdt(buffer) == -1){
    //error ...
  }
}
```

64. (ispit 2011) Operativni sistem Linux

Napisati program koji u optimalnom broju procesa vrši sledeću obradu. Program prvo učita prva dva reda kvadratne matrice dimenzija NxN. Potom računa ostale elemente po sledećoj formuli m[i][j] = (*f[j])(m[i-2][j]). Poznato je da se u sekvencijalnom algoritmu računaju elementi redom s leva na desno, odozgo naniže. Poznato je da svaka od funkcija *f[i] pri računanju rezultata pored parametara koristi i svoj privatni statički podatak. Poznato je i da je trajanje izvršavanja svih funkcija dugo, kao i da je približno isto za sve funkcije. Za komunikaciju i sinhronizaciju koristiti neimenovane cevi.

Rešenje:

```
int pipe(int fd[2]);
int read(int fd, void *Buff, int NumBytes);
int write(int fd, void *Buff, int NumBytes);
int close(int fd);
#define N ...
int (*f[N])(int in)...;

typedef struct {
  int i,j;
  int val
} Item;


int fileDesc[100][2];
int return_pipe[2];

int a[N][N];

void main(){
  Item item;
//citanje prve dve vrste
//...

  pipe( return_pipe );

  for (int i = 0; i < n; ++i){
    if (fork() == 0){
     item.i = i;
     for (int j = 2; j < n; ++j){
         item.val = a[j][i] = (*f[i])(a[j-2][i]);
       item.j = j;
  write(return_pipe[1], &item, sizeof(Item));
      }
      close( return_pipe[0] );
      close( return_pipe[1] );

      exit(0);
    }
  }
  for (int i = 0; i < n*(n-2); ++i){
    read(return_pipe[0], &item, sizeof(Item));
    a[item.j][item.i] = item.val;
  }
  close( return_pipe[0] );
  close( return_pipe[1] );
}
```

65. (ispit 2011) Operativni sistem Linux

Posmatra se sledeći problem. Potrebno je učitati prva dva reda kvadratne matrice dimenzija NxN, a zatim izračunati ostale elemente matrice po sledećoj formuli m[i][j] = (*f[j])(m[i-2][j]). Poznato je da se u sekvencijalnom algoritmu elementi računaju s leva na desno, odozgo naniže. Poznato je da svaka od funkcija *f[i] pri računanju rezultata pored parametara koristi i svoj privatni statički podatak. Jedan programer je predložio sledeće paralelizovano rešenje ovog problema.

```
#define N ...
int (*f[N])(int in)...;

typedef struct {
  int i,j;
  int val;
} Item;

int return_pipe[2];

int a[N][N];

void main(){
  Item item;
//citanje prve dve vrste
//...

  pipe( return_pipe );

  for (int i = 0; i < n; ++i)
    for (int k = 0; k < 2; k++){
      if (fork() == 0){
       item.i = i;
       for (int j = k+2; j < n; j += 2){
          item.val = a[j][i] = (*f[i])(a[j-2][i]);
         item.j = j;
   write(return_pipe[1], &item, sizeof(Item));
        }
        close( return_pipe[0] );
        close( return_pipe[1] );

        exit(0);
      }
    }

  for (int i = 0; i < n*(n-2); ++i){
    read(return_pipe[0], &item, sizeof(Item));
    a[item.j][item.i] = item.val;
  }
  close( return_pipe[0] );
  close( return_pipe[1] );
}
```

Međutim, programer je bio nepažljiv i napravio je ozbiljan previd. Kratko i precizno objasniti šta je previd? Nije dovoljno navesti mesto u kodu na kojem je učinjen previd, već objasniti zašto je to pogrešno.

Rešenje:

Problem je pozivanje opisanih funkcija iz različitih procesa. Kako je u opisu funkcija rečeno da koriste statički podatak, a u svakom adresnom prostoru postoji zasebna kopija tog statičkog podatka, paralelizovana verzija programa neće izgenerisati korektan sadržaj matrice. Razlog tome je što su se u originalnom algoritmu elementi u kolonama izračunavali redom, pa

je za računanje i-tog elementa u koloni korišćen statički podatak nakon računanja elementa i-1, dok se u paralelizovanoj verziji i-ti element računa nakon računanja elementa i-2. Elementi i i i-1 se u ovoj implementaciji računaju u različitim procesima, i zato nije poznat redosled izračunavanja ova dva elementa. Čak i da je poznat redosled izračunavanja, to ne bi ništa značilo jer zbog pozivanja iz dva različita procesa, pri izračunavanju se koriste dva različita statička podatka.

66. (ispit 2011) Operativni sistem Linux

Na programskom jeziku C implementirati funkciju `get` koja iz deljenog bafera uzima po jedan ceo broj. Bafer se nalazi u deljenoj memoriji. Ključ za dohvatanje već postojećeg i inicijalizovanog segmenta deljene memorije se prosleđuje funkciji kao prvi parametar. Smatrati da su na početku bafera zapisani podaci neophodni za implementaciju bafera (uvesti pretpostavku o redosledu potrebnih podataka), nakon čega je smešten sam sadržaj bafera.

Rešenje:

```
include <sys/types.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/sem.h>

typedef struct{
      int capacity;  //kapacitet bafera
      int freeSlot;  //prvo slobodno mesto
      int usedSlot;  //prvo zauzeto mesto
      key_t sems;    //kljuc za dohvatanje niza
} SBuffer;

int get(key_t key){
      int segmentId, semsId, data;
      int* bufferData;
      SBuffer *buffer;
      struct sembuf semOp[2];

      //dohvatanje ID-a deljenog segmenta u kojem je bafer
      segmentId = shmget (key, 0, S_IRUSR | S_IWUSR);

      //mapiranje segmenta deljene memorije u adresni prostor pozivajuceg
proc.
      buffer = (SBuffer*) shmat (segmentId, 0, 0);

      //racunanje adrese pocetka podataka u baferu (na pocetku je struktura
      //sa potrebnim semaforima, kapacitetom i pokazivacima na prvo
slobodno
      //i prvo zauzeto mmesto u kruznom baferu
      bufferData = (int *) ( (char*)buffer + sizeof(SBuffer) );

      semsId = semget(buffer->sems, 0, S_IRUSR | S_IWUSR);

      //wait(mutexGet, itemAvailable);
      semOp[0].sem_num = 1;
      semOp[0].sem_op = -1;
      semOp[0].sem_flg = SEM_UNDO;
      semOp[1].sem_num = 3;
      semOp[1].sem_op = -1;
      semOp[1].sem_flg = SEM_UNDO;
      semop(semsId, semOp, 2);

      //uzimanje elementa iz bafera
      data = bufferData[buffer->usedSlot++];
      if (buffer->usedSlot >= buffer->capacity)
            buffer->usedSlot = 0;

      //signal(mutexGet, spaceAvailable);
      semOp[0].sem_num = 1;
      semOp[0].sem_op = 1;
```

```
        semOp[0].sem_flg = SEM_UNDO;
        semOp[1].sem_num = 2;
        semOp[1].sem_op = 1;
        semOp[1].sem_flg = SEM_UNDO;
        semop(semsId, semOp, 2);

        //izvezivanje bafera iz adresnog prostora
        shmdt(buffer);
        return data;
}
```

1. (Januar 2013) Operativni sistem Windows

U nastavku je data ilustracija upotrebe objekta *critical section* iz dokumentacije za Win32 API. Na jeziku C++ implementirati klasu `Mutex` koja obezbeÿuje objektno orijentisani „omotač“ oko ovih sistemskih poziva i realizuje apstrakciju binarnog semafora za meÿusobno
isključenje niti.

```
// Global variable
CRITICAL_SECTION CriticalSection;

int main( void )
{


    // Initialize the critical section one time only.
    if
    (!InitializeCriticalSectionAndSpinCount(&CriticalSection,
        0x00000400) )
        return;


    //  Release  resources  used  by  the  critical  section
    object. DeleteCriticalSection(&CriticalSection);
}

DWORD WINAPI ThreadProc( LPVOID lpParameter
)
{


    //  Request  ownership  of  the  critical
    section.
    EnterCriticalSection(&CriticalSection);

    // Access the shared resource.

    //  Release  ownership  of  the  critical
    section.
    LeaveCriticalSection(&CriticalSection);


return 1;
}
```

Rešenje:

```
class Mutex {
public:
  Mutex ()
    { InitializeCriticalSectionAndSpinCount(&criticalSection,0x00000400); }
 ~Mutex()
    { DeleteCriticalSection(&criticalSection); }
  void enter ()
    { EnterCriticalSection(&criticalSection); }
  void exit ()
    { LeaveCriticalSection(&criticalSection); }
private:
  CRITICAL_SECTION criticalSection;
};
```