Оперативни системи 1 (ИР2ОС1)

Решени задаци са претходних колоквијума

верзија документа: 1.03.rev0 (23.08.2012)

Садржај:

сва питања на: zb100069d@student.etf.rs ili zukicbojan@gmail.com

Прва група задатака: улаз-излаз

1. (*Mart 2012*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva izlazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;          // Device 1 control register
REG* io1Status =...;        // Device 1 status register
REG* io1Data =...;          // Device 1 data register
REG* io2Ctrl =...;          // Device 2 control register
REG* io2Status =...;        // Device 2 status register
REG* io2Data =...;          // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati kod koji vrši prenos po jednog bloka podataka na svaki od dva uređaja uporedo, na prvi uređaj tehnikom prozivanja (*polling*), a na drugi korišćenjem prekida. Transfer se obavlja pozivom sledeće funkcije iz koje se vraća kada su oba prenosa završena:

```
void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2);
```

Rešenje:

```
static unsigned* io2Ptr = 0;        //pokazivač na sledeći podatak za uređaj 2
static int io2Count = 0;            //broj podataka za prenos uređaju 2
static int io2Completed = 0;        //volatile???

void transfer(unsigned* blk1, int count1, unsigned* blk2, int count2) {
    // I/O 2:
    io2Ptr = blk2;                  //io2Ptr - početna adresa drugog bloka za prenos
    io2Count = count2;              //io2Count - broj podataka u 2. bloku za prenos
    io2Completed = 0;               //prenos još nije završen
    *io2Ctrl = 1;                   //startujemo uređaj 2

    // I/O 1
    *io1Ctrl = 1;                   //startujemo uređaj 1
    while (count1>0) {              //dok god nismo preneli sve podatke
       while (!(*io1Status & 1));   //čekamo dok uređaj ne postane spreman
      *io1Data = *blk1++;           //šaljemo mu naredni podatak
      count1--;                     //smanjujemo broj podataka za prenos za 1
    }
    *io1Ctrl = 0; // Stop I/O 1    //preneli smo čitav blok - zaustavljamo uređaj

   while (!io2Completed);           //čekamo dok drugi uređaj ne završi prenos
}

interrupt void io2Interrupt() {
     *io2Data = *io2Ptr++;          //šaljemo uređaju 2 naredni podatak

     if (--io2Count == 0) {         //ako smo preneli čitav blok, onda
         io2Completed = 1;          //postavljamo "fleg" završetka prenosa
         *io2Ctrl = 0;              //zaustavljamo uređaj 2
     }
}
```

2. (*April 2011*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva izlazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Biti spremnosti statusnih registara oba uređaja vezani su preko logičkog ILI kola na isti signal zahteva za prekid, tako da se isti prekid generiše ako je bilo koji (ili oba) uređaja postavio svoj bit spremnosti. Potrebno je napisati kod koji može da izvrši prenos datog bloka podataka korišćenjem oba uređaja uporedo, tako da se na svaki uređaj prenese po pola datog bloka (ili približno pola, ako je broj reči neparan). Ovaj prenos pokreće se pozivom dole date funkcije `transfer()`. Napisati kod prekidne rutine `ioInterrupt()`.

```
int flag1 = 0, flag2 = 0; // I/O completed
unsigned int index1, count1;
unsigned int index2, count2;
REG* buf1;
REG* buf2;

void transfer (REG* buffer, unsigned int count) {
  count1 = count/2;
  count2 = count1+count%2;
  buf1 = buffer;
  buf2 = buffer+count1;
  index1 = index2 = 0;
  flag1 = flag2 = 0;
  *io1Ctrl = 1;
  *io2Ctrl = 1;
}

int isIOCompleted () {
  return flag1 && flag2;
}

interrupt void ioInterrupt();
```

Rešenje:

```
interrupt void ioInterrupt() {
      if (*io1Status & 1) {              //ako je uređaj 1 spreman
            *io1Data = buf1[index1++];   //šaljemo mu naredni podatak
            if (--count1 == 0) {         //ako smo mu poslali (približno) pola bloka
                  flag1 = 1;             //ažuriramo "fleg" završetka prenosa
                  *io1Ctrl = 0;          //i zaustavljamo uređaj 1
            }
      }
      if (*io2Status  & 1) {             //ako je uređaj 2 spreman
            *io2Data = buf2[index2++];   //šaljemo mu naredni podatak
            if (--count2 == 0) {         //ako smo mu poslali (približno) pola bloka
                  flag2 = 1;             //ažuriramo "fleg" završetka prenosa
                  *io2Ctrl = 0;          //i zaustavljamo uređaj 2
            }
      }
```

```
}
```

3. (*Maj 2011*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati kod funkcije `transfer()` koji vrši prenos datog bloka podataka korišćenjem oba uređaja uporedo, tako da se sa svakog uređaja prenese po pola datog bloka (ili približno pola, ako je broj reči neparan), programiranim ulazom/izlazom sa prozivanjem (*polling*), uz maksimalni paralelizam rada udređaja i prenosa delova bloka.

```
void transfer(REG* buffer, unsigned int count);
```

Rešenje:

```
void transfer (REG* buffer, unsigned int count) {
      unsigned int index1, count1;
      unsigned int index2, count2;
      REG* buf1;
      REG* buf2;

      count1 = count/2;                          //dajemo približno po pola bloka
      count2 = count1+count%2;                   //svakom uređaju za prenos
      buf1 = buffer;                             //prvi podatak za uređaj 1
      buf2 = buffer+count1;                      //prvi podatak za uređaj 2
      index1 = index2 = 0;
      *io1Ctrl = 1;
      *io1Ctrl = 1;

      while ((count1!=0) || (count2!=0)) {       //dok god ima podataka za prenos
            if ((count1!=0) && (*io1Status&1)){  //ako nismo primili sve podatke
                                                 //od uređaja 1 i ako je
                                                 //uređaj 1 spreman
                  buf1[index1++] = *io1Data;     //prihvatamo naredni podatak

                  if (--count1 == 0)             //ako smo primili sve podatke
                        *io1Ctrl = 0;            //zaustavljamo uređaj 1
            }
            if ((count2!=0) && *(*io2Status&1)){  //ako nismo primili sve podatke
                                                  //od uređaja 2 i ako je
                                                  //uređaj 2 spreman
                  buf2[index2++] = *io2Data;      //prihvatamo naredni podatak

                  if (--count2 == 0)              //ako smo primili sve podatke
                        *io2Ctrl = 0;             //zaustavljamo uređaj 2
            }
      }
}
```

4. (*Septembar 2011*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva uređaja; prvi uređaj je ulazni, drugi izlazni:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati program koji učitava po jednu reč sa ulaznog uređaja i odmah tu učitanu reč šalje na izlazni uređaj. Ovaj prenos se ponavlja sve dok se sa ulaznog uređaja ne učita vrednost 0. Oba prenosa vršiti programiranim ulazom/izlazom za prozivanjem (*polling*).

Rešenje:

```
const REG ESC = 0;                          //završavamo prenos kad učitamo nulu

void main () {
     *io1Ctrl = 1;                          //startujemo uređaj 1
     *io2Ctrl = 1;                          //startujemo uređaj 2

     while (1) {                            //vrtimo se u petlji
          while ((*io1Status & 1) == 0);    //čekamo da uređaj 1 postane spreman
                                            //kada postane spreman
          REG data = *io1Data;              //prihvatamo reč koju je poslao uređaj 1

          if (data == ESC)                  //ako je uređaj 1 poslao nulu
               break;                       //završavamo prenos (izlazimo iz petlje)
                                            //u suprotnom
          while ((*io2Status & 1) == 0);    //čekamo da uređaj 2 postane spreman
                                            //kada postane spreman
          *io2Data = data;            //šaljemo mu podatak dobijen od uređaja 1
     }

     *io1Ctrl = 0;                          //zaustavljamo uređaj 1
     *io2Ctrl = 0;                          //zaustavljamo uređaj 2
}
```

5. (*Mart 2010*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva DMA kontrolera:

```
typedef unsigned int REG;
REG* dma1Ctrl =...;     // DMA1 control register
REG* dma1Status =...;   // DMA1 status register
REG* dma1Addr =...;     // DMA1 block address register
REG* dma1Count =...;    // DMA1 block size register
REG* dma2Ctrl =...;     // DMA2 control register
REG* dma2Status =...;   // DMA2 status register
REG* dma2Addr =...;     // DMA2 block address register
REG* dma2Count =...;    // DMA2 block size register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće DMA kontroler, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je DMA kontroler završio prenos. Svaki DMA kontroler upravlja svojom izlaznom periferijom sa kojom direktno komunicira, tako da procesor ne treba (i ne može) direktno da pristupa kontrolerima periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Jezgro operativnog sistema postavlja zahteve za prenos blokova podataka iz memorije na bilo koju od dve periferije uvezivanjem sledećih struktura sa zahtevima u listu:

```
struct DMAReq {
  DMAReq* next;         // Next request in the list
  unsigned int addr;    // Block address
  unsigned int size;    // Block size
};
DMAReq *head, *tail;
```

Na jeziku C napisati potprogram koji uzima jedan po jedan postavljeni zahtev iz liste i zadaje ih bilo kom slobodnom DMA kontroleru. Čekanje da se pojavi prvi zahtev u praznoj listi i da se oslobodi DMA kontroler treba obavljati uposleno, odnosno prozivanjem (*busy waiting*, *polling*).

Rešenje:

```
void DMADriver() {
     while (1){                              //vrtimo se u beskonačnoj petlji
          DMAReq* req;                       //pomoćni pokazivač na strukturu
          while (head == 0);                 //sve dok je lista zahteva prazna
                                             //čekamo da pristigne bar 1 zahtev
          req = head;                        //uzimamo prvi zahtev
          head = head->next;                 //pomeramo se na sledeći u listi
          if (head==0) tail = 0;             //ako nema više zahteva u listi
                                             //ažuriraj pokazivač na kraj
          req->next = 0;                     //raskidamo vezu preuzetog zahteva
                                             //sa ostatkom liste
          //čekamo da bar jedan kontroler postane spreman
          while (!(*dma1Status & 1) && !(*dma2Status & 1));

          if (*dma1Status & 1) {             //ako je kontroler 1 spreman
               *dma1Addr = req->addr;        //šaljemo adresu bloka kontroleru 1
               *dma1Count = req->size;       //šaljemo broj podataka za prenos
               *dma1Ctrl = 1;                //startujemo kontroler 1
          } else                             //u suprotnom, blok šaljemo kontroleru 2
          if (*dma2Status & 1) {             //ako je on spreman
               *dma2Addr = req->addr;        //šaljemo adresu bloka kontroleru 2
               *dma2Count = req->size;       //šaljemo broj podataka za prenos
               *dma2Ctrl = 1;                //startujemo kontroler 2
          }
          free(req);                         //oslobađamo prostor koji je zauzela
      }                                      //struktura zahteva koji smo obradili
```

```
}
```

6. (*Mart 2009*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima nekih periferijskih uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* ioCtrl1 =...;            // PER1 control register
REG* ioStatus1 =...;          // PER1 status register
REG* ioData1 =...;            // PER1 data register
REG* ioCtrl2 =...;            // PER2 control register
REG* ioStatus2 =...;          // PER2 status register
REG* ioData2 =...;            // PER2 data register
REG* ioCtrl3 =...;            // PER3 control register
REG* ioStatus3 =...;          // PER3 status register
REG* ioData3 =...;            // PER3 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke. PER1 i PER2 su ulazne, a PER3 je izlazna periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Date su deklaracije bloka podataka koji treba prenositi:

```
const int N = ...;
unsigned int buf[N];
```

Na jeziku C napisati program koji uporedo sa PER1 i PER2 učitava blok od ukupno N reči, i na PER3 upisuje taj isti blok od N reči, korišćenjem mehanizma prozivanja (engl. *polling*), bez nametanja naizmeničnosti prenosa po jednog podatka (npr. da se uvek jedan podatak prebaci sa PER1, pa onda sa PER2 i tako naizmenično), već podatak sa datom periferijom treba preneti kad god je ona spremna, ne čekajući ostale. Broj podataka koji se čitaju sa pojedinačnih periferija PER1 i PER2 nije unapred poznat, već zavisi od brzine rada periferija.

Rešenje:

```
void main () {
      //indeks prve slobodne lokacije u baferu za upis
      //i indeks prvog raspoloživog podatka za slanje
      int i12 = 0, i3 = 0;

      //startujemo sve tri periferije
      *ioCtrl1 = 1; *ioCtrl2 = 1; *ioCtrl3 = 1;

      //dok god izlazna periferija nije preuzela svih N podataka
      while (i3 < N) {
            //ako bafer nije pun i ako je periferija 1 spremna,
            //upisujemo u bafer podatak procitan sa periferije 1
            if (i12<N && (*ioStatus1 & 1))
                  buf[i12++]=*ioData1;

            //ako bafer nije pun i ako je periferija 2 spremna,
            //upisujemo u bafer podatak procitan sa periferije 2
            if (i12<N && (*ioStatus2 & 1))
                  buf[i12++]=*ioData2;

            //ako u baferu postoji neposlati podatak i ako je periferija 3
            //spremna, šaljemo periferiji 3 naredno podatak iz bafera
            if (i3<i12 && (*ioStatus3&1))
                  *ioData3=buf[i3++];
      }
      //zaustavljamo sve tri periferije
      *ioCtrl1=0; *ioCtrl2=0;  *ioCtrl3=0;
```

```
}
```

7. (*April 2008*) Date su deklaracije pokazivača preko kojih se može pristupiti registrima nekih periferijskih uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* ioCtrl1 =...;          // PER1 control register
REG* ioStatus1 =...;        // PER1 status register
REG* ioData1 =...;          // PER1 data register
REG* ioCtrl2 =...;          // PER2 control register
REG* ioStatus2 =...;        // PER2 status register
REG* ioData2 =...;          // PER2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke. PER1 je ulazna, a PER2 izlazna periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Date su deklaracije dva bloka podataka koje treba preneti:

```
const int N = ...;
unsigned int buf1[N], buf2[N];
```

Na jeziku C napisati program koji uporedo sa PER1 učitava blok od N reči, a na PER2 upisuje blok od N reči, korišćenjem mehanizma prozivanja (engl. *polling*), bez nametanja naizmeničnosti prenosa po jednog podatka (npr. da se uvek jedan podatak prebaci sa PER1, pa onda sa PER2 i tako naizmenično), već podatak sa datom periferijom treba preneti kad god je ona spremna, ne čekajući na onu drugu.

Rešenje:

```
void main () {
      //indeks prve slobodne lokacije u prvom baferu za upis
      //i indeks prvog raspoloživog podatka za slanje u drugom baferu
      int i1 = 0, i2 = 0;

      //startujemo obe periferije
      *ioCtrl1 = 1; *ioCtrl2 = 1 ;

      //dok god nismo upisali N reči u prvi bafer ili
      //dok god nismo preneli N reči iz drugog bafera
      while (i1<N || i2<N) {
            //ako prvi bafer nije "pun" i ako je periferija 1 spremna,
            //upisujemo podatak sa periferije 1 u prvi bafer
            if (i1<N && (*ioStatus1 & 1))
                  buf1[i1++] = *ioData1;

            //ako drugi bafer nije "prazan" i ako je periferija 2 spremna,
            //šaljemo podatak iz drugog bafera periferiji 2
            if (i2<N && (*ioStatus2 & 1))
                  *ioData2 = buf2[i2++];
      }

      //zaustavljamo obe periferije
      *ioCtrl1 = 0; *ioCtrl2 = 0;
}
```

8. (*April 2007*) Predložiti interfejs *DMACtrl* i specifikovati ga apstraktnom klasom na jeziku *C++*. Ovaj interfejs treba da delovima operativnog sistema koji upravljaju uređajima obezbedi uniforman pristup različitim vrstama *DMA* kontrolera, tako što im na isti način zadaje operacije prenosa bloka podataka iz ili u memoriju, kao i način obaveštavanja o završenom prenosu i eventualnoj grešci. Obaveštavanje o završenoj operaciji može da bude prozivanjem (*polling*) ili prekidom sa zadatim ulazom.

Rešenje:

```
enum Status{OK, Error};

class DMA {
      public:
      //ulazni parametri: adresa bloka u memoriji, adresa registra podataka //
      uređaja, adresa statusnog registra, maska za statusni registar, smjer //
      operacije i velicina bloka
      virtual void start(void* mem, void* devData, void* devStatus, int
                          StatusMask, int memToIO, int vel);

      //rezultat: logicka vrijednost koja pokazuje da li je operacija zavrsena
      virtual int is_finished();

      //zadaje kontroleru broj prekidne rutine preko koje traba da obavjesti //
      sistem o kraju operacije
      virtual void setIntNo(int interruptNo);

      //vraca status poslednje zavrsene operacije
      virtual Status getStatus();
}
```

9. (*April 2006*) Na nekom računaru sa multiprogramskim operativnim sistemom neki kontroler periferije nema vezan svoj signal završetka operacije na ulaze za prekid procesora, ali je poznato vreme završetka operacije zadate tom uređaju u najgorem slučaju (maksimalno vreme završetka operacije). Za obradu ulazno/izlaznih operacija zadatih tom uređaju brine se poseban proces tog operativnog sistema. Kakvu uslugu treba operativni sistem da obezbedi, a ovaj proces da koristi, da bi se izbegla tehnika kojom se može detektovati završetak operacije čitanjem statusnog registra i ispitivanjem bita završetka operacije (*polling*), tj. da bi se izbeglo uposleno čekanje (*busy waiting*)? Precizno opisati kako ovaj proces treba da koristi ovu uslugu.

Rešenje:

Operativni sistem treba da obezbedi uslugu uspavljivanja procesa na određeno zadato vreme (na primer, da ga stavi u red uspavanih procesa). Nakon isteka zadatog vremena, proces se "budi" tako što se uklanja iz reda uspavanih procesa i stavlja u red spremnih.

Prototip metode koja se koristi za uspavljivanje tekućeg procesa može da izgleda:

```
void sleep(int time);
```

pri čemu je parametar `time` vremenski interval koji proces treba da provede u "uspavanom stanju".

Proces koji se brine o obradi ulazno-izlaznih operacija treba najpre da izvrši inicijalizaciju operacije (pripremi potrebne podatke, startuje ulazno-izlazni uređaj...), zatim da se uspava na vremenski interval potreban za izvršenje zadate operacije u najgorem slučaju (ili nešto više) i da se na kraju probudi i izvrši zahtevanu obradu nad rezultatom zadate operacije.

Pseudo kod izvršavanja prethodnih koraka je dat sa:

```
const int maxTime = ...; //vreme potrebno za I/O operaciju u najgorem slučaju

while (!end) {
      initialize_IO_operation();       //inicijalizacija I/O operacije
      sleep(maxTime);                  //uspavljivanje procesa
      process_IO_result();             //obrada rezultata operacije
}
```

Друга група задатака: виртуелна меморија

1. (*Mart 2012*) Neki računar podržava straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Stranica je veličine 1KB. U deskriptoru stranice u jednom ulazu u tabeli preslikavanja stranica (PMT) najviši bit ukazuje na to da li je stranica u memoriji ili ne, a najniži biti predstavljaju broj okvira u fizičkoj memoriji u koji se stranica preslikava. Deskriptor stranice ne sadrži druge informacije.

a. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja. Ako je početak PMT nekog procesa na fizičkoj adresi FF00h, na kojoj adresi je deskriptor stranice ABCh tog procesa?
b. Na jeziku C napisati kod funkcije:
`void setPageDescr(unsigned* pmtp, unsigned page, unsigned frame);`
koja u tabelu na čiji početak ukazuje dati pokazivač pmtp, za stranicu sa datim brojem page, upisuje deskriptor tako da se ta stranica preslikava u okvir sa datim brojem frame.

Rešenje:

a. Veličina virtuelnog adresnog prostora je: $VAS = 16\ GB = 2^{34}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 1\ GB = 2^{30}\ B$
Veličina adresibilne jedinice je: $AU = 32\ bit = 2^{2}\ B$
Veličina stranice (okvira) je: $PAGE = 1\ KB = 2^{10}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 32\ bita$
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 28\ bita$

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(PAGE / AU) = 8\ bita$

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj stranice u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, pa je za predstavljanje broja stranice potrebno:

$$VA\_S - OFFSET\_S = 32\ bita - 8\ bita = 24\ bita$$

Dakle, logička struktura virtuelne adrese je: VA(32) = *Page*(24):*Offset*(8) .

Fizička adresa se sastoji od određenog broja bita koji određuju broj okvira u fizičkom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar tog okvira. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, pa je za predstavljanje broja okvira potrebno:

$$PA\_S - OFFSET\_S = 28\ bita - 8\ bita = 20\ bita$$

Dakle, logička struktura fizičke adrese je: PA(28) = *Frame*(20):*Offset*(8) .

Desktriptor stranice ABCh nalazi se na adresi: PMT + ABCh = FF00h + ABCh = 109BCh

b.
```
void setPageDescr(unsigned* pmtp, unsigned page, unsigned frame) {
      //adresa na koju treba upisati deskriptor je: pmtp + page ⇔ pmtp[page]
      //objašnjenje maske: prvo se komplementiranjem nule dobija maska čiji su svi bitovi 1
      //deljenjem maske sa dva, u najviši bit se upisuje 0 (ostali ostaju 1)
      //komplementiranjem nove maske dobijamo masku čiji je najviši bit 1, a svi ostali 0
      //čime obezbeđujemo da najviši bit desktriptora bude 1 (jer je stranica učitana)
```

```
        pmtp[page] = frame | ~((unsigned int)~0 / 2);
}
```

2. (*April 2011*) Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelna adresa je 32-bitna, stranica je veličine je 64KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 4GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u ulazu u PMT označava da data stranica nije učitana u fizičku memoriju (nijedna stranica korisničkog procesa ne preslikava se u okvir 0 gde se inače nalazi interapt vektor tabela). Trenutno stanje PMT dva posmatrana procesa A i B je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	0	0	0	14	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	2314	01AD	0	22	01AE	0	0

Prikazati sadržaj ovih tabela nakon što je operativni sistem završio sve sledeće akcije tim redom:

- obradio straničnu grešku (*page fault*) koju je generisao proces A kada je adresirao adresu 30203h u svom adresnom prostoru, tako što je izbacio stranicu broj 1 procesa B i na njeno mesto učitao traženu stranicu procesa A
- obradio straničnu grešku (*page fault*) koju je generisao proces B kada je adresirao adresu 1F00Fh u svom adresnom prostoru, tako što je izbacio stranicu broj FFh procesa A i na njeno mesto učitao traženu stranicu procesa B
- obradio sistemski poziv procesa A kojim je taj proces zahtevao deljenje svoje stranice broj 2 sa stranicom broj 1 procesa B.

Rešenje:

Virtuelna adresa je širine 32 bita, od čega je za predstavljanje pomeraja u okviru stranice odvojeno:

$$OFFSET\_S = \log_2(PAGE / AU) = \log_2(64KB / 1B) = 16 \text{ bita}$$

Dakle, poslednje 4 cifre heksadekadne adrese predstavljaju pomeraj, a prve 4 (ako nisu izostavljene vodeće nule) predstavljaju broj stranice (tj. ulaza u PMT).

Prva akcija:

adresirana adresa: 30203h ⇒ ulaz 3 u PMT

proces A pristupa ulazu 3 u svojoj PMT, ali stranica sa tim brojem se ne nalazi u memoriji, pa se:

1. *generiše Page fault*
2. *stranica se dovlači sa HDD-a (adresa stranice se nalazi u deskriptoru sa ulaza 3)*
3. *stranica se upisuje u okvir 12 (izbačena stranica procesa B sa ulaza 1 je zauzimala okvir 12)*
4. *ažuriraju se odgovarajući ulazi u obe PMT (ulaz 3 tabele procesa A i ulaz 1 tabele procesa B)*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	12	0	0	14	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	0	2314	01AD	0	22	01AE	0	0

Druga akcija:
adresirana adresa: 1F00Fh ⇒ ulaz 1 u PMT
proces B pristupa ulazu 1 u svojoj PMT, ali stranica sa tim brojem se ne nalazi u memoriji, pa se:

1. *generiše Page fault*
2. *stranica se dovlači sa HDD-a (adresa stranice se nalazi u deskriptoru sa ulaza 1)*
3. *stranica se upisuje u okvir 14 (izbačena stranica procesa A sa ulaza FF je zauzimala okvir 14)*
4. *ažuriraju se odgovarajući ulazi u obe PMT (ulaz FF tabele procesa A i ulaz 1 tabele procesa B)*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	12	0	0	0	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	14	2314	01AD	0	22	01AE	0	0

Treća akcija:

- *proces A zahteva deljenje svoje stranice broj 2 (koja se u tom trenutku ne nalazi u memoriji) sa stranicom broj 1 procesa B (stranica se nalazi u memoriji i zauzima okvir 14);*
- *potrebno je da se u PMT-u procesa A ažurira vrednost sa ulaza 2, tako da ta vrednost bude jednaka vrednosti sa ulaza 1 u PMT-u procesa B;*
- *prethodno ima za posledicu da proces A sada može da pristupa stranici procesa B koja se nalazi u okviru broj 14;*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	14	12	0	0	0	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	14	2314	01AD	0	22	01AE	0	0

napomena: ažurirane vrednosti nakon svake akcije su u tabelama prikazane crvenom bojom

3. (*Maj 2011*) Virtuelna memorija nekog računara organizovana je straničnо. Veličina virtuelnog adresnog prostora je 2 MB, adresibilna jedinica je 16-bitna reč, a veličina stranice je 128 KB. Veličina fizičkog adresnog prostora je 32 MB. Operativni sistem učitava stranice na zahtev, tako što se stranica učitava u prvi slobodni okvir fizičke memorije kada joj se pristupi. Kada se kreira proces, nijedna njegova stranica se ne učitava odmah, već tek kad joj se prvi put pristupi. U početnom trenutku, slobodni okviri fizičke memorije su okviri počev od 10h zaključno sa 1Fh. Neki proces generiše sledeću sekvencu virtuelnih adresa tokom svog izvršavanja (sve vrednosti su heksadecimalne):

30F00, 30F02, 30F04, 922F0, 922F2, 322F0, 322F2, 322F4, 522F0, 522F2, 402F0, 402F2

Prikazati izgled cele tabele preslikavanja stranica (*PMT*) za ovaj proces posle izvršavanja ove sekvence. Za svaki ulaz u *PMT* prikazati indikator prisutnosti stranice u fizičkoj memoriji (0 ili 1) i broj okvira u fizičkoj memoriji u koji se stranica preslikava, ukoliko je stranica učitana; ukoliko nije, prikazati samo ovaj indikator.

Rešenje:

Virtuelna adresa je širine:

$$VA\_S = \log_2(VAS / AU) = \log_2(2MB / 16bit) = 20 \text{ bita}$$

od čega je za predstavljanje pomeraja (*Offset*) u okviru stranice odvojeno:

$$OFFSET\_S = \log_2(PAGE / AU) = \log_2(128KB / 16bit) = 16 \text{ bita}$$

dakle, za predstavljanje broja stranice je preostalo 4 bita, pa je maksimalan broj stranice *Fh:*

$$VA = Page(4):Offset(16)$$

Najniže 4 cifre heksadekadne virtuelne adrese prestavljaju pomeraj, pa za adresu formata *XYYYYh*, cifra X predstavlja broj stranice.

Nijedna stranica procesa se na početku ne nalazi u memoriji, tako da je svaki put kada pristupamo stranici koja se ne nalazi u memoriji potrebno uraditi sledeće:

- dovući stranicu sa HDD-a u memoriju
- ažurirati odgovarajući broj ulaza u *PMT*

Ukoliko se stranica već nalazi u memoriji (zauzima neki okvir), nije potrebno ništa uraditi, jer je stranici već moguće pristupiti preko vrednosti okvira fizičke memorije u koji je smeštena.

Početni izgled PMT-a:

Entry	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Flag	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Frame																

Pošto se na početku nijedna stranica procesa ne nalazi u memoriji, svi indikatori prisutnosti sranice u fizičkoj memoriji su postavljeni na 0.

Za svaku gorenavedenu generisanu virtuelnu adresu je potrebno uraditi sledeće:

1. naći broj stranice u okviru koje se nalazi generisana adresa (najviša heksadekadna cifra adrese)
2. ako se stranica ne nalazi u memoriji, potrebno ju je učitati u prvi slobodni okvir (na početku, prvi slobodni okvir je onaj sa adresom 10h) i ažurirati polja "Flag" i "Frame" na odgovarajućem ulazu
3. ako se stranica već nalazi u memoriji, prelazimo na sledeću (ne diramo vrednosti u PMT-u)

Sekvenca virtuelnih adresa:

1. 30F00h ⇒ ulaz 3 je prazan ⇒ dovlačimo stranicu broj 3 i smeštamo je u okvir broj 10h
2. 30F02h ⇒ stranica 3 je već učitana u memoriju, pa ne radimo ništa
3. 30F04h ⇒ stranica 3 je već učitana u memoriju, pa ne radimo ništa
4. 922F0h ⇒ ulaz 9 je prazan ⇒ dovlačimo stranicu broj 9 i smeštamo je u okvir broj 11h
5. 922F2h ⇒ stranica 9 je već učitana u memoriju, pa ne radimo ništa
6. 322F0h ⇒ stranica 3 je već učitana u memoriju, pa ne radimo ništa
7. 322F2h ⇒ stranica 3 je već učitana u memoriju, pa ne radimo ništa
8. 322F4h ⇒ stranica 3 je već učitana u memoriju, pa ne radimo ništa
9. 522F0h ⇒ ulaz 5 je prazan ⇒ dovlačimo stranicu broj 5 i smeštamo je u okvir broj 12h
10. 522F2h ⇒ stranica 5 je već učitana u memoriju, pa ne radimo ništa
11. 402F0h ⇒ ulaz 4 je prazan ⇒ dovlačimo stranicu broj 4 i smeštamo je u okvir broj 13h
12. 402F2h ⇒ stranica 4 je već učitana u memoriju, pa ne radimo ništa

Konačan izgled PMT-a za dati proces:

Entry	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Flag	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0
Frame				10h	13h	12h				11h						

4. (*Septembar 2011*) Neki sistem podržava segmentnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 16MB, maksimalna veličina segmenta je 256B, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 16MB. U tabeli preslikavanja segmenata (SMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti početna adresa segmenta i stvarna veličina segmenta umanjena za 1 (najveći pomeraj koji se unutar segmenta sme generisati), pošto se informacije o smeštaju segmenata na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u polju za stvarnu veličinu segmenta u ulazu u SMT označava da dati segment nije dozvoljen za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a za segmente koje proces sme da koristi, vrednost 0 u polju za fizičku adresu segmenta označava da dati segment trenutno nije u fizičkoj memoriji. Naravno, nijedan segment ne sme početi pre fizičke adrese 1. Trenutno stanje SMT je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0245	01000140	0127	5400094C	0	CDD01227	34	402001A9	0

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje *SMT* nezavisno od drugih, kao da druge nisu generisane). Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati *MAV*; ako data adresa generiše izuzetak tipa stranične greške (engl. *page fault*), upisati PF; ako data adresa upada u zamku (engl. *trap*) usled prekoračenja veličine segmenta, upisati T; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	222	FF32	4002D8	FE3A	FE14
Rezultat adresiranja					

Rešenje:

Unutar virutelne adrese, za predstavljanje pomeraja (*Offset*) unutar segmenta odvojeno je najnižih:

$$OFFSET\_S = \log_2(SEG\_MAX / AU) = \log_2(256B / 1B) = 8 \text{ bita}$$

Dakle, dve najniže cifre heksadekadne virtuelne adrese određuju pomeraj, a preostale segment.
Isto tako, u jednom ulazu u SMT najniže dve cifre heksadekadne vrednosti predstavljaju stvarnu veličinu segmentu umanjenu za jedan, a preostaje cifre predstavljaju početnu fizičku adresu segmenta.

Virtuelna adresa: 222h
Pomeraj: 22h
Segment/ulaz: 2h

Vrednost na ulazu: 0127h
Stvarna veličina segmenta: 27h
Početna fizička adresa segmenta: 01h
⇒ *Generisana fizička adresa:* *01h + 22h = 23h*

Virtuelna adresa: FF32h
Pomeraj: 32h
Segment/ulaz: FFh

Vrednost na ulazu: 34h

Stvarna veličina segmenta: 34h
Početna fizička adresa segmenta: 00h
⇒ *Page Fault:*

Virtuelna adresa: 4002D8h
Pomeraj: D8h
Segment/ulaz: 4002h

Vrednost na ulazu: 0h
Stvarna veličina segmenta: 0h
Početna fizička adresa segmenta: 0h
⇒ *Memory Access Violation*

Virtuelna adresa: FE3Ah
Pomeraj: 3Ah
Segment/ulaz: FEh

Vrednost na ulazu: CDD01227h
Stvarna veličina segmenta: 27h
Početna fizička adresa segmenta: CDD012h
⇒ *Trap (pomeraj je veći od stvarne veličine segmenta)*

Virtuelna adresa: FE14h
Pomeraj: 14h
Segment/ulaz: FEh

Vrednost na ulazu: CDD01227h
Stvarna veličina segmenta: 27h
Početna fizička adresa segmenta: CDD012h
⇒ *Generisana fizička adresa: CDD012h + 14h = CDD026h*

Konačan izgled tabele:

Virtuelna adresa	222	FF32	4002D8	FE3A	FE14
Rezultat adresiranja	23	PF	MAV	T	CDD026

5. (*Mart 2010*) Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelna adresa je 64-bitna, stranica je veličine je 64KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 4GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u ulazu u PMT označava da data stranica nije učitana u fizičku memoriju (nijedna stranica korisničkog procesa ne preslikava se u okvir 0 gde se inače nalazi interapt vektor tabela). Trenutno stanje PMT dva posmatrana procesa A i B je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	0	0	0	14	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	2314	01AD	0	22	01AE	0	0

Prikazati sadržaj ovih tabela nakon što operativni sistem završi sve sledeće akcije tim redom:

- obradio straničnu grešku (*page fault*) koju je generisao proces A kada je adresirao adresu FE030203h u svom adresnom prostoru, tako što je izbacio stranicu broj 02h procesa B i na njeno mesto učitao traženu stranicu procesa A
- obradio straničnu grešku (*page fault*) koju je generisao proces B kada je adresirao adresu 02FEFFh u svom adresnom prostoru, tako što je izbacio stranicu broj FFh procesa A i na njeno mesto učitao traženu stranicu procesa B
- obradio sistemski poziv procesa A kojim je taj proces zahtevao deljenje svoje stranice broj 2h sa stranicom broj 1h procesa B

Rešenje:

Virtuelna adresa je širine 64 bita, od čega je za predstavljanje pomeraja u okviru stranice odvojeno:

$$OFFSET\_S = \log_2(PAGE / AU) = \log_2(64KB / 1B) = 16 \text{ bita}$$

Dakle, poslednje 4 cifre heksadekadne adrese predstavljaju pomeraj, a prvih 12 (ako nisu izostavljene vodeće nule) predstavljaju broj stranice (tj. ulaza u PMT).

Prva akcija:

adresirana adresa: FE030203h ⇒ ulaz FE03 u PMT

proces A pristupa ulazu FE03 u svojoj PMT, ali stranica sa tim brojem se ne nalazi u memoriji, pa se:

5. *generiše Page fault*
6. *stranica se dovlači sa HDD-a (adresa stranice se nalazi u deskriptoru sa ulaza FE03)*
7. *stranica se upisuje u okvir 2314 (izbačena stranica procesa B sa ulaza 2 je zauzimala okvir 2314)*
8. *ažuriraju se odgovarajući ulazi u obe PMT (ulaz FE03 tabele procesa A i ulaz 2 tabele procesa B)*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...	FE03	...
Vrednost	FE12	FEFF	0	0	0	0	14	FE	0	2314	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...

Vrednost	0	12	0	01AD	0	22	01AE	0	0

Druga akcija:
adresirana adresa: 02FEFFh ⇒ ulaz 2 u PMT
proces B pristupa ulazu 2 u svojoj PMT, ali stranica sa tim brojem se ne nalazi u memoriji, pa se:

5. *generiše Page fault*
6. *stranica se dovlači sa HDD-a (adresa stranice se nalazi u deskriptoru sa ulaza 2)*
7. *stranica se upisuje u okvir 14 (izbačena stranica procesa A sa ulaza FF je zauzimala okvir 14)*
8. *ažuriraju se odgovarajući ulazi u obe PMT (ulaz FF tabele procesa A i ulaz 2 tabele procesa B)*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...	FE03	...
Vrednost	FE12	FEFF	0	0	0	0	0	FE	0	2314	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	14	01AD	0	22	01AE	0	0

Treća akcija:

- *proces A zahteva deljenje svoje stranice broj 2 (koja se u tom trenutku ne nalazi u memoriji) sa stranicom broj 1 procesa B (stranica se nalazi u memoriji i zauzima okvir 12);*
- *potrebno je da se u PMT-u procesa A ažurira vrednost sa ulaza 2, tako da ta vrednost bude jednaka vrednosti sa ulaza 1 u PMT-u procesa B;*
- *prethodno ima za posledicu da proces A sada može da pristupa stranici procesa B koja se nalazi u okviru broj 14;*

Nakon ažuriranja vrednosti ulaza, PMT oba procesa izgleda kao na slici:

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...	FE03	...
Vrednost	FE12	FEFF	12	0	0	0	0	FE	0	2314	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	14	01AD	0	22	01AE	0	0

6. (*Mart 2009*) Neki sistem podržava segmentnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 4GB, maksimalna veličina segmenta je 4MB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 64MB. U tabeli preslikavanja segmenata (SMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti početna adresa segmenta i stvarna veličina segmenta umanjena za 1 (najveći pomeraj koji se unutar segmenta sme generisati), pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u polju za stvarnu veličinu segmenta u ulazu u SMT označava da dati segment nije dozvoljen za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a za segmente koje proces sme da koristi, vrednost 0 u polju za fizičku adresu segmenta označava da dati segment trenutno nije u fizičkoj memoriji. Naravno, nijedan segment ne sme početi pre fizičke adrese 1. Trenutno stanje SMT je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0245	1000140	0127	5400094C	0	2CDD00027	034	43C002001A9	0

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje SMT nezavisno od drugih, kao da druge nisu generisane). Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati MAV; ako data adresa generiše izuzetak tipa stranične greške (engl. *page fault*), upisati PF; ako data adresa upada u zamku (engl. *trap*) usled prekoračenja veličine segmenta, upisati T; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	80013A	EB28C32	3F80028D	403001E2	F0A001E4
Rezultat adresiranja					

Rešenje:

Unutar virutelne adrese, za predstavljanje pomeraja (*Offset*) unutar segmenta odvojeno je najnižih:

$$OFFSET\_S = \log_2(SEG\_MAX / AU) = \log_2(4MB / 1B) = 22 \text{ bita}$$

Virtuelna adresa: 80013Ah (8 = 1000b)
Pomeraj: 00013Ah
Stranica/ulaz: 10b = 2h

Vrednost na ulazu: 0127h
Stvarna veličina segmenta: 0127h
Početna fizička adresa segmenta: 0h
⇒ *Page Faulit*

Virtuelna adresa: EB28C32h (B = 1011b)
Pomeraj: 328C32h
Stranica/ulaz: 1110 10b = 3Ah

Vrednost na ulazu: 0h
Stvarna veličina segmenta: 0h
Početna fizička adresa segmenta: 0h
⇒ *Memory Access Violation*

Virtuelna adresa: 3F80028Dh (8 = 1000b)
Pomeraj: 00028Dh
Stranica/ulaz: 0011 1111 10b = 0FEh

Vrednost na ulazu: 2CDD00027h (D = 1101b)
Stvarna veličina segmenta: 100027h
Početna fizička adresa segmenta: 0010 1100 1101 11h = 0B37h
⇒ *Generisana fizička adresa: 0B37h + 00028Dh = 000DC4h*

Virtuelna adresa: 403001E2h (3 = 0011b)
Pomeraj: 3001E2h
Stranica/ulaz: 0100 0000 00b = 100h

Vrednost na ulazu: 43C002001A9h (2 = 0010b)
Stvarna veličina segmenta: 2001A9h
Početna fizička adresa segmenta: 0100 0011 1100 0000 0000 00b = 10F000h
⇒ *Trap (pomeraj je veći od stvarne veličine segmenta)*

Virtuelna adresa: F0A001E4h (A = 1010b)
Pomeraj: 2001E4h
Stranica/ulaz: 1111 0000 10b = 3C2h

Vrednost na ulazu: 0h
Stvarna veličina segmenta: 0h
Početna fizička adresa segmenta: 0h
⇒ *Memory Access Violation*

Konačan izgled tabele:

Virtuelna adresa	80013A	EB28C32	3F80028D	403001E2	F0A001E4
Rezultat adresiranja	PF	MAV	DC4	T	MAV

7. (*April 2008*) Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 4GB, stranica je veličine 256KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 16GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica kojoj odgovara taj ulaz preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost -1 (sve jedinice) u ulazu u PMT označava da data stranica nije dozvoljena za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a vrednost 0 označava da data stranica jeste dozvoljena za pristup, ali nije u fizičkoj memoriji. (Naravno, prvi i poslednji okvir u fizičkom adresnom prostoru se zbog toga nikada ne koriste za smeštanje stranica procesa.) Trenutno stanje PMT je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Ulaz	0	1	2	3	...	3FE	3FF	400	...
Vrednost	0	25F	0	FF0	-1	2AD0	0	14	-1

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje PMT nezavisno od drugih, kao da druge nisu generisane). Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati MAV; ako data adresa generiše izuzetak tipa stranične greške (engl. *page fault*), upisati PF; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	FFC2673	D385A	FF7FB32	FFFA8C4	10012EB8
Rezultat adresiranja					

Rešenje:

Broj bita potreban za predstavljanje pomeraja (*Offset*) unutar stranice je:

$$\log_2(\text{PAGE} / \text{AU}) = \log_2(256\text{KB} / 1\text{B}) = 18 \text{ bita}$$

Dakle, najnižih 18 bita virtuelne adrese predstavljaju pomeraj unutar stranice, a preostali viši biti predstavljaju broj stranice.

Virtuelna adresa: FFC2673h (C = 1100b)
Pomeraj: 02673h
Stranica/ulaz: 1111 1111 11b = 3FFh
Vrednost na ulazu: 0
⇒ *Page Fault*

Virtuelna adresa: D385Ah (D = 1101b)
Pomeraj: 1385Ah
Stranica/ulaz: 11b = 3h
Vrednost na ulazu: FF0
⇒ *Fizička adresa:* *(FF0h << 18) + 0001 385Ah = 3FC0 0000h + 0001 385Ah = 3FC1 385Ah*

Virtuelna adresa: FF7FB32h (7 = 0111b)
Pomeraj: 3FB32h
Stranica/ulaz: 1111 1111 01b = 3FDh
Vrednost na ulazu: -1
⇒ *Memory Access Violation*

Virtuelna adresa: FFFA8C4h (F = 1111b)
Pomeraj: 3A8C4h
Stranica/ulaz: 1111 1111 11b = 3FFh
Vrednost na ulazu: 0
⇒ *Page Fault*

Virtuelna adresa: 10012EB8h (1 = 0001b)
Pomeraj: 12EB8h
Stranica/ulaz: 0001 0000 0000 00b = 400h
Vrednost na ulazu: 14
⇒ *Fizička adresa:* *(14h << 18) + 0001 2EB8h = 0050 0000h + 0001 2EB8h = 0051 2EB8h*

Konačan izgled tabele:

Virtuelna adresa	FFC2673	D385A	FF7FB32	FFFA8C4	10012EB8
Rezultat adresiranja	PF	3FC1385A	MAV	PF	512EB8

8. (*April 2007*) Neki računar podržava straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Stranica je veličine 1KB. Jedan ulaz u tabeli preslikavanja stranica (PMT) zauzima jednu reč, s tim da najviši bit ukazuje na to da li je stranica u memoriji ili ne, a najniži biti predstavljaju broj okvira u fizičkoj memoriji u koji se stranica preslikava.

a. prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja.
b. na jeziku C napisati kod funkcije:

```
void setPageDescr(unsigned int* pmtp, unsigned int page, unsigned int frame);
```

koja u tabelu na čiji početak ukazuje dati pokazivač `pmtp`, za stranicu sa datim brojem `page`, upisuje deskriptor tako da se ta stranica preslikava u okvir sa datim brojem `frame`.

Rešenje:

a. Veličina virtuelnog adresnog prostora je: $VAS = 16\ GB = 2^{34}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 1\ GB = 2^{30}\ B$
Veličina adresibilne jedinice je: $AU = 32\ bit = 2^{2}\ B$
Veličina stranice (okvira) je: $PAGE = 1\ KB = 2^{10}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 32$ bita
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 28$ bita

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(PAGE / AU) = 8$ bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj stranice u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, pa je za predstavljanje broja stranice potrebno:

VA_S - OFFSET_S = 32 bita - 8 bita = 24 bita

Dakle, logička struktura virtuelne adrese je: VA(32) = *Page*(24):*Offset*(8) .

Fizička adresa se sastoji od određenog broja bita koji određuju broj okvira u fizičkom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar tog okvira. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, pa je za predstavljanje broja okvira potrebno:

PA_S - OFFSET_S = 28 bita - 8 bita = 20 bita

Dakle, logička struktura fizičke adrese je: PA(28) = *Frame*(20):*Offset*(8) .

b.
```
void setPageDescr(unsigned* pmtp, unsigned page, unsigned frame) {
    //adresa na koju treba upisati deskriptor je: pmtp + page ⇔ pmtp[page]
    //objašnjenje maske: prvo se komplementiranjem nule dobija maska čiji su svi bitovi 1
    //deljenjem maske sa dva, u najviši bit se upisuje 0 (ostali ostaju 1)
    //komplementiranjem nove maske dobijamo masku čiji je najviši bit 1, a svi ostali 0
    //čime obezbeđujemo da najviši bit desktriptora bude 1 (jer je stranica učitana)
    pmtp[page] = frame | ~((unsigned int)~0 / 2);
}
```

napomena: zadatak je skoro potpuno isti kao i prvi zadatak iz ove grupe zadataka

9. (*April 2007*) Neki sistem podržava segmentno-straničnu organizaciju virtuelne memorije. Za procese kreirane nad istim programom moguće je uštedeti na potrošnji fizičke memorije tako što se programski kod tog programa u fizičku memoriju smešta samo na jedno mesto, pri čemu se okviri u kojima se nalaze stranice koje pripadaju segmentima koji sadrže samo programski kod fizički dele između ovih procesa (zajednički su).

a. precizno objasniti kako se ovo deljenje stranica može implementirati.
b. čija je odgovornost da obezbedi ovo deljenje stranica, hardvera ili operativnog sistema?
c. precizno objasniti zašto ovakvo deljenje nije moguće za stranice koje pripadaju segmentima koji sadrže stek.

Rešenje:

a. Da bi se deljenje stranica moglo implementirati, potrebno je imati neku strukturu podataka pomoću koje se vodi evidencija o svim pokrenutim programima. Za svaki od pokrenutih programa potrebno je da postoji i lista PCB-ova procesa koji su pokrenuti nad tim programom.
   Prilikom pokretanja novog programa, njegov PCB treba dodati u listu PCB-ova procesa nad tim programom. Ukoliko je ta lista pre toga bila prazna, to je prvi proces kreiran nad tim programom pa treba učitati odgovarajuće podatke u blokove fizičke memorije (i ažurirati SMT/PMT). Ukoliko je lista PCB-ova sadržala bar jedan element, onda već postoji bar jedan pokrenut proces nad tim programom, pa je dovoljno u odgovarajuće ulaze SMT/PMT kopirati vrednosti istih ulaza postojećeg procesa.
   Prilikom izbacivanja ili učitavanja neke deljene stranice, potrebno je da se ažuriraju odgovarajući ulazi u SMT/PMT svakog procesa nad tim programom (jer svi dele tu stranicu).
   Može se primetiti da u ovom rešenju postoji izvesna redundansa podataka u ulazima vezanih za deljene stranice, pa da bi se to izbeglo, moguće je održavati pomoćnu strukturu za deljene stranice, a na koju upućuju odgovarajući ulazi u SMT/PMT.

b. jasno je da odgovornost na OS-u da obezbedi ovakav način deljenja određenih stranica i da vrši njihovo ažuriranje pri bilo kakvoj promeni stanja (učitana nova stranica, izbačena postojeća, ugašen neki proces nad određenim programom...)

c. stranice koje pripadaju segmentima koji sadrže stek sadrže podatke koji su specifični za svaki proces ponaosob (sadrže lokalne podatke tog procesa, "trag" izvršavanja programa...) čime se pravi razlika između više različitih procesa nad istim programom. Samim tim, svaki proces mora da ima svoj sopstveni stek, jer je to deo njegovog konteksta, tako da ne bi bilo ispravno deliti stek sa drugim procesom (na stek mogu da se upisuju vrednosti, a ne samo da se čitaju, tako da bi promena zajedničkog steka od strane jednog procesa uticala na tok izvršavanja drugog procesa)

10. (*April 2006*) Neki računar podržava segmentno-stranični mehanizam virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 8GB, a adresibilna jedinica je 16-bitna reč. Stranica je veličine 512B. Maksimalan broj segmenata u virtuelnom adresnom prostoru je 4. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.

Rešenje:

Veličina fizičkog adresnog prostora je: PAS = 8 GB = 2^{33} B
Veličina adresibilne jedinice je: AU = 16 bit = 2^{1} B
Veličina stranice je: PAGE = 512 B = 2^{9} B

Širina virtuelne adrese je: VA_S = 16 bita
Širina fizičke adrese je: PA_S = $\log_2$(PAS / AU) = 32 bita
Širina pomeraja (*Offset*) unutar stranice: OFFSET_S = $\log_2$(PAGE / AU) = 8 bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj segmenta u virtuelnom adresnom prostoru, određenog broja bita koji određuju broj stranice unutar segmenta i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, dok su za predstavljanje segmenta potrebna 2 bita (jer imamo maksimalno 4 segmenta). Odatle sledi da je za predsavljanje stranice unutar segmenta potrebno:

VA_S - SEGMENT_S - OFFSET_S = 16 bita - 2bita - 8 bita = 6 bita

Dakle, logička struktura virtuelne adrese je: VA(16) = *Segment(2)*:*Page*(6):*Offset*(8) .

Fizička adresa se sastoji od određenog broja bita koji određuju broj bloka u fizičkom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar tog bloka. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 8 bita, pa je za predstavljanje broja bloka potrebno:

PA_S - OFFSET_S = 32 bita - 8 bita = 24 bita

Dakle, logička struktura fizičke adrese je: PA(32) = *Block*(24):*Offset*(8) .

Трећа група задатака: Нити и промена контекста

1. (*Mart 2012*) Dat je sledeći kod koji koristi školsko jezgro:

```
class TreeNode {
      public:
      TreeNode* getLeftChild();
      TreeNode* getRightChild();
      void process();
       ...
};
class TreeVisitor : public Thread {
      public:

      TreeVisitor (TreeNode* root) : myRoot(root) {}


      protected:

      virtual void run();
      void visit(TreeNode*);


      private:

      TreeNode* myRoot;
};

void TreeVisitor::run () {
      visit(myRoot);
}

void TreeVisitor::visit (TreeNode* node) {
      if (node==0) return;
      TreeNode* rn = node->getRightChild();
      if (rn) (new TreeVisitor(rn))->start();
      node->process();
      visit(node->getLeftChild());
}

void userMain () {
      TreeNode* root = ...;
      Thread* thr = new TreeVisitor(root);
      thr->start();
}
```

Napisati C kod koji radi isto, odnosno obilazi dato stablo rekurzivnim kreiranjem niti koje obilaze podstabla, samo korišćenjem nekog sistema u kome se niti kreiraju i odmah pokreću kao parametrizovani pozivi funkcije sledećim sistemskim pozivom:

```
void create_thread (void(*thread_body)(void*), void* param);
```

Klasa `TreeNode` implementirana je sledećim C kodom:

```
struct TreeNode;
TreeNode* getLeftChild (TreeNode*);
TreeNode* getRightChild (TreeNode*);
void process (TreeNode*);
```

Rešenje:

Obilazak stabla se vrši na sledeći način:

- ako je koren *null* završava se obilazak (nema šta da se obiđe)
- u suprotnom, ako desni sin korena nije *null*, pravi se nova nit (aktivan objekat klase `TreeVisitor`) nad njim kao korenom koja vrši obilazak desnog podstabla
- obiđe se koren (`process()`)
- rekurzivno se pozove metoda `visit(TreeNode*)` za obilazak levog podstabla

Dakle, potrebno je implementirati funkciju za obilazak stabla na isti način kao u datoj implementaciji, ali tako da deklaracija funkcije odgovara deklaraciji pokazivača na funkciju koji prihvata funkcija `create_thread` koja pravi i pokreće nit.

Definicija funkcije za obilazak stabla i glavnog programa data je sledećim kodom:

```
void visit (void* nd) {
	//eksplicitno kastovanje argumenta u tip (TreeNode*)
	TreeNode* node = (TreeNode*)nd;

	//ako je koren null, nemamo šta da obilazimo, pa se vraćamo iz funkcije
	if (node==0) return;

	//u suprotnom, nalazimo pokazivač na desnog sina (desno podstablo)
	TreeNode* rn = getRightChild(node);

	//ako desno podstablo postoji, onda pravimo novu nit koja ga obilazi
	if (rn) create_thread(&visit,rn);

	//obilazimo koren
	process(node);

	//rekurzivni poziv obilaska levog podstabla
	visit(getLeftChild(node));
}

void main () {
	//pokazivač na koren stabla za obilazak
	TreeNode* root = ...;

	//pravimo novu nit koja obilazi dato stablo
	create_thread(&visit,root);
}
```

2. (*Mart 2011*) Klasa `Node` čija je delimična definicija data dole predstavlja čvor binarnog stabla. Funkcije `getLeftChild()` i `getRightChild()` vraćaju levo, odnosno desno podstablo datog čvora (tačnije, njegov levi i desni čvor-potomak). Funkcija `visit()` obavlja nekakvu obradu čvora.

```
class Node {
public:
  Node* getLeftChild();
  Node* getRightChild();
  void  visit();
  ...
};
```

Potrebno je obilaziti dato binarno stablo korišćenjem uporednih niti na sledeći način. Ako neka nit trenutno obilazi neki čvor, onda ona treba da napravi novu nit-potomka koja će obići desno podstablo tog čvora, a sama ta nit će obraditi dati čvor i nastaviti sa obilaskom levog podstabla tog čvora, i tako rekurzivno.
Realizovati klasu `TreeVisitor` izvedenu iz klase `Thread` iz školskog jezgra koja realizuje opisani obilazak binarnog stabla. Ova klasa treba da se koristi na sledeći način:

```
Node* tree = ...;  // The root node of a binary tree
TreeVisitor* tv = new TreeVisitor(tree);
tv->start();
```

Rešenje:

Tražena implementacija klase `TreeVisitor` se u potpunosti logički poklapa sa implementacijom datom u tekstu prethodnog zadatka.

Razlike u samom kodu u odnosu na datu implementaciju i implementaciju traženu u ovom zadatku su:

- čvor stabla je predstavljen klasom `TreeNode`, a ne `Node`
- metoda klase kojom je predstavljen čvor nosi naziv `process()`, a ne `visit()`

3. (*Mart 2011*) U nekom 32-bitnom RISC procerosu svi registri su 32-bitni, adrese su 32-bitne, a adresibilna jedinica je bajt. Prevodilac za jezik C za taj procesor povratnu vrednost funkcije prenosi kroz registar r0. Data je implementacija standardne funkcije `setjmp()` za taj procesor i taj prevodilac:

```
int setjmp (jmp_buf buf) {
  asm {
      load  r0,-1*4[sp];      // r0:=buf
      store psw,0*4[r0];
      store r1,1*4[r0];
      store r2,2*4[r0];
      ...
      store r31,31*4[r0];
      pop   r1;               // pop return address to save the caller's context
      store sp,32*4[r0];      // save sp from the caller's context
      store r1,33*4[r0];      // save pc from the caller's context
      push  r1;               // push return address back to the stack
      load  r1,1*4[r0];       // restore r1
      clr r0;                 // r0:=0
  }
}
```

Napisati implementaciju standardne funkcije `longjmp()`:

```
void longjmp (jmp_buf buf, int val);
```

Rešenje:

Pre poziva funkcije `longjmp` na steku su sačuvani `val`, `buf` i `pc`, tim redom. Kod funkcije:

```
void longjmp (jmp_buf buf, int val) {
  asm {

      load r0,-2*4[sp];                  // upisujemo argument val u registar r0
      and  r0,r0,r0;                     // ako je r0 == 0, flag Z se postavlja na 1
      jnz  continue                      // ako je r0 različito od 0, nastavljamo
      load r0,#1                         // u suprotnom, upisujemo 1 u r0, jer povratna
                                         // vrednost funkcije ne sme biti 0
continue:
      load r1,-1*4[sp];                  // argument buf u registar r1

      load psw,0*4[r1];                  // restauiramo psw iz buf
      load r3,3*4[r1];                   // restauiramo registre od r3..
      load r4,4*4[r1];
      ...
      load r31,31*4[r1];                 // ..do r31

      load sp,32*4[r1];                  // restauiramo sp
      load r2,33*4[r1];                  // upisujemo pc u r2 (pc je bio sačuvan u buf)
      push r2                            // stavljamo pc na restauiramo stek,
                                         // jer je vrednost pc-a bila skinuta u setjmp()
      load r2,2*4[r1];                   // r2 nam nije više potreban kao pomoćni
                                         // registar, pa ga restauiramo iz buf
      load r1,1*4[r1];                   // isto to važi i za r1
      ret;                               // vraćamo se iz funkcije na mesto poziva
                                    // setjmp, ali sa drugom povratnom vrednošću
  }
}
```

4. (*Maj 2011*) U nekom 32-bitnom RISC procesoru svi registri su 32-bitni, adrese su 32-bitne, a adresibilna jedinica je bajt. Prevodilac za jezik C za taj procesor povratnu vrednost funkcije prenosi kroz registar r0. Data je jedna nekorektna implementacija standardnih funkcija `setjmp()` i `longjmp()` za taj procesor i taj prevodilac:

```
int setjmp (jmp_buf buf) {
  asm {
      clr r0; // r0:=0
      push r1;
      load r1,-2*4[sp]; // r1:=buf
      store sp,0*4[r1];
      store psw,1*4[r1];
      store r2,2*4[r1];
      store r3,3*4[r1];
      ...
      store r31,31*4[r1];
      store pc,32*4[r1];
      pop r1
  }
}

void longjmp (jmp_buf buf, int val) {
  asm {
      load r0,-2*4[sp]; // r0:=val
      and  r0,r0,r0;    // fix r0 if it is zero
      jnz  continue
      load r0,#1
continue:
      load r1,-1*4[sp]; // r1:=buf
      load sp,0*4[r1];
      load psw,1*4[r1];
      load r2,2*4[r1];
      load r3,3*4[r1];
      ...
      load r31,31*4[r1];
      load pc,32*4[r1];
  }
}
```

Posmatrati upotrebu ovih funkcija, na primer kao u operaciji `dispatch()` školskog jezgra i precizno objasniti šta je problem sa ovom implementacijom.

Rešenje:

Funkcija `longjmp` se oslanja na vrednosti PC-a i registra R1 koje su sačuvane na steku u pozivu funkcije *setjmp*. Međutim, nakon prvog izlaska iz funkcije `setjmp` (kada vraća 1, pre skoka iz `longjmp`-a), sa steka se skidaju svi podaci sačuvani na njemu u funkciji `setjmp` (tačnije, podaci ostaju na steku, ali se pokazivač na vrh steka pomeri, tako da se te lokacije tretiraju kao slobodne, jer su ostale iznad "vrha steka").

Ako se posle prvog povratka iz `setjmp`-a (a pre poziva `longjmp`), unutar tela `if` bloka izvršavaju instrukcije koje koriste stek za čuvanje nekakvih međurezultata ili drugih podataka, velika je verovatnoća da će sačuvane vrednosti PC-a i registra R1 biti pregažene novim.

Ako se to desi, funkcija `longjmp` neće moći korektno da se izvrši, a samim tim ni promena konteksta.

Primer tog slučaja je baš školsko jezgro jer se u kodu funkcije `dispatch()`, nakon povratka iz `setjmp`, izvršava kod koji poziva funkcije `Scheduler`-a.

5. (*Maj 2011*) U nekom operativnom sistemu sistemski poziv `clone()` pravi novu nit (*thread*) kao klon roditeljske niti, sa istovetnim kontekstom izvršavanja i u istom adresnom prostoru kao što je i roditeljska nit – isto kao i `fork()`, samo što pravi nit u istom adresnom prostoru, a ne proces. Ovaj poziv vraća 0 u kontekstu niti-deteta, a vrednost veću od 0 koja predstavlja ID kreirane niti u kontekstu niti-roditelja. Vraćena vrednost manja od 0 označava neuspešan poziv. Sistemski poziv `terminate(int)` gasi nit sa datim ID. Korišćenjem ovog sistemskog poziva realizovati klasu `Thread` sa istim interfejsom kao u školskom jezgru (kreiranje niti nad virtuelnom funkcijom `run()` i pokretanje niti pozivom funkcije `start()`).

Rešenje:

```
class Thread {
      public:
        int start ();             //eksplicitno pokretanje niti


      protected:
        Thread () : myID(0) {}    //konstruktor
        virtual void run() {}     //polimorfna metoda koja obavlja posao niti


      private:
        int myID;                 //atribut potreban za identifikaciju niti
};

int Thread::start () {
      int id = clone();           //poziv funkcije clone() za kreiranje nove niti

      if (id<0) return id;        //ako poziv nije uspeo vraćamo se iz metode,
                                  //jer nit nije uspeštno kreirana

      if (id>0) {                 //ako je poziv uspeo, roditeljskoj niti je vraćen
            myID=id;              //identifkator niti veći od nule, pa ovaj deo koda
            return 0;             //izvršava samo nit roditelj (roditelj postavlja)
      }                           //myID na ID niti deteta, i izlazi iz metode)


      //kako je roditeljska nit u prethodnom if-u izašla iz metode, ovo se izvršava
      //samo u kontekstu niti deteta (nit deteta nastavlja izvršavanje od tačke
      //poziva funkcije clone()

      //nit dete poziva funkciju run() koja obavlja potrebni posao
      run();

                //obe niti dele zajednički adresni prostor, dakle i polje myID;
      //moguće je da je nit dete završila metodu run() i stigla do while petlje
      //pre nego što je nit roditelj uopšte ušla u if blok u kome postavlja myID,
      //zbog toga uposleno čekamo da nit roditelj postavi myID na ID niti deteta
      while (myID==0);

      //kada nit roditelj postavi myID na ID niti deteta, možemo da ugasim nit dete
      //pozivom metode terminate, jer znamo ID te niti (sačuvan je u polju myID)
      terminate(myID);
}
```

6. (*Septembar 2011*) U implementaciji jezgra nekog jednoprocesorskog *time-sharing* operativnog sistema, radi pojednostavljenja celog mehanizma promene konteksta, primenjeno je sledeće neobično rešenje. Promena konteksta vrši se isključivo kao posledica prekida, na samo jednom mestu u kodu koji se izvršava na prekid. Prekidi dolaze od raznih uređaja, u najmanju ruku od vremenskog brojača, jer on u svakom slučaju generiše prekid u konačnom vremenu, bilo zbog toga što je tekućoj niti isteklo dodeljeno procesorsko vreme, ili iz nekog drugog razloga zbog koga je vremenski brojač od strane sistema bio startovan da meri interval i generiše prekid kada on istekne. Zbog toga tekuća nit nikada ne gubi procesor sinhrono, čak ni kada poziva blokirajuću operaciju npr. na semaforu. Umesto toga, ukoliko je potrebno da se nit blokira u nekom blokirajućem pozivu, nit se samo „označi" blokiranom i nastavlja sa izvršavanjem (uposlenim čekanjem) sve dok ne stigne sledeći prekid. Kada takav prekid stigne, kod koji se poziva proverava stanje niti koja je prekinuta i ako je ona označen kao blokirana, vrši samu promenu konteksta.
Na primer, implementacija operacije `wait()` na semaforu u ovom sistemu izgleda ovako:

```
class Thread {
  int isBlocked;
  jmpbuf context;
  static Thread* running;
  ...;
};

void Semaphore::wait () {
  lock();                                 // Disable interrupts
  if (--val<0) {
      Thread::running->isBlocked = 1;     // Mark as blocked,
      queue->put(Thread::running);        // put it in the semaphore queue,
      unlock();                           // enable interrupts,
      while (Thread::running->isBlocked); // and then busy-wait
  }                                       // until it is preempted, suspended, and
                                          // then resumed later
  else unlock();                          // Enable interrupts
}
```

Korišćenjem standardnih bibliotečnih operacija `setjmp()` i `longjmp()`, realizovati prekidnu rutinu `yield()` koja vrši opisanu promenu konteksta na prekid. Klasa Schedule koja realizuje red spremnih niti sa raspoređivanjem ima interfejs kao u školskom jezgru.

Rešenje:

```
//pažnja: yield() je prekidna rutina, pa je potrebno dodati modifikator "interrupt"
void interrupt yield () {
      //čuvamo kontekst trenutno izvršavajuće niti
      if (setjmp(Thread::running->context)==0) {

            //ako je tekuća nit blokirana u nekom blokirajućem pozivu, onda je već
            //smeštena u listu blokiranih, pa nije potrebno da se smešta u Scheduler;
            //ako tekuća nit nije blokirana, onda ju je potrebno smestiti u
            //Scheduler pre preuzimanja i promene konteksta
            if (!Thread::running->isBlocked)
                  Scheduler::put(Thread::running);

            //uzimamo novu nit iz liste spremnih niti i postavljamo je za tekuću
            Thread::running = Scheduler::get();

            //resetujemo vrednost isBlocked atributa, jer nit više nije blokirana
            Thread::running->isBlocked = 0;

            //restauiramo kontekst nove tekuće niti
            longjmp(Thread::running->context,1);
      }
```

```
}
```

7. (*Septembar 2011*) Klasa `Node` čija je delimična definicija data dole predstavlja čvor binarnog stabla. Funkcije `getLeftChild()` i `getRightChild()` vraćaju levo, odnosno desno podstablo datog čvora (tačnije, njegov levi i desni čvor-potomak). Funkcija `visit()` obavlja nekakvu obradu čvora.

```
class Node {
public:
  Node* getLeftChild();
  Node* getRightChild();
  void  visit();
  ...
};
```

Potrebno je obilaziti dato binarno stablo korišćenjem uporednih niti na sledeći način. Ako neka nit trenutno obilazi neki čvor, onda ona treba da napravi novu nit-potomka koja će obići desno podstablo tog čvora, a sama ta nit će obraditi dati čvor i nastaviti sa obilaskom levog podstabla tog čvora, i tako rekurzivno.
Realizovati globalnu operaciju `visit(Node* root)` koja vrši opisani obilazak stabla sa datim korenim čvorom. Niti treba kreirati sistemskim pozivom `fork()` za koga treba pretpostaviti da ima istu sintaksu i semantiku kao i istoimeni Unix sistemski poziv, samo što umesto procesa kreira nit u istom adresnom prostoru roditelja.

Rešenje:

Jedna od mogućih implementacija tražene funkcije:

```
void visit(Node* root) {
      //ako je root null, vraćamo se iz funkcije jer nemamo šta da obilazimo
      if (!root) return;

      //dohvatamo pokazivače na levo i desno podstablo
      Node* ln = root->getLeftChild();
      Node* rn = root->getRightChild();

      //novu nit za obilazak desnog podstabla ima smisla praviti
      //samo ako desno podstablo postoji (pokazivač rn nije null)
      if (rn) {
            //pravimo novu nit, i ako smo u kontekstu izvršavanja nove niti
            if (fork()==0) {
                  //obilazimo desno podstablo, i nakon obilaska se vraćamo iz funkcije
                  //jer ne treba da izvršavamo ostatak "roditeljskog" koda
                  visit(rn);
                  return;
            }
      }

      //ako smo u kontekstu roditeljske niti, onda obilazimo koren
      root->visit();

      //i ako postoji levo podstablo (pokazivač ln nije null),
      //obilazimo i njega rekurzivnim pozivom funkcije visit
      if (ln) visit(ln);
}
```

napomena: na kraju nije potrebno proveravati da li je ln null, jer će se to uraditi na početku narednog poziva funkcije `visit`

8. (*Mart 2010*) U donju tabelu upisati "Da" ukoliko operativni sistem datu operaciju treba, a "Ne" ukoliko ne treba da izvrši prilikom promene konteksta procesa, odnosno niti.

Rešenje:

Operacija	Promena konteksta procesa	Promena konteksta niti
Restauracija programski dostupnih registara procesora za podatke	Da	Da
Invalidacija TLB-a čiji ključevi ne sadrže identifikaciju procesa	Da	Ne
Invalidacija TLB-a čiji ključevi sadrže identifikaciju procesa	Ne	Ne
Restauracija registra koji čuva identifikaciju korisničkog procesa	Da	Ne
Restauracija pokazivača na tabelu preslikavanja stranica (PMTP)	Da	Ne
Restauracija procesorske statusne reči (PSW)	Da	Da
Invalidacija procesorskog keša koji kao ključeve čuva virtuelne adrese	Da	Ne
Invalidacija procesorskog keša koji kao ključeve čuva fizičke adrese	Ne	Ne
Zatvaranje otvorenih fajlova	Ne	Ne
Restauracija pokazivača steka (SP)	Da	Da

napomena: za konkretna objašnjenja pojedinih odgovora pogledajte slajdove sa predavanja

9. (*Mart 2010*) U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao sys_call, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra, pa čak postoje i niti koje obavljaju sve potrebne radnje prilikom promene konteksta korisničkih procesa (osim samog čuvanja i restauracije konteksta procesora), kao što su smeštanje PCB korisničkog procesa koji je do tada bio tekući u odgovarajući red (spremnih ili blokiranih, u zavisnosti od situacije), izbor novog tekućeg procesa iz skupa spremnih, promenu memorijskog konteksta, obradu samog konkretnog sistemskog poziva, itd. Prilikom obrade sistemskog poziva sys_call, prema tome, treba samo oduzeti procesor tekućem korisničkom procesu i dodeliti ga tekućoj kernel niti.
Na PCB tekućeg korisničkog procesa ukazuje globalni pokazivač runningUserProces, a na PCB tekuće interne niti jezgra ukazuje globalni pokazivač runningKernelThread. I interne niti jezgra vrše promenu konteksta između sebe na isti način, pozivom softverskog prekida koji ima istu internu strukturu kao i rutina sys_call. I korisnički iprocesi i interne niti jezgra u svojim PCB strukturama imaju polje u kojima se čuva SP i čiji je pomeraj u odnosu na početak PCB dat simboličkom vrednošću offsSP.
Procesor je RISC, sa *load/store* arhitekturom. Od registara opšte namene poseduje registarski fajl sa registrima R0..R63, SP, PSW i PC. Prilikom prihvatanja prekida i pre skoka u prekidnu rutinu, hardver procesora na steku čuva sve programski dostupne registre, osim SP.
Napomene: Upotreba imena globalne promenljive iz C koda u asembleru datog procesora predstavlja memorijsku adresu te globalne statičke promenljive. Deo memorije koju koristi jezgro (kod i podaci) preslikava se u virtuelni adresni prostor svih korisničkih procesa, tako da prilikom obrade sistemskog poziva i prelaska u kod kernela nema promene memorijskog konteksta.
Na asembleru datog procesora napisati prekidnu rutinu sys_call.

Rešenje:

```
sys_call:   load r0, [runningUserProcess]       ;adresa PCB-a tekućeg korisničkog
                                                ;procesa smešta se u registar r0
            store sp, offsSP[r0]                ;čuva se vrednost sp-a u odgovarajućem
                                                ;polju PCB strukture tekućeg
                                                ;korisničkog procesa
            load r0, [runningKernelThread]      ;adresa PCB-a tekuće interne niti
                                                ;jezgra smešta se u registar r0
            load sp, offsSP[r0]                 ;restauira se vrednost sp-a iz
                                                ;odgovarajućeg polja PCB strukture
                                                ;tekuće interne niti jezgra
            iret                                ;povratak iz prekidne rutine
```

10. (*Mart 2009*) U jezgru nekog operativnog sistema definisani su sledeći makroi, strukture i funkcije:

```
#define saveAll ...     //sve registre osim pc i sp čuva na vrhu steka (menja se sp)
#define restoreAll ...  //sve registre osim pc i sp restaurira sa steka (menja sp)
#define getSP ...       //vraća vrednost sp registra
#define setSP(expr) ... //izračunava izraz i upisuje ga u registar sp

void copy(int* dst, int* src, int n); //kopira niz od n reči sa src na dst

struct PCB {
  int sp;               // sačuvana vrednost sp registra
  ...                   // NIJE DOZVOLJENO PRAVITI PRETPOSTAVKE U VEZI
                        // SA OSTALIM POLJIMA PCB STRUKTURE
};
PCB* running; // The running process
```

Korišćenjem ovih makroa, funkcije i struktura, napisati kod funkcija `setjump()` i `longjump()` koje će se koristiti u funkciji `dispatch()` na sledeći način:

```
void dispatch() {
  lock();
  if (setjump()==0) {
      Scheduler::put(running);
      running = Scheduler::get();
      longjump();
  } else unlock();
}
```

Poznato je da se radi o RISC procesoru koji pored SP i PC registara ima još 16 opštenamenskih registara. Stek raste ka nižim adresama i SP pokazuje na poslednju zauzetu lokaciju na vrhu steka. Povratnu vrednost funkcije uvek vraćaju kroz registar R0, a prevodilac generiše kod tako da sadržaj svih ostalih registara ostane neizmenjen i nakon poziva funkcije (pod uslovom da se ne menja stek). Parametri se funkcijama prenose preko steka, a za čuvanje međurezultata dati kompajler koristi najmanji mogući broj registara. Ako taj broj premašuje broj registara, onda za tu namenu koristi i lokacije na steku. Smatrati da su svi podaci i sve adrese veličine tipa int i da je adresibilna jedinica jedna reč veličine tipa int. U rešenju nije dozvoljeno korišćenje asemblerskog koda.

Rešenje:

```
int setjump(){
      int sp_pom;                                  //pomoćna promenljiva

      saveAll;                                     //čuvamo sve opštenamenske registre na
                                                   //steku (ukupno 16 sačuvanih registara)
      sp_pom = running->sp = getSP;                //pamtimo SP u PCB strukturi tekućeg
                                                   //procesa i u pomoćnoj promenljivoj
      sp_pom--;                                    //umanjujemo pomoćnu promenljivu za jedan
      setSP(sp_pom);                               //upisujemo novu vrednost u SP, tako da
                                                   //sad pokazuje na prvu slobodnu lokaciju
      copy((int*)sp_pom, (int*)sp_pom+17, 1);      //kopiramo vrednost PC-a (sačuvanu na
                                                   //steku pri ulasku u funkciju) na prvu
                                                   //slobodnu lokaciju na steku
      return 0;                                    //povratak iz funkcije (vraćamo 0)
}

void longjump(){
      int sp_pom = running->sp;                    //u pomoćnu promenljivu upisujemo
                                                   //sačuvanu vrednost SP-a (iz PCB-a)
      setSP(sp_pom);                               //postavljamo SP na tu vrednost
      restoreAll;                                  //skidamo sačuvane opštenamenske registre
      return 1;                                    //povratak iz funkcije (vraćamo 1)
```

```
}
```

11. (*Mart 2009*) Dat je sledeći program:

```
#include <stdio.h>
#define N 3

void main () {
  int i, f = 0, s = 0;
  for (i=0; i<N; i++) {
      f = f || fork();
      s += i;
  }
  if (f)                        //uslov if1
      exit(0);
  printf("%d ",s);
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, da je funkcija `printf` bezbedna za poziv u konkurentnim procesima (svaki poziv se izvršava izolovano i bez interakcije sa pozivima iz drugih procesa, tj., uporedne pozive ove funkcije iz različitih procesa sistem sekvencijalizuje), i da uvek koristi isti standardni izlaz, prikazati ispis koji se dobije pokretanjem datog programa? Odgovor obrazložiti.

Rešenje:

Nakon prvog ulaska u *for* petlju kreira se novi proces, pri čemu funkcija `fork()` starom (početnom) procesu vraća kao rezultat ID novog procesa (koji je veći od nule), a novom procesu vraća kao rezultat nulu. Iz toga sledi da će u promenljivu f biti upisana vrednost nula u kontekstu izvršavanja novog procesa, dok će u kontekstu izvršavanja starog procesa u f biti upisana vrednost veća od nule.

Jednom upisana vrednost veća od nule u promenljivu *f* se nikad više neće vratiti na nulu (jedina instrukcija koja menja f koristi logičko ILI koje ne može da vrati bite sa 1 na 0).
Iz prethodnog sledi da će svi procesi koje nakon toga napravi početni proces naslediti od njega vrednost promenljive *f* veću od nule, što znači da će proći uslov if1, pa neće ništa ispisati (izvršiće se `exit(0)` pre toga).

Generalizovano, kada god neki proces kreira novi proces pozivom fork(), njegova vrednost promenljive f će postati veća od 0, pa taj proces i svi procesi koje on nakon toga bude kreirao neće vršiti nikakav ispis na standardnom izlazu.

Dakle, samo proces kojem su svi "preci" bili novokreirani u određenoj iteraciji neće naslediti vrednost promenljive *f* veću od nule (već te ta vrednost ostati nula) i samo taj proces ne prolazi uslov *if1*, što znači da će doći do ispisa.
U svakoj iteraciji se na promenljivu *s* dodaje vrednost brojačke promenljive *i*, pa je vrednost *s* posle tri iteracije petlje jednaka: 0 + 1 + 2 = 3.

Konačno, samo će jedan proces da ispiše vrednost 3.

12. (*April 2008*) Dat je sledeći program:

```
#include <stdio.h>
#define N 3
int pid[N];

void main () {
  int i;

  for (i=0; i<N; i++) pid[i]=0;
  for (i=0; i<N; i++) {
      if (pid[i]==0)                          // uslov if1
         pid[i]=fork();
      if (pid[i]==0)                          // uslov if2
         for (j=i+1; j<N; j++)
            pid[j]=1;
  }
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, koliko će ukupno procesa biti kreirano (direktno ili indirektno) od strane jednog procesa početno kreiranog nad ovim programom, uključujući i taj jedan početni proces? Odgovor obrazložiti.

Rešenje:

Na početku izvršavanja se svi elementi niza *pid* inicijalizuju nulama.

Nakon prvog ulaska u spoljašnju *for* petlju kreira se novi proces, jer je `pid[0]` jednako nuli pa prolazimo *if1.*

Nakon toga samo novi proces prolazi *if2,* jer je njemu fork() vratio nulu kao rezultat funkcije (ta nula se upisuje u `pid[0]` što omogućava da se prođe uslov `if2` za ulazak u unutrašnju *for* petlju). U unutrašnjoj *for* petlji se svi naredni elementi niza *pid* postavljaju na 1, što ima za posledicu da se uslov *if1* nikad ne prolazi, pa se ne mogu kreirati novi procesi.

Dakle, novi proces ne kreira druge procese, već to isključivo radi stari (početni) proces, pa ćemo dalje posmatrati samo njega.

Nakon drugog ulaska u spoljašnju *for* petlju, početni proces opet prolazi uslov *if1* i kreira još jedan novi proces. Početni proces ne prolazi uslov *if2* iz istog razloga kao u prvoj iteraciji. Novi proces prolazi uslov *if2* iz istog razloga kao i novi proces kreiran u prvoj iteraciji, ali opet, ni on više ne može da kreira nove procese, pa ćemo zato samo posmatrati stari (početni) proces.

Nakon trećeg (i poslednjeg) ulaska u spoljašnju *for* petlju, početni proces opet prolazi uslov *if1* i kreira još jedan novi proces. Početni proces ne prolazi uslov *if2* iz istog razloga kao u prvoj iteraciji. Novi proces prolazi uslov *if2* iz istog razloga kao i novi proces kreiran u prvoj iteraciji, ali opet, ni on više ne može da kreira nove procese.

Dakle, nakon sve tri iteracije, jedini proces koji je prolazio uslov *if1* i kreirao nove procese u svakoj iteraciji je početni proces, što znači da je ukupan broj kreiranih procesa (uključujući i početni) jednak 4.

13. (*April 2007*) Dat je sledeći program:

```
#include <stdio.h>
#define N 3
int pid[N];

void main () {
  int i;

  for (i=0; i<N; i++) pid[i]=0;
  for (i=0; i<N; i++) pid[i]=fork();
  for (i=0; i<N; i++) printf("%d ",pid[i]);
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, koliko ukupno nula ispisuju svi procesi kreirani od strane jednog procesa nad ovim programom? Odgovor obrazložiti.

Rešenje:

Na početku izvršavanja se svi elementi niza *pid* inicijalizuju nulama.

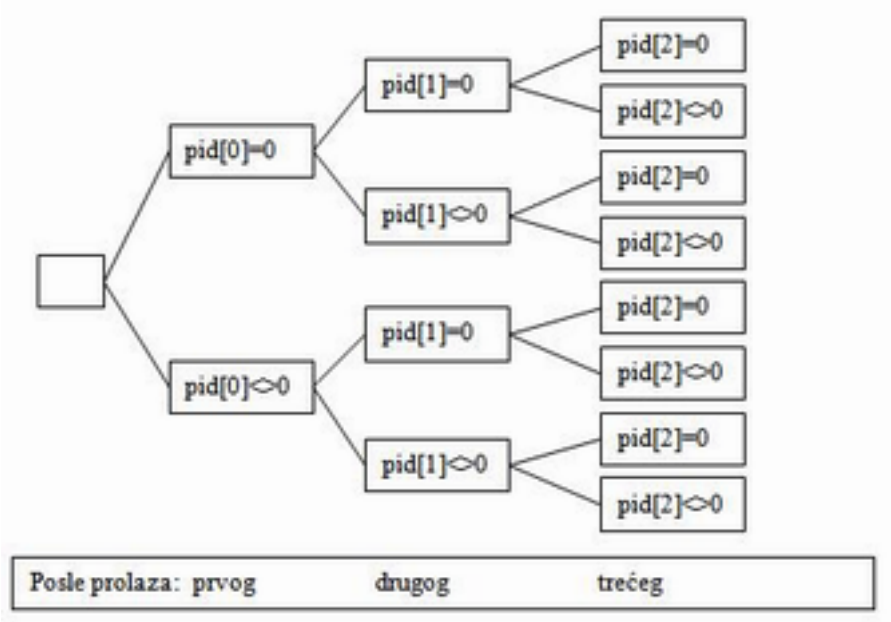
Posle svake iteracije druge for petlje, broj procesa se uvecava dva puta. Na početku postoji samo jedan proces nad datim programom, a pošto imamo tri iteracije kroz petlju, ukupan broj kreiranih procesa je 8.

Posle prve iteracije postojaće dva procesa, od kojih stari ima pid[0]≠0, a novi pid[0]=0. Svaki od njih će u narednim iteracijama kreirati jednak broj procesa, što znači da će na kraju 4 procesa od 8 imati pid[0] =0.

Posle druge iteracije postojaće četiri procesa, od kojih dva stara imaju pid[1]≠0, a dva nova pid[1]=0. Svaki od njih će u narednim iteracijama kreirati jednak broj procesa, što znači da će na kraju 4 procesa od 8 imati pid[1]=0.

Posle treće iteracije postojaće osam procesa, od kojih četiri stara imaju pid[2]≠0, a četiri nova pid[2]=0. Nakon ove iteraacije u petlju se više ne ulazi, što znači da će na kraju 4 procesa od 8 imati pid[2]=0.

Dakle, za svaku poziciju u nizu *pid*, postoje po četiri procesa koji na tom mestu imaju nulu, a kako svaki proces ispisuje sve elemente niza, ukupan broj ispisanih nula biće: 4 + 4 + 4 = 12



14. (*April 2007*) U jezgru nekog operativnog sistema definisane su sledeće strukture podataka:

```
typedef unsigned int PID;
#define MaxProc  ...          // Max number of processes
struct PCB {
  int sp;                     // Saved stack pointer
  PID next;                   // Next PCB in the Ready or a waiting list
  ...
};
PCB* allProcesses[MaxProc];   // Maps a PID to a PCB*
PID running;                  // The running process
PID ready;                    // Head of Ready list
```

Korišćenjem ovih struktura, po uzoru na funkciju datu na predavanjima, na jeziku C i asembleru zamišljenog procesora napisati telo funkcije yield(PCB* current, PCB* next) koja vrši promenu konteksta sa procesa current na proces next; ova funkcija služi samo za realizaciju drugih operacija jezgra i poziva se samo iz koda jezgra. Korišćenjem te funkcije, na jeziku C napisati tela sledećih funkcija:

- `suspend()`: suspenduje (blokira) izvršavanje pozivajućeg procesa sve dok ga neki drugi proces ne „probudi“ pomoću `resume()`;
- `resume(PID pid)`: „budi“ (deblokira) izvršavanje procesa koji je zadat argumentom

Rešenje:

```
void yield(PCB* current, PCB* next){
      asm{
            push R0                 //sve programski dostupne registre
            …                       //stavljamo na stek (osim SP-a)
            push RN
      }                             //čuvamo SP u PCB-u tekućeg procesa
      current->sp = _SP;            //_SP je pseudopromenljiva za SP
      _SP = next->sp;               //učitavamo SP iz PCB-a narednog procesa
      asm{
            pop RN                  //skidamo sa steka sačuvane programski
            …                       //dostupne registre (osim SP-a koji se
            pop R0                  //čuva u okviru PCB-a i već je "skinut")
      }
      return;                       //povratak iz potprograma, kontekst je promenjen
}

void suspend(){
      PCB* old = allProcesses[running];   //old pokazuje na PCB tekućeg procesa
      running = ready;                    //running postaje PID prvog ready procesa
      ready = allProcesses[ready]->next;  //ready postaje PID sledećeg spremnog procesa
      PCB* new = allProcesses[running];   //dohvatamo PCB novog running procesa
      allProcesses[running]->next=0;      //raskidamo njegovu vezu sa listom spremnih
      yield(old, new);                    //vršimo promenu konteksta
      return;                             //vraćamo se iz potprograma
}

void resume(PID pid){                           //ako je pid PID tekućeg procesa, onda
      int ind=(running != pid);                 //on nije uopšte blokiran (očigledno)
      for(PID i = 0;i < MaxProc;i++)            //ako je proces sa PID-om pid blokiran
            if (allProcesses[i].next == pid)    //onda nijedan proces u listi spremnih
                  ind=0;                        //ne pokazuje na njega
      if (ind){                                 //ako je proces stvarno blokiran (ind=1)
            allProcesses[pid]->next = ready;    //vršimo "odblokiranje" tako što
            ready = pid;                        //proces uvezujemo u listu spremnih
      }
```

```
}
```

15. (*April 2006*) U jezgru nekog operativnog sistema definisane se sledeće strukture podataka:

```
typedef unsigned int PID;
#define MaxProc  ...          // Max number of processes
struct PCB {
  PID pid;                    // Process ID
  jmp_buf context;
  PCB* next;                  // Next PCB in the Ready or a waiting list
  ...
};
PCB* allProcesses[MaxProc];   // Maps a PID to a PCB*
PCB* running;                 // The running process
PCB* ready;                   // Head of Ready list
```

Korišćenjem bibliotečnih funkcija setjmp() i longjmp(), na jeziku C napisati tela sledećih funkcija:

- `suspend()`: suspenduje (blokira) izvršavanje pozivajućeg procesa sve dok ga neki drugi proces ne „probudi“ pomoću `resume()`;
- `resume(PID pid)`: „budi“ (deblokira) izvršavanje procesa koji je zadat argumentom.

Rešenje:

Potreban nam je jedan pokazivač na listu PCB-ova blokiranih (suspendovanih) procesa:

```
PCB* blocked;
```

Na početku, lista blokiranih procesa je prazna, pa je:

```
blocked = null;
```

Kada se neka nit blokira usled poziva funkcije `suspend()` potrebno je da njen PCB ubacimo u listu PCB-ova blokiranih niti i ažuriramo pokazivač na prvi element te liste.

Pošto pri pozivu metode `suspend()` za blokiranje procesa, uvek tekući proces dodajemo na početak liste blokiranih procesa, ažuriranje pokazivača na listu blokiranih procesa se vrši tako što se jednostavno on postavi da pokazuje na tekući proces (jer je on dodat na početak liste).

Implementacije traženih funkcija:

```
//funkcija koja suspenduje tekući proces
void suspend () {
      lock ();                            //Kernel mode: zabranjujemo prekide
      if (setjmp(running->context)==0) {  //čuvamo kontekst tekućeg procesa

            running->next = blocked;      //dodajemo tekući proces u listu blokiranih
            blocked = running;            //i ažuriramo pokazivač na listu blokiranih

            running = ready;              //procesor dajemo prvom spremnom procesu
            ready = ready->next;          //ažuriramo listu spremnih procesa

            longjmp(running->context,1);  //restauiramo kontekst novog tekućeg procesa
      } else {
            unlock ();                    //User mode: dozvoljavamo prekide
            return;                       //povratak iz potprograma
      }
}
```

```
//funkcija koja budi proces sa PID-om pid
void resume(PID pid){
      if (blocked == 0) return;           //ako je lista blokiranih procesa prazna
                                          //vraćamo se, jer nemamo šta da odblokiramo

       lock();                            //Kernel mode: zabranjujemo prekide


      //pronalazi PCB procesa koji treba deblokirati na osnovu
      //njegovog PID-a prolaskom kroz listu blokiranih procesa
      PCB *prev = 0, *curr = blocked;
      while (curr != 0 && curr->pid != pid){
            prev = curr;
            curr = curr->next;
      }

      //ako curr ima vrednost null, proces sa PID-om pid ne postoji
      //u listi blokiranih procesa, pa se vraćamo iz potprograma
      if (curr == 0) {
            unlock();                     //User mode: dozvoljavamo prekide
            return;                       //povratak iz potprograma
      }


      //ako je odblokirani proces bio prvi u listi blokiranih
      if (prev == 0)
            blocked = blocked->next;      //ažuriramo pokazivač na početak liste
      else
            prev->next = curr->next;      //u suprotnom samo prevezujemo pkazivače


      curr->next = ready;                 // vraća nađeni PCB u listu spremnih
      ready = curr;                       //ažuriramo pokazivač na listu spremnih

      unlock();                           //User mode: dozvoljavamo prekide
}
```

16. (*April 2006*) U nekom operativnom sistemu postoji sistemska usluga kreiranja niti koja se iz jezika C/C++ poziva pomoću bibliotečne funkcije `create_thread()`:

```
typedef unsigned long PID;                          // Process ID
PID create_thread (void (*body)(void*), void* arg);
```

Ovaj sistemski poziv kreira nit nad C funkcijom na koju ukazuje prvi argument body, pri čemu se pozivu te funkcije kao argument dostavlja pokazivač `arg`.

Za ovaj operativni sistem realizuje se biblioteka za konkurentno izvršavanje C++ programa koja koncept niti podržava klasom `Thread`. Ova klasa realizuje koncept niti kao u školskom jezgru i ima sledeći interfejs:

```
class Thread {
public:
  void start ();              // Starts the thread
protected:
  Thread ();
  virtual void run() {}       // The body of the thread
private:
  PID pid;                    // PID of system thread
};
```

tako da se korisničke niti kreiraju kao u sledećem primeru:

```
class Robot : public Thread {
...
  virtual void run ();
};

Robot* r = new Robot;
r->start();
```

Preslikavanje niti iz korisničke aplikacije na niti operativnog sistema je jedan-na-jedan. Korišćenjem navedenog sistemskog poziva, realizovati operaciju Thread::start() (uz odgovarajuće dodatne delove koda, ako je potrebno).

Rešenje:

U `protected` sekciju klase `Thread` potrebno je dodati sledeći metod:

```
class Thread{                                  //funkciji create_thread ne možemo da
      …                                        //prosledimo nestatičku metodu run() klase
      protected:                               //Thread, pa zbog toga koristimo pomoćnu
        static void starter(void*);            //statičku metodu starter, istog potpisa kao
      …                                        //i funkcija koju prosleđujemo funkciji
};                                             //create_thread preko pokazivača body

void Thread::starter(void* toStart){
      Thread* t = (Thread*)toStart;            //toStart je prosleđeni pokazivač this
      if (t) t->run();                         //pozivamo metodu run() tekućeg objekta
}                                              //čime pokrećemo izvršavanje operacije niti
```

U tom slučaju implementacija metoda `start` izgleda:

```
void Thread::start(){
      pid = create_thread(&starter, this);      //kreiramo novu nit, i njen PID čuvamo
}                                               //u atributu pid klase Thread
```

Четврта група задатака: Синхронизација и комуникација између процеса

1. (*Maj 2011*) Dva uporedna kooperativna procesa A i B razmenjuju podatke tako što proces A izračunava vrednost deljene promenljive a pozivom svoje privatne procedure `compute_a`. Tu vrednost promenljive a koristi proces B tako što na osnovu nje izračunava vrednost deljene promenljive b pozivom svoje privatne procedure `compute_b`. Tu vrednost promenljive b potom koristi proces A za izračunavanje nove vrednosti promenljive a i tako ciklično. Korišćenjem standardnih brojačkih semafora napisati kod procesa A i B sa svim neophodnim deklaracijama. Inicijalno, proces A može odmah da izračuna početnu vrednost za a na osnovu statički inicijalizovane vrednosti za b, dok proces B ne može da izračuna prvu vrednost b dok ne dobije prvu izračunatu vrednost a.

Rešenje:

```
shared var
   a, b : integer := 0;       //statička inicijalizacija deljenih promenljivih
   sa, sb : semaphore := 0;   //statička inicijalizacija semafora

//napomena: oba semafora su na početku inicijalizovana na nulu, što ima za posledicu
//da proces B ne može na početku da izračuna novu vrednost promenljive b, pre nego
//što proces A prvi put ne izračuna novu vrednost promenljive a i pozove sa.signal()
//isto tako, proces A ne može da izračuna drugu novu vrednost promenljive a, pre nego
//što proces B izračuna prvu novu vrednost promenljive b

process a;
begin
  //vrtimo se u petlji i stalno izvršavamo sledeće:
  loop
      //računamo vrednost promenljive a na osnovu nove vrednosti promenljive b
      //napomena: promenljiva b na početku nema novu vrednost, nego statički
      //inicijalizovanu, ali to je dozvoljeno, jer proces A može na početku da izračuna
      //novu vrednost promenljive a na osnovu statički inicijalizovane vrednosti za b
      compute_a(b);

      //signaliziramo procesu B da je nova vrednost promenljive a izračunata
      sa.signal();

      //čekamo da proces B izračuna novu vrednost promenljive b
      sb.wait();
 end;
end;

process b;
begin
  //vrtimo se u petlji i stalno izvršavamo sledeće:
  loop
      //čekamo da proces A izračuna novu vrednost promenljive a
      sa.wait();

      //računamo vrednost promenljive b na osnovu nove vrednosti promenljive a
      compute_b(a);

      //signaliziramo procesu A da je nova vrednost promenljive b izračunata
      sb.signal();
 end;
end;
```

2. (*Maj 2011*) U klasi `Semaphore` postoji privatni podatak član `isLocked` tipa `int` koji služi da obezbedi međusobno isključenje pristupa strukturi semafora u višeprocesorskom operativnom sistemu. Ako je njegova vrednost 1, kod koji se izvršava na nekom procesoru je zaključao semafor za svoj isključivi pristup; ako je vrednost 0, pristup semaforu je slobodan. Korišćenjem operacije `test_and_set()` koja je implementirana korišćenjem odgovarajuće atomične instrukcije procesora, realozovati operaciju `Semaphore::lock()` koja treba da obezbedi međusobno isključenje pristupa strukturi semafora u višeprocesorskom operativnom sistemu, ali tako da bude efikasnija od dole date implementacije tako što ne poziva operaciju `test_and_set` ako je semafor već zaključan od strane drugog procesora, pošto ta operacija ima veće režijske troškove na magistrali računara.

```
void Semaphore::lock () {
  while (test_and_set(this->isLocked));
}
```

Rešenje:

```
void Semaphore::lock () {
   //dok god ne zaključamo semafor
   for (int acquired = 0; !acquired;) {
       //ako je semafor već zaključan od strane drugog procesa,
       //čekamo da taj proces oslobodi (otključa) pristup semaforu

       while (this->isLocked);
       //na ovu liniju dolazimo samo ako smo pre toga detektovali da je
       //semafor otključan, ali to ne znači da u međuvremenu nije uleteo
       //neki drugi proces i zaključao semafor pre nego što smo mi to uradili
       //(iz tog razloga nam je i potrebna for petlja)
       //zato pokušavamo da zauzmemo semafor (isLocked je onda još uvek nula,
       //pa !test_and_set(this->isLocked) vraća 1 i izlazimo iz for petlje)
       //ako je neki drugi proces u međuvremenu opet zauzeo semafor, on je postavio
       //isLocked na 1, pa !test_and_set(this->isLocked) vraća 0 i nastavljamo
       //da se vrtimo u for petlji, sve dok se semafor opet ne oslobodi i
       //pokušamo ponovo da ga zauzmemo

       acquired = !test_and_set(this->isLocked);
   }
}
```

napomena: čim `acquired` postane 1, to znači da je pozivajući proces uspešno zauzeo semafor (zaključao ga)

3. (*Septembar 2011*) Korišćenjem standardnih brojačkih semafora rešiti problem utrkivanja (engl. *race condition*) u kodu datog programa koji simulira rad policijskog helikoptera i automobila, a koji je objašnjen na predavanjima.

```
type Coord = record {
  x : integer;
  y : integer;
};

var sharedCoord : Coord;

process Helicopter
var nextCoord : Coord;
begin
  loop
      computeNextCoord(nextCoord);
      sharedCoord := nextCoord;
  end;
end;

process PoliceCar
begin
  loop
      moveTo(sharedCoord);
  end;
end;
```

Rešenje:

```
type Coord = record {
      x : integer;
      y : integer;
};

var sharedCoord : Coord;
    //helikopter čeka na ovom semaforu da policijska kola pročitaju koordinate
    //vrednost semafora je na početku 1, jer helikopter mora da izračuna početne
    //koordinate koje će proslediti policijskim kolima (da je 0, ostao bi blokiran)
    readyToWrite : Semaphore = 1;
    //policijska kola čekaju na ovom semaforu da helikopter upiše nove koordinate
    readyToRead : Semaphore = 0;


process Helicopter
var nextCoord : Coord;
begin
   loop
      //izračunavamo nove koordinate
      computeNextCoord(nextCoord);
      //čekamo da policijski automobil sačuva prethodne koordinate
      readyToWrite.wait();
      //tek pošto policijski automobil sačuva prethodne koordinate
      //možemo da ih ažuriramo novim (jer ova dodela nije atomična)
      sharedCoord := nextCoord;
      //signaliziramo policijskom automobilu da su nove koordinate
      //spremne i da može da ih pročita
      readyToRead.signal();
   end;
end;
```

```
process PoliceCar
var nextCoord : Coord;
begin
   loop
      //čekamo da policijski helikopter izračuna nove koordinate
      readyToRead.wait();
      //ne znamo koliko može da traje izvršavanje funkcije moveTo
      //pa koristimo pomoćnu promenljivu da sačuvamo nove koordinate
      //čime smo "oslobodili" deljenu promenljivu, pa helikopter može
      //da izračuna nove koordinate dok policijska kola izvršavaju moveTo
      nextCoord := sharedCoord;
      //signaliziramo policijskom helikopteru da nam deljena promenljiva
      //više nije potrebna i da može da upiše novu vrednost u nju
      readyToWrite.signal();
      //policijska kola se pomeraju na nove koordinate (koristimo pomoćnu promenljivu)
      moveTo(nextCoord);
   end;
end;
```

4. (*Septembar 2011*) Izmeniti datu implementaciju operacije `wait` na semaforu u školskom jezgru tako da, pre nego što odmah blokira pozivajuću nit ukoliko je semafor zatvoren, najpre pokuša da uposleno sačeka, ali ograničeno, ponavljajući petlju čekanja najviše `SemWaitLimit` puta. Ostatak klase `Semaphore` se ne menja.

```
void Semaphore::wait () {
  lock(lck);
  if (--val<0)
      block();
  unlock(lck);
}
```

Rešenje:

```
void Semaphore::wait () {
      //ako je vrednost semafora manja ili jednaka nuli, nit će se blokirati
      //na njemu, tako da čekamo dok vrednost semafora ne postane veća od nule
      //vrteći se u praznoj for petlji ili dok ne istekne maksimalno vreme
      //ovog čekanja (kada se udje u petlju najviše SemWaitLimit puta)
      for (int i=0; val<=0 && i<SemWaitLimit; i++);

      //zaključavamo semafor
      lock(lck);

      //ako je vrednost semafora manja od nule, blokiramo pozivajući proces
      if (--val<0)
            block();

      //otključavamo semafor
      unlock(lck);
}
```

važna napomena: `for` petlja mora da se nalazi pre `lock(lck)`, jer se nakon te instrukcije semafor zaključava pa nijedna druga nit ne može da pozove operaciju `signal()` nad ovim istim semaforom i tako spreči blokiranje niti koja je pozvala `wait()` i uposleno čeka.

5. (*Maj 2010*) Kreirano je više procesa istog tipa *P* koji imaju istu sledeću strukturu. U kritičnoj sekciji *A* nalaze se ugnežđene dve kritične sekcije *B* i *C*, s tim da se sekcije *B* i *C* izvršavaju jedna posle druge (nisu ugnežđene). Potrebno je obezbediti sledeću sinhronizaciju: u kritičnoj sekciji *A* može se u jednom trenutku nalaziti najviše *N* ovih procesa tipa *P*, dok se u sekciji *B*, odnosno *C* može nalaziti najviše jedan proces. (Svaka od sekcija *B* i *C* je međusobno isključiva, ali se *B* i *C* ne isključuju međusobno – jedan proces može biti u *B* dok je drugi u *C*.) Korišćenjem standardnih brojačkih semafora obezbediti ovu sinhronizaciju: prikazati strukturu tipa procesa *P*, uz odgovarajuće definicije i inicijalizacije potrebnih semafora.

Rešenje:

```
            //semafor na kome čekaju procesi koji žele da uđu u kritičnu sekciju A
            //vrednost semafora je na početku N, jer toliko procesa može da se nađe
            //u istom trenutku u kritičnoj sekciji A
shared var  mutexA: semaphore:=N;
            //semafor na kome čekaju procesi koji žele da uđu u kritičnu sekciju B
            //vrednost semafora je na početku 1, jer samo jedan proces može da se
            //u nekom trenutku nađe u kritičnoj sekciji B
            mutexB: semaphore:=1;
            //semafor na kome čekaju procesi koji žele da uđu u kritičnu sekciju C
            //vrednost semafora je na početku 1, jer samo jedan proces može da se
            //u nekom trenutku nađe u kritičnoj sekciji C
            mutexC: semaphore:=1;

type P  = process begin

  . . .

  wait(mutexA);                     //poziv operacije wait nad semaforom mutexA,
      <critical section A>          //jer želimo da uđemo u kritičnu sekciju A

      . . .

      wait(mutexB);                 //poziv operacije wait nad semaforom mutexB,
      <critical section B>          //jer želimo da uđemo u kritičnu sekciju B
      <end of critical section B>
      signal(mutexB);               //izlazimo iz kritične sekcije B, pa pozivamo
                                    //operaciju signal nad semaforom mutexB
      . . .

      wait(mutexC);                 //poziv operacije wait nad semaforom mutexC,
      <critical section C>          //jer želimo da uđemo u kritičnu sekciju C
      <end of critical section B>
      signal(mutexC);               //izlazimo iz kritične sekcije C, pa pozivamo
                                    //operaciju signal nad semaforom mutexC
      . . .

      <end of critical section A>
  signal(mutexA);                   //izlazimo iz kritične sekcije A, pa pozivamo
                                    //operaciju signal nad semaforom mutexA
  . . .

end;
```

6. (*Maj 2009*) Realizaciju semafora pomoću C++ klase `Semaphore` date na predavanjima modifikovati tako da se promena konteksta može dogoditi i kod neblokirajućih poziva `wait` i `signal` na semaforu (tj. pri svakom pozivu `signal`, kao i pri pozivu `wait` i kada ne treba blokirati pozivajući proces).

Rešenje:

```
void Semaphore::wait () {
   //zaključavamo semafor
   lock();
   //preuzimanje vršimo čak i kad se nit ne blokira na semaforu,
   //dakle uvek, pa u svakom slučaju treba da sačuvamo njen kontekst
   if (setjmp(Thread::runningThread->context)==0) {
      //ako je nit blokirana na semaforu, stavljamo je u red blokiranih
      if (--val<0)
         blocked.put(Thread::runningThread);
      //u suprotnom je stavljamo u red spremnih niti (u Scheduler)
      else
         Scheduler::put(Thread::runningThread);

      //dohvatamo jednu spremnu nit iz Scheduler-a
      Thread::runningThread = Scheduler::get();
      //i restauiramo njen kontekst
      longjmp(Thread::runningThread->context,1);
  }
  //otključavamo semafor
  unlock();
}

void Semaphore::signal () {
   //zaključavamo semafor
   lock();

   //ako je val bilo manje od nule, postoji bar jedna nit blokirana na
   //semaforu, pa je uzmamo iz reda blokiranih i stavljamo u red spremnih
   if (val++<0)
       Scheduler::put(blocked.get());

   //promenu konteksta treba da vršimo na svaki poziv operacije signal,
   //pa u svakom slučaju treba da sačuvamo kontekst tekuće (pozivajuće) niti
   if (setjmp(Thread::runningThread->context)==0) {

       //stavljamo pozivajuću nit u red spremnih
       Scheduler::put(Thread::runningThread);
       //dohvatamo jednu spremnu nit iz Scheduler-a
       Thread::runningThread = Scheduler::get();
       //i restauiramo njen kontekst
       longjmp(Thread::runningThread->context,1);
   }
   //otključavamo semafor
   unlock();
}
```

7. (*Maj 2009*) Dat je kod dva tipa procesa A i B:

```
shared var count : integer := 0;
            mutex : semaphore:=1;
            gate : semaphore:=1;

type A  = process begin
  ...
  wait(mutex);
    count:=count+1;
    if (count=1) then wait(gate);
  signal(mutex);
  ...  (*  Critical section A *)
  wait(mutex);
    count:=count-1;
    if (count=0) then signal(gate);
  signal(mutex);
  ...
end;

type B  = process begin
  ...
  wait(gate);
  ...  (*  Critical section B *)
  signal(gate);
  ...
end;
```

Ukoliko postoji proizvoljno, ali konačno mnogo aktivnih procesa tipa A i procesa tipa B, precizno odgovoriti i objasniti koliko *istovremeno* može biti procesa u označenim kritičnim sekcijama.

Rešenje:

Mogu se desiti dva različita slučaja:

1. proizvoljno mnogo procesa tipa A u kritičnoj sekciji A i nijedan proces tipa B u kritičnoj sekciji B
2. jedan proces tipa B u kritičnoj sekciji B i nijedan proces tipa A u kritičnoj sekciji A

Prvi slučaj se dešava kada proces tipa A pozove `wait(gate)` pre nego što uradi neki proces tipa B, čime blokira ulazak u kritičnu sekciju B procesima tipa B, a on sam se ne blokira, pa može da uđe u kritičnu sekciju A. Isto tako, pre nego što je taj proces tipa A ušao u kritičnu sekciju A, morao je da pozove `signal(mutex)` čime je omogućio drugim procesima tipa A da napreduju i kasnije uđu u kritičnu sekciju A (oni se ne blokiraju, jer uslov `if (count=1)` nije ispunjen, pa uopšte ne pozivaju `wait(gate)` koji bi ih blokirao).

Drugi slučaj se dešava kada proces tipa B pozove `wait(gate)` pre nego što to uradi neki proces tipa A, čime on ulazi u kritičnu sekciju B, a ujedno i blokira ostalim procesima oba tipa da uđu u sopstvene kritične sekcije (proces tipa B će se blokirati pri pozivu `wait(gate)`, kao i proces tipa A; odatle sledi da blokirani proces tipa A neće moći da pozove `signal(mutex)` i time odblokira ulaz ostalim procesima tipa A koji će ostati blokirani pri pozivu `wait(mutex)`).

8. (*Maj 2008*) U jezgru nekog multiprocesorskog operativnog sistema realizovane su sledeće operacije koje se koriste za obezbeđenje atomičnosti operacija nad semaforom:

```
void Semaphore::lock (int* lck) {
  disable_interrupts();
  while test_and_set(lck);
}

void Semaphore::unlock (int* lck) {
  *lck=0;
  enable_interrupts();
}
```

Koncept semafora realizovan je klasom `Semaphore`, čiji su fragmenti prikazani u nastavku:

```
class Semaphore {
...
private:
  static int commonLock;      // initialized to 0
  int myLock;                 // initialized to 0
...
};
```

Komentarisati razliku između sledeće dve realizacije operacija nad semaforom, odnosno njihovog ulaznog i izlaznog protokola sa ciljem obezbeđenja atomičnosti:

```
lock(&commonLock);                  lock(&myLock);
...                                 ...
unlock(&commonLock);                unlock(&myLock);
```

Posebno komentarisati sledeće aspekte:

- da li su obe varijante sasvim korektne i pod kojim uslovima?
- stepen paralelizma koji ove varijante omogućavaju.

Rešenje:

Ako se neki semafor zaključa prvom varijantom ulaznog protokola, to će imati za posledicu da nijedan drugi proces neće moći da uđe u kod operacije bilo kog drugog semafora, jer je promenljiva nad kojom vršimo zaključavanje statička, što znači da je zajednička za celu klasu, odnosno za sve objekte klase `Semaphore`.

Ako se neki semafor zaključa drugm varijantom ulaznog protokola, to će imati za posledicu da nijedan drugi proces neće moći da uđe u kod operacije tog zaključanog semafora, ali će zato moći da uđe u kod operacije bilo kog drugog (nezaključanog) semafora. Ovo je posledica toga što je promenljiva nad kojom vršimo zaključavanje nestatički atribut klase `Semaphore` što znači da svaki objekat te klase poseduje svoju vlastitu kopiju.

Odatle slede sledeće činjenice:

- ukoliko kod operacija nad semaforom ne uzrokuje konflikte na deljenim strukturama koje se koriste u izvršavanjima na različitim procesorima, onda su obe varijante korektne. U suprotnom, ukoliko ovaj kod uzrokuje ovakve konflikte, npr. korišćenjem istog reda spremnih procesa za različite procesore bez obezbeđenja međusobnog isključenja nad tom strukturom, onda je samo prva varijanta korektna (pod uslovom da ne pravi konflikte sa drugim uslugama operativnog sistema), a druge ne.
- druga varijanta omogućava znatno veći stepen paralelizma, jer ne zaustavlja napredovanje operacije nad jednim semaforom na jednom procesoru zbog toga što je drugi procesor ušao u operaciju nad drugim semaforom, kako to čini prva varijanta.

9. (*Maj 2008*) Jedan proces-proizvođač proizvodi elemente i stavlja ih u ograničeni bafer, dok dva procesa-potrošača uzimaju elemente iz tog bafera. Potrebno je obezbediti odgovarajuću sinhronizaciju između ovih procesa tako da potrošači naizmenično uzimaju po jedan element iz bafera. Realizovati ograničeni bafer koji obezbeđuje ovu sinhronizaciju pomoću klasičnih brojačkih semafora.

Rešenje:

```
class Data;
const int N = ...;            //kapacitet bafera

class BoundedBuffer {
public:
  BoundedBuffer();            //konstruktor
  void  put (Data*);
  Data* get (int consumerID); //consumerID treba da bude 1 ili 2 (ID potrošača)
private:
  Data* buf[N];               //red elemenata ograničenog bafera
  int head, tail;             //početak (glava) i kraj (rep) reda
                              //semafori sa sinhronizaciju
  Semaphore mutex, spaceAvailable, itemAvailable, gate1, gate2;
};

BoundedBuffer::BoundedBuffer () :
  head(0), tail(0),
  mutex(1), spaceAvailable(N), itemAvailable(0),
  gate1(1), gate2(0) {}

void BoundedBuffer::put (Data* d) {
      spaceAvailable.wait();     //ako je bafer pun, čekamo jedno slobodno mesto
      mutex.wait();              //ulazimo u kritičnu sekciju (da li je potreban mutex?)
      buf[tail]=d;               //stavljamo element u bafer
      tail=(tail+1)%N;           //ažuriramo indeks poslednjeg elementa
      mutex.signal();            //izlazimo iz kritične sekcije
      itemAvailable.signal();    //signaliziramo da je još jedan podatak raspoloživ
}
Data* BoundedBuffer::get (int myID) {
      if (myID==1)               //potrošač 1 se blokira na semaforu gate1 ako nije
            gate1.wait();        //njegov red da preuzme jedan element iz bafera
      else if (myID==2)          //potrošač 2 se blokira na semaforu gate2 ako nije
            gate2.wait();        //njegov red da preuzme jedan element iz bafera
      else
            return 0;            //myID je nedozvoljen ID potrošača
      itemAvailable.wait();      //ako je bafer prazan, čekamo dok se ne upiše podatak
      mutex.wait();              //ulazimo u kritičnu sekciju
      Data* d = buf[head];       //čitamo podatak iz bafera
      head=(head+1)%N;           //ažuriramo indeks prvog elementa
      mutex.signal();            //izlazimo iz kritične sekcije
      spaceAvailable.signal();   //signaliziramo da ima slobodnog prostora u baferu
      if (myID==1)               //ako je potrošač 1 uzeo element iz bafera
            gate2.signal();      //onda je na redu potrošač 2 (p2 se blokira na gate2)
      else if (myID==2)          //u suprotnom, ako je potrošač 2 uzeo element iz bafera
            gate1.signal();      //onda je na redu potrošač 1 (p1 se blokira na gate2)
      return d;                  //vraćamo pročitani podatak
}
```

10. (*Maj 2007*) Na raspolaganju su sledeće funkcije:

- `swap(int*,int*)` koja, korišćenjem odgovarajuće mašinske instrukcije, atomično zamenjuje vrednosti dve memorijske reči na adresama zadatim argumentima (C/C++ tip `int` je uvek veličine mašinske reči).
- `set_interrupts(int)` koja maskira sve nemaskirajuće prekide ako je argument jednak 0, odnosno demaskira ih ako je argument jednak 1.

Korišćenjem ovih funkcija realizovati standardni brojački semafor (klasu `Semaphore`) sa operacijama `wait()` i `signal()`, pri čemu se koristi uposleno čekanje umesto suspenzije (blokiranja) pozivajućeg procesa. Realizacija treba da bude prilagođena multiprocesorskom sistemu sa preuzimanjem (*preemptive*).

Rešenje:

```
class Semaphore {
   public:
     Semaphore  (unsigned int value=1);    //konstruktor
     void wait  ();
     void signal();

   protected:
     inline void lock();                   // inline nije semantički bitan
     inline void unlock();                 // inline nije semantički bitan

   private:
     int lock;                             //1-zaključan, 0-otključan
     unsigned int val;                     //vrednost semafora
};

Semaphore::Semaphore (unsigned int v) : lock(0), val(v) {}

void Semaphore::lock () {
  set_interrupts(0);                       //maskiramo prekide
  for (int lck=1; lck;) swap(&lck,&lock);  //ako je semafor već zaključan
                                           //čekamo da se otključa,
}                                          //a zatim ga sami zaključavamo

void Semaphore::unlock () {
  lock=0;                                  //0-semafor je otključan
  set_interrupts(1);                       //dozvoljavamo prekide
}

void Semaphore::wait () {
  for (int done=0; !done; ) {              //ako je poziv "blokirajući"
      while(val==0);                       //čekamo da neko pozove signal()
      lock();                              //zaključavamo semafor
      if (val==0) { unlock(); continue; }  //ako je  međuvremenu drugi proces
                                           //"uleteo" i izvršio wait(), opet čekamo
      done=1;                              //u suprotnom, proces se ne blokira pa
      val--;                               //dekrementiramo vrednost semafora
      unlock();                            //i otključavamo semafor
  }
}

void Semaphore::signal () {
  lock();                     //zaključavamo semafor
  val++;                      //inkrementiramo vrednost semafora
                              //procesi se nikad ne blokiraju, pa ne treba unblock()
  unlock();                   //otključavamo semafor
```

```
}
```

11. (*Maj 2007*) Dat je sledeći interfejs klase koja realizuje ograničeni bafer (*bounded buffer*) kao posrednik između samo jednog proizvođača (*producer*) i samo jednog potrošača (*consumer*) koji su uporedne niti:

```
class BoundedBuffer {
public:
  BoundedBuffer(int capacity);
  void put (char* package, int size);
  void get (char* package, int size);
};
```

U bafer se smeštaju znakovi (`char`). Operacija `put` smešta u bafer dati niz znakova date dužine. Operacija `get` iz bafera uzima `size` znakova i smešta ih u niz na koji ukazuje pokazivač `package`. Ovi nizovi (paketi koji se prenose) su proizvoljne i promenljive dužine (ali svakako manje od kapaciteta bafera). Implementirati ovu klasu korišćenjem standardnih brojačkih semafora.

Rešenje:

```
class BoundedBuffer {
   public:
     BoundedBuffer(int capacity);
     void put (char* package, int size);
     void get (char* package, int size);
   private:
     char* buf;                                  //red elemenata
     int capacity;                               //kapacitet bafera
     int head, tail;                             //početak (glava) i kraj (rep)
     Semaphore spaceAvailable, itemAvailable;    //semafori za sinhronizaciju
};

//konstruktor
BoundedBuffer::BoundedBuffer (int cap) :
  buf(new char[cap]), capacity(cap), head(0), tail(0),
  spaceAvailable(cap), itemAvailable(0) {}

void BoundedBuffer::put (char* p, int n) {
  for (int i=0; i<n; i++) {          //za svaki znak iz niza p
      spaceAvailable.wait();         //ako je bafer već pun, čekamo da se
                                     //jedno mesto oslobodi (get() da pozove signal)
      buf[tail]=p[i];                //ubacujemo element na kraj reda
      tail=(tail+1)%capacity;        //ažuriramo indeks poslednjeg elementa u redu

      itemAvailable.signal();        //signaliziramo da je još jedan podatak raspoloživ
  }
}

void BoundedBuffer::put (char* p, int n) {
  for (int i=0; i<n; i++) {          //za svaki znak iz niza p
      itemAvailable.wait();          //ako je bafer prazan, čekamo da se bar jedan
                                     //znak upiše u bafer (put() da pozove signal)
      p[i]=buf[head]                 //uzimamo znak sa početka reda
      head=(head+1)%capacity;        //ažuriramo indeks prvog elementa u redu

      spaceAvailable.signal();       //signaliziramo da ima slobodnog prostora u baferu
  }
}
```

napomena: ne trebaju nam `mutex`-i za deo koda gde se ažuriraju indeksi i čita/upisuje znak, jer postoje samo po jedan potrošač i proizvođač (kritična sekcija se neće narušiti)

Пета група задатака: Управљање меморијом

1. (*Maj 2011*) U potpunosti realizovati klasu koja apstrahuje dinamički niz elemenata tipa `double` velike dimenzije zadate prilikom inicijalizacije parametrom `size`. U svakom trenutku se u memoriji drži samo jedan keširani blok ovog niza veličine zadate parametrom `blockSize`. Implementacija treba da koristi dinamičko učitavanje bloka u kome se nalazi element kome se pristupa uz zamenu (*swapping*), zapravo neku vrstu preklopa (*overlay*), tako da se u memoriji uvek nalazi samo jedan keširani blok kome se trenutno pristupa, dok se ceo niz nalazi u fajlu. Interfejs ove klase treba da bude:

```
class DLArray {
public:
  DLArray (int size, int blockSize, FHANDLE fromFile);
  double get (int i);                                  // Get element [i]
  void   set (int i, double x);                        // Set element [i]
};
```

Inicijalno se vrednosti celog niza nalaze u binarnom fajlu zadatom trećim argumentom (otvaranje i zatvaranje tog fajla je u odgovornosti korisnika ove klase). Na raspolaganju su sledeće funkcije za pristup fajlu i učitavanje, odnosno snimanje datog niza elemenata tipa double na zadatu poziciju u fajlu. Pozicija se izražava u jedinicama veličine binarnog zapisa tipa `double`, počev od 0, što znači da se fajl kroz ove funkcije može posmatrati kao veliki niz elemenata tipa double:

```
void fread (FHANDLE, int position, double[], int bufferSize);
void fwrite(FHANDLE, int position, double[], int bufferSize);
```

Rešenje:

```
class DLArray {
public:
  inline DLArray (int size, int blockSize, FHANDLE fromFile);

  inline double get (int i);  // Get element [i]
  inline void   set (int i, double x); // Set element [i]

protected:
  inline void save();
  inline void load(int blockNo);
  inline void fetch(int blockNo);

private:
  FHANDLE file;                    //datoteka u kojoj je smešten niz
  int size, blockSize;             //veličina celog niza i učitanog bloka
  int curBlock;                    //indeks trenutno učitanog bloka
  int dirty;                       //da li je trenutno učitani blok menjan
  double* block;                   //trenutno učitani blok
};

//konstruktor
DLArray::DLArray (int s, int bs, FHANDLE f) :
  file(f), size(s), blockSize(bs), curBlock(0), dirty(0) {
  block = new double[bs];     //odvajamo prostor za trenutno učitani blok

  if (block)                  //ako smo uspešno odvojili prostor,
      load(curBlock);         //učitavamo prvi blok sa diska
}

//funkcija koja čuva trenutni blok nazad na disk (u datoteku)
void DLArray::save () {
  fwrite(file,curBlock*blockSize,block,blockSize);    //upisujemo blok na disk
```

```
  dirty=0;                                          //ažuriramo fleg "neizmenjenosti"
}
```

```
//funkcija koja učitava traženi blok sa diska (iz datoteke)
void DLArray::load (int b) {
  curBlock = b;                                          //ažuriramo indeks učitanog bloka
  fread(file,curBlock*blockSize,block,blockSize);        //učitavamo blok iz datoteke

  dirty = 0;                                             //ažuriramo fleg "neizmenjenosti"
}

//dohvatanje bloka ako već nije učitan
void DLArray::fetch(int b) {
  if (curBlock!=b) {                                     //ako traženi blok nije učitan
    if (dirty)                                           //ako je trenutni blok izmenjen,
      save();                                            //čuvamo ga na disku (datoteci),
    load(b);                                             //a onda učitavamo traženi blok
  }
}

//dohvatanje elementa niza
double DLArray::get (int i) {
  if (block==0 || i<0 || i>=size) return 0;    //greška je ako nije odvojen prostor za
                                               //niz ili ako indeks elementa nije validan
  fetch(i/blockSize);                          //dohvatamo blok koji sadrži taj element
  return block[i%blockSize];                   //vraćamo vrednost traženog elementa
}

//izmena vrednosti elementa niza
void DLArray::set (int i, double x) {
  if (block==0 || i<0 || i>=size) return;      //greška je ako nije odvojen prostor za
                                               //niz ili ako indeks elementa nije validan
  fetch(i/blockSize);                          //dohvatamo blok koji sadrži taj element
  if (block[i%blockSize]!=x) {                 //ako je vrednost elementa različita od
      block[i%blockSize]=x;                    //nove vrdnosti, menjamo je i ažuriramo
      dirty=1;                                 //fleg "neizmenjenosti"
  }
}
```

2. (*Maj 2011*) U nekom sistemu sa straničnom organizacijom virtuelne memorije koristi se *copy-on-write* tehnika deljenja stranica između procesa. U deskriptoru stranice u tabeli preslikavanja stranica (PMT) procesa nalaze se samo sledeće informacije: broj okvira u koji se stranica preslikava (0 označava da stranica nije u memoriji, okvir broj 0 se ne dodeljuje procesima) i biti prava pristupa (*R*-dozvoljeno čitanje u fazi izvršavanja instrukcije/dohvatanja operanda, *W*-dozvoljen upis, *E*-dozvoljeno čitanje u fazi dohvatanja instrukcije). Pored PMT, operativni sistem za svaki proces vodi posebnu strukturu podataka koju naziva *VMStruct* i u kojoj se čuvaju podaci koje koristi operativni sistem, a koji nisu potrebni hardveru za preslikavanje adresa. Svaki deskriptor u ovoj strukturi opisuje čitav skup susednih stranica koje predstavljaju logičku celinu, jer su svi ovi podaci za njih isti, i koji se u ovom kontekstu naziva *region*. U ovom deskriptoru čuvaju se sledeći podaci: prva stranica u skupu stranica na koje se deskriptor odnosi (*Start Page#*), broj susednih stranica na koje se deskriptor odnosi (*Region Length*), biti prava pristupa na logičkom nivou (*R*-dozvoljeno čitanje u fazi izvršavanja instrukcije/dohvatanja operanda, *W*-dozvoljen upis, *E*-dozvoljeno čitanje u fazi dohvatanja instrukcije), da li su stranice u ovom regionu deljene tako da ih treba kopirati pri prvom upisu (*Copy-On-Write*: 0-nisu deljene, 1-jesu deljene i treba ih kopirati pri upisu, samo ako je upis dozvoljen na logičkom nivou), kao i mesto na disku gde se zamenjuju stranice iz tog regiona (nije relevantno za ovaj zadatak). Posmatra se proces *Parent* čiji su delovi struktura *PMT* i *VMStruct* dati u nastavku.

PMT

Page#	*Frame#*	*RWE*
A04h	23h	?
BF0h	14h	?
C0Ah	7Ah	?

VMStruct

StartPage #	*Region Length*	*RWE-Copy-On-Write*	*Opis*
A00h	50h	001-0	*Code Region*
B00h	FFh	110-0	*Data Region*
C00h	70h	100-0	*Input Buffer Region*

Ovaj proces izvršava sistemski poziv fork() i uspešno kreira proces potomak *Child*. Prikazati iste delove struktura *PMT* i *VMStruct* za oba procesa *Parent* i *Child* neposredno nakon ovog poziva. Sistem primenjuje *demand paging* strategiju (dohvata stranicu tek kada se prvi put zatraži, ne alocira je odmah pri kreiranju procesa).

Rešenje:

Proces roditelj i proces dete će, nakon poziva `fork()` imati identične delove struktura *PMT* i *VMStruct,* koji će izgledati kao na slici:

PMT

Page#	*Frame#*	*RWE*
A04h	23h	001
BF0h	14h	100
C0Ah	7Ah	100

VMStruct

StartPage#	*Region Length*	*RWE-Copy-On-Write*	*Opis*
A00h	50h	001-0	*Code Region*
B00h	FFh	110-1	*Data Region*
C00h	70h	100-0	*Input Buffer Region*

Napomena: Primetiti da se u opisanom sistemu bit Copy-On-Write (u daljem tekstu CoW) odnosi na čitav set stranica. Zbog toga ovaj bit nije dovoljan da bi se odredilo da li pri upisu u neku stranicu iz seta, tu stranicu treba kopirati. Stranice koje pri upisu stvarno treba kopirati su one koje pripadaju regionu za koji je postavljen bit CoW i kojima je u trenutku upisa bitom W u PMT zabranjen upis (upisana je 0 u bit W). Kada jednom dođe do upisa i samim tim i do kopiranja stranice, za novu kopiju se setuje bit W u PMT i pri kasnijim upisima u tu stranicu ne treba vršiti kopiranje te stranice bez obzira što stranica pripada regionu za koji je setovan CoW bit. Stoga bi se sistem mogao implementirati tako da se u CoW bit jednom upiše vrednost pri pokretanju procesa i više nikada ne menja. U takvoj varijanti, CoW bit bi čak bio suvišan jer bi uvek imao istu vrednost kao i W bit u VMStruct strukturi.

3. (*Septembar 2011*) Klasa `GeoRegion`, čiji je interfejs dat u nastavku, apstrahuje geografski region i implementirana je u potpunosti. Objekti ove klase zauzimaju mnogo prostora u memoriji, pa se mogu učitavati po potrebi, dinamički. Ovo učitavanje obavlja statička operacija `GeoRegion::load`, pri čemu se objekat identifikuje datim nazivom geografskog regiona (niz znakova). Ostale operacije ove klase vraćaju vrednosti nekih svojstava geografskog regiona.
Potrebno je u potpunosti implementirati klasu `GeoRegionProxy` čiji je intefejs dat u nastavku. Objekti ove klase služe kao posrednici (*proxy*) do objekata klase `GeoRegion`, pri čemu pružaju isti interfejs kao i „originali“, s tim da skrivaju detalje implementacije i tehniku dinamičkog učitavanja od svojih korisnika. Korisnici klase `GeoRegionProxy` vide njene objekte na sasvim uobičajen način, mogu ih kreirati datim konstruktorom i pozivati date operacije interfejsa, ne znajući da odgovarajuća struktura podataka možda nije učitana u memoriju.

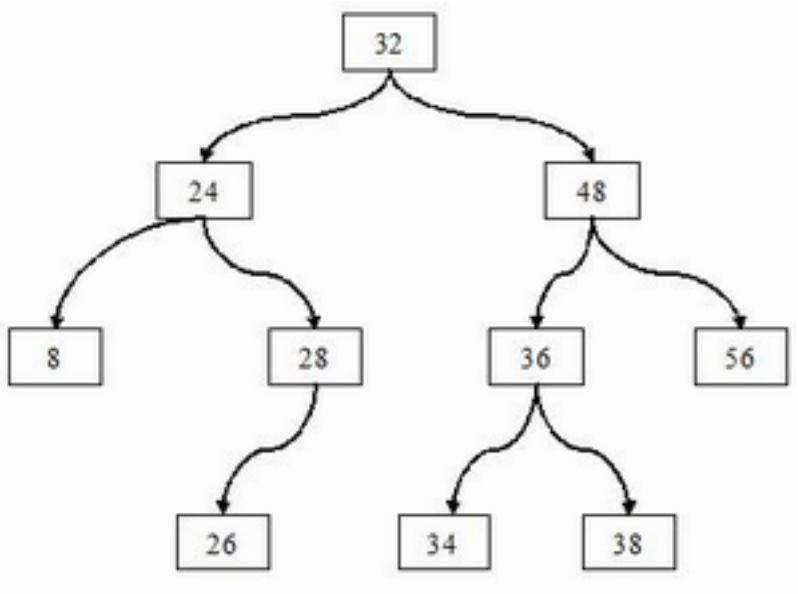
```
class GeoRegion {
public:
  static GeoRegion* load (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};
class GeoRegionProxy {
public:
  GeoRegionProxy (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};
```

Rešenje:

```
class GeoRegionProxy {
   public:
     GeoRegionProxy (char* regionName);
     double getSurface ();
     double getHighestPeak ();
   protected:
    GeoRegion* getServer ();        //učitavanje objekta sa diska (ako nije već učitan)
   private:
    GeoRegion* myServer;            //pokazivač na objekat (null ako nije učitan)
    char* myName;                   //identifikator regiona objekta za učitavanje
};

//konstruktor: myServer je pokazivač na objekat GeoRegion sa zadatim regionName
//pažnja: objekat nije učitan, samo su postavljeni parametri potrebni za učitavanje
GeoRegionProxy::GeoRegionProxy (char* regionName)
  : myServer(0), myName(regionName) {}

//metoda učitava objekat sa diska (ako već nije učitan) sa zadatim parametrima
GeoRegion* GeoRegionProxy::getServer() {
  if (myServer==0) {                    //ako već nije učitam
      myServer=GeoRegion::load(myName);   //učitavamo objekat sa diska
      if (myServer==0) ...                //ako objekat nije uspešno učitan...greška
  }
  return myServer;                      //vraćamo pokazivač na učitani objekat
}

double GeoRegionProxy::getSurface () {
  return getServer()->getSurface();       //učitavamo objekat i pozivamo traženu metodu
}

double GeoRegionProxy::getHighestPeak () {
  return getServer()->getHighestPeak();   //učitavamo objekat i pozivamo traženu metodu
```

```
}
```

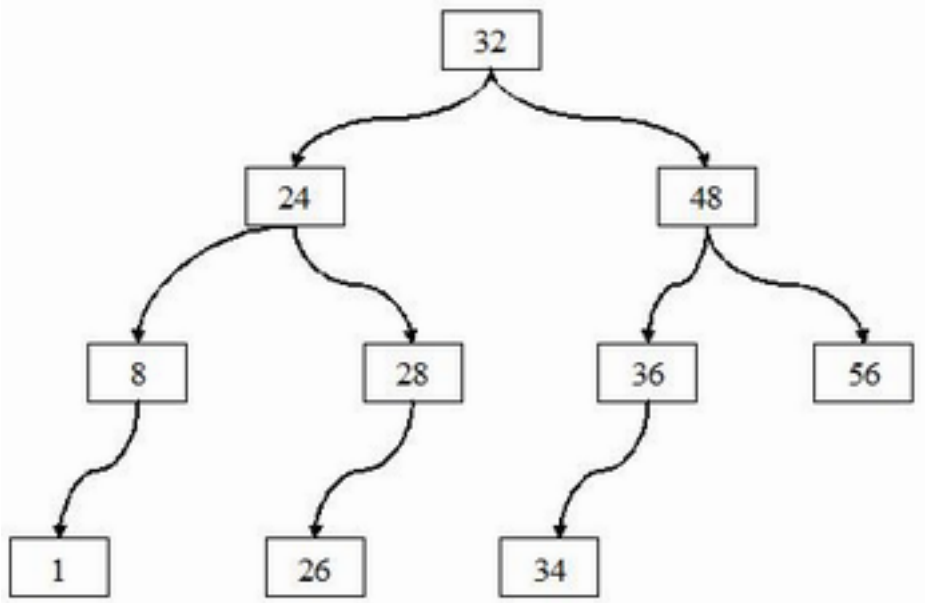
4. (*Septembar 2011*) Neki sistem primenjuje kontinualnu alokaciju memorije sa *best-fit* algoritmom. Radi efikasnijeg pronalaženja najpovoljnijeg dela memorije, sistem održava binarno stablo slobodnih fragmenata memorije, pri čemu svaki čvor ovog binarnog stabla predstavlja jedan slobodni fragment, u levom podstablu su manji, a u desnom veći fragmenti. Na slici je prikazano stanje ovog stabla u datom trenutku (u čvor je upisana veličina fragmenta u jedinicama alokacije). Prikazati stanje ovog stabla nakon alokacije memorije veličine 37 jedinica.



Rešenje:

Potrebno je da pronađemo najmanji slobodan fragment u memoriji veličine veće ili jednake veličini memorije koja se alocira. U ovom slučaju se alocira memorija veličine 37 jedinica, pa je fragment veličine 38 jedinica najmanji slobodni fragment veći ili jednak od 37 jedinica.

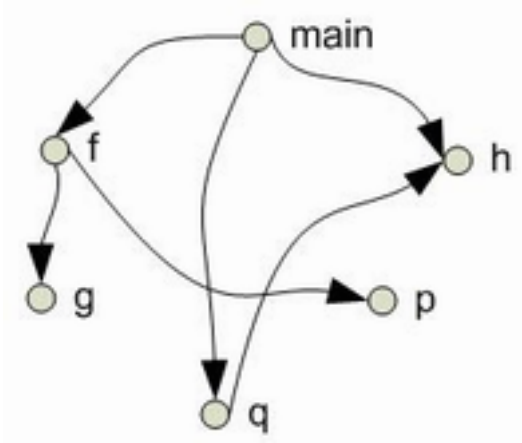
U okviru tog fragmenta alociramo potrebnih 37 jedinica, a preostala jedinica ostaje slobodna, pa je potrebno da ažuriramo binarno stablo slobodnih fragmenata memorije kao na slici:



5. (*Maj 2010*)

a. Ukoliko prevodilac generiše samo uobičajeni kod za povratak iz potprograma (skidanje samo sačuvane povratne adrese sa steka i skok na tu adresu), da li se dve procedure, od kojih jedna poziva onu drugu, mogu nalaziti u dva različita modula-preklopa (*overlays*) koji se učitavaju na isto mesto (jedan preko drugog)? Obrazložiti odgovor.

b. Na slici je dat graf poziva potprograma nekog programa. Čvorovi grafa predstavljaju potprograme, a grana je usmerena od pozivaoca prema pozvanom potprogramu. Za pretpostavke u tački a) i na osnovu zaključka iz te tačke, odgovoriti da li je korektna svaka od sledeće dve konfiguracije preklopa.



Konfiguracija 1:
Modul A: main, h
Modul B: f, g, p
Modul C: q
Preklapaju se modul B i modul C, modul A je uvek učitan.
Korektna? Odgovor: _____

Konfiguracija 2:
Modul A: main, p
Modul B: f, g
Modul C: q, h
Preklapaju se modul B i modul C, modul A je uvek učitan.
Korektna? Odgovor: _____

Rešenje:

a. Ne. Kada se iz pozivajućeg potprograma pozove onaj drugi, na isto mesto modula kome pripada pozivajući potprogram se učitava modul u kome je pozvani potprogram. Kada se vrši povratak iz tog pozvanog potprograma, ukoliko prevodilac generiše samo kod za jednostavni indirektni skok preko adrese skinute sa steka, skok će biti na adresu unutar istog modula, a ne na kod unutar pozivajućeg potprograma, jer je on u modulu koji je izbačen, što nije korekno.
Prema tome, potprogrami koji su u relaciji pozivalac-pozvani se mogu nalaziti ili u istom modulu, ili u dva modula koji se ne preklapaju (ne učitavaju na isto mesto jedan preko drugog).

b. Obe konfiguracije su korektne, pošto je za sve grane zadovoljen uslov iz zaključka prethodne tačke.

6. (*Maj 2010*) Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 64KB. Fizički adresni prostor je veličine 4GB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 1K ulaza. Posmatra se proces koji koristi prvih 400 stranica i poslednjih 64 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

a. prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa
b. kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
c. koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
d. koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje:

a. Veličina virtuelnog adresnog prostora je: $VAS = 4\ GB = 2^{32}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 4\ GB = 2^{32}\ B$
Veličina adresibilne jedinice je: $AU = 1\ B = 2^{0}\ B$
Veličina stranice (okvira) je: $PAGE = 64\ KB = 2^{16}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 32$ bita
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 32$ bita

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(PAGE / AU) = 16$ bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj stranice u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 16 bita, pa je za predstavljanje broja stranice potrebno:

$$VA\_S - OFFSET\_S = 32 \text{ bita} - 16 \text{ bita} = 16 \text{ bita}$$

Jedan deo od tih 16 bita se koristi za indeksiranje PMT-a prvog nivoa, a preostali za indeksiranje PMT-a drugog nivoa. Tabela prvog nivoa ima $1K = 2^{10}$ ulaza, pa je za indeksiranje PMT-a prvog nivoa potrebno:

$$VA\_L1\_S = \log_2(1K) = 10 \text{ bita}$$

Za indeksiranje PMT-a drugog nivoa je onda potrebno:

$$VA\_L1\_S = VA\_S - OFFSET\_S - \ VA\_L1\_S = 6 \text{ bita}$$

Dakle, logička struktura virtuelne adrese je: VA(32) = *Page_L1*(10):*Page_L2*(6):*Offset*(16) .

b. Jedan ulaz u PMT prvog nivoa sadrži fizičku adresu početka PMT-a drugog nivoa, pa je on veličine:

$$PMT1\_ENTRY\_S = \lceil PA\_S / AU \rceil * AU = 4 * AU = 4\ B$$

c. Već smo našli da je veličina jednog ulaza u PMT prvog nivoa veličine 4B, a kako tabela ima 1K ulaza,
ukupna veličina PMT prvog nivoa je:

$$PMT1\_S = 1K * PMT1\_ENTRY\_S = 4\ KB$$

Jedan ulaz u PMT drugog nivoa sadrži broj okvira u koji se stranica preslikava, pa je veličine:

$$PMT2\_ENTRY\_S = PA\_S - OFFSET\_S = 16\ bita = 2\ B$$

Kako za indeksiranje PMT-a drugog nivoa imamo na raspolaganju 6 bita unutar VA, broj ulaza u PMT-u drugog nivoa je 2^6, pa je ukupna veličina PMT-a drugog nivoa:

$$PMT2\_S = 2^6 * PMT2\_ENTRY\_S = 2^7\ B = 128\ B$$

Ako bi neki proces koristio ceo svoj virtuelni adresni prostor, u memoriji bi se nalazio njegov PMT prvog nivoa i svi PMT drugog nivoa, kojih u ovom slučaju ima 1K (broj ulaza u PMT prvog nivoa). Dakle, sve PMT tog procesa bi u memoriji zauzimale:

$$PMT\_S = PMT1\_S + 1K * PMT2\_S = 4\ KB + 128\ KB = 132\ KB$$

d. Za prvih 400 stranica je potrebno da se u memoriji nalaze prvih $\lceil 400 / 64 \rceil = 7$ PMT-a drugog nivoa, a za poslednjih 64 je potrebno da se u memoriji nalaze poslednjih $\lceil 64 / 64 \rceil = 1$ PMT-a drugog nivoa (64 je broj ulaza u jednom PMT-u drugog nivoa). Dakle, u memoriji se nalazi ukupno 8 PMT-a drugog nivoa i (jedini) PMT prvog nivoa, pa oni ukupno zauzimaju:

$$PMT\_S = PMT1\_S + 8 * PMT2\_S = 4\ KB + 1\ KB = 5\ KB$$

7. (*Maj 2010*) U nekom sistemu primenjuje se kontinualna alokacija operativne memorije. Deo definicije strukture PCB je sledeći:

```
struct PCB {
  ...
  void* memLocation;  // Current place in memory
  size_t memSize;  // Size of memory space
};
```

Implementirati operaciju:

```
void relocate(PCB* process, void* newPlace);
```

kojom sistem premešta memorijski prostor procesa sa tekućeg na novozadato (već alocirano) mesto u memoriji.
U datom sistemu, operacija

```
void free (void* addr, size_t size);
```

dealocira (proglašava slobodnim) memorijski prostor veličine size počev od adrese addr.
Pomoć: Standardna bibliotečna funkcija memcpy ima sledeću deklaraciju:

```
void* memcpy (void* destination, const void* source, size_t size);
```

Rešenje:

```
void relocate(PCB* p, void* newPlace) {
  //ako je pokazivač p NULL, onda se vraćamo iz funkcije (geška)
  if (p == 0) return;

  //ako se premešta na isto mesto na kojem je trenutno
  //ili ako je veličina bloka memorije koji se premešta nula
  //ne treba ništa da uradimo (jer se ništa ne bi ni promenilo)
  if (newPlace==p->memLocation || p->memSize==0) return;

  //kopiramo blok memorije na novu lokaciju
  memcpy(newPlace,p->memLocation,p->memSize);

  //oslobađamo prethodno zauzeti memorijski prostor
  free(p->memLocation,p->memSize);

  //ažuriramo trenutnu adresu memorijskog bloka u PCB strukturi
  p->memLocation=newPlace;
}
```

8. (*Maj 2009*) Sledeća dva potprograma čine jedan programski modul koji treba učitavati dinamički (engl. *dynamic loading*):

```
int f (int,int);
double g (double);
```

Ovi potprogrami implementirani su u jednom C fajlu *p.c* koji izgleda ovako:

```
int _f (int,int);
double _g (double);
void* map[2] = {&_f, &_g};

int _f (int x, int y){
  ...
}

double _g (double x) {
  ...
}
```

Ovaj modul biće preveden u fajl *p.obj*. Ovaj modul ne uvozi spoljašnje simbole koji su definisani u drugim modulima, već su u njemu sva adresiranja internih simbola.
Napisati C deo koda glavnog modula programa sa *stub* procedurama za ove potprograme, koje obezbeđuju dinamičko učitavanje ovog modula. Na raspolaganju je sistemska usluga:

```
void* load_module(char* filename);
```

koja učitava izvršni kod (bez zaglavlja sa simbolima) iz *.obj* fajla sa datim imenom u adresni prostor pozivajućeg procesa i vraća adresu na koju je učitan taj modul, a prilikom učitavanja razrešava sva adresiranja internih simbola (u ovom slučaju elemente niza map postavlja na apsolutne adrese funkcija `_f` i `_g` određene prilikom učitavanja). Zanemariti moguće greške prilikom učitavanja modula.

Rešenje:

```
//adresa učitanog modula
static void* module_p = NULL; //nakon učitavanja, module_p je adresa niza map

//stub funkcija za _f
int f (int x, int y) {
  //ako modul još nije učitan (NULL), učitavamo ga pomoću sistemskog
  //poziva load_module i adresu na koju je učitan čuvamo u module_p
  if (module_p == NULL)
      module_p = load_module("p.obj");

  //dohvatamo adresu funkcije _f
  int (*_f)(int,int) = (int(*)(int,int))(module_p[0]);
  //ako nije došlo do greške, pozivamo funkciju i vraćamo njenu povratnu vrednost
  if (module_p!=NULL && _f != NULL) return _f(x,y);
}

//stub funkcija za _g
double g (double x) {
  //ako modul još nije učitan (NULL), učitavamo ga pomoću sistemskog
  //poziva load_module i adresu na koju je učitan čuvamo u module_p
  if (module_p == NULL)
      module_p = load_module("p.obj");

  //dohvatamo adresu funkcije _f
  double (*_g)(double) = (double(*)(double))(module_p[1]);
  //ako nije došlo do greške, pozivamo funkciju i vraćamo njenu povratnu vrednost
```

```
  if (module_p!=NULL && _g != NULL) return _g(x);
}
```

9. (*Maj 2009*) Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 16KB. Fizički adresni prostor je veličine 1GB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 2K ulaza. Posmatra se proces koji koristi prvih 800 stranica i poslednjih 56 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

a. prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa
b. kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
c. koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
d. koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje:

a. Veličina virtuelnog adresnog prostora je: $VAS = 4\ GB = 2^{32}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 1\ GB = 2^{30}\ B$
Veličina adresibilne jedinice je: $AU = 1\ B = 2^{0}\ B$
Veličina stranice (okvira) je: $PAGE = 16\ KB = 2^{14}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 32$ bita
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 30$ bita

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(PAGE / AU) = 14$ bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj stranice u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 14 bita, pa je za predstavljanje broja stranice potrebno:

$$VA\_S - OFFSET\_S = 32 \text{ bita} - 14 \text{ bita} = 18 \text{ bita}$$

Jedan deo od tih 18 bita se koristi za indeksiranje PMT-a prvog nivoa, a preostali za indeksiranje PMT-a drugog nivoa. Tabela prvog nivoa ima $2K = 2^{11}$ ulaza, pa je za indeksiranje PMT-a prvog nivoa potrebno:

$$VA\_L1\_S = \log_2(2K) = 11 \text{ bita}$$

Za indeksiranje PMT-a drugog nivoa je onda potrebno:

$$VA\_L1\_S = VA\_S - OFFSET\_S - \ VA\_L1\_S = 7 \text{ bita}$$

Dakle, logička struktura virtuelne adrese je: VA(32) = *Page_L1*(11):*Page_L2*(7):*Offset*(14) .

b. Jedan ulaz u PMT prvog nivoa sadrži fizičku adresu početka PMT-a drugog nivoa, pa je on veličine:

$$PMT1\_ENTRY\_S = \lceil PA\_S / AU \rceil * AU = 4 * AU = 4\ B$$

c. Već smo našli da je veličina jednog ulaza u PMT prvog nivoa veličine 4B, a kako tabela ima 2K ulaza,
ukupna veličina PMT prvog nivoa je:

$$PMT1\_S = 2K * PMT1\_ENTRY\_S = 8\ KB$$

Jedan ulaz u PMT drugog nivoa sadrži broj okvira u koji se stranica preslikava, pa je veličine:

$$PMT2\_ENTRY\_S = PA\_S - OFFSET\_S = 16\ bita = 2\ B$$

Kako za indeksiranje PMT-a drugog nivoa imamo na raspolaganju 7 bita unutar VA, broj ulaza u PMT-u drugog nivoa je 2^7, pa je ukupna veličina PMT-a drugog nivoa:

$$PMT2\_S = 2^7 * PMT2\_ENTRY\_S = 2^8\ B = 256\ B$$

Ako bi neki proces koristio ceo svoj virtuelni adresni prostor, u memoriji bi se nalazio njegov PMT prvog nivoa i svi PMT drugog nivoa, kojih u ovom slučaju ima 2K (broj ulaza u PMT prvog nivoa). Dakle, sve PMT tog procesa bi u memoriji zauzimale:

$$PMT\_S = PMT1\_S + 2K * PMT2\_S = 8\ KB + 512\ KB = 520\ KB$$

d. Za prvih 800 stranica je potrebno da se u memoriji nalaze prvih $\lceil 800 / 128 \rceil = 7$ PMT-a drugog nivoa, a za poslednjih 56 je potrebno da se u memoriji nalaze poslednjih $\lceil 56 / 128 \rceil = 1$ PMT-a drugog nivoa (128 je broj ulaza u jednom PMT-u drugog nivoa). Dakle, u memoriji se nalazi ukupno 8 PMT-a drugog nivoa i (jedini) PMT prvog nivoa, pa oni ukupno zauzimaju:

$$PMT\_S = PMT1\_S + 8 * PMT2\_S = 8\ KB + 2\ KB = 10\ KB$$

10. (*Maj 2009*) U nekom sistemu primenjuje se kontinualna alokacija operativne memorije, uz primenu algoritma *first fit* i alokaciju u jedinicama veličine 1KB. Ukupan prostor za alokaciju memorije za korisničke procese je veličine 128KB. Posmatra se sledeća sekvenca zahteva za alokacijom i dealokacijom memorije:
A+45, B+23, C+38, B-, D+16, E+16, F+5, C-, G+25
U oznaci svakog zahteva prvo slovo označava proces koji izdaje zahtev, simbol „+“ označava operaciju alokacije memorije, simbol „–„ označava operaciju dealokacije memorije, dok broj označava veličinu memorije koja se alocira izraženu u KB.

a. popuniti sledeću tabelu vrednostima koje opisuju stanje zauzetosti memorije nakon ove sekvence:

Proces	A	D	E	F	G
Adresa početka ($\times 2^{10}$)					

b. Broj slobodnih fragmenata je _____ .
Ukupna veličina slobodne memorije je _____ .
Veličina najvećeg slobodnog fragmenta je _____ .
Veličina najmanjeg slobodnog fragmenta je _____ .

Rešenje:

a. Legenda za jedan fragment: [process:size:first-last]
- process: oznaka procesa koji je alocirao fragment (#, za slobodan fragment)
- size: veličina fragmenta (broj alokacionih jedinica)
- first: početna adresa fragmenta (x 2^{10})
- last: krajnja adresa fragmenta (x 2^{10})

Početni izgled prostora za alokaciju memorije za korisničke procese: [#:128:0-127]

Izgled nakon izvršenja svakog od zahteva:

```
A+45: [A:45:0-44][#:83:45-127]
B+23: [A:45:0-44][B:23:45-67][#:60:68-127]
C+38: [A:45:0-44][B:23:45-67][C:38:68-105][#:22:106-127]
B-:   [A:45:0-44][#:23:45-67][C:38:68-105][#:22:106-127]
D+16: [A:45:0-44][D:16:45-60][#:7:61-67][C:38:68-105][#:22:106-127]
E+16: [A:45:0-44][D:16:45-60][#:7:61-67][C:38:68-105][E:16:106-121][#:6:122-127]
F+5:  [A:45:0-44][D:16:45-60][F:5:61-65][#:2:66-67][C:38:68-105][E:16:106-121][#:6:122-127]
C-:   [A:45:0-44][D:16:45-60][F:5:61-65][#:40:66-105][E:16:106-121][#:6:122-127]
G+25: [A:45:0-44][D:16:45-60][F:5:61-65][G:40:66-90][#:15:91-105][E:16:106-121][#:6:122-127]
```

Sada možemo popuniti tabelu:

Proces	A	D	E	F	G
Adresa početka ($\times 2^{10}$)	0	45	106	61	66

c. Broj slobodnih fragmenata je: 2 ([#:15:91-105] i [#:6:122-127])
Ukupna veličina slobodne memorije je: 21 KB (15 + 6 = 21)
Veličina najvećeg slobodnog fragmenta je: 15 KB ([#:15:91-105])
Veličina najmanjeg slobodnog fragmenta je: 6 KB ([#:6:122-127])

11. (*Maj 2008*) Objasniti na koji način prevodilac i/ili linker podržava to što je u nekim jezicima dozvoljeno da se u različitim modulima pojave definicije više entiteta sa istim imenom, ali u različitim tzv. prostorima imena (engl. *namespace*). Na primer, u jeziku C++, statički podatak-član sa istim imenom m (a koji po definiciji ima eksterno vezivanje, odnosno izvozi se kao simbol iz modula) može sasvim regularno da se definiše u različitim klasama X i Y.

Rešenje:

Iako ovakvi entiteti imaju isto *nekvalifikovano* ime u datom programu, oni imaju različita *potpuno kvalifikovana* imena (engl. *fully qualified name*) koja se sastoje od pune staze imena njihovih okružujućih prostora imena. Na primer, statički podatak-član m klase X ima potpuno kvalifikovano ime `X::m`, dok istoimeni član klase Y ima potpuno kvalifikovano ime `Y::m`; slično važi za ugnežđene prostore imena (npr. `P::Q::R::S::t`). Da bi se omogućilo definisanje istoimenih simbola u različitim prostorima imena, prevodilac jednostavno kao simbol u .obj fajlu definiše potpuno kvalifikovano ime, a ne nekvalifikovano ime, tako da linker vidi to jednoznačno puno ime. Drugim rečima, linker ne poznaje pojam prostora imena niti kvalifikovanog imena, za njega su svi simboli jednostavni nizovi znakova koji moraju biti jednoznačno definisani. Prevodilac obezbeđuje ovu jednoznačnosti korišćenjem punih kvalifikovanih imena kao imena simbola koje ostavlja linkeru.

12. (*Maj 2008*) Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 4KB. Fizički adresni prostor je veličine 256MB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 1K ulaza. Posmatra se proces koji koristi prve 1032 stranice i poslednjih 10 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

a. prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa
b. kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
c. koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
d. koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje:

a. Veličina virtuelnog adresnog prostora je: $VAS = 4\ GB = 2^{32}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 256\ MB = 2^{28}\ B$
Veličina adresibilne jedinice je: $AU = 1\ B = 2^{0}\ B$
Veličina stranice (okvira) je: $PAGE = 4\ KB = 2^{12}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 32$ bita
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 28$ bita

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(PAGE / AU) = 12$ bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj stranice u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar te stranice. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 12 bita, pa je za predstavljanje broja stranice potrebno:

$$VA\_S - OFFSET\_S = 32 \text{ bita} - 12 \text{ bita} = 20 \text{ bita}$$

Jedan deo od tih 20 bita se koristi za indeksiranje PMT-a prvog nivoa, a preostali za indeksiranje PMT-a drugog nivoa. Tabela prvog nivoa ima $1K = 2^{10}$ ulaza, pa je za indeksiranje PMT-a prvog nivoa potrebno:

$$VA\_L1\_S = \log_2(1K) = 10 \text{ bita}$$

Za indeksiranje PMT-a drugog nivoa je onda potrebno:

$$VA\_L1\_S = VA\_S - OFFSET\_S - \ VA\_L1\_S = 10 \text{ bita}$$

Dakle, logička struktura virtuelne adrese je: VA(32) = *Page_L1*(10):*Page_L2*(10):*Offset*(12) .

b. Jedan ulaz u PMT prvog nivoa sadrži fizičku adresu početka PMT-a drugog nivoa, pa je on veličine:

$$PMT1\_ENTRY\_S = \lceil PA\_S / AU \rceil * AU = 4 * AU = 4\ B$$

c. Već smo našli da je veličina jednog ulaza u PMT prvog nivoa veličine 4B, a kako tabela ima 1K ulaza,
ukupna veličina PMT prvog nivoa je:

$$PMT1\_S = 1K * PMT1\_ENTRY\_S = 4\ KB$$

Jedan ulaz u PMT drugog nivoa sadrži broj okvira u koji se stranica preslikava, pa je veličine:

$$PMT2\_ENTRY\_S = PA\_S - OFFSET\_S = 16\ bita = 2\ B$$

Kako za indeksiranje PMT-a drugog nivoa imamo na raspolaganju 10 bita unutar VA, broj ulaza u PMT-u drugog nivoa je 2^{10}, pa je ukupna veličina PMT-a drugog nivoa:

$$PMT2\_S = 2^{10} * PMT2\_ENTRY\_S = 2^{11}\ B = 2\ KB$$

Ako bi neki proces koristio ceo svoj virtuelni adresni prostor, u memoriji bi se nalazio njegov PMT prvog nivoa i svi PMT drugog nivoa, kojih u ovom slučaju ima 1K (broj ulaza u PMT prvog nivoa). Dakle, sve PMT tog procesa bi u memoriji zauzimale:

$$PMT\_S = PMT1\_S + 1K * PMT2\_S = 4\ KB + 2MB$$

d. Za prvih 1032 stranica je potrebno da se u memoriji nalaze prvih $\lceil 1032 / 1024 \rceil = 2$ PMT-a drugog nivoa, a za poslednjih 10 je potrebno da se u memoriji nalaze poslednjih $\lceil 10 / 1024 \rceil = 1$ PMT-a drugog nivoa (1024 je broj ulaza u jednom PMT-u drugog nivoa). Dakle, u memoriji se nalazi ukupno 3 PMT-a drugog nivoa i (jedini) PMT prvog nivoa, pa oni ukupno zauzimaju:

$$PMT\_S = PMT1\_S + 3 * PMT2\_S = 4\ KB + 6\ KB = 10\ KB$$

13. (*Maj 2008*) U nekom operativnom sistemu primenjuje se tehnika *copy-on-write* i učitavanje stranica virtuelne memorije na zahtev (*demand paging*). Za potrebe preslikavanja stranica postoje dve odvojene strukture: tabela preslikavanja stranica (PMT) koju hardver za preslikavanje adresa jedino koristi i koja sadrži *minimum* potrebnih informacija za ovo hardversko preslikavanje, kako bi bila što manja, i struktura koju isključivo koristi operativni sistem i koja čuva ostale informacije o virtuelnom adresnom prostoru procesa. Drugim rečima, u PMT su uključene one i samo one informacije koje su potrebne hardveru za preslikavanje adresa. Za svaku od sledećih informacija navesti da li se nalazi u ovoj PMT ili ne:

Informacija	Da li se nalazi u PMT? (Upisati „Da" ili „Ne")
Indikator da li se stranica nalazi u fizičkoj memoriji.	Da
Indikator da li je stranica uopšte dozvoljena za pristup (registrovana kao korišćeni deo virtuelnog adresnog prostora).	Ne
Indikator da li je hardveru dozvoljen upis u stranicu.	Da
Indikator da je logički dozvoljen upis u stranicu, ali je ona deljena sa *copy-on-write* semantikom.	Ne
Indikator da li je stranica „prljava", odnosno menjana od svog učitavanja.	Da
Broj okvira u koji se stranica preslikava.	Da
Broj bloka na particiji za zamenu stranica.	Ne

14. (*Maj 2007*) Neki prevodilac koji podržava dinamičko učitavanje (*dynamic loading*) na početku svakog modula koji je predviđen za dinamičko učitavanje generiše tabelu adresa potprograma koji su definisani u tom modulu. Adrese su relativne u odnosu na početak modula. Na primer, za modul M u kome su definisani sledeći potprogrami:

```
void f(int);
int g(int,int);
double h(double,int);
```

prevodilac će na samom početku binarnog fajla sa prevodom koda tog modula generisati sledeću strukturu („tabelu“):

```
void* _funtbl[] = { &f, &g, &h };
```

Osim toga, u glavnom modulu datog programa koji se uvek inicijalno učitava pri kreiranju procesa nad tim programom, prevodilac generiše *stub* („patrljak“) za svaki potprogram koji je definisan u nekom modulu koji je predviđen za dinamičko učitavanje.

Računar i operativni sistem na kome se izvršavaju ovakvi programi ne podržavaju virtuelnu memoriju. Operativni sistem obezbeđuje uslugu alokacije dela operativne memorije i učitavanja sadržaja iz fajla u taj deo memorije sledećim sistemskim pozivom:

```
void* alloc_and_load(char* filename);
```

Ovaj poziv vraća adresu u operativnoj memoriji gde je alociran prostor i učitan sadržaj fajla sa datim imenom, odnosno NULL ako je došlo do greške (npr. nema slobodnog prostora ili ne postoji fajl sa datim imenom).

Napisati kod na jeziku C za *stub* („patrljak“) funkcije h iz gore pomenutog modula M čiji je binarni oblik u fajlu „m.dlm“.

Rešenje:

```
double h(double _1, int _2) {
  typedef double (*PFUN)(double,int);     //definicija tipa pokazivača na funkciju h
  static PFUN _my_impl = NULL;            //pokazivač na funkciju h

  if (_my_impl == NULL) {                 //ako modul još nije učitan u memoriju
      //učitavamo modul u memoriju
      void* _m = alloc_and_load("m.dlm");
      //ako učitavanje nije uspelo, prekidamo izvršavanje
      if (_m == NULL) exit(1);
      //u suprotnom, dohvatamo adresu funkcije h iz tog modula i čuvamo je u _my_impl
      _my_impl = (PFUN)((int)((void**)_m)[2]+(int)_m)
  }

  //pozivamo funkciju h i vraćamo njenu povratnu vrednost
  return _my_impl(_1,_2);
}
```

15. (*Maj 2007*) U nekom operativnom sistemu primenjuje se kontinualna alokacija operativne memorije. Alokator održava spisak slobodnih delova memorije kao ulančanu listu uređenih parova (adresa početka slobodnog dela, veličina slobodnog dela). U nekom trenutku lista ima sledeće elemente redom (sve vrednosti su heksadecimalne):
(7F1A, 24), (1FFC, 42), (A000, 20), (0770, 7A), (4010, 68), (3A0A, A0), (01B0, 30)
Koji deo će biti alociran ako se traži alokacija prostora veličine 47h (navesti samo adresu slobodnog dela koji je odabran), ako sistem primenjuje sledeći algoritam alokacije:

a. *first-fit*
b. *best-fit*
c. *worst-fit*.

Rešenje:

a. Algoritam f*irst-fit*: tražimo prvi fragment slobodne memorije čija je veličina veća ili jednaka 47h.
Prvi takav fragment je (0770, 7A), jer je 7A ≥ 47.
Dakle, biće alociran deo memorije počevši od adrese 0770h.

b. Algoritam *best-fit*: tražimo fragment slobodne memorije čija se veličina najmanje razlikuje od 47h, a pritom je veća ili jednaka 47h.
Traženi fragment je (4010, 68), jer je 68-47 = 21h najmanja nenegativna razlika veličine segmenta i potrebne veličine prostora za alokaciju.
Dakle, biće alociran deo memorije počevši od adrese 4010h.

c. Algoritam *worst-fit*: tražimo fragment slobodne memorije čija se veličina najviše razlikuje od 47h, a pritom je veća ili jednaka 47h.
Traženi fragment je (3A0A, A0), jer je A0-47 = 59h najveća nenegativna razlika veličine segmenta i potrebne veličine prostora za alokaciju.
Dakle, biće alociran deo memorije počevši od adrese 3A0Ah.

16. (*Maj 2006*) Precizno objasniti zašto je potrebno linkeru navesti da kao svoj proizvod treba da napravi statičku biblioteku (lib), a ne izvršni program (exe)? Precizno objasniti razlike između postupka i rezultata pravljenja ove dve vrste proizvoda povezivanja.

Rešenje:

Statička biblioteka sadrži zaglavlje sa spiskom simbola koje izvozi i uvozi, i samo telo biblioteke (kod), drugim rečima, proizvod je istog oblika kao i proizvod prevođenja, dok izvršni fajl sadrži telo (kod) i zaglavlje u kome se nalazi adresa prve instrukcije koja treba da se izvrši. Zbog ove razlike u proizvodima, linkeru je potrebna informacija šta da napravi. Osim toga, u samom postupku razrešavanja simbola, prilikom pravljenja izvršnog fajla, postojanje nedefinisanog, a referisanog simbola se nakon prvog prolaza prijavljuje kao greška. U slučaju pravljenja biblioteke, ovakav slučaj je dozvoljen.

17. (*Maj 2006*) Virtuelni adresni prostor sistema je 1GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan segmentno. Svaki proces može imati najviše 64 segmenta. Fizički adresni prostor je veličine 512MB. Prostor za zamenu (*swap space*) na disku je duplo veći od fizičke memorije i koristi se za čuvanje zamenjenih segmenata kojima se direktno pristupa na disku (na „presnoj" particiji, ne kroz fajl-sistem). Pristup segmentima se kontroliše pomoću tri bita zaštite koji se označavaju sa R (*read*), W (*write*) i E (*execute*). Kada se stranica nalazi u memoriji, prostor koji je zauzimala na disku se oslobađa za zamenu drugih stranica. Odgovoriti na sledeće pitanja i kratko, ali precizno obrazložiti odgovor: ako se deskriptori segmenata u tabeli preslikavanja segmenata (SMT) maksimalno kompaktno smeštaju, kolika je veličina SMT za jedan proces?

Rešenje:

Veličina virtuelnog adresnog prostora je: $VAS = 1\ GB = 2^{30}\ B$
Veličina fizičkog adresnog prostora je: $PAS = 512\ MB = 2^{29}\ B$
Veličina adresibilne jedinice je: $AU = 1\ B = 2^{0}\ B$
Veličina *swap* diska je: $SWAP = 1\ GB = 2^{30}\ B$

Širina virtuelne adrese je: $VA\_S = \log_2(VAS / AU) = 30$ bita
Širina fizičke adrese je: $PA\_S = \log_2(PAS / AU) = 29$ bita
Širina adrese na *swap* disku je: $SWAP\_S = \log_2(SWAP / AU) = 30$ bita

Maksimalan broj segmenata je: $SEG\_MAX = 64$
Maksimalna veličina jednog segmenta: $SEG\_SIZE = VAS / SEG\_MAX = 8\ MB = 2^{24}\ B$

Širina pomeraja (*Offset*) unutar stranice/okvira: $OFFSET\_S = \log_2(SEG\_SIZE / AU) = 24$ bita

Virtuelna adresa se sastoji od određenog broja bita koji određuju broj segmenta u virtuelnom adresnom prostoru i određenog broja bita koji određuju pomeraj unutar tog segmenta. Već smo utvrdili da je za predstavljanje pomeraja potrebno odvojiti 24 bita, pa je za predstavljanje broja segmenta potrebno:

$$VA\_S - OFFSET\_S = 30 \text{ bita} - 24 \text{ bita} = 6 \text{ bita}$$

Dakle, logička struktura virtuelne adrese je: VA(32) = *Segment*(6):*Offset*(24) .

U deskriptoru segmenta se čuvaju sledeće informacije:

- da li je taj segment u memoriji (1 bit)
- pravo pristupa (RWE) tom segmentu (3 bita)
- stvarna veličina segmenta (24 bita)
- ako je u memoriji, koja je početna (fizička) adresa segmenta (29 bita)
  ako nije u memoriji, gde se nalazi na swap disku (30 bita)
  (jedno od ova dva se čuva u zajedničkih 30 bita)

Sledi da je veličina jednog ulaza (deskriptora) u SMT: 1 + 3 + 24 + 30 = 58 bita (ili 64 bita, ako koristimo ceo broj adresibilnih jedinica).

Ukupna veličina SMT-a je onda jednaka veličini jednog ulaza pomnoženog sa brojem ulaza (koji je jednak maksimalnom broju segmenata, odnosno 64):

$$SMT\_SIZE = 64 * 58 \text{ bita} = 464\ B$$
(ili 512 B ako koristimo ceo broj adresibilnih jedinica za jedan ulaz)

18. (*Maj 2006*) U nekom sistemu sa straničnom organizacijom virtuelne memorije u datom trenutku raspored stranica tri procesa po okvirima u fizičkoj memoriji izgleda redom ovako (oznake su složene redom po rastućem broju okvira, slova A, B i C označavaju procese kojima pripadaju stranice u tim okvirima, brojevi označavaju brojeve stranica tih procesa, K označava okvir koji pripada jezgru OS-a, a DLL označava okvir čiji je sadržaj DLL koga koriste sva tri procesa A, B i C):
K, A1, B3, C1, DLL, C0, B2, K, K
U virtuelnom adresnom prostoru procesa A dati DLL se nalazi na stranici broj 2, a u prostorima procesa B i C na stranici broj 5. Napisati kako izgledaju tabele preslikavanja stranica za sva tri procesa A, B i C (u tabelu upisati brojeve okvira i T ili F kao oznaku da li je stranica u memoriji ili ne, za prvih 6 stranica).

Strana:	0	1	2	3	4	5
Proces A						
Proces B						
Proces C						

Rešenje:
Raspored stranica po okvirima, uz broj okvira u koji se preslikava stranica je:
0:K, 1:A1, 2:B3, 3:C1, 4:DLL, 5:C0, 6:B2, 7:K, 8:K

Proces A:
- 1:A1 - stranica 1 se preslikava u okvir 1
- 4:DLL - zajednička biblioteka nalazi se na stranici 2 (i okviru 4)

Proces B:
- 2:B3 - stranica 3 se preslikava u okvir 2
- 6:B2 - stranica 2 se preslikava u okvir 6
- 4:DLL - zajednička biblioteka nalazi se na stranici 5 (i okviru 4)

Proces C:
- 3:C1 - stranica 1 se preslikava u okvir 3
- 5:C0 - stranica 0 se preslikava u okvir 5
- 4:DLL - zajednička biblioteka nalazi se na stranici 5 (i okviru 4)

Na osnovu prethodnog, možemo popuniti tabelu:

Strana:	0	1	2	3	4	5
Proces A	F	T 1	T 4	F	F	F
Proces B	F	F	T 6	T 2	F	T 4
Proces C	T 5	T 3	F	F	F	T 4

Шеста група задатака: улазно-излазни подсистем

1. (*Januar 2011*) U nekom modularnom operativnom sistemu drajveri za raznovrsne ulazno/izlazne blokovski orijentisane uređaje registruju se kao moduli na sledeći način. Sistem održava jedinstvenu, centralizovanu tabelu registrovanih drajvera `blockIOMap` u kojoj svaki ulaz odgovara jednom registrovanom drajveru. Prilikom instalacije, svaki drajver u novi ulaz ove tabele upisuje adresu svoje tabele tipa `BlockIOVectorTable` sa adresama svojih funkcija zaduženih za odgovarajuće ulazno/izlazne operacije. Date su sledeće deklaracije:

```
// I/O request:
struct BlockIORequest {...};

// I/O operation:
typedef int (*BlockIOOp) (BlockIORequest*);

// I/O driver vector table:
struct BlockIOVectorTable {
  BlockIOOp open, close, read, write;
};

// I/O device driver map:
BlockIOVectorTable* blockIOMap[MAXDEVICES];
extern int numOfRegisteredDrivers;
typedef int HANDLE;
```

a. Implementirati sledeće funkcije sistema kojima se zadaju operacije uređaju datom prvim argumentom:
```
int open  (HANDLE device, BlockIORequest* req);
int close (HANDLE device, BlockIORequest* req);
int read  (HANDLE device, BlockIORequest* req);
int write (HANDLE device, BlockIORequest* req);
```

b. Implementirati operaciju za registraciju drajvera koja vraća identifikator uređaja koji će se koristiti kao „ručka“ u daljim operacijama sa tim uređajem, a predstavlja broj ulaza u tabeli regostrovanih uređaja (-1 u slučaju greške):
```
HANDLE blockIODriverReg (BlockIOVectorTable*);
```

Rešenje:

a. implementacija traženih funkcija:

```
int open (HANDLE device, BlockIORequest* req) {
   //ako drajver sa indeksom "device" ne postoji, vracamo se iz f-je
   if (device >= numOfRegisteredDrivers) return -1;

   //dohvatamo tabelu sa adresama funkcija za traženi drajver
   BlockIOVectorTable* vt = blockIOMap[device];
   //ako tabela ne postoji, vracamo se iz f-je
   if (vt == 0) return -1;
   //u suprotnom dohvatamo pokazivač na funkciju "open"
   BlockIOOp op = vt->open;
   //ako funkcija "open" ne postoji vracamo se iz f-je
   if (op == 0) return -1;
   //u suprotnom pozivamo f-ju "open" preko pokazivača
   //i vraćamo njenu povratnu vrednost kao rezultat f-je
   return (*op)(req);
}
```

Analogno za ostale funkcije (svako "open" treba da se zameni sa imenom funkcije koja se implementira).

b. implementacija tražene funkcije:

```
HANDLE blockIODriverReg (BlockIOVectorTable* vt) {
   //ako je tabela registrovanih drajvera popunjena do kraja
   //onda vracamo kao rezultat f-je -1 (greška, drajver nije registrovan)
   if (numOfRegisteredDrivers >= MAXDEVICES)
      return -1;

   //u suprotnom, vršimo instalaciju drajvera upisujući u prvi
   //slobodan ulaz adresu tabele sa adresama f-ja tog drajvera
   blockIOMap[numOfRegisteredDrivers] = vt;

   //kao rezultat vraćamo broj ulaza u tabeli registrovanih drajvera
   //koji odgovara novoregistrovanom drajveru, a ujedno uvećavamo i broj
   //registrovanih drajvera za jedan (registrovali smo jedan novi drajver)
   return numOfRegisteredDrivers++;
}
```

2. (*Februar 2011*) Implementirati internu nit jezgra, zajedno sa odgovarajućom prekidnom rutinom, koja uzima jedan po jedan zahtev za ulaznom operacijom na nekom ulaznom uređaju iz ograničenog bafera i obavlja prenos bloka podataka zadat tim zahtevom korišćenjem dva raspoloživa kanala nekog *DMA* uređaja. Ova dva kanala omogućavaju dva uporedna prenosa sa istog uređaja. Ograničeni bafer, realizovan klasom `InputBuffer`, u sebi ima implementiranu sinhronizaciju proizvođača i potrošača. Zahtev ima sledeću strukturu:

```
struct InputRequest {
  char* buffer;         // Buffer with data (block)
  unsigned int size;    // Buffer (blok) size
  Semaphore* toSignal;  // Semaphore to signal on request completion
};
```

Kada se završi prenos zadat zahtevom, potrebno je signalizirati semafor na koga ukazuje `InputRequest::toSignal` i obrisati zahtev. Date su deklaracije pokazivača preko kojih se može pristupiti registrima *DMA* uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* dmaCtrl[2] =...;         // control registers for 2 DMA channels
REG* dmaStatus[2] =...;       // status registers for 2 DMA channels
REG* dmaBlkAddr[2] =...;      // data block address registers for 2 DMA channels
REG* dmaBlkSize[2] =...;      // data block size registers for 2 DMA channels
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće prenos na datom *DMA* kanalu, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je dati *DMA* kanal spreman za novi prenos podatka (inicijalno je tako postavljen). Postavljanje bilo kog od ova dva bita spremnosti kada neki kanal završi prethodni prenos generiše signal zahteva za prekid na istom *IRQ* ulazu.
Na raspolaganju za implementaciju potrebne sinhronizacije između prekidne rutine i niti su standardni brojački semafori, čija se operacija *signal* može pozivati iz prekidne rutine.

Rešenje:

Implementacija interne niti jezgra:

```
//klasa InputTread treba da bude "singleton", jer nema smisla
//imati više od jednog aktivnog objekta ove klase (jedna instanca
//već kontroliše prenos zahteva na oba kanala DMA kontrolera)
class InputThread : public Thread {
   public:
      //semafor endOfTransfer je inicijalno postavljen na 2, jer
      //postoje dva raspoloživa kanala DMA kontrolera koji ga koriste
      InputThread () : Thread (), endOfTransfer(2) {}

   protected:
      virtual void run ();

   private:
      //prekidna rutina je prijateljska funkcija klase InputThread
      //jer je potrebno da ima pristup semaforu endOfTransfer
      friend void ìnputThreadInterrupt();

      //zahtevi za ulaznim operacijama za oba kanala DMA kontrolera
      static InputRequest* pendingRequest[2];

      //semafor kojim se signalizira završetak prenosa jednog od
      //dva kanala DMA kontrolera (oba koriste samo ovaj semafor)
      static Semaphore endOfTransfer;
};
```

```
void InputThread::run () {
   //pri pokretanju niti nijedan zahtev još nije poslat DMA
   //kontroleru, pa odgovarajuće pokazivače postavljamo na 0 (null)
   pendingRequest[0] = 0;
   pendingRequest[1] = 0;

   //pomoćna promenljiva
   int i = 0;

   //beskonačna petlja - nit se prekida jedino ako dođe do greške
   while (1) {
      //čekamo da DMA završi prenos zadat nekom od dva kanala
      //i postane spreman za prenos narednog bloka podataka
      endOfTransfer.wait();

      //ispitujemo koji DMA kanal je spreman
      for (i = 0; i < 2; i++)
         if (*dmaStatus[i] & 1) break;

      //neočekivana vrednost (postoje samo kanali 0 i 1) - prekid rada
      if (i >= 2) return;

      //ako je, pre nego što je postao spreman, preko tog DMA kanala
      //vršen prenos bloka podataka, treba da signaliziramo semafor
      //koji je specificiran u samoj strukturi obrađenog zahteva
      if (pendingRequest[i])
      {
         //ako je semafor koji treba da signaliziramo zadat
         //(nije null), onda pozivamo njegovu operaciju signal()
         if (pendingRequest[i]->toSignal)
            pendingRequest[i]->toSignal->signal();

         //brišemo obrađeni zahtev iz memorije
         delete pendingRequest[i];
         //i ažuriramo pokazivač na tekući zahtev
         pendingRequest[i]=0;
      }

      //dohvatamo novi zahtev za ulaznom operacijom iz ograničenog bafera,
      //postavljamo ga za tekući, inicijalizujemo DST i CNT registar odgovarajućeg
      //kanala DMA kontrolera i startujemo njegov rad upisom 1 u CR registar
      pendingRequest[i] = InputBuffer::get();
      *dmaBlkAddress[i] = pendingRequest[i]->buffer;
      *dmaBlkSize[i] = pendingRequest[i]->size;
      *dmaCtrl[i] = 1;
   }
}

//u prekidnoj rutini DMA kontrolera je potrebno signalizirati
//semafor endOfTransfer da bi javili niti da je prenos završen
interrupt void inputThreadInterrupt () {
  InputThread::endOfTransfer.signal();
}
```

3. (*Jun 2011*) Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja element tipa `Data*` pozivom operacije `put()`, čime se element stavlja u trenutni „izlazni“ bafer od dva interna bafera, dok potrošač uzima elemente iz trenutnog „ulaznog“ bafera pozivom operacije `get()`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Za sinhronizaciju koristiti semafore.

```
class Data;

class DoubleBuffer {
  public:
    DoubleBuffer (int size);
    void  put (Data*);
    Data* get ();
  private:
    ...
};
```

Rešenje:

```
class Data;

class DoubleBuffer {
   public:
      DoubleBuffer (int size);
      void  put (Data*);
      Data* get ();

   private:
      Semaphore inputBufReady, outputBufReady;
      Data* buffer[2];
      int size, head, tail, slots, items, inputBuf, outputBuf;
};


DoubleBuffer::DoubleBuffer (int sz) : inputBufReady(1), outputBufReady(0) {
   buffer[0] = new (Data*)[sz];
   buffer[1] = new (Data*)[sz];
   size = sz;
   head = tail = 0;
   slots = size;
   items = 0;
   inputBuf = 0;
   outputBuf = 1;
}

void DoubleBuffer::put (Data* d) {
   if (slots == 0) {
      inputBufReady.wait();
      outputBuf = !outputBuf;
      slots = size;
      tail = 0;
   }
   buffer[outputBuf][tail++] = d;
   slots--;
   if (slots == 0)
      outputBufReady.signal();
```

```
}

Data* DoubleBuffer::get () {
   if (items == 0) {
      outputBufReady.wait();
      inputBuf = !inputBuf;
      items = size;
      head = 0;
   }
   Data* d = buffer[inputBuf][head++];
   items--;
   if (items == 0)
      inputBufReady.signal();
   return d;
}
```

4. (*Septembar 2011*) Dat je interfejs klase koja apstrahuje jedan sekvencijalni, blokovski, ulazni uređaj sa blokom veličine `BlockSize` znakova:

```
class BlockDevice {
   public:
      void open ();                       // Open the device channel
      void close ();                      // Close the device channel
      void loadBlock (char* buffer);       // Read next block to the given buffer
};
```

Implementirati klasu `CharDevice` sa dole datim interfejsom koja adaptira interfejs datog blokovskog ulaznog uređaja u interfejs sekvencijalnog, ali znakovnog ulaznog uređaja. Konstruktor i destruktor ove klase treba da otvaraju, odnosno zatvaraju dati blokovski uređaj.

```
class CharDevice {
   public:
      CharDevice (BlockDevice* bd);
      ~CharDevice ();
      char getChar ();
};
```

Korišćenjem klase `CharDevice` ilustrovati primerom učitavanje niza znakova sa ulaznog uređaja sve dok se ne učita znak `'\0'`.

Rešenje:

```
class CharDevice {
   public:
      //konstruktor klase CharDevice
      CharDevice (BlockDevice* bd) : myDev(bd), cursor(BlockSize) {
         //ako je zadati pokazivač na uređaj različit od null,
         //otvaramo zadati blokovski uređaj pozivom metode open()
         if (myDev) myDev->open();
      }

      //destruktor klase CharDevice
      ~CharDevice () {
         //ako je zadati pokazivač na uređaj različit od null,
         //zatvaramo zadati blokovski uređaj pozivom metode close()
         if (myDev) myDev->close();
      }

      //metoda za dohvatanje jednog znaka sa zadatog uređaja
      char getChar ();


   protected:
      //pomoćna metoda za učitavanje bloka podataka sa uređaja
      void load ();


   private:
      //pokazivač na zadati blokovski ulazni uređaj
      BlockDevice* myDev;
      //bafer za smeštanje bloka učitanog sa uređaja
      char buffer[BlockSize];
      //indeks narednog podatka koji se čita iz bafera
      int cursor;
};
```

```
void CharDevice::load () {
   //ako je ulazni uređaj zadat i ako smo ispraznili bafer
   if (myDev  &&  cursor == BlockSize) {
       //učitavamo novi blok podataka sa uređaja i smeštamo ga u bafer
       myDev->loadBlock(buffer);
       //ažuriramo vrednost indeksa podatka za prenos iz bafera
       cursor = 0;
   }
}

char CharDevice::getChar () {
   //pozivamo load() da bismo učitali naredni blok ako je bafer prazan
   load();
   //ako nije došlo do greške, uzimamo naredni podatak iz bafera,
   //ažuriramo indeks narednog podatka za prenos i vracamo rezultat
   if (cursor < BlockSize) return buffer[cursor++];
   //u suprotnom, ako je došlo do greške, kao rezultat vraćamo '\0'
   else return '\0';
}


void main () {
   //definicija jednog blokovskog uređaja
   BlockDevice bd = ...;

   //znakovni uređaj koji adaptira gornji blokovski uređaj
   CharDevice input(bd);

   //čitamo podatke sa uređaja, znak po znak, sve dok ne pročitamo '\0'
   for (char c=input.getchar(); c!='\0'; c=input.getchar())
    ...
}
```

5. (*Januar 2010*) Na nekom računaru sa multiprogramskim operativnim sistemom neki kontroler periferije nema vezan svoj signal završetka operacije na ulaze za prekid procesora, ali je poznato vreme završetka operacije zadate tom uređaju u najgorem slučaju (maksimalno vreme završetka operacije). Za obradu ulazno/izlaznih operacija zadatih tom uređaju brine se poseban proces tog operativnog sistema. Kakvu uslugu treba operativni sistem da obezbedi, a ovaj proces da koristi, da bi se izbegla tehnika kojom se može detektovati završetak operacije čitanjem statusnog registra i ispitivanjem bita završetka operacije (*polling*), tj. da bi se izbeglo uposleno čekanje (*busy waiting*)? Precizno opisati kako ovaj proces treba da koristi ovu uslugu.

Rešenje:

Operativni sistem treba da obezbedi uslugu suspenzije procesa na zadati interval realnog vremena („uspavljivanje"). Kada zada operaciju uređaju, ovaj proces treba da se suspenduje na vreme koje je veće ili jednako maksimalnom trajanju operacije na uređaju. Nakon isteka tog vremena, kada se proces deblokira, operacija je sigurno završena i proces može da nastavi sa zadavanjem nove operacije.

napomena: pogledati kod iz zadatka 9 iz prve grupe zadataka

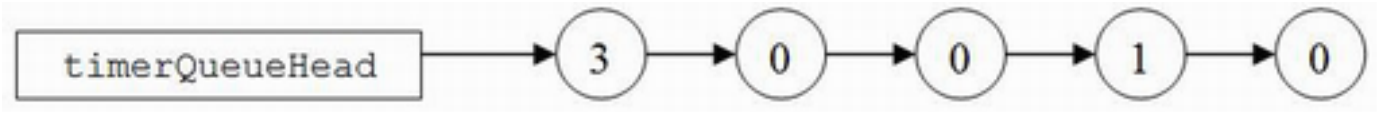
6. (*Februar 2010*) U nekom sistemu podržan je samo asinhroni izlaz na izlazni uređaj pomoću sledećeg *API*. Funkcija
`IOReqID output (IODevID deviceID, IOReq* request);`
zadaje (asinhrono) izlaznu operaciju specifikovanu drugim argumentom na uređaju identifikovanom prvim argumentom. Ova funkcija odmah vraća kontrolu pozivaocu, uz identifikaciju zadate operacije (rezultat tipa `IOReqID` je veći od 0 u slučaju ispravno zadatog zahteva). Funkcija
`void ioWait (IOReqID);`
blokira pozivajući proces sve dok operacija identifikovana argumentom nije završena u potpunosti.
Pomoću ovih funkcija realizovati funkciju koja, u odnosu na jedan argument, može zadati operaciju sinhrono ili asinhrono, prema želji pozivaoca.

Rešenje:

Realizacija tražene funkcije:

```
IDReqID output (IODevID deviceID, IOReq* request, int sync) {
   //zadajemo asinhrono izlaznu operaciju pozivom f-je output()
   //i njenu povratnu vrednost smeštamo u promenljivu ioReqID
   IOReqID ioReqID = output(deviceID,request);

   //ako je zatraženo da se operacija realizuje sinhrono (sync je 1), onda treba
   //da blokiramo pozivajući proces, pozivom f-je ioWait(), sve dok se zadata
   //operacija ne završi (naravno, to ima smisla raditi jedino ako je operacija
   //ispravno zadata, tj ako je f-ja output() vratila vrednost veću od nule)
   if (sync  &&  ioReqID > 0) ioWait(ioReqID);

   //vraćamo pozivaocu ID zadate operacije
   return ioReqID;
}
```

7. (*Jun 2010*) U nekom sistemu obezbeđena je podrška procesima za usluge merenja vremena i uspavljivanja procesa na zadato vreme. U podsistemu za merenje vremena ovakvi zahtevi smeštaju se u jednostruko ulančanu listu elemenata tipa `TimerRequest` čija je deklaracija data dole. Na početak ove liste zahteva ukazuje globalni pokazivač `timerQueueHead`. Svaki zahtev tipa `TimerRequest` sadrži pokazivač event na događaj na kome je blokiran proces koji se uspavao na zadato vreme i koji treba „probuditi“ (signaliziranjem tog događaja) kada to vreme dođe. Poseban hardverski uređaj (tajmer) generiše periodične prekide. Zahtevi se u red smeštaju po hronološkom redosledu, tako da u svakom zahtevu polje `time` predstavlja dužinu intervala (izraženu u jedinici vremena koja je jednaka periodi periodičnog prekida sa hardverskog tajmera) koji treba da protekne od „buđenja“ prethodnog zahteva u listi do trenutka „buđenja“ posmatranog zahteva u listi. Prvi zahtev u listi treba „probuditi“ za time perioda u odnosu na trenutak poslednjeg prekida. Na primer, za red zahteva na donjoj slici, prva tri zahteva treba probuditi za 3 „otkucaja“ tajmera, a četvrti i peti za 1 nakon njih, odnosno za 4 od poslednjeg otkucaja.




```
struct TimerRequest {
  TimerRequest* next;          // Next in the list of timer requests
  unsigned long int time;
  Event* event;
};

extern TimerRequest* timerQueueHead;
```

Realizovati prekidnu rutinu `timer()` koja se poziva na svaki periodični prekid od vremenskog brojača i koja treba da odradi odgovarajući posao ažuriranja liste zahteva i buđenja procesa kojima je isteklo vreme. Operacija `signal()` događaja može da se pozove iz prekidne rutine, jer je za to i predviđena. Prekidna rutina je uvek atomična, jer se radi o jednoprocesorskom sistemu koji hardverski maskira druge prekide pri prihvatanju prekida. Nakon obrade zahteva, ne treba ga dealocirati iz memorije, pošto je to odgovornost onoga ko je zahtev postavio.

Rešenje:

Realizacija tražene prekidne rutine:

```
interrupt void timer () {
   //ako je lista zahteva prazna, vraćamo se, jer nemamo šta da radimo
   if (timerQueueHead == 0) return;

   //u prekidnu rutinu ulazimo kada protekne jedna perioda hardverskog tajmera,
   //pa smanjujemo dužinu intervala čekanja prvog zahteva u listi za jednu periodu
   timerQueueHead->time--;

   //dok god ima zahteva na početku liste čije je vreme čekanja isteklo
   //i dok god ne dođemo do kraja liste zahteva
   //(moguće je da je vreme čekanja svih zahteva u listi isteklo)
   while (timerQueueHead && timerQueueHead->time==0) {
      //”buđenje” prvog zahteva u listi pozivom signal()
      timerQueueHead->event->signal();
      //izbacujemo “probuđeni” zahtev iz liste zahteva
      timerQueueHead = timerQueueHead->next;
  }
}
```

8. (*Septembar 2010*) Tehnikom baferisanja potrebno je omogućiti blokovski pristup nekom znakovno orijentisanom ulaznom uređaju. U nastavku je dat interfejs (deo definicije) klase `Buffer` koja obezbeđuje ovu adaptaciju. `Capacity` je kapacitet bafera, dok je `BlockSize` veličina bloka. Jedna interna nit jezgra učitava jedan po jedan znak sa ulaznog uređaja i upisuje u bafer pozivom operacije `put()` ove klase. Korisničke niti mogu da pozivaju operaciju `get()` kojom u svoj memorijski bafer, na koga ukazuje argument `buf`, mogu da preuzmu jedan ceo blok učitanih znakova veličine `BlockSize`.

```
const int Capacity = ..., BlockSize = ...;      // Capacity>BlockSize

class Buffer {
  public:
    Buffer ();
    void put (char c);
    void get (char* buf);
    ...
};
```

Realizovati u potpunosti klasu `Buffer` uz svu potrebnu sinhronizaciju, korišćenjem standardnih brojačkih semafora.

Rešenje:

```
const int Capacity = ..., BlockSize = ...;

class Buffer {
   public:
      //konstruktor klase Buffer
      Buffer () : head(0), tail(0), count(0),
                  mutex(1), blockAvailable(0), spaceAvailable(Capacity) {}
      void put (char c);
      void get (char* buf);

   private:
      //trenutni broj podataka u bloku koji se formira
      int count;
      int head, tail
      //bafer za podatke
      char buffer[Capacity];

      //semafori za kritične sekcije, signalizaciju dostupnosti bloka
      //i za signalizaciju da li u baferu ima slobodnog prostora
      Semaphore mutex, blockAvailable, spaceAvailable;
};

void Buffer::put (char c) {
   //čekamo ako je bafer pun
   spaceAvailable.wait();
   //zaključavamo kritičnu sekciju
   mutex.wait();
   //ubacujemo zadati podatak u bafer
   buffer[tail] = c;
   //ažuriramo vrednost pokazivača na kraj reda
   tail = (tail + 1) % Capacity;
   //uvećavamo broj podataka u trenutno formirajućem bloku
   count++;
   //ako je blok formiran, signaliziramo odgovarajući semafor
   //i krećemo sa formiranjem novog bloka postavljanjem "count" na 0
   if (count==BlockSize) { count=0; blockAvailable.signal(); }
   //otključavamo kritičnu sekciju
```

```
    mutex.signal();
}
```

```
void Buffer::get (char* buf) {
   //čekamo ako nijedan blok još nije formiran
   blockAvailable.wait();
   //zaključavamo kritičnu sekciju
   mutex.wait();
   //za svaki podatak iz bloka koji prebacujemo
   for (int i=0; i<BlockSize; i++) {
      //šaljemo podatak na periferiju
      buf[i] = buffer[head];
      //ažuriramo vrednost pokazivača na početak reda
      head = (head + 1) % Capacity;
      //oslobodili smo jedno mesto u baferu, pa to
      //signaliziramo na odgovarajućem semaforu
      spaceAvailable.signal();
   }
   //otključavamo kritičnu sekciju
   mutex.signal();
}
```

9. (*Oktobar 2010*) Implementirati internu nit jezgra, zajedno sa odgovarajućom prekidnom rutinom, koja uzima jedan po jedan zahtev za izlaznom operacijom na nekom izlaznom znakovnom uređaju iz ograničenog bafera i obavlja prenos bloka znakova zadat tim zahtevom korišćenjem mehanizma prekida.
Ograničeni bafer, realizovan klasom `OutputBuffer`, u sebi ima implementiranu sinhronizaciju proizvođača i potrošača.
Zahtev ima sledeću strukturu:

```
struct OutputRequest {
  char* buffer;               // Buffer with data (block)
  int size;                   // Buffer (block) size
  Semaphore* toSignal;        // Semaphore to signal on request completion
};
```

Kada se završi prenos zadat zahtevom, potrebno je signalizirati semafor na koga ukazuje `OutputRequest::toSignal` i obrisati zahtev. Date su deklaracije pokazivača preko kojih se može pristupiti registrima uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* ioCtrl =...;             // control register
REG* ioStatus =...;           // status register
char* ioData =...;            // data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke.
Na raspolaganju za implementaciju potrebne sinhronizacije između prekidne rutine i niti su binarni semafori – događaji, čija se operacija `signal` može pozivati iz prekidne rutine.

Rešenje:

```
//klasa OutputThread treba da bude singleton
//jer nema potrebe za više instanci ove klase
class OutputThread : public Thread {
   public:
      OutputThread () : Thread (), pendingRequest(0),
                        index(0), endOfTransfer(0) {}

   protected:
      virtual void run ();

   private:
      //prekidna rutina je prijateljska f-ja klase OutputThread
      //jer je potrebno da može da pristupi semaforu endOfTransfer
      //i trenutnom zahtevu za izlaznom operacijom (pendingRequest)
      friend void outputThreadInterrupt;

      //trenutni zahtev za izlaznom operacijom koji se obrađuje
      static OutputRequest* pendingRequest;

      //broj trenutno prenetih podataka (znakova) iz bloka
      //trenutno obrađivanog zahteva za izlaznom operacijom
      static int index;

      //semafor (dogadjaj) koji se signalizira u prekidnoj rutini
      //kada se izvrši prosleđeni zahtev za izlaznom operacijom
      static Event endOfTransfer;
};
```

```
void OutputThread::run () {
   while (1) {
      //uzimamo jedan zahtev za izlaznom operacijom iz bafera
      pendingRequest = OutputBuffer::get();

      //ako zahtev ne postoji ili ako je veličina bloka za prenos nula,
      //preskačemo ostatak operacije i uzimamo novi zahtev iz ograničenog bafera
      if (pendingRequest==0 || pendingRequest->size==0)
            continue;

      //broj prenetih podataka iz bloka postavljamo na nulu
      index = 0;

      //startujemo uređaj
      *ioCtrl=1;

      //čekamo da se završi zadati prenos celog bloka
      endOfTransfer.wait();

      //ako je semafor koji treba da se signalizira različit od null,
      //izvršavamo operaciju signal() nad tim semaforom
      if (pendingRequest->toSignal) pendingRequest->toSignal->signal();

      //brišemo prethodno obrađeni zahtev
      delete pendingRequest;
  }
}

interrupt void outputThreadInterrupt () {
   //šaljemo uređaju naredni podatak iz bloka i uvećavamo
   //vrednost indeksa tako da pokazuje na naredni podatak
   *ioData=OutputThread::pendingRequest->buffer[OutputThread::index++];

   //ako smo preneli sve podatke iz bloka
   if (index >= OutputThread::pendingRequest->size) {
      //zausavljamo uređaj
      *ioCtrl = 0;
      //i signaliziramo internoj niti jezgra da je prenos bloka gotov
      OutputThread::endOfTransfer.signal();
   }
}
```

10. (*Januar 2009)* U školskom jezgru realizovati klasu `IOStream`. Svaki objekat ove klase treba da obavi zadatu ulaznu ili izlaznu operaciju sa zadatom periferijom korišćenjem mehanizma prekida. Novi objekat ove klase treba inicijalizovati sledećim parametrima: broj ulaza prekida koji generiše ova periferija, adresa upravljačkog registra, adresa registra za podatke periferijskog uređaja, vrsta operacije (ulaz ili izlaz), adresa memorijskog bafera sa podacima tipa `unsigned int` koje treba preneti, veličina bafera i pokazivač na semafor koji treba signalizirati po završetku operacije (može biti i *null*, kada ne treba signalizirati završetak). U upravljačkim registrima periferija najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke; postavljanje ovog registra generiše prekid procesoru. Svi registri su veličine jedne mašinske reči (tip `unsigned int)` i preslikani su u memorijski adresni prostor, pa im se može pristupiti uobičajenim operacijama čitanja ili upisa preko pokazivača koji imaju vrednosti njihovih adresa.
Jedan ulaz u interapt vektor tabeli inicijalizuje se pozivom:
`void init_IVT (int intNumber, interrupt void (*intRoutine)(int));`
Prilikom poziva prekidne rutine, procesor na stek stavlja broj ulaza tog prekida koga prekidna rutina vidi kao svoj (jedini) argument tipa `int`. Iz prekidne rutine dozvoljeno je pozivati operaciju `signal` semafora. Smatrati da u jednom trenutku za jednu periferiju postoji najviše jedan zahtev.

Rešenje:

```
typedef unsigned int REG;
enum IOKind { in, out };            //tip periferije

const int NumOfInterrupts = ...     //ukupan broj ulaza u IVT

struct IOStreamParams {       //struktura koja sadrži inicijalizacione parametre
  REG *ctrl, *data, *buf;
  IOKind knd;
  int size;
  int i;
  Semaphore* sem;
}

IOStreamParams ioParams[NumOfInterrupts];        //niz skupa parametara za svaki ulaz

interrupt void ioStreamIntRoutine (int intNo);   //deklaracija prekidne rutine

class IOStream {
public:
  IOStream (int intNo, REG* ctrl, REG* data, IOKind knd,
            REG* buf, int size, Semaphore* sem = 0);
};

IOStream::IOStream (int _intNo, REG* _ctrl, REG* _data,
                  IOKind _knd, REG* _buf, int _size, Semaphore* _sem = 0)
{
   //inicijalizujemo zadati ulaz u IVT pozivom f-je init_IVT()
   init_IVT(_intNo,ioStreamIntRoutine);
   //inicijalizujemo parametre prenosa za periferiju sa tim ulazom
   ioParams[_intNo].ctrl = _ctrl;
   ioParams[_intNo].data = _data;
   ioParams[_intNo].knd = _knd;
   ioParams[_intNo].buf = _buf;
   ioParams[_intNo].size =_size;
   ioParams[_intNo].sem = _sem;
   ioParams[_intNo].i = 0;
   //i na kraju startujemo periferiju
   *_ctrl=1;
```

```
}
```

```
interrupt void ioStreamIntRoutine (int intNo) {
   //ako je periferija čiji je broj ulaza u IVT jedan intNo ulazna,
   //učitavamo sa nje podatak i smeštamo ga na naredno mesto u baferu
   if (ioParams[intNo].knd == in)
      ioParams[intNo].buf[ioParams[intNo].i] = *ioParams[intNo].data;

   //ako je periferija čiji je broj ulaza u IVT jedan intNo izlazna,
   //čitamo naredni podatak iz bafera i šaljemo ga periferiji
   if (ioParams[intNo].knd == out)
      *ioParams[intNo].data = ioParams[intNo].buf[ioParams[intNo].i];

   //ako nismo preneli sve podatke (ceo blok), izlazimo iz prekidne rutine
   if (++ioParams[intNo].i < ioParams[intNo].size) return;

   //u suprotnom, zaustavljamo periferiju
   *ioParams[intNo].ctrl = 0;

   //i, ako je vrednost semafora za signalizaciju različita
   //od null, pozivamo operaciju signal() nad tim semaforom
   if (ioParams[intNo].sem)
      ioParams[intNo].sem->signal();
}
```

11. (*Februar 2009)* U nekom disk-podsistemu postoje dva istovetna fizička diska, svaki sa svojim posebnim *DMA* kanalom, koji imaju identičan sadržaj, odnosno koji sadrže identične kopije istog sadržaja, pa se spolja logički posmatraju kao jedan jedinstven disk. Smisao ove konfiguracije jeste povećanje pouzdanosti celog disk-podsistema: ukoliko otkaže jedan fizički disk, sadržaj je raspoloživ na drugom sve dok se onaj pokvareni ne zameni. Precizno objasnite kako zahteve za čitanje i zahteve za upis na logički disk treba usmeravati na fizičke diskove. Obratiti pažnju da se može ostvariti i poboljšanje performansi (povećanje propusnosti ukupnog disk-podsistema) pri operacijama čitanja.

Rešenje:

Svaki zahtev za upis na logički disk treba proslediti ka oba fizička diska (odnosno oba DMA kanala), kako bi se ažurirale obe kopije i time očuvala njihova identičnost. Zahtev za čitanje sa logičkog diska može da se usmeri na bilo koji fizički disk, jer se traženi sadržaj može dobiti od bilo koga (oba imaju identičan sadržaj). Radi povećanja propusnosti, zahteve za čitanje treba ravnomerno raspoređivati na fizičke diskove, npr. prosto naizmenično ili nekim sofisticiranijim algoritmom raspoređivanja, tako da fizički diskovi mogu da različite zahteve za čitanje opslužuju paralelno.

12. (*Jun 2009)*

a. Korišćenjem operacije:
   `void putc(Device*, char);`
   koja na dati znakovno orijentisani izlazni uređaj izbacuje jedan znak, realizovati operaciju
   `void writeln(Device*, const char*);`
   koja na dati uređaj izbacuje dati niz znakova u jednoj liniji (i liniju završava znakom za novi red).
b. Posmatra se neki sekvencijalni, izlazni, blokovski orijentisani uređaj sa blokom veličine *N* i operacijom koja na uređaj izbacuje dati blok:
   `void output(const char* block);`
   Korišćenjem ove operacije, realizovati operaciju
   `void putc(char);`
   koja izbacuje jedan znak na dati uređaj, uz baferisanje (znak po znak se upisuje u bafer, a kada se bafer napuni, izbacuje se ceo bafer kao blok).

Rešenje:

a. Realizacija tražene operacije:

```
void writeln(Device* d, const char* s) {
   //svaki znak iz niza "s" ispisujemo redom na uređaj
   for (;*s; s++) putc(d,*s);

   //na kraju ispisujemo i "\n" da bi prešli u novi red
   putc(d,'\n');
}
```

b. Realizacija tražene operacije:

```
void putc(char c) {
   //bafer veličine "N" znakova
   static char buffer[N];
   //indeks prve slobodne lokacije u baferu
   static int cursor = 0;

   //ubacujemo zadati znak u bafer
   buffer[cursor++] = c;

   //ako je bafer pun, treba izbaciti ceo blok na uređaj
   if (cursor == N)
   {
      //izbacivanje bloka na uređaj
      output(buffer);
      //praznimo bafer
      cursor=0;
   }
}
```

13. (*Septembar 2009)*

a. Umesto tehnike *spooling*, pristup uporednih procesa štampaču u nekom sistemu omogućuje se međusobnim isključenjem pomoću operacije rezervacije: svaki proces, pre nego što zatraži bilo koju operaciju sa štampačem, mora da ga rezerviše, a tu rezervaciju otpušta tek kada završi sa celim jednim poslom štampe. Šta je nedostatak ove tehnike?
b. Neki uređaj koji proizvodi podatke je blokovski uređaj i podatke isporučuje u blokovima veličine 256 bajtova. Drugi uređaj na koji treba slati ove podatke je znakovni uređaj. Kojom tehnikom je potrebno adaptirati ove različite veličine jedinica prenosa?

Rešenje:

a. Problem je u veoma slaboj konkurentnosti: svi drugi procesi koji su došli do stanja u kome bi želeli da pokrenu svoj posao štampe moraju da čekaju dok proces koji je rezervisao štampač ne završi ceo svoj posao. Time su poslovi štampe od različitih procesa potpuno sekvencijalizovani, a mogu da traju relativno dugo. Zbog toga je odziv sistema veoma loš, a zadržavanje procesa u sistemu značajno produženo.

b. Tehnikom baferisanja.

14. (*Oktobar 2009)* Na nekom računaru sa multiprogramskim operativnim sistemom neki kontroler periferije nema vezan svoj signal završetka operacije na ulaze za prekid procesora, ali je poznato vreme završetka operacije zadate tom uređaju u najgorem slučaju (maksimalno vreme završetka operacije). Za obradu ulazno/izlaznih operacija zadatih tom uređaju brine se poseban proces tog operativnog sistema. Kakvu uslugu treba operativni sistem da obezbedi, a ovaj proces da koristi, da bi se izbegla tehnika kojom se može detektovati završetak operacije čitanjem statusnog registra i ispitivanjem bita završetka operacije (*polling*), tj. da bi se izbeglo uposleno čekanje (*busy waiting*)? Precizno opisati kako ovaj proces treba da koristi ovu uslugu.

Rešenje:

Potrebno je obezbediti sistemsku uslugu (poziv) „uspavljivanja" (suspenzije) procesa na zadato vreme. Opisani proces bi, nakon pokretanja ulazno/izlazne operacije, pozvao ovu uslugu i suspendovao se na vreme zaveršetka ove operacije u najgorem slučaju. Kada se ponovo aktivira, proces može da ispita samo uspešnost završetka zadate operacije, odnosno postojanja eventualne greške, a ne i spremnost uređaja za novu operaciju.

15. (*Januar 2008)*

a. Koja tehnika omogućava da se upis na neki izlazni, znakovno orijentisani uređaj sa strane procesa vrši kao blokovski upis? Precizno obrazložiti odgovor.
b. Koja tehnika omogućava da zahtev za štampanjem koji je dat pre isključenja računara bude obrađen nakon ponovnog uključenja računara? Precizno obrazložiti odgovor.

Rešenje:

a. Baferisanje, po principu „proizvođač-potrošač". Korisnički proces upisuje u bafer blok podataka (kao „proizvođač"), dok poseban proces koji kontroliše uređaj (kao „potrošač") iz bafera uzima znak po znak i neposredno ga šalje na uređaj.

b. *Spooling*. Zahtev je upisan u fajl koji se smešta na određeno mesto (u određeni sistemski direktorijum). Taj fajl ostaje na istom mestu i po isključenju računara. Po ponovnom uključenju računara, pokreće se *spooler* koji pronalazi fajlove na ovom mestu i jedan po jedan obrađuje slanjem na štampač.

16. (*Februar 2008)* Potrebno je realizovati drajver (*device driver*) za jedan izlazni, znakovno orijentisani uređaj, pri čemu korisnički proces može zadati prenos čitavog niza znakova koje će *DMA* upisivati na uređaj znak po znak. Korisnički proces koji zadaje izlaznu operaciju svoj zahtev predstavlja sledećom strukturom:

```
struct DeviceRequest {
   char* block;               // Pokazivač na niz znakova koje treba preneti na uređaj
   unsigned long size;        // Veličina niza znakova
   Semaphore* toSignal;       // Semafor koji treba signalizirati po završetku operacije
   unsigned int status;       // Status izvršene operacije dobijen od uređaja
};
```

Korisnički proces svoj zahtev zapisan u ovakvoj strukturi (osim polja status koje upisuje drajver po završenoj operaciji) upisuje u ograničeni bafer koji predstavlja red postavljenih zahteva, a potom se blokira na standardnom brojačkom semaforu na koji ukazuje pokazivač `toSignal`. Ograničenom baferu pristupa se sledećim operacijama koje su blokirajuće ukoliko je bafer pun, odnosno prazan:

```
void putDeviceRequest (DeviceRequest*);   // Stavlja jedan zahtev u red
DeviceRequest* getDeviceRequest();        // Uzima jedan zahtev iz reda
```

Sama izlazna operacija obavlja se preko *DMA* uređaja kome se prenos niza znakova `block` veličine `size` iz memorije na uređaj zadaje i pokreće sledećom neblokirajućom operacijom:

```
void DMARequest (char* block, unsigned long size);
```

Kada završi prenos bloka podataka, *DMA* ne generiše prekid, već postavlja najniži bit svog statusnog registra na 1. Vrednost statusnog registra *DMA* može se dobiti operacijom:

```
unsigned int DMAStatus ();
```

Korišćenjem navedenih operacija (za koje se pretpostavlja da su implementirane) i koncepta niti, sa interfejsom kao što je definisano projektnim zadatkom za domaći rad, napisati kompletan kod ovog drajvera uređaja.

Rešenje:

```
#include "thread.h"

extern DeviceRequest* getDeviceRequest();
extern void DMARequest(char* block, unsigned long size);
extern unsigned int DMAStatus();

class DeviceDriver : public Thread {
   protected:
      DeviceDriver () : Thread() { start(); }
      virtual void run ();
   private:
      static DeviceDriver instance;
};

DeviceDriver DeviceDriver::instance;            //klasa DeviceDriver je singleton

void DeviceDriver::run () {
   while (1) {
       //uzimamo jedan zahtev iz reda zahteva
       DeviceRequest* dr = getDeviceRequest();
       //pokrećemo operaciju prosleđivanjem zahteva DMA kontroleru
       DMARequest(dr->block,dr->size);
       //čekamo da DMA završi prenos (ispitujemo bit spremnosti)
       unsigned int status = DMAStatus();
       while ((status & 1) == 0) status = DMAStatus();
       //ažuriramo vrednost statusnog polja obrađenog zahteva
       dr->status = status;
       //signaliziramo semafor, jer je operacija završena
       dr->toSignal->signal();
   }
```

```
}
```

17. (*Jun 2008)* Drajver nekog izlaznog uređaja obezbeđuje sledeću funkciju koja asinhrono pokreće operaciju izlaza datog bloka znakova zadate dužine:
`int async_write (char* buffer, int size, Semaphore* toSignalOnCompletion);`
Završetak operacije drajver signalizira semaforom na koga ukazuje treći argument (ukoliko pokazivač nije *null*).
Poznato je da sam uređaj interno omogućava paralelizam izlaznih operacija tako što može uporedo da vrši *N* zadatih operacija izlaza preko isto toliko svojih *DMA* kanala. Ukoliko se zada nova operacija a postoji slobodan kanal, drajver će tu operaciju pokrenuti na slobodnom kanalu, a funkcija `async_write` će vratiti 1. U suprotnom, ako su svi kanali zauzeti, drajver odbija zahtev (ne pokreće operaciju), a funkcija vraća -1.
Korišćenjem ove funkcije i klasičnih brojačkih semafora realizovati funkciju:
`void optimized_write (char* buffer, int size);`
koja asinhrono pokreće zadatu operaciju izlaza na ovom uređaju, ali tako da uređaj bude optimalno iskorišćen. To znači da ona treba da asinhrono pokrene traženu operaciju izlaza ako ima slobodnog kanala, tako da se svih *N* kanala maksimalno iskoriste, ali da u slučaju zauzetosti svih kanala uređaja ne odbije postavljeni zahtev, već blokira pozivajući proces sve dok se slobodan kanal ne pojavi, a onda obradi taj zahtev.

Rešenje:
Realizacija tražene funkcije:

```
void optimized_write (char* buffer, int size) {
   //broj DMA kanala
   static const int N = ...;

   //semafor na kojem će se blokirati pozivajući proces
   //ako su svi kanali zauzeti - proces će se odblokirati
   //tek pošto se oslobodi bar jedan kanal DMA kontrolera
   static Semaphore sync(N);

   //čekamo na semaforu jedan slobodan kanal
   sync.wait();

   //pozivamo f-ju za asinhrono pokretanje operacije izlaza prosleđujući
   //joj adresu gore definisanog semafora kao treći argument - kada se
   //operacija završi, drajver će završetak signalizirati tim semaforom,
   //što će omogućiti da se blokirani proces odblokira (ako postoji)
   //i zada novu izlaznu operaciju na oslobođenom kanalu DMA kontrolera
   async_write(buffer, size, &sync);
}
```

18. (*Septembar 2008)* Predložiti API za rad sa nekim izlaznim, znakovno orijentisanim nedeljivim uređajem sa kojim se izlaz obavlja mehanizmom *spooling*, kako bi se omogućilo njegovo uporedno korišćenje, čak i iz više „sesija“ unutar istog korisničkog procesa.

Rešenje:

```
// Identifier of one job – a session with the given device.
// All output within one session goes into one spooling file.
typedef int JOBID;

// Creates a new job with the device and returns the job ID (-1 on error):
JOBID createJob ();

// Outputs a string of characters within the given job;
// returns the status (0 – Ok, <0 – error);
int output (JOBID, char*);

// Closes the given job and submits it to the spooler;
// returns the status (0 – Ok, <0 – error);
int close (JOBID);
```

19. (*Oktobar 2008)* Korišćenjem koncepta niti i semafora u školskom jezgru realizovati klasu `IOStream`. Svaki objekat ove klase treba da u posebnoj niti obavi zadatu ulaznu ili izlaznu operaciju sa zadatom periferijom korišćenjem mehanizma prozivanja (*polling*). Novi objekat ove klase treba inicijalizovati sledećim parametrima: adresa upravljačkog registra, adresa statusnog registra, adresa registra za podatke periferijskog uređaja, vrsta operacije (ulaz ili izlaz), adresa memorijskog bafera sa podacima tipa unsigned int koje treba preneti, veličina bafera i pokazivač na semafor koji treba signalizirati po završetku operacije (može biti i *null*, kada ne treba signalizirati završetak). Nit koja obavlja ulaznu ili izlaznu operaciju treba da se pokrene odmah pri kreiranju ovakvog objekta, bez potrebe za posebnom akcijom iz koda koji kreira objekat. U upravljačkim registrima periferija najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke. Svi registri su veličine jedne mašinske reči (tip unsigned int) i preslikani su u memorijski adresni prostor, pa im se može pristupiti uobičajenim operacijama čitanja ili upisa preko pokazivača koji imaju vrednosti njihovih adresa.

Rešenje:

```
typedef unsigned int REG;
enum IOKind { in, out };      //tip periferije

class IOStream : public Thread {
public:
  IOStream (REG* ctrl, REG* stat, REG* data, IOKind knd,
            REG* buf, int size, Semaphore* sem = 0);

private:
  virtual void run ();

  REG *ctrl, *stat, *data, *buf;    //adrese registara i bafera
  IOKind knd;                       //zadati tip periferije
  int size;                         //veličina bloka za prenos
  Semaphore* sem;                   //semafor za signaliziranje
};

//konstruktor sa inicijalizacionom listom
IOStream::IOStream (REG* _ctrl, REG* _stat, REG* _data, IOKind _knd, REG* _buf, int
                    _size, Semaphore* _sem = 0): ctrl(_ctrl), stat(_stat), data(_data),
                                                 knd(_knd), buf(_buf), size(_size),
                                                 sem(_sem){
   //pokrećemo nit
   start();
}

void IOStream::run () {
   //startujemo periferiju upisom 1 u njen CR.Start bit
   *ctrl = 1;
   //u petlji prenosimo jedan po jedan podatak, prozivanjem
   for (int i=0; i<size; i++) {
      //čekamo da periferija postane spremna za naredni prenos
      while (*stat & 1 == 0);
      //ako je periferija ulazna, upisujemo podatak sa sa nje u bafer
      if (knd == in) buf[i] = *data;
      //ako je periferija izlazna, upisujemo podatak iz bafera u nju
      if (knd == out) *data = buf[i];
   }
   //završili smo prenos čitavog bloka, pa zaustavljamo periferiju
   *ctrl=0;
   //i signaliziramo semafor, ako nije null
```

```
    if (sem) sem->signal();
}
```

20. (*Januar 2007)*

a. Drajver (*device driver*) nekog diska kešira blokove sa diska u operativnoj memoriji. Prokomentarisati šta su prednosti, a šta nedostaci sledećih tehnika upisa podataka na disk ovog drajvera:
   - *asinhroni* upis ili *write-back*: blok iz keša se upisuje na disk odloženo, najkasnije onda kada se zamenjuje drugim blokom sa diska;
   - *sinhroni* upis ili *write-through*: svaki upis u blok u kešu se odmah prosleđuje i na disk.

b. Neki proces otvara datoteku sa najavom da će joj pristupati isključivo sekvencijalno. Koje tehnike biste predložili da bi se performanse pristupa ovoj datoteci poboljšale?

Rešenje:

a. Asinhroni (*write-back*): bolje performanse, jer se višestruki upisi u blok reflektuju samo u jedan odloženi upis na disk, ali povećanja osetljivost na otkaze, jer se prilikom otkaza sistema gube podaci koji nisu upisani na disk. Sinhroni (*write-through*) – obratno.

b. Keširanje i dohvatanje unapred (*read-ahead*).

21. (*Februar 2007)* Neki senzorski ulazni uređaj potrebno je očitavati periodično sa periodom `T`, s tim da svako očitavanje senzora vraća vrednost tipa `char`. Potrebno je obezbediti *API* funkciju

```
char get_sensor_value();
```

koju će korisničke niti pozivati iz svog konteksta, a kojom će ovaj uređaj koristiti kao ulazni, sekvencijalni, znakovno orijentisani uređaj. Ukoliko korisnička nit ovu funkciju poziva suviše često, tako da ona nema na raspolaganju dovoljno vrednosti već očitanih sa senzora, pozivajuća nit treba da se blokira.

Na raspolaganju su sledeće funkcije:

```
char read_sensor ();          // Reads one value from the sensor
void sleep (Time);            // Suspends the caller thread for the given time
```

Korišćenjem ovih funkcija, niti i ostalih potrebnih elemenata, implementirati traženu *API* funkciju. Pretpostaviti da se niti definišu kako je to dato na predavanjima.

Rešenje:

```
//standardni ograničeni bafer u koji se
//smeštaju očitane vrednosti senzora
BoundedBuffer sensorBuf;


class SensorReader : public Thread {
   protected:
      virtual void run ();
};

void SensorReader::run () {
  while (1) {
      //na svakih T milisekundi
      sleep(T);
      //očitavamo vrednost senzora
      char c = read_sensor();
      //i stavljamo je u ograničeni bafer
      sensorBuf.put(c);
  }
}

char get_sensor_value () {
   //da li prvi put pozivamo funkciju
   static int firstCall = 1;

   //ako funkciju pozivamo prvi put
   //potrebno je napraviti novu nit za
   //očitavanje senzora i startovati je
   if (firstCall) {
      firstCall = 0;
      SensorReader* sensorReader = new SensorReader();
      sensorReader->start();
   }

   //uzimamo jednu očitanu vrednost iz bafera
   //i vraćamo je kao rezultat funkcije
   //pozivajući proces će se blokirati ako je bafer prazan
   return sensorBuf.get();
}
```

22. (*Jun 2007)* Predložiti API, u smislu funkcija za sistemske pozive i njihovih deklaracija na jeziku C, za sledeće usluge nekog operativnog sistema vezanih za realno vreme:

a. očitaj i vrati tekući datum i vreme;
b. pokreni datu operaciju u zadato vreme.

Precizno objasniti način korišćenja predloženih sistemskih poziva (može i na primeru)

Rešenje:

Navedeno je samo po jedno moguće rešenje, od mnogo različitih mogućih.

a.

```
struct date {
  int day, month, year;
};

struct time {
  int hour, min, sec, millisec;
};

struct date_time {
  date d; time t;
};

date_time get_date_time ();
```

Primer korišćenja:

```
date_time now = get_date_time();
printf("Today is %d/%d/%d.\n", now.d.month, now.d.day, now.d.year);
printf("It is %d:%d:%d now.\n", now.t.hour, now.t.min, now.t.sec);
```

b. Operacija koju treba uraditi u zadato vreme može biti zadata kao funkcija (*callback* mehanizam) ili kao nit ili proces koga treba pokrenuti u zadato vreme. Od ovoga zavisi i izgled sistemskog poziva, na primer:

```
void schedule (date_time, void(*function)());
void schedule (date_time, PID to_activate_process_id);
```

Kod drugog navedenog načina potrebno je za svaku željenu operaciju kreirati proces (ili nit) koji se na svom početku suspenduje nekim posebnim sistemskim pozivom, a biva aktiviran kada dođe zadato vreme.

23. (*Jun 2007)* Drajver nekog izlaznog, sekvencijalnog, blokovski orijentisanog uređaja nudi sledeću operaciju u svom interfejsu:
`void write (char* data, int size);`
Poziv operacije `write()` je sinhroni – blokira pozivajuću nit dok se zahtev u celini ne izvrši. Korišćenjem ove operacije i već realizovanih koncepta semafora i niti, sa interfejsima kao što je definisano projektnim zadatkom za domaći rad, realizovati sloj softvera koji bi obezbedio funkciju `awrite(char*, int)` koja omogućuje asinhrono postavljanje istog takvog zahteva, bez blokiranja i čekanja pozivajuće niti da se zahtev ispuni.

Rešenje:

```
//struktura koja apstrahuje zahtev za izlaznom operacijom
struct Request {
   //blok podataka za prenos
   char* data;
   //veličina bloka
   int size;
   //konstruktor zahteva
   Request (char* d, int s) : data(d), size(s) {}
};

//klasa Adapter je singleton
class Adapter : public Thread {
   protected:
      Adapter () : Thread () { start(); }
      virtual void run ();

      static Adapter* instance;
};

//standardni ograničeni bafer u koji se
//smeštaju zahtevi za izlaznom operacijom
BoundedBuffer buf;
//inicijalizacija jedine instance klase Adapter
Adapter* Adapter::instance = new Adapter;

void Adapter::run () {
   while (1) {
      //uzimamo jedan zahtev iz reda
      Request r = buf.get();
      //pozivamo zadatu sinhronu operaciju upisa
      //nit adaptera će se blokirati pri ovom pozivu
      write(buf.data, buf.size);
   }
}

//funkcija za postavljanje asinhronog zahteva treba samo da
//ubaci zahtev u red zahteva, a nit adaptera je ona koja treba
//da procesira sam zahtev (na taj način se pozivajuća nit ne
//blokira, već se blokira nit adaptera pri pozivu operacije write)
void awrite (char* d, int s) {
   buf.put(Request(d,s));
}
```

24. (*Septembar 2007)* Na raspolaganju je sledeći API jednog blokovski orijentisanog uređaja sa direktnim pristupom:

```
void _seek (long position);               // Move the cursor to the new position
void _read (void* buffer, long size);     // Read a block to a buffer
void _write(void* buffer, long size);     // Write a block from a buffer
```

Korišćenjem ovog raspoloživog API-a realizovati sledeći API:

```
void read (void* buffer, long position, long size);
void write(void* buffer, long position, long size);
```

Rešenje:

Funkcije koje treba da budu realizovane se od zadatih funkcija razlikuju po tome što treba da čitaju/ upisuju blok podataka sa proizvoljno zadate pozicije, a ne sa trenutne pozicije.

```
void read (void* buffer, long position, long size) {
   //pomeramo kursor na zadatu poziciju
   _seek(position);
   //čitamo blok podataka zadate veličine sa zadate pozicije
   _read(buffer, size);
}

void write (void* buffer, long position, long size) {
   //pomeramo kursor na zadatu poziciju
   _seek(position);
   //upisujemo blok podataka zadate veličine od zadate pozicije
   _write(buffer, size);
}
```

25. (*Oktobar 2007)* U nekom sistemu, sistemski poziv za slanje bloka podataka nekog uređaja podržava i asinhroni i sinhroni modalitet, na šta ukazuje argument `sync` (1 - sinhrono slanje, 0 - asinhrono slanje):

```
void write (void* buffer, long size, int sync);
```

Ova operacija formira zahtev za operaciju sa uređajem sledećeg oblika:

```
struct DeviceReq {
  void* data;                 // data block
  long size;                  // size of the data block
  Semaphore* semToSignal;
};
```

u kome `semToSignal` ukazuje na semafor koji će interni kernel proces koji obrađuje zahteve signalizirati kada taj zahtev bude obrađen; ukoliko je ovaj pokazivač 0, završetak operacije se neće signalizirati. Ovako formiran zahtev smešta se u red zahteva na uređaju operacijom:

```
void putDeviceRequest (DeviceReq*);
```

Kada obradi zahtev, koji mora biti kreiran kao dinamička instanca strukture `DeviceReq`, pomenuti interni kernel proces ga uništava (briše).
Realizovati operaciju `write()`.

Rešenje:

```
void write (void* buffer, long size, int sync) {
   //semafor koji koristimo u slučaju da je poziv sinhroni
   Semaphore* sem = 0;
   //ako poziv treba da bude sinhroni, pravimo novi semafor i inicijalizujemo
   //ga nulom da bi se pozivajući proces blokirao nakon smeštanja zahteva u red
   if (sync) sem = new Semaphore(0);

   //pravimo novi zahtev za operacijom slanja bloka podataka
   //i postavljamo njegove parametre na odgovarajuće vrednosti
   DeviceReq* dr = new DeviceReq;
   dr->data = buffer;
   dr->size = size;
   dr->semToSignal = sem;

   //stavljamo napravljeni zahtev u red
   putDeviceRequest(dr);

   //ako poziv treba da bude sinhroni
   if (sem) {
      //blokiramo pozivajući proces na semaforu i čekamo
      //da se prethodno zadata operacija slanja završi
      sem->wait();
      //kada se operacija završi, i proces se odblokira
      //možemo da obrišemo semafor, jer više nije potreban
      delete sem;
   }
}
```

26. (*Januar 2006)* Posmatra se neki fizički sekvencijalni ulazni uređaj sa koga se učitava ograničeni tok znakova veličine `SIZE` znatno manje od virtuelnog adresnog prostora procesa, a sa koga se jedan znak učitava funkcijom:
`char getchar();`
Potrebno je obezbediti programski interfejs (*API*) za ovaj uređaj koji će ga tehnikom baferisanja učiniti logički (virtuelno) uređajem sa direktnim pristupom. Treba realizovati sledeće funkcije ovog interfejsa (uz odgovarajuće strukture podataka):
`void seek(int position);`
`int read(char* buf, int sz);`
Funkcija `seek()` postavlja „kurzor“ za čitanje sa ovog virtuelnog uređaja na zadatu poziciju. Funkcija `read()` prenosi niz znakova sa virtuelnog uređaja počev od pozicije kurzora u niz `buf` (alociran od strane pozivaoca); maksimalno se prenosi `sz` znakova (to je veličina niza `buf`), odnosno onoliko koliko znakova preostaje u ulaznom toku (koji je veličine `SIZE`) počev od mesta kurzora; funkcija vraća broj stvarno učitanih znakova (`sz` ili manje). Funkcija `read()` treba da učitava znakove sa fizičkog uređaja u bafer samo koliko je potrebno: ako bafer ne sadrži sve znakove koji se traže u funkciji r`ead()`, ona treba da učita najmanje što može (ne odmah ceo ulazni tok).

Rešenje:

```
//veličina ulaznog toka
const int SIZE = ...;
//pomoćni ograničeni bafer
char buffer[SIZE];
//trenutna pozicija kursora
int cursor = 0;
//broj znakova učitanih sa ulaznog toka u pomoćni bafer
int numOfLoaded = 0;

void seek (int pos) {
   //ako je nova vrednost kursora validna, ažuriramo je
   if (pos >= 0  &&  pos < SIZE) cursor = pos;
}

int read (char* b, int sz) {
   //broj znakova koje je potrebno učitati u ograničeni bafer
   //da bi se pozivaocu funkcije prenelo traženih "sz" znakova
   int toLoad = cursor + sz - numOfLoaded;

   //ako je pozivalac zatražio više znakova nego što ima do kraja
   //ulaznog toka, onda učitavamo sve preostale znakove sa ulaza
   if (toLoad > SIZE - numOfLoaded) toLoad = SIZE - numOfLoaded;

   //učitavamo potreban broj znakova
   //sa ulaznog toka u ograničeni bafer
   for (; toLoad > 0; toLoad--)
      buffer[numOfLoaded++] = getchar();

   //promenljiva za čuvanje broja prenesenih znakova
   int num = 0;
   //dok god ne pročitamo sve tražene znakove ili
   //dok ne dođemo do poslednjeg znaka u baferu
   for (; num < sz  &&  cursor + num < SIZE; num++)
      //prenosimo naredni znak u zadati niz (bafer)
      b[num] = buffer[cursor + num];

   //vraćamo broj prenesenih znakova
   return num;
}
```

27. (*April 2006)*

a. Ako sistem koristi *spooling*, kakva je po svojoj prirodi operacija slanja niza znakova na štampač koju poziva korisnički proces, posmatrana „sa kraja na kraj“, tj. iz perspektive korisničkog procesa na operaciju koju zadaje i uređaj koji treba da je izvrši - asinhrona ili sinhrona (blokirajuća)?
b. Povećanje bafera i blokova prenosa prilikom upisa na disk ima mnoge pozitivne efekte, poput ređih i efikasnijih operacija sa uređajima. Međutim, kakav je negativan uticaj povećanja bafera, u pogledu posledica u slučaju otkaza?

Rešenje:

a. Uvek asinhrona, jer proces zadaje operaciju i nastavlja svoje izvršavanje, ne čekajući da se uređaj završi zadatu operaciju. Operacija neće ni početi pre nego što dati proces „zatvori" svoj posao sa uređajem (a posao može da uključuje više ovakvih operacija) i kada taj posao dođe na red za obradu od strane *spooler*-a. Zato pozivajući proces nikada i ne treba da čeka na završetak ove operacije.

b. Veći baferi znače i ređe slanje podataka na disk, što znači vežu štetu u slučaju otkaza: više podataka duže vreme nije snimljeno na disk i ostaje trajno izgubljeno u slučaju otkaza računara.

28. (*Jun 2006)* Posmatra se neki sekvencijalni, blokovski orijentisan izlazni uređaj na koji se blok podataka fiksne dužine `BLKSIZE` bajtova sa adrese zadate argumentom upisuje operacijom (pretpostaviti da je *C* tip `char` veličine jednog bajta):
`void writeblk(char*);`

Potrebno je obezbediti programski interfejs (*API*) za ovaj uređaj koji će ga tehnikom baferisanja učiniti logički (virtuelno) znakovno orijentisanim uređajem sa sekvencijalnim upisom. Treba realizovati funkciju ovog interfejsa (uz odgovarajuće strukture podataka):
`void putchar(char);`

Bafer treba isprazniti upisom na uređaj čim se sakupi dovoljno bajtova za ceo blok, sinhrono, u istom kontekstu pozvane funkcije putchar(). Ova funkcija ne mora da bude *thread-safe*, tj. ne treba obezbediti međusobno isključenje za pristup više niti istog procesa.

Rešenje:
Realizacija tražene funkcije:

```
void putchar(char c) {
   //uređaj je blokovski, pa mu podatke moramo slati u blokovima, a ne pojedinačno
   //pojedinačne podatke (znakove), stavljamo u ograničeni bafer, a kada se bafer
   //napuni, možemo da pošaljemo sadržaj čitavog bafera kao blok podataka uređaju
   static char buffer[BLKSIZE];
   //trenutni broj podataka u baferu
   static int size = 0;

   //ubacujemo zadati podatak (znak) u bafer i uvećavamo
   //indeks prve slobodne pozicije u ograničenom baferu
   buffer[size++] = c;

   //ako je bafer pun, treba njegov sadržaj poslati uređaju
   if (size == BLKSIZE) {
      //šaljemo sadržaj bafera kao blok podataka izlaznom uređaju
      writeblk(buffer);
      //praznimo ograničeni bafer
      size = 0;
   }
}
```

29. (*Jun 2006)*

a. Da li je potrebno baferisanje da bi se realizovao API koji:
    i. za znakovni uređaj obezbeđuje blokovski pristup;
    ii. za uređaj sa direktnim pristupom obezbeđuje sekvencijalni pristup (uz isti tip podataka koji se prenose)?

   Precizno obrazložiti odgovore.
b. Navesti i precizno opisati mehanizam kojim se baferi i keševi za I/O operacije kernela u potpunosti štite od slučajnog ili malicioznog upisa od strane korisničkih procesa.

Rešenje:

a.

i. Da, jer se mora sakupiti ceo blok znakova u bafer koji se onda prenosi na uređaj ili sa uređaja.
ii. Ne, jer se uređaju sa direktnim pristupom uvek može pristupati i sekvencijano, pa nije potrebno čuvati podatke u baferu – oni se odmah mogu proslediti.

b. Mehanizam virtuelne memorije: okviri fizičke memorije u kojima se nalaze baferi i keševi I/O podsistema se ne preslikavaju u adresne prostore korisničkih procesa (nema ih u njihovim tabelama preslikavanja), pa im oni jednostavno nikako ne mogu pristupiti („ne vide" ih).

30. (*Septembar 2006)*

a. Predložiti najjednostavniji interfejs generičkog drajvera za sve blokovski orijentisane uređaje sa direktnim pristupom, pogodne da se na njima organizuje fajl sistem. (Interfejs treba da obezbedi učitavanje i upisivanje na nivou bloka, ne na nivou fajla. Koncept fajla i operacije sa njim treba da obezbedi fajl sistem koji koristi usluge ovog drajvera. Interfejs treba da bude dovoljan da se pomoću njega izgradi fajl sistem.)
b. Ako kontroler uređaja poseduje sopstveni, hardverski keš i bafer koji obezbeđuju veoma brz odziv i kod operacija čitanja i kod operacija upisa bloka, samerljiv sa odzivom operativne memorije, kojom tehnikom programiranog ulaza-izlaza treba realizovati drajver ovakvog uređaja: prozivanjem (*polling*) ili korišćenjem prekida? Kratko obrazložiti odgovor.

Rešenje:

a.

```
typedef unsigned long BlockSize;
typedef unsigned long BlockNo;

//tipovi statusa izvršene operacije
enum IOStatus {Err, OK};

class DeviceDriver {
   public:
      //f-ja koja vraća veličinu bloka sa kojim radi uređaj
      virtual BlockSize getBlockSize  ();
      //broj blokova podataka na uređaju
      virtual BlockNo   getNumOfBlocks();

      //f-ja za čitanje određenog bloka sa uređaja i upis tog bloka u memoriju
      virtual IOStatus  read (BlockNo, void* buffer);
      //f-ja za upis zadatog bloka iz memorije u uređaj, na odgovarajuću poziciju
      virtual IOStatus  write(BlockNo, void* buffer);

};
```

b. Prozivanjem (*polling*), jer je u ovom slučaju efikasnije od obrade prekida (koja uključuje dodatne režije).

31. (*Oktobar 2006)* Na raspolaganju je drajver (*device driver*) za jedan izlazni, znakovno orijentisani uređaj, pri čemu korisnički proces može zadati prenos čitavog niza znakova koje će drajver upisivati na uređaj znak po znak. Korisnički proces koji zadaje izlaznu operaciju svoj zahtev predstavlja sledećom strukturom:

```
struct DeviceRequest {
  char* block;          // pokazivač na niz znakova koje treba preneti na uređaj
  unsigned long size;   // veličina niza znakova
  Semaphore* toSignal;  // semafor koji treba signalizirati po završetku op.
};
```

Korisnički proces svoj zahtev zapisan u ovakvoj strukturi upisuje u ograničeni bafer koji predstavlja red postavljenih zahteva. Kada završi celokupnu operaciju, drajver signalizira ovaj semafor; ako je u zahtevu `toSignal` postavljen na *NULL*, drajver neće signalizirati završetak operacije. Semafor je standardni, brojački semafor sa uobičajenim operacijama `signal` i `wait`. Ograničenom baferu pristupa se sledećim operacijama koje su blokirajuće ukoliko je bafer pun, odnosno prazan:

```
void putDeviceRequest (DeviceRequest*);   // stavlja jedan zahtev u red
DeviceRequest* getDeviceRequest();        // uzima jedan zahtev iz reda
```

Potrebno je realizovati „omotač“ (*wrapper*) za ovu uslugu, odnosno bibliotečne funkcije

```
void syncWrite (char*);                   // synchronous write
void asyncWrite (char*);                  // asynchronous write
```

Obe funkcije primaju kao argument pokazivač na niz znakova završen znakom `'\0'`. Funkcija `syncWrite()` treba da blokira pozivajući proces sve do završetka cele operacije upisa niza znakova, dok funkcija `asyncWrite()` treba samo da zada operaciju i vrati kontrolu pozivajućem procesu, bez blokiranja do završetka izlazne operacije.

Rešenje:

```
//u zaglavlju string.h se nalazi f-ja strlen koja nam je potrebna da
//bismo odredili broj znakova u stringu koji šaljemo izlaznom uređaju
#include <string.h>


void syncWrite (char* str) {
   //dohvatamo dužinu stringa koji upisujemo
   unsigned long int len = strlen(str);
   //potreban nam je semafor koji će drajver da signalizira nakon završetka
   //upisa - inicijalno je postavljen na nula da bi se pozivajući proces
   //blokirao na semaforu odmah nakon zadavanja operacije (sinhroni poziv)
   Semaphore* sync = new Semaphore(0);

   //pravimo novi zahtev za zslaznom operacijom sa odgovarajućim parametrima
   DeviceRequest* dr = new DeviceRequest;
   //adresa početka bloka
   dr->block = str;
   //broj podataka (znakova) u bloku
   dr->size = len;
   //semafor koji drajver treba da signalizira
   dr->toSignal = sync;

   //postavljamo zahtev u red zahteva (ograničeni bafer)
   putDeviceRequest(dr);
   //blokiramo pozivajući proces i čekamo da se zadata operacija završi
   sync->wait();
   //zahtev je obrađen, možemo da obrišemo semafor
   delete sync;
```

```
}
```

```
void asyncWrite (char* str) {
   //dohvatamo dužinu stringa koji upisujemo
   unsigned long int len = strlen(str);

   //pravimo novi zahtev za zslaznom operacijom sa odgovarajućim parametrima
   DeviceRequest* dr = new DeviceRequest;
   //adresa početka bloka
   dr->block = str;
   //broj podataka (znakova) u bloku
   dr->size = len;
   //poziv treba da bude asinhroni, tako da nam semafor, na kome se pozivajući
   //proces blokira čekajući na završetak zadate operacije, nije potreban (NULL)
   dr->toSignal = NULL;

   //postavljamo zahtev u red zahteva (ograničeni bafer)
   putDeviceRequest(dr);
}
```

32. (*Jun 2005)* Potrebno je realizovati drajver (*device driver*) za jedan izlazni, znakovno orijentisani uređaj, pri čemu korisnički proces može zadati prenos čitavog niza znakova koje će *DMA* upisivati na uređaj znak po znak. Korisnički proces koji zadaje izlaznu operaciju svoj zahtev predstavlja sledećom strukturom:

```
struct DeviceRequest {
  char* block;          // Pokazivač na niz znakova koje treba preneti na uređaj
  unsigned long size;   // Veličina niza znakova
  Semaphore* toSignal;  // Semafor koji treba signalizirati po završetku operacije
};
```

Korisnički proces svoj zahtev zapisan u ovakvoj strukturi upisuje u ograničeni bafer koji predstavlja red postavljenih zahteva, a potom se blokira na semaforu na koji ukazuje pokazivač `toSignal`. Semafor je standardni, brojački semafor sa uobičajenim operacijama `signal` i `wait`. Ograničenom baferu pristupa se sledećim operacijama koje su blokirajuće ukoliko je bafer pun, odnosno prazan:

```
void putDeviceRequest (DeviceRequest*);   // Stavlja jedan zahtev u red
DeviceRequest* getDeviceRequest();        // Uzima jedan zahtev iz reda
```

Sama izlazna operacija obavlja se preko *DMA* uređaja kome se prenos niza znakova `block` veličine `size` iz memorije na uređaj zadaje i pokreće sledećom neblokirajućom operacijom:

```
void DMARequest (char* block, unsigned long size);
```

Kada završi prenos bloka podataka, *DMA* generiše prekid kome pripada ulaz 10h u vektor tabeli.
Korišćenjem navedenih operacija (za koje se pretpostavlja da su implementirane) i već realizovanih koncepta događaja i niti, sa interfejsima kao što je definisano projektnim zadatkom za domaći rad, napisati kompletan kod ovog drajvera uređaja.

Rešenje:

```
//zaglavlja sa potrebnim definicijama klasa iz skolskog jezgra
#include "event.h"
#include "thread.h"

//deklaracije datih operacija
extern DeviceRequest* getDeviceRequest();
extern void DMARequest (char* block, unsigned long size);


//deklaracija prekidne rutine DMA kontrolera
void interrupt deviceInt (...);

//konstruktor događaja za završetak prenosa
Event deviceEvent(0x10, deviceInt);

//u prekidnoj rutini je potrebno samo signalizirati događaj
void interrupt deviceInt (...) {
  deviceEvent.signal();
}


//klasa DeviceDriver je singleton
class DeviceDriver : public Thread {
   protected:
      //drajver startujemo odmah nakon kreiranja
      DeviceDriver () : Thread() { start(); }
      virtual void run ();


   private:
```

```
      static DeviceDriver instance;
};

//jedina instanca klase DeviceDriver
DeviceDriver DeviceDriver::instance;


void DeviceDriver::run () {
   while (1) {
      //uzimamo jedan zahtev iz reda zahteva
      DeviceRequest* dr = getDeviceRequest();

      //zadajemo izlaznu operaciju DMA kontroleru
      //koristeći parametre prethodno uzetog zahteva
      DMARequest(dr->block,dr->size);

      //čekamo da DMA kontroler javi da je prenos završen
      deviceEvent.wait();

      //signaliziramo semafor koji je potrebno signalizirati
      //pri završetku zadate izlazne operacije
      dr->toSignal->signal();
   }
}
```

33. (*Septembar 2005)* Analogno-digitalna (*A/D*) konverzija je postupak konverzije odbirka nekog analognog signala na ulazu u računar u njegovu digitalnu aproksimaciju. *A/D* konvertor je ulazni uređaj koji vrši ovakvu konverziju. Da bi *A/D* konvertor izvršio konverziju, potrebno mu je najpre zadati (pokrenuti) operaciju konverzije upisom 1 u bit *START* njegovog upravljačkog registra. Da bi izvršio konverziju, *A/D* konvertoru je potrebno izvesno vreme. Po završetku konverzije, *A/D* konvertor upisuje digitalni rezultat u svoj registar za podatke i signalizira završetak operacije postavljanjem bita *EOC* (*end of conversion*) u svom statusnom registru. Ukoliko je ovaj indikator povezan na odgovarajući način, završetak konverzije generiše prekid procesoru. Poznato je da *A/D* konverzija svakako traje najviše 50ms, ali može da traje i znatno kraće. Na raspolaganju su sledeće funkcije iz *C API* nekog operativnog sistema:

```
void startAD()          //pokreće A/D konverziju prostim upisom 1 u bit START A/D konvertora
int getEOC()            //čita statusni registar A/D konvertora i vraća vrednost bita EOC (0 ili
1)
int getData()           //čita registar podataka A/D konvertora i vraća pročitanu vrednost
void waitAD()           //suspenduje pozivajući proces sve dok se ne dogodi prekid zbog EOC
void delay(unsigned ms) //suspenduje pozivajući proces na vreme zadato argumentom u ms
```

Korišćenjem ovih funkcija, realizovati sledeće funkcije:

- `int adConvert()`: sinhrona, blokirajuća operacija koja treba da izvrši *A/D* konverziju i vrati dobijenu digitalnu vrednost, pod pretpostavkom da *EOC* jeste vezan na ulaz zahteva za prekid.
- `int adConvert()`: sinhrona, blokirajuća operacija koja treba da izvrši *A/D* konverziju i vrati dobijenu digitalnu vrednost, pod pretpostavkom da *EOC* nije vezan na ulaz zahteva za prekid.
- `int getADConv()`: sinhrona, neblokirajuća operacija koja treba da zada *A/D* konverziju, uposleno čeka neko kratko vreme (po Vašem izboru) i vrati dobijenu digitalnu vrednost (koja je uvek nenegativna) ako je konverzija završena, odnosno -1 ako konverzija nije završena u tom vremenu.

Rešenje:

Realizacija traženih funkcija:

```
int adConvert () {
   //pokrećemo konverziju
   startAD();
   //blokiramo pozivajući proces dok se konverzija ne završi
   waitAD();
   //vraćamo rezultat konverzije
   return getData();
}

int adConvert () {
   //pokrećemo konverziju
   startAD();
   //blokiramo pozivajući proces za maksimalno vreme trajanja konverzije
   delay(50);
   //vraćamo rezultat konverzije
   return getData();
}

int getADConv () {
   //pokrećemo konverziju
   startAD();
   //uposleno čekamo određeno vreme
   for (int i=0; i<...; i++);
   //ako je konverzija završena, vraćamo rezultat konverzije
   //u suprotnom, kao rezultat vraćamo vrednost -1
   if (getEOC()) return getData(); else return -1;
}
```

34. (*Oktobar 2005)*

a. Kojom tehnikom se fizički nedeljivi izlazni uređaj može učiniti logički (virtuelno) deljivim između procesa koji ga uporedo koriste?
b. Kojom tehnikom se fizički sekvencijalni ulazni uređaj sa koga se učitava ograničeni niz podataka veličine znatno manje od virtuelnog adresnog prostora procesa može učiniti logički (virtuelno) uređajem sa direktnim pristupom?

Rešenje:

a. *Spooling*.

b. Baferisanje.

Седма група задатака: фајл систем

1. (*Januar 2011*) Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa kombinovanim indeksom u jednom i dva nivoa. Polje `index1` u strukturi `FCB` sadrži broj fizičkog bloka u kome je indeks prvog nivoa (čiji ulazi sadrže brojeve blokova sa sadržajem), dok polje `index2` sadrži broj fizičkog bloka u kome je indeks drugog nivoa (čiji ulazi sadrže brojeve blokova sa indeksima u kojima su brojevi blokova sa sadržajem). Broj fizičkog bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog).

a. Realizovati funkciju:
   `int f_blkalloc(FCB* file, BLKNO logical, BLKNO physical);`
   koja ažurira indekse datog fajla tako da proglašava dati logički blok alociranim u datom fizičkom bloku. U slučaju uspeha funkcija vraća 0, u slučaju greške -1. Pretpostaviti da je indeks prvog nivoa uvek inicijalno alociran, dok se
   indeks drugog nivoa alocira samo po potrebi. Na raspolaganju je funkcija:
   `void* f_getblk(BLKNO physical, int toLoad=1);`
   koja vraća pokazivač na keširani fizički blok sa datim brojem i učitava ga u keš ako je `toLoad=1` i ako blok već nije
   u kešu, kao i funkcija:
   `BLKNO f_blkalloc();`
   koja alocira jedan slobodan fizički blok i vraća njegov broj (0 u slučaju neuspeha).
b. Kolika je maksimalna moguća veličina fajla, ako je `INDEXSIZE=1024`, a tip `BLKNO` je 64-bitni neoznačeni ceo broj?

Rešenje:

a.

```
//funkcija za alokaciju novog fizičkog bloka
//za indekse prvog ili drugog nivoa
//uz inicijalizaciju svih ulaza nulama
BLKNO f_newblk () {
   //pokušavamo da alociramo jedan fizički blok
   //ako ne uspemo, vraćamo indikator greške
   BLKNO blk = f_blkalloc();
   if (blk == 0) return 0;

   //pokušavamo da dohvatimo adresu alociranog bloka
   //ako ne uspemo, vraćamo indikator greške
   BLKNO* p = (BLKNO*)f_getblk(blk, 0);
   if (p == 0) return 0;

   //inicijalizcaija svih ulaza nulama
   for (int i=0; i<INDEXSIZE; i++) p[i] = 0;

   //kao rezultat, funkciaj vraća broj alociranog bloka
   return blk;
}

int f_blkalloc(FCB* file, BLKNO lb, BLKNO pb) {
   //ako je pokazivač na FCB strukturu jednak null ili ako je
   //broj fizičkog bloka nedozvoljen za alokaciju (blok broj 0)
   //funkcija vraća indikator greške (rezultat -1)
   if (file==0 || pb==0) return -1; // Error

   //ako logički blok sa brojem "lb" pripada indeksu prvog nivoa...
```

```
   if (lb < INDEXSIZE) {
      //dohvatamo adresu fizičkog bloka sa indeksom prvog nivoa
      BLKNO* pIndex = (BLKNO*)f_getblk(file->index1);
      //ako je vraćeni pokazivač jednak null, greška
      if (pIndex == 0) return -1;
      //u suprotnom, ažuriramo vrednost traženog logičkog bloka
      pIndex[lb]=pb;
   }
   //ako logički blok sa brojem "lb" ne pripada indeksu prvog nivoa...
   else {
      //broj indeksa drugog nivoa i broj ulaza u tom indeksu
      //koji odgovara logičkom bloku sa brojem "lb"
      int i2 = (lb - INDEXSIZE) / INDEXSIZE;
      int i1 = (lb - INDEXSIZE) % INDEXSIZE;
      //ako je broj ulaza u indeks drugog nivoa veći od
      //najvećeg broja ulaza u indeksu drugog nivoa, greška
      if (i2 >= INDEXSIZE) return -1;

      //ako indeks drugog nivoa nije alociran
      if (file->index2 == 0) {
         //pokušavamo alokaciju...
         file->index2 = f_newblk();
         //..ako alokacija ne uspe, greška
         if (file->index2 == 0) return -1;
      }

      //dohvatamo adresu fizičkog bloka sa indeksom drugog nivoa
      BLKNO* pIndex2 = (BLKNO*)f_getblk(file->index2);
      //ako je vraćeni pokazivač jednak null, greška
      if (pIndex2 == 0) return -1;

      //dohvatamo broj fizičkog bloka odgovarajućeg indeksa
      //prvog nivoa (onaj kojem pripada "lb" logički blok)
      BLKNO index1 = pIndex2[i2];
      //ako taj indeks prvog nivoa nije alociran, pokušavamo
      //alokaciju, ako ne uspemo, prijavljujemo grešku
      if (index1 == 0) {
            pIndex2[i2] = f_newblk();
            if (pIndex2[i2] == 0) return -1;
      }
      //dohvatamo adresu fizičkog bloka sa indeksom prvog nivoa
      BLKNO* pIndex1 = (BLKNO*)f_getblk(pIndex2[i2]);
      //ažuriramo vrednost traženog logičkog bloka
      pIndex1[i1] = pb;
   }
   //ažuriranje je uspelo, vraćamo nulu
   return 0;
}
```

b.

Broj ulaza indeksa prvog nivoa: `INDEXSIZE`

Broj ulaza indeksa drugog nivoa je `INDEXSIZE`, a kako svaki ulaz odgovara jednom dodatnom indeksu prvog nivoa, onda je maksimalan broj ulaza u sve indekse prvog nivoa: `INDEXSIZE + INDEXSIZE * INDEXSIZE`

Veličina jednog bloka je:

```
BLOCK_SIZE = INDEXSIZE * sizeof(BLKNO) = 1K * 8B = 8KB
```

Svaki ulaz u indeks tabeli prvog nivoa pokazuje na jedan fizički blok, pa je maksimalna veličina fajla:

```
MAX_FILE_SIZE = (INDEXSIZE + INDEXSIZE * INDEXSIZE) * BLOCK_SIZE = 8MB + 8GB
```

2. (*Februar 2011*) Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa neograničenim brojem blokova za indeks fajla, ulančanim u jednostruku listu. Polje `index` u strukturi `FCB` sadrži broj prvog indeksnog bloka u listi. Broj narednog indeksnog bloka u listi nalazi se na samom početku svakog indeksnog bloka. Broj bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), uključujući i prvi ulaz koji ukazuje na sledeći indeksni blok, a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog).

a. Realizovati funkciju koja se koristi kod direktnog pristupa fajlu:
   `void* f_getblk(FCB* file, BLKNO logical);`
   koja vraća pokazivač na dohvaćeni i keširani blok sa podacima fajla sa datim logičkim brojem. Na raspolaganju je funkcija:
   `void* f_getblk(BLKNO physical);`
   koja vraća pokazivač na keširani blok sa datim fizičkim brojem i učitava ga u keš ako je potrebno.
b. Koliko je blokova raspoloživo za podatke nekog fajla za koga je alocirano ukupno 3K blokova na disku (i za indeksne blokove i za blokove za podatke zajedno), ako je `INDEXSIZE=1024`?

Rešenje:

a.

```
//format jednog indeksnog bloka
struct Block {
   BLKNO next;                //naredni indeksni blok
   BLKNO index[INDEXSIZE-1];  //ulazi indeksnog bloka
};

void* f_getblk(FCB* file, BLKNO lb) {
   //ako je pokazivač na FCB jednak null, greška
   if (file == 0) return 0;

   //dohvatamo adresu prvog indeksnog bloka u listi
   Block* blk = (Block*)f_getblk(file->index);
   //ako nismo uspeli da pristupimo indeksnom bloku, greška
   if (blk == 0) return 0;

   //sve dok ne dođemo do indeksnog bloka kome pripada zadati logički blok broj "lb"
   while (lb >= INDEXSIZE-1) {
      lb -= INDEXSIZE - 1;
      //dohvatamo naredni indeksni blok iz liste
      blk = (Block*)f_getblk(blk->next);
      //ako nismo uspeli da pristupimo indeksnom bloku, greška
      //(moguće je da je logički blok broj "lb" izvan opsega)
      if (blk == 0) return 0;
   }
   //vraćamo adresu dohvaćenog fizičkog bloka sa datim logičkim brojem
   return f_getblk(blk->index[lb]);
}
```

b.

Ako je broj indeksnih blokova jednak ***IB***, onda je na disku odvojeno ukupno:

$N = IB * 1023 + \lceil IB * 1023 / 1023 \rceil = IB * 1024$

gde je IB * 1023 broj blokova odvojenih za podatke. Kako je N = 3K = 3 * 1024, sledi da je IB = 3, pa je ukupan broj blokova odvojen za podatke jednak:

DB = IB * 1023 = 3 * 1023 = 3069

3. (*Jun 2011*) U nekom operativnom sistemu struktura `PCB` sadrži polje `open_files` koje je pokazivač na tabelu otovorenih fajlova tog procesa. U toj tabeli (zapravo nizu), svaki ulaz je struktura koja predstavlja deskriptor jednog otvorenog fajla. U toj strukturi postoji celobrojno polje `access` čija vrednost 1 označava da je dati proces otvorio dati fajl sa pravom na upis, a vrednost 0 označava da je dati proces otvorio dati fajl samo sa pravom na čitanje. Promenljiva tipa `FHANDLE` predstavlja indeks u tabeli otvorenih fajlova datog procesa u čijem je kontekstu otvaren fajl.
Implementirati operaciju `check_access()` čiji je potpis dat, a koju poziva fajl podsistem u svakom sistemskom pozivu za pristup sadržaju fajla radi provere prava datog procesa na izvršavanje date operacije nad sadržajem fajla. Argument `p` je pokazivač na `PCB` procesa koji je izdao sistemski poziv, argument `f` je identifikator fajla, a argument `write` ima vrednost 0 ako sistemski poziv zahteva samo čitanje, odnosno vrednost 1 ako sistemski poziv zahteva upis u dati fajl.

```
int check_access (PCB* p, FHANDLE f, int write);
```

Rešenje:

```
int check_access (PCB* p, FHANDLE f, int write) {
   //pristupamo ulazu FHANDLE u tabeli otvorenih fajlova
   //pozivajućeg procesa i proveravamo da li je tražena
   //operacija dozvoljena

   //ako je: write == 0, onda je operacija dozvoljena uvek (čitanje)
   //ako je: write == 1, onda moramo da proverimo da li je access == 1

   //ako je operacija dozvoljena, funkcija vraća 1, u suprotnom 0
   return p->open_files[FHANDLE].access || !write;
}
```

4. (*Jun 2011*) Neki fajl sistem koristi *FAT*, uz dodatni mehanizam detekcije i oporavka od korupcije ulančanih lista na sledeći način. Kada se *FAT* kešira u memoriji, vidi se kao niz struktura tipa `FATEntry`. Ova struktura ima dva celobrojna polja. Polje `next` je broj ulaza u *FAT* u kome se nalazi sledeći element u ulančanoj listi; vrednost 0 označava kraj liste. Polje `fid` ove strukture sadrži identifikator fajla kome pripada taj element liste. U strukturi `FCB` celobrojno polje `id` predstavlja identifikator datog fajla, a polje `head` sadrži redni broj ulaza u *FAT* koji je prvi element u ulančanoj listi datog fajla.
Implementirati funkciju čiji je potpis dat dole. Ona treba da proveri konzistentnost ulančane liste datog fajla, proverom da li svi elementi liste pripadaju baš tom fajlu. Ukoliko je sve u redu, ova funkcija treba da vrati 1. Ukoliko naiđe na element u listi koji je pogrešno ulančan, odnosno ne pripada tom fajlu (tako što polje `fid` ne odgovara identifikatoru tog fajla), taj pogrešno ulančani ostatak liste treba da „odseče“ postavljanjem terminatora liste (vrednost 0 u polje `next`) u poslednji ispravan element liste (ili glavu liste, ako je prvi element pogrešan) i da vrati 0.

```
FATEntry FAT[...];

int check_consistency (FCB* file);
```

Rešenje:

```
int check_consistency (FCB* file) {
   //ako je pokazivač na FCB jednak null, greška
   if (file == 0) return 0;

   //dohvatamo identifikator fajla iz FCB-a
   FID fid = file->id;

   //postavljamo pokazivač "cur" da pokazuje na ulaz u FAT prvog
   //elementa ulančane liste datog fajla, a pokazivač "prev" na null
   int cur = file->head, prev = 0;

   //prolazimo kroz ulančanu listu elemenata u FAT-u
   //sve dok ne dođemo do kraja ulančane liste datog fajla
   for (; cur != 0; prev = cur, cur = FAT[cur].next)
      //ili dok ne detektujemo grešku (pogrešan ID)
      if (FAT[cur].fid != fid) {
         //odsecamo neispravni deo liste
         if (prev) FAT[prev].next = 0;
         else file->head = 0;

      //vraćamo indikator greške
      return 0;
   }

   //cela lista je u redu, vraćamo 1
   return 1;
}
```

5. (*Septembar 2011*) Dat je deo *API* za neki fajl sistem:

```
// File handle
typedef int FHANDLE;

FHANDLE fopen (char* fname, int accessFlags);
int fclose (FHANDLE);
int fread (FHANDLE, unsigned int offset, unsigned int size, void* buffer);
int fwrite (FHANDLE, unsigned int offset, unsigned int size, void* buffer);

// Returns the file size
unsigned int fsize (FHANDLE);
```

Sve funkcije osim `fsize` vraćaju pozitivnu vrednost u slučaju uspeha, a negativnu vrednost sa kodom greške u slučaju neuspeha. Koncept kurzora nije ugrađen u ovaj fajl sistem, već funkcije za čitanje i upis u fajl zadaju pomeraj (*`offset`*) od koga se čita/upisuje.

Korišćenjem ovog API realizovati klasu koja apstrahuje fajl sa ugrađenim kurzorom i sledećim interfejsom:

```
class File {
  public:
    File (char* fname, int accessFlags);
    ~File ();
    int read (void* buffer, unsigned int size);
    int write (void* buffer, unsigned int size);

    // Move the cursor to the given offset
    int seek (unsigned int offset);
};
```

Rešenje:

```
class File {
public:
  File (char* fname, int accessFlags);
 ~File ();
  int read (void* buffer, int size);
  int write (void* buffer, int size);
  int seek (unsigned int offset); // Move the cursor to the given offset
private:
  FHANDLE fh;
  unsigned int cursor;
};
File::File (char* fname, int af) : cursor(0) {
  fh=fopen(fname,af);
}

File::~File () { fclose(fh); }

int File::read (void* buffer, unsigned int size) {
  int ret = fread(fh,cursor,size,buffer);
  seek(cursor+size);
  return ret;
}

int File::write (void* buffer, unsigned int size) {
  int ret = fwrite(fh,cursor,size,buffer);
  seek(cursor+size);
  return ret;
}
```

```
int File::seek (unsigned int offset) {
  unsigned int size = fsize(fh);
  if (offset>=size) cursor=size;
  else cursor=offset;
  return 1;
}
```

6. (*Septembar 2011*) Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa indeksima u dva nivoa. Polje `index` u strukturi *FCB* sadrži broj indeksnog bloka prvog nivoa. Broj bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog).
Realizovati funkciju koja se koristi kod direktnog pristupa fajlu:
`void* f_getblk(FCB* file, BLKNO logical);`
koja vraća pokazivač na dohvaćeni i keširani blok sa podacima fajla sa datim logičkim brojem. Na raspolaganju je funkcija:
`void* f_getblk(BLKNO physical);`
koja vraća pokazivač na keširani blok sa datim fizičkim brojem i učitava ga u keš ako je potrebno.

Rešenje:

```
void* f_getblk(FCB* file, BLKNO lb) {
   //ako je pokazaivač na FCB jednak null, greška
   if (file == 0) return 0;

   //formiramo ulaz u indeks prvog nivoa na kojem se nalazi broj
   //bloka sa indeksom drugog nivoa kome pripada logički blok "lb"
   unsigned long int entry = lb / INDEXSIZE;
   //ako je broj ulaza veći od maksimalnog, greška (preveliki "lb")
   if (entry >= INDEXSIZE) return 0;

   //dohvatamo adresu indeksa prvog nivoa
   BLKNO* index = (BLKNO*)f_getblk(file->index);
   //ako ne uspemo da pristupimo indeksu prvog nivoa, greška
   if (index==0) return 0;

   //dohvatamo broj bloka odovarajućeg indeksa drugog nivoa
   BLKNO index2blk = index[entry];
   //ako se taj indeks drugog nivoa ne koristi, greška (prekoračenje)
   if (index2blk == 0) return 0;

   //dohvatamo adresu indeksa drugog nivoa
   index = (BLKNO*)f_getblk(index2blk);
   //ako ne uspemo da pristupimo indeksu drugog nivoa, greška
   if (index == 0) return 0;

   //broj ulaza u indeksu drugog nivoa kojem odgovara dati logički blok
   entry = lb % INDEXSIZE;
   //vraćamo adresa fizičkog bloka sa datim logičkim brojem
   return f_getblk(index[entry]);
}
```

7. (*Januar 2010*)

a. Neki fajl sistem podržava implicitno zaključavanje fajla prilikom njegovog otvaranja. Postoje dve vrste ključa: deljeni (*shared*, S), koji se traži prilikom otvaranja fajla samo za čitanje (proces koji je tako otvorio fajl ima pravo samo da čita iz fajla) i ekskluzivni (*exclusive*, X), koji se traži prilikom otvaranja fajla i za upis (proces koji je otvorio fajl ima pravo upisa). Popuniti sledeću tabelu upisivanjem oznaka onih procesa čiji će zahtev za otvaranjem istog fajla biti ispunjen, za svaki od dva data nezavisna slučaja. Procesi postavljaju zahteve redom navedenim u drugoj koloni, pri čemu oznaka npr. A-Rd označava da proces A postavlja zahtev za otvaranjem fajla za čitanje, a B-Wr da proces B postavlja zahtev za otvaranjem fajla za upis.

Slučaj	Zahtevi za otvaranje fajla	Procesi koji su uspeli da otvore fajl
1	A-Rd, B-Rd, C-Wr, D-Rd, E-Wr	
2	A-Wr, B-Wr, C-Rd, D-Rd, E-Rd	

b. Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove, sa ulančavanjem blokova sa indeksima u listu. Na prvi indeksni blok u listi ukazuje polje `indexBlockNo` u `FCB` (*file control block*). Svaki indeksni blok sadrži `EntriesInBlock` ulaza tipa `PBlockNo` koji sadrže brojeve fizičkih blokova sa podacima, dok naredni (poslednji) ulaz tipa `PBlockNo` u indeksnom bloku sadrži broj sledećeg indeksnog bloka u listi (0 za kraj liste). `FCB` je deklarisan na sledeći način:

```
typedef unsigned long int PBlockNo; // block number
struct FCB {
  ...
  unsigned long sizeInBlocks;       // Current file size in number of blocks
  PBlockNo indexBlockNo;            // Pointer to the first index block
  ...
};
```

Projektuje se sloj fajl sistema koji se bavi organizacijom fajla (*file organization module*). Ovaj modul održava memorijski keš na nivou fizičkih blokova diska, pa se učitani blokovi sa diska uvek nalaze u prostoru odvojenom za keš. Postojeća funkcija:

```
Block* getDiskBlock (PBlockNo physicalFileBlockNo);
```

po potrebi učitava i vraća pokazivač na učitani blok za zadatim fizičkim rednim brojem na disku. Potrebno je realizovati funkciju:

```
Block* getDataBlock(FCB* file, unsigned long logicalBlockNo);
```

koja treba da u memoriju učita blok sa logičkim rednim brojem datim drugim argumentom fajla čiji je `FCB` zadat prvim argumentom i da vrati pokazivač na taj blok.

Rešenje:

a. Ako prvi proces otvara fajl za čitanje (deljeni ključ), naredni procesi koji otvaraju isti fajl ne mogu da traže ekskluzivni ključ (ne smeju da otvaraju fajl za upis), već jedino deljeni (samo čitanje), inače će njihov zahtev biti odbijen.
Ako prvi proces otvara fajl za pisanje (ekskluzivni ključ), naredni procesi neće moći da otvore fajl bez obzira na tip pristupa koji su zatražili (deljeni ili ekskluzivni).
Na osnovu prethodnog, možemo da popunimo traženu tabelu:

Slučaj	Zahtevi za otvaranje fajla	Procesi koji su uspeli da otvore fajl
1	A-Rd, B-Rd, C-Wr, D-Rd, E-Wr	A, B, D
2	A-Wr, B-Wr, C-Rd, D-Rd, E-Rd	A

b.

```
Block* getDataBlock(FCB* file, unsigned long bn) {
   //ako je pokazivač na FCB jednak null, greška
   if (file == 0) return -1;
   //ako je broj blokova fajla manji ili jednak indeksu traženog, greška
   if (bn >= file->sizeInBlocks) return -1;
   //ako indeksni blok ne postoji (jednak je null), greška
   if (file->indexBlockNo == 0) return -1;

   //dohvatamo pokazivač na indeksni blok
   Block* index = getDiskBlock(file->indexBlockNo);
   //ako nismo uspeli da dohvatimo pokazivač, greška
   if (index == 0) return -1;

   //dok god ne dođemo do indeksnog fajla kome pripada traženi blok
   while (bn >= EntriesInBlock) {
      bn -= EntriesInBlock;
      //dohvatamo broj bloka sa narednim indeksnim blokom
      PBlockNo pbn = ((PBlockNo*)index)[EntriesInBlock];
      //ako taj indeksni blok ne postoji, greška
      if (pbn == 0) return -1;
      //dohvatamo pokazivač na naredni indeksni blok
      index = getDiskBlock(pbn);
      //ako nismo uspeli da dohvatimo pokazivač, greška
      if (index == 0) return -1;
   }
   //dohvatamo broj fizičkog bloka sa datim brojem logičkog bloka
   PBlockNo pbn = ((PBlockNo*)index)[bn];
   //pozivamo datu funkciju i vraćamo pokazivač na
   //učitani blok sa zadatim fizičkim rednim brojem
   return getDiskBlock(pbn);
}
```

8. (*Februar 2010*)

a. Neki fajl sistem podržava direktorijume sa strukturom usmerenog acikličkog grafa (*DAG*). Posmatraju se dva različita pristupa realizaciji operacije brisanja fajla sa zadatim simboličkim imenom (stazom, referencom): brisanje fajla i svih drugih referenci prilikom bilo kog (tj. prvog) zahteva sa brisanjem (tj. preko bilo koje reference) i brisanje fajla tek pri zahtevu za brisanjem preko poslednje preostale reference (pre toga se brišu samo reference, ne i fajl). Koja varijanta je po vašem mišljenu jednostavnija za implementaciju i zbog čega?

b. Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove. U jedan indeksni blok može da stane *N* pokazivača na blokove na disku. Odrediti maksimalnu veličinu fajla (u blokovima) za svaku od sledećih varijanti organizacije indeksa:
    1. Jedan indeksni blok na koga ukazuje određeno polje u *FCB*.
    2. Ulančavanje indeksnih blokova, pri čemu na prvi indeksni blok ukazuje određeno polje u *FCB*, a poslednji ulaz u svakom indeksnom bloku je pokazivač na sledeći indeksni blok u lancu
    3. Indeksiranje u dva nivoa, s tim da na indeksni blok prvog nivoa ukazuje određeno polje u *FCB*, a svaki indeks prvog nivoa sadrži pokazivače na indeksne blokove drugog nivoa.

Rešenje:

a. Jednostavnija je varijanta sa brisanjem fajla tek prilikom uklanjanja poslednje reference. Za to rešenje dovoljno je u FCB imati brojač referenci i ažurirati taj brojač prilikom svakog kreiranja ili brisanja reference, dok je dovoljna jednosmerna strukturna veza od referenci (u direktorijumima) prema fajlovima (tj. prema FCB), a ne i obratno. Za onu drugu varijantu potrebno je čuvati i veze u suprotnom smeru (od fajla prema referencama), kao i implementirati brisanje svih referenci prilikom brisanja fajla.

b. 1. N blokova 2. Nema logičkog ograničenja 3. N^2 blokova.

9. (*Jun 2010*)

a. Neki fajl sistem podržava montiranje nekog direktorijuma fajl sistema sa nekog eksternog uređaja ili udaljenog računara u bilo koji prazan lokani direktorijum, i to na više mesta (u različite prazne lokalne direktorijume). U fajl sistemu eksternog uređaja *X* nalazi se direktorijum `\first\second` u kome su tri fajla: `a.txt`, `b.txt` i `c.txt`. Šta će ispisati poslednja naredba u sledećoj sekvenci naredbi (sve su izvršene uspešno; prvi argument komandi `mount` i `copy` je izvorište, drugi odredište):

```
mount X:\first\second \home\buzz
mount X:\first\second \home\foo
del \home\foo\a.txt
copy \home\buzz\b.txt \home\foo\d.txt
dir \home\buzz
```

b. Neki fajl sistem koristi ulančani pristup alokaciji blokova za fajlove, pri čemu se u prvoj reči (tip `unsigned int`) svakog bloka alociranog za sadržaj fajla nalazi broj bloka koji je sledeći u lancu (0 za kraj). U `FCB` fajla polje `head` ukazuje na prvi, polje `tail` na poslednji blok u lancu alociranih blokova, a polje `size` sadrži broj blokova alociranih za sadržaj. Podsistem za alokaciju slobodnih blokova takođe ulančava slobodne blokove i pri alokaciji nekoliko slobodnih blokova od jednom vraća ceo lanac alociranih blokova. Postoji i podsistem koji kešira blokove sa sadržajem fajlova. U sistemu postoje realizovane sledeće funkcije:

`int alloc(int numOfBlocks,unsigned int* head,unsigned int* tail);`
alocira lanac od `numOfBlocks` blokova i broj prvog bloka u alociranom lancu upisuje u promenljivu na koju ukazuje argument `head`, a poslednjeg u promenljivu na koju ukazuje `tail`; u slučaju uspeha vraća 0, u slučaju greške vraća kod greške (!=0).

`int block_write(unsigned int blockNo,int word,int length,unsigned int* buf)`
u podsistemu koji kešira blokove, u blok `blockNo`, počev od reči sa brojem `word` (broji se počev od 0), upisuje `length` reči sa lokacije na koju ukazuje `buf`; u slučaju uspeha vraća 0, u slučaju greške vraća kod greške (!=0).

Realizovati funkciju
`int append(FCB* file, int numOfBlocks);`
koja sadržaj datog fajla na čiji `FCB` ukazuje prvi argument proširuje dodavanjem `numOfBlocks` slobodnih blokova na
kraj lanca. Ova funkcija u slučaju uspeha treba da vraća 0, a u slučaju greške da vraća kod greške (!=0).

Rešenje:

a. *Komanda*: `mount X:\first\second \home\buzz`
Rezultat: `\home\buzz ⇔ X:\first\second`

Komanda: `mount X:\first\second \home\foo`
Rezultat: `\home\foo ⇔ X:\first\second`

Komanda: `del \home\foo\a.txt`
Rezultat: `izbrisan je fajl a.txt iz X:\first\second`

Komanda: `copy \home\buzz\b.txt \home\foo\d.txt`
Rezultat: `fajl b.txt je kopiran u X:\first\second\d.txt`

Komanda: `dir \home\buzz`
Rezultat: `ispis sadržaja direktorijuma X:\first\second :`

```
b.txt
c.txt
d.txt
```

b.

```
int append(FCB* file, int numOfBlocks) {
   //promenljive za smeštanje broja prvog i poslednjeg alociranog bloka
   unsigned int appendChainHead = 0, appendChainTail = 0;

   //pokušaj alociranja numOfBlocks blokova
   int status = alloc(numOfBlocks, &appendChainHead, &appendChainTail);
   //ako dođe do greške pri alokaciji, vraćamo status greške
   if (status != 0) return status;

   //ako fajl već sadrži bar jedan blok
   if (file->tail) {
      //moramo da ažuriramo prvu reč poslednjeg bloka,
      //tako da "pokazuje" na prvi alocirani blok
      status = block_write(file->tail, 0, 1, &appendChainHead);
      //ako se desi greška pri upisu, vraćamo status greške
      if (status != 0) return status;
   }
   //u suprotnom, fajl ne sadrži nijedan blok
   //pa je potrebno ažurirati "pokazivač" na prvi
   //blok fajla tako da pokazuje na prvi alocirani blok
   else
      file->head = appendChainHead;

   //ažuriramo "pokazivač" na poslednji blok fajl tako
   //da "pokazuje" na poslednji alocirani blok
   file->tail = appendChainTail;
   //uvećavamo veličinu fajla za broj alociranih blokova
   file->size += numOfBlocks;

   //sve je proteklo bez greške, vraćamo 0
   return 0;
}
```

10. (*Septembar 2010*)

a. Objasniti zašto se prava pristupa do fajla (prava na izvršenje određenih operacija nad fajlom) po pravilu čuvaju u tabeli otvorenih fajlova koja pripada kontekstu procesa, a ne u globalnoj (sistemskoj) tabeli otvorenih fajlova zajedničkoj za sve procese.

b. Upoređuju se sledeća tri načina alokacije blokova za sadržaj fajla:
   A. Kontinualna alokacija, s tim da se za fajl odmah prilikom njegovog kreiranja alocira onoliko blokova kolika je maksimalna dozvoljena veličina fajla, iako ne moraju svi blokovi biti zauzeti sadržajem; u *FCB* je pokazivač na prvi blok i broj zauzetih blokova.
   B. Ulančana alokacija, s tim da su u *FCB* pokazivači na prvi i poslednji blok sa sadržajem fajla, a pokazivači na sledeći blok u fajlu su na kraju svakog bloka sa sadržajem.
   C. Indeksna alokacija, s tim da je u *FCB* pokazivač na (jedini) indeksni blok.

   Za svaki od ovih načina alokacije posmatra se broj blokova koje je potrebno učitati u memoriju, pod sledećim pretpostavkama:
   - svaka operacija posmatra se nezavisno, za isto početno stanje;
   - početno stanje je takvo da je u memoriju učitan *FCB* i nijedan drugi blok, osim onih koji su eksplicitno navedeni u operaciji.

   Operacije su sledeće:
   1. Direktan pristup *n*-tom bloku sa sadržajem u odnosu na početak fajla (nije učitan indeksni blok za način *C*).
   2. Sekvencijalni pristup: pristup bloku *n*+1 nakon što je već učitan blok *n* (time je već učitan i indeksni blok za način *C*).
   3. Proširenje sadržaja fajla (ispod maksimalne dozvoljene veličine fajla) slobodnim blokom koji je već alociran i učitan u memoriju (nije učitan indeksni blok za način *C*).

   U donju tabelu upisati koliko blokova treba učitati za svaki od navedenih načina i svaku od navedenih operacija.

	A	B	C
1.			
2.			
3.			

Rešenje:

a. Po pravilu, opšta prava pristupa do fajla definišu se na nivou korisnika. Svaki proces se pokreće „u vlasništvu" datog korisnika, pa time i nasleđuje prava pristupa do fajla podešena za tog korisnika. Zato se prava pristupa do fajla razlikuju od procesa do procesa i nisu globalna za dati fajl u celom sistemu.

Bez obzira na opšta prava koja korisnik koji je kreirao proces ima do fajla, API za fajl sistem po pravilu omogućava da proces otvori fajl i da pri tome deklariše kako će ga koristiti (koje grupe operacija će koristiti). Ovo omogućava kontrolu grešaka u samom programu, tako što sistem zabranjuje one operacije koje nisu predviđene pri otvaranju fajla, smatrajući ih prekršajem zbog greške u programu. Zato se prava pristupa razlikuju od procesa do procesa, čak i za istog korisnika.

b. Popunjena tabela:

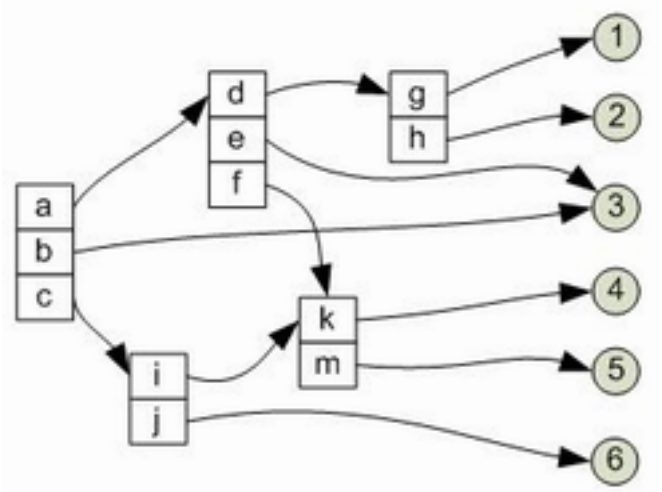
	A	B	C
1.	1	n	2
2.	1	1	1
3.	0	1	1

11. (*Oktobar 2010*)

a. U nekom fajl sistemu struktura direktorijuma je aciklički usmereni graf (DAG). Postoje sledeće sistemske komande:
   - `copy src dst`: kopira fajl do koga se može doći stazom `src` i registruje kopiju pod punim imenom (stazom) `dst`;
   - `alias src dst`: registruje fajl do koga se može doći stazom `src` pod još jednim punim imenom (stazom) `dst`, bez kopiranja samog fajla.

   Na donjoj slici prikazano je posmatrano početno stanje sistema; direktorijumi su prikazani kao pravougaonici sa ulazima u kojima su nazivi, a fajlovi su prikazani krugovima. Prikazati stanje ovog sistema nakon izvršavanja sledeće sekvence komandi:

```
copy   /a/e     /a/d/n
alias  /a/d/n   /c/i/p
copy   /b       /c/q
alias  /b       /r
```

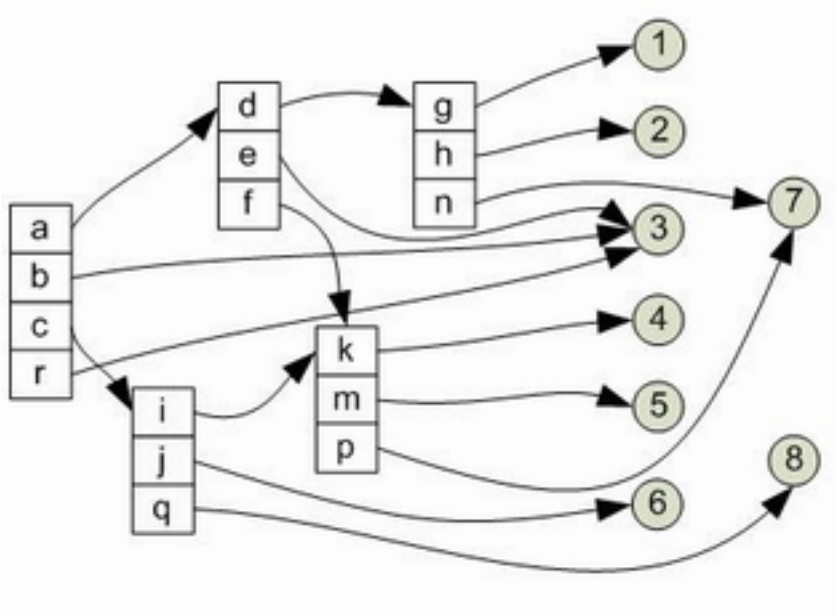


b. U nekom fajl sistemu direktorijum i fajl se uopšteno nazivaju *ulazom* (*entry*) i predstavljaju se istom strukturom FCB. U sistemu su implementirane sledeće elementarne operacije:
   - `int f_find_entry(char* name, FCB** fcb)`: pronalazi ulaz sa zadatom punom stazom (imenom) i u `*fcb` upisuje pokazivač na `FCB` strukturu tog ulaza;
   - `int f_find_entry(FCB* dir, char* name, FCB** fcb)`: pronalazi ulaz u direktorijumu datom prvim argumentom sa zadatim imenom (nazivom bez staze) datim drugim argumentom i u `*fcb` upisuje pokazivač na `FCB` strukturu tog ulaza;
   - `int f_create_file(FCB** fcb)`: kreira novi fajl u fajl sistemu i u `*fcb` upisuje pokazivač na inicijalizovanu `FCB` strukturu tog kreiranog fajla;
   - `f_register_entry(FCB* dir, FCB* entry, char* name)`: u direktorijum dat prvim argumentom registruje ulaz dat drugim argumentom pod nazivom datim trećim argumentom;
   - `f_remove_entry(FCB* dir, char* name)`: iz direktorijuma datog prvim argumentom izbacuje ulaz pod nazivom datim drugim argumentom (ne briše taj ulaz, već ga samo izbacuje iz direktorijuma, a taj ulaz ostaje kao objekat u fajl sistemu);
   - `f_delete_entry(FCB* entry)`: iz fajl sistema uklanja (briše, uništava) dati ulaz.

   Sve ove operacije u slučaju uspeha vraćaju vrednost 0, a u slučaju greške ostavljaju sistem u konzistentnom stanju, kao da promena nije ni započeta i vraćaju kod greške koji je različit od 0. Korišćenjem ovih operacija implementirati sledeće operacije koje treba da vraćaju status na isti način:
   - `int f_create_file(char* dname, char* fname)`: u direktorijumu sa punim imenom (sa stazom) zadatim prvim argumentom kreira novi fajl sa imenom zadatim drugim argumentom;
   - `int f_move_entry(char* sdname, char* fname, char* tdname)`: iz direktorijuma sa punim imenom zadatim prvim argumentom premešta ulaz sa imenom zadatim drugim argumentom u direktorijum sa punim imenom zadatim trećim argumentom.

Rešenje:

a. Stanje sistema nakon izvršenja date sekvence naredbi:



b. Implementacija traženih operacija:

```
int f_create_file(char* dname, char* fname) {
   int status = 0;
   FCB* dir = 0;                                  //pokazivač na FCB traženog direktorijuma
   status = f_find_entry(dname,&dir);             //tražimo zadati direktorijum
   if (status != 0) return status;                //ako je došlo do greške, povratak
   FCB* file = 0;                                 //pokazivač na FCB novog fajla
   status = f_create_file(&file);                 //pravimo novi fajl
   if (status != 0) return status;                //ako je došlo do greške, povratak
   status = f_register_entry(dir,file,fname);     //registrujemo novi fajl u direktorijumu
   if (status != 0) f_delete_entry(file);         //ako je došlo do greške, brišemo fajl
   return status;                                 //vraćamo konačni status
}

int f_move_entry(char* sdname, char* fname, char* tdname) {
   int status = 0;
   FCB* sdir = 0;                                 //pokazivač na FCB izvornog direktorijuma
   status = f_find_entry(sdname,&sdir);           //tražimo izvorni direktorijum
   if (status != 0) return status;                //ako je došlo do greške, povratak
   FCB* tdir = 0;                                 //pokazivač na FCB krajnjeg direktorijuma
   status = f_find_entry(tdname,&tdir);           //tražimo krajnji direktorijum
   if (status != 0) return status;                //ako je došlo do greške, povratak
   FCB* file = 0;                                 //pokazivač na FCB traženog fajla
   status = f_find_entry(sdir,fname,&file);       //tražimo zadati fajl
   if (status != 0) return status;                //ako je došlo do greške, povratak
   status = f_remove_entry(sdir,fname);           //brišemo fajl iz izvornog direktorijuma
   if (status != 0) return status;                //ako je došlo do greške, povratak
   status = f_register_entry(tdir,file,fname);    //dodajemo fajl u krajnji direktorijum
   if (status != 0)                               //ako je došlo do greške
      f_register_entry(sdir,file,fname);          //dodajemo ponovo fajl u izvorni
                                                  //direktorijum (konzistentno stanje)
   return status;                                 //vraćamo konačni status
}
```

12. (*Januar 2009*)

a. Neki fajl sistem podržava organizaciju direktorijuma u obliku usmerenog acikličkog grafa (*DAG*). Zbog ovakve organizacije neophodni su sistemski pozivi za dve različite operacije kopiranja datog fajla na odredišno mesto. Predložiti ova dva sistemska poziva, napisati njihove deklaracije na jeziku *C* i objasniti njihovo značenje. U oba poziva zadaju se identifikator fajla koji se kopira („ručka“) i naziv (puna putanja) koji će imati iskopirani fajl.

b. U nekom fajl sistemu direktorijum se implementira kao običan fajl čiji je sadržaj organizovan korišćenjem *hash* tabele na sledeći način. U prvom delu sadržaja je sama *hash* tabela koja ima 4 ulaza (0..3). Svaki ulaz ima sadržaj koji predstavlja redni broj (počev od 1) zapisa u ulančanoj listi zapisa za taj ulaz. Ovi zapisi nalaze se u nastavku *hash* tabele u sadržaju. Vrednost 0 u ulazu označava praznu listu. U svakom zapisu u listama nalazi se naziv elementa direktorijuma (niz znakova), identifikator objekta u fajl sistemu kojim je predstavljen taj element (broj *inode* objekta) i pokazivač na sledeći zapis u listi zapisa za isti ulaz u *hash* tabeli. Ovaj pokazivač ima vrednost koja se tumači na isti način kao i u ulazu *hash* tabele (0 – *null*, >0 – redni broj). U nekom direktorijumu, koji je inicijalno bio prazan, kreirani su tim redom sledeći fajlovi (u trećoj koloni su date *hash* vrednosti njihovih naziva):

Naziv	*inode#*	*hash value*
`alfa`	100	3
`beta`	200	0
`gama`	300	1
`delta`	400	3
`eta`	500	1

Popuniti sadržaj *hash* tabele i lista zapisa u njenom nastavku za posmatrani direktorijum. *Hash* tabela:

Ulaz	Pokazivač na prvi zapis u listi
0	2
1	3
2	0
3	1

Lista zapisa:

Ulaz	Naziv	*inode#*	Sledeći
1	`alfa`	100	4
2	`beta`	200	0
3	`gama`	300	5
4	`delta`	400	0

5	eta	500	0

Rešenje:

a. Pravi potpuno novi fajl na odredištu sa iskopiranim sadržajem, tj. u odredišnom direktorijumu kreira ulaz sa zadatim nazivom koji ukazuje na novokreirani fajl kao objekat u sistemu (novi *inode*):

```
error_code fileDeepCopy (FHANDLE src, char* dstFileName);
```

Kopira samo referencu, tj. u odredišnom direktorijumu kreira ulaz sa zadatim nazivom koji ukazuje na isti fajl kao objekat u sistemu (isti *inode*):

```
error_code fileShallowCopy (FHANDLE src, char* dstFileName);
```

Reakcija oba sistemska poziva na izuzetne situacije može da bude ista. Izuzetne situacije su, na primer: odredišni direktorijum ne postoji, u odredišnom direktorijumu već postoji ulaz sa datim nazivom, pozivajući proces nema pravo upisa u odredišni direktorijum.

b. Popunjenje tabele:

Hash tabela:

Ulaz	Pokazivač na prvi zapis u listi
0	2
1	3
2	0
3	1

Lista zapisa:

Ulaz	Naziv	*inode#*	Sledeći
1	alfa	100	4
2	beta	200	0
3	gama	300	5
4	delta	400	0
5	eta	500	0

13. (*Februar 2009*)

a. U operativnim sistemima uobičajen je protokol u kome se svaki proces pokreće „u ime“ nekog korisnika (koji je njegov „pokretač“ ili „vlasnik“), i to istog onog u čije ime je pokrenut i roditeljski proces koji je pokrenuo taj proces odgovarajućim sistemskim pozivom za kreiranje procesa, pri čemu se početni procesi (npr. školjka) pokreću u ime korisnika koji se prijavio na sistem implicitno po prijavljivanju (*log-in*) tog korisnika. Prava pristupa do fajlova određuju se na nivou korisnika i grupa korisnika. Precizno obrazložiti kakve bi posledice u smislu pristupa do fajlova imalo uvođenje mogućnosti da se u sistemskom pozivu za kreiranje procesa proces-dete kreira tako da se izvršava „u ime“ nekog drugog korisnika, a ne istog onog u čije ime se izvršava i proces-roditelj, pri čemu bi ta mogućnost bila dozvoljena svim procesima.

b. Neki fajl sistem koristi *FAT* (*File Allocation Table*) pristup alokaciji blokova za fajlove. *FAT* se kešira u memoriju u strukturu deklarisanu na sledeći način:

```
typedef unsigned long int FATIndex;    // FAT entry number
const FATIndex FATSize = ...;          // FAT (and disk) size

struct FATEntry {
   unsigned short int isFree;    // Is this block free (1) or allocated (0)
   FATIndex next;    // Next block in the file's block chain; 0 for terminator
};

FATEntry fat[FATSize] ; // The FAT
```

Kontrolni blok fajla (*File Control Block*, *FCB*) je deklarisan na sledeći način:

```
struct FCB {
   …
   FATIndex sizeInBlocks;  // Current file size in number of blocks
   FATIndex firstBlockNo;  // Pointer to the first block (FAT entry number)
   …
};
```

Potrebno je realizovati funkciju:

```
void clearFile (FCB* file);
```

koja treba da „obriše“ sadržaj fajla, odnosno dealocira blokove koje fajl zauzima, bez uklanjanja fajla kao objekta iz fajl sistema.

Rešenje:

a. To bi otvorilo veliku „sigurnosnu rupu“ (*security hole*) u sistemu provere prava pristupa do fajlova. Naime, sistem provere prava pristupa do fajla upoređuje pravo *procesa* da izvrši odgovarajući sistemski poziv za pristup do nekog fajla. Pri tome uzima u obzir onog korisnika „u čije ime“ se taj proces izvršava, odnosno korisnika koji je „vlasnik“ tog procesa. Na osnovu odgovarajućih pravila, proverava se da li je taj korisnik „vlasnik“ fajla ili pripada određenoj grupi korisnika itd. Ako bi se proces-dete kreirao tako da se izvršava u ime nekog drugog korisnika, onda bi na taj način neko slučajno ili zlonamerno mogao da kreira proces koji bi smeo da pristupi fajlovima do kojih inače korisnik u čije ime se izvršava roditeljski proces ne bi imao prava pristupa, što bi probilo zamišljeni sistem zaštite pristupa do fajlova.

b.

```
void clearFile (FCB* file) {
   //ako je pokazivač na FCB jednak null ili ako je veličina fajla 0, povratak
   if (file == NULL || file->sizeInBlocks == 0) return;
   //prolazimo kroz sve blokove fajla i postavljamo fleg "isFree" na 1
   for (FATIndex fi = file->firstBlockNo;  fi != 0; fi = fat[fi].next)
      if (fi >= 0 && fi < FATSize) fat[fi].isFree = 1;
   //ažuriramo FCB samog fajla
   file->firstBlockNo = 0;
```

```
    file->sizeInBlocks = 0;
}
```

14. (*Jun 2009*)

a. U opštem slučaju, da li ista vrednost identifikatora („ručke“) fajla (`FHANDLE`) koju su dva procesa dobila prilikom operacija otvaranja fajla koje su izvršili, znači da oni pristupaju istom fizičkom fajlu preko tih identifikatora? Precizno obrazložiti.

b. Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove. U strukturi `FCB` polje `index` ukazuje na indeks tog fajla; i struktura `FCB` i indeks su učitani u memoriju (keširani) prilikom otvaranja fajla. U indeksu, vrednost 0 označava da dati blok nije alociran (*null*). Operacija `freeBlock` dealocira dati blok na disku. Date su sledeće deklaracije:

```
typedef unsigned long int BlockNo;     // Block number
const int FIndexSize = ...;            // File index size
typedef BlockNo FIndex[FIndexSize]; // File index

struct FCB {
  …
  long int sizeInBlocks;         // Current file size in number of blocks
  FIndex index;                        // File index
};

void freeBlock(BlockNo);
```

Potrebno je realizovati funkciju:

```
void clearFile (FCB* file);
```

koja treba da „obriše“ sadržaj fajla, odnosno dealocira blokove koje fajl zauzima, bez uklanjanja fajla kao objekta iz fajl sistema.

Rešenje:

a. Ne. Stvarna vrednost identifikatora (ručke) u mnogim sistemima ima samo lokalni opseg, unutar procesa koji je otvorio fajl, jer predstavlja identifikator ulaza u tabeli otvorenih fajlova samo tog procesa (npr. broj/indeks ulaza u toj tabeli). Zbog toga dva procesa mogu imati istu ovakvu vrednust, a da zapravo ti ulazi predstavljaju sasvim različite fajlove.

b.

```
void clearFile (FCB* file) {
   //ako je pokazivač na FCB jednak null, greška (povratak iz f-je)
   if (file == 0) return;
   //čitamo svaki ulaz indeksa zadatog fajla
   for (int i = 0; i < FIndexSize; i++) {
      //i dohvatamo broj fizičkog bloka
      BlockNo b = file->index[i];
      //ako je blok alociran
      if (b != 0) {
         //dealociramo ga
         freeBlock(b);
         //ažuriramo dealocirani ulaz indeksa zadatog fajla
         file->index[i] = 0;
      }
   }
   //postavljamo veličinu fajla na nula blokova
   file->sizeInBlocks = 0;
}
```

15. (*Septembar 2009)*

a. Neki fajl sistem pruža sledeće operacije u svom *API* za tekstualne fajlove:
   - `FHANDLE open(char* filename)`: otvara fajl sa datim imenom.
   - `void close(FHANDLE)`: zatvara dati fajl.
   - `int size(FHANDLE)`: vraća trenutnu veličinu sadržaja fajla u znakovima.
   - `void append(FHANDLE,int)`: proširuje sadržaj fajla za dati broj znakova na kraju.
   - `void seek(FHANDLE,int)`: postavlja kurzor datog fajla na datu poziciju (redni broj znaka počev od 0).
   - `void write(FHANDLE,char*)`: na poziciju kurzora datog fajla upisuje dati niz znakova, ne uključujući završni znak `'\0'`, i pomera kurzor iza upisanog niza znakova.

   Napisati program koji na kraj postojećeg fajla sa imenom `proba.txt` upisuje sve što je uneseno preko standardnog ulaza, sve dok se na ulazu ne unese znak `'X'`. Zanemariti sve potencijalne greške u ulazu/izlazu.

b. Neki fajl sistem koristi indeksirani pristup alokaciji fajlova. U strukturi *FCB* polje *index* predstavlja indeks tog fajla; i struktura *FCB* i indeks su učitani u memoriju (keširani) prilikom otvaranja fajla. U indeksu, vrednost 0 označava da dati blok nije alociran (*null*). Operacija `allocBlock` alocira novi blok na disku i vraća njegov broj; u slučaju greške, ova funkcija vraća 0. Date su sledeće deklaracije:

```
typedef unsigned long int BlockNo;        // Block number
const int FIndexSize = ...;               // File index size
typedef BlockNo FIndex[FIndexSize];       // File index
const unsigned long int FBlockSize = ...; // Block size in bytes

struct FCB {
  ...
  unsigned long int size;                 // Current file size in bytes
  FIndex index;                           // File index
};

BlockNo allocBlock();
```

   Potrebno je realizovati funkciju:
   `unsigned long int append (FCB* file, unsigned long int bytes);`
   koja treba da „proširi“ sadržaj fajla za dati broj bajtova, po potrebi alocirajući nove blokove za sadržaj fajla. Ova funkcija treba da vrati broj bajtova za koji je uspela da proširi fajl; u slučaju uspeha, ovaj broj je jednak traženom, a u slučaju nedostatka prostora, on je manji. Veličina sadržaja fajla ne mora biti zaokružena na veličinu bloka.

Rešenje:

a.

```
#include <stdio.h>

void main () {
   FHANDLE output = open("proba.txt");  //otvaramo fajl sa zadatim imenom
   int size = size(output);             //dohvatamo trenutnu veličinu fajla
   char c[2];                           //upisujemo znak po znak, ali niz znakova
   c[1]='\0';                           //koji upisujemo mora da se završi sa '\0'
   while () {
      getc(&c[0]);                      //čitamo znak sa standardnog ulaza
      if (c[0] == 'X') break;           //ako je pročitani znak 'X', prekidamo slanje
      append(output, 1);                //proširujemo fajl za jedan znak na kraju
      seek(output,size++);              //pomeramo kursor na kraj fajla
      write(output,c);                  //upisujemo pročitani znak na kraj fajla
   }
   close(output);                       //zatvaramo fajl
}
```

b.

```
unsigned long int append (FCB* file, unsigned long int bytes) {
  if (file==0 || bytes==0) return 0;
  // Allocate the internal fragment of the last block:
  unsigned long int remainder = file->size%FBlockSize;
  if (remainder>0) remainder = FBlockSize-remainder;
  if (bytes<=remainder) {
      file->size+=bytes;
      return bytes; // The fragment is big enough
  }
  // The fragment is not big enough, continue
  file->size+=remainder;
  unsigned long int bytesAllocated = remainder;
  bytes-=remainder;
  long int block = file->size/FBlockSize;
  if (remainder!=0) block++;
  // Allocate as many new blocks as necessary:
  while (bytes>=FBlockSize && block<FIndexSize) {
      BlockNo newBlock = allocBlock();
      if (newBlock==0) return bytesAllocated;
      file->index[block++]=newBlock;
      file->size+=FBlockSize;
      bytes-=FBlockSize;
      bytesAllocated+=FBlockSize;
  }
  if (bytes==0 || block>=FIndexSize) return bytesAllocated;
  // Allocate one new block for the remainder:
  BlockNo newBlock = allocBlock();
  if (newBlock==0) return bytesAllocated;
  file->index[block]=newBlock;
  file->size+=bytes;
  bytesAllocated+=bytes;
  return bytesAllocated;
}
```

16. (*Oktobar 2009)*

a. U nekom fajl sistemu struktura direktorijuma ima oblik usmerenog acikličkog grafa (*DAG*), pri čemu se isti fajl (kao objekat u fajl sistemu) može referencirati iz više različitih direktorijuma pod različitim imenima. Na primer, sledeće staze predstavljaju isti fajl:
`/mydocs/letters/to_john.doc` i `/correspondence/john/letter_17_09_2009.doc`.
Gde se čuva ime fajla u ovom sistemu, da li kao svaki drugi atribut fajla u *FCB* ili negde drugde? Precizno obrazložiti odgovor.

b. Neki fajl sistem koristi *FAT* (*File Allocation Table*) pristup alokaciji blokova za fajlove. *FAT* se kešira u memoriju u strukturu deklarisanu na sledeći način:

```
typedef unsigned long int FATIndex;       // FAT entry number
const FATIndex FATSize = ...;             // FAT (and disk) size

struct FATEntry {
  FATIndex next;              // Next block in the block chain; 0 for terminator
};

FATEntry fat[FATSize] ;             // The FAT
FATIndex freeHead;                  // The head of the list of free blocks
FATIndex freeTail;                  // The tail of the list of free blocks
```

Slobodni ulazi ulančani su u jednostruku listu na čiji prvi ulaz pokazuje `freeHead`, a na poslednji `freeTail`.
Kontrolni blok fajla (*File Control Block*, *FCB*) je deklarisan na sledeći način:

```
struct FCB {
  ...
  FATIndex sizeInBlocks;      // Current file size in number of blocks
  FATIndex head;              // Pointer to the first block (head FAT entry number)
  FATIndex tail;              // Pointer to the last block (tail FAT entry number)
  ...
};
```

Potrebno je realizovati funkciju:

```
void clearFile (FCB* file);
```

koja treba da „obriše“ sadržaj fajla, odnosno dealocira blokove koje fajl zauzima, bez uklanjanja fajla kao objekta iz fajl sistema.

Rešenje:

a. U ovom sistemu ime fajla nije atribut fajla kao objekta koji se čuva u FCB, jer ono nije jedinstveno. Ime fajla je zapravo identifikator fajla u samo jednom direktorijumu iz koga se taj fajl referencira, a kojih može biti više. Zbog toga se ime fajla čuva u samom direktorijumu, tj. u strukturi koja implementira direktorijum kao objekat fajl sistema, i u svakom direktorijumu u kome je fajl sadržan ono je potencijalno različito. Direktorijum preslikava to ime (kao identifikator) u fajl kao jedinstveni objekat sa identitetom (predstavljen strukturom FCB).

b.

```
void clearFile (FCB* file) {
   //ako je pokazivač na FCB fajla jednak null, ili ako je veličina fajla nula
   //ili ako je jedan od pokazivača na prvi ili poslednji blok fajla nula
   //samo izlazimo iz funkcije, jer nemamo šta da radimo ili je to stanje greška
   if (file==0 || file->sizeInBlocks == 0 || file->head==0 || file->tail==0)
      return;

   //ako postoje slobodni blokovi na disku, onda samo
   //nadovezujemo listu blokova fajla na listu slobodnih blokova
   if (freeHead != 0)
      fat[freeTail].next = file->head;
   //u suprotnom, slobodnih blokova nema, pa blokovi fajla postaju
   //jedini slobodni blokovi (prvi slobodni blok = prvi blok fajla)
   else
      freeHead = file->head;

   //ažuriramo pokazivač na poslednji slobodni blok
   freeTail = file->tail;

   //ažuriramo vrednosti u FCB-u fajla
   file->head = 0;
   file->tail = 0;
   file->sizeInBlocks = 0;
}
```

17. (*Januar 2008*)

a. U cilju optimizacije vremena pristupa fajlovima, neki fajl sistem za isti fajl pokušava da alocira blokove na disku koji su međusobno što bliži. Koja struktura podataka za evidenciju slobodnih blokova je efikasnija u smislu brzine alokacije blokova prilikom proširenja postojećeg fajla – ulančana lista slobodnih blokova ili bit-vektor? Obrazložiti.

b. Neki operativni sistem na komandu `dir` iz komandne linije ispisuje listu fajlova u direktorijumu koji je naveden kao parametar ove komande, s tim da za svaki fajl ispiše najpre njegov jedinstveni identifikator u fajl sistemu, a onda i njegov nejedinstveni naziv i eventualno ostale atribute, ukoliko su zahtevani konfiguracionim parametrima ove komande. Fajl sistem podržava strukturu direktorijuma u obliku usmerenog acikličkog grafa (*DAG*), a isti fajl može imati različita imena u različitim direktorijumima. Dati su rezultati nekoliko sukcesivnih komandi `dir`:

Komanda	Rezultat
`dir /`	`1 a.dir` `2 b.dir`

`dir /a`	`3 c.dir`
	`6 d.txt`
	`7 e.doc`

`dir /b`	`5 f.exe` `7 g.doc`
`dir /a/c`	`4 h.exe` `5 k.exe`

Nakon komande brisanja fajla `/b/f.exe`, koja je implementirana tako da briše fajl i sve reference na njega, popuniti prethodnu tabelu rezultatima koje ispisuju date komande `dir`.

Rešenje:

a. Bit-vektor, zato što omogućava direktan pristup. Kod alokacije novih blokova za postojeći fajl može se direktno (i efikasno) pristupiti delu vektora koji reprezentuje blokove koji su susedni blokovima koje taj fajl već zauzima i među njima pronaći slobodni. Ulančana lista, sa druge strane, zahteva sekvencijalni prolaz kroz listu slobodnih blokova, od početka pa do mesta koje je blizu blokovima koji pripadaju datom fajlu.

b. Popunjena tabela nakon brisanja svih referenci na fajl sa ID = 5:

Komanda	Rezultat
`dir /`	`1 a.dir` `2 b.dir`

`dir /a`	`3 c.dir`
	`6 d.txt`
	`7 e.doc`

`dir /b`	`7 g.doc`
`dir /a/c`	`4 h.exe`

18. (*Februar 2008*)

a. U nekom fajl sistemu primenjuje se sistem provere prava pristupa koja se definišu preko tri bita (rwx) za korisnika koji je kreirao fajl (tzv. vlasnik, engl. *owner*) i za sve ostale korisnike. Dat je prikaz jedne interakcije korisnika i sistema preko komandne linije:

```
> dir -all
my.dir   02/06/2008 12:34 rw-r--
  a.txt  02/06/2008 12:40 rw-r--
  p.exe  02/06/2008 12:45 rwxr--
> run p.exe
Error: Access denied.
> delete a.txt
```

Da li će ova poslednja operacija brisanja fajla (`delete a.txt`) biti uspešna? Obrazložiti.

b. Posmatraju se tri načina rukovanja slobodnim prostorom u nekom fajl sistemu:
   1. Pomoću bit-vektora koji se smešta u tačno određene blokove na disku.
   2. Pomoću ulančane liste, pri čemu se pokazivači za ulančavanje smeštaju neposredno u slobodne blokove.
   3. Pomoću ulančane liste slobodnih blokova, ali uz korišćenje *FAT* (*File Allocation Table*).

   Uporediti sledeće karakteristike ovih tehnika i odgovoriti koja je od navedenih tehnika efikasnija:
   i. Koja tehnika, 1 ili 2, je generalno efikasnija u smislu korišćenja prostora na disku i količine korisnog prostora za podatke?
   ii. Koja tehnika, 1 ili 2, je generalno efikasnija u smislu brzine alokacije nekoliko slobodnih blokova na disku?
   iii. Koja tehnika, 2 ili 3, je generalno efikasnija u smislu brzine alokacije nekoliko slobodnih blokova na disku?

   Odgovore obrazložiti.

Rešenje:

a. Neće, već će sistem prijaviti grešku zbog nedozvoljenog pristupa. Naime, iz rezultata komande run p.exe vidi se da korisnik čija je interakcija prikazana nema prava izvršavanja ovog fajla, pa je on onaj na koga se odnose druga tri znaka rwx u listi prava pristupa. Bez obzira da li sistem operaciju brisanja fajla smatra operacijom izmene (pisanja) direktorijuma ili samog fajla, ni jedna od njih nije dozvoljena datom korisniku, pa neće biti dozvoljeno ni brisanje.

b. Odgovori na pitanja:

i. Tehnika 2 (ulančana lista), jer se kod nje strukture za evidenciju slobodnog prostora smeštaju u same slobodne blokove, dok se kod bit-vektora alocira poseban prostor na disku za smeštanje bit-vektora i time smanjuje koristan prostor za podatke.

ii. Tenika 1 (bit-vektor), jer se pretraga i alokacija slobodnih blokova generalno svodi na učitavanje jednog ili nekoliko susednih blokova sa delom bit-vektora, a zatim na iterativnom postupku pristupa mašinskim rečima i ispitivanju bita u njima, što su efikasne mašinske operacije. Osim toga, ovi blokovi sa bit-vektorom se mogu keširati u posebnom kešu i zato biti stalno prisutni u operativnoj memoriji. Kod tehnike 2 se u svakoj iteraciji prolaska kroz listu pristupa novom bloku na disku.

iii. Tehnika 3 (FAT). Iako se obe svode na prolazak kroz ulančanu listu, kod FAT se pristupa manjem fizičkom prostoru, odnosno učitavanju manjeg broja blokova sa samom FAT. Kod tehnike 2 se u svakoj iteraciji prolaska kroz listu pristupa novom bloku na disku. Osim toga, FAT se može u celini ili delimično keširati u posebnom kešu i zato biti stalno prisutna u memoriji.

19. (*Jun 2008*) Klasičan koncept fajla sa svojim operacijama ima grubo sledeću semantiku:

- fajl se može kreirati, pri čemu se veličina njegovog sadržaja postavlja na 0 (prazan sadržaj). Operacija kreiranja fajla ujedno i otvara taj fajl u kontekstu procesa koji ga je kreirao.
- postojeći (već kreiran) fajl se može otvoriti i zatvoriti.
- proces može direktno pristupati bilo kom delu fajla, na nivou bajta, ali samo u okvirima njegove trenutne veličine sadržaja. Svaki pristup iza kraja trenutnog sadržaja fajla je nelegalan i generiše grešku. Pristup podrazumeva mogućnost čitanja ili upisa niza bajtova proizvoljne veličine, pod navedenim ograničenjima.
- postojeći sadržaj fajla se može proširiti posebnom operacijom dodavanja proizvoljnog sadržaja na kraj, pri čemu postoji podrazumevani sadržaj ako se on u operaciji ne navede eksplicitno (npr. nule).

Umesto ovakvog koncepta, u nekom fajl sistemu koristi se koncept fajla sa nešto izmenjenom semantikom, tako da fajl predstavlja veliki, ali ograničeni virtuelni memorijski prostor na nekom uređaju (npr. disku). Svi fajlovi imaju isti veliki „kapacitet“, odnosno veličinu tog prostora. Definisan je podrazumevani sadržaj celog tog prostora (npr. sve nule). Proces može proizvoljno pristupati bilo kom delu tog prostora, čitajući ili upisujući niz bajtova proizvoljne veličine (naravno, u okvirima navedenog virtuelnog prostora). Važno je uočiti da nije neophodno da proces „proširuje“ sadržaj fajla kako je gore opisano, pa ta operacija i ne postoji. Naravno, fajl se i dalje može kreirati, otvoriti i zatvoriti.

a. Predložiti aplikativni programski interfejs ovako izmenjenog koncepta fajla. Navesti zaglavlja svih relevantnih operacija prema datim opisima i objasniti njihovo značenje. Nije potrebno voditi računa o pravima pristupa do fajla za korisnike i njihove procese.
b. Predložiti i objasniti implementaciju ovakvog koncepta fajla. Konkretno, objasniti sledeće:
    i. koju biste metodu alokacije koristili, kontinualnu, ulančanu (osnovnu ili *FAT*), ili indeksnu (u jednom ili više nivoa ili kombinovano)?
    ii. precizno objasnite kako biste obezbedili da sadržaj fajla zauzima na uređaju samo neophodan prostor, a nikako prostor za smeštanje celog virtuelnog sadržaja fajla (koji velikim delom može imati podrazumevani sadržaj jer nije redefinisan, odnosno ostao je „prazan“).
    iii. opišite kako biste principijelno realizovali operacije čitanja i upisa niza bajtova sa tako predloženim načinom alokacije fajla.

Rešenje:

a. *Operacija*: `FHANDLE fcreate(char* fname);`
Značenje: Kreira i otvara fajl sa zadatim punim imenom i vraća „ručku“ do kreiranog fajla ukoliko je operacija uspela. Puno ime uključuje i stazu do direktorijuma u kome se kreira fajl.
Ukoliko operacija nije uspela, FHANDLE nosi informaciju o vrsti greške (npr. negativne vrednosti). Greške uključuju nepostojanje direktorijuma u kome se kreira fajl, slučaj da već postoji fajl sa datim imenom, nedovoljno prostora na uređaju za kreiranje fajla, grešku u pristupu uređaju.

Operacija: `FHANDLE fopen(char* fname);`
Značenje: Otvara postojeći fajl sa zadatim punim imenom i vraća „ručku“ do otvorenog fajla ukoliko je operacija uspela. Ukoliko operacija nije uspela, FHANDLE nosi informaciju o vrsti greške (npr. negativne vrednosti). Greške uključuju nepostojanje fajla sa datim punim imenom i grešku u pristupu uređaju.

Operacija: `void fclose(FHANDLE);`
Značenje: Zatvara otvoreni fajl identifikovan datom ručkom.

Operacija: `int fread(FHANDLE, long fpos, void* buf, long size);`
Značenje: Iz fajla identifikovanog datom ručkom učitava blok bajtova veličine size sa pozicije fpos u odnosu na početak fajla („adresa“ u celokupnom virtuelnom prostoru fajla) u prostor u memoriji na adresi buf. Vraća 0 ako je operacija uspela. Ukoliko operacija nije uspela, vraća informaciju o vrsti greške (npr. negativne vrednosti). Greške uključuju prekoračenje virtuelnog opsega fajla, neispravnu ručku, grešku u pristupu uređaju.

Operacija: `int fwrite(FHANDLE, long fpos,void* buf, long size);`
Značenje: U fajl identifikovan datom ručkom upisuje blok bajtova veličine size na pozicije fpos u odnosu na početak fajla („adresa“ u celokupnom virtuelnom prostoru fajla) iz prostora u memoriji na adresi buf.

Vraća 0 ako je operacija uspela. Ukoliko operacija nije uspela, vraća informaciju o vrsti greške (npr. negativne vrednosti). Greške uključuju prekoračenje virtuelnog opsega fajla, neispravnu ručku, grešku u pristupu uređaju.

b. Opisani koncept fajla zapravo je logički ekvivalentan konceptu virtuelne memorije, pa i organizacija može da bude slična.

i. Najpogodnije je koristiti indeksnu alokaciju sa indeksom u više nivoa.
   Zbog mogućnosti da ceo virtuelni prostor fajla bude „retko popunjen" na potpuno proizvoljnim mestima, pri čemu nikako ne treba alocirati fizički prostor na uređaju za ceo (veliki) virtuelni prostor fajla, kontinualna alokacija i ulančana alokacija (uključujući i FAT kao njenu varijantu) ne dolaze u obzir. Slično, kombinovana indeksna varijanta nema mnogo smisla, jer se fajl „ne širi od početka", pa povećanje stepena indeksiranja za „veće fajlove" sa smanjenje za „manje" nema smisla, pošto svi fajlovi imaju (virtuelni) sadržaj iste veličine.
ii. Prema tome, najpogodnija je indeksna alokacija, po analogiji sa straničenjem i PMT kod virtuelne memorije. Da bi se podržao veći virtuelni prostor fajla, uz povećanje efikasnosti korišćenja fizičkog prostora za jako retko popunjene fajlove, pogodno je indeksiranje u više nivoa.
   Kao što je rečeno, indeksiranje u više nivoa inherentno omogućava efikasno smeštanje retko popunejnih fajlova. Svaki ulaz u indeksu prvog ili višeg nivoa može imati posebnu *null* vrednost koja ukazuje da deo virtuelnog prostora fajla koji adresira taj ulaz ima podrazumevani sadržaj (npr. sve nule), pa se ne mora alocirati prostor za indeks narednog nivoa, odnosno za same podatke, za taj deo virtuelnog prostora, kao kod PMT.
iii. Kada operacija čitanja ili upisa u fajl preslikava adresirani deo virtuelnog prostora fajla, ona će pristupati odgovarajućem ulazu u indeksu prvog ili višeg nivoa. Ukoliko naiđe na ne-*null* vrednosti u indeksima svih nivoa, izvršiće preslikavanje do kraja i pristupiće fizičkom bloku sa podacima na čitanje ili upis, u zavisnosti od operacije.
   Ukoliko operacija čitanja naiđe na *null* vresnost u indeksu, u memorijski bafer će upisati sve podrazumevane vrednosti (0) za taj deo adresiranog prostora. Na taj način se postiže efekat da taj deo virtuelnog prostora fajla „postoji", odnosno može mu se pristupati, ali ima podrazumevane vrednosti pošto u njih ništa još nije upisivano.
   Slično, ukoliko operacija upisa naiđe na *null* vresnost u indeksu, alociraće fizički prostor na uređaju za smeštanje indeksa i podataka iz memorijskog bafera za taj deo adresiranog prostora, ažurirati indekse i upisati te podatke. Na taj način se postiže efekat da taj deo virtuelnog prostora fajla dobija upisane vrednosti koje se mogu kasnije čitati.

20. (*Septembar 2008*)

a. U nekom operativnom sistemu postoji koncept tekućeg direktorijuma procesa, pri čemu proces može promeniti svoj tekući direktorijum odgovarajućim sistemskim pozivom, ali mu se dodeljuje podrazumevani tekući direktorijum prilikom pokretanja. Navesti bar dve različite mogućnosti – koji direktorijum se dodeljuje kao podrazumevani tekući direktorijum.

b. U nekom fajl sistemu direktorijum se smešta na isti način kao i fajl i opisuje se *FCB* strukturom (*File Control Block*). Direktorijum se implementira kao *hash* tabela sa *N* ulaza, pri čemu je u svakom ulazu pokazivač na ulančanu listu vrednosti svih ključeva koji se preslikavaju u taj ulaz u tabeli ili *NULL*, ukoliko takvih nema. Direktorijum preslikava naziv elementa direktorijuma (fajla ili poddirektorijuma) u vrednost koja je identifikator *FCB*-a tog elementa. Date su sledeće deklaracije:

```
#define N ...
typedef long int FCBID;                    // FCB Identifier

struct DirEntry {
  char filename[MAX_FILENAME_LEN];         // Null-terminated string
  FCBID fcb;
  unsigned long next_dir_entry;            // Next dir entry in the linked list
}
```

Sadržaj direktorijuma organizovan je na sledeći način:

- na početku je niz od *N* elemenata tipa `unsigned long` koji predstavljaju ulaze u *hash* tabeli. Svaki ulaz sadrži redni broj sloga tipa `DirEntry` koji je prvi u ulančanoj listi onih koji se preslikavaju u taj ulaz; vrednost 0 označava *NULL* vrednost.
- odmah iza je neograničeni niz slogova tipa `DirEntry`, pri čemu svaki ima redni broj sloga u nizu koji je sledeći u njegovoj ulančanoj listi (to je `next_dir_entry`). Slogovi se broje počev od 1, a vrednost 0 označava *NULL*.

Na raspolaganju je funkcija:

```
int hash_code(char*);
```

koja vraća *hash* vrednost u opsegu 0..*N-1* za zadati ključ-ime fajla, kao i funkcija:

```
void* f_read(FCBID, unsigned long offset, unsigned long size);
```

koja po potrebi učitava i kešira deo fajla sa zadatim *FCB*-om i vraća pokazivač na bafer u koji je učitan taj deo fajla. Deo se identifikuje relativnom pozicijom u odnosu na početak sadržaja fajla (u `sizeof(char)` kao jediničnoj veličini, početak je 0), a funkciji se zadaje i njegova veličina (opet u jedinicama `sizeof(char)`).

Realizovati funkciju:

```
void dir(FCBID dir);
```

koja na standardni izlaz ispisuje sadržaj datog direktorijuma, i to u jednoj liniji samo po jedan naziv elementa tog direktorijuma. Redosled ispisa može da bude proizvoljan, prilagođen implementaciji.

Rešenje:

a. Neke mogućnosti:

- tekući direktorijum procesa-roditelja koji pokreće dati proces; specijalni slučaj ovoga je tekući direktorijum procesa koji izvršava interpreter komandne linije;
- direktorijum u kome se nalazi fajl programa koji izvršava dati proces;
- podrazumevani direktorijum pridružen korisniku koji je „vlasnik" pokrenutog procesa (a koji je i „vlasnik" roditeljskog procesa);
- zadaje se kao obavezni parametar pokretanja procesa.

b. Realizacija tražene funkcije:

```
typedef unsigned long Offset;
typedef unsigned long DirEntryIndx;
#include <stdio.h>
```

```
void dir(FCBID dir) {
  static Offset htbl_size = N*sizeof(DirEntryIndx);
  DirEntryIndx* hash_tbl = (DirEntryIndx*)f_read(dir,0,htbl_size);
  for (int i = 0; i<N; i++)
      for (DirEntryIndx cur_indx = hash_tbl[i]; cur_indx>0;) {
      Offset offset = htbl_size+(cur_indx-1)*sizeof(DirEntry);
        DirEntry* dir_entry = (DirEntry*)f_read(dir,offset,sizeof(DirEntry));
        printf("%s\n",dir_entry->filename);
      cur_indx = dir_entry->next_dir_entry;
    }
}
```

21. (*Oktobar 2008*)

a. U nekom fajl sistemu podržano je zaključavanje fajlova od strane procesa koji ih otvaraju, uz postojanje koncepta *deljenog* (*shared*) i *ekskluzivnog* (*exclusive*) ključa. Proces fajl zaključava implicitno, prilikom njegovog otvaranja, kada u sistemskom pozivu deklariše da li taj fajl otvara samo za čitanje (*read-only*) ili i za izmene (*write-enabled*). Proces koji fajl otvara samo za čitanje zahteva i, ukoliko su zadovoljeni uslovi, dobija deljeni ključ; proces koji fajl otvara i za izmene zahteva i, ukoliko su zadovoljeni uslovi, dobija ekskluzivni ključ. Deljeni ključ se može dobiti ako i samo ako ni jedan drugi proces trenutno nema ekskluzivni ključ; ekskluzivni ključ se može dobiti ako i samo ako ni jedan drugi proces nema bilo kakav ključ nad istim fajlom. Ukoliko proces ne može da dobije traženi ključ, operacija otvaranja fajla završava se greškom.
U ovom sistemu izvršena je sledeća sekvenca sistemskih poziva; pozivi su označeni simbolički, gde `Open` znači otvaranje, a `Close` zatvaranje fajla koji je identifikovan prvim argumentom, dok drugi argument označava vrstu operacije (R-*read-only*, W-*write-enabled*); identifikator ispred dvotačke označava proces koji je izvršio taj poziv:
`P1:Open(F1,W), P2:Open(F2,W), P3:Open(F3,R), P1:Close(F1)`
Navesti koje od sledećih sekvenci koje se, svaka za sebe i nezavisno, posmatraju kao nastavak ove sekvence, izvršavaju bez greške, a koje sa greškom:
   i. `P2:Open(F3,R), P2:Open(F1,W)`
   ii. `P1:Open(F3,R), P1:Open(F2,R)`
   iii. `P2:Close(F2), P2:Open(F1,W), P1:Open(F3,R)`

b. U nekom fajl sistemu podržano je učitavanje blokova fajla unapred radi efikasnijeg sekvencijalnog pristupa, u skladu sa ponašanjem koje proces pokazuje prilikom pristupa datom otvorenom fajlu i „stepena sekvencijalnosti“ u tom pristupu na sledeći način. U strukturi koja predstavlja ulaz u tabeli otvorenih fajlova lokalnoj za dati proces:

```
const int NLB = ...;
typedef unsigned long int BlockNo;
struct PFileDescr {
  ...
  BlockNo lastBlocks[NLB];
  int tailInLastBlocks, numOfLastBlocks;
};
```

polje `lastBlocks` predstavlja kružni bafer, realizovan pomoću niza i indeksa `tailInLastBlocks` (ukazuje na poslednje upisani element), u kome se nalazi `numOfLastBlocks` brojeva poslednje pristupanih logičkih blokova.
Projektuje se modul fajl sistema koji se bavi organizacijom fajla (*file organization module*). Ovaj modul održava memorijski keš na nivou blokova diska, pa se učitani blokovi sa diska uvek nalaze u prostoru odvojenom za keš. Kada proces zahteva pristup logičkom bloku broj *n* datog fajla, ovaj modul učitaće taj i još *k* logički narednih blokova u keš, pri čemu je *k* jednako celobrojnoj (odsečenoj) polovini dužine sekvence logički susednih blokova kojima se poslednje pristupalo. Na primer, ako je sekvenca blokova kojima se poslednje pristupalo (hronološki) ..., 2, 6, 3, 7, *k* je jednako 1/2=0; ako je sekvenca ..., 6, 5, 7, 8, 9, *k* je jednako 3/2=1. Postojeća funkcija:
`Block* readFileBlock (FCB* file, BlockNo logicalFileBlockNo);`
u memorijski keš učitava blok sa logičkim rednim brojem datim drugim argumentom fajla čiji je *FCB* zadat prvim argumentom i vraća pokazivač na taj blok.
Potrebno je realizovati funkciju:
`Block* readFileBlockWithPrefetching (FCB*, PFileDescr*, BlockNo);`
koja obavlja ovo isto, ali sa opisanim dohvatanjem blokova unapred.

Rešenje:

a. *i*) Bez greške. *ii*) Sa greškom. *iii*) Bez greške.

b.

```
Block* readFileBlockWithPrefetching (FCB* fcb, PFileDescr* fd, BlockNo bn){
  if (fcb==0 || fd==0) return 0; // Error!

  // Calculate the sequence length:
  unsigned int seqLen = 1;
```

```
  int i=fd->tailInLastBlocks;
  for (int n=1; n<fd->numOfLastBlocks; n++, i=(i+NLB-1)%NLB, seqLen++)
      if (fd->lastBlocks[i] != fd->lastBlocks[(i+NLB-1)%NLB]+1) break;
  seqLen /= 2;

  // Store the referenced block in lastBlocks:
  fd->tailInLastBlocks = (fd->tailInLastBlocks+1)%NLB;
  fd->lastBlocks[fd->tailInLastBlocks] = bn;
  if (fd->numOfLastBlocks<NLB) fd->numOfLastBlocks++;

  // Fetch the blocks:
  Block* b = readFileBlock (fcb,bn);
  for (unsigned int j=1; j<=seqLen; j++)
      readFileBlock(fcb,bn+j);
  return b;
}
```