1. (10 poena) Ulaz/izlaz

Dat je neki sekvencijalni, blokovski orijentisani ulazni uređaj sa koga se blok znakova veličine `BlockSize` učitava na zadatu adresu funkcijom:

```
void readBlock(char* addr);
```

Od ovog uređaja napraviti apstrakciju sekvencijalnog, znakovno orijentisanog ulaznog uređaja (ulazni tok), odnosno realizovati funkciju koja učitava znak po znak sa tog uređaja:

```
char getchar();
```

Ignorisati sve greške.

Rešenje:

```
char buffer[BlockSize];
int idx = BlockSize;

char getchar() {
    if (idx == BlockSize) {
        readBlock(buffer);
        idx = 0;
    }
    return buffer[idx++];
}
```

1. (10 poena) Ulaz/izlaz

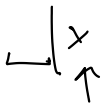
Dat je proceduralni interfejs prema nekom blokovski orijentisanom ulaznom uređaju sa direktnim pristupom:

```
extern const int BlockSize;
extern int BlockIOHandle;
long getSize(BlockIOHandle handle);
int readBlock(BlockIOHandle handle, long blockNo, char* addr);
```

Uređaj se identifikuje „ručkom“ tipa `BlockIOHandle`, a blok je veličine `BlockSize` znakova. Operacija `getSize` vraća ukupnu veličinu sadržaja (podataka) na uređaju (u znakovima), a operacija `readBlock` učitava blok sa zadatim brojem u bafer na zadatoj adresi u memoriji i vraća 0 u slučaju uspeha. Obe operacije vraćaju negativnu vrednost u slučaju greške, uključujući i pokušaj čitanja bloka preko granice veličine sadržaja.

Korišćenjem ovog interfejsa implementirati sledeći objektno orijentisani interfejs prema ovom uređaju, koji od njega čini apstrakciju ulaznog toka, odnosno znakovno orijentisanog ulaznog uređaja sa direktnim pristupom:

```
class IOStream {
public:
  IOStream (BlockIOHandle d);
  int seek (long offset);
  int getChar (char& c);
};
```



Operacija `seek` postavlja poziciju „kurzora“ za čitanje na zadatu poziciju (pomeraj počev od znaka na poziciji 0), a operacije `getChar` čita sledeći znak sa tekuće pozicije kurzora u izlazni argument `c` i pomera kurzor za jedno mesto unapred. U slučaju bilo kakve greške, uključujući i pomeranje kurzora preko veličine sadržaja ili čitanje znaka kada je kurzor stigao do kraja sadržaja, operacije treba da vrate negativnu vrednost, a nulu u slučaju uspeha.

Rešenje:

```
class IOStream {

    BlockIOHandle handle;

    IOStream ( BlockIOHandle d ) : handle(d) {}

    long offset = 0;

    char Block [BlockSize];

    long BlockNum = -1;


    int seek ( long offset) {

        if ( offset < 0 || offset >= getSize(handle)) {

            this->offset = -1;

            return -1;

        }

        this->offset = offset;

    }.
```

```
int getChar(char &c) {
    if (offset < 0 || offset >= getSize(handle)) {
        return -1;
    }

    int BlockNumNext = offset / BlockSize
    if (BlockNum != BlockNumNext) {
        int ret = readBlock(handle, BlockNumNext, block);
        if (ret == 0) {
            BlockNum = BlockNumNext;
        } else {
            return ret;
        }
    }

    c = block[offset - BlockNum * BlockSize];
    offset += 1;
    return 0;
}
```

Zadatak 3. (jun14k3-1)

Dat je neki sekvencijalni, znakovno orijentisani ulazni uređaj (ulazni tok) sa koga se znak učitava sledećom funkcijom:

```
char getchar();
```

Od ovog uređaja napraviti apstrakciju sekvencijalnog, blokovski orijentisanog ulaznog uređaja, sa koga se blok veličine `BlockSize` učitava na zadatu adresu funkcijom:

```
void readBlock(char* addr);
```

Ignorisati sve greške.

```
void readBlock(char *addr) {
    for (int i = 0; i < BlockSize; i++) {
        addr[i] = getchar();
    }
}
```

1. (10 poena) Ulaz/izlaz

U nekom sistemu svaki drajver blokovski orijentisanog uređaja („diska“) registruje sledeću strukturu (tabelu) koja sadrži pokazivače na funkcije koje implementiraju odgovarajuće operacije sa tim uređajem:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number

typedef int (*DiskOperation)(BlkNo block, Byte* buffer);

struct DiskOperationsTable {
  int isValid;
  DiskOperation readBlock, writeBlock;
  DiskOperationsTable () : isValid(0), readBlock(0), writeBlock(0) {}
};
```

Sistem organizuje tabelu registrovanih drajvera za priključene uređaje kao niz ovih struktura, s tim da polje `isValid==1` označava da je dati element niza zauzet (validan, postavljen, odnosno disk je registrovan), a 0 da je ulaz slobodan (disk nije registrovan):

```
const int MaxNumOfDisks; // Maximal number of registered disk devices
DiskOperationsTable disks[MaxNumOfDisks];
```

Sistem preslikava simbolička imena dodeljena priključenim uređajima, u obliku slova abecede, brojevima ulaza u tabeli `disks` (u opsegu od 0 do `MaxNumOfDisks-1`).

a)(5) Realizovati funkcije:

```
int readBlock(int diskNo, BlkNo block, Byte* buffer);
int writeBlock(int diskNo, BlkNo block, Byte* buffer);
```

koje treba da pozovu odgovarajuću implementaciju operacije drajvera (polimorfno, dinamičkim vezivanjem) za zadati uređaj. (Ove funkcije poziva interna kernel nit kada opslužuje zahteve za operacijama sa diskovima, da bi inicijalizovala prenos na odgovarajućem uređaju.)

b)(5) Realizovati funkciju koja registruje operacije drajvera za dati disk:

```
int registerDriver(int diskNo, DiskOperation read, DiskOperation write);
```

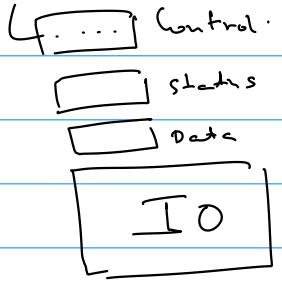
U slučaju greške, sve ovde navedene funkcije vraćaju negativnu vrednost, a u slučaju uspeha vraćaju 0.

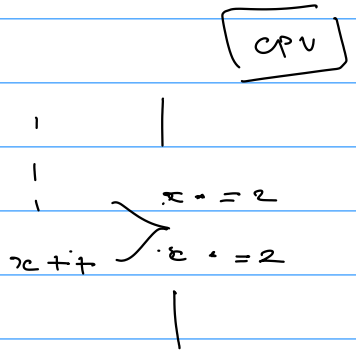
```
int readBlock (int diskNo, BlkNo block, Byte * buffer) {
    if ( diskNo < 0 || diskNo > MaxNumOfDisks ) {
        return -1;
    }
    if ( disks [diskNo] != 1 ) {
        return -1;
    }
    if ( disks [diskNo]. readBlock == 0 ) {
        return -1;
    }
    return ( disks [diskNum]. readBlock ) ( block, buffer );
}
```

```
int registerDriver(int diskNum, DiskOperation read, DiskOperation write){
    if (diskNum < 0 || diskNum >= MaxNumOfDisks){
        return -1;
    }
    if (disks[diskNum] == 1){
        return -1;
    }
    disks[diskNum].valid = 1;
    disks[diskNum].readBlock = read;
    disks[diskNum].writeBlock = write;
}
```

Data, Status, Control.

0xFA...10





A.c

```
static int n;
```

B.c

```
extern int n;
```

```
fcn{
    static int n;
}
```

1. (10 poena)

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazno/izlazna uređaja:

```
typedef volatile 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`).

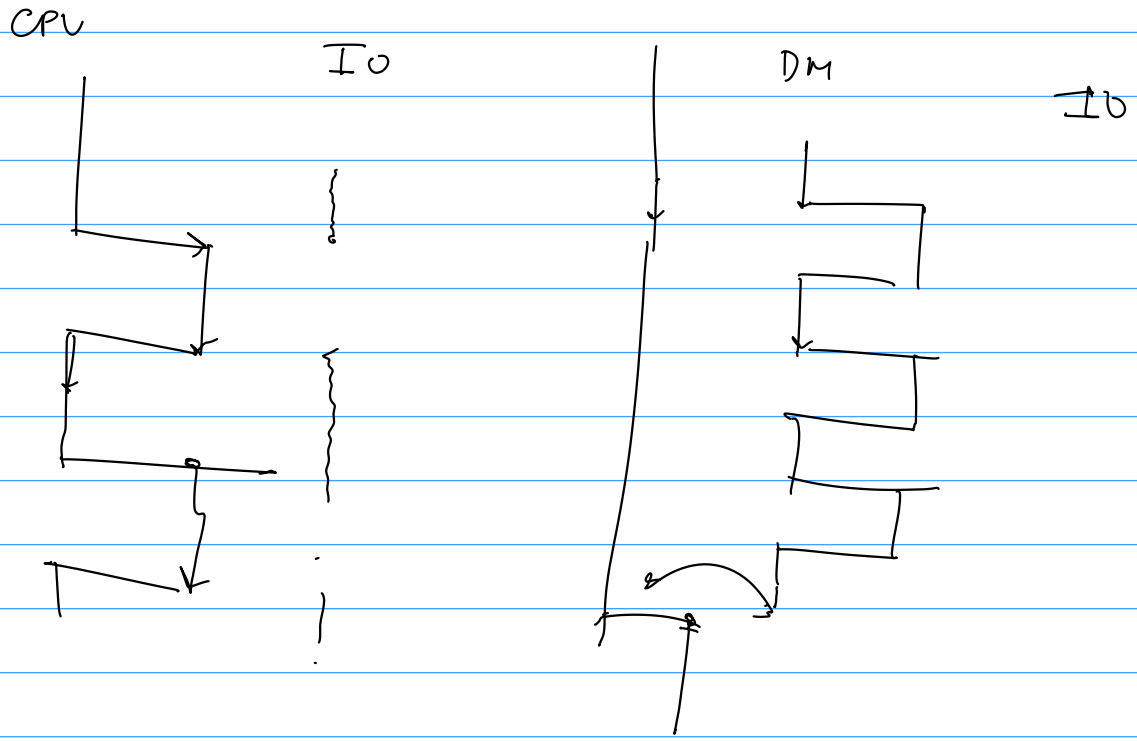
Potrebno je napisati funkciju `transfer` koja najpre vrši ulaz bloka podataka zadate veličine sa prvog uređaja korišćenjem tehnike prozivanja (*polling*), a potom izlaz tog istog učitanog bloka podataka na drugi uređaj korišćenjem prekida, i vraća kontrolu pozivaocu tek kada se oba prenosa završe.

Rešenje:

```
char * buffer = null; int idx = 0; int buffer_size;
bool io2Complete = false;

void transfer (int size) {
    buffer = new char[size]; buffer_size = size;
    *io1Ctrl = 1;
    for (int i = 0; i < size; i++) {
        while ( (*io1Status) & 1 != 1 );
        buffer[i] = *io1Data;
    }
    *io1Ctrl = 0;
    *io2Ctrl = 1;
    while (!io2Complete);
    delete []buffer;
}

interrupt void handleIO2() {
    *io2Data = buffer[idx];
    idx += 1;
    if (idx == buffer_size) {
        io2Complet = true;
        *io2Ctrl = 0;
    }
}
```



Zadatak 6. (jun12k1-1)

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog DMA uređaja:

```
typedef unsigned int REG;
REG* dmaCtrl =...; // control register
REG* dmaStatus =...; // status register
REG* dmaBlkAddr =...; // data block address register
REG* dmaBlkSize =...; // data block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos preko DMA, a u statusnom registru najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je DMA spreman za novi prenos podatka (inicijalno je tako postavljen). Postavljanje bita spremnosti kada DMA završi zadati prenos generiše signal zahteva za prekid procesoru. Zahtevi za izlaznim operacijama na nekom izlaznom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct OutputRequest {
 char* buffer; // Buffer with data (block)
 unsigned int size; // Buffer (blok) size
 void (*callBack)(OutputRequest*); // Call-back function
 OutputRequest* next; // Next in the list
};
```

17

Kada se završi prenos zadat jednim zahtevom, potrebno je pozvati funkciju na koju ukazuje pokazivač `callBack` u tom zahtevu, sa argumentom koji ukazuje na taj zahtev. Ovu funkciju implementira onaj ko je zahtev postavio i služi da mu signalizira da je zahtev obrađen. Obrađeni zahtev ne treba brisati iz liste (to je odgovornost onog ko je zahtev postavio).

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()`, koja obavlja sve prenose zadate zahtevima u listi na čiji prvi zapis ukazuje argument `ioHead`.

```
void transfer (OutputRequest* ioHead);
interrupt void dmaInterrupt ();
```

```
bool dmaCompleted = false;

void transfer (OutputRequest *ioHead) {
    OutRequest *req = ioHead;

    while (req ≠ null) {
        *dmaBlkAddr = req→buffer;
        *dmaBlkSize = req→size;
        :dmaCompleted = false;
        *dmaCtrl = 1;

        while (!dmaCompleted);
        (req→signal)(req);
        req = req→next;
    }
}

interrupt void dmaInterrupt() {
        dmaCompleted = true;
}
```

1. (10 poena)

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog ulaznog uređaja i registru posebnog uređaja – vremenskog brojača:

```
typedef unsigned int REG;
REG* ioCtrl =...;    // Device control register
REG* ioData =...;    // Device data register
REG* timer = ...;    // Timer
```

Učitavanje svakog pojedinačnog podatka sa ovog ulaznog uređaja zahteva se posebnim upisom vredosti 1 u najniži bit upravljačkog registra ovog uređaja. Spremnost ulaznog podatka u registru za podatke uređaj ne signalizira nikakvim signalom, ali je sigurno da je ulazni podatak spreman u registru podataka najkasnije 50 ms nakon zadatog zahteva (upisa u kontrolni registar).

Na magistralu računara vezan je i registar posebnog uređaja, vremenskog brojača. Upisom celobrojne vrednosti *n* u ovaj registar, vremenski brojač počinje merenje vremena od *n* ms i, nakon isteka tog vremena, generiše prekid procesoru.

Na jeziku C napisati kod operacije `transfer()` zajedno sa odgovarajućom prekidnom rutinom za prekid od vremenskog brojača `timerInterrupt()`, koja obavlja učitavanje bloka podataka zadate dužine na zadatu adresu u memoriji sa datog ulaznog uređaja.

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

Rešenje:

```
bool dataReady = false;

void transfer (REG * buffer, unsigned int count) {
    for (int i=0; i<count; i++) {
        dataReady = false;
        *ioCtrl = 1;
        *timer = 50;
        while (!dataRedy);
        buffer[i] = *ioData;
    }
}

interrupt void timerInterrupt() {
    dataReady = true;
}
```

1. (10 poena)

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* dma1Address =...;   // DMA1 block address register
REG* dma1Count =...;     // DMA1 block size register
REG* dma2Ctrl =...;      // DMA2 control register
REG* dma2Status =...;    // DMA2 status register
REG* dma2Address =...;   // DMA2 block address register
REG* dma2Count =...;     // DMA2 block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos jednog bloka preko DMA, a u statusnom registru najniži bit je bit završetka prenosa (*TransferComplete*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Kada DMA konotroler završi zadati prenos, on se automatski zaustavlja (nije ga potrebno zaustavljati upisom u upravljački registar). Završetak prenosa sa bilo kog DMA kontrolera generiše isti zahtev za prekid procesoru (signali završetka operacije sa dva DMA kontrolera vezani su na ulazni zahtev za prekid preko OR logičkog kola).

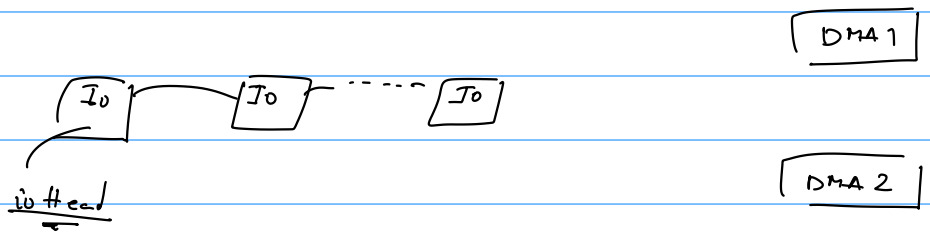
Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko bilo kog od ova DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada kernel u listu stavi novi zahtev, pozvaće operaciju `transfer()` koja treba da pokrene prenos za taj zahtev na bilo kom trenutno slobodnom DMA kontroleru (u slučaju da su oba kontrolera zauzeta ne treba ništa uraditi). Zahtev koji se dodeli nekom od DMA kontrolera na obradu izbacuje se iz liste. Kada se završi prenos zadat jednim zahtevom na jednom DMA kontroleru, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi na tom DMA kontroleru, i potom izbaciti zahtev iz liste. Obratiti pažnju na to da oba DMA kontrolera mogu završiti prenos i generisati prekid istovremeno. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.)

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer ();
interrupt void dmaInterrupt ();
```




```
IORequest *dma1_pending = 0;
IORequest *dma2_pending = 0;
void start_dma1() {
    if (dma1_pending != null) return;
    if (ioHead == null) return;
    dma1_pending = ioHead;
    ioHead = ioHead -> next;
    dma1_pending -> next = null;
```

INT

```
    *dma1Address = dma_pending->buffer;
    *dma1Count = dma_pending->size;
    *dma1Ctrl = 1;
}
void start_dma2() {
    if (dma_pending != null) return;
    if (iotail == null) return;
    dma_pending = iotail;
    iotail = iotail->next;
    dma_pending->next = null;
    *dma2Address = dma2_pending->buffer;
    *dma2Count = dma2_pending->size;
    *dma2Ctrl = 1;
}
```

10

```
interrupt void dma1Interrupt() {
    if (*dma1Stat & 1) {                    Transfer completed.
        if (*dma1Stat & 2) {                Error
            dma_pending->status = -1;
        } else {
            dma_pending->status = 0;
        }
        dma1_pending = null;
        start_dma1();
    }

    if (*dma2Stat & 1) {
        if (*dma2Stat & 2) {
            dma2_pending->status = -1;
        } else {
            dma2_pending->status = 0;
        }
        dma2_pending = null;
        start_dma2();
    }
}
```

```
void transfer() {
        start_dma1();
        start_dma2();
}
```

1. (10 poena) Ulaz/izlaz

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera tastature preko koga stižu znakovi otkucani na tastaturi:

```
typedef volatile unsigned REG;
REG* ioStatus =...;     // status register
char* ioData =...;      // data register
```

Kontroler tastature poseduje interni memorijski modul koji služi za prihvatanje jednog ili više znakova otkucanih na tastaturi (hardverski bafer). Kada se ovom baferu pojave znakovi (jedan ili više), kontroler generiše prekid. Tada se iz registra za podatke mogu čitati pristigli znakovi sukcesivnim operacijama čitanja, jedan po jedan, sve dok je bit spremnosti (*ready*) u razredu 0 statusnog registra postavljen na 1. Naredni nalet pristiglih znakova će ponovo generisati prekid.

Za smeštanje znakova učitanih sa tastature kernel koristi svoj (softverski) ograničeni bafer veličine 256 znakova. U ovaj bafer upisuju se znakovi učitani sa kontrolera tastature sve dok u njemu ima mesta; znakovi koji ne mogu da stanu u bafer se jednostavno odbacuju. Iz ovog bafera znakove uzimaju različiti uporedni tokovi kontrole (procesi) pozivom operacije `getc`; ukoliko u baferu nema znakova, pozivajući tok kontrole treba da se suspenduje dok znakova ne bude. Sinhronizacija se može obavljati semaforima čiji je interfejs isti kao u školskom jezgru. Pretpostaviti sledeće:

- prekidna rutina izvršava se međusobno isključivo sa operacijama na semaforu;
- operacija `signal` na semaforu može se pozivati iz prekidne rutine, jer ona ne radi nikakvu promenu konteksta;
- procesor maskira prekide pri obradi prekida; prekid sa kontrolera tastature se može eksplicitno maskirati pozivom `kbint_mask()`, a demaskirati pozivom `kbint_unmask()`.

Implementirati opisani podsistem: bafer kernela (operaciju `getc` i sve druge potrebne operacije) i prekidnu rutinu kontrolera tastature.

Rešenje:

```
#define BSIZE 256
char kb_buffer[BSIZE];
int head=0, tail=0;                    [head, tail)
Semaphor getc_mutex(1);                . . .)[7,7).
Semaphor buffer_count(0);

char getc() {
        buffer_count.wait();
        getc_mutex.wait()
        kbint_mask();
```

```
    char c = kb_cutter[head];
    head = (head+1) % BSIZE;
    kbint_unmask();
    getc_mutex.signal();
    return c;
}

interrupt kb_interrupt() {
    while (*io_status & 1) {
        kb_cutter[tail] = *io_Data;
        tail = (tail+1) % BSIZE;
        cutter_count.signal();
        if (head == tail) break;
    }
}
```

1. (10 poena) Ulaz/izlaz

U ulazno-izlaznom podsistemu nekog operativnog sistema zahtevi za operacijama sa diskom predstavljaju se instancama strukture `DiskRequest`. Jedan zahtev odnosi se na prenos `blockCount` susednih blokova počev od bloka broj `startBlockNo`, a prenos se vrši sa baferom `buffer` i u smeru koji zadaje polje `dir`.

```
struct DiskRequest {
  uint32  blockCount;
  uint32  startBlockNo;
  uint32* buffer;
  enum Dir : uint32 {in=0, out=1};
  Dir dir;
  ... // Other details irrelevant here
};
```

Za svaki fizički uređaj formira se poseban red `RequestQueue` ovakvih zahteva. Svaki takav red opslužuje po jedna interna nit jezgra koja izvršava funkciju `diskDriver` prikazanu dole. Ova funkcija uzima jedan po jedan zahtev iz reda i svaki zahtev zadaje drajveru uređaja `dd` pozivom njegove operacije `startTransfer`. Potom se blokira na semaforu `semComplete` koji drajver uređaja treba da signalizira kada zahtev bude opslužen, a onda dalje obaveštava podnosioca zahteva (izostavljeni detalji).

```
void diskDriver (RequestQueue* rque, IBlockDeviceDriver* dd) {
  while (true) {
    DiskRequest* req = rque->getRequest();// Blocks until a request arrives
    dd->startTransfer(req);
    semComplete->wait();
    ... // Notify the request initiator
  }
}
```

Interfejs drajvera uređaja `IBlockDeviceDriver` dat je dole. Kada se drajver instalira, jezgro poziva njegovu operaciju `init` za inicijalizaciju samog drajvera, zadajući semafor koji treba signalizirati nakon svakog opsluženog zahteva (`semComplete` pomenut gore).

```
class IBlockDeviceDriver {
public:
  virtual int  init (Semaphore* complete) = 0;
  virtual void startTransfer (DiskRequest*) = 0;
};
```

Tokom inicijalizacije u svojoj operaciji `init` drajver može da poziva sledeće operacije jezgra:

- `uint32* requestDMAChannel()`: zahteva od jezgra zauzimanje jednog slobodnog DMA kontrolera (kanala) za svoje potrebe; jezgro dodeljuje takav kontroler na upotrebu drajveru, ukoliko ga ima, i vraća adresu početka regiona memorijski mapiranih upravljačkih registara dodeljenog DMA kontrolera; ukoliko slobodnog DMA kontrolera nema, vraća *null*;
- `uint32 requestIVTEntry(void (*transferComplete)(void*),void* ptr))`: od jezgra traži zauzeće i inicijalizaciju jednog slobodnog ulaza u IVT; ukoliko ne uspe, vraća negativnu vrednost; ukoliko uspe, inicijalizuje taj ulaz u IVT adresom funkcije zadate prvim parametrom i vraća taj broj ulaza; jezgro obezbeđuje da se prekidna rutina poziva sa parametrom `ptr` (prekidna rutina na koju je usmeren jedan ulaz u IVT poziva se uvek sa istim tim parametrom).

Ukoliko inicijalizacija ne uspe, funkcija `init` treba da vrati negativnu vrednost, 0 za uspeh.
Svaki DMA kontroler obavlja prenos sa samo jednim diskom i ima dva 32-bitna upravljačka registra u svom regionu memorijski mapiranih registara. Prvi registar nalazi se na pomeraju nula od početka regiona, a prilikom inicijalizacije DMA kontrolera u ovaj registar potrebno je upisati broj ulaza u IVT koji će ovaj DMA kontroler koristiti prilikom prekida koji generiše kada završi svaku zadatu operaciju. Drugi registar nalazi se na pomeraju 4 (adresibilna jedinica je bajt, adrese su 32-bitne) od početka regiona, a jedna operacija prenosa DMA kontroleru se zadaje sukcesivnim upisima u ovaj isti upravljački registar na sledeći način (nije potrebno čekati na dozvolu za sledeći upis, ovi upisi mogu da idu odmah jedan iza drugog): najpre je u ovaj registar potrebno upisati adresu bafera, zatim broj prvog bloka na disku, zatim broj susednih blokova koje treba preneti i konačno smer prenosa (0 ili 1). Ovaj poslednji upis ujedno i pokreće prenos. Po završetku prenosa DMA kontroler generiše prekid.

Realizovati u potpunosti klasu `DiskDeviceDriver` koja implementira interfejs `IBlockDeviceDriver`, a koja realizuje drajver za prenos sa disk uređajem korišćenjem DMA kontrolera. Jedan objekat ove klase treba da bude zadužen za jedan fizički disk uređaj.

Rešenje:

```
void wrapper(void *ptr) {
    ((DiskDeviceDriver *)ptr)->interrupt_handler()
}

class DiskDeviceDriver : public BlockDeviceDriver {
    Semaphore *semComplete;
    uint32 *dma_regs;
    int ivt_idx;
    int init(Semaphore *complete) {
        semComplet = complete;
        dma_regs = requestDMAChannel();
        if (dma_regs == null) return -1;
        ivt_idx = requestIVTEntry(wrapper, this);
        if (ivt_idx < 0) return -1;
        return 0;
    }
    void transfer(DiskRequest *req) {
        dma_regs[0] = ivt_idx;
        dma_regs[1] = req->buffer;
        dma_regs[2] = req->startBlkNum;
        dma_regs[3] = req->blockCount;
        dma_regs[4] = req->dir;
    }
    void interrupt_handler() {
        semComplete->signal();
    }
};
```