# I/O Moduli

UDŽBENIK POGLAVLJE 5, STRANICE 70-88.

# Drajveri tastature i ekrana

- Klasa **Display_driver** sadrži drajver ekrana koji upravlja kontrolerom ekrana.
- Kontroler ekrana (objekat **display_controller**) sadrži registar stanja (**display_controller.status_reg**) i registar podataka (**display_controller.data_reg**).
- Prikaz znaka na ekranu je moguć ako **registar stanja** sadrži konstantu **DISPLAY_READY**.
- U tom slučaju se **kod znaka** smešta u **registar podataka**, a u registar stanja se smešta konstanta **DISPLAY_BUSY**.
- Ova konstanta ostaje u registru stanja **dok traje** prikaz znaka.

# Drajveri tastature i ekrana

- Po **prikazu znaka**, kontroler ekrana smešta u registar stanja konstantu **DISPLAY_READY** (podrazumeva se da se ova vrednost nalazi **u registru stanja na početku rada** kontrolera ekrana).
- **Pokušaj niti da prikaže znak**, dok je u registru stanja konstanta **DISPLAY_BUSY**, **zaustavlja aktivnost niti**.
- Nastavak aktivnosti niti usledi **nakon obrade prekida ekrana**, koja objavljuje da je **prikaz prethodnog znaka završen**.

# Drajveri tastature i ekrana

- **Zaustavljanje** i **nastavak** aktivnosti niti omogućuje polje **displayed_char** klase **Display_driver**.
- Opisano ponašanje drajvera ekrana ostvaruju operacije **character_put()** i **interrupt_handler()** klase **Display_driver**.
- **Broj vektora prekida** ekrana određuje konstanta **DISPLAY**.

# Drajver ekrana

```
class Display_driver : public Driver {
        static Event displayed_char;
        static void interrupt_handler();
        Display_driver(const Display_driver&);
        Display_driver& operator=(const Display_driver&);
public :
        Display_driver()
        { start_interrupt_handling(DISPLAY,
          interrupt_handler); };
        void character_put(const char c);
};

Display_driver::Event
Display_driver::displayed_char;
```

# Drajver ekrana

```
void
Display_driver::interrupt_handler()
{
        displayed_char.signal();
}

void
Display_driver::character_put(const char c)
{
        Atomic_region ar;
        if(display_controller.status_reg == DISPLAY_BUSY)
                displayed_char.expect();
        display_controller.data_reg = c;
        display_controller.status_reg = DISPLAY_BUSY;
}

static Display_driver display_driver;
```

# Drajver tastature

- Klasa **Keyboard_driver** sadrži drajver tastature koji upravlja kontrolerom tastature.
- Kontroler tastature (objekat **keyboard_controller**) sadrži **registar podataka** (**keyboard_controller.data_reg**).
- Podrazumeva se da pritisak dirke na tastaturi:
  - dovede do **smeštanja koda** odgovarajućeg znaka u **registar podataka.**
  - Izazove **prekid tastature**.
- Pomenuti kod znaka se preuzima iz **registra podataka** u **obradi prekida tastature** i smešta u **cirkularni bafer**, ako on nije pun.

# Drajver tastature

- Cirkularnom baferu odgovara polje **buffer** klase **Keyboard_driver**.
- Njeno polje **count** određuje **popunjenost** ovog **bafera**.
- Indekse cirkularnog bafera sadrže polja **first_full** i **first_empty** klase **Keyboard_driver**.
- Pokušaj **niti da preuzme znak**, kada je cirkularni bafer **prazan**, **zaustavlja njenu aktivnost**.

# Drajver tastature

- Nastavak aktivnosti niti usledi **nakon obrade prekida tastature**.
- Zaustavljanje i nastavak aktivnosti niti omogućije polje **pressed** klase **Keyboard_driver**.
- Opisano ponašanje drajvera tastature ostvaruju operacije **character_get()** i **interrrupt_handler()** klase **Keyboard_driver**.
- Broj **vektora prekida** tastature određuje konstanta **KEYBOARD**.

# Drajver tastature

```
const unsigned
KEYBOARD_BUFFER_SIZE = 1024;

class Keyboard_driver : public Driver {
        static Event pressed;
        static char buffer[KEYBOARD_BUFFER_SIZE];
        static unsigned count;
        static unsigned first_full;
        static unsigned first_empty;
        static void interrupt_handler();
        Keyboard_driver(const Keyboard_driver&);
        Keyboard_driver& operator=(const Keyboard_driver&);
public:
        Keyboard_driver()
        { start_interrupt_handling(KEYBOARD,
          interrupt_handler); };
        char character_get();
};
```

# Drajver tastature

```
Keyboard_driver::Event
Keyboard_driver::pressed;
char Keyboard_driver::buffer[KEYBOARD_BUFFER_SIZE];
unsigned Keyboard_driver::count = 0;
unsigned Keyboard_driver::first_full = 0;
unsigned Keyboard_driver::first_empty = 0;

void Keyboard_driver::interrupt_handler()
{
        if(count<KEYBOARD_BUFFER_SIZE) {
                buffer[first_empty++] =
                keyboard_controller.data_reg;
                if(first_empty == KEYBOARD_BUFFER_SIZE)
                        first_empty = 0;
                count++;
                pressed.signal();
        }
}
```

# Drajver tastature

```
char Keyboard_driver::character_get()
{
	char c;
	Atomic_region ar;
	if(count==0)
		pressed.expect();
	c = buffer[first_full++];
	if(first_full == KEYBOARD_BUFFER_SIZE)
		first_full = 0;
	count--;
	return c;
}

static Keyboard_driver keyboard_driver;
```

# Znakovni ulaz-izlaz

- Prilikom ulaza-izlaza znakova moguća su štetna preplitanja. Sprečavanje štetnih preplitanja ulaznih i izlaznih operacija podrazumeva da su one **međusobno isključive**.
- Njihova međusobna isključivost se može ostvariti, ako se tastatura, odnosno ekran **zaključa** (zauzme) pre i **otključa** (oslobodi) nakon korišćenja, prilikom svakog izvršavanja odgovarajuće operacije.

# Znakovni ulaz-izlaz

- Neuspešan pokušaj zaključavanja uređaja dovodi do zaustavljanja izvršavanja ovakve operacije, dok zaključavanje ne postane moguće.
- Zaključavanje tastature i ekrana, tokom izvršavanja ulaznih i izlaznih operacija, se zasniva na **korišćenju propusnica**.
- Posebne propusnice reprezentuju **ekran** i **tastaturu**.
- **Zauzimanje propusnice** odgovara **zaključavanju** njenog **uređaja**, a **oslobađanje propusnice** odgovara **otključavanju** dotičnog **uređaja**.
- Time se obezbeđuje međusobna isključivost pojedinačnih ulaznih i izlaznih operacija.

# Znakovni ulaz-izlaz

- Klasa **Terminal_out** omogućuje **znakovni izlaz**, odnosno **prikaz znaknova na ekranu**.
- Ona sadrži operacije koje omogućuju formatiranje prikazivanog podatka (njegovo pretvaranje u niz znakova).
- Oznake **%6d, %6u, %11d i %11u** određuju broj cifara u decimalnom formatu u kome se prikazuju cifre celih označenih (d) i neoznačenih (u) brojeva, a oznaka **% .3e** određuje broj cifara iza decimalne tačke u decimalnom formatu u kome se prikazuju cifre razlomljenih brojeva.
- Za prikaz niza znakova (znakovni izlaz) zadužena je operacija **string_put()** klase **Terminal_out**, koja se brine i o **zaključavanju ekrana**.

# Klasa Terminal_out

```
const char endl[] = "\n";

class Terminal_out : private mutex {
        Terminal_out(const Terminal_out&);
        Terminal_out& operator=(const Terminal_out&);
        void string_put(const char* string);
public:
        Terminal_out() {};
        Terminal_out& operator<<(int number);
        Terminal_out& operator<<(unsigned int number);
        Terminal_out& operator<<(short number);
        Terminal_out& operator<<(unsigned short number);
        Terminal_out& operator<<(long number);
        Terminal_out& operator<<(unsigned long number);
        Terminal_out& operator<<(double number);
        Terminal_out& operator<<(char character);
        Terminal_out& operator<<(const char* string);
        friend class Terminal_in;
};
```

# Klasa Terminal_out

```
static const char* SHORT_FORMAT = "%6d";
static const char* UNSIGNED_SHORT_FORMAT = "%6u";
static const char* INT_FORMAT = "%11d";
static const char* UNSIGNED_FORMAT = "%11u";
static const char* DOUBLE_FORMAT = "% .3e";
static const int SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
static const int INT_SIGNIFICANT_FIGURES_COUNT = 10;
static const int DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;

void Terminal_out::string_put(const char* string)
{
        lock();
        while(*string != '\0')
                display_driver.character_put(*string++);
        unlock();
}
```

# Klasa Terminal_out

```
Terminal_out& Terminal_out::operator<<(int number)
{
	char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, INT_FORMAT, number);
	string_put(string);
	return *this;
}

Terminal_out& Terminal_out::operator<<(unsigned int number)
{
	char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, UNSIGNED_FORMAT, number);
	string_put(string);
	return *this;
}
```

# Klasa Terminal_out

```
Terminal_out& Terminal_out::operator<<(short number)
{
	char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, SHORT_FORMAT, number);
	string_put(string);
	return *this;
}

Terminal_out& Terminal_out::operator<<(unsigned short number)
{
	char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, UNSIGNED_SHORT_FORMAT, number);
	string_put(string);
	return *this;
}
```

# Klasa Terminal_out

```
Terminal_out& Terminal_out::operator<<(long number)
{
	char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, INT_FORMAT, number);
	string_put(string);
	return *this;
}

Terminal_out& Terminal_out::operator<<(unsigned long number)
{
	char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, UNSIGNED_FORMAT, number);
	string_put(string);
	return *this;
}
```

# Klasa Terminal_out

```
Terminal_out& Terminal_out::operator<<(double number)
{
	char string[DOUBLE_SIGNIFICANT_FIGURES_COUNT + 2];
	sprintf(string, DOUBLE_FORMAT, number);
	string_put(string);
	return *this;
}

Terminal_out& Terminal_out::operator<<(char character)
{
	char string[2];
	string[0] = character;
	string[1] = '\0';
	string_put(string);
	return *this;
}
```

# Klasa Terminal_out

```
Terminal_out& Terminal_out::operator<<(const char* string)
{
        string_put(string);
        return *this;
}

Terminal_out cout;
```

# Klasa Terminal_in

- Klasa **Terminal_in** omogućuje preuzimanje znakova (znakovni ulaz).
- Ona obezbeđuje **zaključavanje tastature** dok se izvršava njena operacija **string_get()**, kao i **zaključavanje ekrana**, radi **eha znakova**, preuzetih u ovoj operaciji.
- Operacija **string_get()** poziva operaciju **edit()** klase **Terminal_in** koja je zadužena za **eho znakova na ekranu** i njihovo primitivno editiranje.
- Preostale operacije klase **Terminal_in** koriste operaciju **string_get()** za preuzimanje nizova znakova koji odgovaraju raznim tipovima podataka.

# Klasa Terminal_in

```
const int INPUT_BUFFER_LENGTH = 128;

class Terminal_in : private mutex {
        Terminal_in(const Terminal_in&);
        Terminal_in& operator=(const Terminal_in&);
        unsigned index;
        char c;
        bool pressed_enter;
        char buff[INPUT_BUFFER_LENGTH];
        inline void edit();
        void string_get(unsigned figures_count);
public:
        Terminal_in() {};
        Terminal_in& operator>>(int &number);
        Terminal_in& operator>>(short &number);
        Terminal_in& operator>>(long &number);
        Terminal_in& operator>>(double &number);
        Terminal_in& operator>>(char &character);
};
```

# Klasa Terminal_in

```
static const int SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
static const int INT_SIGNIFICANT_FIGURES_COUNT = 10;
static const int DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;

#define CHAR_ESC (27)
#define CHAR_LF ('\n')
#define CHAR_BS1 ('\b')
#define CHAR_BS2 (127)
```

# Klasa Terminal_in

```
void Terminal_in::edit() //radi eho znakova preuzetih sa tastature na ekran
{
        switch(c) {
                case CHAR_ESC:
                        buff[index++]=c;
                        display_driver.character_put('^'); //esc caret karakter
                        break;
                case CHAR_LF:
                        pressed_enter = true;
                        break;
                case CHAR_BS1:
                case CHAR_BS2:
                        if(index>0) {
                                buff[--index]='\0';
                                display_driver.character_put('\b'); //pomeranje kursora nazad
                                display_driver.character_put(' ');  //stavljanje space-a
                                display_driver.character_put('\b'); //pomeranje kursora nazad
                        }
                        break;
                default:
                        buff[index++]=c;
                        display_driver.character_put(c);
                        break;
        }
        buff[index]='\0';
}
```

# Klasa Terminal_in

```
void Terminal_in::string_get(unsigned figures_count)
{
        lock();                                                    //zakljucaj tastaturu
        index = 0;
        pressed_enter = false;
        c = keyboard_driver.character_get();
        cout.lock();                                  //zakljucaj terminal
        edit();                                                    //ispisi prvi karakter
        while((index < (figures_count - 1)) &&
                        !pressed_enter) {  //ispisuj do entera tj. do precizn.
                c = keyboard_driver.character_get();
                edit();
        }
        cout.unlock();                  //otkljucaj terminal
        unlock();                                                  //otkljucaj tastaturu
}
```

# Klasa Terminal_in

```
Terminal_in& Terminal_in::operator>>(int& number)
{
        string_get(INT_SIGNIFICANT_FIGURES_COUNT);
        number = (int)strtol(buff, 0, 10);
        return *this;
}

Terminal_in& Terminal_in::operator>>(short& number)
{
        string_get(SHORT_SIGNIFICANT_FIGURES_COUNT);
        number = (int)strtol(buff, 0, 10);
        return *this;
}

Terminal_in& Terminal_in::operator>>(long& number)
{
        string_get(INT_SIGNIFICANT_FIGURES_COUNT);
        number = (int)strtol(buff, 0, 10);
        return *this;
}
```

# Klasa Terminal_in

```
Terminal_in& Terminal_in::operator>>(double& number)
{
        string_get(DOUBLE_SIGNIFICANT_FIGURES_COUNT);
        number = strtod(buff, 0);
        return *this;
}

Terminal_in& Terminal_in::operator>>(char& character)
{
        string_get(1);
        character = buff[0];
        return *this;
}

Terminal_in cin;
```

# Klasa Disk_driver

- Klasa **Disk_driver** sadrži drajver diska koji upravlja DMA **kontrolerom diska**.
- DMA kontroler diska (objekat **disk_controller**) sadrži:
  - registar bloka (disk_controller.block_reg)
  - registar bafera (disk_controller.buffer_reg)
  - registar smera prenosa (disk_controller.operation_reg)
  - registar stanja (disk_controller.status_reg).

# Klasa Disk_driver

- Podrazumeva se da nit pokreće prenos bloka tako što:

1)u registar bloka smesti broj prenošenog bloka

2)u registar bafera smesti adresu bafera koji učestvuje u prenosu

3)u registar smera prenosa smesti konstantu DISK_READ ili DISK_WRITE

4)u registar stanja konstantu DISK_STARTED

- Nakon toga aktivnost niti se **zaustavi** dok traje prenos bloka.
- Kraj prenosa bloka objavi **prekid diska**.
- Zaustavljanje i nastavak aktivnostiniti omogućuje polje **ready** klase Disk_driver. Opisano ponašanje drajvera diska ostvaruju operacije **block_transfer()** i **interrupt_handler()** klase Disk_driver.
- Broj **vektora prekida** diska određuje konstanta **DISK**.

# Klasa Disk_driver

```
class Disk_driver : public Driver {
        static Event ready;
        static void interrupt_handler();
        Disk_driver(const Disk_driver &);
        Disk_driver& operator=(const Disk_driver &);
public:
        Disk_driver(void) { start_interrupt_handling(DISK,
                                                     interrupt_handler); }
        inline int block_transfer(char* buffer, unsigned block,
                                                     Disk_operations operation);
};

Disk_driver::Event Disk_driver::ready;

void Disk_driver::interrupt_handler()
{
        ready.signal();
}
```

# Klasa Disk_driver

```
int Disk_driver::block_transfer(char* buffer,
                                unsigned block, Disk_operations operation)
{
        int r = -1;
        if(block < DISK_BLOCKS) {
                Atomic_region ar;
                disk_controller.block_reg = block;
                disk_controller.buffer_reg = buffer;
                disk_controller.operation_reg = operation;
                disk_controller.status_reg = DISK_STARTED;
                ready.expect();
                r = 0;
        }
        return r;
}

static Disk_driver disk_driver;
```

# Blokovski ulaz-izlaz

- Klasa Disk opisuje rukovanje virtuelnim diskom.
- Ova klasa definiše operacije **`block_get()`** i **`block_put()`** koje omogućuju **preuzimanje bloka** sa diska i **smeštanje bloka** na disk.
- Prvi parametar ovih operacija pokazuje na niz od 512 bajta radne memorije koji učestvuje u prebacivanju bloka, a drugi parametar određuje broj bloka (u rasponu od 0 do 999).
- Obe operacije su blokirajuće.

# Blokovski ulaz-izlaz

- Pošto brzina pomeranja glave diska ograničava ukupnu brzinu diska, važno je **skratiti put** koji glava diska prelazi.
- Optimizacija kretanja glave diska je moguća kada se skupi, više zahteva za čitanjem ili pisanjem.
- Optimizacija se svodi na opsluživanje zahteva u **redosledu** staza na koje se oni odnose, a **ne** u **hronološkom** redosledu pojave zahteva.
- Na taj način se izbegava da glava diska **osciluje** između vanjskih i unutrašnjih staza diska.

# Blokovski ulaz-izlaz

- Da bi optimizacija kretanja glave diska bila moguća, neophodno je uticati na **redosled zahteva** za čitanjem ili pisanjem blokova.
- To postavlja specifične zahteve na implementaciju klase **`condition_variable`** (proširenje klase).
- Podrazumeva se da odabrani redosled deskriptora niti u listi uslova nastaje na osnovu **privezaka** koji se dodeljuju svakom deskriptoru prilikom njegovog uvezivanja u ovu listu.
- **Privesci** imaju oblik **neoznačenih celih brojeva**, a njihovo dodeljivanje deskriptoru omogućuje dodatni, drugi parametar operacije wait().

# Blokovski ulaz-izlaz

- Uticaj na redosled deskriptora niti u listi uslova omogućuju operacije **`first()`**, **`next()`** i **`last()`** klase **`condition_variable`**.
- Operacija **`first()`** omogućuje pozicioniranje **pre prvog** deskriptora u listi uslova.
- Operacija **`next()`** omogućuje pozicioniranje **pre narednog** ili **iza poslednjeg** deskriptora u listi uslova.
- Operacija **`last()`** omogućuje pozicioniranje **iza poslednjeg** deskriptora u listi uslova.
- Podrazumeva se da operacija **`notify_one()`** klase **`condition_variable`** uvek izvezuje deskriptor **sa početka** liste uslova.

# Blokovski ulaz-izlaz

- Optimizaciju kretanja glave diska omogućuje operacija **optimize()** klase Disk.
- U situaciji kada je disk **slobodan** (kada polje state klase Disk sadrži konstantu **FREE**), poziv operacije **optimize()** dovodi do poziva **blokirajuće** operacije **disk_driver.block_transfer()**.
- U toku njenog izvršavanja disk je **zaposlen** (polje state sadrži konstantu **BUSY**), a aktivnost pozivajuće niti je **zaustavljena**.
- Ako u ovoj situaciji **više niti**, **jedna za drugom**, pozove operaciju **optimize()**, njihova aktivnost se zaustavlja, a njihovi deskriptori se uvezuju u **jednu od dve liste** uslova, koje odgovaraju polju **q** klase **Disk**.

# Blokovski ulaz-izlaz

- Polje **index** ove klase indeksira ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između **trenutnog položaja glave diska i njegovog oboda** ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između **centra rotacije ploče diska i trenutnog položaja njegove glave**.
- Podatak o **trenutnom položaju glave diska** (odnosno, o bloku koji se upravo čita) sadrži polje **boundary** klase Disk.
- Polje index može imati vrednost 0 ili 1.
- U pomenutim listama uslova deskriptori su poređani u rastućem redosledu brojeva blokova koje niti čitaju (što je u skladu sa optimizacijom kretanja glave diska).
- Pomenuti brojevi blokova predstavljaju **priveske** deskriptora iz listi uslova.

# Blokovski ulaz-izlaz

```
const int DISK_ERROR = -1;

class Disk {
        mutex mx;
        enum Optimized_disk_state { FREE, BUSY };
        Optimized_disk_state state;
        unsigned boundary;
        condition_variable q[2];
        int index;
        Disk(const Disk&);
        Disk& operator=(const Disk&);
        int optimize(char* buffer, unsigned block,
                                    Disk_operations operation);
public:
        Disk() : state(FREE), boundary(0), index(0) {};
        int block_get(char* buffer, unsigned block);
        int block_put(char* buffer, unsigned block);
};
```

# Blokovski ulaz-izlaz

```
int
Disk::optimize(char* buffer, unsigned block, Disk_operations operation)
{
        unsigned tag;
        int i;
        int status;
        {
                unique_lock<mutex> lock(mx);
                if(state == BUSY) {
                        i = index;
                        if(block < boundary)
                                i = ((i == 0) ? (1) : (0));
                        if(q[i].first(&tag))
                                do {
                                        if(block < tag)
                                                break;
                                } while(q[i].next(&tag));
                        q[i].wait(lock, block);
                }
                state = BUSY;
                boundary = block;
        }
```

# Blokovski ulaz-izlaz

```
        status = disk_driver.block_transfer(buffer,
                                block, operation);
        {
                unique_lock<mutex> lock(mx);
                state = FREE;
                if(!q[index].first())
                        index = ((i == 0) ? (1) : (0));
                q[index].notify_one();
        }
        return status;
}

int Disk::block_get(char* buffer, unsigned block)
{
        int status;
        status = optimize(buffer, block, DISK_READ);
        return status;
}
```

# Blokovski ulaz-izlaz

```
int Disk::block_put(char* buffer, unsigned block)
{
        int status;
        status = optimize(buffer, block, DISK_WRITE);
        return status;
}

Disk disk;
```