Operativni Sistemi - Simulator operativnog sistema 1

Veljko Petrović

Jul, 2025

Uvod

Šta simuliramo

- Ovo je simulator operativnog sistema koji radi pod određenim znatnim ograničenjima
- Simulira (isključivo) jednoprocesorni operativni sistem i to kroz konkurentnu biblioteku
- Šta znači da simulira kroz konkurentnu biblioteku? Da je interfejs prema ovome onaj isti C++ interfejs ka nitima, sa tom razlikom da umesto da koristimo POSIX niti, mi sami realizujemo prebacivanje između niti i to u obliku simulatora operativnog sistema.
- Radi (isključivo) na 32-bitnoj x86 arhitekturi koju ste koristili prošli semestar.

Šta simuliramo

- Ništa od ovih stvari nisu 100% aktuelne, ali su odličan smanjeni skup funkcionalnosti koji nam može pomoći da razumemo osnovne principe.
- Gde je prigodno tokom kursa, mi skačemo u kod Linux kernela da vidimo apsolutno aktuelnu i ozbiljnu verziju ovoga
- Termin koji povremeno koristimo za simulator je i CppTss pošto implementiramo podskup za niti iz ranog 2011 standarda za C++ zato što je relativno jednostavan za razumevanje.

Šta ne simuliramo

- Ništa od ovoga nije pravi operativni sistem u klasičnom smislu

- Ako vas zanima kako se operativni sistem startuje od nule na pravom hardveru: sačekajte. Pričaćemo i o tome.
- Za sada nas zanima da povežemo gradivo ovde sa gradivom iz AR na nečemu što je dovoljno pojednostavjeno da celo može da stane na predavanja.

Osnove simulatora

Atomski regioni

- Stvaranje atomskih regiona omogućuje klasa `Atomic_region`.
- Njen konstruktor onemogućuje prekide, a destruktor vraća prekide u stanje koje je prethodilo akciji konstruktora.

Atomski regioni

```
{
    Atomic_region ar;
    //kod
}
```

Atomski regioni

- Upotrebu atomskog regiona ilustruje primer rukovanja pozicijom kursora u kome nisu moguća štetna preplitanja niti i obrada prekida, jer telo operacije `get()` klase `Position` obrazuje atomski region.
- Pošto se operacija `set()` poziva samo iz obrada prekida, njeno telo po definiciji obrazuje atomski region (jer su prekidi onemogućeni u toku obrade prekida, barem kod nas), pa je tako osigurana međusobna isključivost operacija klase `Position`.

Primer sa kursorom

```
1 class Position {
2     int x, y;
3 public:
4     Position();
5     void set(int new_x, int new_y);
6     void get(int* current_x, int* current_y);
7 };
```

Primer sa kursorom

```
8  Position::Position(){
9      x = 0;
10     y = 0;
11 }
12
13 void Position::set(int new_x, int new_y){
14     //ne mora atomski ako su prekidi prekida zabranjeni
15     x = new_x;
16     y = new_y;
17 }
```

Primer sa kursorom

```
18 void Position::get(int* current_x, int* current_y){
19     Atomic_region ar;
20     *current_x = x;
21     *current_y = y;
22 }
```

Klasa `Driver`

- Pisanje drajvera olakšava klasa `Driver`.
- Nju nasleđuju klase koje opisuju ponašanje drajvera.
- Drajvere karakteriše saradnja obrađivača prekida i pozadinskih niti.
- U okviru klase, koja opisuje ponašanje drajvera, obrađivač prekida se predstavlja u obliku funkcije bez povratne vrednosti i bez parametara.
- Ova funkcija mora biti `static`, da bi se mogla koristiti njena adresa.
- Takva moraju biti i sva polja klase kojima ona pristupa.

Klasa `Driver`

- Operacija `start_interrupt_handling()` klase `Driver` omogućuje smeštanje adrese obrađivača prekida u tabelu prekida.
- Prvi argument poziva ove operacije predstavlja broj vektora prekida, a drugi adresu obrađivača prekida.
- Saradnja obrađivača prekida i pozadinskih niti podrazumeva da pozadinske niti čekaju dešavanje spoljnih događaja, a da obrađivači prekida objavljuju dešavanje spoljnih događaja.

Neki primeri iz stvarne prakse – Linux

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

Neki primeri iz stvarne prakse – Linux

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

Neki primeri iz stvarne prakse – Linux

```
static irqreturn_t intr_handler(int irq, void *dev)
```

`Event` i `Driver`

- Klasa `Driver` ima kao povezanu klasu `Event`
- `signal` metoda
  - Poziva se iz atomskog regiona
  - Nit prelazi u čekanje
  - Prekljućivanje na spremnu nit
- `expect` metoda
  - Poziva se iz obrađivača prekida
  - Nit koja čeka najduže prelazi u stanje spremna
  - Poziva se na kraju obrade prekida samo jednom

Drajver za rukovanje vremenom

- Ovaj drajver registruje proticanje vremena u računaru brojanjem otkucaja sata.
- Otkucaji, čiji period zavisi od takta procesora, nisu uvek podesni za određivanje vremena u danu, predstavljenog brojem

sati, minuta i sekundi, jer sekunda ne može uvek da se izrazi celim brojem perioda ovakvih otkucaja.
- Zato je zgodno uvesti dodatni sat, čiji otkucaji (prekidi) imaju period od tačno jedne sekunde.
- Ova komponenta se, tradicionalno, zove 'Real Time Clock' u računarima i vi je imate na matičnoj ploči, tipično zakčenoj za SRAM modul i CR2032 bateriju.

Drajver za rukovanje vremenom

- Rukovanje ovim satom, odnosno rukovanje vremenom opisuje klasa `Timer_driver`.
- Njena tri celobrojna polja: `hour`, `minute` i `second` sadrže broj proteklih sati, minuta i sekundi.
- Početni sadržaj ovih polja određuje konstruktor klase `Timer_driver`.

Drajver za rukovanje vremenom

- Pored toga on u element tabele prekida, koga indeksira konstanta `TIMER` (broj vektora prekida dodatnog sata), smesti adresu njene operacije `interrupt_handler()`, koja je zadužena za periodičnu izmenu sadržaja polja `hour`, `minute` i `second`, sa periodom od jedne sekunde.

Drajver za rukovanje vremenom

- Klasa `Timer_driver` sadrži i operacije `set()` i `get()` za zadavanje i preuzimanje sadržaja njenih polja.
- To su osetljive operacije, čije preplitanje sa obradom prekida sata je štetno.
- Na primer, ako se obrada prekida sata desi nakon preuzimanja sadržaja polja hour, a pre preuzimanja sadržaja polja minute, i ako je, uz to, broj minuta pre periodične izmene bio na granici od 59, tada preuzeto vreme kasni iza stvarnog za 60 minuta.
- Da bi se ovakva štetna preplitanja sprečila, preuzimanja i zadavanja sadržaja njenih polja moraju da budu u atomskim regionima.

Klasa `Timer_driver`

```
1 class Timer_driver : public Driver {
2     static int hour;
3     static int minute;
4     static int second;
```

```
5       static void interrupt_handler();
6   public:
7       Timer_driver()
8       { start_interrupt_handling(TIMER,
9         interrupt_handler); };
10      void set(const int h, const int m,
11                  const int s);
12      void get(int* h, int* m, int* s) const;
13  };
```

Klasa `Timer_driver`

```
14  int Timer_driver::hour = 0;
15  int Timer_driver::minute = 0;
16  int Timer_driver::second = 0;
```

Klasa `Timer_driver`

```
17  void Timer_driver::interrupt_handler() {
18      if(second < 59)
19          second++;
20      else {
21          second = 0;
22          if(minute < 59) minute++;
23          else {
24              minute = 0;
25              if(hour < 23) hour++;
26              else hour = 0;
27          }
28      }
29  }
```

Klasa `Timer_driver`

```
30  void Timer_driver::set(const int h, const int m, const int s){
31      Atomic_region ar;
32      hour = h;
33      minute = m;
34      second = s;
35  }
```

Klasa `Timer_driver`

```
36  void Timer_driver::get(int* h, int* m, int* s) const{
37      Atomic_region ar;
```

```
38      *h = hour;
39      *m = minute;
40      *s = second;
41  }
```

Klasa `Sleep_driver`

- Upotrebu klase `Driver` ilustruje i primer drajvera koji omogućuje uspavljivanje jedne niti dok ne protekne zadani broj otkucaja sata.
- Ovo uspavljivanje može da se prikaže kao očekivanje dešavanja zadanog broja otkucaja sata.
- To opisuje klasa `Sleep_driver`.
- Njena operacija `simple_sleep_for()` omogućuje jednoj niti da zaustavi svoju aktivnost dok se ne desi zadani broj otkucaja sata.
- Zadatak obrađivača prekida (operacije `interrupt_handler()`) je da odbroji zadani broj otkucaja sata i da nakon toga signalizira da je moguć nastavak aktivnosti uspavane niti.

Klasa Sleep_driver

```
1   class Sleep_driver: public Driver {
2       static unsigned long countdown;
3       static Event alarm;
4       static void interrupt_handler();
5   public:
6       Sleep_driver(
7          { start_interrupt_handling(TIMER,
8                interrupt_handler); };
9       void simple_sleep_for(
10                unsigned long duration);
11  };
```

Klasa Sleep_driver

```
12  unsigned long Sleep_driver::countdown = 0;
13
14  void Sleep_driver::interrupt_handler(){
15      if((countdown > 0) && ((--countdown) == 0))
16          alarm.signal();
17  }
```

Klasa Sleep_driver

```
18 void Sleep_driver::simple_sleep_for(unsigned long duration){
19     Atomic_region ar;
20     if(duration > 0) {
21         countdown = duration;
22         alarm.expect();
23     };
24 }
```

Ulazno/izlazni moduli

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()
```

Drajver Ekrana

```
        { 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();
}
```

Drajver Ekrana

```
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;
```

Drajver Ekrana

```
}

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

```
1  const unsigned
2  KEYBOARD_BUFFER_SIZE = 1024;
3
4  class Keyboard_driver : public Driver {
5      static Event pressed;
6      static char buffer[KEYBOARD_BUFFER_SIZE];
7      static unsigned count;
```

Drajver Tastature

```
8      static unsigned first_full;
9      static unsigned first_empty;
```

```
10      static void interrupt_handler();
11      Keyboard_driver(const Keyboard_driver&);
12      Keyboard_driver& operator=(const Keyboard_driver&);
13  public:
14      Keyboard_driver()
```

Drajver Tastature

```
15      { start_interrupt_handling(KEYBOARD,
16        interrupt_handler); };
17      char character_get();
18  };
19
20  Keyboard_driver::Event
21  Keyboard_driver::pressed;
```

Drajver Tastature

```
22  char Keyboard_driver::buffer[KEYBOARD_BUFFER_SIZE];
23  unsigned Keyboard_driver::count = 0;
24  unsigned Keyboard_driver::first_full = 0;
25  unsigned Keyboard_driver::first_empty = 0;
26
27  void Keyboard_driver::interrupt_handler(){
28      if(count<KEYBOARD_BUFFER_SIZE) {
```

Drajver Tastature

```
29          buffer[first_empty++] =
30          keyboard_controller.data_reg;
31          if(first_empty == KEYBOARD_BUFFER_SIZE)
32              first_empty = 0;
33          count++;
34          pressed.signal();
35      }
36  }
```

Drajver Tastature

```
37  char Keyboard_driver::character_get(){
38      char c;
39      Atomic_region ar;
40      if(count==0)
41          pressed.expect();
```

Drajver Tastature

```
42      c = buffer[first_full++];
43      if(first_full == KEYBOARD_BUFFER_SIZE)
44          first_full = 0;
45      count--;
46      return c;
47  }
48  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`

```
1  const char endl[] = "\n";
2
3  class Terminal_out : private mutex {
4      Terminal_out(const Terminal_out&);
5      Terminal_out& operator=(const Terminal_out&);
6      void string_put(const char* string);
7  public:
```

Klasa `Terminal_out`

```
8      Terminal_out() {};
9      Terminal_out& operator<<(int number);
10     Terminal_out& operator<<(unsigned int number);
11     Terminal_out& operator<<(short number);
12     Terminal_out& operator<<(unsigned short number);
13     Terminal_out& operator<<(long number);
14     Terminal_out& operator<<(unsigned long number);
```

Klasa `Terminal_out`

```
15     Terminal_out& operator<<(double number);
16     Terminal_out& operator<<(char character);
17     Terminal_out& operator<<(const char* string);
18     friend class Terminal_in;
19 };
20
21 static const char* SHORT_FORMAT = "%6d";
```

Klasa `Terminal_out`

```
22 static const char* UNSIGNED_SHORT_FORMAT = "%6u";
23 static const char* INT_FORMAT = "%11d";
24 static const char* UNSIGNED_FORMAT = "%11u";
25 static const char* DOUBLE_FORMAT = "% .3e";
26 static const int SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
27 static const int INT_SIGNIFICANT_FIGURES_COUNT = 10;
28 static const int DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;
```

Klasa `Terminal_out`

```
29
30  void Terminal_out::string_put(const char* string)
31  {
32      lock();
33      while(*string != '\0')
34          display_driver.character_put(*string++);
35      unlock();
```

Klasa `Terminal_out`

```
36  }
37
38  Terminal_out& Terminal_out::operator<<(int number)
39  {
40      char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
41      sprintf(string, INT_FORMAT, number);
42      string_put(string);
```

Klasa `Terminal_out`

```
43      return *this;
44  }
45
46  Terminal_out& Terminal_out::operator<<(unsigned int number)
47  {
48      char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
49      sprintf(string, UNSIGNED_FORMAT, number);
```

Klasa `Terminal_out`

```
50      string_put(string);
51      return *this;
52  }
53
54  Terminal_out& Terminal_out::operator<<(short number)
55  {
56      char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
```

Klasa `Terminal_out`

```
57      sprintf(string, SHORT_FORMAT, number);
58      string_put(string);
59      return *this;
```

```
60  }
61
62  Terminal_out& Terminal_out::operator<<(unsigned short number)
63  {
```

Klasa `Terminal_out`

```
64      char string[SHORT_SIGNIFICANT_FIGURES_COUNT + 2];
65      sprintf(string, UNSIGNED_SHORT_FORMAT, number);
66      string_put(string);
67      return *this;
68  }
69
70  Terminal_out& Terminal_out::operator<<(long number)
```

Klasa `Terminal_out`

```
71  {
72      char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
73      sprintf(string, INT_FORMAT, number);
74      string_put(string);
75      return *this;
76  }
```

Klasa `Terminal_out`

```
77  Terminal_out& Terminal_out::operator<<(unsigned long number)
78  {
79      char string[INT_SIGNIFICANT_FIGURES_COUNT + 2];
80      sprintf(string, UNSIGNED_FORMAT, number);
81      string_put(string);
82      return *this;
83  }
```

Klasa `Terminal_out`

```
84  Terminal_out& Terminal_out::operator<<(double number)
85  {
86      char string[DOUBLE_SIGNIFICANT_FIGURES_COUNT + 2];
87      sprintf(string, DOUBLE_FORMAT, number);
88      string_put(string);
89      return *this;
90  }
```

Klasa `Terminal_out`

```
91  Terminal_out& Terminal_out::operator<<(char character)
92  {
93      char string[2];
94      string[0] = character;
95      string[1] = '\0';
```

Klasa `Terminal_out`

```
96      string_put(string);
97      return *this;
98  }
99
100 Terminal_out& Terminal_out::operator<<(const char* string)
101 {
102     string_put(string);
```

Klasa `Terminal_out`

```
103     return *this;
104 }
105
106 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`

```
1   const int INPUT_BUFFER_LENGTH = 128;
2
3   class Terminal_in : private mutex {
4       Terminal_in(const Terminal_in&);
```

```
5       Terminal_in& operator=(const Terminal_in&);
6       unsigned index;
7       char c;
```

Klasa `Terminal_in`

```
8       bool pressed_enter;
9       char buff[INPUT_BUFFER_LENGTH];
10      inline void edit();
11      void string_get(unsigned figures_count);
12  public:
13      Terminal_in() {};
14      Terminal_in& operator>>(int &number);
```

Klasa `Terminal_in`

```
15      Terminal_in& operator>>(short &number);
16      Terminal_in& operator>>(long &number);
17      Terminal_in& operator>>(double &number);
18      Terminal_in& operator>>(char &character);
19  };
20
21  static const int SHORT_SIGNIFICANT_FIGURES_COUNT = 5;
```

Klasa `Terminal_in`

```
22  static const int INT_SIGNIFICANT_FIGURES_COUNT = 10;
23  static const int DOUBLE_SIGNIFICANT_FIGURES_COUNT = 10;
24
25  #define CHAR_ESC (27)
26  #define CHAR_LF ('\n')
27  #define CHAR_BS1 ('\b')
28  #define CHAR_BS2 (127)
```

Klasa `Terminal_in`

```
29
30  void Terminal_in::edit() //radi eho znakova preuzetih sa tastature na ekran
31  {
32      switch(c) {
33          case CHAR_ESC:
34              buff[index++]=c;
35              display_driver.character_put('^'); //esc caret karakter
```

Klasa `Terminal_in`

```
36              break;
37          case CHAR_LF:
38              pressed_enter = true;
39              break;
40          case CHAR_BS1:
41          case CHAR_BS2:
42              if(index>0) {
```

Klasa `Terminal_in`

```
43                  buff[--index]='\0';
44                  display_driver.character_put('\b'); //pomeranje kursora nazad
45                  display_driver.character_put(' ');  //stavljanje space-a
46                  display_driver.character_put('\b'); //pomeranje kursora nazad
47              }
48              break;
49          default:
```

Klasa `Terminal_in`

```
50              buff[index++]=c;
51              display_driver.character_put(c);
52              break;
53      }
54      buff[index]='\0';
55  }
```

Klasa `Terminal_in`

```
56  void Terminal_in::string_get(unsigned figures_count)
57  {
58      lock();                         //zakljucaj tastaturu
59      index = 0;
60      pressed_enter = false;
61      c = keyboard_driver.character_get();
62      cout.lock();                    //zakljucaj terminal
```

Klasa `Terminal_in`

```
63      edit();                         //ispisi prvi karakter
64      while((index < (figures_count - 1)) &&
65              !pressed_enter) {  //ispisuj do entera tj. do preciz.
66          c = keyboard_driver.character_get();
```

```
67          edit();
68      }
69      cout.unlock();              //otkljucaj terminal
```

Klasa `Terminal_in`

```
70      unlock();                    //otkljucaj tastaturu
71  }
72
73  Terminal_in& Terminal_in::operator>>(int& number)
74  {
75      string_get(INT_SIGNIFICANT_FIGURES_COUNT);
76      number = (int)strtol(buff, 0, 10);
```

Klasa `Terminal_in`

```
77      return *this;
78  }
79
80  Terminal_in& Terminal_in::operator>>(short& number)
81  {
82      string_get(SHORT_SIGNIFICANT_FIGURES_COUNT);
83      number = (int)strtol(buff, 0, 10);
```

Klasa `Terminal_in`

```
84      return *this;
85  }
86
87  Terminal_in& Terminal_in::operator>>(long& number)
88  {
89      string_get(INT_SIGNIFICANT_FIGURES_COUNT);
90      number = (int)strtol(buff, 0, 10);
```

Klasa `Terminal_in`

```
91      return *this;
92  }
93
94  Terminal_in& Terminal_in::operator>>(double& number)
95  {
96      string_get(DOUBLE_SIGNIFICANT_FIGURES_COUNT);
97      number = strtod(buff, 0);
```

Klasa `Terminal_in`

```
98      return *this;
99  }
100
101 Terminal_in& Terminal_in::operator>>(char& character)
102 {
103     string_get(1);
104     character = buff[0];
```

Klasa `Terminal_in`

```
105     return *this;
106 }
107
108 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:
  - u registar bloka smesti broj prenošenog bloka
  - u registar bafera smesti adresu bafera koji učestvuje u prenosu
  - u registar smera prenosa smesti konstantu `DISK_READ` ili `DISK_WRITE`
  - u registar stanja konstantu `DISK_STARTED`

Klasa `Disk_driver`

- 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`

```
1  class Disk_driver : public Driver {
2      static Event ready;
3      static void interrupt_handler();
4      Disk_driver(const Disk_driver &);
5      Disk_driver& operator=(const Disk_driver &);
6  public:
7      Disk_driver(void) { start_interrupt_handling(DISK,
```

Klasa `Disk_driver`

```
8                              interrupt_handler); }
9      inline int block_transfer(char* buffer, unsigned block,
10                             Disk_operations operation);
11 };
12 Disk_driver::Event Disk_driver::ready;
```

Klasa `Disk_driver`

```
13 void Disk_driver::interrupt_handler()
14 {
15     ready.signal();
16 }
17
18 int Disk_driver::block_transfer(char* buffer,
19                 unsigned block, Disk_operations operation)
```

Klasa `Disk_driver`

```
20 {
21     int r = -1;
22     if(block < DISK_BLOCKS) {
23         Atomic_region ar;
24         disk_controller.block_reg = block;
25         disk_controller.buffer_reg = buffer;
26         disk_controller.operation_reg = operation;
```

Klasa `Disk_driver`

```
27         disk_controller.status_reg = DISK_STARTED;
28         ready.expect();
29         r = 0;
```

```
30      }
31      return r;
32  }
```

Klasa `Disk_driver`

```
33  static Disk_driver disk_driver;
```

Blokovski ulaz-izlaz

- Klasa `Disk` opisuje rukovanje virtuelnim (magneto-rotacionim) 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.

Klasa `Disk`

```
const int DISK_ERROR = -1;

class Disk {
    mutex mx;
    enum Optimized_disk_state { FREE, BUSY };
    Optimized_disk_state state;
    unsigned boundary;
```

Klasa `Disk`

```
    condition_variable q[2];
    int index;
    Disk(const Disk&);
    Disk& operator=(const Disk&);
    int optimize(char* buffer, unsigned block,
                    Disk_operations operation);
public:
```

Klasa `Disk`

```
    Disk() : state(FREE), boundary(0), index(0) {};
    int block_get(char* buffer, unsigned block);
    int block_put(char* buffer, unsigned block);
};

int
Disk::optimize(char* buffer, unsigned block, Disk_operations operation)
```

Klasa `Disk`

```
{
    unsigned tag;
    int i;
    int status;
    {
        unique_lock<mutex> lock(mx);
        if(state == BUSY) {
```

Klasa `Disk`

```
            i = index;
            if(block < boundary)
                i = ((i == 0) ? (1) : (0));
```

```
32              if(q[i].first(&tag))
33                  do {
34                      if(block < tag)
35                          break;
```

Klasa `Disk`

```
36                  } while(q[i].next(&tag));
37              q[i].wait(lock, block);
38          }
39          state = BUSY;
40          boundary = block;
41      }
42      status = disk_driver.block_transfer(buffer,
```

Klasa `Disk`

```
43                  block, operation);
44      {
45          unique_lock<mutex> lock(mx);
46          state = FREE;
47          if(!q[index].first())
48              index = ((i == 0) ? (1) : (0));
49          q[index].notify_one();
```

Klasa `Disk`

```
50      }
51      return status;
52  }
53
54  int Disk::block_get(char* buffer, unsigned block)
55  {
56      int status;
```

Klasa `Disk`

```
57      status = optimize(buffer, block, DISK_READ);
58      return status;
59  }
60
61  int Disk::block_put(char* buffer, unsigned block)
62  {
63      int status;
```

Klasa `Disk`

```
63      status = optimize(buffer, block, DISK_WRITE);
64      return status;
65  }
66
67  Disk disk;
```

Pitanja

Pitanja

- Do čega dovodi pokušaj niti da preuzme znak kada je cirkularni bafer drajvera tastature prazan?
- Šta se desi kada se napuni cirkularni bafer drajvera tastature?
- Šta se desi u obradi prekida diska?