Operativni Sistemi - Simulator operativnog sistema 2

Veljko Petrović

Jul, 2025

Virtuelna Mašina Simulatora

Upozorenje

- Termin 'virtuelna mašina' se koristi u više značenja u računarskim naukama
- Značenje kako ga koristimo ovde nije nijedno od onih na koje ste vi navikli
- Znači, u ovom slučaju, više *simulator* hardvera.

VM

- Razvojna verzija konkurentne biblioteke CppTss sadrži virtuelnu mašinu koja za potrebe ostatka ove biblioteke:
  - emulira kontrolere tastature, ekrana i diska
  - emulira mehanizam prekida
  - podržava okončanje izvršavanja konkurentnog programa
  - podržava rukovanje pojedinim bitima memorijskih lokacija
  - podržava rukovanje numeričkim koprocesorom (Numeric Processor Extension – NPX)
  - podržava rukovanje stekom

Neophodna zaglavlja

```
1 #include <stdlib.h>
2 #include <strings.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <termios.h>
6 #include <signal.h>
7 #include <sys/time.h>
```

```
#include <string.h>
#include <stdio.h>
```

Emulacija mehanizma prekida

- Emulacija mehanizma prekida se zasniva na:
  - uvođenju (emulirane) tabele prekida
  - uvođenju (emuliranog) bita prekida
  - obezbeđenju nezavisnosti (asinhronosti) između prekida i izvršavanja konkurentnog programa

Emulacija mehanizma prekida

- Potrebe CppTss biblioteke su uzrokovale da (emulirana) tabela prekida sadrži pet elemenata.
- Prvi od njih je namenjen za vektor obrađivača hardverskog izuzetka (`FLOATING POINT EXCEPTION`), a drugi je rezervisan za vektor obrađivača prekida sata.
- Preostala tri su predviđena za vektore obrađivača prekida tastature, ekrana i diska.

Emulacija mehanizma prekida

```
static const unsigned
INTERRUPT_TABLE_VECTOR_COUNT = 5;
enum Vector_numbers { FP_EXCEPTION, TIMER, KEYBOARD, DISPLAY, DISK };
```

Emulacija mehanizma prekida

- Svrha (emuliranog) bita prekida je da označi da li je ili nije omogućena obrada prekida. Ovakav bit zaista postoji u `FLAGS` registru, recimo, x86 procesora i kontroliše se sa `sti` i `cli` instrukcijama.
- Emulacija bita prekida je ostvarena pomoću promenljive `interrupts_enabled` i funkcija `ad__disable_interrupt()` i `ad__restore_interrupts()`.
- Promenljiva `interrupts_enabled` sadrži (emulirani) bit prekida. Njena vrednost određuje da li su (emulirani) prekidi omogućeni (konstanta true) ili ne.

Emulacija mehanizma prekida

- Prva od njih, izmenom (emuliranog) bita prekida, onemogućuje (emulirane) prekide, a druga poništava efekte prve, vraćajući

(emulirani) bit prekida u prethodno stanje.

- Kada operacija `ad__restore_interrupts()` utvrdi da je došlo do odlaganja obrade (emuliranih) prekida (`Interrupt::pending`), ona pokreće prethodno odloženu emulaciju kontrolera pozivom operacije `controller_emulator()` klase `Interrupt`.
- Nakon toga se registruje da nema više odloženih obrada (emuliranih) prekida.

Emulacija prekida

```
1  bool interrupts_enabled = true;
2  inline static bool ad__disable_interrupts(){
3      bool saved_interrupts_enabled = interrupts_enabled;
4      interrupts_enabled = false;
5      return saved_interrupts_enabled;
6  }
```

Emulacija prekida

```
7  inline void ad__restore_interrupts(bool saved_interrupts_enabled){
8      if(saved_interrupts_enabled && interrupt.pending) {
9          interrupt.controller_emulator();
10         interrupt.pending = false;
11     }
12     interrupts_enabled = saved_interrupts_enabled;
13 }
```

Emulacija mehanizma prekida

- Nezavisnost (emuliranih) prekida od izvršavanja konkurentnog programa se ostvaruje pomoću mehanizma signala Linux-a.
- Ovaj mehanizam omogućuje da se na pojavu signala reaguje izvršavanjem odabrane funkcije (user level exception handling).
- Ova funkcija se naziva obrađivač signala.
- Signali su unapred definisani, a svaki od njih je pridružen jednoj vrsti događaja, kao što je isticanje zadanog vremenskog intervala (SIGVTALRM) ili pojava hardverskog izuzetka (SIGFPE).

Emulacija mehanizma prekida

- Kada se takav događaj desi mehanizam signala zaustavi izvršavanje programa (u toku koga se desio dotični događaj), radi pokretanja izvršavanja odgovarajućeg obrađivača signala.
- Nakon obrade dotičnog signala moguć je nastavak zaustavljenog izvršavanja programa.

- Obrađivač signala je funkcija koja opisuje korisničku reakciju na pojavu odabranog signala.
- Funkcija postaje obrađivač signala kada se poveže sa odgovarajućim signalom.

Emulacija mehanizma prekida

- Pojava signala SIGVTALRM nije zavisna od izvršavanja konkurentnog programa.
- Zadatak obrađivača signala SIGVTALRM je da pozove operaciju kontrolera i tako izazove obradu nekog od (emuliranih) prekida, a zadatak obrađivača signala SIGFPE je da izazove obradu izuzetka.
- Za razliku od obrade izuzetaka, koja se uvek obavlja bez odlaganja, obrada (emuliranih) prekida se obavezno odlaže ako su (emulirani) prekidi onemogućeni.
- Do obrade prethodno onemogućenog (emuliranog) prekida dolazi tek nakon omogućenja (emuliranih) prekida.

Emulacija mehanizma prekida

- U slučaju konkurentnog programa, pojava signala podstakne mehanizam signala da zaustavi zatečenu aktivnost niti i pokrene odgovarajućeg obrađivača signala.
- Ako u sklopu obrade (emuliranog) prekida, koju izazove ovaj obrađivač signala, dođe do prekljućivanja na drugu nit, započeta obrada signala će biti završena tek nakon ponovnog prekljućivanja na prethodno zaustavljenu nit.
- Da bi se u međuvremenu mogli obraditi novi signali, neophodno je da se razna izvršavanja obrađivača signala mogu preklapati.
- Za takve obrađivače signala se kaže da su višeulazni (reentrant).

Emulacija mehanizma prekida

- Klasa `Linux_signals` opisuje reakciju na Linux signale.
- Njen konstruktor koristi njeno polje sa i sistemski poziv `sigaction()` da saopšti da njena operacija `signal_handler()` ima ulogu obrađivača signala `SIGVTALRM` i `SIGFPE`.
- Ovaj konstruktor koristi konstantu `SA_NODEFER` (koju upisuje u polje `sa.sa_flags`) da saopšti da nema odlaganja obrada signala (da je obrađivač signala višeulazni).
- Destruktor klase `Linux_signals` poništava akcije njenog konstruktora.
- Obrađivač signala `Linux_signals::signal_handler()` u slučaju signala SIGVTALRM pozove emulaciju prekida (`Interrupt::emulation()`),

a u slučaju signala SIGFPE pozove obrađivača izuzetka (`Interrupt::handler()`) koji opslužuje hardverski izuzetak.

Emulacija prekida

```
1  class Linux_signals {
2    struct sigaction sa; // struktura za definisanje signala
3    static void signal_handler(int signal);
4    Linux_signals(const Linux_signals &);
5    Linux_signals &operator=(const Linux_signals &);
6
7  public:
```

Emulacija prekida

```
8    Linux_signals();
9    ~Linux_signals();
10 };
11
12 void Linux_signals::signal_handler(int signal) // override signala
13 {
14   switch (signal) {
```

Emulacija prekida

```
15   case SIGVTALRM:
16     interrupt.emulation();
17     break;
18   case SIGFPE:
19     interrupt.handler(FP_EXCEPTION);
20     break;
21   }
```

Emulacija prekida

```
22 }
23
24 Linux_signals::Linux_signals() {
25   sa.sa_handler = signal_handler;
26   sigemptyset(&(sa.sa_mask));   // Ne blokiramo nijedan signal
27   sa.sa_flags = SA_NODEFER;     // obrada bez odlaganja
28   sigaction(SIGVTALRM, &sa, 0); // promena signala za
```

Emulacija prekida

```
29    sigaction(SIGFPE, &sa, 0);     // SIGVTALRM i SIGFPE
30  }
31
32  Linux_signals::~Linux_signals() {
33    sa.sa_handler = SIG_DFL; // vracanje na default
34    sigaction(SIGVTALRM, &sa, 0);
35    sigaction(SIGFPE, &sa, 0);
```

Emulacija prekida

```
36  }
37
38  Linux_signals linux_signals;
39
40  const int LINUX_TIMER_INTERVAL = 10;
41
42  class Linux_timer {
```

Emulacija prekida

```
43    struct itimerval itimer;
44    Linux_timer(const Linux_timer &);
45    Linux_timer &operator=(const Linux_timer &);
46
47  public:
48    Linux_timer();
49    ~Linux_timer();
```

Emulacija prekida

```
50  };
51
52  Linux_timer::Linux_timer() // usec vreme se meri u mikrosekundama
53  {
54    itimer.it_interval.tv_sec = 0; // tekuća vrednost
55    itimer.it_interval.tv_usec = LINUX_TIMER_INTERVAL * 1000;
56    itimer.it_value.tv_sec = 0; // vrednost kojom se resetuje
```

Emulacija prekida

```
57    itimer.it_value.tv_usec = LINUX_TIMER_INTERVAL * 1000;
58    setitimer(ITIMER_VIRTUAL, &itimer, 0);
59  }
```

```
60
61 Linux_timer::~Linux_timer() {
62   itimer.it_value.tv_sec = 0;
63   itimer.it_value.tv_usec = 0;
```

Emulacija prekida

```
64   setitimer(ITIMER_VIRTUAL, &itimer, 0); // user mode timer
65 }
66
67 static Linux_timer linux_timer;
```

Emulacija mehanizma prekida

- Klasa `Interrupt`:
  - uvodi (emuliranu) tabelu prekida (sadržanu u nizu vector)
  - omogućuje registrovanje odlaganja obrade (emuliranog) prekida (polje pending)
  - reguliše redosled pozivanja obrađivača pojedinih (emuliranih) prekida (polja controller_turn i timer_turn)

Emulacija mehanizma prekida

- (Emulirana) tabela prekida se inicijalizuje tako da njeni elementi sadrže vektor podrazumevajućeg obrađivača prekida: default_interrupt_handler().
- Operacija handler() klase Interrupt posreduje u pozivu obrađivača (emuliranog) prekida.
- Operacija emulation() registruje odlaganje obrade prekida ili pokreće emulaciju kontrolera pozivom operacije controller_emulator().
- U svakoj parnoj emulaciji kontrolera poziva se obrađivač (emuliranog) prekida sata (sa brojem vektora TIMER).
- U svakoj neparnoj emulaciji kontrolera obavlja se, u kružnom redosladu, emulacija samo jednog od kontrolera (display_controller.output(), keyboard_controller.input() ili disk_controller.transfer()).
- Operacija ad_set_vector() omogućuje izmenu vektora prekida.

Emulacija prekida

```
1 class Interrupt {
2   static void (*vector[INTERRUPT_TABLE_VECTOR_COUNT])();
3   bool pending;
4   int controller_turn;
5   bool timer_turn;
```

```
6    Interrupt(const Interrupt &);
7    Interrupt &operator=(const Interrupt &);
```

Emulacija prekida

```
8  public:
9    Interrupt();
10   inline void handler(unsigned index);
11   inline void emulation();
12   inline void controller_emulator();
13   friend inline void ad__restore_interrupts(bool new_interrupt_status);
```

Emulacija prekida

```
14   friend inline void ad__set_vector(int index, void (*handler)());
15 };
16
17 void default_interrupt_handler() {}
18
19 void (*Interrupt::vector[INTERRUPT_TABLE_VECTOR_COUNT])() = {
20     default_interrupt_handler};
```

Emulacija prekida

```
21 Interrupt::Interrupt() : pending(false), controller_turn(0), timer_turn(true) {}
22
23 void Interrupt::handler(unsigned index) { (vector[index])(); }
24
25 void Interrupt::emulation() {
26   if (!interrupts_enabled)
```

Emulacija prekida

```
27     interrupt.pending = true;
28   else {
29     interrupts_enabled = false;
30     controller_emulator();
31     interrupts_enabled = true;
32   }
33 }
```

Emulacija prekida

```
34 void Interrupt::controller_emulator() {
35   bool interrupt_emulated;
```

```
36    int counter = 0;
37    if (timer_turn) {
38      timer_turn = false;
39      handler(TIMER);
```

Emulacija prekida

```
40    } else {
41      timer_turn = true;
42      do {
43        switch (controller_turn) {
44        case 0:
45          interrupt_emulated = display_controller.output();
46          controller_turn = 1;
```

Emulacija prekida

```
47          break;
48        case 1:
49          interrupt_emulated = keyboard_controller.input();
50          controller_turn = 2;
51          break;
52        case 2:
53          interrupt_emulated = disk_controller.transfer();
```

Emulacija prekida

```
54          controller_turn = 0;
55          break;
56        }
57      } while (!interrupt_emulated && ++counter < 3);
58    }
59  }
```

Emulacija prekida

```
60  inline void ad__set_vector(int index, void (*handler)()) {
61    Interrupt::vector[index] = handler;
62  }
63
64  Interrupt interrupt;
```

Emulacija kontrolera tastature

- Klasa `Keyboard_controller` opisuje kontroler tastature.

- Njeno polje `data_reg` predstavlja registar podataka kontrolera, a njena operacija `input()` opisuje ponašanje kontrolera tastature.
- Ako postoji znak za preuzimanje, on se preuzima u okviru operacije `input()` posredstvom odgovarajućeg Linux sistemskog poziva.
- Ujedno se poziva obrađivač prekida tastature posredstvom operacije `handler()`:
- `interrupt.handler(KEYBOARD)`
- klase `Interrupt` koja emulira tabelu prekida.
- Operacija `input()` se periodično poziva u toku emulacije kontrolera.

Emulacija kontrolera tastature

```
1  class Keyboard_controller {
2    char data_reg;
3    bool input();
4    Keyboard_controller(const Keyboard_controller &);
5    Keyboard_controller &operator=(const Keyboard_controller &);
6
7  public:
```

Emulacija kontrolera tastature

```
8    Keyboard_controller(){};
9    friend class Interrupt;
10   friend class Keyboard_driver;
11 };
12
13 bool Keyboard_controller::input() {
14   bool interrupt_emulated = false;
```

Emulacija kontrolera tastature

```
15   unsigned read_count;
16   char c;
17   read_count = read(STDIN_FILENO, &c, 1); // sys call
18   if (read_count > 0) {
19     data_reg = c;
20     interrupt.handler(KEYBOARD);
21     interrupt_emulated = true;
```

Emulacija kontrolera tastature

```
22    }
23    return interrupt_emulated;
24  }
25
26  Keyboard_controller keyboard_controller;
```

Emulacija kontrolera ekrana

- Klasa `Display_controller` opisuje kontroler ekrana.
- Njena polja `data_reg` i `status_reg` predstavljaju registre podataka i stanja kontrolera, a njena operacija `output()` opisuje ponašanje kontrolera ekrana.
- Ako postoji znak za prikazivanje, on se prikazuje u okviru operacije `output()` posredstvom odgovarajućeg Linux sistemskog poziva.
- Ujedno se poziva obrađivač prekida ekrana posredstvom operacije `handler()` `interrupt.handler(DISPLAY)` klase `Interrupt` koja emulira tabelu prekida.
- Operacija `output()` se periodično poziva u toku emulacije kontrolera.

Emulacija kontrolera ekrana

```
1  enum Display_status { DISPLAY_READY, DISPLAY_BUSY };
2
3  class Display_controller {
4    char data_reg;
5    Display_status status_reg;
6    bool output();
7    Display_controller(const Display_controller &);
```

Emulacija kontrolera ekrana

```
8    Display_controller &operator=(const Display_controller &);
9
10 public:
11   Display_controller() : status_reg(DISPLAY_READY){};
12   friend class Interrupt;
13   friend class Display_driver;
14 };
```

Emulacija kontrolera ekrana

```
15 bool Display_controller::output() {
16   bool interrupt_emulated = false;
17   if (status_reg == DISPLAY_BUSY) {
18     write(STDOUT_FILENO, &data_reg, 1); // sys call
19     status_reg = DISPLAY_READY;
20     interrupt.handler(DISPLAY);
```

Emulacija kontrolera ekrana

```
21     interrupt_emulated = true;
22   }
23   return interrupt_emulated;
24 }
25
26 Display_controller display_controller;
```

Emulacija kontrolera tastature i ekrana

- Prethodne dve klase koriste tastaturu i ekran Linux terminala.
- Za potrebe emulacije neophodno je isključiti eho (echo) Linux terminala, prevesti Linux terminal u režim rada bez linijskog editiranja (raw mode) i obezbediti da poziv čitanja znaka sa terminala bude neblokirajući.

Emulacija kontrolera tastature i ekrana

- Sve prethodno obezbeđuje klasa `Linux_terminal` koji koristi polje `ots` ove klase da sačuva zatečeni režim rada Linux terminala, a `ts` polje da zada njegov novi režim rada.
- Prethodno zatečeni režim rada Linux terminala ponovo uspostavlja destruktor klase `Linux_terminal`.
- Ovaj destruktor se poziva na kraju aktivnosti procesa i radi provere da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome one pripadaju i radi obaveštenja o prevremenom završetku ovakvih niti.

Rukovanje terminalom

```
1 class Linux_terminal {
2   struct termios ts;
3   struct termios ots;
4   Linux_terminal(const Linux_terminal &);
```

```
5     Linux_terminal &operator=(const Linux_terminal &);
6
7   public:
```

Rukovanje terminalom

```
8     Linux_terminal();
9     ~Linux_terminal();
10  };
11
12  Linux_terminal::Linux_terminal() {
13    tcgetattr(STDIN_FILENO, &ts); // preuzmi parametre terminala
14    ots = ts;
```

Rukovanje terminalom

```
15    ts.c_lflag &= ~ECHO;   // zabrana eha na ekran
16    ts.c_lflag &= ~ICANON; // flag nekanonskog moda
17    ts.c_cc[VTIME] = 0;    // timeout u decisekunda za nekanonski read
18    ts.c_cc[VMIN] = 0;     // minimalni broj karaktera za nekanonski read
19    tcsetattr(STDIN_FILENO, TCSANOW, &ts); // postavi parametre term
20  }
```

Rukovanje terminalom

```
22  Linux_terminal::~Linux_terminal() {
23    if (undetached_threads())
24                write(STDOUT_FILENO, &"\nERROR: DESTROYING UNDETACHED
25               THREADS!\n", 40);
26      else
27          write(STDOUT_FILENO, &"\n", 1);
28      tcsetattr(STDIN_FILENO, TCSANOW, &ots); //povrati parametre
```

Rukovanje terminalom

```
29  }
30
31  static Linux_terminal linux_terminal;
```

Emulacija kontrolera diska

- Klasa `Disk_controller` opisuje ponašanje kontrolera diska.
- Njena polja `operation_reg`, `buffer_reg`, `block_reg` i `status_reg` odgovaraju registrima smera prenosa, bafera, bloka i stanja

kontrolera, a njena operacija `transfer()` opisuje ponašanje kontrolera diska.
- Konstruktor klase `Disk_controller` koristi sistemski poziv `calloc()` radi zauzimanja radne memorije, u kojoj se čuvaju blokovi emuliranog diska.

Emulacija kontrolera diska

- Inercija diska se emulira brojanjem poziva operacije `transfer()`.
- Kada broj poziva ove operacije postane jednak procenjenom broju vremenskih jedinica, potrebnom za prenos bloka, tada se obavi prenos bloka u okviru ove operacije i ujedno se pozove obrađivač prekida diska posredstvom operacije `handler()` `interrupt.handler(DISK)` klase Interrupt koja emulira tabelu prekida.
- Operacija transfer() se periodično poziva u toku emulacije kontrolera.

Emulacija diska

```
1  enum Disk_operations { DISK_READ, DISK_WRITE };
2  enum Disk_status { DISK_STARTED, DISK_ACTIVE, DISK_STOPPED };
3  const unsigned BLOCK_SIZE = 512;
4  const unsigned DISK_BLOCKS = 1000;
5  typedef char Disk_block[BLOCK_SIZE];
6
7  class Disk_controller {
```

Emulacija diska

```
8    Disk_block *disk_space;
9    unsigned accessed_last;
10   unsigned transfer_time;
11   Disk_operations operation_reg;
12   char *buffer_reg;
13   unsigned block_reg;
14   Disk_status status_reg;
```

Emulacija diska

```
15   bool transfer();
16   Disk_controller(const Disk_controller &);
17   Disk_controller &operator=(const Disk_controller &);
18
19 public:
```

```
20    Disk_controller();
21    ~Disk_controller();
```

Emulacija diska

```
22    friend class Interrupt;
23    friend class Disk_driver;
24  };
25
26  Disk_controller::Disk_controller()
27      : accessed_last(0), transfer_time(0), status_reg(DISK_STOPPED) {
28    disk_space = (Disk_block *)calloc(DISK_BLOCKS, sizeof(Disk_block));
```

Emulacija diska

```
29  }
30
31  Disk_controller::~Disk_controller() { free(disk_space); }
32
33  const int SECTORS_PER_TRACK = 10;
34  const int TRANSFER_TIME_AND_ROTATIONAL_DELAY = 2;
```

Emulacija diska

```
36  bool Disk_controller::transfer() {
37    bool interrupt_emulated = false;
38    Disk_block *block_pointer;
39    int block_distance;
40    if (status_reg == DISK_STARTED) { // deo simulacije inercije rotacije diska
41      status_reg = DISK_ACTIVE;
42      block_distance = accessed_last - block_reg;
```

Emulacija diska

```
43      if (block_distance < 0)
44        block_distance = -block_distance;
45      transfer_time = TRANSFER_TIME_AND_ROTATIONAL_DELAY;
46      transfer_time += block_distance / SECTORS_PER_TRACK; // vreme rotacije
47      accessed_last = block_reg;
48    }
49    if ((status_reg == DISK_ACTIVE) && (--transfer_time == 0)) {
```

Emulacija diska

```
50      block_pointer = disk_space + block_reg;
51      if (operation_reg == DISK_WRITE)
52        bcopy(buffer_reg, block_pointer, BLOCK_SIZE); // kopiraj bajte
53      else
54        bcopy(block_pointer, buffer_reg, BLOCK_SIZE); // kopiraj bajte
55      status_reg = DISK_STOPPED;
56      interrupt.handler(DISK);
```

Emulacija diska

```
57      interrupt_emulated = true;
58    }
59    return interrupt_emulated;
60  }
61
62  Disk_controller disk_controller;
```

Okončanje izvršavanja konkurentnog programa

- Izvršavanje konkurentnog programa se okončava sistemskim pozivom `exit()`. To omogućuje funkcija `ad__report_and_finish()`

Okončanje izvršavanja konkurentnog programa

```
1  inline void ad__report_and_finish(const char* message_string){
2      int length = 0;
3      while(message_string[length] != 0)
4          length++;
5      write(STDERR_FILENO, message_string, length);
6      write(STDERR_FILENO, "\n", 1);
7      exit(1);
8  }
```

ASM direktiva

- C/C++ standard podrazumeva postojanje ASM direktive koja omogućava da se u C/C++ kod umetne asemblerski kod date platforme.
- Standard ne definiše u detalje kako tačno ova funkcija treba da radi.
- Mi radimo sa GCC kompajlerom, te stoga koristimo sintaksu koju uvodi GCC.

Vrste GCC ASM direktive

- U okviru GCC-a, postoje dve varijante ASM direktive:
  - Osnovna (basic) i
  - Proširena (extended)
- Osnovna služi da se samo navedu ASM komande, jedna za drugom, i ništa preko toga.
- Proširena, omogućava integraciju između ASM koda i C koda kroz ulazno/izlazne parametre.

Osnovna ASM direktiva

```
asm asm-qualifiers ( AssemblerInstructions )
```

Osnovna ASM direktiva

- Gde `asm-qualifiers` može biti:
  - `volatile` — kaže kompajleru da ne optimizuje, podrazumevano za osnovni ASM kod
  - `inline` — kaže kompajleru da minimizuje procenjenu veličinu ASM koda

AssemblerInstructions

- Sastoje se od više linija od kojih je svaka u navodima i svaka se završava sa `\n\t`

```
asm ("movl %eax, %ebx\n\t"
     "movl $56, %esi\n\t"
     "movl %ecx, $label(%edx,%ebx,$4)\n\t"
     "movb %ah, (%ebx)");
```

ANSI standardan C

- Ponekad, ako se želi držati strogog ANSI standarda, uzimajući u obzir nešto neobičnih osobina ASM direktive u GCC-u, može se mesto ‘asm’ koristiti ‘**asm**.’ Nama to može trebati ako želimo da koristimo -std opciju zbog C++11 opcija
- GCC tretira obe stvari apsolutno identično.

Proširen ASM

- Osnovni ASM nema jednostavan način da radi sa C kodom. Recimo, ako želite da u njemu modifikujete neku promenljivu iz C koda, to je nemoguće.

- Stoga postoji proširen ASM koji to dozvoljava i čija se upotreba preporučuje.
- Ograničenje u upotrebi ovakve ASM direktive jeste da se to mora činiti iz nekakve funkcije/metode.

Sintaksa proširenog ASM-a bez naredbe skoka

```
asm asm-qualifiers (
        AssemblerTemplate
        : OutputOperands
        : InputOperands
        : Clobbers
    )
```

Sintaksa proširenog ASM-a sa naredbama skoka

```
asm asm-qualifiers (
        AssemblerTemplate
        :
        : InputOperands
        : Clobbers
        : GotoLabels
    )
```

`asm-qualifiers`

- `volatile` — isključuje stanovite optmizacije što je neophodno ako naš kod ima pobočne efekte, tj. ako radi nešto preko manipulacije ulaznih u izlazne vrednosti.
- `inline` — kao ranije
- `goto` — informišemo kompajler da asm kod može skočiti na neku od labela koje smo specificirali u 'GotoLabels' parametru. Ako je to ikako moguće, ovo valja izbeći.

AssemblerTemplate

- Ponaša se kao instrukcije kod osnovne ASM direktive uz dve ključne razlike:
  - Kada označavamo registre, umesto da kažemo `%eax`, recimo, moramo reći `%%eax`.
  - Možemo da mesto parametara asemblerskih instrukcija da stavimo `%n` gde `n` nekakav broj i asm će umesto te oznake umetnuti vrednost koju pod tim brojem prosleđujemo iz C koda kroz specifikacije koje se nalaze u `OutputOperands` i `InputOperands`

OutputOperands/InputOperands

- U oba slučaja su zarezima razdvojene liste koje smeju biti prazne.
- U oba slučaja, takođe, elementi liste su oblika `"ograničenje" (izraz)`
- Gde je ograničenje string sa raznim simbolima koji definišu kako koristimo dati operand, dok je izraz nekakav C izraz (gotovo uvek promenljiva) koju umećemo u naš ASM kod.

Ograničenja

Karakter	Značenje
r	Ovaj operand ide u neki registar opšte namene, ali ne specificiram koji.
a	Ovaj operand ide u, u zavisnosti od bitaže, %eax, %ax, %al
b	Ovaj operand ide u, u zavisnosti od bitaže, %ebx, %bx, %bl
c	Ovaj operand ide u, u zavisnosti od bitaže, %ecx, %cx, %cl
d	Ovaj operand ide u, u zavisnosti od bitaže, %edx, %dx, %dl
S	Ovaj operand ide u, u zavisnosti od bitaže, %esi, %si
D	Ovaj operand ide u, u zavisnosti od bitaže, %edi, %di

Ograničenja

Karakter	Značenje
m	Ovaj operand je isključivo u nekoj memorijskoj lokaciji, bilo kojoj
o	Ovaj opreand je isključivo u memorijskoj lokaciji, i to nekoj koja je takva da dodavanje malog celog broja ravnoj širini tekuće reči u bajtovima i dalje proizvodi validnu adresu.
=	U ovaj operand samo pišemo, ne čitamo, uvek se stavlja za output operande.
broj	Ako stavimo broj kao ograničenje, onda to znači da istu promenljivu koristimo i za ulaz i za izlaz.
i	operand je konstantan celi broj čija se vrednost zna za vreme asembliranja
E/F	operand je konstantan floating point broj čija se vrednost zna za vreme asembliranja

Clobbers

- Ovo je lista stringova u duplim navodnicima razdvojenih zarezima, koja sadrži sve registre koji se menjaju kao pobočni efekat operacija koje izvršavamo.

- To govori kompajleru da ne očekuje da te vrednosti ostanu iste što utiče na optimizaciju.
- Osim što možemo staviti, npr, "eax" ili "ecx" ovde, može se navesti i "memory" što znači da se modifikuje sadržaj memorije na koji se ne referencira u output sekciji. Kod koji stavlja memory u clobber listu mora biti volatile.

Rukovanje pojedinim bitima memorijskih lokacija

- Emulacija hardvera je namenjena za platforme zasnovane na i386 (i novijim) procesorima koji podržavaju asemblerske naredbe za:
  - dobijanje indeksa najznačajnijeg postavljenog bita u reči `bsr`
  - postavljanje datog bita reči `bts`
  - za njegovo čišćenje `btr`
- Funkcije `ad__get_index_of_most_signifcant_set_bit()`, `ad__set_bit()` i `ad__clear_bit()` posreduju u korišćenju ovih asemblerskih naredbi.

Bitwise asembler

```
1  inline static int
2  ad__get_index_of_most_significant_set_bit(unsigned priority_bits) {
3    int index;                // poziv asm bit scan reverse
4    asm("bsr %1, %0"          //%1 indeks msb, %0 ulazni biti
5        : "=r"(index)         // uvek ide prvo izlazni operand
6        : "r"(priority_bits) // pa ulazni operand
7    );
```

Bitwise asembler

```
8    return index;
9  }
10
11 inline static unsigned ad__set_bit(unsigned priority_bits,
12                                    int index) { // poziva asm bit test and set
13   asm("bts %1, %0"                             //%1 indeks bita, %ulazni biti
14       : "=r"(priority_bits)                    // ulazno izlazni operand pb
```

Bitwise asembler

```
15       : "r"(index), "0"(priority_bits)         // ulazni operand indeks i
16   );                                           // izlazni operand pb
17   return priority_bits;
```

```
18  }
19
20  inline static int ad__clear_bit(unsigned priority_bits,
21                                  int index) { // poziva asm bit test and reset
```

Bitwise asembler

```
22    asm("btr %1, %0" : "=r"(priority_bits) : "r"(index), "0"(priority_bits));
23    return priority_bits;
24  }
```

Rukovanje numeričkim koprocesorom

- Prekljućivanje procesora sa jedne niti na drugu podrazumeva da se sačuva kontekst (sadržaj registara) prve niti i uspostavi kontekst druge niti.
- Kontekst se čuva na steku niti i obuhvata i registre numeričkog koprocesora.
- Klasa `I387_npx` omogućuje pripremu i preuzimanje inicijalnog sadržaja registara numeričkog koprocesora.
- Konstruktor klase `I387_npx` inicijalizuje registre numeričkog koprocesora i smešta njihov inicijalni sadržaj u polje `initial_context` ove klase pomoću asemblerskih naredbi `fninit` i `fnsave`.
- Operacija `initial_context_get()` klase `I387_npx` omogućuje preuzimanje inicijalnog sadržaja registara numeričkog koprocesora.

Rukovanje FPU mehanizmom

```
1   const unsigned i387_SAVE_REGION_SIZE = 0x6c; // velicina FPU steka
2
3   class I387_npx {
4     static char initial_context[i387_SAVE_REGION_SIZE];
5     I387_npx(const I387_npx &);
6     I387_npx &operator=(const I387_npx &);
```

Rukovanje FPU mehanizmom

```
8   public:
9     I387_npx();
10    inline void initial_context_get(Stack_item *stack_top);
11  };
12
13  char I387_npx::initial_context[i387_SAVE_REGION_SIZE] = {0};
```

Rukovanje FPU mehanizmom

```
15  I387_npx::I387_npx() {
16    asm volatile(        // volatile indikacija kompajleru da ne optimizuje
17        "fninit \n\t"     // inicijalizacija FPU, status, tag, IP, DP
18        "fnsavel %0 \n\t" // sacuvaj FPU state u initial_context
19        :
20        : "m"(*initial_context));
21  }
```

Rukovanje FPU mehanizmom

```
22
23  void I387_npx::initial_context_get(Stack_item *stack_top) {
24    for (unsigned i = 0; i < i387_SAVE_REGION_SIZE; i++)
25      ((char *)stack_top)[i] = initial_context[i];
26  }
27
28  static I387_npx i387_npx;
```

Rukovanje stekom

- Za uspeh preključivanja je neophodno da funkcija preključivanja na steku druge niti zatekne ispravan frejm (ako je nit već bila aktivna) ili ako ima spremljen inicijalni kontekst.
- To je obezbeđeno kada se procesor preključuje na nit koja je već bila aktivna. Ali, ako se procesor prvi put preključuje na neku nit, on na njenom steku mora zateći frejm sa ranije pripremljenim njenim inicijalnim kontekstom.

Rukovanje stekom

- Podrazumeva se da prvo preključivanje na neku nit dovodi do početka izvršavanja funkcije koja opisuje dotičnu nit.
- Da bi izvršavanje ove funkcije bilo moguće, neophodno je na steku niti pripremiti frejm njenog poziva sa odgovarajućom povratnom adresom.
- Kao povratna adresa služi adresa funkcije `destroy()`.
- Do izvršavanja funkcije koja opisuje nit dolazi nakon povratka iz funkcije preključivanja, ako se na steku niti pripremi i frejm poziva funkcije preključivanja u kome se kao povratna adresa koristi adresa funkcije koja opisuje nit.
- Pomenuta dva frejma (frejm poziva funkcije koja opisuje nit i frejm poziva funkcije preključivanja) na steku stvarane niti pripremi funkcija `ad__stack_init`

Inicijalizacija steka

```
1   const int INITIAL_FLAGS = 0x0200;
2
3   static inline void ad__stack_init(Stack_item **stack_top,
4                                     unsigned thread_function) {
5     *(--(*stack_top)) = (Stack_item)destroy;
6     *(--(*stack_top)) = (Stack_item)thread_function;
7     *(--(*stack_top)) = (Stack_item)0;
```

Inicijalizacija steka

```
8     *(--(*stack_top)) = (Stack_item) true;
9     *(--(*stack_top)) = (Stack_item)INITIAL_FLAGS;
10    *(--(*stack_top)) = (Stack_item)0;
11    *(--(*stack_top)) = (Stack_item)0;
12    *(--(*stack_top)) = (Stack_item)0;
13    *(--(*stack_top)) = (Stack_item)0;
14    *(--(*stack_top)) = (Stack_item)0;
```

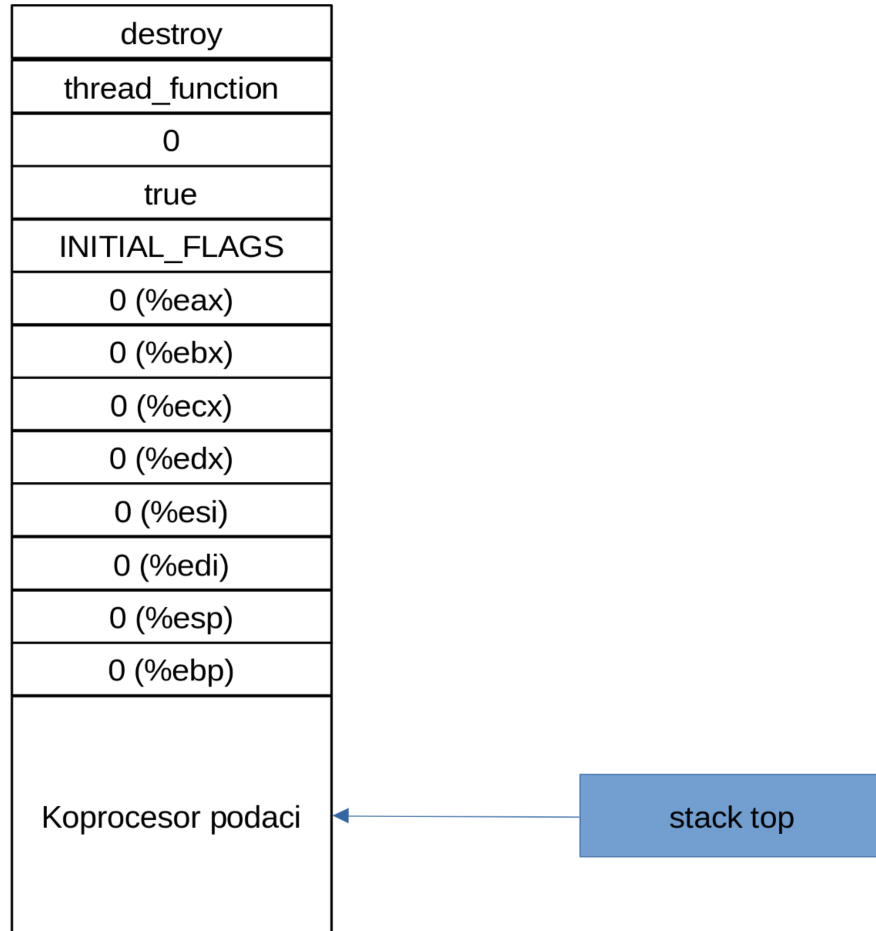
Inicijalizacija steka

```
15    *(--(*stack_top)) = (Stack_item)0;
16    *(--(*stack_top)) = (Stack_item)0;
17    *(--(*stack_top)) = (Stack_item)0;
18    *stack_top = (Stack_item *)(((size_t)(*stack_top)) - i387_SAVE_REGION_SIZE);
19    i387_npx.initial_context_get(*stack_top);
20  }
```

Inicijalizacija steka

```
22  extern "C" void ad__stack_swap(Stack_item **const old_stack,
23                                 const Stack_item *new_stack);
```

Šematski prikaz steka



Rukovanje stekom

- Funkcija `ad__stack_init` na stek smesti:
  - Adresu funkcije destroy() kao povratnu adresu funkcije koja opisuje stvaranu nit
  - Adresu funkcije thread_function() kao povratnu adresu funkcije preključivanja
  - Kontekst niti na koju se procesor prvi put preključuje:
  - Pokazivač prethodnog frejma (0) – nema prethodnog frejma
  - Početno stanje emuliranog bita prekida (true)
  - Početno stanje status (flag) registra – INITIAL_FLAGS
  - Početni sadržaj registara procesora (za registre opšte namene 0, za sadržaj koprocesora rezultat poziva operacije

initial_context_get() klase i387_npx

Swap

- Funkcija ad__stack_swap() ima ulogu funkcije prekljucivanja. Pošto je ona napisana asemblerskim jezikom, radi provere ispravnosti njenih poziva uvedena je njena C deklaracija.

Swap

```
1  .text
2  .align 2
3  .globl ad__stack_swap
4
5  ad__stack_swap:
6      pushl %ebp  //cuvanje zatecenog pokazivaca stek frejma
7      movl %esp,%ebp  //postavljanje novog pokazivaca frejma
```

Swap

```
8      movl 8(%ebp),%edx//adresa vrha steka iz deskriptora niti sa
9               // koje se procesor prekljucuje
10
11     mov interrupts_enabled,%eax  //na stek aktivne niti se smesta
12     push %eax       //zateceno stanje bita prekida
13     pushf           //smestanje flags registra na stek
14     pusha           //smestanje svih registara opste namene na stek
```

Swap

```
15     sub i387_SAVE_REGION_SIZE,%esp
16     fnsavel (%esp)  //smestanje sadrzaja reg koprocesora na stek
17
18         mov %esp,(%edx)//adresa vrha steka se smesti u deskriptor
19             //do tada aktivne niti (lokacija u %edx)
20     mov 12(%ebp),%esp//u pokazivac steka se prebaci adresa vrha
21             //steka iz deskriptora novoaktivirane niti
```

Swap

```
22     frstorl (%esp)  //sa novog steka se preuzmu novi sadrzaji
23                 //registara numerickog koprocesora
24     add i387_SAVE_REGION_SIZE,%esp
25     popa        //sa novog steka se preuzima sadrzaj
26             //registara opste namene
```

```
27      popf        //sa novog steka se preuzima sadrzaj
28              //status registara flags
```

Swap

```
29      pop %eax
30      mov %eax,interrupts_enabled //sa novog steka se preuzme
31                      //sadrzaj bita prekida i frejm
32                      //pointera
33      popl %ebp
34      ret
```

Izvršilac simulatora

Delovi CppTss izvršioca

- Deo CppTss-a koji ima ulogu jezgra operativnog sistema se naziva izvršilac.
- Funkcionalna sličnost CppTss izvršioca sa operativnim sistemom ima za posledicu sličnost njihovih izvedbi.
- Struktura CppTss izvršioca se može predstaviti pomoću istih slojeva kao i struktura operativnog sistema.
- Ključna razlika je da CppTss izvršilac ne sadrži modul za rukovanje datotekama, jer CppTss ne podržava pojam datoteke.

Delovi CppTss izvršioca

Modul	Elementi u kodu
sistemske niti	thread_wake_up_deamon() thread_destroyer_deamon() thread_zero() klasa Delta
modul za rukovanje procesima	klasa thread klasa Thread_image
modul za rukovanje datotekama	-
modul za rukovanje random memorijom	klasa Memory_fragment

Delovi CppTss izvršioca

Modul	Elementi u kodu
modul za rukovanje kontrolerima	klasa Timer_driver klasa Exception_driver klasa Driver
modul za rukovanje procesorom	klasa condition_variable klasa unique_lock klasa mutex klasa Kernel klasa Atomic_region klasa Ready_list klasa Descriptor klasa Permit klasa List_link klasa Failure

Klasa Failure

```
1  enum Failure_codes { MEMORY_SHORTAGE, NOTIFY_OUTSIDE_EXCLUSIVE_REGION };
2  class Failure {
3  protected:
4    const char *f_name;
5    Failure_codes f_code;
6
7  public:
```

Klasa Failure

```
8    Failure(const char *fname, const Failure_codes fcode)
9        : f_name(fname), f_code(fcode){};
10   inline const char *name() const { return f_name; };
11   inline Failure_codes code() const { return f_code; };
12 };
13 Failure failure_memory_shortage("MEMORY SHORTAGE!", MEMORY_SHORTAGE);
14 Failure
```

Klasa Failure

```
15     failure_notify_outside_exclusive_region("NOTIFY OUTSIDE EXCLUSIVE REGION!",
16                                             NOTIFY_OUTSIDE_EXCLUSIVE_REGION);
```

Klasa List_link

```
17 class List_link {
18   List_link *left;
19   List_link *right;
20   List_link(const List_link &);
21   List_link &operator=(const List_link &);
```

```
22
23 public:
```

Klasa List_link

```
24   List_link() {
25     right = this;
26     left = this;
27   };
28   List_link *left_get() const { return left; };
29   List_link *right_get() const { return right; };
30   void insert(List_link *const link);
```

Klasa List_link

```
31   List_link *extract();
32   bool empty() const { return (this == right); };
33   bool not_empty() const { return !empty(); };
34 };
35 void List_link::insert(List_link *const link) {
36   link->left = left;
37   link->right = this;
```

Klasa List_link

```
38   left->right = link;
39   left = link;
40 }
41 List_link *List_link::extract() {
42   List_link *p = right;
43   right->right->left = this;
44   right = right->right;
```

Klasa List_link

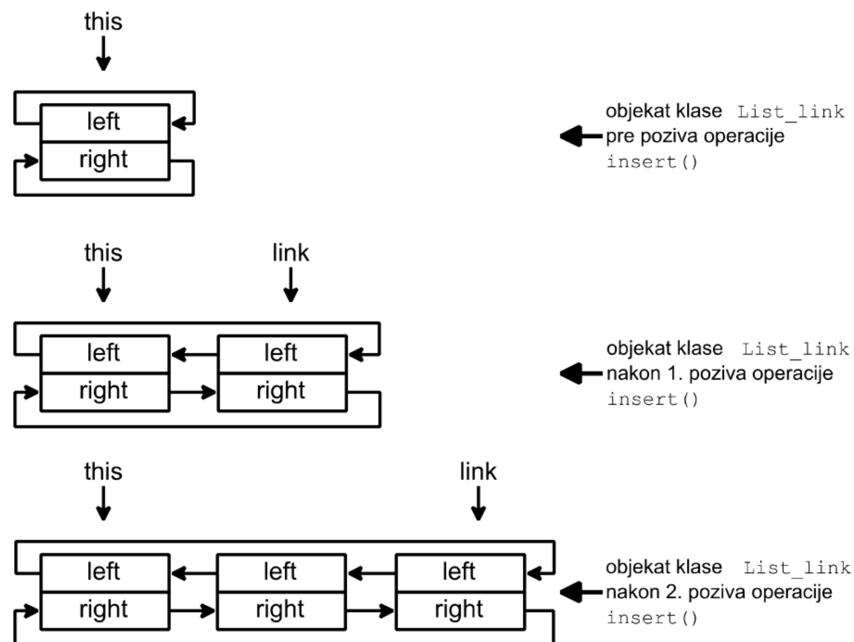
```
45   return p;
46 }
```

Klasa List_link

- Klasa List_link omogućuje obrazovanje dvosmerne cirkularne liste.
- Nju predstavlja objekat ove klase, koji tada istovremeno služi kao njen početak i kraj.

- Takođe se podrazumeva da se oni uvezuju na njen kraj i da se izvezuju sa njenog početka.
- Dvosmerna cirkularna lista se obrazuje pomoću polja left i right.
- Operacije klase List_link omogućuju preuzimanje vrednosti ovih polja: left_get(), right_get()
- uvezivanje novog elementa na kraj liste - insert()
- izvezivanje elementa sa početka liste - extract()
- proveru da li u listi ima uvezanih elemenata - not_empty()
- proveru da li je lista prazna - empty()

Klasa List_link



Klasa Permit

```
class Permit {
  bool free;
  Permit *previous;
  List_link admission_list;
  List_link fulfilled_list;

public:
```

Klasa Permit

```
8    Permit() {
9      free = true;
10     previous = 0;
11   };
12   inline bool not_free() const { return (free == false); };
13   inline void take() { free = false; };
14   inline void release() { free = true; };
```

Klasa Permit

```
15   inline void admission_insert(List_link *link) {
16     admission_list.insert(link);
17   };
18   inline void fulfilled_insert(List_link *link) {
19     fulfilled_list.insert(link);
20   };
21   inline List_link *admission_extract() { return admission_list.extract(); };
```

Klasa Permit

```
22   inline List_link *fulfilled_extract() { return fulfilled_list.extract(); };
23   inline bool admission_not_empty() { return admission_list.not_empty(); };
24   inline bool fulfilled_not_empty() { return fulfilled_list.not_empty(); };
25   friend class Descriptor;
26 };
```

Klasa Permit

- Klasa Permit opisuje rukovanje propusnicama. Polje free klase Permit čuva stanje propusnice.
- Lista (polje previous) je vezana za nit koja je dobila pomenute propusnice. Propusnice se uvezuju u ovu listu u redosledu u kome ih je nit dobila, a izvezuju iz nje u obrnutom redosledu, jer se u obrnutom redosledu propusnice vraćaju.
- Oko polja admission_list klase Permit se obrazuje lista deskriptora niti koje čekaju na propusnicu da bi ušle u isključivi region, a oko njenog polja fulfiled_list se obrazuje lista deskriptora niti koje čekaju na propusnicu nakon ispunjenja uslova.

Klasa Permit

- Operacije klase Permit:
- not_free() - da li je propusnica zauzeta

- take() - zauzimanje propusnice
- release() - oslobadjanje propusnice
- admission_insert(), admission_extract(), admission_not_empty(), fullfilled_insert(), fulfilled_extract(), fulfilled_not_empty() - za rukovanje listama deskriptora niti

Klasa Permit

- Pozivi operacija not_free() i take() moraju biti u istom atomskom regionu , inače se može desiti da više niti jedna za drugom, proverom ustanovi da je ista propusnica slobodna i da zatim, jedna za drugom, zauzme istu propusnicu.
- Pozivi operacija za rukovanje listama deskriptora moraju biti u atomskom regionu.

Klasa Descriptor

```
1  typedef int Stack_item;
2  class Descriptor : private List_link { // nasledjuje se klasa List_link
3    // da bi bilo moguce deskriptore niti uvezivati u liste
4  protected:
5    Stack_item *stack_top; // pokazivac na vrh steka niti
6    int priority;          // prioritet niti
7    Permit *last;          // adresa poslednje dobijene propusnice
```

Klasa Descriptor

```
8    unsigned tag;          // privezak deskriptora niti
9  public:
10   Descriptor();
11   inline unsigned tag_get() const { return tag; };
12   inline void tag_set(unsigned t) { tag = t; };
13   inline void link_permit(Permit *const permit); // uvezivanje
14   // propusnice u listu propusnica prilikom ulaska niti u kriticnu sekciju
```

Klasa Descriptor

```
15   inline Permit *ulink_permit(); // izvezivanje propusnice iz
16   // liste propusnica niti prilikom njenog izlazka iz kriticne sekcije
17   friend class Ready_list;
18   friend class Kernel;
19   friend class thread;
20 };
21 Descriptor::Descriptor() {
```

Klasa Descriptor

```
22      stack_top = 0;
23      priority = 0;
24      last = 0;
25      tag = 0;
26    }
27    void Descriptor::link_permit(Permit *const permit) {
28      permit->previous = last;
```

Klasa Descriptor

```
29      last = permit;
30    }
31    Permit *Descriptor::ulink_permit() {
32      Permit *permit = last;
33      last = permit->previous;
34      return permit;
35    }
```

Klasa Descriptor

- Klasa Descriptor određuje deskriptor niti.
- Ona nasleđuje klasu List_link, da bi bilo moguće deskriptore niti uvezivati u liste.
- Polje stack_top klase Descriptor sadrži pokazivač (adresu) vrha steka niti.
- Prioritet niti je sadržan u polju priority ove klase.
- Polje last klase Descriptor sadrži adresu poslednje dobijene propusnice.
- Polje tag ove klase je namenjeno za smeštanje priveska deskriptora niti.

Klasa Descriptor

- Klasa Descriptor nudi operacije za pristup nekim od njenih polja: tag_get(), tag_set(), kao i operacije za uvezivanje propusnice u listu propusnica niti prilikom njenog ulaska u kritični region: link_permit(), odnosno za izvezivanje propusnice iz liste propusnica niti prilikom njenog izlaska iz kritičnog regiona: ulink_permit().

Klasa Ready_list

```
1  const unsigned PRIORITY_NUMBER = 32;
2  enum Priority {
3    TERMINAL = -1,
4    ZERO = 0,
5    PR01,
6    PR02,
7    PR03,
```

Klasa Ready_list

```
8    PR04,
9    PR05,
10   PR06,
11   PR07,
12   PR08,
13   PR09,
14   PR10,
```

Klasa Ready_list

```
15   PR11,
16   PR12,
17   PR13,
18   PR14,
19   PR15,
20   PR16,
21   PR17,
```

Klasa Ready_list

```
22   PR18,
23   PR19,
24   PR20,
25   PR21,
26   PR22,
27   PR23,
28   PR24,
```

Klasa Ready_list

```
29   PR25,
30   PR26,
31   PR27,
```

```
32    PR28,
33    PR29,
34    PR30,
35    SYSTEM = 31
```

Klasa Ready_list

```
36  };
37  class Ready_list {
38    unsigned priority_bits;
39    List_link ready[PRIORITY_NUMBER];
40    Ready_list(const Ready_list &);
41    Ready_list &operator=(const Ready_list &);
```

Klasa Ready_list

```
43  public:
44    Ready_list() : priority_bits(0){};
45    int highest() const;
46    void insert(Descriptor *d);
47    Descriptor *extract();
48    bool higher_than(Descriptor *d) const;
49  };
```

Klasa Ready_list

```
50  int Ready_list::highest() const {
51    int n = 0;
52    if (priority_bits != 0)
53      n = ad__get_index_of_most_significant_set_bit(priority_bits);
54    return n;
55  }
56  void Ready_list::insert(Descriptor *d) {
```

Klasa Ready_list

```
57    if (d->priority != TERMINAL) {
58      priority_bits = ad__set_bit(priority_bits, d->priority);
59      ready[d->priority].insert(d);
60    }
61  }
62  Descriptor *Ready_list::extract() {
63    Descriptor *d;
```

Klasa Ready_list

```
64    d = (Descriptor *)(ready[highest()].extract());
65    if (ready[d->priority].empty())
66      priority_bits = ad__clear_bit(priority_bits, d->priority);
67    return d;
68  }
69  bool Ready_list::higher_than(Descriptor *d) const {
70    return (highest() > d->priority);
```

Klasa Ready_list

```
71  }
72  static Ready_list ready;
```

Klasa Ready_list

- Klasa Ready_list omogućuje rukovanje spremnim nitima.
- Primer takvog rukovanja je brzo pronalaženje najprioritetnije niti među spremnim nitima, što je osnov za ispunjenje zahteva da je uvek aktivna najprioritetnija nit.
- Radi toga, svakom od prioriteta niti se dodeljuje posebna lista spremnih niti i podrazumeva se da se deskriptor spremne niti uvek uvezuje na kraj liste spremnih niti koja odgovara prioritetu dotične niti.
- Takođe se podrazumeva da se deskriptor spremne niti uvek izvezuje sa početka odabrane liste spremnih niti.
- Na ovaj način spremne niti istog prioriteta se uvek aktiviraju u redosledu u kome su postajale spremne.

Klasa Ready_list

- Sve liste spremnih niti zajedno formiraju multi-listu (Ready::ready).
- Rukovanje ovom multi-listom obuhvata:
- Dobijanje prioriteta najprioritetnije neprazne liste spremnih niti: Ready::highest()
- Uvezivanje deskriptora niti na kraj odgovarajuće liste spremnih niti: Ready::insert()
- izvezivanje deskriptora niti sa početka najprioritetnije neprazne liste spremnih niti: Ready::extract()
- poređenje prioriteta najprioritetnije neprazne liste spremnih niti sa prioritetom zadane niti: Ready::higher_than()

Klasa Ready_list

- Pošto je multi-lista niz listi spremnih niti, deskriptor spremne niti se uvezuje na kraj liste spremnih niti koju direktno indeksira prioritet ove niti.
- Međutim, za operaciju izvezivanja deskriptora iz najprioritetnije neprazne liste spremnih niti, potrebno je prvo pronaći najprioritetniju nepraznu listu spremnih niti.
- Brzo pronalaženje najprioritetnije neprazne liste spremnih niti se ostvaruje tako da se svaka lista spremnih niti reprezentuje jednim bitom koji sadrži 1 ako je lista neprazna, a 0 ako je lista prazna.

Klasa Ready_list

- Ove bite sadrži Ready::priority_bits, tako da se na značajnijim pozicijama nalaze biti prioritetnijih listi spremnih niti.
- Na najmanje značajnoj poziciji je bit nulte liste spremnih niti, sa prioritetom 0.
- U njoj se nalazi posebna nulta nit, sa prioritetom 0.
- Ona angažuje procesor kada nema drugih spremnih niti.
- Kada je nulta nit u stanju "spremna", tada je njen deskriptor uvezan u nultu listu spremnih niti. Nulta nit može biti još samo u stanju "aktivna".
- Ona u to stanje prelazi kada ne postoji neka druga nit koja može da zaposli procesor.

Klasa Ready_list

- Najniži prioritet spremnih niti 0 (ZERO) je rezervisan za nultu nit, a najviši prioritet spremnih niti 31 (SYSTEM) je rezervisan za sistemske niti.
- Između se nalaze prioriteti korisničkih niti (PR01, PR02, ..., PR30).
- Za uništavane niti, koje čekaju da budu uništene i koje više ne mogu biti spremne (a ni aktivne), uveden je poseban završni prioritet -1 (TERMINAL) koji omogućuje njihovo posebno tretiranje u okviru operacije Ready::insert().

Klasa Atomic_region

```
class Atomic_region {
  bool flags;
  Atomic_region(const Atomic_region &);
  Atomic_region &operator=(const Atomic_region &);
```

```
5
6 public:
7   Atomic_region();
```

Klasa Atomic_region

```
8   ~Atomic_region();
9 };
10 Atomic_region::Atomic_region() { flags = ad__disable_interrupts(); }
11 Atomic_region::~Atomic_region() { ad__restore_interrupts(flags); }
```

Klasa Kernel

- Klasa Kernel omogućuje rukovanje procesorom. Rukovanje procesorom se svodi na preključivanje procesora sa jedne niti na drugu.
- Za preključivanje je neophodno imati adresu deskriptora aktivne niti sa koje se procesor preključuje (active), adresu deskriptora niti na koju se procesor preključuje (pretender), kao i adresu deskriptora niti sa koje se procesor preključio (former).
- Operaciju preključivanja poziva privatna operacija switch_to(), koja postavlja pokazivač deskriptora aktivne niti active i pokazivač deskriptora niti sa koje se procesor preključio former.

Klasa Kernel

- Preključivanje je nužno vezano za raspoređivanje (scheduling), odnosno za izbor niti na koju se procesor preključuje.
- Ciljevi raspoređivanja kod CppTss izvršioca su da uvek bude aktivna najprioritetnija spremna nit i da se ravnomerno deli vreme procesora između spremnih niti istog prioriteta. Do raspoređivanja dolazi:
- kada se pojavi spremna nit sa višim prioritetom od aktivne niti (schedule())
- na kraju kvantuma (periodic_schedule())

Klasa Kernel

- Klasa Kernel nasleđuje klasu Deskriptor da bi jedini objekat klase Kernel reprezentovao deskriptor main() niti.
- Konstruktor ove klase proglašava aktivnom main() nit. Pošto main() nit koristi stek konkurentnog programa kao svoj stek, za nju nije potrebno zauzeti stek.

- U nadležnosti klase Kernel nije samo preključivanje, nego i podrška viših slojeva iz hijerarhijske strukture CppTss izvršioca.

Klasa Kernel

- Funkcije:
- make_ready() - omogućava aktivnost niti
- expect() - omogućuje očekivanje dešavanja događaja
- signal() - omogućuje objavu dešavanja događaja
- exclusive_in() - za ulazak u isključivi region
- exclusive_out() - za izlazak iz isključivog regiona
- wait() - očekivanje ispunjenja uslova
- notify_one() - objava ispunjenja uslova
- U operacijama deljene promenljive kernel se koriste atomski regioni radi zaštite njene konzistentnosti, kao i konzistentnosti argumenata iz poziva ovih operacija.

Klasa Kernel

```
1  class Kernel : public Descriptor {
2    Descriptor *active;
3    Descriptor *pretender;
4    Descriptor *former;
5    Kernel(const Kernel &);
6    Kernel &operator=(const Kernel &);
7    inline void switch_to(Descriptor *const d);
```

Klasa Kernel

```
8    inline void schedule();
9    inline void periodic_schedule();
10
11 public:
12   Kernel() : active(0), pretender(0), former(0) {
13     priority = PR15;
14     active = this;
```

Klasa Kernel

```
15   };
16   inline void make_ready(Descriptor *const d);
17   inline void expect(List_link *const waiting_list);
18   inline void signal(List_link *const waiting_list);
19   inline void exclusive_in(Permit *const permit);
```

```
20    inline void wait(const unsigned t, List_link *const waiting_list);
21    inline void notify_one(List_link *const waiting_list);
```

Klasa Kernel

```
22    inline void exclusive_out();
23    inline Descriptor *active_get() const { return active; };
24    friend void yield();
25    friend class Timer_driver;
26  };
27  void Kernel::switch_to(Descriptor *const d) {
28    former = active;
```

Klasa Kernel

```
29    active = d;
30    ad__stack_swap(&(former->stack_top), active->stack_top);
31  }
32  void Kernel::schedule() {
33    if (ready.higher_than(active)) {
34      ready.insert(active);
35      pretender = ready.extract();
```

Klasa Kernel

```
36      switch_to(pretender);
37    }
38  }
39  void Kernel::periodic_schedule() {
40    ready.insert(active);
41    pretender = ready.extract();
42    if (active != pretender)
```

Klasa Kernel

```
43      switch_to(pretender);
44  }
45  void Kernel::make_ready(Descriptor *const d) {
46    Atomic_region ar;
47    ready.insert(d);
48  }
49  void Kernel::expect(List_link *const waiting_list) {
```

Klasa Kernel

```
50    waiting_list->insert(active);
51    pretender = ready.extract();
52    switch_to(pretender);
53  }
54  void Kernel::signal(List_link *const waiting_list) {
55    if (waiting_list->not_empty()) {
56      pretender = (Descriptor *)waiting_list->extract();
```

Klasa Kernel

```
57      ready.insert(pretender);
58      schedule();
59    }
60  }
61  void Kernel::exclusive_in(Permit *const permit) {
62    Atomic_region ar;
63    if (permit->not_free()) {
```

Klasa Kernel

```
64      permit->admission_insert(active);
65      pretender = ready.extract();
66      switch_to(pretender);
67    } else {
68      permit->take();
69      active->link_permit(permit);
70    }
```

Klasa Kernel

```
71  }
72  void Kernel::wait(const unsigned t, List_link *const waiting_list) {
73    Atomic_region ar;
74    Permit *permit = active->ulink_permit();
75    active->tag = t;
76    if (permit->fulfilled_not_empty()) {
77      pretender = (Descriptor *)permit->fulfilled_extract();
```

Klasa Kernel

```
78      pretender->link_permit(permit);
79      ready.insert(pretender);
80    } else if (permit->admission_not_empty()) {
```

```
81      pretender = (Descriptor *)permit->admission_extract();
82      pretender->link_permit(permit);
83      ready.insert(pretender);
84    } else
```

Klasa Kernel

```
85      permit->release();
86    waiting_list->insert(active);
87    pretender = ready.extract();
88    switch_to(pretender);
89  }
90  void Kernel::notify_one(List_link *const waiting_list) {
91    Permit *permit = active->last;
```

Klasa Kernel

```
92    if (permit == 0)
93      throw &failure_notify_outside_exclusive_region;
94    Atomic_region ar;
95    if (waiting_list->not_empty()) {
96      pretender = (Descriptor *)waiting_list->extract();
97      permit->fulfilled_insert(pretender);
98    }
```

Klasa Kernel

```
99  }
100 void Kernel::exclusive_out() {
101   Atomic_region ar;
102   Permit *permit = active->ulink_permit();
103   if (permit->fulfilled_not_empty()) {
104     pretender = (Descriptor *)permit->fulfilled_extract();
105     pretender->link_permit(permit);
```

Klasa Kernel

```
106     ready.insert(pretender);
107   } else if (permit->admission_not_empty()) {
108     pretender = (Descriptor *)permit->admission_extract();
109     pretender->link_permit(permit);
110     ready.insert(pretender);
111   } else
112     permit->release();
```

Klasa Kernel

```
113   schedule();
114 }
115 static Kernel kernel;
116 void yield() {
117   Atomic_region set_up;
118   kernel.periodic_schedule();
119 }
```

Klasa mutex

```
120 class mutex : private Permit {
121   mutex(const mutex &);
122   mutex &operator=(const mutex &);
123
124 public:
125   mutex(){};
126   void lock();
```

Klasa mutex

```
127   void unlock();
128 };
129 void mutex::lock() { kernel.exclusive_in(this); }
130 void mutex::unlock() { kernel.exclusive_out(); }
```

Klasa unique_lock

```
131 template <class MUTEX> class unique_lock {
132   unique_lock(const unique_lock &);
133   unique_lock &operator=(const unique_lock &);
134
135 public:
136   unique_lock(MUTEX &mx);
137   ~unique_lock();
```

Klasa unique_lock

```
138 };
139 template <class MUTEX> unique_lock<MUTEX>::unique_lock(MUTEX &mx) {
140   kernel.exclusive_in((Permit *)&mx);
141 }
142 template <class MUTEX> unique_lock<MUTEX>::~unique_lock() {
```

```
143   kernel.exclusive_out();
144 }
```

Klasa condition_variable

```
145 class condition_variable {
146   List_link list_head;
147   List_link *position;
148
149 public:
150   condition_variable() { position = &list_head; };
151   void wait(unique_lock<mutex> &lock, unsigned t = 0);
```

Klasa condition_variable

```
152   void notify_one();
153   bool first(unsigned *t = 0);
154   bool last();
155   bool next(unsigned *t = 0);
156   bool attach_tag(unsigned t);
157 };
158 void condition_variable::wait(unique_lock<mutex> &lock, unsigned t) {
```

Klasa condition_variable

```
159   kernel.wait(t, position);
160   position = &list_head;
161 }
162 void condition_variable::notify_one() {
163   kernel.notify_one(&list_head);
164   position = &list_head;
165 }
```

Klasa condition_variable

```
166 bool condition_variable::first(unsigned *t) {
167   bool r = false;
168   if (list_head.not_empty()) {
169     position = list_head.right_get();
170     if (t != 0)
171       *t = ((Descriptor *)position)->tag_get();
172     r = true;
```

Klasa condition_variable

```
173   }
174   return (r);
175 }
176 bool condition_variable::last() {
177   bool r = false;
178   if (list_head.not_empty()) {
179     position = &list_head;
```

Klasa condition_variable

```
180     r = true;
181   }
182   return (r);
183 }
184 bool condition_variable::next(unsigned *t) {
185   bool r = false;
186   if (position != &list_head) {
```

Klasa condition_variable

```
187     position = position->right_get();
188     if (position != &list_head) {
189       if (t != 0)
190         *t = ((Descriptor *)position)->tag_get();
191       r = true;
192     }
193   }
```

Klasa condition_variable

```
194   return (r);
195 }
196 bool condition_variable::attach_tag(unsigned t) {
197   bool r = false;
198   if (position != &list_head) {
199     ((Descriptor *)position)->tag_set(t);
200     r = true;
```

Klasa condition_variable

```
201   }
202   return (r);
203 }
```

Klasa Driver

- Klasa Driver omogućuje smeštanje adrese obrađivača prekida u tabelu prekida: Driver::start_interrupt_handling().
- Pored toga, ova klasa uvodi definiciju klase Event koja omogućuje zaustavljanje aktivnosti niti do dešavanja događaja i objavu dešavanja događaja.

Klasa Driver

```
1  class Driver {
2  protected:
3    void start_interrupt_handling(Vector_numbers vector_number,
4                                  void (*handler)());
5    class Event {
6      List_link list_head;
```

Klasa Driver

```
8    public:
9      void expect();
10     void signal();
11   };
12 };
13 void Driver::start_interrupt_handling(Vector_numbers vector_number,
14                                       void (*handler)()) {
```

Klasa Driver

```
15   Atomic_region ar;
16   ad__set_vector(vector_number, handler);
17 }
18 void Driver::Event::expect() { kernel.expect(&list_head); }
19 class Driver {
20 protected:
21   void start_interrupt_handling(Vector_numbers vector_number,
```

Klasa Driver

```
22                                 void (*handler)());
23   class Event {
24     List_link list_head;
25 
26   public:
```

```
27      void expect();
28      void signal();
```

Klasa Driver

```
29    };
30  };
31  void Driver::start_interrupt_handling(Vector_numbers vector_number,
32                                        void (*handler)()) {
33    Atomic_region ar;
34    ad__set_vector(vector_number, handler);
35  }
```

Klasa Driver

```
36  void Driver::Event::expect() { kernel.expect(&list_head); }
37  void Driver::Event::signal() { kernel.signal(&list_head); }
```

Klase Exception_driver i Timer_driver

- Iz klase Driver su izvedene klase Exception_driver i Timer_driver.
- Prva od njih omogućuje reakciju na pojavu hardverskih izuzetaka, radi izazivanja prevremenog kraja konkurentnog programa.
- Druga od ovih klasa omogućuje rukovanje vremenom.
- Klasa Exception_driver uvodi operaciju interrupt_handler(). Ova operacija zaustavlja izvršavanje konkurentnog programa.

Klase Exception_driver i Timer_driver

- Rukovanje vremenom obuhvata:
- brojanje otkucaja sata, radi praćenja proticanja sistemskog vremena
- odbrojavanje otkucaja sata preostalih do kraja kvantuma aktivne niti
- odbrojavanja otkucaja sata preostalih do buđenja uspavane niti
- Kada broj otkucaja, preostalih do isticanja kvantuma aktivne niti, padne na nulu, potrebno je pokrenuti periodično raspoređivanje.

Klase Exception_driver i Timer_driver

- Takođe, kada broj otkucaja, preostalih do buđenja uspavane niti, padne na nulu, potrebno je probuditi sve niti za koje je nastupio trenutak buđenja.

- Svi prethodno pobrojani poslovi se nalaze u nadležnosti operacije interrupt_handler() koju uvodi klasa Timer_driver.
- Polje current_ticks ove klase sadrži sistemsko vreme, polje countdown sadrži broj otkucaja do buđenja, a polje rest broj otkucaja do isticanja kvantuma.
- Funkcija now() vraća sadržaj polja current_ticks, odnosno vraća sistemsko vreme.

Klase Exception_driver i Timer_driver

```
1  const unsigned long QUANTUM = 2;
2  class Exception_driver : public Driver {
3    static void interrupt_handler();
4
5  public:
6    Exception_driver() {
7      start_interrupt_handling(FP_EXCEPTION, interrupt_handler);
```

Klase Exception_driver i Timer_driver

```
8    };
9  };
10 void Exception_driver::interrupt_handler() {
11   ad__report_and_finish("\nHARDWARE EXCEPTION!\n");
12 }
13 static Exception_driver exception_driver;
14 class Timer_driver : public Driver {
```

Klase Exception_driver i Timer_driver

```
15   static unsigned long current_ticks;
16   static unsigned long countdown;
17   static unsigned long rest;
18   static unsigned long quantum;
19   static Event alarm;
20   static void interrupt_handler();
```

Klase Exception_driver i Timer_driver

```
22 public:
23   Timer_driver() { start_interrupt_handling(TIMER, interrupt_handler); };
24   friend unsigned long now();
25   friend class Delta;
26 };
```

```
27 unsigned long Timer_driver::current_ticks = 0;
28 unsigned long Timer_driver::countdown = 0;
```

Klase Exception_driver i Timer_driver

```
29 unsigned long Timer_driver::rest = QUANTUM;
30 Timer_driver::Event Timer_driver::alarm;
31 void Timer_driver::interrupt_handler() {
32   current_ticks++;
33   if ((--rest) == 0)
34     rest = quantum;
35   if ((countdown > 0) && ((--countdown) == 0))
```

Klase Exception_driver i Timer_driver

```
36     alarm.signal();
37   else if (rest == quantum)
38     kernel.periodic_schedule();
39 }
40 static Timer_driver timer_driver;
41 unsigned long now() {
42   Atomic_region ar;
```

Klase Exception_driver i Timer_driver

```
43   return timer_driver.current_ticks;
44 }
```

Klasa Memory_fragment

- Klasa Memory_fragment omogućuje rukovanje slobodnom radnom memorijom. Slobodnu radnu memoriju obrazuje celi broj jedinica sastavljenih od UNIT bajta.
- Rukovanje slobodnom radnom memorijom podrazumeva da se uvek zauzima, odnosno da se uvek oslobađa celi broj ovih jedinica.
- Zauzimanja ovakvih zona slobodne radne memorije, odnosno njihova oslobađanja uzrokuju iscepkanost slobodne radne memorije u odsečke.
- Odsečci se zato uvezuju u jednosmernu listu, uređenu u rastućem redosledu njihovih početnih adresa.
- Radi toga, početak svakog odsečka sadrži svoju veličinu, izraženu u pomenutim jedinicama od po UNIT bajta, i pokazivač narednog odsečka.

- Veličinu odsečka i pokazivač narednog odsečka sadrže polja size i next klase Memory_fragment.

Klasa Memory_fragment

- Konstruktor klase Memory_fragment opisuje obrazovanje liste odsečaka slobodne radne memorije, sastavljene od stalnog odsečka čija veličina je 0 i od odsečka koji obuhvata raspoloživu slobodnu radnu memoriju.
- Stalnom (prvom) odsečku odgovara objekt memory klase Memory_fragment, koji je jedini objekt ove klase. Dodavanju drugog odsečka prethodi provera da li je obezbeđeno dovoljno radne memorije za potrebe konkurentnog programa.
- Ako nije, izvršavanje konkurentnog programa se završava uz poruku INITIAL MEMORY SHORTAGE.
- Pošto je UNIT jednak veličini stranice, uvek se zauzima toliko radne memorije da u nju može da stane traženi broj stranica, a da početak raspoložive slobodne radne memorije bude postavljen na početak prve stranice.

Klasa Memory_fragment

- Zauzimanje slobodne radne memorije omogućuje operacija take() klase Memory_fragment.
- Zauzima se jedna jedinica od UNIT bajta više nego što je traženo.
- Ona prethodi preostalim zauzetim jedinicama memorije i sadrži ukupnu veličinu zauzete memorije. Ova veličina se koristi prilikom kasnijeg oslobađanja zauzete memorije.
- Zauzimanju prethodi pretraživanje liste odsečaka, radi pronalaženja prvog dovoljno velikog odsečka.
- Pretraživanje uvek počinje od stalnog odsečka. Ako se pronađe dovoljno velik odsečak, traženi bajti se zauzimaju s njegovog kraja. Ako pronađeni odsečak obuhvata baš traženi broj bajta, tada se on isključuje iz liste i zauzimaju se svi njegovi bajti.

Klasa Memory_fragment

- Oslobađanje prethodno zauzete radne memorije omogućuje operacija free() klase Memory_fragment.
- Za oslobađanje je neophodno u listi odsečaka pronaći odsečak iza koga će se oslobađani odsečak uvezati u ovu listu.
- Pre uvezivanja proverava se da li oslobađani odsečak može da se spoji u jedan odsečak sa svojim prethodnikom i sa svojim sledbenikom.

- Odsečak se uvezuje u pomenutu listu samo ako ovo spajanje nije moguće.

Klasa Memory_fragment

- Prethodno opisane operacije klase Memory_fragment su namenjene za zauzimanje i oslobađanje radne memorije prilikom stvaranja i uništavanja objekata pojedinih klasa.
- Da bi se njihova namena ostvarila, neophodno je da ove operacije pozivaju globalni operatori new() i delete().
- Ali, tada razne niti mogu da pozivaju operacije klase Memory_fragment posredstvom prethodna dva operatora i da tako ugroze konzistentnost liste odsečaka.
- Da bi se to sprečilo, ova klasa nasleđuje klasu mutex i tako omogućuje zaključavanje i otključavanje njenog jedinog objekta memory. To je obezbeđeno u definicijama funkcija operator new() i operator delete(), radi ostvarenja međusobne isključivosti različitih pristupanja listi odsečaka.

Klasa Memory_fragment

```
1  const size_t UNIT = PAGE_SIZE;
2  class Memory_fragment : public mutex {
3    size_t size;
4    Memory_fragment *next;
5    Memory_fragment(const Memory_fragment &);
6    Memory_fragment &operator=(const Memory_fragment &);
```

Klasa Memory_fragment

```
8  public:
9    Memory_fragment();
10   void *take(size_t size);
11   void free(void *address);
12 };
13 Memory_fragment::Memory_fragment() : size(0), next(this) {
14   size_t free_memory = 2000 * UNIT;
```

Klasa Memory_fragment

```
15   size_t beginning = (size_t)malloc(free_memory + UNIT - 1);
16   if (beginning == 0)
17     ad__report_and_finish("INITIAL MEMORY SHORTAGE");
18   else {
19     beginning = (beginning + UNIT - 1) & ~(UNIT - 1);
```

```
20      next = (Memory_fragment *)beginning;
21      next->size = free_memory;
```

Klasa Memory_fragment

```
22      next->next = this;
23    }
24  }
25  void *Memory_fragment::take(size_t size) {
26    size += 2 * UNIT - 1; // 1 UNIT za broj zauzetih i drugi ako
27    size -= size % UNIT;  // size nije ceo broj unita
28    Memory_fragment *m = 0;
```

Klasa Memory_fragment

```
29    Memory_fragment *p = this;
30    while (p->next != this) {
31      if ((p->next->size) < size)
32        p = p->next;
33      else if (p->next->size == size) {
34        m = p->next;
35        p->next = p->next->next;
```

Klasa Memory_fragment

```
36        break;
37      } else {
38        p->next->size -= size;
39        m = (Memory_fragment *)((size_t)(p->next) + (p->next->size));
40        break;
41      }
42    }
```

Klasa Memory_fragment

```
43    if (m == 0)
44      throw &failure_memory_shortage;
45    m->size = size;
46    return (void *)((size_t)m + UNIT);
47  }
48  void Memory_fragment::free(void *address) {
49    if (address != 0) {
```

Klasa Memory_fragment

```
50      Memory_fragment *a = (Memory_fragment *)((size_t)address - UNIT);
51      Memory_fragment *p = this;
52      while (p->next != this)
53        if (a > p->next)
54          p = p->next;
55        else
56          break;
```

Klasa Memory_fragment

```
57      if ((((size_t)p) + (p->size)) == ((size_t)a)) {
58        p->size += a->size;
59        if ((((size_t)p) + (p->size)) == ((size_t)(p->next))) {
60          p->size += p->next->size;
61          p->next = p->next->next;
62        }
63      } else if ((((size_t)a) + (a->size)) == ((size_t)(p->next))) {
```

Klasa Memory_fragment

```
64        a->size += p->next->size;
65        a->next = p->next->next;
66        p->next = a;
67      } else {
68        a->next = p->next;
69        p->next = a;
70      }
```

Klasa Memory_fragment

```
71    }
72  }
73  static Memory_fragment memory;
74  void *operator new(size_t size) {
75    void *memory_block;
76    memory.lock();
77    try {
```

Klasa Memory_fragment

```
78      memory_block = memory.take(size);
79    } catch (...) {
80      memory.unlock();
```

```
81      throw;
82    }
83    memory.unlock();
84    return memory_block;
```

Klasa Memory_fragment

```
85  }
86  void operator delete(void *address) {
87    memory.lock();
88    memory.free(address);
89    memory.unlock();
90  }
```

Klase Thread_image i thread

- Rukovanje nitima omogućuju klase Thread_image i thread, kao i definicije funkcija thread_destroyer_deamon(), destroy() i undetached_threads().
- Klasa Thread_image opisuje sliku niti, sastavljenu od:
- deskriptora niti
- uslova (ended) koji omogućuje čekanje završetka aktivnosti niti
- oznake da regularan kraj aktivnosti niti može da nastupi kao posledica kraja aktivnosti procesa kome dotična nit pripada (detached)
- steka niti

Klase Thread_image i thread

- Klasa thread omogućuje:
- međusobnu isključivost svojih operacija (mx)
- brojanje niti za koje koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome dotične niti pripadaju (undetached_threads_number)
- uništavanje niti (termination, terminating)
- pristup slici niti (ti)
- Konstruktor klase thread omogućuje kreiranje niti. U toku kreiranja niti pripremi se njen stek za prekljućivanje, da bi automatski započelo izvršavanje funkcije koja opisuje ponašanje niti nakon prvog prekljućivanja na nit.
- Završetak aktivnosti niti se otkriva u okviru operacija join() i detach() na osnovu završnog prioriteta niti (TERMINAL).

Klase Thread_image i thread

- Na završetku aktivnosti niti, u toku izvršavanja funkcije destroy(), omogućuje se nastavak aktivnosti niti koja čeka dotični završetak i oslobađanje prostora koga zauzima slika niti, što je u nadležnosti sistemske niti thread_destroyer_deamon().
- Funkcija undetached_threads() omogućuje proveru da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome dotične niti pripadaju, da bi se na kraju aktivnosti procesa ukazalo na prevremeni završetak ovakvih niti.

Klase Thread_image i thread

```
1  const unsigned DEFAULT_STACK_SIZE = 4096;
2  class Thread_image : public Descriptor {
3    condition_variable ended;
4    bool detached;
5    Stack_item stack[DEFAULT_STACK_SIZE];
6    Thread_image(const Thread_image &);
7    Thread_image &operator=(const Thread_image &);
```

Klase Thread_image i thread

```
8
9  public:
10   Thread_image(void (*thread_function)(), Priority p);
11   friend class thread;
12   friend void destroy();
13 };
14 Thread_image::Thread_image(void (*thread_function)(), Priority p)
```

Klase Thread_image i thread

```
15     : detached(false) {
16   stack_top = &(stack[DEFAULT_STACK_SIZE]);
17   ad__stack_init(&stack_top, (unsigned)thread_function);
18   priority = p;
19 }
20 class thread {
21   static mutex mx;
```

Klase Thread_image i thread

```
22    static unsigned undetached_threads_number;
23    static condition_variable termination;
24    static Thread_image *terminating;
25    Thread_image *ti;
26    thread(const thread &);
27    thread &operator=(const thread &);
```

Klase Thread_image i thread

```
29  public:
30    thread(void (*thread_function)(), Priority p = PR15);
31    void join();
32    void detach();
33    friend void thread_destroyer_deamon();
34    friend void destroy();
35    friend bool undetached_threads();
```

Klase Thread_image i thread

```
36  };
37  mutex thread::mx;
38  unsigned thread::undetached_threads_number = 0;
39  condition_variable thread::termination;
40  Thread_image *thread::terminating;
41  thread::thread(void (*thread_function)(), Priority p) {
42    ti = new Thread_image(thread_function, p);
```

Klase Thread_image i thread

```
43    unique_lock<mutex> lock(mx);
44    undetached_threads_number++;
45    kernel.make_ready(ti);
46  }
47  void thread::join() {
48    unique_lock<mutex> lock(mx);
49    if (ti->priority != TERMINAL)
```

Klase Thread_image i thread

```
50      ti->ended.wait(lock);
51  }
52  void thread::detach() {
53    unique_lock<mutex> lock(mx);
```

```
54    if ((ti->priority != TERMINAL) && (!ti->detached)) {
55      ti->detached = true;
56      undetached_threads_number--;
```

Klase Thread_image i thread

```
57    }
58  }
59  void thread_destroyer_deamon() {
60    for (;;) {
61      unique_lock<mutex> lock(thread::mx);
62      thread::termination.wait(lock);
63      delete thread::terminating;
```

Klase Thread_image i thread

```
64    }
65  }
66  void destroy() {
67    unique_lock<mutex> lock(thread::mx);
68    thread::terminating = (Thread_image *)kernel.active_get();
69    while (thread::terminating->ended.last())
70      thread::terminating->ended.notify_one();
```

Klase Thread_image i thread

```
71    if (!thread::terminating->detached)
72      thread::undetached_threads_number--;
73    thread::terminating->priority = TERMINAL;
74    thread::termination.notify_one();
75  }
76  bool undetached_threads() { return (thread::undetached_threads_number > 0); }
```

Klasa Delta

- Klasa Delta i funkcije thread_wake_up_deamon(), sleep_for() i sleep_until() zajedno omogućuju uspavljivanje i buđenje niti, a funkcija thread_zero() opisuje aktivnost nulte (sistemske) niti.
- sleep_for() omogućuje uspavljivanje aktivne niti dok ne protekne zadani broj otkucaja sata
- sleep_until() omogućuje uspavljivanje aktivne niti dok ne nastupi zadani trenutak sistemskog vremena.

Klasa Delta

- Oko polja list klase Delta se formira lista deskriptora uspavanih niti.
- Da se za svaku uspavanu nit ne bi proveravalo, nakon svakog otkucaja, da li je nastupilo vreme njenog buđenja, deskriptori uspavanih niti se uvezuju u listu u hronološkom redosledu buđenja niti.
- Svakom od ovih deskriptora je dodeljen privezak koji pokazuje relativno vreme buđenja (relativni broj otkucaja do buđenja) u odnosu na prethodnika u listi.
- Ovakva lista se zove delta lista.
- Zahvaljujući delta listi, nakon svakog otkucaja potrebno je proveriti da li je nastupio trenutak buđenja samo za nit koja se najranije budi, odnosno samo za prvi deskriptor iz delta liste.
- Pošto može da bude više niti, čije buđenje je vezano za isti trenutak, unapred nije poznato koliko niti treba probuditi nakon otkucaja sata.

Klasa Delta

- Operaciju awake() klase Delta poziva sistemska nit Wake_up_daemon(). Vreme njenog buđenja je uvek jednako najranijem vremenu buđenja korisničkih niti.
- Nakon buđenja, sistemska nit budi sve korisničke niti sa početka delta liste, za koje je nastupio trenutak buđenja.
- Čekanje buđenja omogućuje poziv operacije Timer_driver::alarm.expect().

Klasa Delta

- Dužinu čekanja određuje vrednost lokalne promenljive tag, kada ima uspavanih korisničkih niti (na čije prisustvo ukazuje vrednost lokalne promenljive sleeping).
- Dužina čekanja se skraćuje za vrednost lokalne promenljive passed_ticks, koja registruje vreme proteklo na buđenju korisničkih niti.

Klasa Delta

- Operaciju sleep() klase Delta poziva, posredstvom funkcije sleep_for(), korisnička nit, da bi se njen deskriptor uključio u delta listu, a njena aktivnost privremeno zaustavila.
- Konstruktor klase Delta omogućuje kreiranje sistemskih niti korišćenjem bezimenih (privremenih) objekata klase thread.

- Konzistentnost delta liste štite isključivi regioni u telima operacija awake() i sleep() klase Delta.