# Virtuelna Mašina CppTss sistema

UDŽBENIK, STRANCE 88-109

# Virtuelna mašina CppTss-A

- 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

# Virtuelna mašina CppTss-A

- Sistemske biblioteke korišćene u emulaciji hardvera (datoteka includes.cpp):

```
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <signal.h>
#include <sys/time.h>
#include <string.h>
#include <stdio.h>
```

- Zahvaljujući emulaciji hardvera konkurentni program ne pristupa stvarnom, nego **emuliranom** hardveru, pa se izvršava kao običan (neprivilegovan) **Linux proces**.
- Zato, u slučaju grešaka, on **ne može ugroziti** funkcionisanje operativnog sistema u okviru koga se izvršava.

# 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**.

```
-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 mehanizma prekida

```
bool interrupts_enabled = true;
inline static bool ad__disable_interrupts()
{
        bool saved_interrupts_enabled = interrupts_enabled;
        interrupts_enabled = false;
        return saved_interrupts_enabled;
}

inline void ad__restore_interrupts(bool saved_interrupts_enabled)
{
        if(saved_interrupts_enabled && interrupt.pending) {
                interrupt.controller_emulator();
                interrupt.pending = false;
        }
        interrupts_enabled = saved_interrupts_enabled;
}
```

# 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 **preključivanja** na drugu nit, započeta obrada signala će biti **završena** tek nakon ponovnog **preključivanja** 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 mehanizma prekida

```
class Linux_signals {
        struct sigaction sa;   //struktura za definisanje signala
        static void signal_handler(int signal);
        Linux_signals(const Linux_signals&);
        Linux_signals& operator=(const Linux_signals&);
public:
        Linux_signals();
        ~Linux_signals();
};

void Linux_signals::signal_handler(int signal) //override signala
{
        switch(signal) {
        case SIGVTALRM:
                interrupt.emulation();
                break;
        case SIGFPE:
                interrupt.handler(FP_EXCEPTION);
                break;
        }
}
```

# Emulacija mehanizma prekida

```
Linux_signals::Linux_signals()
{
        sa.sa_handler=signal_handler;
        sigemptyset(&(sa.sa_mask));   //Ne blokiramo nijedan signal
        sa.sa_flags=SA_NODEFER;       //obrada bez odlaganja
        sigaction(SIGVTALRM, &sa, 0); //promena signala za
        sigaction(SIGFPE, &sa, 0);    //SIGVTALRM i SIGFPE
}

Linux_signals::~Linux_signals()
{
        sa.sa_handler = SIG_DFL;          //vracanje na default
        sigaction(SIGVTALRM, &sa, 0);
        sigaction(SIGFPE, &sa, 0);
}

Linux_signals linux_signals;
```

# Emulacija mehanizma prekida

- Konstruktor klase Linux_timer obezbedi da se signal SIGVTALRM dešava svakih 10 milisekundi.

```
const int LINUX_TIMER_INTERVAL = 10;

class Linux_timer {
        struct itimerval itimer;
        Linux_timer(const Linux_timer&);
        Linux_timer& operator=(const Linux_timer&);
public:
        Linux_timer();
        ~Linux_timer();
};
```

# Emulacija mehanizma prekida

```
Linux_timer::Linux_timer()  //usec vreme se meri u mikrosekundama
{
        itimer.it_interval.tv_sec = 0; //tekuća vrednost
        itimer.it_interval.tv_usec = LINUX_TIMER_INTERVAL * 1000;
        itimer.it_value.tv_sec = 0; //vrednost kojom se resetuje
        itimer.it_value.tv_usec = LINUX_TIMER_INTERVAL * 1000;
        setitimer(ITIMER_VIRTUAL, &itimer, 0);
}

Linux_timer::~Linux_timer()
{
        itimer.it_value.tv_sec = 0;
        itimer.it_value.tv_usec = 0;
        setitimer(ITIMER_VIRTUAL, &itimer, 0); //user mode timer
}

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 mehanizma prekida

```
class Interrupt {
        static void (*vector[INTERRUPT_TABLE_VECTOR_COUNT])();
        bool pending;
        int controller_turn;
        bool timer_turn;
        Interrupt(const Interrupt&);
        Interrupt& operator=(const Interrupt&);
public:
        Interrupt();
        inline void handler(unsigned index);
        inline void emulation();
        inline void controller_emulator();
        friend inline void ad__restore_interrupts(
                                bool new_interrupt_status);
        friend inline void ad__set_vector(int index, void (*handler)());
};
```

# Emulacija mehanizma prekida

```
void default_interrupt_handler()
{
}

void (*Interrupt::vector[INTERRUPT_TABLE_VECTOR_COUNT])()
= {default_interrupt_handler};

Interrupt::Interrupt()
: pending(false), controller_turn(0), timer_turn(true)
{
}

void Interrupt::handler(unsigned index)
{
        (vector[index])();
}
```

# Emulacija mehanizma prekida

```
void Interrupt::emulation()
{
        if(!interrupts_enabled)
                interrupt.pending = true;
        else {
                interrupts_enabled = false;
                controller_emulator();
                interrupts_enabled = true;
        }
}
```

# Emulacija mehanizma prekida

```
void Interrupt::controller_emulator()
{
        bool interrupt_emulated;
        int counter = 0;
        if(timer_turn) {
                timer_turn = false;
                handler(TIMER);
        } else {
                timer_turn = true;
                do {
                switch(controller_turn) {
                    case 0:
                        interrupt_emulated = display_controller.output();
                        controller_turn = 1;
                        break;
                    case 1:
                        interrupt_emulated = keyboard_controller.input();
                        controller_turn = 2;
                        break;
                    case 2:
                        interrupt_emulated = disk_controller.transfer();
                        controller_turn = 0;
                        break;
                }
                } while(!interrupt_emulated && ++counter < 3);
        }
}
```

# Emulacija mehanizma prekida

```
inline void ad__set_vector(int index, void (*handler)())
{
        Interrupt::vector[index] = handler;
}

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

```
class Keyboard_controller {
        char data_reg;
        bool input();
        Keyboard_controller(const Keyboard_controller&);
        Keyboard_controller&
                operator=(const Keyboard_controller&);
public:
        Keyboard_controller() {};
        friend class Interrupt;
        friend class Keyboard_driver;
};
```

# Emulacija kontrolera tastature

```
bool Keyboard_controller::input()
{
	bool interrupt_emulated = false;
	unsigned read_count;
	char c;
	read_count = read(STDIN_FILENO, &c, 1); //sys call
	if(read_count > 0) {
		data_reg = c;
		interrupt.handler(KEYBOARD);
		interrupt_emulated = true;
	}
	return interrupt_emulated;
}

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

```
enum Display_status { DISPLAY_READY, DISPLAY_BUSY };

class Display_controller {
        char data_reg;
        Display_status status_reg;
        bool output();
        Display_controller(const Display_controller&);
        Display_controller& operator=(
                                    const Display_controller&);
public:
        Display_controller() : status_reg(DISPLAY_READY) {};
        friend class Interrupt;
        friend class Display_driver;
};
```

# Emulacija kontrolera ekrana

```
bool Display_controller::output()
{
	bool interrupt_emulated = false;
	if(status_reg == DISPLAY_BUSY) {
		write(STDOUT_FILENO, &data_reg, 1); //sys call
		status_reg = DISPLAY_READY;
		interrupt.handler(DISPLAY);
		interrupt_emulated = true;
	}
	return interrupt_emulated;
}

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.

# Emulacija kontrolera tastature i ekrana

```
class Linux_terminal {
        struct termios ts;
        struct termios ots;
        Linux_terminal(const Linux_terminal&);
        Linux_terminal& operator=(const Linux_terminal&);
public:
        Linux_terminal();
        ~Linux_terminal();
};

Linux_terminal::Linux_terminal()
{
        tcgetattr(STDIN_FILENO, &ts);   //preuzmi parametre terminala
        ots = ts;
        ts.c_lflag &= ~ECHO;          //zabrana eha na ekran
        ts.c_lflag &= ~ICANON;                    //flag nekanonskog moda
        ts.c_cc[VTIME] = 0; //timeout u decisekunda za nekanonski read
        ts.c_cc[VMIN] = 0;        //minimalni broj karaktera za nekanonski read
        tcsetattr(STDIN_FILENO, TCSANOW, &ts); //postavi parametre term
}
```

# Emulacija kontrolera tastature i ekrana

```
Linux_terminal::~Linux_terminal()
{
        if(undetached_threads())
                write(STDOUT_FILENO, &"\nERROR: ABORTION OF UNDETACHED
                        THREADS!\n", 40);
        else
                write(STDOUT_FILENO, &"\n", 1);
        tcsetattr(STDIN_FILENO, TCSANOW, &ots); //povrati parametre
}

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 kontrolera diska

```
enum Disk_operations { DISK_READ, DISK_WRITE };
enum Disk_status { DISK_STARTED, DISK_ACTIVE, DISK_STOPPED };
const unsigned BLOCK_SIZE = 512;
const unsigned DISK_BLOCKS = 1000;
typedef char Disk_block[BLOCK_SIZE];

class Disk_controller {
        Disk_block* disk_space;
        unsigned accessed_last;
        unsigned transfer_time;
        Disk_operations operation_reg;
        char* buffer_reg;
        unsigned block_reg;
        Disk_status status_reg;
        bool transfer();
        Disk_controller(const Disk_controller&);
        Disk_controller& operator=(const Disk_controller&);
public:
        Disk_controller();
        ~Disk_controller();
        friend class Interrupt;
        friend class Disk_driver;
};
```

# Emulacija kontrolera diska

```
Disk_controller::Disk_controller()
: accessed_last(0), transfer_time(0), status_reg(DISK_STOPPED)
{
        disk_space = (Disk_block*) calloc(DISK_BLOCKS, sizeof(Disk_block));
}

Disk_controller::~Disk_controller()
{
        free(disk_space);
}

const int SECTORS_PER_TRACK = 10;
const int TRANSFER_TIME_AND_ROTATIONAL_DELAY = 2;
```

# Emulacija kontrolera diska

```
bool Disk_controller::transfer()
{
        bool interrupt_emulated = false;
        Disk_block* block_pointer;
        int block_distance;
        if(status_reg == DISK_STARTED) { //deo simulacije inercije rotacije diska
                status_reg = DISK_ACTIVE;
                block_distance = accessed_last - block_reg;
                if(block_distance < 0)
                        block_distance = -block_distance;
                transfer_time = TRANSFER_TIME_AND_ROTATIONAL_DELAY;
                transfer_time += block_distance / SECTORS_PER_TRACK; //vreme rotacije
                accessed_last = block_reg;
        }
        if((status_reg == DISK_ACTIVE) && (--transfer_time == 0)) {
                block_pointer = disk_space + block_reg;
                if(operation_reg == DISK_WRITE)
                        bcopy(buffer_reg, block_pointer, BLOCK_SIZE); //kopiraj bajte
                else
                        bcopy(block_pointer, buffer_reg, BLOCK_SIZE); //kopiraj bajte
                status_reg = DISK_STOPPED;
                interrupt.handler(DISK);
                interrupt_emulated = true;
        }
        return interrupt_emulated;
}

Disk_controller disk_controller;
```

# Okončanje izvršavanja konkurentnog programa

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

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

# 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 )
```

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

Primer:

```
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.

# Rukovanje pojedinim bitima memorijskih lokacija

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

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

# Rukovanje pojedinim bitima memorijskih lokacija

```
inline static int
ad__clear_bit(unsigned priority_bits, int index)
{   //poziva asm bit test and reset
        asm ("btr %1, %0"
                : "=r"(priority_bits)
                : "r"(index), "0"(priority_bits)
        );
        return priority_bits;
}
```

# Rukovanje numeričkim koprocesorom

- Preključivanje 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.

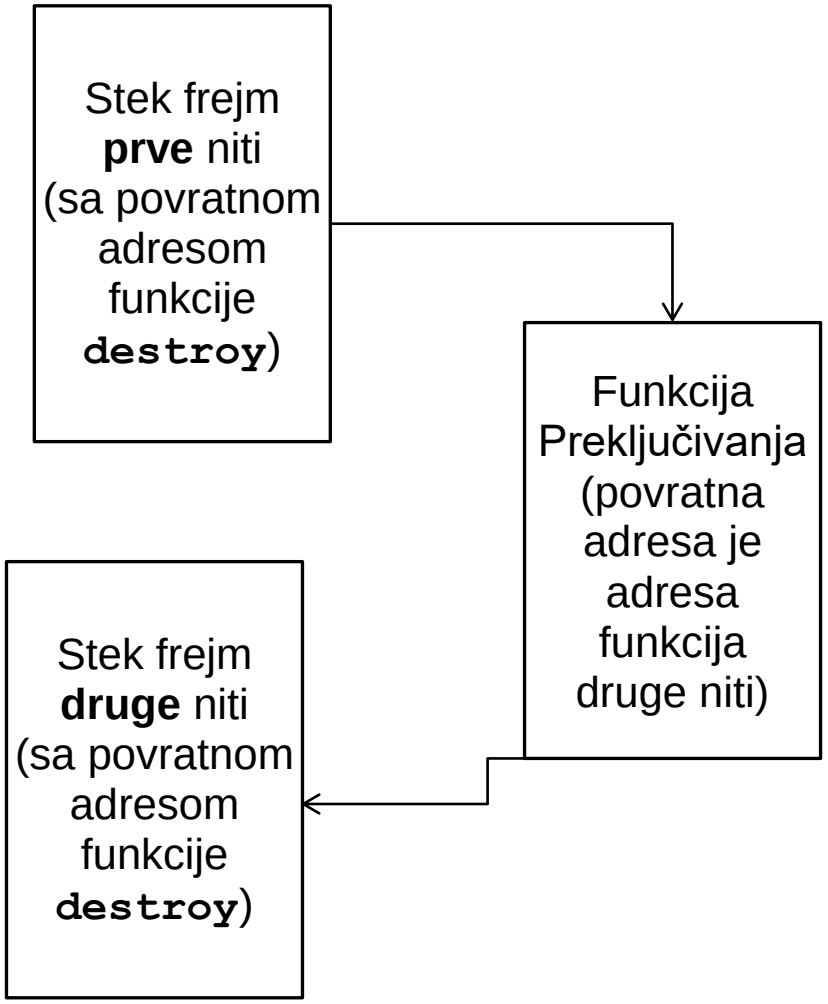
# Rukovanje numeričkim koprocesorom

```
const unsigned i387_SAVE_REGION_SIZE = 0x6c; //velicina FPU steka

class I387_npx {
        static char initial_context[i387_SAVE_REGION_SIZE];
        I387_npx(const I387_npx&);
        I387_npx& operator=(const I387_npx&);
public:
        I387_npx();
        inline void initial_context_get(Stack_item* stack_top);
};

char I387_npx::initial_context[i387_SAVE_REGION_SIZE] = {0};

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

# Rukovanje numeričkim koprocesorom

```
void I387_npx::initial_context_get(Stack_item* stack_top) {
        for(unsigned i = 0; i < i387_SAVE_REGION_SIZE; i++)
                        ((char*)stack_top)[i] = initial_context[i];
}

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.

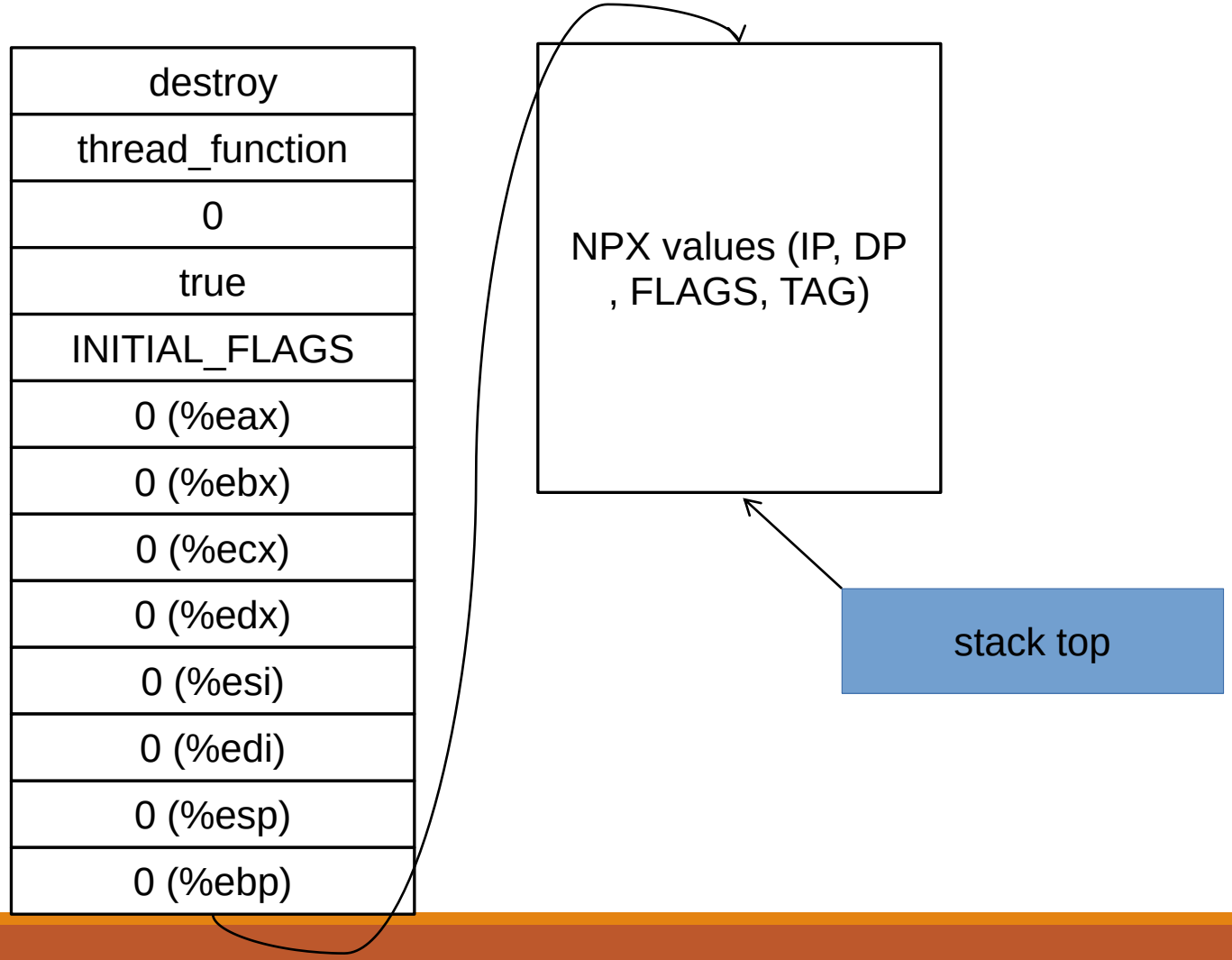
# Rukovanje stekom

- 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**

# Rukovanje stekom

```
const int INITIAL_FLAGS = 0x0200;

static inline void ad__stack_init(Stack_item** stack_top,
                                  unsigned thread_function)
{
        *(--(*stack_top)) = (Stack_item) destroy;
        *(--(*stack_top)) = (Stack_item) thread_function;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) true;
        *(--(*stack_top)) = (Stack_item) INITIAL_FLAGS;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *(--(*stack_top)) = (Stack_item) 0;
        *stack_top = (Stack_item*)(((size_t)(*stack_top)) -
                                   i387_SAVE_REGION_SIZE);
        i387_npx.initial_context_get(*stack_top);
}

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

# Rukovanje stekom



# 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`**

# Rukovanje stekom (čuvanje steka u desktriptor niti koja je bila aktivna)

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

```
.text
.align 2
.globl ad__stack_swap

ad__stack_swap:
        pushl %ebp       //cuvanje zatecenog pokazivaca stek frejma
        movl %esp,%ebp   //postavljanje novog pokazivaca frejma
        movl 8(%ebp),%edx//adresa vrha steka iz deskriptora niti sa
                         // koje se procesor prekljucuje

        mov interrupts_enabled,%eax  //na stek aktivne niti se smesta
        push %eax                    //zateceno stanje bita prekida
        pushf                        //smestanje flags registra na stek
        pusha                        //smestanje svih registara opste namene na stek
        sub i387_SAVE_REGION_SIZE,%esp
        fnsavel (%esp)   //smestanje sadrzaja reg koprocesora na stek
```

# Rukovanje stekom (prebacivanje desktiptora spremne niti u stek aktivne)

```
mov %esp,(%edx)//adresa vrha steka se smesti u deskriptor
                //do tada aktivne niti (lokacija u %edx)
mov 12(%ebp),%esp//u pokazivac steka se prebaci adresa vrha
                //steka iz deskriptora novoaktivirane niti
frstorl (%esp)   //sa novog steka se preuzmu novi sadrzaji
                          //registara numerickog koprocesora
add i387_SAVE_REGION_SIZE,%esp
popa             //sa novog steka se preuzima sadrzaj
                //registara opste namene
popf             //sa novog steka se preuzima sadrzaj
                //status registara flags
pop %eax
mov %eax,interrupts_enabled      //sa novog steka se preuzme
                                 //sadrzaj bita prekida i frejm
                                 //pointera

popl %ebp
ret
```

# Rukovanje stekom