Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1
Nastavnik: prof. dr Dragan Milićev
Školska godina: 2025/2026.

Projekat za domaći rad
Projektni zadatak

Verzija dokumenta: 1.0

Važne napomene: Pre čitanja ovog teksta, **obavezno** pročitati opšta pravila predmeta i pravila vezana za izradu domaćih zadataka! Pročitati potom ovaj tekst **u celini i pažljivo**, pre započinjanja realizacije ili traženja pomoći. Ukoliko u zadatku nešto nije dovoljno precizno definisano ili su postavljeni kontradiktorni zahtevi, student treba da uvede razumne pretpostavke, da ih temeljno obrazloži i da nastavi da izgrađuje preostali deo svog rešenja na temeljima uvedenih pretpostavki. Zahtevi su namerno nedovoljno detaljni, jer se od studenata očekuje kreativnost i profesionalni pristup u rešavanju praktičnih problema!

Sadržaj

Uvod

Cilj ovog projekta jeste realizacija malog, ali sasvim funkcionalnog jezgra (engl. *kernel*) operativnog sistema koji podržava niti (engl. *multithreaded operating system*) sa deljenjem vremena (engl. *time sharing*). U daljem tekstu, ovakav sistem biće kratko nazivan samo *jezgrom*.

U okviru ovog projekta treba realizovati alokator memorije i upravljanje nitima. Jezgro treba da obezbedi koncept niti (engl. *thread*), semafora i podršku deljenju vremena (engl. *time sharing*), kao i asinhronu promenu konteksta i preotimanje (engl. *preemption*) na prekid od tajmera i od tastature.

Jezgro treba da bude realizovano kao „bibliotečno“, tako da korisnički program (aplikacija) i samo jezgro dele isti adresni prostor, odnosno da predstavljaju statički povezan jedinstven program unapred učitan u operativnu memoriju računara. Konkurentni procesi kreirani unutar aplikacije biće zapravo samo „laki“ procesi, tj. niti (engl. *thread*) pokrenuti unutar tog programa. Ovakva konfiguracija karakteristična je za ugrađene (engl. *embedded*) sisteme, koji ne izvršavaju proizvoljne programe koji se učitavaju i izvršavaju na zahtev korisnika, već izvršavaju samo onaj program (zajedno sa operativnim sistemom) koji je već ugrađen u ciljni hardver.

Jezgro se implementira za arhitekturu procesora RISC-V i školskog računara sa ovim procesorom. Za implementaciju se može koristiti asembler za ovaj procesor i jezik C/C++. Implementirano jezgro će se izvršavati u virtuelnom okruženju – emulatoru procesora RISC-V.

Opšti zahtevi

Odnos jezgra i korisničke aplikacije

Jezgro treba realizovati na jeziku C++, uz korišćenje asemblera za ciljni procesor po potrebi. Korisnička aplikacija sadržaće test primere i biće obezbeđena kao skup izvornih fajlova koje treba prevesti i povezati sa prevedenim kôdom jezgra i datim bibliotekama `app.lib` i `hw.lib` u jedinstven program (`.exe`). Biblioteka `app.lib` sadržaće preveden i povezan kôd korisničkog test programa. Biblioteka `hw.lib` sadržaće module koji se obezbeđuju kao moduli koji pristupaju (zamišljenom, virtuelnom) hardveru, odnosno moduli od kojih će jezgro zavisiti (engl. *stub*).

Glavni program, tj. izvor kontrole toka korisničke aplikacije treba da bude u funkciji:

```
void userMain ();
```

Funkcija `main()` nema argumente niti povratnu vrednost i ostaje u nadležnosti samog jezgra, pa realizacija jezgra može da pod svojom kontrolom ima radnje koje će se izvršiti pri pokretanju programa, a zatim treba da pokrene nit nad funkcijom `userMain()`.

Odnos jezgra i sistema-domaćina

Jezgro i korisnička aplikacija treba da se posmatraju kao jedinstven izvršni program dobijen prevođenjem i statičkim povezivanjem kôda na izvornom programskom jeziku na kom su realizovani. Oni će biti pokrenuti unutar edukativnog operativnog sistema xv6 kao *sistema-domaćina*. Ovaj sistem xv6 je za ovu priliku značajno modifikovan tako što su mu izbačene mnoge funkcionalnosti (promena konteksta i raspoređivanje procesa, upravljanje memorijom, fajl podsistem, upravljanje diskom itd.). Sistem-domaćin ima ulogu samo da se sam pokrene i inicijalizuje ciljni hardver, potom napravi samo jedan proces sa virtuelnim adresnim prostorom koji zauzima celu raspoloživu fizičku memoriju, učita kôd programa koji čine realizovano jezgro i sa njim povezana aplikacija i zatim pokrene njegovo izvršavanje u kontekstu tog jedinog procesa. Osim toga, sistem-domaćin obezbeđuje i osnovne usluge hardvera: periodičan prekid od tajmera i pristup konzoli (tastaturi i ekranu). Na PC računarima i njihovim operativnim sistemima cela ova računarska platforma (procesor RISC-V, tajmer i konzola, kao i sistem-domaćin xv6) simuliraju se odgovarajućom virtuelnom mašinom (emulatorom).

Sve ovo jednostavno znači da će realizovano jezgro posmatrati svoju platformu kao jednostavan računar sa RISC-V procesorom i jedinstvenim adresnim prostorom operativne memorije (samo fizički adresni prostor) u koji je učitan izvršiv program dobijen povezivanjem jezgra i aplikacije, kao što je to slučaj kod ugrađenih sistema.

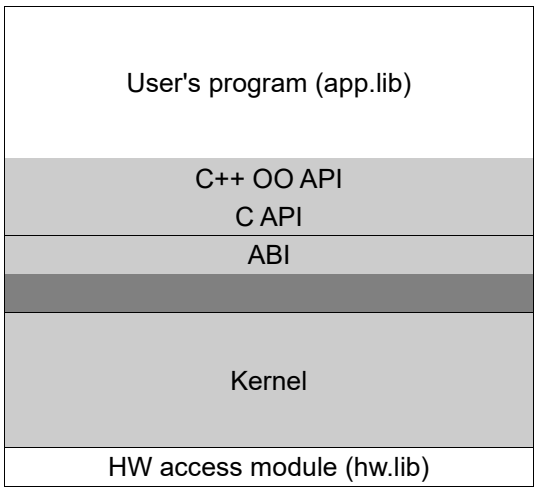
Jezgro i korisnička aplikacija moraju da se kao program regularno završe, naravno ukoliko u samoj korisničkoj aplikaciji nema neregularnosti. To znači da po završetku svih niti pokrenutih u korisničkoj aplikaciji ceo program treba regularno da se završi. Test primeri ispitivača biće regularni, pa svaki neregularan završetak programa znači neregularnost u samom jezgru, osim ako za neki konkretan test primer nije drugačije naznačeno.

U realizaciji jezgra nije dozvoljeno koristiti usluge operativnog sistema-domaćina niti operativnog sistema PC računara na kom se sve ovo izvršava, a koje se odnose na koncepte niti ili procesa, semafora, prekida, sinhronizacije i komunikacije između niti ili procesa, itd. Drugim rečima, sve zahtevane koncepte i funkcionalnosti potrebno je realizovati u potpunosti samostalno i „od nule“.

Osim biblioteka eksplicitno navedenih ovde koje će biti posebno pripremljene, u realizaciji jezgra ne treba koristiti nikakve druge, pa ni standardne C/C++ biblioteke, statičke ili dinamičke, jer one po pravilu sadrže sistemske pozive operativnog sistema-domaćina.

Interfejs jezgra

Jezgro treba da obezbedi tri vrste interfejsa prema korisničkom programu, slojevito organizovane kao na sledećoj slici. Neki sloj *A,* koji je na slici prikazan iznad drugog sloja *B*, koristi usluge tog drugog sloja *B* da bi ispunio svoje odgovornosti prema sloju prikazanom iznad sloja *A*.



Zadatak je realizovati sve osenčene slojeve softvera. Modul za pristup hardveru (`hw.lib`) biće raspoloživ i dat u obliku statičke biblioteke. Korisnički program (`app.lib`) za testiranje biće dat kao statička biblioteka (`app.lib`), a student treba da napravi i svoje primere korisničkog programa koji testiraju jezgro. Svi prikazani slojevi statički su povezani u jedinstven izvršni program.

Programski kôd svih prikazanih slojeva izvršava se u istom (jedinstvenom) adresnom prostoru. Programski kôd jezgra i modula za pristup hardveru (softver ispod tamne debele linije) izvršava se u privilegovanom (sistemskom) režimu rada ciljnog procesora, dok se slojevi iznad jezgra izvršavaju u neprivilegovanom (korisničkom) režimu rada procesora. Jezgro (i njegova funkcija `main`) će inicijalno biti pokrenuti u sistemskom režimu.

ABI (engl. *application binary interface*) je binarni interfejs sistemskih poziva koji se vrše pomoću softverskog prekida ciljnog procesora. Ovaj sloj obezbeđuje prenos argumenata sistemskog poziva preko registara procesora, prelazak u privilegovani režim rada procesora i prelazak na kôd jezgra.

C API (engl. *application programming interface*) je klasičan, proceduralan (ne objektno orijentisan) programski interfejs sistemskih poziva implementiran kao skup funkcija. Ove funkcije u svojoj implementaciji mogu imati jedan sistemski poziv, više njih ili nijedan sistemski poziv iz sloja ABI, u zavisnosti od svoje uloge. Tako su ove funkcije zapravo omotač (engl. *wrapper*) oko interfejsa ABI.

C++ API je objektno orijentisan API koji pruža objektni pogled na koncepte koje jezgro podržava. Implementiran je kao jednostavan objektno orijentisan omotač oko funkcija iz sloja C API pisan na jeziku C++.

Detalji ovih interfejsa dati su u narednim odeljcima.

C API

Funkcije ovog interfejsa opisane su u sledećoj tabeli, a deklaracije su date u fajlu `syscall_c.hpp`.

Broj	Potpis	Objašnjenje
0x01	`void* mem_alloc (size_t size);`	Alocira prostor od (najmanje) `size` bajtova memorije, zaokruženo i poravnato na blokove veličine `MEM_BLOCK_SIZE`. Vraća pokazivač na deo alociranog prostora od kojeg do kraja datog prostora ima (najmanje) `size` bajtova u slučaju uspeha, a *null* u slučaju neuspeha. `MEM_BLOCK_SIZE` je celobrojna konstanta veća od ili jednaka 64, a manja od ili jednaka 1024.
0x02	`int mem_free (void*);`	Oslobađa prostor prethodno alociran pomoću `mem_alloc`. Vraća 0 u slučaju uspeha, negativnu vrednost u slučaju greške (kôd greške). Argument mora imati vrednost vraćenu iz `mem_alloc`. Ukoliko to nije slučaj, ponašanje je nedefinisano: jezgro može vratiti grešku ukoliko može da je detektuje ili manifestovati bilo kakvo drugo predvidivo ili nepredvidivo ponašanje.

0x11	`class _thread;`
	`typedef _thread* thread_t;`
	
	`int thread_create (`
	`  thread_t* handle,`
	`  void(*start_routine)(void*),`
	`  void* arg`
	`);`
		Pokreće nit nad funkcijom `start_routine`, pozivajući je sa argumentom `arg`. U slučaju uspeha, u `*handle` upisuje „ručku“ novokreirane niti i vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).
		„Ručka“ je interni identifikator koji jezgro koristi da bi identifikovalo nit (pokazivač na neku internu strukturu/objekat jezgra čije je ime sakriveno iza sinonima `_thread`).

0x12	`int thread_exit ();`	Gasi tekuću nit. U slučaju neuspeha vraća negativnu vrednost (kôd greške).
0x13	`void thread_dispatch ();`	Potencijalno oduzima procesor tekućoj i daje nekoj drugoj (ili istoj) niti.

0x21	`class _sem;`
	`typedef _sem* sem_t;`
	
	`int sem_open (`
	`  sem_t* handle,`
	`  unsigned init`
	`);`
		Kreira semafor sa inicijalnom vrednošću `init`. U slučaju uspeha, u `*handle` upisuje ručku novokreiranog semafora i vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).
		„Ručka“ je interni identifikator koji jezgro koristi da bi identifikovalo semafore (pokazivač na neku internu strukturu/objekat jezgra čije je ime sakriveno iza sinonima `_sem`).

0x22	`int sem_close (sem_t handle);`	Oslobađa semafor sa datom ručkom. Sve niti koje su se zatekle da čekaju na semaforu se deblokiraju, pri čemu njihov `wait` vraća grešku. U slučaju uspeha vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).
0x23	`int sem_wait (sem_t id);`	Operacija `wait` na semaforu sa datom ručkom. U slučaju uspeha vraća 0, a u slučaju neuspeha, uključujući i slučaj kada je semafor dealociran dok je pozivajuća nit na njemu čekala, vraća negativnu vrednost (kôd greške).
0x24	`int sem_signal (sem_t id);`	Operacija `signal` na semaforu sa datom ručkom. U slučaju uspeha vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).
0x25	`int sem_wait_n(sem_t id, unsigned n);`	Operacija `wait` na semaforu sa datom ručkom, koja zahteva **n** jedinica resursa. Ukoliko je trenutna vrednost semafora veća ili jednaka **n**, vrednost se umanjuje za **n** i funkcija se odmah uspešno završava. U suprotnom, pozivajuća nit se blokira dok ne postane moguće izvršiti operaciju (ili dok semafor ne bude dealociran). U slučaju uspeha vraća 0, a u slučaju neuspeha, uključujući i slučaj kada je semafor dealociran dok je nit čekala, vraća negativnu vrednost (kôd greške).

0x26	`int sem_signal_n(sem_t id, unsigned n);`	Operacija *signal* na semaforu sa datom ručkom, koja povećava vrednost semafora za **n**. Ukoliko postoje niti koje čekaju na semaforu, može doći do deblokiranja jedne ili više niti. U slučaju uspeha vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).

0x31	`typedef unsigned long time_t;`
	
	`int time_sleep (time_t);`
		Uspavljuje pozivajuću nit na zadati period u internim jedinicama vremena (periodama tajmera). U slučaju uspeha vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).

0x41	`const int EOF = -1;`
	
	`char getc ();`
		Učitava jedan znak iz bafera znakova učitanih sa konzole. U slučaju da je bafer prazan, suspenduje pozivajuću nit dok se znak ne pojavi. Vraća učitani znak u slučaju uspeha, a konstantu `EOF` u slučaju greške.

0x42	`void putc (char);`	Ispisuje dati znak na konzolu.

Nit se kreira sa podrazumevanom veličinom steka (`DEFAULT_STACK_SIZE`) i podrazumevanom veličinom vremenskog odreska (`DEFAULT_TIME_SLICE`). Ove konstante deklarisane su u `hw.h`, a definisane u `hw.lib`:

```
extern const size_t DEFAULT_STACK_SIZE;
extern const time_t DEFAULT_TIME_SLICE;
```

Memorijski prostor koji je slobodan za alokaciju počinje od adrese `HEAP_START_ADDR`, a završava se na adresi `HEAP_END_ADDR-1`. Ove konstante, kao i konstanta `MEM_BLOCK_SIZE`, deklarisane su u `hw.h`, a definisane u `hw.lib`:

```
extern const void*  HEAP_START_ADDR, HEAP_END_ADDR;
extern const size_t MEM_BLOCK_SIZE;
```

ABI

Sistemski poziv u ovom sloju obavlja se softverskim prekidom (odgovarajućom instrukcijom procesora). Parametri se u sistemski poziv prenose kroz registre procesora na sledeći način:

- `a0`: kôd sistemskog poziva, jednak broju iz prve kolone tabele date za C API;
- `a1, a2`, ...: parametri sistemskog poziva, redom po potpisu iz C API-a sleva nadesno;
- `a0`: povratna vrednost.

Svi potpisi navedenih sistemskih poziva iz C API-a se u potpunosti preslikavaju navedenom konvencijom na odgovarajuće ABI pozive i imaju istu semantiku, osim sledećih izuzetaka:

- Sistemski poziv broj 0x01, `mem_alloc`, ima isti potpis, samo što parametar (`size`) izražava veličinu prostora u blokovima, a ne u bajtovima. To znači da funkcija `mem_alloc` iz C API-a treba da zadatu vrednost u bajtovima zaokruži na cele blokove (tako da zahtevani prostor stane u te blokove) i izrazi u blokovima pre nego što izvrši ovaj sistemski poziv ABI-a.
- Sistemski poziv broj 0x11, `thread_create`, ima ABI potpis ekvivalentan sledećem C potpisu:

`int thread_create (`
`  thread_t* handle,`
`  void(*start_routine)(void*),`
`  void* arg,`
`  void* stack_space`
`);`
	Kreira nit nad funkcijom `start_routine`, pozivajući je sa argumentom `arg`, smeštajući stek te niti u već odvojen prostor na čiju poslednju lokaciju ukazuje `stack_space`. U slučaju uspeha, u `*handle` upisuje ručku novokreirane niti i vraća 0, a u slučaju neuspeha vraća negativnu vrednost (kôd greške).

To znači da sistemski poziv u ABI sloju kreira nit sa zadatom pozicijom steka, pa funkcija `thread_create` iz C API-a treba najpre da alocira stek (sistemskim pozivom `mem_alloc`) pre nego što izvrši ovaj sistemski poziv ABI-a.

C++ API

Klase ovog interfejsa treba definisati u fajlu `syscall_cpp.hpp`. Interfejsi ovih klasa imaju sledeći oblik:

```
#ifndef _syscall_cpp
#define _syscall_cpp

#include "syscall_c.hpp"

void* ::operator new (size_t);
void  ::operator delete (void*);

class Thread {
public:
  Thread (void (*body)(void*), void* arg);
  virtual ~Thread ();

  int start ();

  static void dispatch ();
  static int sleep (time_t);

protected:
  Thread ();
  virtual void run () {}

private:
  thread_t myHandle;
  void (*body)(void*); void* arg;
};


class Semaphore {
public:

  Semaphore (unsigned init = 1);
  virtual ~Semaphore ();

  int wait ();
  int signal ();

private:
  sem_t myHandle;

};


class PeriodicThread : public Thread {
public:
  void terminate ();

protected:
  PeriodicThread (time_t period);
  virtual void periodicActivation () {}

private:
  time_t period;
};


class Console {
public:
  static char getc ();
  static void putc (char);
};
```

```
#endif
```

Globalne operatorske funkcije `new` i `delete` treba implementirati tako da obmotavaju sistemske pozive `mem_alloc` i `mem_free`, respektivno. Na taj način operatori `new` i `delete` jezika C++ biće preusmereni na ove sistemske pozive. Apstraktna klasa `PeriodicThread` služi za podršku periodičnim nitima sa periodom zadatom konstruktorom. Korisnik treba da izvede klasu iz ove i redefiniše polimorfnu operaciju `periodicActivation` u kojoj zadaje operaciju za jednu periodičnu aktivaciju ove niti. Klasa `Console` je uslužna, fasadna klasa koja samo obezbeđuje odgovarajući prostor imena za operacije koje obmotavaju sistemske pozive za pristup konzoli. Značenje ostalih klasa i njihovih operacija je očigledno ili je dato na nastavi. U slučaju da je redefinisana operacija `run` u izvedenoj klasi, ali i pozvan konstruktor osnovne klase koji prima pokazivač na funkciju, što je nepredviđen i nekorektan način korišćenja klase `Thread` iz izvedene klase, ponašanje treba da bude kao da je samo postavljen pokazivač na funkciju. Drugim rečima, ukoliko je konstruktorom postavljen pokazivač na funkciju, operaciju `run` treba ignorisati u svakom slučaju.

Postojanje destruktora u interfejsu označava da se dati objekti mogu uništavati i da je eventualno potrebno obezbediti odgovarajuću sinhronizaciju ili dealokaciju sistemskim pozivom. Greške vraćene iz sistemskih poziva signalizirati povratnim vrednostima (gde je to moguće) na isti način kao i u C API-ju.

Zadatak je u potpunosti implementirati sve ove klase i operacije. Kako bi korisnički kôd koji se daje u biblioteci `app.lib` bio kompatibilan sa ovim definicijama, definicije ovih klasa ne smeju se proširivati bilo kakvim nestatičkim podacima članovima, osnovnim klasama, niti se sme menjati redosled i skup virtuelnih funkcija članica. Ne smeju se menjati ni potpisi postojećih funkcija članica.

Opis platforme

Kratak prikaz arhitekture i asemblera procesora RISC-V

Razmatra se varijanta RV64IMA procesora RISC-V. Zvanična i potpuna specifikacija RISC-V arhitekture, "The RISC-V Instruction Set Manual", data je na sajtu: https://riscv.org/technical/specifications/

Osnovne karakteristike arhitekture

- 64-bitni troadresni RISC procesor sa *load*/*store* arhitekturom. Svi registri su 64-bitni.
- Adresibilna jedinica je bajt. Podaci koji se prenose iz memorije i u memoriju su širine 1, 2, 4 ili 8 bajtova. Podrazumevano je niži bajt na nižoj adresi (engl. *little endian*), mada se to može konfigurisati drugačije.
- Instrukcije su širine 32 bita.
- Dva režima rada procesora koji se koriste u projektu (treći je irelevantan za projekat): korisnički i sistemski.
- Registri dostupni i u korisničkom i u sistemskom režimu[1][2]:
  - `sp`: pokazivač steka;
  - `ra`: registar za povratnu adresu pozvanog potprograma;
  - `s0..s11`: registri opšte namene čiju će vrednost očuvati pozvani potprogram;
  - `a0..a7`: registri za prenos argumenata i povratne vrednosti potprograma;
  - `t0..t6`: privremeni registri;
  - `zero`: „registar" ožičen na nulu (upis nema efekta, čitanje uvek vraća 0);
  - `gp` i `tp` su registri posebne namene.
- Registri dostupni samo u sistemskom režimu:
  - `sstatus`: statusni registar;
  - `sip` i `sie`: registri za prekide;
  - `sscratch`: privremeni registar;
  - `sepc`: sačuvana vrednost registra `pc` u korisničkom režimu;
  - `scause`: opis razloga za prelazak u sistemski režim;
  - `stvec`: adresa prekidne rutine, poravnata na 4 bajta;
- Registar `pc` ukazuje na tekuću instrukciju i nije direktno dostupan programu za upis.
- Stek[3]:
  - raste ka nižim adresama;
  - `sp` pokazuje na poslednju zauzetu lokaciju;
  - vrednost registra `sp` mora biti deljiva sa 16.

---

1 Sve ove registre prevodilac potencijalno koristi, pa na to treba obratiti pažnju pri čuvanju konteksta procesora, posebno kod asinhrone promene konteksta.

2 Svi ovi registri su zapravo potpuno ravnopravni i jednaki (osim registra ožičenog na nulu u koji upis nema efekta) registri opšte namene koji čine registarski fajl ovog procesora (nazivaju se `x0..x31`), prema opštoj filozofiji RISC arhitektura. Njihova upotreba za odgovarajuću specifičnu namenu koja je ovde opisana je stvar konvencije koju prevodilac za C/C++ poštuje. Navedeni mnemonici su zapravo samo sinonimi za odgovarajuće registre koje asembler i prevodilac tretiraju na odgovarajući način.

3 Ovaj procesor zapravo uopšte ne poznaje koncept steka, niti ga na bilo koji način koristi, eksplicitno (nema specijalizovane operacije tipa *push* i *pop*) ili implicitno (ne koristi ga čak ni prilikom obrade prekida/izuzetaka/sistemskih poziva). Organizacija steka je u isključivoj nadležnosti prevodioca ili programera na asembleru.

- RV64IMA podržava samo sledeće kategorije instrukcija opšte RISC-V arhitekture (ostale kategorije nisu podržane):
  - I - celobrojne instrukcije
  - M - množenje/deljenje celih brojeva
  - A - atomične instrukcije, za atomičan složeni pristup memoriji; ova grupa instrukcija, između ostalog, sadrži i instrukcije za *spin lock*, ali kako se ovaj projekat ograničava na jednoprocesorski sistem, one nisu neophodne.
- Usled odsustva F, D i Q ekstenzija , ovaj procesor nema direktnu podršku za aritmetiku u pokretnom zarezu (engl. *Floating point*), pa stoga ni u jezgru, ni u korisničkom programu ne treba koristiti ovu aritmetiku.[4]

Pregled nekih instrukcija i načina adresiranja

- *Load*/*store* instrukcije:

```
l{b|h|w|d} reg, offset(reg)
s{b|h|w|d} reg, offset(reg)
```

  - sufiks `b`, `h`, `w` i `d` definiše širinu podatka koji se prenosi: 1B, 2B, 4B i 8B, respektivno;
  - `offset(reg)`: definiše adresu podatka - registarsko indirektno adresiranje sa označenim pomerajem širine 12 bita.
- `call`: pseudoinstrukcija za poziv funkcije:
  - operand je ime funkcije (npr. `call f`);
  - prevodilac prevodi pseudoinstrukciju u niz instrukcija koje vrše skok na funkciju;
  - obično se prevodi u relativan skok (u odnosu na registar `pc`) pomoću instrukcija `auipc` i `jalr`; instrukcija `auipc` vrednost registra `pc` sabira sa zadatim označenim pomerajem i dobijenu vrednost upisuje u odredišni registar; instrukcija `jalr` vrši skok na adresu koja je zapisana u odredišnom registru, uz opcioni označeni pomeraj, i prilikom skoka pamti povratnu adresu sledeće instrukcije u registar `ra`;
- `csr`: instrukcije za pristup sistemskim registrima:
  - `csrr reg, sreg`: upis vrednosti sistemskog registra `sreg` u korisnički registar `reg`;
  - `csrw reg, sreg`: upis vrednosti korisničkog registra `reg` u sistemski registar `sreg`;
  - `csrrw reg1, sreg, reg2`: u sistemski registar `sreg` se upisuje podatak iz korisničkog registra `reg2`, a stara vrednost registra `sreg` se upisuje u korisnički registar `reg1`.
- Asemblerski kôd se može pisati u posebnim fajlovima. Asemblerski fajl mora imati ekstenziju `.S`. U ovakvom fajlu kôd se piše isto kao primer asemblerskog kôda unutar C funkcije datog u narednom odeljku.
- Pojedinačne instrukcije se mogu ugraditi u C/C++ kôd pomoću asemblerskog bloka. U nastavku su dati primeri `asm` blokova za `gcc` prevodilac za čitanje i upis registara dostupnih u sistemskom režimu, kao i instrukciju koja ne pristupa C/C++ podacima:

[4] Ova aritmetika može biti podržana i softverskim bibliotekama, pa prevodilac može da prevodi aritmetičke operacije u pokretnom zarezu u pozive potprograma koji tu aritmetiku obavljaju softverski, ali to nije od interesa za ovaj projekat. Jednostavno, ne treba koristiti aritmetičke operacije sa racionalnim brojevima, niti za tim ima potrebe.

```
uint64 x;

// Upis vrednosti registra sstatus u automatsku promenljivu x
asm volatile("csrr %0, sstatus" : "=r" (x));

// Upis vrednosti automatske promenljive x u registar sstatus
asm volatile("csrw sstatus, %0" : : "r" (x));

// Instrukcija koja učitava podatak iz memorije u registar s0
asm volatile("ld s0, 8(sp)");
```

Konvencije C/C++ prevodioca za poziv funkcije

- Pozvana funkcija je dužna da obezbedi da nakon povratka iz nje registri opšte namene (`s0..s11`) i registar `sp` imaju istu vrednosti kao i pre poziva funkcije.
- Povratna adresa se pamti u registru `ra`.
- Parametri se prenose preko registara `a0…a7` (redom sleva nadesno u potpisu funkcije), a preko memorije ukoliko ne mogu da se smeste u registar.
- Povratna vrednost se prenosi preko registra `a0` i po potrebi `a1`.
- Pseudoinstrukcija `ret` upisuje vrednost registra `ra` u registar `pc`.
- Primer prevoda funkcije:

```
int f (int a, int b) {
  if (a == 0)
    return 0;
  return f(a - 1, b) + b;
}
```

```
f:
   bnez a0, offset  // bnez – branch on not equal to zero
   ret              // u a0 je povratna vrednost
offset:
   addi sp, sp, -32 // prostor na steku za sačuvane i privremene vrednosti
   sd ra, 24(sp)    // čuva povratnu adresu pre poziva funkcije
   sd s0, 16(sp)    // čuva registre iz grupe s0..s11 koje koristi
   sd s1, 8(sp)
   addi s0, sp, 32  // s0 se koristi kao „frame pointer“[5]
   mv s1,a1         // čuva vrednost a1 u s1 jer može da se promeni
   addiw a0, a0 ,-1 // argumenti poziva funkcije se stavljaju u a0 i a1
   // Naredne dve instrukcije rezultat pseudoinstrukcije call f
   auipc ra, 0x0
   jalr -36(ra)     // skok na funkciju f uz čuvanje povratne adrese u ra
   addw a0, a0, s1
   ld ra, 24(sp)    // restaurira se vrednost ra radi povratka
   ld s0, 16(sp)    // restaurira registre iz grupe s0..s11
   ld s1, 8(sp)
   addi sp, sp, 32  // sp se vraća na prethodno stanje
   ret
```

[5] *Frame pointer* je pokazivač na oblast steka od koje počinje aktivacioni blok sa automatskim podacima tekućeg potprograma. Jedan registar se odabira za tu namenu i na početku izvršavanja potprograma postavlja na odgovarajuću vrednost koja ukazuje na vrh steka. Prevodioci često koriste ovu tehniku da bi adresirali automatske podatke na steku uvek relativno u odnosu na ovaj pokazivač koji se tokom izvršavanja potprograma ne menja, kako bi koristili konstantne pomeraje, a ne u odnosu na *sp*, koji može da se menja tokom izvršavanja potprograma.

Obrada sistemskih poziva, izuzetaka i prekida

- U sistemski režim prelazi se instrukcijom softverskog prekida `ecall`, izuzetkom ili spoljašnjim prekidom.
- Prilikom obrade sistemskog poziva, izuzetka ili prekida, procesor radi sledeće:
  - vrednost registra `pc` upisuje u registar `sepc` (adresa instrukcije `ecall` ili adresa prve neizvršene/prekinute instrukcije);
  - u registar `sstatus` upisuje sledeće vrednosti:
    - u bit `SPP` (bit 8) vrednost koja pokazuje iz kog režima se dogodio skok (vrednost 0 – korisnički režim, vrednost 1 – sistemski režim);
    - u bit `SIE` (bit 1) nulu, čime se maskiraju spoljašnji prekidi; u korisničkom režimu se ovaj bit ignoriše – prekidi su podrazumevano dozvoljeni;
    - u bit `SPIE` (bit 5) prethodnu vrednost bita `SIE`.
  - u registar `scause` upisuje sledeće vrednosti:
    - u bit najveće težine (BNT) informaciju o tome da li se dogodio spoljašnji prekid ili nije;
    - u ostale bite upisuje razlog; sledeća tabela prikazuje moguće razloge (prikazani su samo najznačajniji):

BNT	Vrednost	Opis
1	1	Softverski prekid iz trećeg, najprivilegovanijeg režima rada procesora[6]
1	9	Spoljašnji hardverski prekid
0	2	Ilegalna instrukcija
0	5	Nedozvoljena adresa čitanja
0	7	Nedozvoljena adresa upisa
0	8	`ecall` iz korisničkog režima
0	9	`ecall` iz sistemskog režima

  - registar `stvec` sadrži oznaku režima `MODE` (definisan pomoću najniža dva bita) i baznu adresu `BASE` (preostali biti predstavljaju viših 30 bita bazne adrese); vrednost 0 za `MODE` označava direktni režim, procesor skače na adresu `BASE` za instrukciju `ecall`, izuzetke i spoljašnje prekide[7], a dok vrednost 1 označava vektorski režim, u kom procesor skače na adresu `BASE` samo za instrukciju `ecall` i izuzetke, dok za spoljašnje prekide skače na adresu dobijenu sabiranjem `BASE` i broja datog spoljašnjeg prekida pomnoženog sa četiri.
- Sistemski registri se čitaju pomoću instrukcije `csr`, koja je dostupna samo u sistemskom režimu.

[6] Ovaj procesor poseduje dva donekle različita načina izazivanja softverskog prekida. Jedan je instrukcijom `ecall` koja je namenjena za sistemske pozive iz korisničkog ili sistemskog režima rada procesora, dok se drugi mehanizam pokreće nešto drugačije i namenjen je za korišćenje samo u trećem, najprivilegovanijem režimu rada procesora koji u ovom projektu nije vidljiv studentima i ne treba ga razmatrati. Pristup hardveru tajmera moguć je samo iz tog trećeg režima rada procesora, a njegov hardverski spoljašnji prekid obrađuje se u tom režimu. Da bi se studentima olakšalo korišćenje, prekid od tajmera prosleđuje se jezgru kao pomenuti drugi tip softverskog prekida, koji se obrađuje principijelno na isti način kao i `ecall`. Da bi se prepoznao prekid od tajmera, dovoljno je prepoznati ovaj razlog prekida – on će poticati samo od tajmera.

[7] To znači da ovaj procesor ima jednu jedinstvenu prekidnu rutinu za sve vrste prekida.

- Povratak iz sistemskog režima radi se instrukcijom `sret`, dostupnom samo u sistemskom režimu. Ova instrukcija ima sledeće dejstvo:
  - režim u koji se prelazi je definisan bitom `SPP`;
  - bit `SIE` dobija vrednost bita `SPIE`;
  - registar `pc` dobija vrednost registra `sepc`.
- Ostali registri se ne čuvaju hardverski, već je to odgovornost prekidne rutine.
- Registar `sip` sadrži informaciju o tome koji zahtevi za prekid su trenutno aktivni. Bit `SSIP` (bit 1) označava da postoji zahtev za softverski prekid. U `SSIP` bit može da se upiše vrednost. Upis jedinice postavlja zahtev za softverskim prekidom, dok upis nule označava da se softverski prekid obradio (ovo svakako treba uraditi nakon obrade prekida). Bit `SEIP` (bit 9) označava da postoji zahtev za spoljašnji hardverski prekid.
- Registar `sie` je registar za maskiranje prekida, pa sadrži informaciju o tome koji prekidi su dozvoljeni. Vrednost ovog registra se uzima u obzir i prilikom izvršavanja u korisničkom režimu. Bit `SSIE` (bit 1) označava da li su dozvoljeni softverski prekidi. Bit `SEIE` (bit 9) označava da li su dozvoljeni spoljašnji hardverski prekidi. Ukoliko se program izvršava u sistemskom režimu i bit `SIE` u registru `sstatus` ima vrednost 0, vrednost registra `sie` se ignoriše.
- Prekid od tajmera je realizovan kao softverski prekid[8]. Prekid od tajmera može se prepoznati po vrednosti 1 samo u bitima najmanje i najveće težine u registru `scause`. Tajmer generiše prekid deset puta u sekundi.
- Prekid od konzole je realizovan kao spoljašnji hardverski prekid. Postoji kontroler prekida preko kog se može dobiti informacija o tome koji uređaj je generisao prekid. Za to služi C funkcija `plic_claim` čija je deklaracija data u zaglavlju `hw.h`. Povratna vrednost ove funkcije je broj prekida. Prekid od konzole ima broj 10 (`0x0a`). Nakon obrade prekida, kontroler prekida treba obavestiti da je prekid obrađen i to putem funkcije `plic_complete` čija je deklaracija data u zaglavlju `hw.h`. Jedini parametar ove funkcije je broj prekida koji je obrađen.

Konzola

- Konzola je ovde eksterni terminal sa kojim računar komunicira serijskom vezom[9] (tzv. UART protokol). Serijsku vezu sa konzolom ostvaruje kontroler serijske veze. Zato program ovde zapravo interaguje samo sa kontrolerom te serijske veze.
- Kontroler serijske veze ima svoj interni bafer za prijem podataka sa konzole, pri čemu podatak stiže na svaki taster pritisnut na tastaturi. Dakle, na pritisnut taster (odnosno na prijem podatka sa serijske veze), odnosno na pojavu prvog znaka u hardverskom baferu kontrolera, kontroler generiše prekid, a onda se znakovi mogu čitati iz tog bafera sve dok je bit spremnosti postavljen. Kad to više nije tako, taj bafer je prazan i znakova više nema. Pojava sledećeg naleta znakova će opet izazvati jedan prekid (i postavljen bit spremnosti) i bujicu od nekoliko znakova.
- Slično važi i za obrnuti smer slanja podataka na konzolu: kada je kontroler spreman za slanje podataka, on generiše prekid. Tada se podaci mogu prenositi u kontroler sve dok je bit spremnosti postavljen.
- Kontroler konzole generiše isti prekid i kad je spreman za prijem podatka za slanje na konzolu i kad ima znak sa tastature spreman za čitanje.

---

[8] Ovakva predstava je posledica prilagođenja okruženja samo za potrebe ovog projekta, radi jednostavnijeg korišćenja. Da bi se prepoznao prekid od tajmera, dovoljno je prepoznati ovaj razlog prekida – on će poticati samo od tajmera. Videti objašnjenje u prethodnoj fusnoti.

[9] Serijska veza označava komunikacionu liniju preko koje se podaci prenose redom bit po bit.

- Kontroler poseduje tri registra: jedan statusni, jedan za prijem podataka i jedan za slanje podataka. Registri za podatke su veličine jednog bajta. Adrese registara su date kao konstante u zaglavlju `hw.h`: `CONSOLE_STATUS`, `CONSOLE_TX_DATA` i `CONSOLE_RX_DATA`. U statusnom registru bit na poziciji 0 označava da se iz kontrolera konzole može pročitati podatak koji je stigao od konzole, dok bit na poziciji 5 označava da kontroler konzole može primiti jedan podatak za slanje na konzolu. U okviru jednog prekida mogu se prebacivati podaci sve dok su odgovarajući statusni biti na jedinici.

Zaustavljanje emulatora RISC-V procesora iz programskog koda

Emulator RISC-V procesora je moguće zaustaviti izvršavanjem odgovarajućih instrukcija. Upisom 32-bitne vrednosti `0x5555` na adresu `0x100000` emulator RISC-V procesora se zaustavlja. Na ovaj način je moguće zaustaviti proces emulatora nakon što završi korisnički program.

Uputstva za razvojno okruženje

Razvojno okruženje je dato u okviru virtuelne mašine koja se može pokrenuti na PC računaru amd64 arhitekture pod operativnim sistemom Windows ili Linux uz pomoć programa VMWare Workstation Player (besplatno je dostupan na sajtu proizvođača za nekomercijalnu upotrebu). Virtuelna mašina je dostupna na sajtu predmeta u okviru sekcije „Projektni zadatak“. Ista virtuelna mašina će biti dostupna i na laboratorijskim računarima tokom odbrane projekta. Za vreme odbrane projekta veza ka internetu u laboratoriji je isključena, pa samim tim student nije u mogućnosti da dodatno instalira softver na virtuelnoj mašini, već samo može koristi ono što inicijalno postoji na virtuelnoj mašini koja je data na sajtu predmeta.

Preporučeno razvojno okruženje je CLion, koje je i instalirano na virtuelnoj mašini. Licencu za upotrebu u akademske svrhe studenti mogu da traže od proizvođača ako se registruju sa punom studentskom imejl adresom (treba koristiti domen student.etf.bg.ac.rs). Na računarima u laboratoriji u kojima se brani projekta obezbeđene su plivajuće (engl. *floating*) licence koje studenti mogu koristiti kada nemaju pristup internetu.

Na sajtu predmeta, u sekciji „Projektni zadatak“ obezbeđena je arhiva sa osnovnim stvarima potrebnim za početak izrade projekta. U arhivi se nalazi direktorijum u kome se nalaze obezbeđene biblioteke (`hw.lib`, `mem.lib` i `console.lib`), odgovarajuća zaglavlja, fajl `Makefile`, pomoćni fajlovi za povezivanje i debagovanje, kao i direktorijumi `src` i `h` u koje student treba da ubaci svoj programski kôd. Nakon raspakivanja arhive, dati direktorijum se može učitati u razvojno okruženje CLion pomoću opcije „Open project“ iz menija. Dalje se razvojno okruženje može koristi za unos kôda, pokretanje i debagovanje, kao i za razvoj bilo kod drugog softvera uz neka dodatna podešavanja koja će biti opisana u sledećoj sekciji.

Program se prevodi i njegovo izvršavanje pokreće pomoću komande `make`[10] uz pomoć datog fajla `Makefile`. Sve je podešeno tako da se studentski kôd koji se nalazi u direktorijumu `src` prevodi na odgovarajući način. Fajlovi sa ekstenzijom `.cpp` se prevode kao C++ kôd, fajlovi sa ekstenzijom `.S` se prevode kao asemblerski kôd. Zaglavlja se uzimaju iz direktorijuma `h`. Nakon prevođenja se svi objektni fajlovi i sve biblioteke uvezuju u jedan binarni izvršni fajl pod nazivom `kernel` koji sadrži mašinski kôd za procesor RISC-V. Student ne treba da menja `Makefile`, osim u slučaju kada ne želi da koristi neku od obezbeđenih biblioteka (`mem.lib` ili `console.lib`). U fajlu `Makefile` postoji linija:

```
LIBS = lib/mem.lib lib/hw.lib lib/console.lib
```

[10] Komanda `make` se detaljno izučava na predmetu Praktikum iz operativnih sistema. Na sajtu tog predmeta se nalaze materijali koji objašnjavaju komandu `make`.

Ukoliko ne treba koristiti neku biblioteku, iz te linije je samo treba ukloniti (npr. ako nije potrebna biblioteka `mem.lib` treba obrisati `lib/mem.lib`).

Komandu `make` student može pokrenuti iz razvojnog okruženja CLion pomoću opcije „Make“ iz menija. Prilikom pokretanja potrebno je dostaviti cilj (engl. *target*) komandi `make`. Student može koristiti tri cilja: `qemu` za prevođenje i pokretanje, `qemu-gdb` za prevođenje i pokretanje u režimu debagovanja i `clean` za čišćenje[11] projekta, čime će se ukloniti svi fajlovi koji su rezultati prevođenja i povezivanja.

Detaljno i ilustrovano uputstvo za korišćenje razvojnog okruženja dato je na sajtu predmeta u sekciji „Projektni zadatak“.

Za prevođenje i povezivanje koristi se prevodilac *gcc* za RISC-V procesor.

Uputstva za izvršno okruženje

Za pokretanje izvršnog fajla koristi se emulator *qemu*. On omogućava emulaciju procesora RISC-V, potrebnih periferija, kao i izvršavanje programa na tom emuliranom procesoru. Komanda `make` prilikom izvršavanja ciljeva `qemu` i `qemu-gdb` pokreće emulator *qemu*, zadaje mu konfiguraciju i izvršni fajl. Emulator *qemu* pokreće izvršni program i povezuje se sa terminalom razvojnog okruženja, nakon čega će program koji se izvršava koristiti taj terminal kao svoj standardni ulaz i izlaz. Nakon završetka programa, emulator *qemu* ostaje uključen i potrebno ga je ugasiti pomoću opcije „Stop“ iz menija razvojnog okruženja.

Emulator *qemu* pruža opciju za udaljeno[12] (engl. *remote*) debagovanje. Komanda `make` pokreće emulator *qemu* za debagovanje prilikom izvršavanja cilja `qemu-gdb`. Emulator *qemu* će osluškivati na nekom portu čiji će broj ispisati na terminalu. Emulator *qemu* neće pokrenuti program sve dok se ne započne debagovanje. Pomoću programa za debagovanje *gdb* moguće je uspostaviti sesiju debagovanja sa emulatorom *qemu*. To se radi dostavljanjem IP adrese i broja porta na kojem emulator *qemu* očekuje poruke (npr. `localhost:26000`). Nakon uspostavljanja sesije program se izvršava u debag režimu. Program *gdb* moguće je pokrenuti iz razvojnog okruženja CLion. Sve mogućnosti okruženja CLion za debagovanje se mogu koristiti (npr. *breakpoint*, *watch*, izvršavanje u režimu instrukcija po instrukciju, itd.). Razvojno okruženje CLion ima mogućnost i direktnog pristupa programu *gdb*, što se može iskoristiti za pristup korisnim informacijama. Npr. program *gdb* ima komandu `info reg` koja ispisuje sadržaj svih programski dostupnih registara procesora. Prekid izvršavanja se postiže pritiskom na opciju „Stop“ iz menija razvojnog okruženja CLion.

[11] Komanda `make` bi trebalo da detektuje koji su se fajlovi promenili od prethodnog prevođenja i da samo njih prevede. Ukoliko se primeti da komanda nije prevela sve izmenjene fajlove, projekat treba očistiti i prevesti ispočetka. Taj problem se najčešće dešava kada su urađene promene samo u zaglavljima.

[12] Udaljeno debagovanje u ovom slučaju omogućava da se program koji se debaguje izvršava u jednom procesu dok se program za debagovanje izvršava u drugom procesu. Ta dva procesa međusobno komuniciraju preko tzv. softverskih priključnica (engl. *socket*).

Smernice za rešavanje zadatka

U ovom poglavlju date su neke smernice za izradu definisanog projekta. Sadržaj ovog poglavlja nije deo obavezujućih zahteva ovog projekta, već ga treba shvatiti samo kao pomoć u implementaciji projekta, kao niz saveta za to kako pristupiti izradi ovog zadatka. Naravno, student je potpuno slobodan da ne postupi po ovim smernicama i napravi neko svoje, drugačije rešenje, uz pretpostavku da za to ima dobre, tehnički ispravne razloge.

Arhitektura i glavne projektne odluke

Zahtevani sistem je daleko od sasvim jednostavnog softverskog sistema[13] i po svojoj složenosti značajno prevazilazi programerske zadatke koje je većina studenata do sada imala prilike da rešava. Zato je pri njegovoj izradi važno poštovati sve principe softverskog inženjerstva.[14] Nikako ne treba žuriti ka kodovanju (engl. *rush to code*). Najpre treba dobro osmisliti i projektovati sistem, u glavi (ili uz pomoć papira ili nekog softverskog alata) osmisliti njegovu arhitekturu i osnovne module, a potom definisati njihove odgovornosti i interfejse. Osim toga, treba dobro rešiti neke ključne probleme i elemente najvažnijih mehanizama (scenarija), najpre u mislima, a ako je potrebno, i uz pomoć brzih i kratkih programskih proba – prototipova koje obavezno treba odbaciti nakon eksperimentisanja. Tek kada se postigne dobar stepen razumevanja svih ovih elemenata, treba krenuti ka programskoj implementaciji i to iterativno, u malim inkrementima, uz temeljno testiranje napravljenih delova pre njihove dalje nadogradnje.

Arhitektura

Pošto su osnovni koncepti, funkcionalnosti i sistemski pozivi, kao i interfejsi zahtevanog sistema već definisani, jedno od početnih pitanja u koncipiranju implementacije svakog operativnog sistema jeste pitanje njegove opšte arhitekture. O ovom pitanju biće više reči u predmetu „Operativni sistemi 2“. U ovom projektu dovoljno je (i to se i predlaže) jezgro realizovati kao monolitan softver, u smislu da se sav kôd jezgra izvršava u istom adresnom prostoru, u privilegovanom režimu rada procesora, dok se međusobni pozivi usluga unutar jezgra realizuju kao najobičniji pozivi potprograma u istom toku kontrole. To, naravno, ne isključuje, već podrazumeva dobru objektnu dekompoziciju na klase i njihove hijerarhije, uz polimorfne pozive operacija tamo gde je to primereno. Moguće su i druge, bitno drugačije arhitekture, ali o njima ovde neće biti reči. Osim samog kôda jezgra, arhitektura samog interfejsa jezgra već je definisana kao slojevita na opisani način, tako da je i to pitanje unapred rešeno.

Osim definisanja arhitekture, potrebno je doneti i neke osnovne projektne odluke koje utiču na celokupan dizajn jezgra i koje su opisane u nastavku.

Monoprocesorski ili multiprocesorski sistem

Današnji realni operativni sistemi opšte namene su multiprocesorski, jer su svi današnji računari opšte namene takvi. Naravno, osnovna prednost ovakvih sistema jeste mogućnost paralelne obrade, odnosno izvršavanje uporednih tokova kontrole na različitim

[13] Ovaj sistem je, istina, i daleko jednostavniji od mnogih složenih sistema kakvi su danas vrlo rašireni.

[14] Za detaljnija objašnjenja ovih principa i njihovog sprovođenja pomoću objektne paradigme programiranja čitalac se upućuje na materijale za predmet „Objektno orijentisano programiranje“, http://oop.etf.rs.

procesorima.[15] Međutim, implementacija multiprocesorskog sistema nosi odgovarajuće dodatne izazove u implementaciji:

- potrebno je iskoristiti odgovarajuću procesorsku podršku za identifikaciju procesora na kom se izvršava kôd, kako bi se obezbedio pristup baš onim strukturama podataka koje se odnose samo na tekući procesor (npr. tekući proces, skup spremnih procesa);
- potrebno je obezbediti međusobno isključenje pristupa kritičnim sekcijama kôda jezgra od strane različitih procesora odgovarajućim konstruktima tj. instrukcijama procesora za „zaključavanje vrtenjem“ (engl. *spin lock*);
- kôd jezgra ne može da se izvršava na jednom zajedničkom steku, jer se ne sme dozvoliti da različiti procesori kôd jezgra izvršavaju istovremeno na istom steku; najmanje je potrebno da svaki procesor ima takav svoj stek;
- potrebno je obezbediti odgovarajuće raspoređivanje procesa na više procesora, što se može uraditi na više različitih načina, pa ovo pitanje zahteva donošenje nekoliko drugih projektnih odluka; o ovoj temi govori se na predmetu „Operativni sistemi 2“.

U svakom slučaju, ovaj projekat podrazumeva implementaciju jednoprocesorskog jezgra, pa su ova pitanja irelevantna, a implementacija jezgra značajno jednostavnija.

Preotimanje

Sledeća važna odluka jeste ta da li sistem koji se pravi omogućava preotimanje (engl. *preemption*) za vreme izvršavanja kôda korisničkih procesa (u slučaju ovog projekta, korisničkih niti), odnosno da li podržava i asinhronu promenu konteksta (a ne samo sinhronu). Ova odluka ostavljena je studentu – student se može opredeliti da ne radi ovaj deo projekta (videti poglavlje „Način ocenjivanja“ ovog dokumenta). Sistem koji omogućava preotimanje za vreme izvršavanja korisničkog kôda je svakako napredniji, jer obezbeđuje brži odziv sistema na događaje u svom okruženju, pošto omogućava preotimanje na hardverske prekide i time omogućava da proces koji čeka na signalizirani događaj dobije procesor odmah po obradi prekida, a ne tek kada proces koji koristi procesor sam izazove sinhronu promenu konteksta. Zbog toga su svi današnji operativni sistemi opšte namene takvi. Međutim, implementacija ovakvog sistema je složenija, upravo zbog toga što asinhroni prekidi stižu u nepredvidivim trenucima vremena, što znači da se mogu pojaviti tokom izvršavanja bilo kog dela kôda. Zbog toga je potrebno mnogo više pažnje posvetiti međusobnom isključenju kritičnih sekcija kôda jezgra.

Čak i ako se student opredeli za implementaciju sistema koji omogućava preotimanje tokom izvršavanja kôda korisničkih niti, sledeće važno pitanje jeste to da li se preotimanje procesora, odnosno asinhrona promena konteksta, može dešavati i tokom *izvršavanja kôda jezgra*. Sistemi koji omogućavaju ovakvo preotimanje imaju, logično, još bolji odziv, a takva jezgra nazivaju se *jezgra sa preotimanjem* (engl. *preemptive kernel*). Međutim, implementacija ovakvog jezgra je još složenija. U okviru ovog projekta nije neophodno praviti ovakvo jezgro sa preotimanjem, ali ambiciozniji studenti mogu da se upuste i u ovakvu implementaciju.

Međusobno isključenje

U slučaju da jezgro ne omogućava preotimanje tokom svog izvršavanja, ceo kôd jezgra se može posmatrati kao jedinstvena kritična sekcija tokom čijeg izvršavanja nije moguća promena konteksta. Ovo se može jednostavno obezbediti maskiranjem prekida prilikom ulaska u kôd jezgra na svim mestima (što procesor i sam radi opisanim mehanizmom obrade prekida, izuzetka i sistemskog poziva). Alternativno, prekidi se mogu dozvoliti tokom izvršavanja kôda

[15] Pod „procesorom“ ovde podrazumevamo sve vrste procesorskih jedinica koje mogu izvršavati instrukcije, poput procesora, procesorskih jezgara (engl. *core*), procesorskih niti (engl. *thread*), specijalizovanih koprocesora različite namene i slično.

jezgra, s tim da se u prekidnoj rutini samo evidentira njihovo pojavljivanje, a onda se oni obrade sekvencijalno, u nekom trenutku izvršavanja kôda jezgra, npr. pozivom potprograma koji je za to namenjen na određenom mestu. Kako nema više procesora, *spin lock* nije potreban.

Ukoliko se pak student opredeli za složeniju i napredniju implementaciju jezgra koje omogućava preotimanje tokom izvršavanja svog kôda, mora da posveti pažnju rešavanju sledećih problema i pitanja:

- treba identifikovati deljene strukture podataka kojima se može pristupati iz konteksta različitih niti i operacije koje nad njima rade; svakoj strukturi treba pridružiti odgovarajuću sinhronizacionu primitivu ili promenljivu, a operaciju posmatrati kao kritičnu sekciju;
- međusobno isključenje sasvim bazičnih kritičnih sekcija jezgra, onih koje se koriste za implementaciju promene konteksta i semafora, treba obezbediti maskiranjem prekida (*spin lock* nije potreban, jer je sistem jednoprocesorski);
- u zavisnosti od projektne odluke opisane u narednom odeljku, kako će biti objašnjeno kasnije, međusobno isključenje ostalih kritičnih sekcija može se obezbediti semaforima, ako se one izvršavaju u kontekstu korisničkih niti (iako u privilegovanom režimu rada procesora);
- potrebno je paziti na sprečavanje mrtve blokade (engl. *deadlock*), recimo poštovanjem uvek istog redosleda zaključavanja kritičnih sekcija; o ovom problemu više se govori u predmetu „Operativni sistemi 2“;
- ukoliko postoji ugnežđivanje kritičnih sekcija koje se isključuju maskiranjem prekida, potrebno je paziti da se prekid demaskira tek pri izlasku iz krajnje spoljašnje ugnežđene sekcije; ovo se može rešiti npr. brojanjem nivoa dubine ugnežđivanja.

Stek na kom se izvršava kôd jezgra

Naredna važna odluka koju treba doneti jeste ta na kom steku se izvršava kôd jezgra. U najjednostavnijoj varijanti, kôd jezgra se, sve do trenutka promene konteksta (tj. steka), može izvršavati na steku tekuće niti koja je pozvala sistemski poziv ili tokom čijeg izvršavanja se dogodio prekid. Međutim, zbog potencijalne korupcije sadržaja memorije zbog prekoračenja prostora koji je za taj stek odvojen, uvek je sigurnije preusmeriti izvršavanje kôda jezgra na stek koji se nalazi na nekom drugom mestu i za koji je odvojen dovoljan prostor za najdublje ugnežđivanje poziva potprograma u jezgru.[16] U tom slučaju, moguće su sledeće varijante:

- ceo kôd jezgra izvršava se na jednom jedinom steku, sve do povratka konteksta izvršavanja niti kojoj se vraća kontrola po izlasku iz sistemskog poziva ili obrade prekida; ova varijanta nije moguća ukoliko jezgro omogućava preotimanje tokom izvršavanja svog kôda, dok u slučaju multiprocesorskog sistema, svaki procesor mora imati svoj poseban ovakav stek;
- svakoj korisničkoj niti pridružuje se poseban dodatni stek koji suštinski predstavlja logički nastavak steka korišćenog za izvršavanje korisničkog kôda, samo je alociran na drugom mestu; na ovaj način, svaki deo kôda jezgra izvršava se u kontekstu neke niti, sve do samog trenutka promene konteksta tj. steka; tada se izvršavanje potprograma jezgra može shvatiti kao prosto izvršavanje ugnežđenih potprograma u kontekstu iste kontrole toka – korisničke niti, sve do samog trenutka promene konteksta, kada se izvršavanje prebacuje u kontekst neke druge niti i na drugi stek.

[16] Preciznije, za ugnežđene pozive sa najvećim gabaritom automatskih podataka alociranih na steku za ugnežđene pozive. Po pravilu, jezgro ne koristi rekurzije, jer za tim uglavnom nema potrebe.

Memorijski kontekst

U opštem slučaju multiprocesnog operativnog sistema, jedan važan aspekt jeste memorijski kontekst u kom se izvršava kôd jezgra. U ovom projektu ovo pitanje je pojednostavljeno, jer se sav kôd, i korisničkih niti i jezgra, izvršava u istom, jedinstvenom memorijskom kontekstu, pa o ovome ne treba voditi računa.

Zaključak

Na kraju, zaključimo da osim samog postupka promene konteksta procesora i obrade sistemskih poziva/prekida/izuzetka, kôd jezgra možemo posmatrati kao standardan, objektno orijentisan softver pisan na jeziku C++, u kom objekti klasa sarađuju uobičajenim pozivima (potencijalno polimorfnih) operacija. Naravno, u tom kôdu ne treba koristiti nikakve bibliotečne funkcije, što uključuje i zabranu poziva funkcija za alokaciju memorije, ali i operatora `new` za pravljenje dinamičkih objekata, jer ovaj operator podrazumevano uključuje poziv bibliotečne funkcije za alokaciju memorije. Ukoliko jezgro ne podržava preotimanje tokom svog izvršavanja, nikakva dodatna sinhronizacija nije potrebna u ovom kôdu; u suprotnom, treba obezbediti sinhronizaciju kako je opisano. Obrada sistemskih poziva/prekida/izuzetaka, kao i sam postupak promene konteksta zahtevaju posebnu pažnju.

Ključne apstrakcije

Nakon utvrđivanja glavnih projektnih odluka opisanih u prethodnom odeljku, predlaže se koncipiranje ključnih apstrakcija koje, kao C++ klase, čine implementaciju jezgra. Jedan predlog su sledeće apstrakcije[17]:

- `MemoryAllocator`: *singleton*[18] klasa koja obezbeđuje usluge alokacije i dealokacije memorije kontinualnom alokacijom, kako je opisano u zahtevima.
- `Thread` ili `PCB`: klasa koja apstrahuje nit i čuva njen kontekst, kao i druge potrebne atribute.
- `Scheduler`: *singleton* klasa koja implementira raspoređivač procesa, tj. algoritam raspoređivanja.
- `Semaphore`: klasa koja apstrahuje semafor i operacije nad njim.
- `Console`: *singleton* klasa koja implementira spregu ka konzoli.

Tokom osmišljavanja mehanizama interakcije, odnosno scenarija u implementaciji jezgra, treba definisati interfejs ovih klasa predviđajući u tom interfejsu one operacije koje su potrebne korisnicima tih klasa, odnosno one koje ispunjavaju odgovornosti tih klasa. Ove klase treba potom implementirati postupno, inkrementalno, kako se implementiraju delovi jezgra. Njihova implementacija svakako zahteva donošenje daljih projektnih odluka na nižem nivou detalja.

---

[17] Kako bi se identifikatori koji su defnisani u jezgru razlikovali od onih koji označavaju iste ili slične koncepte i operacije u njegovom API-u, može se koristiti poseban prostor imena na jeziku C++ (engl. *namespace*) ili se identifikatori definisani u jezgru mogu posebno označiti npr. prefiksom `k`. Ova druga tehnika se često koristi u implementaciji stvarnih kernela koji su implementirani na jeziku C (npr. `kmalloc` za funkciju koja alocira memoriju za potrebe jezgra).

[18] Engl. *Singleton* („usamljenik“) je projektni obrazac kojim se obezbeđuje da klasa ima samo jedan objekat, lako dostupan na unapred definisan način. Za objašnjenje ovog obrasca pogledati materijale za predmet „Objektno orijentisano programiranje“ (http://oop.etf.rs) ili npr. sledeći izvor: https://en.wikipedia.org/wiki/Singleton_pattern. Alternativa primeni ovog obrasca je uslužna klasa – klasa koja nije predviđena za instanciranje, već predstavlja samo logičko pakovanje uslužnih operacija i sadrži samo statičke funkcije članice i po potrebi statičke podatke članove.

Jedan aspekt o kom treba voditi računa jeste alokacija objekata klasa koje su namenjene za dinamičko instanciranje (od pomenutih, klase `Thread` i `Semaphore`, kao i druge slične u implementaciji jezgra). Potreba da se ovi objekti dinamički prave i uništavaju u odgovarajućim scenarijima, tipično implementaciji odgovarajućih sistemskih poziva, ne može se jednostavno rešiti pravljenjem dinamičkih objekata ugrađenim operatorom `new` na jeziku C++ jer se ovaj operator oslanja na sistemski poziv `mem_alloc`, kako je ranije rečeno, pa bi u tom slučaju sam kernel pozivao sopstveni sistemski poziv. Umesto toga, za ove potrebe, predlaže se uvođenje odgovarajućih statičkih operacija posmatranih klasa koje su zadužene za dinamičko pravljenje i uništavanje njihovih objekata, ili samo za alokaciju prostora za te objekte preklapanjem operatorskih funkcija `new` i `delete` za ove klase (nakon čega se mogu koristiti operatori `new` i `delete` za pravljenje i uništavanje dinamičih objekata). Kada je u pitanju način alokacije prostora za te objekte, neke moguće opcije su sledeće, a izbor se ostavlja studentu:

- *Statička alokacija*. Unapred, statički alocirati niz pregradaka za smeštanje ovih objekata i voditi evidenciju zauzetih i slobodnih pregradaka pri dinamičkoj alokaciji i dealokaciji objekata. U slučaju da slobodnih nema, operacija treba da vrati grešku (ili podigne izuzetak), što se propagira i na odgovarajući sistemski poziv u kom se sve to dešava. Posledica ovakvog rešenja je uvođenje logičkih ograničenja u kapacitet jezgra (ograničen broj napravljenih niti i semafora).
- *Dinamička alokacija*. Ovo naprednije, fleksibilnije (jer ne uvodi logička ograničenja), ali i složenije rešenje podrazumeva dinamičku alokaciju nizova korišćenjem alokatora memorije jezgra po potrebi, a potom korišćenje tih nizova kao kod statičke alokacije. Potpuno slobodan niz se može i dealocirati.

Opisanu funkcionalnost alokacije pregradaka i niza dobro je izdvojiti u posebnu klasu i to šablonsku, čime se olakšava njena upotreba na različitim mestima, odnosno klasama.

Preotimanje, promena konteksta i raspoređivanje

Okolnost koja olakšava implementaciju preotimanja i sistemskih poziva jeste ta što dati procesor ima jedinstvenu rutinu za obradu sistemskog poziva, spoljašnjeg prekida i izuzetka, onu na koju ukazuje vrednost registra `stvec`, dok sadržaj odgovarajućih registara daje informaciju o uzroku skoka u ovu rutinu, kako je opisano.[19] Zbog toga je potrebno i dovoljno napraviti jednu prekidnu rutinu za obradu svih tih uzroka. Ta prekidna rutina predstavlja jedno jedino mesto prelaska iz kôda korisničkih niti u kôd jezgra i nazad, kao i iz korisničkog režima rada procesora u sistemski režim i nazad.

U slučaju da se jezgro implementira tako da se kôd jezgra izvršava na steku koji pripada pojedinačnoj niti, kao višenitni program, sve do promene konteksta, odnosno prelaska na neki drugi stek, pozivi funkcija koriste stek tekuće niti. Kako prevodilac generiše odgovarajuće instrukcije za čuvanje na steku onih registara koje koristi pozvani potprogram, celo izvršavanje pomenute prekidne rutine i svih ugnežđenih poziva može se posmatrati kao najobičnije ugnežđivanje poziva, pa u prekidnoj rutini nema potrebe čuvati ceo kontekst procesora, osim onih registara koje sama prekidna rutina neposredno menja; te vrednosti se čuvaju na steku, kao i obično. U slučaju da asinhronih promena konteksta nema, ovo je i dovoljno. Međutim, kod ashinronih prekida postoji sledeći potencijalni problem. Prevodilac ili programer na asembleru u korisničkom kôdu može koristiti bilo koji registar programski

[19] Čak i da to nije slučaj, isti efekat bi se mogao relativno jednostavno proizvesti pravljenjem odgovarajućih kratkih prekidnih rutina za različite uzroke, koje samo upisuju informaciju o uzroku na odgovarajuće mesto i potom prelaze na izvršavanje istog zajedničkog koda. Problem bi moglo da predstavlja pitanje mesta u koje treba da se upiše informacija o uzroku i eventualna potreba korišćenja registara za taj upis, o čijem čuvanju onda treba voditi dodatnog računa.

dostupan u korisničkom režimu rada procesora; na primer, može koristiti privremene (`t`) registre posmatranog procesora RISC-V. Kada generiše kôd za obične odnosno sinhrone pozive potprograma, prevodilac to može raditi tako da ti registri na mestu poziva nikada nisu „živi“, kako se kaže u žargonu, odnosno da njihova vrednost ne mora da se čuva jer se više ne koristi posle tačke poziva potprograma. Međutim, kako se asinhroni prekid može dogoditi i u trenutku kada su te vrednosti još uvek „žive“, a mogu se prepisati tokom izvršavanja kôda jezgra, potrebno je i njih sačuvati. Čuvanje vrednosti takvih registara može obezbediti i prevodilac automatskim generisanjem kôda za ulazak i izlazak iz funkcije označene specifikatorom `interrupt` ili sličnim, ukoliko podržava takvo označavanje. Međutim, ovi aspekti su jako zavisni od prevodioca i procesora i sa njima treba biti veoma pažljiv.

Zato ova prekidna rutina u ovom slučaju postojanja asinhronih prekida treba da uradi sledeće:

1. Pređe na odgovarajući sistemski deo steka tekuće niti i sačuva sve ili samo neke registre programski dostupne u korisnikom režimu na ovom steku; u opštijem slučaju, a svakako ako je podržana asinhrona promena konteksta, treba ih sačuvati sve.
2. Obradi uzrok skoka u prekidnu rutinu (sistemski poziv, izuzetak ili prekid). U okviru ove obrade može doći i do promene tekuće niti i promene konteksta, odnosno tekućeg steka.
3. Po potrebi restaurira registre koje je čuvala na ulasku u prekidnu rutinu i vrati se iz nje.

Sama promena konteksta u ovom slučaju obavlja se onda kada to odgovara funkcionalnosti jezgra i treba je lokalizovati u poseban potprogram poput potprograma `yield` čiji su primeri dati na nastavi. Ovaj potprogram treba da sačuva kontekst tekuće niti na mesto predviđeno za to (stek tekuće niti ili PCB), prebaci se na stek odredišne niti i povrati kontekst te niti:

```
void yield (Thread* oldThread, Thread* newThread);
```

U slučaju da se jezgro implementira tako da se kôd jezgra izvršava na jednom, zajedničkom steku, izvršavanje kôda jezgra može se konceptualno posmatrati kao izvršavanje posebnog toka kontrole, tj. posebne, jedne niti jezgra. Zato se prelazak u kôd jezgra iz koda korisničke niti i obratno, koji se dešava u pomenutoj prekidnoj rutini, može posmatrati kao promena konteksta sa korisničke niti na nit kernela kod ulaska u prekidnu rutinu i obratno prilikom povratka iz nje. Ova rutina zato treba da uradi sledeće:

1. Pređe na odgovarajući sistemski stek i sačuva kontekst procesora, tj. sve registre programski dostupne u korisničkom režimu rada procesora u mesto za čuvanje konteksta niti.
2. Obradi uzrok skoka u rutinu (sistemski poziv, izuzetak ili prekid). U okviru ove obrade može doći i do promene tekuće niti (ali ne i promene konteksta tj. steka).
3. Vrati kontekst procesora tekuće niti, vrati korisnički stek i vrati se iz rutine.

I u ovom slučaju se korak 1 konceptualno može razumeti kao prelazak iz konteksta (i steka) korisničke niti u kontekst jedne jedine niti jezgra, dok korak 3 radi obratno. Zbog toga se i ovi koraci mogu implementirati potprogramima sličnim pomenutom potprogramu `yield`, pri čemu nit jezgra izvršava korak 2.[20]

Prema tome, promena konteksta za navedene dve različite varijante implementacije jezgra može se posmatrati na sledeći način:

- U slučaju da se jezgro implementira tako da se kôd jezgra izvršava na steku koji pripada pojedinačnoj niti, kao višenitni program: sav kôd jezgra, sav posao koji treba uraditi u jezgru, izvršava se uvek u kontekstu (tj. na steku) neke niti, a tek u trenutku kada procesor treba oduzeti jednoj i dati drugoj niti, negde u dubini jezgra, npr. u operacijama

[20] Ovakvi tokovi kontrole koji eksplicitno jedan drugom prebacuju kontrolu pozivom poput ovde pomenutog `yield`, nazivaju se *korutine* (engl. *coroutine*).

semafora ili kada se to odluči iz drugog razloga, prelazi se na kontekst druge niti (operacijom `yield`).

- U slučaju da se jezgro implementira tako da se kôd jezgra izvršava na jednom, zajedničkom steku: promena konteksta (steka) se radi na samom ulasku u kôd jezgra (tj. u prekidnu rutinu), kao i na izlasku iz kôda jezgra (tj. iz prekidne rutine), dok se ceo kôd jezgra i sav posao koji ono treba da radi odvija na jednom zajedničkom sistemskom steku.

Naravno, u obe varijante, poslovi jezgra mogu se obavljati i u nitima koje je samo jezgro pokrenulo, ali se onda te niti tretiraju isto kao i sve ostale, osim što se njihov kôd potencijalno izvršava u privilegovanom režimu rada procesora. Ovo se može izvesti tako da se razlikuju dve vrste niti, odnosno da se za svaku nit može odrediti da li njeno telo treba izvršavati u korisničkom ili sistemskom režimu rada procesora: korisničke niti su uvek ove prve, dok interne niti jezgra mogu biti ove druge. Pri restauraciji konteksta onda treba postavi bit `SPP` procesora na odgovarajuću vrednost (podsetnik: ovaj bit definiše režim u koji se prelazi po povratku iz prekidne rutine).

Sadržaj koraka 2 u oba slučaja najbolje je izdvojiti u poseban potprogram koji se piše na jeziku C/C++. U okviru ovog potprograma treba izvršiti razgranati skok na potprogram za obradu odgovarajućeg uzroka, uključujući i razgranati skok na obradu odgovarajućeg sistemskog poziva. Za ovaj razgranati skok, osim `switch` konstrukta, može se koristiti i skok dinamičkim vezivanjem preko niza pokazivača na funkcije čiji elementi odgovaraju odgovarajućim kôdovima sistemskih poziva. Olakšavajuću okolnost u implementaciji ovih poziva stvara činjenica da se na ovoj platformi parametri poziva funkcija prenose kroz registre, a parametri sistemskih poziva na nivou ABI-a su već u registrima, pa ovo pojednostavljuje poziv C funkcije iz prekidne rutine.

Čuvanje i restauracija konteksta (implementacija pomenutog potprograma `yield` u prvom, odnosno koraka 1 i 3 u drugom slučaju) mogu se pisati na asembleru datog procesora. Jedna projektna odluka koju ovom prilikom treba doneti jeste to gde se čuva kontekst procesora. Ukoliko se kôd jezgra izvršava na jednom zajedničkom steku (drugi pomenuti slučaj), kontekst procesora se mora čuvati u strukturi koja predstavlja PCB. U suprotnom, to se može raditi u strukturi PCB ili na steku niti.

Sledeća projektna odluka jeste to kako predstavljati stanja niti i skup spremnih niti. Jedna moguća varijanta jeste organizovanjem ulančane liste (ili neke druge dinamičke strukture koja bolje odgovara odabranom algoritmu raspoređivanja) u kojoj su spremne niti, koja može biti dodeljena u odgovornost klasi `Scheduler`, kao što je urađeno na predavanjima. Pritom treba dobro razmisliti o načinu implementacije ulančane liste. Dinamička alokacija struktura, npr. korišćenjem alokatora unutar jezgra, u kojoj su samo pokazivači za ulančavanje i pokazivač na nit, podrazumeva dodatne režijske troškove pri svakom ubacivanju i izbacivanju elementa liste, kako u pogledu vremena koje je potrebno za dinamičku alokaciju i dealokaciju tih struktura, tako i u pogledu zauzeća memorije, zbog čuvanja dodatnih informacija oko tih struktura koje su potrebne alokatoru. Osim toga, to može da uzrokuje veliku fragmentaciju memorije koja se koristi za pojedinačnu alokaciju ovako malih struktura (samo nekoliko pokazivača) u zajedničkom memorijskom prostoru u kom se alociraju i druge strukture (engl. *heap*). Efikasnije rešenje je zato pokazivače za ulančavanje organizovati unutar same strukture koja predstavlja nit, ali tada treba dobro paziti da se isti pokazivači ne koriste istovremeno za ulančavanje u različite strukture u kojima isti objekat može biti istovremeno uključen. Osim toga, treba paziti i da ova implementacija ne naruši enkapsulaciju.[21]

[21] O ovakvoj implementaciji na jezicima C i C++ detaljna diskusija data je u materijalima za predmet „Objektno orijentisano programiranje“ (http://oop.etf.rs).

Konačno, potrebno je implementirati neki algoritam raspoređivanja unutar klase `Scheduler`. Izbor algoritma raspoređivanja se u potpunosti ostavlja studentu. Dovoljno je implementirati najjednostavniji, FIFO (FCFS) algoritam, ali ambiciozniji studenti mogu se opredeliti i za implementaciju nekog od naprednijih algoritama raspoređivanja.[22]

Treba obratiti pažnju i na to kako rešiti situaciju u kojoj u skupu spremnih nema korisničkih niti. Najjednostavnija rešenja se oslanjaju na to da samo jezgro uvek ima spremne niti; ako to nisu niti koje rade neke korisne režijske poslove, poput „čišćenja đubreta“ (engl. *garbage collection*), što je izraz koji se odnosi na odloženu dealokaciju prostora koji se više ne koristi, kompakcije prostora ili nečeg drugog, može se prosto obezbediti jedna „besposlena“ (engl. *idle*) nit koja ne radi ništa (vrti se u praznoj petlji) i koja procesor dobija samo ako nema drugih spremnih niti.

Implementacija nekih zahtevanih funkcionalnosti

U ovom odeljku će ukratko biti prokomentarisani samo neki aspekti implementacije zahtevanih funkcionalnosti jezgra, uključujući i sistemske pozive.

Alokaciju i dealokaciju prostora (klasa `MemoryAllocator`) treba implementirati nekim algoritmom kontinualne alokacije (*first fit* ili *best fit*), čiji se izbor ostavlja studentu. O ovoj alokaciji dovoljno je rečeno na nastavi, a postoje i rešeni zadaci koji pokazuju elemente njene implementacije.

Prilikom kreiranja niti, osim kreiranja objekta koji je predstavlja u jezgru, o čemu je već bilo reči, najveći izazov jeste formiranje početnog konteksta niti. Ovo se može uraditi tako što se poziv tela korisničke niti (funkcija na koju ukazuje odgovarajući argument sistemskog poziva) „umota“ u jednu u jezgru definisanu funkciju „omotač“ (engl. *wrapper*), a početni kontekst niti uvek postavi tako da počne izvršavanje od ove funkcije-omotača[23]. Na ovom mestu treba posebno paziti na to da se taj kontekst postavi baš onako kako je potrebno da se prilikom restauracije tog konteksta na opisani način izvršavanje prebaci na telo ove funkcije. Takođe treba rešiti i to kako će ova funkcija-omotač dohvatiti pokazivač na funkciju korisničke niti; ovde postoji nekoliko jednostavnih rešenja koja se ostavljaju studentu na izbor. Iz ove funkcije-omotača izvršavanje ne bi smelo nikada da se vraća u pozivaoca (jer ga ona i nema), pa ona, nakon poziva funkcije koja predstavlja telo korisničke niti, treba da ima u sebi sistemski poziv za gašenje niti koji se izvršava ukoliko se nit pre toga već nije ugasila ovakvim pozivom iz kôda korisničke niti. Treba primetiti to da se telo ove funkcija-omotača, kao i u nju ugnežđenog poziva tela korisničke niti, izvršava u korisničkom režimu rada procesora.

Implementacija sistemskih poziva `thread_exit` i `thread_dispatch` ne nosi nikakve posebne teškoće, nakon što je sve do sada rečeno uzeto u obzir i valjano rešeno. Slično važi i za semafore i implementaciju operacija sa njima, o čemu je već dovoljno rečeno na nastavi.

Posebnu pažnju zaslužuje implementacija sistemskog poziva `time_sleep`, odnosno funkcionalnosti uspavljivanja niti na zadato vreme i njeno buđenje. Jedna moguća jednostavna implementacija je sledeća. Uspavane niti mogu se organizovati u ulančanu listu, uređenu po vremenu buđenja[24]. Za svaki element liste (uspavanu nit) može se čuvati samo relativno vreme (ovde izraženo u celom broju perioda tajmera) buđenja u odnosu na prethodni element u listi; ova vrednost može biti i 0, ukoliko je data nit zakazala buđenje u istom trenutku kao i prethodna u listi. Za prvi element u listi ova vrednost izražava vremenski interval do trenutka buđenja u odnosu na sadašnji trenutak. Ovakva struktura zahteva samo malo složeniju operaciju umetanja

---

[22] O algoritmima raspoređivanja procesa detaljno se govori u predmetu „Operativni sistemi 2“.

[23] Jednostavnije je da funkcija-omotač nije nestatička funkcija članica klase, jer bi u tom slučaju njoj morao da se postavi kao skriveni argument i pokazivač `this`, pod uslovom da joj je njegova vrednost uopšte potrebna.

[24] Uzeti u obzir ranije dat komentar o implementaciji dinamički ulančanih lista.

u listu, ali je zato njeno ažuriranje na svaku periodu tajmera jednostavno: dekrementira se pomenuta vrednost samo prvog elementa u listi, što je krajnje efikasno, i ukoliko je ta vrednost tada došla do nule, u red spremnih se vraćaju sve niti sa početka liste koje imaju pomenutu vrednost jednaku nuli.

Ulaz/izlaz

Kako je opisano, prenos podataka iz kontrolera konzole i ka njemu zahteva prozivanje (ispitivanje bita spremnosti, engl. *polling*), s tim da pojava novog znaka pristiglog sa tastature ili spremnost kontrolera da prihvati novi znak za slanje na ekran izaziva prekid. Naravno, prenos znaka na konzolu ili čekanje znaka sa konzole nije dobro raditi uposlenim čekanjem u kontekstu obrade sistemskog poziva, jer bi to moglo neodređeno dugo da zadrži procesor u sistemskom pozivu[25]. Zato je potrebno razdvojiti sistemski poziv od prenosa podataka sa kontrolera konzole i na njega. To se, naravno, radi baferisanjem, pa je potrebno uvesti (logički) neograničeni ili ograničeni bafer ne samo za prihvat znakova sa tastature, već i za prenos znakova na ekran[26].

To znači sledeće:

- U obradi sistemskog poziva `putc` znak treba smestiti u izlazni bafer (korisnička nit je proizvođač). Ukoliko je bafer pun, pozivajuću nit treba blokirati ili joj odmah treba vratiti grešku. Posebna nit jezgra je potrošač: ona uzima znak po znak iz izlaznog bafera i prenosi ga kontroleru konzole, uz prozivanje; ukoliko je bafer prazan, ovu nit treba blokirati sve dok se u baferu ne pojavi znak.
- U obradi prekida zbog znaka pristiglog sa tastature ulazne znakove, koje treba čitati sa kontrolera konzole sve dok ih ima (prozivanjem), treba smeštati u ulazni bafer (prekidna rutina je proizvođač). Da prekidna rutina ne bi trajala predugo ukoliko znakova ima previše, može se i ograničiti broj učitanih znakova. Ukoliko je bafer pun, pritisnuti znakovi se mogu prosto odbaciti ili se njihovo učitavanje može ostaviti za kasnije (proizvođač, čija je kontrola toka ovde izvorno u hardveru, ne može se prosto „blokirati“). U obradi sistemskog poziva `getc` znak treba uzeti iz ulaznog bafera (korisnička nit je potrošač). Ukoliko je bafer prazan, pozivajuću nit treba blokirati.

Tehnika prozivanja je detaljno obrađena na nastavi. Interne niti jezgra treba pokrenuti prilikom inicijalizacije jezgra. Bitno je obezbediti da se njihovo telo, za razliku od tela korisničkih niti, izvršava u sistemskom režimu rada procesora, kako bi one mogle da pristupe registrima hardvera, na način objašnjen ranije.

Posebnu pažnju zahteva implementacija sinhronizacije i blokiranja ovih internih niti jezgra i korisničke niti. U slučaju da se jezgro implementira tako da se kôd jezgra izvršava na steku koji pripada pojedinačnoj niti, kao višenitni program, za ovu sinhronizaciju se mogu koristiti semafori jezgra, kao što je to pokazano na nastavi.

Ukoliko se pak jezgro implementira tako da se kôd jezgra izvršava na jednom zajedničkom sistemskom steku, situacija je nešto drugačija. Naime, zahtev koji je korisnička nit postavila nekim sistemskim pozivom, pri čemu taj zahtev sadrži neke zadate informacije, npr. `putc`, ili se mora prihvatiti i upisati kao zahtev u neki bafer, listu, red ili slično, ili se mora odbiti vraćanjem greške; ne može se prosto pokušati smeštanje i suspendovati pozivajuća nit ako to smeštanje nije moguće (npr. jer je bafer pun), jer je kontekst pozivajuće niti već napušten i ta nit će odmah nastaviti izvršavanje povratkom iz sistemskog poziva čim taj kontekst bude

[25] Podrazumeva se ovde da su prekidi maskirani za sve vreme trajanja obrade sistemskog poziva. Ukoliko to nije slučaj, i ukoliko se jezgro implementira kao višenitno, onda navedeno ne predstavlja problem, ali problem jeste uporedni pristup više niti samom kontroleru konzole.

[26] Imati na umu da je prenos znakova ka konzoli zapravo prenos serijskom vezom, a slanje znaka zahteva određeno vreme za prenos, pa upis narednog znaka u registar podataka kontrolera konzole ne mora biti moguć jer je hardverski bafer kontrolera možda pun, pošto prethodno upisani znakovi još nisu poslati.

povraćen. Drugim rečima, podatak koji pozivajuća nit prenosi kao argument sistemskog poziva ne može se čuvati na steku dok je nit suspendovana, da bi se onda nastavilo izvršavanje kôda jezgra koji će taj podatak preuzeti i smestiti dalje gde je potrebno kad to bude moguće.

Dakle, u ovom slučaju, treba jednostavno ispitati da li u baferu ima mesta i ako ima, smestiti znak bez ikakve posebne sinhronizacije[27], a ukoliko nema, vratiti grešku.

Druga opcija je sasvim drugačija: uslovna sinhronizacija za upis u ograničeni bafer može se obezbediti korišćenjem semafora, ali unutar tela funkcije `putc` C API-a, dok sistemski poziv `putc` na nivou ABI-a može samo da upiše u bafer, bez sinhronizacije. Međusobno isključenje obezbeđuje sama implementacija ovog sistemskog poziva, tako da o tome ne treba brinuti unutar C funkcije. Ovo je jednostavnije, ali manje robusno rešenje, jer ne može da spreči da kôd korisničke niti pozove direktno ABI sistemski poziv, bez potrebne uslovne sinhronizacije. Zato takvo rešenje ne bi bilo valjano u realnom operativnom sistemu.

Asinhroni prekid i tajmer

Ukoliko je sve urađeno valjano prema datim uputstvima, jezgro će podržati asinhronu promenu konteksta na spoljašnje prekide koji u ovom slučaju dolaze iz dva izvora: konzole i tajmera.

Na prekid od tajmera treba principijelno uraditi sledeće dve stvari:

- Ažurirati preostalo vreme izvršavanja tekućoj niti umanjenjem za jednu periodu tajmera i, ukoliko je dodeljeno vreme isteklo, uraditi promenu tekuće niti i konteksta. Naravno, svaki put kada se neka nit bira da bude tekuća, treba joj dodeliti vremenski odsečak (kvantum) za izvršavanje. Vrednost preostalog vremena najlakše je čuvati u jednoj statičkoj promenljivoj (jer se odnosi samo na jednu, tekuću nit) koja je propisno enkapsulirana iza odgovarajućeg interfejsa. Odgovornost za ovo treba pažljivo dodeliti odgovarajućoj klasi.
- Ažurirati evidenciju uspavanih niti kako je ranije opisano i po potrebi „probuditi“ odgovarajuće niti.

Implementacija interfejsnih slojeva

Implementacija prekidne rutine prema ranije datim uputstvima predstavlja implementaciju ABI sloja interfejsa jezgra.

Implementacija C API sloja interfejsa je potom jednostavna. Pošto se argumenti funkcija, kao i sistemskih poziva prenose u registrima procesora, implementacija odgovarajuće C funkcije API-a je jednostavna: ona samo treba da pripremi svoje parametre u odgovarajuće argumente i izvrši instrukciju sistemskog poziva. Pošto je ova operacija identična i zajednička za sve pozive, dobro je izdvojiti je u jednu funkciju opšteg oblika koja prima potpuno opšte argumente i izvršava instrukciju `ecall`. Onda se funkcije C API-a svode samo na poziv ove opšte funkcije sa argumentima složenim i konvertovanim na odgovarajući način. Mali i jednostavan izuzetak su pomenute C API funkcije koje treba samo malo da prerade argumente sistemskog poziva ili imaju dva ili više sistemskih poziva u sebi.

Ni implementacija C++ API-a nije mnogo složenija. Ovaj API samo adaptira neobjektni, C API u objektni, C++ API. Primeri ovakve adaptacije prikazani su na nastavi, a svode se na to da se apstrakcije jezgra predstavljaju klasama čiji objekti u sebi nose (kao atribute) identifikatore (ručke) odgovarajućih objekata napravljenih u jezgru, a svoje funkcije

[27] Pretpostavlja se sve vreme da je međusobno isključenje obezbeđeno na nivou celog kôda jezgra, zaključavanjem, odnosno maskiranjem prekida odmah na ulasku u prekidnu rutinu.

članice preusmeravaju na pozive funkcija C API-a, uz prosleđivanje tog identifikatora kao odgovarajućeg parametra. Slično važi i za konstruktore i destruktore.

Predlog redosleda izrade

Ovde je dat predlog jednog razumnog, ne obavezno najboljeg i nikako jedinog redosleda razvoja rešenja. Kao što je rečeno u prvom odeljku ovog poglavlja, pre implementacije treba doneti one najvažnije projektne odluke koje imaju uticaja na ceo sistem, dobro osmisliti i isprojektovati ključne delove sistema, definisati ključne apstrakcije (klase), dodeliti im odgovornosti i osmisliti scenarije interakcija koje ispunjavaju predviđene funkcionalnosti. Nakon toga se može pristupiti implementaciji jednog po jednog dela, odnosno jedne po jedne funkcionalnosti, iterativno i inkrementalno[28], uz detaljno testiranje, na primer sledećim redom:

1. Alokator memorije (klasa `MemoryAllocator`).
2. Prekidna rutina: samo prenos argumenata i prelazak u odgovarajući režim rada procesora, bez promene konteksta, bez ulaska dalje u jezgro, samo kao potprogram koji se iz korisničkog programa (bez niti) poziva kao „običan“ potprogram, samo preko instrukcije `ecall`, a ne radi više ništa već se odmah vraća nazad.
3. Razgranati skok na implementaciju pojedinačnog sistemskog poziva u prekidnoj rutini. C funkcija koja implementira zajednički deo sistemskih poziva. Sistemski pozivi `mem_alloc` i `mem_free`: poziv iz prekidne rutine, ABI i C API. Testiranje iz „običnog“ C programa (bez niti) pozivom C API funkcije kao običnog potprograma.
4. Kostur jezgra: kostur klase `Thread` i klasa `Scheduler` u potpunosti.
5. Sistemski stek (zajednički ili pojedinačan za nit). Kompletiranje prekidne rutine čuvanjem i restauracijom registara. Promena konteksta: u operaciji `yield` ili u prekidnoj rutini.[29]
6. Formiranje početnog konteksta niti. Sistemski pozivi `dispatch` i `thread_create` (ABI i C API). Testiranje pomoću niti koje se nikada ne završavaju (program gasiti nasilno iz operativnog sistema domaćina).
7. Gašenje niti. Sistemski poziv `thread_exit` (C API i ABI).
8. Semafori: klasa `Semaphore` u implementaciji jezgra u potpunosti, svi sistemski pozivi za operacije sa semaforima (ABI i C API).
9. Asinhroni prekid od tajmera, raspodela vremena (promena konteksta na istek vremenskog odsečka, engl. *time sharing*).
10. Uspavljivanje i buđenje niti, sistemski poziv `time_sleep` (C API i ABI).
11. Konzola, izlazni smer (slanje podataka na ekran): bafer, interna nit jezgra, sistemski poziv `putc` (C API i ABI).
12. Konzola, ulazni smer (prijem podataka sa tastature): bafer, prekidna rutina, sistemski poziv `getc` (C API i ABI).
13. C++ API u celini.

---

[28] Izraz „iterativno i inkrementalno“ označava proces razvoja softvera u kome se softver razvija u iteracijama, pri čemu se u svakoj iteraciji prolazi kroz faze detaljnog projektovanja, implementacije i testiranja određenog manjeg podskupa funkcionalnosti, a u svakoj narednoj iteraciji sistem nadograđuje novim funkcionalnostima istim takvim celokupnim postupkom. Jedan od ciljeva ovakvog postupka jeste i taj da se što pre dođe do minimalnog, ali sasvim funkcionalnog i izvršivog kostura i jezgra projektovanog sistema koje se može testirati i koje služi kao radna podloga za dalji inkrementalni razvoj. Na taj način se smanjuju rizici daljeg razvoja, ali i postižu drugi pozitivni efekti, između ostalih i psihološki (podiže se samopouzdanje i vera u uspeh poduhvata).

[29] Rezultat ovog koraka nije sasvim funkcionalna verzija, pa se može testirati samo parcijalno, ne i u potpunosti dok se ne završi i naredni korak.

Način ocenjivanja

Način predaje projekta

Projekat se predaje isključivo kao jedna zip arhiva. Sadržaj arhive treba podeliti u dva direktorijuma: *src* i *inc*. U prvi direktorijum (*src*) treba smestiti sve *.cpp* i `.S` fajlove, a u drugi (*inc*) sve *.h/.hpp* fajlove koji su rezultat izrade projekta. Opisani sadržaj ujedno treba da bude i jedini sadržaj arhive (arhiva ne sme sadržati ni izvršive fajlove, ni biblioteke, ni *.cpp* i *.h/.hpp* fajlove koji predstavljaju bilo kakve testove, niti bilo šta što ovde nije navedeno). Predaju se isključivo fajlovi sa izvornim kodom; nije dozvoljeno predavati kompletne *git* repozitorijume (iako se *git* može koristiti u razvoju) ili bilo kakve druge fajlove koji nisu tekstualni fajlovi sa izvornim kodom rešenja (C, C++, asembler).

Projekat je moguće predati (engl. *upload*) više puta, ali do trenutka koji će preko imejl liste biti objavljen za svaki ispitni rok i koji će uvek biti prvi radni dan pre ispita. Na serveru uvek ostaje samo poslednja predata verzija i ona će se koristiti na odbrani.

Za izlazak na ispit neophodno je predati projekat (prijava ispita i položeni kolokvijumi su takođe preduslovi za izlazak na ispit). Nakon isteka roka predati projektni zadaci se brišu, pa je u slučaju ponovnog izlaska na ispit potrebno ponovo postaviti ažurnu verziju projektnog zadataka.

Sajt za predaju projekta je:
http://rti.etf.bg.ac.rs/domaci/index.php?servis=os1_projekat
Projekat predat posle zadatog roka se ne uzima u razmatranje. Nepoštovanje ostalih pravila za predaju projekta povlači negativne poene.

Način ocenjivanja projekta

Projekat se može uraditi u celini ili delimično, podeljen na sledeće delove – zadatke, koji se boduju najviše sa datim brojem poena:

Broj	Naziv	Sadržaj zadatka	Broj poena
1	Alokacija memorije	Sistemski pozivi `mem_alloc` i `mem_free`.	5
2	Niti	Niti i sistemski pozivi `thread_create`, `thread_exit` i `thread_dispatch`. Samo sinhrona promena konteksta prilikom bilo kog sistemskog poziva.	10
3	Semafori	Semafori i sistemski pozivi `sem_open`, `sem_close`, `sem_wait` i `sem_signal`.	5
4	Asinhrona promena konteksta	Deljenje vremena i asinhrona promena konteksta na prekid od tajmera i tastature. Sistemski pozivi `thread_sleap`, `getc` i `putc`, kao i klasa `PeriodicThread`.	10
5	Bonus	Projekat odbranjen u predroku	10

Svaki zadatak obuhvata implementaciju sva tri sloja interfejsa jezgra za sistemske pozive obuhvaćene tim zadatkom.

Ukoliko student ne želi da radi zadatak 1 (alokacija memorije), potrebno je da preuzme gotov modul koji sadrži urađen alokator tako što će u svoj projekat uvezati i biblioteku `mem.lib`. Ova biblioteka sadrži C/C++ implementaciju koja se može koristiti unutar jezgra, ali ne

implementira nijedan sloj interfejsa jezgra; taj deo ostaje studentu da uradi. Ova implementacija ne poseduje bilo kakvu sinhronizaciju u sebi i nije *thread-safe.*

Ukoliko student ne želi da radi zadatak 4 (asinhrona promena konteksta), na raspolaganju je postavljena gotova prekidna rutina koja obrađuje prekide od tajmera i konzole. Potrebno je tada da student preuzme gotov modul koji sadrži implementirane funkcije `getc` i `putc` tako što će u svoj projekat uvezati i biblioteku `console.lib`. Ova biblioteka sadrži C/C++ implementaciju koja se može koristiti unutar jezgra, ali ne implementira nijedan sloj interfejsa jezgra; taj deo ostaje studentu da uradi. Ova implementacija ne poseduje bilo kakvu sinhronizaciju u sebi i nije *thread-safe.* Takođe, ove funkcije u toku svog izvršavanja mogu dozvoliti prekide. Da bi date gotove funkcije `putc` i `getc` radile, potrebno je u prekidnoj rutini pozvati funckiju koja obrađuje prekide od konzole pod nazivom `console_handler`.

Zadaci broj 3, 4 i 5 prirodno zahtevaju urađen zadatak broj 2.

Za uspešnu odbranu projekta potrebno je ispuniti sledeće uslove:

- uraditi i proći javne testove za delove projekta koji ukupno nose najmanje 20 poena;
- na odbrani projekta (tajni testovi, modifikacije, usmeni odgovori i obrazloženja) dobiti najmanje 15 poena ukupno za sve urađene delove (urađeni delovi ne moraju dobiti maksimalan navedeni broj poena).

Način provere projekta

Ispravnost i kvalitet urađenog projekta biće proveravani sprovođenjem javnih i tajnih testova, kao i opcionih testova performansi (samo za bonus poene).

Javni testovi biće dostupni studentima unapred, tokom izrade projekta u otvorenom obliku (otvorenog kôda).

Tajni testovi neće biti dostupni studentima tokom izrade projekta, već će biti sprovedeni u zatvorenom obliku prilikom testiranja projekta. Ovi testovi proveravaju sve aspekte navedenih zahteva, i to:

- propisnu alokaciju i dealokaciju memorije;
- propisno pokretanje i gašenje niti;
- ispravno konkurentno izvršavanje niti u slučaju sinhrone promene konteksta, sinhronizacione primitive;
- ispravno konkurentno izvršavanje sa asinhronom promenom konteksta (deljenje vremena i prekidi).

Testovi performansi su tajni testovi koji mogu da ispituju sledeće parametre sistema:

- režijsko vreme promene konteksta;
- režijsko vreme operacija sa semaforom;
- vreme odziva na prekid;
- broj konkurentnih niti koje se mogu pokrenuti i izvršavati dok sistem ne uđe u preopterećenje;
- broj niti koje se mogu uspešno raspoređivati sa sve manjim vremenskim intervalom;
- zauzeće memorije za strukture koje realizuju niti, semafore i događaje;
- druge vremenske i prostorne parametre i ograničenja sistema.

Nezadovoljenje bilo kog javnog testa povlači odbijanje čitavog zadatka ili celog projekta sa brojem bodova 0. Nezadovoljenje nekog tajnog testa ili testa performansi nosi odgovarajući negativan broj poena za zadatak koji se testira.

Osim testova koji će biti sprovedeni bez uvida u sam izvorni kôd projekta, od studenta se može zahtevati da na licu mesta, tokom odbrane projekta, u zadatom vremenu samostalno uradi jednu ili više manjih modifikacija svog projekta prema zahtevima definisanim na odbrani.

Ove modifikacije se takođe mogu proveravati javnim i tajnim testovima. Neurađena modifikacija povlači negativne poene ili odbijanje celog projekta ili nekog zadatka.

Na odbrani projekta će biti pregledan izvorni kôd projekta i ocenjivan njegov kvalitet: stil pisanja programa i njegova urednost i stilska ujednačenost, poštovanje principa proceduralnog i objektnog programiranja, dobra arhitektura, raspodela odgovornosti, dekompozicija, jasni i precizni interfejsi i enkapsulacija, jasni i kratki potprogrami i slično. Uočeni nedostaci povlače negativne poene ili potpuno odbijanje zadatka ili celog projekta sa brojem poena 0.

Konačno, na odbrani projekta ispitivač može proveravati samostalnost u izradi projekta i poznavanje detalja projekta i gradiva predmeta na kom se projekat zasniva postavljanjem usmenih pitanja. Neznanje, nesigurnost ili nepreciznost u odgovorima može nositi negativne poene ili odbijanje celog projekta.

Zapisnik revizija

Ovaj zapisnik sadrži spisak izmena i dopuna ovog dokumenta po verzijama.