Elektrotehnički fakultet u Beogradu
Katedra za računarsku tehniku i informatiku

Predmet: Operativni sistemi 1 (13E112OS1, 13S112OS1)
Nastavnik: prof. dr Dragan Milićev
Školska godina: 2018/2019. (Zadatak važi počev od junskog roka 2019.)

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!

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, realizovaće se jedan podsistem ovog jezgra – podsistem za upravljanje nitima. Ovaj podsistem treba da obezbedi koncept niti (engl. *thread*), kao i usluge kreiranja i pokretanja niti, zatim koncept semafora i događaja vezanog za prekid, i podršku deljenju vremena (engl. *time sharing*).

Jezgro treba da bude realizovano tako da korisnički program (aplikacija) i samo jezgro dele isti adresni prostor, odnosno da predstavljaju jedinstven program. Konkurentni procesi kreirani unutar aplikacije biće zapravo samo "laki" procesi, tj. niti (engl. *thread*). Aplikacija i jezgro treba da se prevode i povezuju u jedan isti program, tj. da se pišu kao "jedinstven" izvorni kod.

U cilju demonstracije i lakšeg razvoja, testiranja i eksperimentisanja, jezgro i aplikacija treba da se izvršavaju na PC računaru, pod operativnim sistemom Windows (bilo kojim 32-bitnim), kao konzolna aplikacija.

Opšti zahtevi

Konstrukcija jezgra i odnos sa sistemom-domaćinom

Jezgro i korisnička aplikacija treba da se posmatraju kao jedinstven program (.exe) dobijen prevođenjem i povezivanjem koda na izvornom programskom jeziku u kome su realizovani. Oni treba da se pokreću unutar operativnog sistema Windows na PC računaru kao jedinstvena konzolna aplikacija (.exe), odnosno proces. Operativni sistem Windows biće tako nazivan *sistemom-domaćinom* (engl. *host system*).

Jezgro i korisnička aplikacija moraju da se regularno završe kao konzolna aplikacija, 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.

U realizaciji jezgra nije dozvoljeno koristiti usluge operativnog sistema-domaćina 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".

U realizaciji jezgra dozvoljeno je koristiti samo standardne (statički povezivane) C/C++ biblioteke koje su potpuno prenosive, tj. nezavisne od platforme (hardvera i operativnog sistema). Nije dozvoljeno koristiti bilo kakve specifične biblioteke sistema-domaćina ili biblioteke sa dinamičkim povezivanjem (DLL). Jezgro i korisnička aplikacija kao jedinstven program treba da budu regularno izvršivi na praktično svakom PC računaru; bilo kakva zavisnost od konkretnog hardvera ili konfiguracije računara nije ispravna. Taj program treba da bude potpuno samostalan i nezavisan od bilo koje druge softverske komponente. On treba da se izvršava jednostavno pokretanjem .exe fajla sa bilo kog medijuma.

Odnos jezgra i korisničke aplikacije

Jezgro treba realizovati na jeziku C++, uz opciono korišćenje asemblera za PC procesore. Korisnička aplikacija sadržaće test primere i biće obezbeđena kao skup izvornih fajlova koje treba prevesti i povezati sa prevedenim kodom jezgra i datom bibliotekom `applicat.lib` u jedinstven konzolni program (.exe). Data biblioteka sadržaće module koji se obezbeđuju kao moduli koji pristupaju (zamišljenom, virtuelnom) hardveru, u zavisnosti od zadatka, odnosno moduli od kojih će jezgro zavisiti (engl. *stub*), a čije su funkcionalnosti i interfejsi opisani u okviru samog zadatka.

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

```
int userMain (int argc, char* argv[]);
```

Argumenti i povratna vrednost ove funkcije imaju istu ulogu kao i argumenti funkcije `main()`, koje i treba proslediti ovoj funkciji. Drugim rečima, argumente programa iz komandne linije sistema-domaćina treba proslediti funkciji `userMain()` i rezultat funkcije `userMain()` treba vratiti sistemu-domaćinu kao rezultat funkcije `main()`.

Funkcija `main()` može da ostane u nadležnosti samog jezgra, pa realizacija jezgra može da pod svojom kontrolom ima radnje koje će se izvršiti pri pokretanju programa od strane sistema-domaćina, a zatim može, na primer, da pokrene nit nad funkcijom `userMain()`.

Zadatak 1: Podsistem za upravljanje nitima

Uvod

Cilj ovog zadatka jeste realizacija podsistema jezgra za upravljanje nitima (engl. *thread*). Zahtevi u okviru ovog zadatka jesu realizacija:

- koncepta niti (engl. *thread*), uz operacije kreiranja i pokretanja niti;
- promene konteksta (engl. *context switch*) i preuzimanja (engl. *preemption*) i to na sledeće načine:
  - eksplicitnim zahtevom same niti (sinhrono, eksplicitno preuzimanje);
  - zbog pojave prekida (asinhrono, implicitno preuzimanje);
  - zbog isteka dodeljenog vremena (asinhrono, implicitno preuzimanje), kao podrška deljenju vremena (engl. *time sharing*);
- koncepta standardnog semafora;
- koncepta događaja (zapravo binarnog semafora) na kome samo jedna nit može da čeka blokirana, a koji se signalizira zadatim prekidom.

Koncept niti

Opis zadatka

Potrebno je obezbediti koncept niti. Sve niti pokrenute unutar aplikacije dele isti adresni prostor (statičke i dinamičke podatke), ali poseduju sopstvene kontrolne stekove na kojima se čuva trag ugnježdenih poziva funkcija, kao i automatski podaci tih funkcija.

Nit se kreira nad određenom funkcijom i kada se pokrene izvršavanje niti, novi tok kontrole počinje od poziva te funkcije. Dalji tok niti zavisi od sadržaja te funkcije.
Potrebno je obezbediti sledeće operacije nad nitima:

- kreiranje niti: ova operacija samo kreira novu nit u jezgru nad određenom funkcijom, ali ne pokreće njeno izvršavanje; u slučaju da je prosleđena vrednost parametra `timeSlice=0`, kreirana niti ima neograničeni vremenski interval izvršavanja.
- pokretanje niti: ova operacija pokreće novi tok izvršavanja nad funkcijom nad kojom je kreirana nit;
- eksplicitno preuzimanje (engl. *dispatch*);
- Identifikacioni broj: sistem treba da obezbedi da svaka nit prilikom kreiranja dobije jedinstveni identifikacioni broj (ID). Sistem takodje treba da obezbedi dohvatanje niti samo na osnovu identifikacionog broja, kao i dohvatanje identifikacionog broja (ID) tekuće niti (*running*) i proizvoljne niti (posedovanjem pokazivača na objekat koji predstavlja tu nit).

Rešenje treba da obezbedi mogućnost:

- kreiranja neograničenog broja korisničkih niti (gornja granica mogućeg broja korisničkih niti može biti ograničena isključivo raspoloživom memorijom);
- alokacije prostora za kontrolni stek niti maksimalne veličine 64KB.

Niti na jeziku C++

Potrebno je koncept niti realizovati klasom `Thread` čija je kompletna definicija zadata u zaglavlju `thread.h` i izgleda ovako:

```
// File: thread.h
#ifndef _thread_h_
#define _thread_h_

typedef unsigned long StackSize;
const StackSize defaultStackSize = 4096;
typedef unsigned int Time; // time, x 55ms
const Time defaultTimeSlice = 2; // default = 2*55ms
typedef int ID;

class PCB; // Kernel's implementation of a user's thread

class Thread {
public:

  void start();
  void waitToComplete();
  virtual ~Thread();

  ID getId();
  static ID getRunningId();
  static Thread * getThreadById(ID id);

protected:
  friend class PCB;
  Thread (StackSize stackSize = defaultStackSize, Time timeSlice =
defaultTimeSlice);
  virtual void run() {}

private:
  PCB* myPCB;
};

void dispatch ();

#endif
```

Klasa `PCB` predstavlja implementaciju niti unutar samog jezgra i njena konstrukcija je u potpunoj nadležnosti samog rešenja. Podatak-član `myPCB` klase `Thread` ukazuje na objekat unutar jezgra koji realizuje datu korisničku nit (zapravo njen PCB). Na ovaj način, implementacija jezgra i korisnička aplikacija potpuno su razdvojene i kompilaciono nezavisne: korisnička aplikacija koristi samo klasu `Thread` i zaglavlje `thread.h` (koje ne treba menjati), tj. kreira objekte klase `Thread`, koji su jedan-na-jedan povezani sa objektima klase `PCB`. Klasa `PCB` je, sa druge strane, u isključivoj nadležnosti implementacije i praktično nedostupna i nevidljiva za korisničku aplikaciju. Sve operacije nad jednom korisničkom niti kao objektom klase `Thread` jezgro treba da realizuje korišćenjem odgovarajućeg objekta klase `PCB` na koga ukazuje `myPCB`.

Parametri konstruktora klase `Thread` koji su opcioni (imaju definisane podrazumevane vrednosti ukoliko se izostave) zadaju veličinu memorijskog prostora koji treba alocirati za kontrolni stek niti, kao i veličinu maksimalnog vremenskog intervala koji se dodeljuje niti pri izvršavanju u umnošcima od 55 milisekundi (drugim rečima, vremenski interval se zadaje kao broj perioda između uzastopnih prekida tajmera).

Nit se kreira kao objekat klase izvedene iz klase `Thread`, a kod niti se zadaje redefinisanjem polimorfne operacije `run()`. Nit se pokreće pozivom operacije `start()`, a završava se implicitno, kada se završi operacija `run()`. Na primer:

```
#include "thread.h"

class Buffer;
class Data;

class Producer : public Thread {
public:
  Producer (Buffer* buf, int numOfItems) : myBuffer(buf), n(numOfItems) {}
  virtual ~Producer() {waitToComplete();}
protected:
  Data* produce();
  virtual void run();
private:
  int n;
  Buffer* myBuffer;
};

void Producer::run () {
  for (int i=0; i<n; i++) {
    Data* d = produce();
    if (myBuffer) myBuffer->put(d);
  }
}

int userMain (int,char*[]) {
  Producer* p = new Producer(...);
  p->start();
  ...

  delete p;  // Here, the destructor invocation ensures
             // that the producer has finished
}
```

Operacija `waitToComplete()` blokira pozivajuću nit sve dok nit predstavljena objektom nad kojim se ova operacija izvršava nije završena. Takođe, destruktor klase `Thread`, osim drugih potrebnih radnji (brisanje niti iz jezgra), treba najpre da izvrši poziv baš ove funkcije `waitToComplete()`, kako nit koja poziva destruktor (u čijem kontekstu se želi brisanje objekta klase `Thread`), ne bi obrisala objekat klase `Thread` pre nego što je njena nit (zapravo funkcija `run()`) stvarno završila (razmisliti i objasniti zašto i kako bi ovde nastao problem).

Promena konteksta i preuzimanje

Opis zadatka

Potrebno je realizovati promenu konteksta (engl. *context switch*) i preuzimanje u sledećim situacijama:

1. Na eksplicitni zahtev niti (sinhrono preuzimanje) pozivom funkcije `dispatch()` koja vrši eksplicitno preuzimanje.
2. Na pojavu prekida (asinhrono preuzimanje), ali samo uz korišćenje koncepta događaja koji će biti opisan naknadno.
3. Zbog operacije na nekoj sinhronizacionoj primitivi, tj. na semaforu ili događaju, kako je opisano kasnije (sinhrono preuzimanje).
4. Po isteku vremenskog intervala dodeljenog niti za izvršavanje (engl. *time slice*). Vremenski interval koji se dodeljuje procesu pri svakom povratku konteksta zadaje se

prilikom kreiranja niti, kao parametar (umnožak od 55ms). Vrednost 0 ovog parametra označava da vreme izvršavanja date niti nije ograničeno, već će se ona izvršavati (kada dobije procesor) sve dok ne dođe do preuzimanja iz nekog drugog razloga.

5. Eksplicitnim pozivom operacije `waitToComplete()`.

Važna napomena:
Ako u cilju realizacije raspodele vremena student preusmerava periodični prekid od hardverskog vremenskog brojača PC računara (prekidi broj 08h ili 1Ch), u redefinisanoj prekidnoj rutini treba obavezno da pozove funkciju:

```
void tick();
```

koja će biti implementirana u test-okruženju. Ovaj poziv treba da obezbedi da i test-okruženje bude "svesno" protoka vremena, pošto će protok vremena biti korišćen u određenim testovima i okruženju za simulaciju I/O uređaja.

Opis okruženja

Samo raspoređivanje, odnosno izbor niti kojoj će biti dodeljen procesor iz skupa spremnih niti, ne treba realizovati, pošto će ovaj modul biti obezbeđen spolja. Drugim rečima, ovaj modul koji će biti obezbeđen spolja sadrži i listu spremnih PCB-ova, kao i sam algoritam raspoređivanja. Deklaracije interfejsa ovog modula nalaze se u zaglavlju `schedule.h` i izgledaju ovako:

```
// File: schedule.h
class PCB;

class Scheduler {
public:
  static void put (PCB*);
  static PCB* get ();
};
```

Operacija `get()` vraća onu nit iz reda spremnih koja je po algoritmu raspoređivanja izabrana da bude sledeća za izvršavanje i izbacuje tu nit iz reda spremnih. Operacija `put()` smešta datu nit u red spremnih.

Posebne napomene

Prilikom realizacije koda za promenu konteksta i ostalih delova jezgra, studenti treba da budu maksimalno pažljivi, jer svaka, pa i najmanja greška može da uzrokuje razne neprijatne situacije u izvršavanju programa (neregularan završetak, blokiranje, pa čak i resetovanje računara). Student treba da bude svestan da je svaka ovakva pojava po pravilu uzrokovana nedostatkom u samoj realizaciji, a ne nekim drugim spoljnim faktorom. Sve ovakve nedostatke studenti treba sami da otkriju i razreše – bilo čija pomoć može samo da olakša rešavanje, ali nikako i da reši problem, ne zbog toga što neko drugi (uključujući i nastavnika i asistenta) ne želi da pomogne, već što jednostavno to nije u stanju. Ovo je zbog toga što je jedino autor ovako složenog programa u najboljoj situaciji da dobro razume njegovu konstrukciju i pronađe njegove nedostatke, a ovakva vrsta programiranja je uvek veoma osetljiva i logički zamršena.

Kada rešava ovakve probleme, student može da koristi pomoćne alatke (npr. debager), ali će mu one, na žalost, često biti od veoma male ili nikakve pomoći, jednostavno zato što su navedeni problemi najčešće asinhroni, sporadični i nereproducibilni (ne mogu se kontrolisano ponavljati). Zbog toga je veoma često najbolji, a ponekad i jedini način da se problem reši pažljiva analiza i razmišljanje o tome šta može biti uzrok problema. Da bi to postigao, student mora jako dobro da poznaje teoriju operativnih sistema, zatim način

funkcionisanja hardvera i softvera sa kojim intereaguje, a potom i da razume svaki detalj sopstvenog rešenja. Često je potrebno pronaći i proučiti dokumentaciju o hardverskom i softverskom okruženju na PC računaru, ali i u glavi simulirati razne scenarije događaja koji se ne mogu tačno reprodukovati u stvarnosti, već samo u mislima, kako bi se uočio problem.

Naredne napomene i preporuke predstavljaju samo deo ovakvih iskustava i činjenica koje treba imati u vidu prilikom realizacije:

1. Prilikom promene konteksta, treba voditi računa da se čuva i restaurira sve što čini kontekst.
2. Kada se preuzima (preusmerava) prekid iz postojećeg sistema, a to posebno važi za preusmeravanje periodičnog prekida sa vremenskog brojača, potrebno je obezbediti sledeće:
   - da se prilikom svakog prekida od vremenskog brojača pored radnji koje je student predvideo da se rade u prekidnoj rutini, obavezno obave i radnje koje sam sistem predviđa na periodični prekid; ovo se može jednostavno uraditi pozivom stare prekidne rutine (na koju je ukazivao stari vektor koji je preusmeren) iz nove prekidne rutine;
   - da se po završetku programa, tj. prilikom gašenja celog programa, restaurira stari vektor, kako bi sistem nastavio da funkcioniše korektno.
3. Potrebno je strogo voditi računa da su sve kritične sekcije samog jezgra propisno zaštićene od asinhronih "upada", tj. od asinhronih prekida.
4. S obzirom na to da će se testiranje vršiti i na virtuelnoj mašini, obezbediti zabranu preuzimanja bez zabrane prekida, npr. na način prikazan u laboratorijskoj vežbi.
5. Treba imati u vidu da bibliotečne funkcije koje su realizovane u standardnim bibliotekama za konzolne aplikacije najčešće nisu predviđene za konkurentan rad, ili da su eventualno bezbedne za konkurentan rad, ali samo ako se koriste procesi i niti samog sistema-domaćina, što u ovom slučaju nije dozvoljeno. Pošto će studentsko rešenje imati sopstvenu realizaciju niti, ove operacije po pravilu nisu bezbedne za konkurentan rad, tj. ne sme se dozvoliti da jedna nit započne bibliotečnu operaciju, a onda dođe do preuzimanja i druga nit započne tu istu operaciju (nisu *reentrant*). Ovo posebno treba voditi računa kod sledećih bibliotečnih operacija:
   - Operacije alokacije memorije (operator `new` na jeziku C++), pošto one pristupaju strukturama podataka ugrađenog alokatora dinamičke memorije izvršnog okruženja jezika.
   - Ulazno/izlazne operacije, recimo operacije ispisa na konzolu (`operator<<` na jeziku C++).

   Mogući pristupi rešavanju ovakvih problema jesu:
   - Obezbediti propisno međusobno isključenje na mestima u jezgru na kojima se ovakve operacije koriste.
   - "Preoteti" opisane operacije, tj. prepisati ih tako da budu međusobno isključive, odnosno "umotati" ih u sopstvene istoimene operacije koje će pozivajući kod pozivati, a koje će obezbediti međusobno isključenje do bibliotečnih realizacija.

   Iako ovo nisu jedini mogući uzroci navedenih problema, oni su najčešći. Studenti treba da obezbede odgovarajuća rešenja, ali tako da rešenja ne umanjuju konkurentnost više nego što je to potrebno. To znači da preuzimanje treba zabraniti samo onda kada je to zaista neophodno, ne više od toga. Zabrana preuzimanja preko potrebne mere smanjuje konkurentnost programa.
6. Poziv virtuelne metode klase za neki objekat nije moguće izvršiti direktno iz koda jezgra, s obzirom da iz globalne funkcije nije moguće pristupiti zaštićenoj metodi neke klase (npr. metodi `run()` klase izvedene iz klase `Thread`), pa je potrebno definisati

statičku metodu klase `PCB` (`PCB` je prijateljska klasa klase `Thread`), čija će se adresa pamtiti na steku date niti i koja će polimorfno izvršavati kod ove metode.
Na primer, na sledeći način:

```
void PCB::wrapper()
{
     PCB::running->myThread->run();
...
```

7. Prilikom testiranja i razvoja sistema obratiti pažnju na sledeće:
   - Po implementaciji svakog podmodula sistema obavezno napisati kratke testove koji testiraju njegovu funkcionalnost, a tek onda, ukoliko je sve u redu, integrisati ga u ostatak sistema. U suprotnom može se dogoditi da se neka trivijalna greška propagira, pa je za njeno otkrivanje kasnije potrebno mnogostruko više vremena nego na samom početku.
   - Pokrenuti javne testove za različite vrednosti ulaznih parametara i pratiti očekivane rezultate.
   - Pre predaje obavezno isprobati svoj projekat i na računarima u laboratorijama na kojima se projekat demonstrira.
8. Obavezno proveriti verziju rešenja namenjenog za predaju kako ne bi bila predata pogrešna verzija.

Na kraju, bez obzira na poznavanje ovih problema, student će morati da uloži mnogo truda da napravi ispravno rešenje. Sigurno će sasvim uobičajena potreba biti nasilno gašenje programa od strane sistema-domaćina, a možda i resetovanje računara. Rešavanjem ovakvih problema na najbolji način se mogu razumeti i naučiti koncepti niti i promene konteksta. Naravno, cilj ovog projekta nije da studente muči bez razloga, već da ih na ovaj način uvede u vode pravog profesionalnog programiranja, čija je svakodnevica puna baš takvih problema!

Koncept semafora

Opis zadatka

Potrebno je realizovati koncept standardnog brojačkog semafora sa operacijama *wait* i *signal* sa vremenski ograničenim intervalom čekanja. Ukoliko je vrednost semafora negativna, tada na semaforu mora biti onoliko blokiranih niti koliko je apsolutna vrednost semafora.

Operacija signal se razlikuje od uobičajene po tome što prihvata i vraća po jedan ceo broj (*int*). Ukoliko je prihvaćeni broj nula, operacija *signal* se ponaša na standardan način, a povratna vrednost je nula. Ako je vrednost prihvaćenog broja pozitivna, treba odblokirati onoliko niti kolika je vrednost prihvaćenog broja. Ukoliko nema toliko blokiranih, već manje, treba odblokirati sve niti na tom semaforu. Vrednost semafora se uvećava za prihvaćeni ceo broj. Povratna vrednost označava broj odblokiranih niti. Ukoliko je prihvaćeni broj negativan, operacija *signal* treba da bude bez dejstva, a povratna vrednost da bude prihvaćeni broj.

Operacija *wait* se razlikuje od uobičajene po tome što prihvata i vraća po jedan ceo broj (*int*). Prihvaćeni ceo broj označava maksimalno trajanje blokiranosti pozivajuće niti (umnožak od 55ms). Zadata vrednost 0 znači da dužina čekanja nije vremenski ograničena. Ukoliko je nit deblokirana zbog isteka vremena, rezultat operacije wait je vrednost 0, dok je u ostalim slučajevima vrednost 1.

Semafor na jeziku C++

Potrebno je koncept semafora realizovati klasom `Semaphore` čija je kompletna definicija zadata u zaglavlju `semaphor.h` i izgleda ovako:

```
// File: semaphor.h
#ifndef _semaphor_h_
#define _semaphor_h_

typedef unsigned int Time;

class KernelSem;

class Semaphore {
public:
  Semaphore (int init=1);
  virtual ~Semaphore ();

  virtual int wait (Time maxTimeToWait);
  virtual int signal(int n=0);

  int val () const;  // Returns the current value of the semaphore

private:
  KernelSem* myImpl;
};

#endif
```

Klasa `KernelSem` predstavlja implementaciju semafora unutar samog jezgra i njena konstrukcija je u potpunoj nadležnosti samog rešenja. Podatak-član `myImpl` klase `Semaphore` ukazuje na objekat unutar jezgra koji realizuje dati semafor, tj. klasa `Semaphore` predstavlja samo omotač (*wrapper*) oko klase `KernelSem` i enkapsulira pozive svih njenih metoda. Na ovaj način, slično kao kod niti, implementacija jezgra i korisnička aplikacija potpuno su razdvojene i kompilaciono nezavisne: korisnička aplikacija koristi samo klasu `Semaphore` i zaglavlje `semaphore.h` (koje ne treba menjati), tj. kreira objekte klase `Semaphore`, koji su jedan-na-jedan povezani sa objektima klase `KernelSem`. Klasa `KernelSem` je, sa druge strane, u isključivoj nadležnosti implementacije i praktično nedostupna i nevidljiva za korisničku aplikaciju. Sve operacije nad jednim korisničkim semaforom kao objektom klase `Semaphore` jezgro treba da realizuje korišćenjem odgovarajućeg objekta klase `KernelSem` na koga ukazuje `myImpl`.

Koncept događaja

Potrebno je realizovati koncept događaja, koji predstavlja binarni semafor na kome može biti blokirana nit, a koji se signalizira prekidom. Drugim rečima, koncept događaja omogućava da se neka nit blokira čekajući na prekid. Na ovaj način može se realizovati koncept *sporadičnog procesa* (*niti*), tj. procesa koji se aktivira na pojavu spoljašnjeg događaja (predstavljenog prekidom), obavlja neku svoju aktivnost, a potom se ponovo blokira čekajući na sledeću pojavu istog događaja. Ukoliko se prekid dešava periodično, onda se isti koncept može iskoristiti za realizaciju *periodičnog procesa* (*niti*).

Pošto prekidna rutina ne može da bude nestatička metoda neke klase, za implementaciju opisanog sistema događaja potrebno je implementirati klasu *IVTEntry* i makro *PREPAREENTRY*. Klasa *IVTEntry* treba da enkapsulira sve podatke i operacije nad tim

podacima koji su potrebni za evidenciju događaja vezanog za neki prekid (tabela prekida ima ukupno 256 ulaza). Ova klasa treba da obezbedi mogućnost dohvatanja pokazivača na objekat *IVTEntry* koji je vezan za zadati ulaz, jer će jedini parametar konstruktora događaja biti redni broj ulaza u IVT. Makro *PREPAREENTRY* treba da za korisnika generiše definicije svih potrebnih podataka i funkcija za jedan ulaz u IVT. Makro treba da prihvata dva parametra: prvi je broj ulaza za koji se generišu definicije i na osnovu kojeg makro svaki put generiše drugačije nazive definisanih promenljivih i funkcija, a drugi logička vrednost koja govori da li treba pozivati staru prekidnu rutinu (1 – treba, 0 – ne treba). Objekat i funkcija koje je neophodno definisati su:

- objekat tipa `IVTEntry` u kojem će biti sačuvani svi potrebni podaci vezani za ulaz za koji se definiše objekat;
- prekidna rutina koja svaki put kada je pozvana treba da pozove signal na događaju koji je vezan za taj ulaz; sve potrebne informacije se čuvaju u objektu tipa `IVTEntry`.

Koncept događaja treba da omogući sledeće radnje:

- inicijalizaciju, zadavanjem broja ulaza u vektor tabeli (IVT) za čiji se prekid vezuje dati događaj; pri inicijalizaciji treba obaviti sve potrebne radnje, između ostalog i inicijalizaciju ili preusmeravanje starog vektora iz datog ulaza; smatrati da se nad jednim ulazom u vektor tabeli može kreirati samo jedan objekat događaja.
- operaciju *wait* kojom se pozivajuća nit blokira; obezbediti da samo nit koja je kreirala događaj može na njemu i da se blokira; poziv ove operacije iz niti koja nije vlasnik događaja nema efekta;
- operaciju *signal* koja deblokira nit blokiranu na događaju i koju treba da pozove prekidna rutina koja je vezana za prekid događaja, a koju obezbeđuje korisnik; preciznije, korisnička prekidna rutina koja je vezana za isti ulaz u IVT za koji je vezan i događaj, treba samo da pozove ovu operaciju *signal* događaja koji odgovara tom prekidu.

Kada se dogodi prekid, treba uvek vršiti preuzimanje (naravno, pod uslovom da se izvršavanje ove prekidne rutine nije ugnezdilo u izvršavanje nekog drugog dela sistemskog koda), ali tako što će se deblokirana nit koja čeka na događaj najpre smestiti u red spremnih (operacijom `Scheduler::put()`), a onda iz reda spremnih uzeti naredna nit za izvršavanje (operacijom `Scheduler::get()`). To znači da se može dogoditi čak i da ista, prekinuta nit nastavi sa izvršavanjem, ili da to bude neka druga nit od onih koje su aktivirane prekidom, što zavisi isključivo od algoritma raspoređivanja koji je nepoznat i nedostupan delu jezgra koji realizuje student, pošto je enkapsuliran (sakriven) iza interfejsa zadatog u zaglavlju `schedule.h`.

Događaj na jeziku C++

Koncept događaja treba realizovati klasom `Event` čija je kompletna definicija zadata u zaglavlju `event.h` i izgleda ovako:

```
// File: event.h
#ifndef _event_h_
#define _event_h_
```

```
typedef unsigned char IVTNo;
class KernelEv;

class Event {
public:
   Event (IVTNo ivtNo);
  ~Event ();

   void wait  ();

protected:
   friend class KernelEv;
   void signal(); // can call KernelEv

private:
   KernelEv* myImpl;
};
#endif
```

Klasa `KernelEv` predstavlja implementaciju događaja unutar samog jezgra i ima isti smisao kao i klasa `KernelSem` za semafor.

Zadatak 2: Signali

Potrebno je realizovati koncept asinhronih signala koje niti mogu slati jedne drugima. Signal nekoj niti može poslati bilo koja nit ili sam sistem. Nit može primiti signal u bilo kom trenutku. Po prijemu signala nit treba u svom kontekstu da obradi signal pomoću funkcija definisanih za obradu tog signala (engl. *signal handlers*). Za vreme obrade jednog signala ne sme da dođe do promene konteksta, dok prekidi moraju biti dozvoljeni. Ukoliko nit primi više signala, potrebno ih je izvršiti redom kojim su pristigli. Niti može da se postavi veći broj funkcija (`SignalHandler`) za obradu svakog od signala koje pri obradi signala treba pozivati u redosledu kojim su registrovane. Ukoliko ne postoji nijedna funkcija za obradu signala, operacija signal nema efekta. Takođe, potrebno je realizovati uklanjanje svih funkcija za obradu određenog signala. Postoji metoda `swap(SignalID id, SignalHandler hand1, SignalHandler hand2)` za zamenu redosleda pozivanja funkcija za obradu određenog signala. Ukoliko prosleđene funkcije nisu registrovane za obradu navedenog signala, metoda `swap` nema efekta. Postoji više mogućih signala koji se mogu poslati niti i oni se identifikuju rednim brojem. Redni brojevi signala kreću se od 0 do 15. Signal može da se blokira na nivou jedne niti i na nivou celog sistema; dok je signal blokiran, odlaže se njegova obrada sve dok signal ne bude odblokiran. Nit koja se kreira nasleđuje sva podešavanja vezana za signale od niti koja ju je napravila.

U zaglavlje `thread.h` treba dodati sledeće:

```
typedef void (*SignalHandler)();

typedef unsigned SignalId;

class Thread {
	...
	void signal(SignalId signal);

	void registerHandler(SignalId signal, SignalHandler handler);
	void unregisterAllHandlers(SignalID id);
	void swap(SignalId id, SignalHandler hand1, SignalHandler hand2);

	void blockSignal(SignalId signal);
	static void blockSignalGlobally(SignalId signal);
	void unblockSignal(SignalId signal);
	static void unblockSignalGlobally(SignalId signal);



}
```

Postoji podrazumevana funkcija za obradu signala za redni broj 0, koja nasilno prekida nit i oslobađa sve resurse koja je ona zauzela. Podrazumevanu funkciju koja obrađuje signal za redni broj 0 treba implementirati. Obezbediti da se signali za redne brojeve 1 i 2 šalju od strane sistema. Signal za redni broj 1 se prosleđuje niti kada se nit koju je ona napravila završi. Signal za redni broj 2 se prosleđuje niti kada se ona završi.

Napomena: Funkcionalnosti opisane u prvom zadatku predstavljaju osnovne funkcionalnosti, stoga kompletna realizacija projektnog zadatka sa ovim skupom funkcionalnosti može maksimalno nositi 20 poena. Funkcionalnosti opisane u drugom zadatku predstavljaju napredne funkcionalnosti, pa kompletna realizacija projektnog zadatka uključujući i ove funkcionalnosti može maksimalno nositi 30 osvojenih poena, odnosno dodatne poene u predroku.

Testovi

Javni testovi

Javni test-program predstavlja sistem od nekoliko proizvođača (engl. *producer*) i jednog potrošača (engl. *consumer*) koji razmenjuju podatke preko ograničenog bafera (engl. *bounded buffer*). Proizvođači i potrošači su realizovani kao niti, a ograničeni bafer kao pasivni objekat (struktura) sa međusobnim isključenjem i uslovnom sinhronizacijom obezbeđenim pomoću semafora. Podatak koji se jedinično upisuje i čita iz bafera jeste jedan znak (`char`).
Proizvođači su sledeći:

- N ($1 \leq N \leq 20$) niti nad istim kodom koje ciklično generišu po jedan znak i stavljaju u bafer.
- Jedna nit koja se aktivira na prekid sa tastature (na svaki pritisak tastera), očitava pritisnuti taster i znak tog tastera stavlja u bafer.

Potrošač ciklično uzima znak po znak iz bafera i ispisuje ga na standardni izlaz.
Kada se na tastaturi pritisne taster *Escape*, program obezbeđuje da se sve niti propisno gase, pri čemu se potrošač gasi tek kada isprazni ceo bafer (pre gašenja ispisuje poruku da je sve regularno završeno).
Ovaj test program testiraće se u dve varijante: kada sve niti imaju neograničen vremenski interval (parametar `timeSlice=0`), i kada sve niti imaju podrazumevani vremenski interval (parametar `timeSlice=defaultTimeSlice`). Javni test program dat je u arhivi `javni_test.zip`.

Tajni testovi

Tajni testovi proveravaju sve aspekte navedenih zahteva, i to:

- propisno kreiranje i gašenje niti;
- ispravno konkurentno izvršavanje niti u slučaju determinističkog izvršavanja (eksplicitno preuzimanje, nema prekida niti isteka vremenskog intervala, deterministički algoritam raspoređivanja, sinhronizacione primitive);
- ispravno konkurentno izvršavanje sa deljenjem vremena u slučaju nedeterminističkog izvršavanja (ograničen vremenski interval, sinhronizacione primitive);
- ispravno aktiviranje niti na prekide.

Testovi koji ispituju slučajeve sa nedeterminističkim ponašanjem (ograničeni vremenski interval, postojanje prekida), mogu i da postepeno povećavaju opterećenje sistema (sve manji vremenski interval, sve češći prekidi), sve do granice do koje sistem to može da obradi. Sistem treba da bude što je moguće otporniji na ovakva opterećenja, u smislu da u slučaju preopterećenja i dalje nastavi da radi i da se propisno gasi, ali sa degradiranom

funkcionalnošću (ne izvršava sve zadate poslove, ne odgovara na sve prekide i slično, ali ipak i dalje radi – ne pada).

Testovi performansi

Testovi performansi 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 PCB, semafore i događaje;
- druge vremenske i prostorne parametre i ograničenja sistema.

Zaključak

Potrebno je realizovati opisane podsisteme jezgra prema datim zahtevima na jeziku C++. Kao integrisano okruženje za razvoj programa (engl. *integrated development environment*, IDE) moguće je koristiti Eclipse sa podešenim bcc toolchain setom ili Borland C++ 3.1,
radi obezbeđenja kompatibilnosti sa testovima.

Pravila za predaju projekta
Projekat se predaje isključivo kao jedna zip arhiva. Sadržaj arhive podeliti u dva foldera: *src* i *h*. U prvom folderu (*src*) treba da budu smešteni svi *.cpp* fajlovi koji su rezultat izrade projekta, a u drugom folderu (*h*) treba da budu svi *.h* fajlovi koji su rezultat izrade projekta. Opisani sadržaj ujedno treba da bude i jedini sadržaj arhive (arhiva ne sme sadržati ni izvršne fajlove, ni biblioteke, ni *.cpp* i *.h* fajlove koji predstavljaju bilo kakve testove, niti bilo šta što iznad nije opisano). Projekat je moguće predati (*upload*) više puta, ali do trenutka koji će preko *e-mail* liste biti objavljen za svaki ispitni rok i koji će uvek biti 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 kolokvijumi su takođe preduslovi za izlazak na ispit). Nakon isteka roka predati projektni zadaci se brišu, pa u slučaju ponovnog izlaska na ispit potrebno je ponovo postaviti ažurnu verziju projektnog zadataka.

Sajt za predaju projekta je http://rti.etf.bg.ac.rs/domaci/index.php?servis=os1_projekat

Nepoštovanje pravila za predaju projekta povlači negativne poene.

Zapisnik revizija

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

Verzija 1.0

Strana	Izmena