

Univerzitet u Beogradu  
Elektrotehnički fakultet



Katedra za računarsku  
tehniku i informatiku

# **Projekat iz predmeta Operativni sistemi 1**

Avgust 2006.

Student:  
Marina Plakalović  
Br. indeksa: 277/04

---

Fajlovi:

def.h .....	1
thread.h .....	2
thread.cpp .....	3
pcb.h .....	4
pcb.cpp .....	5
queue.h .....	7
queue.cpp .....	8
semaphor.h .....	9
semaphor.cpp .....	10
kernsem.h .....	11
kernsem.cpp .....	12
event.h .....	13
event.cpp .....	14
kerne.h .....	15
kerne.cpp .....	16
idle.h .....	17
idle.cpp .....	18
timer.h .....	19
timer.cpp .....	20
main.cpp .....	22

---

```

//def.h

#ifndef _DEF_H_
#define _DEF_H_

#define INT_LOCK {asm{pushf; cli;}} // zabranjuju se prekidi
#define INT_UNLOCK {asm {popf}} //vraca se prethodno stanje

#define INT_TIMER 0x08 // ulaz u IVT za prekid timera
#define INT_MASK 0x0200 // maska za postavljanje bita I na 1

#define NEW 0 // stanja kroz koja prolazi nit
#define RUNNABLE 1
#define RUN 2
#define UNRUNNABLE 3
#define STOP 4

#define MIL_SEC 55 //vreme izmedju dva prekida timera
#define MAX_SIZE 16384

// cuva se vrednost SP-a
#define GET_STACK(tsp, tss) \
    asm{mov tsp, sp; \
        mov tss, ss}

// vraca se sacuvana vrednost SP-a
#define SET_STACK(tsp, tss) \
    asm{mov sp, tsp; \
        mov ss, tss}

// stavlja se PSW na stek i dozvoljavaju prekidi
#define PUSH_PSW \
    asm{pushf; \
        pop ax; \
        or ax, INT_MASK; \
        push ax}

// stavlja se vrednost PC-ja na stek
#define PUSH_PC(tcs, tip) \
    asm{mov ax, tcs; \
        push ax; \
        mov ax, tip; \
        push ax}

// programski dostupni registri na stek
#define PUSH_REG \
    asm{push ax; \
        push bx; \
        push cx; \
        push dx; \
        push es; \
        push ds; \
        push si; \
        push di; \
        push bp}

#endif

```

```

// thread.h

#ifndef _THREAD_H_
#define _THREAD_H_

class PCB;

typedef float Time;
typedef unsigned long StackSize;

const StackSize defaultStackSize = 4096;
const Time defaultTimeSlice = 100.;

class Thread
{
public:
    void start();
    void waitToComplete();

    virtual ~Thread();

protected:
    friend class Timer; //extra
    friend class IdleThread; //extra
    friend class PCB;

    Thread(StackSize stackSize=defaultStackSize, Time timeSlice=defaultTimeSlice);
    virtual void run() {}

private:
    PCB* myPCB;
};

void dispatch();
extern void tick();
Time minTimeSlice();

#endif

```

```

//thread.cpp

#include "def.h"
#include "pcb.h"
#include "schedul.h"
#include "thread.h"
#include "timer.h"
#include <stdlib.h>

Thread::Thread(StackSize stackSize, Time timeSlice) {
    INT_LOCK;
    myPCB = new PCB(this, stackSize, timeSlice);
    INT_UNLOCK;
}

Thread::~~Thread() {
    INT_LOCK;
    waitToComplete();
    if (myPCB!=NULL) delete myPCB;
    INT_UNLOCK;
}

void Thread::start() {
    INT_LOCK;
    if (myPCB->get_status()==NEW) {
        myPCB->set_status(RUNNABLE); // markiranje stanja spremne niti
        myPCB->create_PCB();         // pocetni kontekst
        Scheduler::put(myPCB);       // smesta se u red spremnih
    }
    INT_UNLOCK;
}

void Thread::waitToComplete() {
    INT_LOCK;
    //running ne ceka novu nit, samu sebe, gotovu nit, pocetnu i besposlenu nit
    int pom=(myPCB->get_status()==NEW || myPCB->get_status()==RUN ||
    myPCB->get_status()==STOP || this==PCB::loading || this==PCB::idle);
    if (pom) {INT_UNLOCK; return;}
    PCB::running->set_status(UNRUNNABLE);
    //tekuca nit se blokira i stavlja u red blokiranih
    myPCB->myQueue->put(PCB::running);
    dispatch(); //iz reda spremnih preuzme neka druga nit
    INT_UNLOCK;
}

void dispatch() // promena konteksta niti obavlja u prekidnoj rutini
{
    INT_LOCK;
    //expl odredjuje da li je explicitno trazena promena konteksta
    Timer::expl = 1;
    Timer::timer_INT();
    Timer::expl = 0;
    INT_UNLOCK;
}

void tick();

Time minTimeSlice() {
    return MIL_SEC; // najmanje vreme trajanja niti bez prekida
}

```

```

//pcb.h

#ifndef _PCB_H_
#define _PCB_H_

#include "idle.h"
#include "thread.h"
#include "queue.h"

typedef unsigned long Long;
class Queue;

class PCB
{
public:
    PCB(Thread* myT, StackSize stackSize, Long timeSlice);
    ~PCB();

    void create_PCB(); //kreira pocetni kontekst steka
    void set_status(int s){status=s;} //postavlja stanje niti
    int get_status(){return status;} // vraca stanje niti
    //const int PID() const {return ID;} // vraca identifikaciju niti

    // staticki metod od cije adrese se zapocinje izvršenje instrukcija
    static void run_wrapper();
    inline void run(); // poziva run() metod pridružene niti

    static Thread* loading; //pocetna nit koja se završava kada se završi sistem
    static Idle* idle; //besposlena nit koja se poziva kada je Scheduler prazan
    static PCB* running; // PCB tekuće niti

protected:
    friend class Thread;
    friend void dispatch();
    friend class Timer;

private:
    //static int ukID;
    //int ID;
    Thread* myThread;
    unsigned int sp, ss;
    volatile int status;
    volatile Long period; // vreme proteklo od startovanja niti u milisekundama
    // koliko puta treba da dodje prekid da bi nit bila prekinuta
    volatile int numer;
    Long pcbTimeSlice;
    StackSize pcbStackSize;
    char* mySP;
    Queue* myQueue; // red niti koje čekaju da se završi ova
};

#endif

```

```

//pcb.cpp

#include "def.h"
#include "pcb.h"
#include "schedul.h"
#include "timer.h"
#include <dos.h>
#include <stdlib.h>

Thread* PCB::loading=NULL;
Idle* PCB::idle=NULL;
PCB* PCB::running=NULL;
//int PCB::ukID=0;

PCB::PCB(Thread* myT, StackSize ss, Long ts)
{
    INT_LOCK;
    //ID=++ukID;
    myThread = myT;
    status = NEW;
    period = 0.;
    numer=(int) (pcbTimeSlice/minTimeSlice());
    pcbTimeSlice = ts;
    if(ss>MAX_SIZE)pcbStackSize=MAX_SIZE;
    else pcbStackSize = ss;
    mySP = NULL;
    myQueue= new Queue();
    INT_UNLOCK;
}

PCB::~~PCB()
{
    INT_LOCK;
    if(myQueue!=NULL) delete myQueue;
    if(mySP!=NULL) delete mySP;
    INT_UNLOCK;
}

void PCB::create_PCB()
{
    INT_LOCK;

    static volatile unsigned newSP, newSS, newIP, newCS, oldSS, oldSP;

    mySP =new char[pcbStackSize]; // alocira se parce memorije za stek

    //SP pokazuje na najvisu adresu parceta memorije posto stek raste na dole
    newSP = sp = FP_OFF(mySP + pcbStackSize-1);
    newSS = ss = FP_SEG(mySP + pcbStackSize-1);

    // u PC se stavlja adresa staticke f-je run_wrapper()
    newIP=FP_OFF(&(run_wrapper));
    newCS=FP_SEG(&(run_wrapper));

    GET_STACK(oldSP, oldSS); // sacuva se stari SP
    SET_STACK(newSP, newSS); // SP postavi da pokazuje na novo parce memorije

    PUSH_PSW; // na stek stavlja PSW

    PUSH_PC(newCS, newIP); //PC

```

```

    PUSH_REG;                                //programski dostupni registri

    GET_STACK(newSP, newSS);                  // novi SP pokazuje na vrh novokreiranog steka
    SET_STACK(oldSP, oldSS);                  // stari SP pokazuje tamo gde je i ranije

    sp =newSP;                                // SP PCB-a sada pokazuje na svoj stek
    ss= newSS;

    INT_UNLOCK;
}

void PCB::run_wrapper()
{
    running->run();
}

void PCB::run()
{
    myThread->run();

    INT_LOCK;
    //nakon sto se zavrsi run() metod niti, obavljaju se zavrсна podesavanja:
    set_status(STOP);                          //markiranje završnog statusa

    //sve niti koje su blokirane cekale kraj ove niti se stave u red spremnih
    while(myQueue->size() > 0 )
    {
        PCB* temp =myQueue->get();
        temp->set_status(RUNNABLE);
        Scheduler::put(temp);
    }
    dispatch();
    //prelaskom na drugu nit dejstvo INT_LOCK nije vidljivo,ne treba INT_UNLOCK;
}

```



```

//queue.h

#ifndef _QUEUE_H_
#define _QUEUE_H_

#include "pcb.h"

struct Elem
{
    PCB* pcb;
    Elem* next;
};

class Queue
{
public:
    Queue();
    ~Queue();
    int size() const{return len;} // duzina reda
    void put(PCB*); // stavlja PCB u red
    PCB* get(); // uzme PCB iz reda

private:
    Elem* first;
    int len;

    Queue( const Queue& q){}; // zabranjuje se stvaranje kopije reda
    Queue& operator =(const Queue& q){return *this;};

};

#endif

```

```
//queue.cpp
```

```
#include "def.h"  
#include "timer.h"  
#include "queue.h"  
#include <stdlib.h>
```

```
Queue::Queue() {  
    INT_LOCK;  
    first=0; len=0;  
    INT_UNLOCK;  
}
```

```
Queue::~~Queue() {  
    INT_LOCK;  
    while(first) {  
        Elem *old=first;  
        first=first->next;  
        delete old;  
    }  
    first=0; len=0;  
    INT_UNLOCK;  
}
```

```
void Queue::put(PCB* p) {  
    INT_LOCK;  
    Elem *tek=first, *pret=NULL;  
    while(tek && tek->pcb!=p)  
    {  
        pret=tek; tek=tek->next;  
    }  
    if (tek!=NULL) {INT_UNLOCK; return;}  
    Elem* newer=new Elem();  
    if(!first)  
    {  
        first=newer;  
        first->next=NULL;  
        first->pcb=p;  
    } else  
    {  
        pret->next=newer;  
        pret->next->next=NULL;  
        pret->next->pcb=p;  
    }  
    len++;  
    INT_UNLOCK;  
}
```

```
PCB* Queue::get() {  
    if(first==NULL) return 0;  
    INT_LOCK;  
    len--;  
    Elem* temp = first;  
    PCB* p=temp->pcb;  
    first = first->next;  
    INT_UNLOCK;  
    delete temp;  
    return p;  
}
```

```
//semaphor.h

#ifndef _SEMAPHOR_H_
#define _SEMAPHOR_H_

extern int semPreempt;

class KernelSem;

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

    void wait ();
    void signal();

    int val () const; // Returns the current value of the semaphore
private:
    KernelSem* myImpl;
};

#endif
```

```
//semaphor.cpp

#include "def.h"
#include "kernsem.h"
#include "semaphor.h"
#include <stdlib.h>

int semPreempt = 0;

Semaphore::Semaphore(int init)
{
    INT_LOCK;
    myImpl = new KernelSem(init);
    INT_UNLOCK;
}

Semaphore::~Semaphore()
{
    INT_LOCK;
    if (myImpl!=NULL) delete myImpl;
    INT_UNLOCK;
}

void Semaphore::wait()
{
    INT_LOCK;
    myImpl->wait();
    INT_UNLOCK;
}

void Semaphore::signal()
{
    INT_LOCK;
    myImpl->signal();
    INT_UNLOCK;
}

int Semaphore::val() const
{
    return myImpl->val;
}
```

```
//kernsem.h

#ifndef _KERNSEM_H_
#define _KERNSEM_H_

#include "queue.h"

class Queue;

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

    void sleep();
    void wait();
    void wakeup();
    void signal();
    void signalAll();

protected:
    friend class Semaphore;

private:
    int val;
    Queue* semQueue;
};

#endif
```

```

//kernsem.cpp

#include "kernsem.h"
#include "schedul.h"
#include "timer.h"
#include "pcb.h"

KernelSem::KernelSem(int init) {
    val = init;
    semQueue=new Queue();
}

KernelSem::~KernelSem() {
    if (semQueue!=NULL){ // neregularna situacija
        signalAll(); // ukoliko jos niti ceka na semaforu, stave se u red spremnih
        delete semQueue;
    }
}

void KernelSem::wait(){
    if(--val<0) // ukoliko val<0 nit koja poziva wait() se blokira na semaforu
        sleep();
    else if(semPreempt)
        dispatch();
}

void KernelSem::sleep() {
    (PCB::running)->set_status(UNRUNNABLE);
    semQueue->put(PCB::running); //red niti blokiranih na semaforu
    dispatch();
}

void KernelSem::signal(){
    if(val++<0)
        wakeup(); // niti se oslobadjaju sev dok ih ima blokiranih
    else if (semPreempt)
        dispatch();
}

void KernelSem::wakeup() {
    PCB* t = semQueue->get();
    t->set_status(RUNNABLE);
    Scheduler::put(t);
}

void KernelSem::signalAll() {
    INT_LOCK;
    while (semQueue->size()>0)
    {
        PCB* p = semQueue->get();
        Scheduler::put(p);
    }
    INT_UNLOCK;
}

```

```
//event.h

#ifndef _EVENT_H_
#define _EVENT_H_

typedef unsigned int IVTNo;
typedef void interrupt (*InterruptHandler) (...);

class KernelEv;

class Event {
public:
    Event (IVTNo, InterruptHandler);
    ~Event ();

    void wait ();
    void signal();

    void oldHandler (); // Simply calls the old handler
private:
    KernelEv* myImpl;
};

#endif
```

```
//event.cpp

#include "def.h"
#include "event.h"
#include "kerne.h"

Event::Event(IVTNo n, InterruptHandler h)
{
    INT_LOCK;
    myImpl = new KernelEv(n, h);
    INT_UNLOCK;
}

Event::~Event()
{
    INT_LOCK;
    delete myImpl;
    INT_UNLOCK;
}

void Event::wait()
{
    INT_LOCK;
    myImpl->wait();
    INT_UNLOCK;
}

void Event::signal()
{
    INT_LOCK;
    myImpl->signal();
    INT_UNLOCK;
}

void Event::oldHandler()
{
    myImpl->oldHandler();
}
```



```
//kernevent.h

#ifndef _KERNEVENT_H_
#define _KERNEVENT_H_

#include "event.h"
#include "queue.h"

class Queue;

class KernelEv {
public:
    KernelEv(IVTNo, InterruptHandler);
    ~KernelEv();

    void wait();
    void signal();

protected:
    friend class Event;

private:
    IVTNo ivtEntry;
    InterruptHandler oldHandler;

    Queue* evQueue;
};

#endif
```

```

//kernev.cpp

#include "kernev.h"
#include "thread.h"
#include "timer.h"
#include "pcb.h"
#include "schedul.h"
#include <dos.h>
#include <stdlib.h>

KernelEv::KernelEv(IVTNo n, InterruptHandler h)
{
    evQueue=new Queue();
    ivtEntry = n;
    oldHandler = getvect(n); // sacuva se pr. rutina koja odgovara ulazu n
    setvect(n, h);           // na njeno mesto postavi nova h
}

KernelEv::~KernelEv()
{
    setvect(ivtEntry, oldHandler); // restauracija stare prekidne rutine
    if(evQueue!=NULL) delete evQueue;
}

void KernelEv::wait()
{
    (PCB::running)->set_status(UNRUNNABLE);
    // svaka nit koja pozove wait() stavi se u red blokiranih da ceka dogadjaj
    evQueue->put(PCB::running);
    dispatch();
}

// desio se dogadjaj, sve niti koje su ga cekale mogu da nastave rad
void KernelEv::signal()
{
    while( evQueue->size() > 0 )
    {
        PCB* temp = evQueue->get();
        temp->set_status(RUNNABLE);
        Scheduler::put(temp);
    }
    dispatch();
}

```

```
//idle.h

#ifndef _IDLETHR_H_
#define _IDLETHR_H_

#include "thread.h"

class Thread;

class Idle: public Thread { //besposlena nit
public:
    Idle();
    virtual void run();
    void start();
};

#endif
```

```
//idle.cpp

#include "def.h"
#include "idle.h"
#include "pcb.h"
#include "timer.h"

Idle::Idle(): Thread(256, minTimeSlice()) {}

void Idle::run()
{
    while (1);           // ne radi nista, vrti se u beskonacnoj petlji
}

void Idle::start()
{
    INT_LOCK;
    //ne stavlja u red spremnih vec explicitno poziva je red spremnih prazan
    myPCB->set_status(RUNNABLE);
    myPCB->create_PCB();
    INT_UNLOCK;
}
```

```
//timer.h

#ifndef _TIMER_H_
#define _TIMER_H_

#include "event.h"
#include "idle.h"
#include "pcb.h"
#include "thread.h"

//fiktivna klasa koja enkapsulira radnje potrebne za startovanje sistema
class Timer
{
public:
    Timer();
    ~Timer();

    static void inic();
    static void restore();

    static void dispatch();
    static volatile int expl;

    static void interrupt (*oldRoutine) (...);
    static void interrupt timer_INT (...);
};

#endif
```

```

//timer.cpp

#include "def.h"
#include "kerne.h"
#include "pcb.h"
#include "schedul.h"
#include "timer.h"
#include <dos.h>
#include <stdlib.h>

volatile int Timer::expl = 0;
void interrupt (*Timer::oldRoutine) (...) = NULL;

Timer::Timer()
{
    INT_LOCK;
    inic();
    PCB::loading = new Thread(256, minTimeSlice());
    // nit koja zapocinje do prve promene konteksta
    PCB::loading->myPCB->set_status(RUN);
    PCB::running = (PCB*) PCB::loading->myPCB;
    PCB::idle = new Idle();
    PCB::idle->start();
    INT_UNLOCK;
}

Timer::~Timer()
{
    INT_LOCK;
    restore();
    delete PCB::idle;
    delete PCB::loading;
    INT_UNLOCK;
}

void Timer::inic()
{
    INT_LOCK;
    oldRoutine = getvect(INT_TIMER); // sacuva se stara prekidna rutina
    setvect(INT_TIMER, timer_INT); //na mesto prekidne rutine timera stavlja
    //redefinisana prekidna rutina koja obavlja promenu konteksta
    INT_UNLOCK;
}

void Timer::restore()
{
    INT_LOCK;
    setvect(INT_TIMER, oldRoutine); // vraca se sve kao sto je i bilo
    INT_UNLOCK;
}

```

```

void interrupt Timer::timer_INT(...) // redefinisana prekidna rutina
{
    static unsigned int tsp, tss;
    static PCB *newPCB;

    if(!expl ) // provearava se da li je explicitno pozvan dispatch()
    {
        tick();
        // ako nije poziva se tick() i stara prekidna rutina za svaki slucaj
        oldRoutine();

        // ukoliko data nit ne treba da se prekida ili ukoliko nije jos doslo
        // vreme za prekid izlazi se iz rutine
        if ( PCB::running->period < PCB::running->pcbTimeSlice
            || PCB::running->pcbTimeSlice == 0) return;
        else{
            PCB::running->period += (Long)minTimeSlice();
            // u suprotnom povecava vreme proteklo od startovanja niti
            PCB::running->numer--; } // smanjuje brojac prekida
        }
        expl=0; // za svaki slucaj

        if(PCB::running->get_status()==RUN && PCB::running != PCB::idle->myPCB ){
            PCB::running->set_status(RUNNABLE);
            Scheduler::put((PCB*)PCB::running); }
            // tekuca nit se stavlja u red onih koje cekaju na procesor

        newPCB = Scheduler::get(); // iz reda spremnih se uzima nova nit
        if (newPCB == NULL) newPCB = PCB::idle->myPCB;
        //ako je prazan Scheduler koristimo idle nit
        newPCB->set_status(RUN); // nova nit postaje aktivna nit

        //izvrsi se zamena konteksta tako sto se razmene pokazivaci na stek
        GET_STACK(tsp, tss);

        PCB::running->sp = tsp;
        PCB::running->ss = tss;
        PCB::running = newThread; // running je sad nova nit
        tsp = PCB::running->sp;
        tss = PCB::running->ss;
        // startovanjem nove niti njeno proteklo vreme se postavlja na 0
        PCB::running->period= 0.;

        SET_STACK(tsp, tss);

        return;
    }
}

```

```
//main.cpp

#include "timer.h"
#include "def.h"
#include "thread.h"

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

int main(int argc, char* argv[]) {
    Timer* t=new Timer();
    int value = userMain(argc, argv);
    delete t;
    return value;
}
```