

Електротехнички факултет у Београду

Пројекат из предмета Оперативни системи 1

професор
др Драган Милићев

датум предаје
30. мај 2005.

студент
Тинтор Марко 03/127

Напомене

Претпоставке:

- Scheduler::get враћа 0 ако нема спремних нити

У класу Thread је додато следеће:

- ИМЕ НИТИ

```
Thread(const char* name, StackSize stackSize=defaultStackSize, Time  
timeSlice=defaultTimeSlice);  
const char* getName();
```

- дохватање нити која се тренутно извршава

```
static Thread* current();
```

- забрана копирања

```
private: Thread(Thread&) {}
```

У класу Semaphore је додато следеће:

- пропуштање свих нити које чекају на семафору

```
void signalAll();
```

У класу Event је додато следеће:

- број позива прекидне рутине почев од стварања објекта класе Event (потребно за домаћи 2)

```
unsigned long count() const;
```

Дотате су следеће класе за синхронизацију:

Mutex - семафор који пропушта највише једну нит

RWMutex – семафор који може да пропусти једног писача или више читача

Barrier - семафор који увек блокира

FastSem – ефикаснија варијанта класе Semaphore

класе за закључавање и откључавање мутекса

Lock<Mutex>

Unlock<Mutex>

Тестови се налазе у user\user.cpp

TEST_BASIC	две нити које штампају знакове на екран
TEST_SEM	кружни бафер, прве две нити га пуне, друге две нити га празне
TEST_EVENT	нит чека на притиске тастера са тастатуре
TEST_ASYNC	две нити које се синхронизују преко упосленог чекања, једини начин промене нити је преко часовника
TEST_200	200 нити које штампају бројеве на екран

изворни код:

user\thread.h	4
kthread.h	5
thread.cpp	7
user\queue.h	10
arch.h	11
arch.cpp	12
main.cpp	14
user\sync.h	15
user\semaphor.h	17
sync.cpp	18
user\event.h	22
event.cpp	23
std.h	25

user\thread.h

```
#ifndef __USER_THREAD_H__
#define __USER_THREAD_H__

typedef float Time; // time in milliseconds
typedef unsigned long StackSize;

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

class KThread;

class Thread
{
public:
    Thread(StackSize stackSize=defaultStackSize, Time
timeSlice=defaultTimeSlice);
    virtual ~Thread();

    void start();
    void waitToComplete();

    // EXTRA!
    Thread(const char* name, StackSize stackSize=defaultStackSize, Time
timeSlice=defaultTimeSlice);

    const char* getName();
    static Thread* current();

    void join() {waitToComplete();}

protected:
    virtual void run() = 0;

private:
    friend class KThread;
    KThread* kthread;

    // non-copyable object
    Thread(Thread&) {}
};

void dispatch();
Time minTimeSlice();

#endif
```

kthread.h

```
#ifndef __KERNEL_THREAD_H__
#define __KERNEL_THREAD_H__

#include "user\queue.h"
#include "user\thread.h"

#include "arch.h"
#include "std.h"
#include "schedule.h"

enum ThreadState {Created, Active, Blocked, Finished};

struct KThread: public ListItem
{
    KThread(const char* n): state(Created), name(n),
        stack(0), maxTimeSlice(0) {}
    KThread(const char* n, int stackSize, Time timeSlice, Thread* thread);

protected:
    friend class Core;
    friend class Thread;

    static void main();

    // stack
    unsigned ss, sp;
    byte* stack;

    unsigned maxTimeSlice; // in TIMER_PERIOD milliseconds
    ThreadState state;

    // list of threads waiting for this thread to complete
    Queue waitList;
    // interface object
    Thread* thread;
    // thread name
    const char* name;
};

struct IdleThread: public KThread
{
    IdleThread();
    static void main();
};

class PCB;
```

```

struct Core
{
    Core();

    void mainloop();

    KThread* getRunning() const {return running;}
    void switchThread();
    void xdispatch() {in_dispatch=1; timerISR();}

    void activate(KThread *a)
    {
        assert(a);
        assert(a->state==Blocked || a->state==Created);
        a->state = Active;
        Scheduler::put((PCB*)a);
    }

    // block current running thread
    void block(Queue& queue)
    {queue.push(running); blocked++; running->state=Blocked; xdispatch();}
    // unblock one thread from queue
    void unblock(Queue& queue)
    {assert(queue); blocked--; activate((KThread*)queue.pop());}

    // finish running thread
    void finish();

    static void interrupt timerISR();

private:
    volatile short in_dispatch;
    ISR oldTimerISR;

    // currently running thread
    KThread* running;
    // number of currently blocked threads
    int blocked;
    // initial and idle thread
    KThread init, idle;
    // time from start (in TIMER_PERIOD milliseconds)
    ulong ticks;
    // time of next switch (in TIMER_PERIOD milliseconds)
    ulong switchTime;
};

extern Core core;

#endif

```

thread.cpp

```
#include "user\event.h"
#include "user\thread.h"
#include "user\semaphor.h"
#include "user\sync.h"
#include "user\queue.h"

#include "kthread.h"
#include "arch.h"
#include "std.h"

Time minTimeSlice() {return TIMER_PERIOD;}

Core core;

// -----

Thread::Thread(StackSize stackSize, Time timeSlice)
{
    INT_DISABLE;
    kthread = new KThread("default", stackSize, timeSlice, this);
    assert(kthread);
}

Thread::Thread(const char* n, StackSize stackSize, Time timeSlice)
{
    INT_DISABLE;
    kthread = new KThread(n, stackSize, timeSlice, this);
    assert(kthread);
}

Thread::~~Thread()
{
    waitToComplete();
    INT_DISABLE;
    delete kthread;
    INT_ENABLE;
}

void Thread::start()
{
    INT_DISABLE;
    if(kthread->state == Created) core.activate(kthread);
    INT_ENABLE;
}

const char* Thread::getName()
{
    return kthread->name;
}

Thread* Thread::current()
{
    return core.getRunning()->thread;
}
```

```

void Thread::waitToComplete()
{
    INT_DISABLE;
    if(kthread->state != Finished && kthread->state != Created)
        core.block(kthread->waitList);
    INT_ENABLE;
}

// -----

KThread::KThread(const char* n, int stackSize, Time timeSlice, Thread* _thread)
    : finishing(0), state(Created), name(n)
{
    INT_ENABLE;

    maxTimeSlice = timeSlice==0.0f ? 0xFFFF :
(unsigned)Min((long)(timeSlice/TIMER_PERIOD + 0.5), (long)0xFFFF);
    thread = _thread;

    stack = createStack(stackSize, KThread::main);
    ss = mySS; sp = mySP;
}

void KThread::main()
{
    Thread::current()->run();

    INT_DISABLE;
    core.finish();
    core.xdispatch();
}

// -----

void dispatch()
{
    INT_DISABLE;
    core.xdispatch();
    INT_ENABLE;
}

void idleMain()
{
    while(1) asm hlt;
}

```

```

Core::Core(): running(&init), init("init"), idle("idle"), blocked(0), ticks(0),
switchTime(0), in_dispatch(0)
{
    idle.stack = createStack(64, idleMain);
    idle.ss = mySS; idle.sp = mySP;
    idle.wakeUpTime = INF_TIME;
}

void Core::finish()
{
    running->state = Finished;
    while(running->waitList) unblock(running->waitList);
}

void Core::switchThread()
{
    running->sp = mySP;
    running->ss = mySS;

    if(running->state==Active) Scheduler::put((PCB*)running);

    running = (KThread*)Scheduler::get();
    if(!running) running = (blocked > 0 ? &idle : &init);

    mySP = running->sp;
    mySS = running->ss;

    switchTime = ticks + running->maxTimeSlice;
}

void Core::mainloop()
{
    INT_DISABLE;
    oldTimerISR = overrideISR(TIMER_IVTN, timerISR);
    xdispatch();
    overrideISR(TIMER_IVTN, oldTimerISR);
    INT_ENABLE;
}

void tick();

void interrupt Core::timerISR()
{
    if(core.in_dispatch || ++core.ticks >= core.switchTime)
    {
        GET_STACK;
        core.switchThread();
        SET_STACK;
    }
    if(!core.in_dispatch) {tick(); core.oldTimerISR();}
    else core.in_dispatch = 0;
}

```

user\queue.h

```
#ifndef __QUEUE_H__
#define __QUEUE_H__

#include "std.h"

class ListItem
{
public:
    ListItem(): next(0) {}
    ListItem* getNext() {return next;}

private:
    ListItem* next;

    friend class Queue;
};

struct Queue
{
    Queue() {first=last=0;}
    operator bool() const {return first!=0;}

    // add to end
    void push(ListItem* a)
    {
        assert(a);
        assert(!a->next);
        if(last) last->next = a; else first = a;
        last = a;
    }

    // remove from start
    ListItem* pop()
    {
        assert(first);
        ListItem* a = first;
        if(!a->next) last = 0;
        first = a->next; a->next = 0;
        return a;
    }

protected:
    ListItem *first, *last;
};

#endif
```

arch.h

```
// architecture specific code goes here

#ifndef __KERNEL_ARCH_H__
#define __KERNEL_ARCH_H__

#include "std.h"

// -----

#define TIMER_PERIOD 55 // milliseconds
#define TIMER_IVTN 0x08

#define INT_ENABLE asm sti
#define INT_DISABLE asm cli

/* BCC 3.1 BUG
struct IntDisable
{
    IntDisable() {INT_DISABLE;}
    ~IntDisable() {INT_ENABLE;}
};*/

// -----

typedef void (interrupt *ISR)();
ISR overrideISR(unsigned n, ISR newISR);

// -----

#define FLAG_INT 0x00000200 // interrupt flag in status register

extern unsigned mySS, mySP;
// SIDE EFFECT: writes to mySS and mySP
byte* createStack(unsigned long stackSize, void(*mainFunc)());

#define GET_STACK asm {mov mySP, sp; mov mySS, ss}
#define SET_STACK asm {mov sp, mySP; mov ss, mySS}

// -----

#endif
```

arch.cpp

```
// architecture specific code goes here

#include "arch.h"
#include <dos.h>

ISR overrideISR(unsigned n, ISR newISR)
{
    unsigned newSEG = FP_SEG(newISR), newOFF = FP_OFF(newISR), oldSEG, oldOFF;
    n *= 4;

    asm {
        push bx
        push ax
        push es

        mov ax, 0
        mov es, ax
        mov bx, word ptr n

        // sacuvaj staru
        mov ax, word ptr es:bx
        mov word ptr oldOFF, ax
        mov ax, word ptr es:bx+2
        mov word ptr oldSEG, ax

        // podesi novu
        mov ax, word ptr newOFF
        mov word ptr es:bx, ax
        mov ax, word ptr newSEG
        mov word ptr es:bx+2, ax

        pop es
        pop ax
        pop bx
    }

    return (ISR)MK_FP(oldSEG,oldOFF);
}
```

```

static unsigned newCS, newIP, newSP, newSS, oldSS, oldSP;
unsigned mySS, mySP;

byte* createStack(unsigned long stackSize, void(*func)())
{
    newCS = FP_SEG((void*)func);
    newIP = FP_OFF((void*)func);

    byte* stack = new byte[stackSize];
    newSS = FP_SEG(stack+stackSize);
    newSP = FP_OFF(stack+stackSize);

    GET_STACK;
    asm {
        mov ss, newSS
        mov sp, newSP

        pushf
        pop ax
        or ax, FLAG_INT
        push ax

        push newCS
        push newIP

        mov ax, 0
        push ax
        push bx
        push cx
        push dx
        push es
        push ds
        push si
        push di
        push bp
        // dec sp, 18

        mov newSP, sp
    }
    SET_STACK;

    mySS = newSS;
    mySP = newSP;
    return stack;
}

```

main.cpp

```
#include "user\thread.h"
#include "kthread.h"
#include "std.h"

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

class MainThread: public Thread
{
public:
    MainThread(int Argc, char** Argv): Thread("main"),
        argc(Argc), argv(Argv) {}
    int result() {return argc;}

private:
    void run() {argc = userMain(argc, argv);}
    int argc;
    char** argv;
};

int main(int argc, char* argv[])
{
    MainThread mt(argc, argv);
    mt.start();
    core.mainloop();
    return mt.result();
}
```

user\sync.h

```
#ifndef __USER_SYNC_H__
#define __USER_SYNC_H__

#include "user\queue.h"

// -----

class Mutex
{
public:
    Mutex(): locked(0) {}
    ~Mutex() {assert(!waitList);}

    void lock();
    bool tryLock();
    void unlock();

private:
    int locked;
    Queue waitList;
};

class RWMutex
{
public:
    RWMutex(): access(0) {}
    ~RWMutex() {assert(!writerList && !readerList);}

    void readLock();
    bool tryReadLock();
    void readUnlock();

    void writeLock();
    bool tryWriteLock();
    void writeUnlock();

private:
    int access;
    // if access>0 then access readers
    // if access<0 then 1 writers
    Queue writerList, readerList;
};

class Barrier
{
public:
    ~Barrier() {assert(!waitList);}

    void wait();
    void signal();
    void signalAll();

private:
    Queue waitList;
};
```

```
class FastSem
{
public:
    FastSem(int init=1): value(init) {}
    ~FastSem() {assert(!waitList);}

    void wait();
    void signal();
    void signalAll();

private:
    int value;
    Queue waitList;
};

#endif
```

user\semaphor.h

```
#ifndef __USER_SEMAPHORE_H__
#define __USER_SEMAPHORE_H__

extern int semPreempt;

class KSemaphore;

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

    void wait();
    void signal();

    int val() const;

    // EXTRA!
    void signalAll(); // signals to all threads that are waiting on sem
    void lock() {void wait();}
    void unlock() {void signal();}

private:
    KSemaphore* sem;
};

template<class Mutex>
struct Lock
{
    Lock(Mutex& mutex): m(mutex) {m.lock();}
    ~Lock() {m.unlock();}
private:
    Mutex& m;
};

template<class Mutex>
struct Unlock
{
    Unlock(Mutex& mutex): m(mutex) {m.unlock();}
    ~Unlock() {m.lock();}
private:
    Mutex& m;
};

#endif
```

sync.cpp

```
#include "user\semaphor.h"
#include "user\event.h"
#include "user\sync.h"
#include "user\queue.h"

#include "kthread.h"

// -----

int semPreempt = 0;

struct KSemaphore
{
    int value;
    Queue waitList; // first in first out
};

Semaphore::Semaphore(int init)
{
    INT_DISABLE;
    sem = new KSemaphore;
    INT_ENABLE;
    sem->value = init;
}

Semaphore::~Semaphore()
{
    INT_DISABLE;
    Assert(!sem->waitList);
    delete sem;
    INT_ENABLE;
}

void Semaphore::wait()
{
    INT_DISABLE;
    if(--sem->value < 0) core.block(sem->waitList);
    else if(semPreempt) core.xdispatch();
    INT_ENABLE;
}

void Semaphore::signal()
{
    INT_DISABLE;
    if(++sem->value <= 0) core.unblock(sem->waitList);
    if(semPreempt) core.xdispatch();
    INT_ENABLE;
}

int Semaphore::val() const
{
    INT_DISABLE;
    int a = sem->value;
    INT_ENABLE;
    return a;
}
```

```

void Semaphore::signalAll()
{
    INT_DISABLE;
    while(sem->value < 0)
    {
        sem->value++;
        core.unblock(sem->waitList);
    }
    assert(!sem->waitList);
    if(semPreempt) core.xdispatch();
    INT_ENABLE;
}

// -----

void Mutex::lock()
{
    INT_DISABLE;
    if(locked) core.block(waitList); else locked = 1;
    INT_ENABLE;
}

void Mutex::unlock()
{
    INT_DISABLE;
    if(waitList) core.unblock(waitList); else locked = 0;
    INT_ENABLE;
}

bool Mutex::tryLock()
{
    INT_DISABLE;
    bool a = locked ? 0 : (locked=1);
    INT_ENABLE;
    return a;
}

// -----

void RWMutex::readLock()
{
    INT_DISABLE;
    if(access < 0) core.block(readerList); else access++;
    INT_ENABLE;
}

bool RWMutex::tryReadLock()
{
    INT_DISABLE;
    bool a;
    if(access < 0) a=0; else {access++; a=1;}
    INT_ENABLE;
    return a;
}

```

```

void RWMutex::readUnlock()
{
    INT_DISABLE;
    if(access > 1) access--;
    else if(writerList && access > 0)
    {
        access = -1;
        core.unblock(writerList);
    }
    INT_ENABLE;
}

void RWMutex::writeLock()
{
    INT_DISABLE;
    if(access != 0) core.block(writerList); else access--;
    INT_ENABLE;
}

bool RWMutex::tryWriteLock()
{
    INT_DISABLE;
    bool a;
    if(access != 0) a=0; else {access--; a=1;}
    INT_ENABLE;
    return a;
}

void RWMutex::writeUnlock()
{
    INT_DISABLE;
    if(access < 0)
        if(writerList) core.unblock(writerList);
        else if(readerList)
        {
            core.unblock(readerList);
            access = 1;
        }
        else access = 0;
    INT_ENABLE;
}

// -----

void Barrier::wait()
{
    INT_DISABLE;
    core.block(waitList);
    INT_ENABLE;
}

void Barrier::signal()
{
    INT_DISABLE;
    if(waitList) core.unblock(waitList);
    INT_ENABLE;
}

```

```

void Barrier::signalAll()
{
    INT_DISABLE;
    while(waitList) core.unblock(waitList);
    INT_ENABLE;
}

// -----

void FastSem::wait()
{
    INT_DISABLE;
    assert(value>0 || !waitList);
    if(--value < 0) core.block(waitList);
    assert(value>0 || !waitList);
    INT_ENABLE;
}

void FastSem::signal()
{
    INT_DISABLE;
    assert(value>0 || !waitList);
    if(++value <= 0) core.unblock(waitList);
    assert(value>0 || !waitList);
    INT_ENABLE;
}

void FastSem::signalAll()
{
    INT_DISABLE;
    assert(value>0 || !waitList);
    if(value < 0)
    {
        while(waitList) core.unblock(waitList);
        value = 0;
    }
    assert(!waitList);
    INT_ENABLE;
}

```

user\event.h

```
#ifndef __USER_EVENT_H__
#define __USER_EVENT_H__

typedef unsigned int IVTNo;

class KEvent;

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

    void wait();

    // EXTRA!
    // koliko se prekida dogodilo?
    unsigned long count() const;

private:
    KEvent* ev;
};

#endif
```

event.cpp

```
#include "user\event.h"
#include "user\queue.h"

#include "kthread.h"

// -----

struct KEvent
{
    KEvent(ISR NewISR): ref(0), newISR(NewISR), count(0) {}
    ~KEvent() {if(ref > 0) overrideISR(ivtn, oldISR);}

    KEvent* get(IVTNo n)
    {
        if(ref > 0 && ivtn != n) return 0;
        if(ref == 0) oldISR = overrideISR(ivtn = n, newISR);
        ref++;
        return this;
    }

    void release()
    {
        assert(ref > 0);
        if(--ref==0) overrideISR(ivtn, oldISR);
    }

    void wait()
    {
        core.block(waitList);
    }

    void notify();

    unsigned long count;

private:
    ISR newISR, oldISR;
    IVTNo ivtn;
    // number of Event objects connected to this object
    unsigned ref;
    // list of threads waiting on this event
    Queue waitList;
};

void KEvent::notify()
{
    #ifdef LOG
    printf("interrupt %Xh\n", ivtn);
    #endif
    count++;
    while(waitList) core.unblock(waitList);
    oldISR();
}
```

```

void interrupt eventISR0();
void interrupt eventISR1();
void interrupt eventISR2();
void interrupt eventISR3();
void interrupt eventISR4();

static KEvent event[] = {
    KEvent(eventISR0), KEvent(eventISR1), KEvent(eventISR2),
    KEvent(eventISR3), KEvent(eventISR4)
};
#define EVENT_COUNT (sizeof(event)/sizeof(event[0]))

void interrupt eventISR0() {event[0].notify();}
void interrupt eventISR1() {event[1].notify();}
void interrupt eventISR2() {event[2].notify();}
void interrupt eventISR3() {event[3].notify();}
void interrupt eventISR4() {event[4].notify();}

// -----

Event::Event(IVTNo ivtn): ev(0)
{
    INT_DISABLE;
    for(unsigned i=0; i<EVENT_COUNT && !ev; i++) ev = event[i].get(ivtn);
    INT_ENABLE;
    if(!ev)
    {
        INT_DISABLE;
        printf("ERROR: out of event interrupt service routines!");
        exit(1);
    }
}

// unblock waiting thread
Event::~Event()
{
    INT_DISABLE;
    ev->release();
    INT_ENABLE;
}

unsigned long Event::count() const
{
    INT_DISABLE;
    unsigned long c = ev->count;
    INT_ENABLE;
    return c;
}

void Event::wait()
{
    INT_DISABLE;
    ev->wait();
    INT_ENABLE;
}

```

std.h

```
#ifndef __STANDARD_H__
#define __STANDARD_H__

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

typedef int bool;
typedef unsigned char byte;
typedef unsigned long ulong;

inline int Min(int a, int b) {return a < b ? a : b;}
inline int Max(int a, int b) {return a > b ? a : b;}

inline long Min(long a, long b) {return a < b ? a : b;}
inline long Max(long a, long b) {return a > b ? a : b;}

// ne rade u BCC 3.1!
//template<class T> T Min(T a, T B) {return a<b ? a : b;}
//template<class T> T Max(T a, T B) {return a>b ? a : b;}

#endif
```