Contents

1 README

```
roeia1, maayanimi
Roei Avrahami (304986714), Ma'ayan Mizrahi (304901259)
EX: 2

FILES:
uthreads.cpp - implementation of uthreads.hh
Thread.hh    - a class representing a single thread.
Thread.cpp   - implementation of Thread.hh
Makefile     - generating the static library libuthreads.a.
README       - this file.

REMARKS:
using principles seen in class and demos, we implemented the desired interface.
for that purpose we used a class Thread, containing all relevant information and
data for a single thread.

database choices: the threads themselves are kept in a map. as indices we use
the threads' id's. all the other containers keep only integers - one integer for
current running thread, and 4 lists for each priority queue and blocked queue. list
was chosen as it is easy to implement a queue with it, with access to the first
element, and ability to push at the end. it is also useful as a pool, with
access to any other element by searching one of its attributes (in our case,
id).

the implementation is rather simple, using few private methods in addition to
those described in the interface. the switchThread function carry most of
the work, picking next thread to run from the priority queues and switch the
threads. to avoid signal races we used our setAlaram method which block and unblock the signals,
encapsulating each atomic block of commands, preventing popping signals to
intervene while at it.

ANSWERS:
1.
RR:
pro - all the threads will run eventually without expecting any starvation,
because each thread has the same limit time to run.
In our case this pro is irrelevant because we choose the next thread according to his priority.
con - long threads will never finish in one run although it might be with the highest priority.
In out case this con is relevant because we set a same limit running time for each thread.
PQ:
pro - threads running according to priority and in case of starvation it will be a "smart" one,
the ones with the lowest priority will be the least ones to run.
In our case this pro is relevant because we choose our next thread according to his color priority.
con - there might be a starvation which mean a thread won't run for a long time.
In our case this con is relevant because as we said earlier we use color priority in choosing the next thread.

2. Example for a user-level event that will imply blocking a thread:
A game waiting for an input from the user, the game will be blocked untill the user will give an input (keyboard click
or a mouse movement for example).
```

2 Makefile

```
INCS=-I. -I/cs/course/current/os/lib/ -Luthreads
CFLAGS = -Wall -g -c $(INCS)
LFLAGS = -Wall -g $(INCS)
TARGETS = libuthreads.a

all: $(TARGETS)

Thread.o: Thread.cpp Thread.h
    g++ $(CFLAGS) Thread.cpp -o Thread.o

uthreads.o: uthreads.cpp Thread.h
    g++ $(CFLAGS) uthreads.cpp -o uthreads.o

libuthreads.a: Thread.o uthreads.o
    ar rc libuthreads.a  Thread.o uthreads.o
    ranlib libuthreads.a

clean:
    rm -r -f *.o libuthreads.a

.PHONY: clean all


```

3 Thread.h

```
1  /*
2   * Thread.h
3   *
4   *  Created on: Apr 16, 2015
5   *      Author: roeia1
6   */
7
8  #ifndef THREAD_H_
9  #define THREAD_H_
10
11 #include <valgrind/valgrind.h>
12 #include <valgrind/memcheck.h>
13 #include "uthreads.h"
14 #include <setjmp.h>
15 #include <signal.h>
16
17 enum State{READY, RUNNING, BLOCKED};
18
19 /**
20  * Inner class that represent Thread.
21  */
22 class Thread{
23
24 private:
25     unsigned int _id;
26     Priority _pr;
27     char* _stack;
28     State _state;
29     int _quantumCounter;
30     sigjmp_buf _env;
31
32 public:
33     /**
34      * constructor of Thread.
35      */
36     Thread(int id, Priority pr,void (*f)(void));
37
38     /**
39      * Distructor for thread,
40      */
41     ~Thread();
42
43     /**
44      * increment the quantom of a thread.
45      */
46     void incQuantom();
47
48     /**
49      * return the priority color of the thread.
50      */
51     Priority getPriority();
52
53     /**
54      * return the state of the tread.
55      */
56     State getState();
57
58     /**
59      * set the state of the tread.
```

```
60          */
61         void setState(State st);
62
63         /**
64          * return the quantomCounter of the thread.
65          */
66         int getQuantomCounter();
67
68         /**
69          * return the thread Id.
70          */
71         int getID();
72
73         /**
74          *
75          */
76         sigjmp_buf* getEnv();
77     };
78
79
80     #endif /* THREAD_H_ */
```

4 Thread.cpp

```
1  /*
2   * Thread.cpp
3   *
4   *  Created on: Apr 16, 2015
5   *      Author: roeia1
6   */
7
8  #include "Thread.h"
9  #include "uthreads.h"
10 #include <iostream>
11 using namespace std;
12 #ifdef __x86_64__
13 /* code for 64 bit Intel arch */
14
15 typedef unsigned long address_t;
16 #define JB_SP 6
17 #define JB_PC 7
18
19 /* A translation is required when using an address of a variable.
20  Use this as a black box in your code. */
21 address_t translate_address(address_t addr)
22 {
23     address_t ret;
24     asm volatile("xor    %%fs:0x30,%0\n"
25             "rol    $0x11,%0\n"
26             : "=g" (ret)
27               : "0" (addr));
28     return ret;
29 }
30
31 #else
32 /* code for 32 bit Intel arch */
33
34 typedef unsigned int address_t;
35 #define JB_SP 4
36 #define JB_PC 5
37
38 /* A translation is required when using an address of a variable.
39  Use this as a black box in your code. */
40 address_t translate_address(address_t addr)
41 {
42     address_t ret;
43     asm volatile("xor    %%gs:0x18,%0\n"
44             "rol    $0x9,%0\n"
45             : "=g" (ret)
46               : "0" (addr));
47     return ret;
48 }
49
50 #endif
51
52 /**
53  * constructor of Thread.
54  */
55 Thread::Thread(int id, Priority pr, void (*f)(void)) :
56                 _id(id), _pr(pr), _state(READY), _quantumCounter(0)
57 {
58     _stack = new char[STACK_SIZE];
59     address_t sp, pc;
```

```
60
61      sp = (address_t)_stack + STACK_SIZE - sizeof(address_t);
62      pc = (address_t) f;
63
64      sigsetjmp(_env, 1);
65      (_env->__jmpbuf)[JB_SP] = translate_address(sp);
66      (_env->__jmpbuf)[JB_PC] = translate_address(pc);
67      sigemptyset(&_env->__saved_mask);
68
69  }
70
71  /**
72   * Distructor for thread,
73   */
74  Thread::~Thread()
75  {
76      delete (_stack);
77  }
78
79  /**
80   * increment the quantom of a thread.
81   */
82  void Thread::incQuantom()
83  {
84      _quantumCounter++;
85  }
86
87  /**
88   * return the priority color of the thread.
89   */
90  Priority Thread::getPriority()
91  {
92      return _pr;
93  }
94
95  /**
96   * return the state of the tread.
97   */
98  State Thread::getState()
99  {
100     return _state;
101 }
102
103 /**
104  * set the state of the tread.
105  */
106 void Thread::setState(State st)
107 {
108     _state = st;
109 }
110
111 /**
112  * return the quantomCounter of the thread.
113  */
114 int Thread::getQuantomCounter()
115 {
116     return _quantumCounter;
117 }
118
119 /**
120  * return the thread Id.
121  */
122 int Thread::getID()
123 {
124     return _id;
125 }
126 /**
127  *
```

```
128     */
129        sigjmp_buf* Thread::getEnv()
130    {
131        return &_env;
132    }
```

5 uthreads.cpp

```cpp
/*
 * uthreads.cpp
 *
 *  Created on: Apr 12, 2015
 *      Author: roeia1
 */
#include <map>
#include <list>
#include <cstdlib>
#include <iostream>
#include <setjmp.h>
#include <signal.h>
#include <sys/time.h>
#include <queue>
#include <signal.h>
#include <unistd.h>
#include <stdexcept>
#include "uthreads.h"
#include "Thread.h"
using namespace std;

#define SEC 1000000
#define UNBLOCK_ALARM 0
#define BLOCK_ALARM 1
#define TERMINATE_SIG 32
#define BLOCK_SIG 33
#define THREAD_ERR "thread library error: "
#define SYS_ERR "system error: "
#define LIST_BAD_ALLOC "failed allocate thread list.\n"
#define THREAD_BAD_ALLOC "failed allocate thread.\n"
#define INVALID_ID "invalid thread's id.\n"
#define MASK_FAIL "failed to create mask.\n"

sigset_t blockedMasks;

//number of so far used quantum (of all threads)
int quantumCounter;

//quantum size in u-seconds
int myQuantomUsecs;

//the id of current running thread
int runningThread;

// the priority queues of ready threads.
list<int> redQ, orangeQ, greenQ;

// restor the blocked threads.
list<int> blockedList;

//restore all the available id to a new thread.
priority_queue<int,vector<int>,greater<int> > availbleIDq;

//the actual container of all threads, indexed by thread-id
map<int, Thread*> threadMap;



/*********************
```

```
 * forward declaration.
 *********************/
void switchThreads(int sig);
void setAlarm(int flag);
void deleteMap();
unsigned int nextThread(int id);
void resetTimer();
void dummyFunc(){cout<<"very bad"<<endl;}// use for the main entry.



/*
 * Initialize the thread library
 * Return error if the given quantum_usecs is invalid (should be non-negative).
 */
int uthread_init(int quantum_usecs)
{
    if (quantum_usecs <= 0)
    {
        cerr << "thread library error: quantum_usecs input invalid.\n";
        return -1;
    }
    myQuantomUsecs = quantum_usecs;
    quantumCounter = 1;
    runningThread = 0;
    //init the available id queue.
    for (int i = 1; i < MAX_THREAD_NUM; ++i)
    {
        availbleIDq.push(i);
    }
    signal(SIGVTALRM, switchThreads);
    if(sigemptyset(&blockedMasks) != 0)
    {
        cerr<<THREAD_ERR<<MASK_FAIL;
    }

    if(sigaddset(&blockedMasks, SIGVTALRM) != 0){
        cerr<<THREAD_ERR<<MASK_FAIL;
    }
    Thread* mainThread = new Thread(0,ORANGE,dummyFunc);
    mainThread->incQuantom();
    mainThread->setState(RUNNING);
    resetTimer();
    threadMap[0] = mainThread;
    return 0;
}


/*
 * Create a new thread whose entry point is f
 */
int uthread_spawn(void (*f)(void), Priority pr)
{
    if(f==NULL)
    {
        cerr<<THREAD_ERR<<"given function pointer is NULL\n";
        return -1;
    }

    if(availbleIDq.empty())
    {
        cerr<<THREAD_ERR<<"reached the maximal number of threads.\n";
        return -1;
    }
    setAlarm(BLOCK_ALARM);
    Thread* newThread = new Thread(availbleIDq.top(),pr,f);
    availbleIDq.pop();
    switch (pr) {
```

```
128         case RED:
129             try
130             {
131                 redQ.push_back(newThread->getID());
132             }
133             catch(bad_alloc &e)
134             {
135                 cerr<<THREAD_ERR<<LIST_BAD_ALLOC;
136                 setAlarm(UNBLOCK_ALARM);
137                 return -1;
138             }
139             break;
140         case ORANGE:
141             try
142             {
143                 orangeQ.push_back(newThread->getID());
144             }
145             catch(bad_alloc &e)
146             {
147                 cerr<<THREAD_ERR<<LIST_BAD_ALLOC;
148                 setAlarm(UNBLOCK_ALARM);
149                 return -1;
150             }
151             break;
152         case GREEN:
153             try
154             {
155                 greenQ.push_back(newThread->getID());
156             }
157             catch(bad_alloc &e)
158             {
159                 cerr<<THREAD_ERR<<LIST_BAD_ALLOC;
160                 setAlarm(UNBLOCK_ALARM);
161                 return -1;
162             }
163             break;
164         }
165         threadMap[newThread->getID()] = newThread;
166         setAlarm(UNBLOCK_ALARM);
167         return newThread->getID();
168     }
169 
170 
171     /*
172      * Terminate a thread
173      * Return error if the given id is invalid (not exist).
174      */
175     int uthread_terminate(int tid)
176     {
177 
178         setAlarm(BLOCK_ALARM);
179         if (tid == 0)
180         {
181             deleteMap();
182             exit(0);
183         }
184 
185         if(runningThread == tid)
186         {
187             switchThreads(TERMINATE_SIG);
188         }
189         else
190         {
191             // Check if the thread exists in map
192             try
193             {
194                 Thread* threadToDelete = threadMap.at(tid);
195                 if (threadToDelete->getState() == BLOCKED)
```

```
            {
                blockedList.remove(tid);
            }
            else
            {
                switch (threadToDelete->getPriority())
                {
                case RED:
                    redQ.remove(tid);
                    break;
                case ORANGE:
                    orangeQ.remove(tid);
                    break;
                case GREEN:
                    greenQ.remove(tid);
                    break;
                }
            }
            availbleIDq.push(tid);
            threadMap.erase(tid);
            delete threadToDelete;
        }
        catch(out_of_range &e)
        {
            cerr<<THREAD_ERR<<INVALID_ID;
            setAlarm(UNBLOCK_ALARM);
            return -1;
        }
    }
    setAlarm(UNBLOCK_ALARM);
    return 0;
}


/**
 *  Suspend a thread.
 *  Return error if the given id is invalid (not exist).
 */
int uthread_suspend(int tid)
{
    setAlarm(BLOCK_ALARM);
    if(tid == 0) // thread is main error - can't suspend the main.
    {
        cerr<<THREAD_ERR<<"can't suspend main thread.\n";
        setAlarm(UNBLOCK_ALARM);
        return -1;
    }

    if(tid != runningThread) // blocking not the running thread
    {
        try
        {
            Thread* threadToBlock = threadMap.at(tid);
            if(threadToBlock->getState() != BLOCKED)
            {
                switch (threadToBlock->getPriority()) {
                case RED:
                    redQ.remove(tid);
                    break;
                case ORANGE:
                    orangeQ.remove(tid);
                    break;
                case GREEN:
                    greenQ.remove(tid);
                    break;
                }
                blockedList.push_back(tid);
                threadToBlock->setState(BLOCKED);
```

```
            }
            setAlarm(UNBLOCK_ALARM);
        }
        catch(out_of_range &e)
        {
            cerr<<THREAD_ERR<<INVALID_ID;
            setAlarm(UNBLOCK_ALARM);
            return -1;
        }
    }
    else // blocking the running thread
    {
        setAlarm(UNBLOCK_ALARM);
        switchThreads(BLOCK_SIG);
    }
    setAlarm(UNBLOCK_ALARM);
    return 0;
}


/**
 *  Resume a thread that was suspended.
 *  If the the given thread is'nt blocked from before - ignore.
 *  Return error if the given id is invalid (not exist).
 */
int uthread_resume(int tid)
{
    setAlarm(BLOCK_ALARM);
    try
    {
        Thread* threadToResume = threadMap.at(tid);
        if(threadToResume->getState() == BLOCKED)
        {
            threadToResume->setState(READY);
            blockedList.remove(tid);
            switch (threadToResume->getPriority()) {
            case RED:
                redQ.push_back(tid);
                break;
            case ORANGE:
                orangeQ.push_back(tid);
                break;
            case GREEN:
                greenQ.push_back(tid);
                break;
            }
        }
    }
    catch(out_of_range &e)
    {
        cerr<<THREAD_ERR<<INVALID_ID;
        setAlarm(UNBLOCK_ALARM);
        return -1;
    }
    setAlarm(UNBLOCK_ALARM);
    return 0;
}


/*
 * Get the id of the calling thread
 */
int uthread_get_tid()
{
    return runningThread;
}

/*
```

```
 * Get the total number of library quantums.
 */
int uthread_get_total_quantums()
{
    return quantumCounter;
}


/*
 *  Get the number of thread quantums.
 *  Return error if the given id is invalid (not exist).
 */
int uthread_get_quantums(int tid)
{
    int threadQuantom;
    try
    {
        Thread* t = threadMap.at(tid);
        threadQuantom = t->getQuantomCounter();
    }
    catch(out_of_range &e)
    {
        cerr<<THREAD_ERR<<INVALID_ID;
        return -1;
    }
    return threadQuantom;
}


/**
 * this function switch between two threads according to the transition label (Preempt, Terminate, Blocked).
 */
void switchThreads(int sig)
{
    setAlarm(BLOCK_ALARM);
    int prevThread = runningThread;
    runningThread = nextThread(runningThread);

    //only the main thread is available. so we continue running with it without switching.
    if(runningThread == prevThread)
    {
        Thread* runt = threadMap.at(runningThread);
        quantumCounter++;
        runt->setState(RUNNING);
        runt->incQuantom();
        setAlarm(UNBLOCK_ALARM);
        return;
    }

    Thread* prevt = threadMap.at(prevThread);
    Thread* runt = threadMap.at(runningThread);
    int ret_val = sigsetjmp(*prevt->getEnv(),1);
    if (ret_val == 1) {
        setAlarm(UNBLOCK_ALARM);
        return;
    }
    quantumCounter++;
    runt->setState(RUNNING);
    runt->incQuantom();

    //Switch the threads according to the given signal case.
    switch(sig)
    {

    case BLOCK_SIG:
        blockedList.push_back(prevThread);
        prevt->setState(BLOCKED);
        break;
```

```
400
401         case TERMINATE_SIG:
402             threadMap.erase(prevThread);
403             delete prevt;
404             availbleIDq.push(prevThread);
405             break;
406
407         case SIGVTALRM:
408             switch (prevt->getPriority())
409             {
410             case RED:
411                 redQ.push_back(prevThread);
412                 break;
413             case ORANGE:
414                 orangeQ.push_back(prevThread);
415                 break;
416             case GREEN:
417                 greenQ.push_back(prevThread);
418                 break;
419             }
420             prevt->setState(READY);
421             break;
422         }
423
424         if (sig != SIGVTALRM)
425         {
426             resetTimer();
427         }
428         siglongjmp(*runt->getEnv(),1);
429     }
430
431
432     /**
433      * Return the next thread id to run- according to the priority of the threads.
434      */
435     unsigned int nextThread(int id)
436     {
437         if(!redQ.empty())
438         {
439             id = redQ.front();
440             redQ.pop_front();
441             return id;
442         }
443         if (!orangeQ.empty())
444         {
445             id = orangeQ.front();
446             orangeQ.pop_front();
447             return id;
448         }
449         if(!greenQ.empty())
450         {
451             id = greenQ.front();
452             greenQ.pop_front();
453         }
454         return id;
455     }
456
457
458     /*
459      * Add or remove clock alarm signal to blocked signals mask
460      */
461     void setAlarm(int flag)
462     {
463         switch (flag) {
464         case UNBLOCK_ALARM:
465
466             if(sigprocmask(SIG_UNBLOCK, &blockedMasks, NULL)!=0){
467                 cerr<<SYS_ERR<<"unblocking signal failed.\n";
```

```
468             }
469             sigset_t sigset;
470             sigpending(&sigset);
471             int x, res;
472             res = sigismember(&sigset, SIGVTALRM);
473             if(res)
474             {
475                 sigwait(&blockedMasks, &x);
476             }
477             break;
478         case BLOCK_ALARM:
479             if(sigprocmask(SIG_BLOCK, &blockedMasks, NULL)!=0){
480                 cerr<<SYS_ERR<<"blocking signal failed.\n";
481             }
482             break;
483         default:
484             break;
485         }
486     }
487 
488 
489     /*
490      * clear all heap allocated memory
491      */
492     void deleteMap()
493     {
494         map<int, Thread*>::iterator it;
495         it = threadMap.begin();
496         while(it!=threadMap.end()){
497             delete (it->second);
498             it++;
499         }
500     }
501 
502     /**
503      * Reset the time structure.
504      */
505     void resetTimer()
506     {
507         struct itimerval tv;
508         tv.it_value.tv_sec = myQuantomUsecs/SEC;
509         tv.it_value.tv_usec = myQuantomUsecs%SEC;
510         tv.it_interval.tv_sec = myQuantomUsecs/SEC;
511         tv.it_interval.tv_usec = myQuantomUsecs%SEC;
512         if(setitimer(ITIMER_VIRTUAL, &tv, NULL) != 0)
513         {
514             cerr<<SYS_ERR<<"set timer failed\n";
515         }
516     }
```