Contents

1 README

```
akiva_s,roigreenberg
Akiva Sygal(305277220), Roi Greenberg(305571234)
Ex: 2

FILES:
uthreads.cpp -- a file that implements the given uthreads.h
makefile

REMARKS:

data structures we used: a struct for thread, consisting of most of the PCB
componets as we saw in class. for the priorities queue we used a struct
consisted of 3 deques (because a simple c++ queue cannot be iterated, and we
wanted to have an acces not only to the item in the front, for deleting etc)
for each priority color. in addition we made an array consisting of all the
PCB of the threads that are not terminated, so the queus of threads would
contain only their indexes adn not the PCBs itselves.

we used a "switch" function with the principal shown in class, with the
addition of signals handeling to ensure that while the "switch" is running
the clock cannot ask for another "switch" just after it. we reset the clock
before we jump (=sigsetjmp), since we can also jump to "uthread_spawn" in the
first time a thread is being switched to. a terminated thread is identified
by changing its id to -1.



ANSWERS:

question 1:
RR- a pro is that no starvation can happen. a con is that if something needs
an immediate response by some thread we can't push it up and should wait to
his turn. PQ - a pro is that we can make a fundamental process to run more
than others. con - starvation can happen if there are long-running priorated
procesess. in our implementation starvation can still happen therotically,
but we can priorate importante threads to run more than the less important
ones.

question 2:
a user can actively pause a movie and though block a media process, or ask
for some infromation that has to be searched in the disk and block process
meanwhile.
```

2 Makefile

```
CC=g++
RANLIB=ranlib

LIBSRC=uthreads.cpp
LIBOBJ=$(LIBSRC:.cpp=.o)


INCS=-I.
CFLAGS = -Wall -g -std=c++11 $(INCS)
CPPFLAGS = -Wall -g -std=c++11
LOADLIBES = -L./

COMPILE.cc=$(CC) $(CPPFLAGS) -c

OSMLIB = libuthreads.a
TARGETS = $(OSMLIB)

TAR=tar
TARFLAGS=-cvf
TARNAME=ex2.tar
TARSRCS=$(LIBSRC) uthreads.h Makefile README

all: $(TARGETS)

$(LIBOBJ): $(LIBSRC) $(LIBSRC:.cpp=.h)
    $(COMPILE.cc) $% $<

$(TARGETS): $(LIBOBJ)
    $(AR) $(ARFLAGS) $@ $^
    $(RANLIB) $@

clean:
    $(RM) $(TARGETS) $(OSMLIB) $(OBJ) $(LIBOBJ) *~ *core

depend:
    makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)

tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)


```

3 uthreads.h

```
1  #ifndef _UTHREADS_H
2  #define _UTHREADS_H
3
4  /*
5   * User-Level Threads Library (uthreads)
6   * Author: OS, huji.os.2015@gmail.com
7   */
8
9  #define MAX_THREAD_NUM 100 /* maximal number of threads */
10 #define STACK_SIZE 4096 /* stack size per thread (in bytes) */
11
12 /* External interface */
13 typedef enum Priority { RED, ORANGE, GREEN } Priority;
14
15 /* Initialize the thread library */
16 int uthread_init(int quantum_usecs);
17
18 /* Create a new thread whose entry point is f */
19 int uthread_spawn(void (*f)(void), Priority pr);
20
21 /* Terminate a thread */
22 int uthread_terminate(int tid);
23
24 /* Suspend a thread */
25 int uthread_suspend(int tid);
26
27 /* Resume a thread */
28 int uthread_resume(int tid);
29
30
31 /* Get the id of the calling thread */
32 int uthread_get_tid();
33
34 /* Get the total number of library quantums */
35 int uthread_get_total_quantums();
36
37 /* Get the number of thread quantums */
38 int uthread_get_quantums(int tid);
39
40 #endif
```

4 uthreads.cpp

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

```
60  #endif
61
62
63  //macro for all the system calls that return -1 if failed.
64  #define SYS_ERROR(syscall, text) if ((syscall) == -1) \
65      {cerr << "system error: " << text; exit(1); }
66
67  //macro for all the library functions that return -1 if failed.
68  #define LBR_ERROR(text) cerr << "thread library error: " << text; return -1;
69
70  typedef enum State {running, ready, blocked} State;
71
72  /*
73   * struct for thread
74   */
75  typedef struct
76  {
77      int tid = -1;
78      void* ptr_stack;
79      sigjmp_buf env;
80      Priority pr;
81      int quantums;
82      State state;
83  public:
84      void thread_init(int tid, void* ptr_stack, \
85              Priority pr, State state=ready);
86  }Thread;
87
88  /*
89   * initiates thread's id, stack,quantums, priority and state
90   */
91  void Thread::thread_init(int tid, void* ptr_stack, Priority pr, State state)
92  {
93          this->tid = tid;
94          this->ptr_stack = ptr_stack;
95          this->pr = pr;
96          this->quantums = 0;
97          this->state = state;
98  }
99
100 int totalQuantums = 1; // counter for total quantom
101
102 //the time interval
103 struct itimerval tv;
104 int quantomsSec;
105 int quantomsUsec;
106
107 Thread threads[MAX_THREAD_NUM]; //array for all the threads
108 deque<int> priorities[3]; //the priorities queues
109 list<int> suspends;//list of blocked threads
110 int current = 0; //the current running thread, start with the 0 thread
111 sigset_t signal_set; // signals for blocking
112
113
114 /*
115  * set the timer with the time interval
116  * first it sets the timer to 0 then to the given time.
117  */
118 void set_timer()
119 {
120     //use to reset the timer
121     tv.it_value.tv_sec = 0;  // first time interval, seconds part
122     tv.it_value.tv_usec = 0; //first time interval, microseconds part
123     tv.it_interval.tv_sec = 0; // following time intervals, seconds part
124     tv.it_interval.tv_usec = 0; // following time intervals, microseconds part
125
126     SYS_ERROR(setitimer(ITIMER_VIRTUAL, &tv, NULL), "setitimer failed\n");
127
```

```
128     tv.it_value.tv_sec = quantomsSec;
129     tv.it_value.tv_usec = quantomsUsec;
130     tv.it_interval.tv_sec = quantomsSec;
131     tv.it_interval.tv_usec = quantomsUsec;
132
133     SYS_ERROR(setitimer(ITIMER_VIRTUAL, &tv, NULL), "setitimer failed\n");
134 }
135
136
137
138 /*
139  * return the first unused id or -1 if reaches to maximum threads
140  */
141 int find_first_free_id(Thread threads[])
142 {
143     for (int i = 1; i < MAX_THREAD_NUM; i++)
144     {
145         if (threads[i].tid == -1)
146         {
147             return i;
148         }
149     }
150     return -1;
151 }
152
153 /*
154  * return the next-to-run thread
155  * (if only main is left, returns 0)
156  */
157 int next_thread()
158 {
159     int next = 0;
160     for (int i = 0; i < 3; i++)
161     {
162         if (!priorities[i].empty())
163         {
164             next = priorities[i].front();
165             priorities[i].pop_front();
166             break;
167         }
168
169     }
170     return next;
171 }
172
173 /*
174  * push the thread to the ready queue
175  */
176 void push_to_ready (int tid)
177 {
178     threads[tid].state = ready;
179     priorities[threads[tid].pr].push_back(tid);
180 }
181
182
183 /*
184  * removes the thread from the ready queue
185  */
186 void remove_from_ready(int tid)
187 {
188     SYS_ERROR(sigprocmask(SIG_BLOCK, &signal_set, NULL), \
189                 "blocking signal failed\n");
190
191     deque<int> &current_deque = priorities[threads[tid].pr];
192
193     for (deque<int>::iterator it = current_deque.begin(); \
194         it != current_deque.end(); ++it)
195     {
```

```
196         if (*it == tid)
197             {
198                 current_deque.erase(it);
199                 break;
200             }
201     }
202 
203     SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
204                 "unblocking signal failed\n");
205 }
206 
207 /*
208  * switches between threads
209  */
210 void switchThreads(int sig)
211 {
212     totalQuantums++;
213 
214     int ret_val = sigsetjmp(threads[current].env,1);
215 
216     if (ret_val == 1)
217     {
218         return;
219     }
220 
221     int prev = current;
222     current = next_thread();
223     threads[current].state = running;
224 
225     //push the previous thread to ready queue if it's not blocked
226     // and not the only active thread
227     if (threads[prev].state != blocked && prev != current)
228     {
229         push_to_ready(prev);
230     }
231 
232     //removing signal from pending
233     sigset_t pending;
234     SYS_ERROR(sigemptyset(&pending), "sigemptyset failed\n");
235     SYS_ERROR(sigpending(&pending), "sigpending failed\n");
236     int waitSig = 1;//only to be used as parameter to syscalls. no real meaning
237 
238     int sigMember = sigismember(&pending, SIGVTALRM);
239 
240     SYS_ERROR(sigMember, "sigismember failed\n");
241 
242     if (sigMember == 1)
243     {
244         if(sigwait(&pending, &waitSig) > 0)
245             {
246                 cerr << "system error: sigwait failed\n";
247                 exit(1);
248             }
249     }
250 
251 
252     threads[current].quantums++;
253 
254     set_timer();
255 
256     siglongjmp(threads[current].env,1);
257 
258 }
259 
260 
261 
262 /* Initialize the thread library */
263 int uthread_init(int quantum_usecs)
```

```
264 {
265
266     if (quantum_usecs <= 0)
267     {
268         LBR_ERROR( "non-positive quantum usecs\n");
269     }
270
271     if(signal(SIGVTALRM, switchThreads) == SIG_ERR)
272     {
273         cerr << "system error: signal syscall failed\n";
274         exit(1);
275     }
276
277     SYS_ERROR(sigemptyset(&signal_set), "sigemptyset failed\n");
278     SYS_ERROR(sigaddset(&signal_set, SIGVTALRM), "sigaddset failed\n");
279
280     //divides the given time to seconds and microsecs.
281     quantomsSec = quantum_usecs / 1000000;
282     quantomsUsec = quantum_usecs % 1000000;
283
284     //"spawn" the 0 thread(the main)
285     threads[0].thread_init(0, NULL, ORANGE, running);
286     threads[0].quantums = 1;
287
288     set_timer();
289
290     sigsetjmp(threads[0].env, 1);
291
292     return 0;
293 }
294
295 /* Create a new thread whose entry point is f */
296 int uthread_spawn(void (*f)(void), Priority pr)
297 {
298     SYS_ERROR(sigprocmask(SIG_BLOCK, &signal_set, NULL), \
299                 "blocking signal failed\n");
300
301     address_t sp, pc;
302     int tid;
303
304     if((tid = find_first_free_id(threads)) == -1)
305     {
306         LBR_ERROR( "maximum threads\n");
307     }
308
309
310     void* ptr_stack = malloc(STACK_SIZE);
311     if (ptr_stack == NULL)
312     {
313         cerr << "system error: malloc failed\n";
314         exit(1);
315     }
316
317     sp = (address_t)ptr_stack + STACK_SIZE - sizeof(address_t);
318     pc = (address_t)f;
319
320     threads[tid].thread_init(tid, ptr_stack, pr);
321     threads[tid].state=ready;
322     push_to_ready(threads[tid].tid);
323
324     sigsetjmp(threads[tid].env, 1);
325     (threads[tid].env->__jmpbuf)[JB_SP] = translate_address(sp);
326     (threads[tid].env->__jmpbuf)[JB_PC] = translate_address(pc);
327
328     SYS_ERROR(sigemptyset(&threads[tid].env->__saved_mask),\
329                           "sigemptyset failed\n");
330
331
```

```
332     SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
333                 "unblocking signal failed\n");
334 
335     return tid;
336 }
337 
338 /* Terminate a thread */
339 int uthread_terminate(int tid)
340 {
341     SYS_ERROR(sigprocmask(SIG_BLOCK, &signal_set, NULL), \
342                 "blocking signal failed\n");
343 
344     if ((tid < 0) || (threads[tid].tid < 0))
345     {
346         SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
347                         "unblocking signal failed\n");
348         LBR_ERROR("no such thread\n");
349     }
350 
351     if (tid)
352     {
353         free(threads[tid].ptr_stack);
354         threads[tid].tid = -1;
355         threads[tid].quantums = 0;
356         if (threads[tid].state == running)
357         {
358             threads[tid].state = blocked; //use to prevent switch to push it back to queue
359             switchThreads(0);
360 
361             SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
362                             "unblocking signal failed\n");
363 
364             return 0;
365         }
366         else if (threads[tid].state == ready)
367         {
368             remove_from_ready(tid);
369         }
370         else
371         {
372             suspends.remove(tid);
373         }
374     }
375     else
376     {
377         for (int i = 1; i < MAX_THREAD_NUM; i++)
378         {
379             if (threads[i].tid != -1)
380             {
381                 free(threads[i].ptr_stack);
382                 threads[i].tid = -1;
383             }
384         }
385         exit(0);
386     }
387 
388     SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
389                 "unblocking signal failed\n");
390 
391     return 0;
392 }
393 
394 /* Suspend a thread */
395 int uthread_suspend(int tid)
396 {
397     SYS_ERROR(sigprocmask(SIG_BLOCK, &signal_set, NULL), \
398                 "blocking signal failed\n");
399 
```

```
400     if ((tid <= 0) || (threads[tid].tid < 0))
401     {
402         SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
403                         "unblocking signal failed\n");
404         if (tid == 0)
405         {
406             LBR_ERROR("cannot suspend main thread\n");
407         }
408         else
409         {
410             LBR_ERROR("no such thread\n");
411         }
412     }
413 
414     if (threads[tid].state != blocked)
415     {
416         suspends.push_front(tid);
417         if (threads[tid].state == ready)
418         {
419             remove_from_ready(tid);
420             threads[tid].state = blocked;
421         }
422         else
423         {
424             threads[tid].state = blocked;
425             switchThreads(0);
426         }
427     }
428 
429     SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
430                 "unblocking signal failed\n");
431 
432     return 0;
433 }
434 
435 /* Resume a thread */
436 int uthread_resume(int tid)
437 {
438     SYS_ERROR(sigprocmask(SIG_BLOCK, &signal_set, NULL), \
439                 "blocking signal failed\n");
440 
441     if ((tid < 0) || (threads[tid].tid < 0))
442     {
443         SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
444                         "unblocking signal failed\n");
445         LBR_ERROR("no such thread\n");
446 
447     }
448 
449     if (threads[tid].state == blocked)
450     {
451         suspends.remove(tid);
452         threads[tid].state = ready;
453         push_to_ready(tid);
454     }
455 
456     SYS_ERROR(sigprocmask(SIG_UNBLOCK, &signal_set, NULL), \
457                 "unblocking signal failed\n");
458 
459     return 0;
460 }
461 
462 
463 /* Get the id of the calling thread */
464 int uthread_get_tid()
465 {
466     return current;
467 }
```

```
468
469 /* Get the total number of library quantums */
470 int uthread_get_total_quantums()
471 {
472     return totalQuantums;
473 }
474
475 /* Get the number of thread quantums */
476 int uthread_get_quantums(int tid)
477 {
478     if ((tid < 0) || (threads[tid].tid < 0))
479         {
480             LBR_ERROR( "no such thread\n");
481         }
482     return threads[tid].quantums;
483 }
```