Contents

1 Basic Test Results

```
1  ============== Tar Content Test ==============
2  found README
3  found Makefile
4  tar content test PASSED!
5
6  ============== logins ==============
7  login names mentioned in file:  tomka,alonemanuel
8  Please make sure that these are the correct login names.
9
10 ============== make Command Test ==============
11 g++ -Wall -std=c++11 -g -I.   -c -o uthreads.o uthreads.cpp
12 g++ -Wall -std=c++11 -g -I.   -c -o thread.o thread.cpp
13 g++ -Wall -std=c++11 -g -I.   -c -o sleeping_threads_list.o sleeping_threads_list.cpp
14 ar rv libuthreads.a uthreads.o thread.h thread.o sleeping_threads_list.h sleeping_threads_list.o
15 a - uthreads.o
16 a - thread.h
17 a - thread.o
18 a - sleeping_threads_list.h
19 a - sleeping_threads_list.o
20 ranlib libuthreads.a
21
22 ar: creating libuthreads.a
23
24 make command test PASSED!
25
26 ============== Linking Test ==============
27
28
29 Linking PASSED!
30
31 Pre-submission passed!
32 Keep in mind that this script tests only basic elements of your code.
```

2 README

```
tomka, alonemanuel
Tom Kalir (316426485), Alon Emanuel (205894058)
EX: 2

FILES:
sleeping_threads_list.h -- a file with some code
sleeping_threads_list.cpp -- a file with some code
thread.h -- a file with some code
thread.cpp -- a file with some code
uthreads.cpp -- a file with some code

REMARKS:

ANSWERS:

Question 1:


Question 2:

Advantages:
1. Memory and resource management is made easier and more beneficial -
when a tab is closed (or minimized\hidden), its resources can
be freed or shared among other currently-active processes.
2. Bugs and crashes are focused to the specific tab (=process), thus giving
the browser a more stable performance that can 'survive' tab-specific errors.

Disadvantages:
1. As opposed to user-level threads, each process creates a lot of overhead
and is considered to be slow and inefficient.
2. Although resource management is indeed efficient in chrome's implementation,
but resource *use* and demand is greater, due to the need to store data for each of these
processes (which are not 'hidden' from the OS like user-level threads are).


Question 3.a:

Question 3.b:

Question 3.c:

Question 4:
As described in "man itimer":
Real time 'passes' as real-life time,
while virtual time passes only when the process is executing.
We use *virtual* time in our quantum timer - it sends a signal when a virtual quantum has passed.
We use *real* time in our sleep timer - it send a signal when a real time 'unit' has passed.
```

3 Makefile

```
CC=g++
CXX=g++
RANLIB=ranlib

LIBSRC=uthreads.cpp thread.h thread.cpp sleeping_threads_list.h sleeping_threads_list.cpp
LIBOBJ=$(LIBSRC:.cpp=.o)

INCS=-I.
CFLAGS = -Wall -std=c++11 -g $(INCS)
CXXFLAGS = -Wall -std=c++11 -g $(INCS)

OSMLIB = libuthreads.a
TARGETS = $(OSMLIB)

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

all: $(TARGETS)

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

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

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

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

4 sleeping threads list.h

```
1  #ifndef SLEEPING_THREADS_LIST_H
2  #define SLEEPING_THREADS_LIST_H
3
4  #include <deque>
5  #include <sys/time.h>
6
7  using namespace std;
8
9  struct wake_up_info
10 {
11     int id;
12     timeval awaken_tv;
13 };
14
15 class SleepingThreadsList
16 {
17
18     deque <wake_up_info> sleeping_threads;
19
20 public:
21
22     SleepingThreadsList();
23
24     /*
25      * Description: This method adds a new element to the list of sleeping
26      * threads. It gets the thread's id, and the time when it needs to wake up.
27      * The wakeup_tv is a struct timeval (as specified in <sys/time.h>) which
28      * contains the number of seconds and microseconds since the Epoch.
29      * The method keeps the list sorted by the threads' wake up time.
30     */
31     void add(int thread_id, timeval timestamp);
32
33     void remove(int tid);
34
35     /*
36      * Description: This method removes the thread at the top of this list.
37      * If the list is empty, it does nothing.
38     */
39     void pop();
40
41     /*
42      * Description: This method returns the information about the thread (id and time it needs to wake up)
43      * at the top of this list without removing it from the list.
44      * If the list is empty, it returns null.
45     */
46     wake_up_info *peek();
47
48 };
49
50 #endif
```

5 sleeping threads list.cpp

```cpp
#include "sleeping_threads_list.h"

SleepingThreadsList::SleepingThreadsList() {
}


/*
 * Description: This method adds a new element to the list of sleeping
 * threads. It gets the thread's id, and the time when it needs to wake up.
 * The wakeup_tv is a struct timeval (as specified in <sys/time.h>) which
 * contains the number of seconds and microseconds since the Epoch.
 * The method keeps the list sorted by the threads' wake up time.
*/
void SleepingThreadsList::add(int thread_id, timeval wakeup_tv) {

    wake_up_info new_thread;
    new_thread.id = thread_id;
    new_thread.awaken_tv = wakeup_tv;

    if(sleeping_threads.empty()){
        sleeping_threads.push_front(new_thread);
    }
    else {
        for (deque<wake_up_info>::iterator it = sleeping_threads.begin(); it != sleeping_threads.end(); ++it){
            if(timercmp(&it->awaken_tv, &wakeup_tv, >=)){
                sleeping_threads.insert(it, new_thread);
                return;
            }
        }
        sleeping_threads.push_back(new_thread);
    }
}

void SleepingThreadsList::remove(int tid){
    for (deque<wake_up_info>::iterator it = sleeping_threads.begin(); it != sleeping_threads.end(); ++it){
        if(it->id == tid){
            sleeping_threads.erase(it);
            return;
        }
    }
}

/*
 * Description: This method removes the thread at the top of this list.
 * If the list is empty, it does nothing.
*/
void SleepingThreadsList::pop() {
    if(!sleeping_threads.empty())
        sleeping_threads.pop_front();
}

/*
 * Description: This method returns the information about the thread (id and time it needs to wake up)
 * at the top of this list without removing it from the list.
 * If the list is empty, it returns null.
*/
wake_up_info* SleepingThreadsList::peek(){
    if (sleeping_threads.empty())
        return nullptr;
```

```
60          return &sleeping_threads.at(0);
61      }
62
```

6 thread.h

```
1  //
2  // Created by kalir on 27/03/2019.
3  //
4
5  #ifndef EX2_THREAD_H
6
7  #include "uthreads.h"
8  #include <stdio.h>
9  #include <setjmp.h>
10 #include <signal.h>
11 #include <unistd.h>
12 #include <sys/time.h>
13 #include <memory>
14
15 #define EX2_THREAD_H
16 #define READY 0
17 #define BLOCKED 1
18 #define RUNNING 2
19
20 typedef unsigned long address_t;
21 #define JB_SP 6
22 #define JB_PC 7
23
24
25 using std::shared_ptr;
26
27 class Thread
28 {
29
30 private:
31     static int num_of_threads;
32 protected:
33     int _id;
34     int _state;
35     int _stack_size;
36     int _quantums;
37
38 public:
39
40     void (*func)(void);
41
42     char *stack;
43     sigjmp_buf env[1];
44     address_t sp, pc;
45
46
47     Thread(void (*f)(void) = nullptr, int id=0);
48
49     int get_id() const
50     { return _id; };
51
52     int get_state()
53     { return _state; };
54
55     void set_state(int state)
56     { _state = state; };
57
58     int get_quantums()
59     {
```

```
60          return _quantums;
61      }
62
63      void increase_quantums()
64      {
65          _quantums++;
66      }
67
68      bool operator==(const Thread &other) const;
69  };
70
71  #endif //EX2_THREAD_H
```

7 thread.cpp

```cpp
//
// Created by kalir on 27/03/2019.
//

#include <iostream>
#include "uthreads.h"
#include "thread.h"
#include <memory>
#define STACK_SIZE 4096 /* stack size per thread (in bytes) */

using std::cout;
using std::endl;


int Thread::num_of_threads = 0;

/* A translation is required when using an address of a variable.
   Use this as a black box in your code. */
address_t translate_address(address_t addr)
{
    address_t ret;
    asm volatile("xor    %%fs:0x30,%0\n"
                 "rol    $0x11,%0\n"
    : "=g" (ret)
    : "0" (addr));
    return ret;
}

/**
 * @brief Constructor of a thread object
 * @param f thread function address
 */
Thread::Thread(void (*f)(void), int id) : _id(id), _state(READY), _stack_size(STACK_SIZE), _quantums(0), func(f)
{
    stack = new char[STACK_SIZE];
//    if (num_of_threads)
//    {
//    cout << "Creating thread!" << endl;
    sp = (address_t) stack + STACK_SIZE - sizeof(address_t);
    pc = (address_t) f;
    sigsetjmp(env[0], 1);
    (env[0]->__jmpbuf)[JB_SP] = translate_address(sp);
    (env[0]->__jmpbuf)[JB_PC] = translate_address(pc);
    sigemptyset(&env[0]->__saved_mask);
//    }
    num_of_threads++;
}


bool Thread::operator==(const Thread &other) const
{
    return _id == other.get_id();
}

```

8 uthreads.cpp

```cpp

#include "uthreads.h"
#include "thread.h"
#include <list>
#include <unordered_map>
#include <algorithm>
#include <iostream>
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <memory>
#include "sleeping_threads_list.h"

// Constants //
#define SYS_ERR_CODE 0
#define THREAD_ERR_CODE 1
#define MAX_THREAD_NUM 100 /* maximal number of threads */
#define STACK_SIZE 4096 /* stack size per thread (in bytes) */
#define TIMER_SET_MSG "setting the timer has failed."
#define INVALID_ID_MSG "thread ID must be between 0 and "+ to_string(MAX_THREAD_NUM) + "."
/* External interface */



#define ID_NONEXIST_MSG "thread with such ID does not exist."

#define BLOCK_MAIN_MSG "main thread cannot be blocked."

#define NEG_TIME_MSG "time must be non-negative."

#define MAX_THREAD_MSG "max number of threads exceeded."
// Using //

using std::cout;
using std::endl;

using std::shared_ptr;


// Static Variables //
int total_quantums;

sigjmp_buf env[2];

sigset_t sigs_to_block;

/**
 * @brief map of all existing threads, with their tid as key.
 */
std::unordered_map<int, shared_ptr<Thread>> threads;
/**
 * @brief list of all ready threads.
 */
std::list<shared_ptr<Thread>> ready_threads;
/**
 * @brief the current running thread.
 */
shared_ptr<Thread> running_thread;
/**
```

```
60   * @brief list of all current sleeping threads (id's).
61   */
62  SleepingThreadsList sleeping_threads;
63  /**
64   * @brief timers.
65   */
66
67  struct itimerval quantum_timer, sleep_timer;
68  /**
69   * @brief sigactions.
70   */
71  struct sigaction quantum_sa, sleep_sa;
72
73
74  // Helper Functions //
75
76
77  void block_signals()
78  {
79
80      sigprocmask(SIG_BLOCK, &sigs_to_block, NULL);
81
82  }
83
84  void unblock_signals()
85  {
86      sigprocmask(SIG_UNBLOCK, &sigs_to_block, NULL);
87
88  }
89
90  unsigned int get_min_id()
91  {
92      block_signals();
93
94      for (unsigned int i = 0; i < threads.size(); ++i)
95      {
96          if (threads.find(i) == threads.end())
97          {
98              unblock_signals();
99              return i;
100         }
101     }
102     unblock_signals();
103     return (unsigned int) threads.size();
104
105 }
106
107 /**
108  * @brief exiting due to error function
109  * @param code error code
110  * @param text explanatory text for the error
111  */
112 int print_err(int code, string text)
113 {
114     block_signals();
115     string prefix;
116     switch (code)
117     {
118         case SYS_ERR_CODE:
119             prefix = "system error: ";
120             break;
121         case THREAD_ERR_CODE:
122             prefix = "thread library error: ";
123             break;
124     }
125     cout << prefix << text << endl;    // TODO change to cout
126     if (code == SYS_ERR_CODE)
127     {
```

```
128             exit(1);    // TODO we need to return on failures, but exit makes it irrelevant
129         }
130         else
131         {
132             unblock_signals();
133             return -1;
134         }
135 
136     }
137 
138     void next_sleeping()
139     {
140         block_signals();
141         wake_up_info *last_sleeping = sleeping_threads.peek();
142         if (last_sleeping != nullptr)
143         {
144             // update sleep_timer values
145             sleep_timer.it_value.tv_sec = last_sleeping->awaken_tv.tv_sec;
146             sleep_timer.it_value.tv_usec = last_sleeping->awaken_tv.tv_usec;
147             if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
148             {
149                 print_err(SYS_ERR_CODE, TIMER_SET_MSG);
150             }
151         }
152         else
153         {
154             sleep_timer.it_value.tv_sec = 0;
155             sleep_timer.it_value.tv_usec = 0;
156         }
157         unblock_signals();
158     }
159 
160     void create_main_thread()
161     {
162         shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread());
163         threads[new_thread->get_id()] = new_thread;
164         running_thread = new_thread;
165         running_thread->increase_quantums();
166     }
167 
168     bool does_exist(std::list<shared_ptr<Thread>> lst, int tid)
169     {
170         block_signals();
171         for (std::list<shared_ptr<Thread>>::iterator it = lst.begin(); it != lst.end(); ++it)
172         {
173             if ((*it)->get_id() == tid)
174             {
175                 unblock_signals();
176                 return true;
177             }
178         }
179         unblock_signals();
180         return false;
181     }
182 
183     void init_sigs_to_block()
184     {
185         sigemptyset(&sigs_to_block);
186         sigaddset(&sigs_to_block, SIGALRM);
187         sigaddset(&sigs_to_block, SIGVTALRM);
188     }
189 
190 
191     timeval calc_wake_up_timeval(int usecs_to_sleep)
192     {
193         block_signals();
194         timeval now, time_to_sleep, wake_up_timeval;
195         gettimeofday(&now, nullptr);
```

```
196     time_to_sleep.tv_sec = usecs_to_sleep / 1000000;
197     time_to_sleep.tv_usec = usecs_to_sleep % 1000000;
198     timeradd(&now, &time_to_sleep, &wake_up_timeval);
199     unblock_signals();
200     return wake_up_timeval;
201 }
202 
203 /**
204  * @brief make the front of the ready threads list the current running thread.
205  */
206 void ready_to_running(bool is_blocking = false)
207 {
208     block_signals();
209     int ret_val = sigsetjmp(running_thread->env[0], 1);
210     if (ret_val == 1)
211     {
212         unblock_signals();
213         return;
214     }
215     if (!is_blocking)
216     {
217         // push the current running thread to the back of the ready threads
218         ready_threads.push_back(running_thread);
219     }
220     // pop the topmost ready thread to be the running thread
221     running_thread = ready_threads.front();
222     // increase thread's quantum counter
223     running_thread->increase_quantums();
224     total_quantums++;
225     ready_threads.pop_front();
226     // jump to the running thread's last state
227     if (setitimer(ITIMER_VIRTUAL, &quantum_timer, NULL))
228     {
229         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
230     }
231     unblock_signals();
232     siglongjmp(running_thread->env[0], 1);
233 }
234 
235 shared_ptr<Thread> get_ready_thread(int tid)
236 {
237     for (std::list<shared_ptr<Thread>>::iterator it = ready_threads.begin(); it != ready_threads.end(); ++it)
238     {
239         if ((*it)->get_id() == tid)
240         {
241             return *it;
242         }
243     }
244     return nullptr;
245 }
246 
247 
248 bool is_id_invalid(int tid)
249 {
250     return ((tid < 0) || (tid > MAX_THREAD_NUM));
251 
252 }
253 
254 bool is_id_nonexisting(int tid)
255 {
256     return threads.find(tid) == threads.end();
257 }
258 
259 bool is_main_thread(int tid)
260 {
261     return tid == 0;
262 }
263 
```

```
264 bool is_time_invalid(int time)
265 {
266     return time < 0;
267 }
268
269 bool is_running_thread(int tid)
270 {
271     return tid == running_thread->get_id();
272 }
273
274 // Handlers //
275 void quantum_handler(int sig)
276 {
277
278     block_signals();
279     ready_to_running();
280     unblock_signals();
281 }
282
283
284 void sleep_handler(int sig)
285 {
286     block_signals();
287     cout << "woke up" << endl;
288
289     uthread_resume(sleeping_threads.peek()->id);
290     sleeping_threads.pop();
291     next_sleeping();
292     unblock_signals();
293 }
294
295
296 void init_quantum_timer(int quantum_usecs)
297 {
298     quantum_timer.it_value.tv_sec = quantum_usecs / 1000000;
299     quantum_timer.it_value.tv_usec = quantum_usecs % 1000000;
300     quantum_sa.sa_handler = &quantum_handler;
301     if (sigaction(SIGVTALRM, &quantum_sa, NULL) < 0)
302     {
303         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
304     }
305 }
306
307 void init_sleep_timer()
308 {
309     sleep_timer.it_value.tv_sec = 0;
310     sleep_timer.it_value.tv_usec = 0;
311     sleep_timer.it_interval.tv_usec = 0;
312     sleep_timer.it_interval.tv_usec = 0;
313     sleep_sa.sa_handler = &sleep_handler;
314     if (sigaction(SIGALRM, &sleep_sa, NULL) < 0)
315     {
316         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
317     }
318
319 }
320
321
322 // API Functions //
323
324 /*
325  * Description: This function initializes the thread library.
326  * You may assume that this function is called before any other thread library
327  * function, and that it is called exactly once. The input to the function is
328  * the length of a quantum in micro-seconds. It is an error to call this
329  * function with non-positive quantum_usecs.
330  * Return value: On success, return 0. On failure, return -1.
331 */
```

```
332 int uthread_init(int quantum_usecs)
333 {
334     init_sigs_to_block();
335     block_signals();
336     // quantum_usecs cannot be negative
337     if (is_time_invalid(quantum_usecs))
338     {
339         return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
340     }
341     // 1 because of the main thread
342     total_quantums = 1;
343     // init timers
344     init_quantum_timer(quantum_usecs);
345     init_sleep_timer();
346     // set quantum timer
347     if (setitimer(ITIMER_VIRTUAL, &quantum_timer, NULL))
348     {
349         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
350     }
351     // create main thread
352     create_main_thread();
353     // init blocked signals set
354     return 0;
355 }
356
357
358 /*
359  * Description: This function creates a new thread, whose entry point is the
360  * function f with the signature void f(void). The thread is added to the end
361  * of the READY threads list. The uthread_spawn function should fail if it
362  * would cause the number of concurrent threads to exceed the limit
363  * (MAX_THREAD_NUM). Each thread should be allocated with a stack of size
364  * STACK_SIZE bytes.
365  * Return value: On success, return the ID of the created thread.
366  * On failure, return -1.
367 */
368 int uthread_spawn(void (*f)(void))
369 {
370     block_signals();
371     if (threads.size() == MAX_THREAD_NUM)
372     {
373         return (print_err(THREAD_ERR_CODE, MAX_THREAD_MSG));
374     }
375     // create new thread
376     shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread(f, get_min_id()));
377     threads[new_thread->get_id()] = new_thread;
378     ready_threads.push_back(new_thread);
379     unblock_signals();
380     return new_thread->get_id();
381 }
382
383
384 /*
385  * Description: This function terminates the thread with ID tid and deletes
386  * it from all relevant control structures. All the resources allocated by
387  * the library for this thread should be released. If no thread with ID tid
388  * exists it is considered an error. Terminating the main thread
389  * (tid == 0) will result in the termination of the entire process using
390  * exit(0) [after releasing the assigned library memory].
391  * Return value: The function returns 0 if the thread was successfully
392  * terminated and -1 otherwise. If a thread terminates itself or the main
393  * thread is terminated, the function does not return.
394 */
395 int uthread_terminate(int tid)
396 {
397     block_signals();
398     if (is_id_invalid(tid))
399     {
```

```
400         unblock_signals();
401         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
402     }
403     if (is_id_nonexisting(tid))
404     {
405         unblock_signals();
406         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
407     }
408     //TODO: consider an error and memory deallocation
409     if (is_main_thread(tid))
410     {
411         unblock_signals();
412         exit(0);
413     }
414     // terminate running thread
415     if (is_running_thread(tid))
416     {
417         threads.erase(tid);
418         ready_to_running(true);
419     }
420         // terminate non running thread
421     else
422     {
423         if (does_exist(ready_threads, tid))
424         {
425             ready_threads.remove(threads[tid]);
426         }
427         else
428         {
429             if (sleeping_threads.peek() != nullptr)
430             {
431                 int curr_sleeper_id = sleeping_threads.peek()->id;
432                 sleeping_threads.remove(tid);
433                 if (curr_sleeper_id == tid)
434                 {
435                     next_sleeping();
436                 }
437             }
438             threads.erase(tid);
439         }
440     }
441     unblock_signals();
442     return 0;
443 }
444
445
446 /*
447  * Description: This function blocks the thread with ID tid. The thread may
448  * be resumed later using uthread_resume. If no thread with ID tid exists it
449  * is considered as an error. In addition, it is an error to try blocking the
450  * main thread (tid == 0). If a thread blocks itself, a scheduling decision
451  * should be made. Blocking a thread in BLOCKED state has no
452  * effect and is not considered an error.
453  * Return value: On success, return 0. On failure, return -1.
454 */
455 int uthread_block(int tid)
456 {
457     block_signals();
458     if (is_id_invalid(tid))
459     {
460         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
461     }
462     if (is_id_nonexisting(tid))
463     {
464         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
465     }
466     if (is_main_thread(tid))
467     {
```

```
468             return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
469         }
470 
471         // if thread is the running thread, run the next ready thread
472         if (is_running_thread(tid))
473         {
474             unblock_signals();
475             ready_to_running(true);
476         }
477 
478         shared_ptr<Thread> to_delete = get_ready_thread(tid);
479         // block thread (remove from ready)
480         if (to_delete != nullptr)
481         {
482             ready_threads.remove(to_delete);
483         }
484         unblock_signals();
485         return 0;
486     }
487 
488 
489     /*
490      * Description: This function resumes a blocked thread with ID tid and moves
491      * it to the READY state. Resuming a thread in a RUNNING or READY state
492      * has no effect and is not considered as an error. If no thread with
493      * ID tid exists it is considered an error.
494      * Return value: On success, return 0. On failure, return -1.
495     */
496     int uthread_resume(int tid)
497     {
498         block_signals();
499         if (is_id_invalid(tid))
500         {
501             return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
502         }
503         if (is_id_nonexisting(tid))
504         {
505             return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
506         }
507         shared_ptr<Thread> curr_thread = threads[tid];
508         // if thread to resume is not running or already ready
509         if (!does_exist(ready_threads, tid) && !is_running_thread(tid))
510         {
511             ready_threads.push_back(curr_thread);
512         }
513         unblock_signals();
514         return 0;
515     }
516 
517 
518     /*
519      * Description: This function blocks the RUNNING thread for user specified micro-seconds (REAL
520      * time).
521      * It is considered an error if the main thread (tid==0) calls this function.
522      * Immediately after the RUNNING thread transitions to the BLOCKED state a scheduling decision
523      * should be made.
524      * Return value: On success, return 0. On failure, return -1.
525     */
526     int uthread_sleep(unsigned int usec)
527     {
528         block_signals();
529         if (is_time_invalid(usec))
530         {
531             unblock_signals();
532             return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
533         }
534         if (is_main_thread(running_thread->get_id()))
535         {
```

```
536         unblock_signals();
537         return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
538     }
539     if (usec == 0)
540     {
541         ready_to_running();
542         unblock_signals();
543         return 0;
544     }
545     if (sleeping_threads.peek() == nullptr)
546     {
547 
548         // update sleep_timer values
549         sleep_timer.it_value.tv_sec = usec / 1000000;
550         sleep_timer.it_value.tv_usec = usec % 1000000;
551         if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
552         {
553             print_err(SYS_ERR_CODE, TIMER_SET_MSG);
554         }
555     }
556     sleeping_threads.add(running_thread->get_id(), calc_wake_up_timeval(usec));
557     ready_to_running(true);
558     unblock_signals();
559     return 0;
560 }
561 
562 
563 /*
564  * Description: This function returns the thread ID of the calling thread.
565  * Return value: The ID of the calling thread.
566 */
567 int uthread_get_tid()
568 {
569     return running_thread->get_id();
570 }
571 
572 /*
573  * Description: This function returns the total number of quantums since
574  * the library was initialized, including the current quantum.
575  * Right after the call to uthread_init, the value should be 1.
576  * Each time a new quantum starts, regardless of the reason, this number
577  * should be increased by 1.
578  * Return value: The total number of quantums.
579 */
580 int uthread_get_total_quantums()
581 {
582     return total_quantums;
583 }
584 
585 /*
586  * Description: This function returns the number of quantums the thread with
587  * ID tid was in RUNNING state. On the first time a thread runs, the function
588  * should return 1. Every additional quantum that the thread starts should
589  * increase this value by 1 (so if the thread with ID tid is in RUNNING state
590  * when this function is called, include also the current quantum). If no
591  * thread with ID tid exists it is considered an error.
592  * Return value: On success, return the number of quantums of the thread with ID tid.
593  *                  On failure, return -1.
594 */
595 int uthread_get_quantums(int tid)
596 {
597     if (is_id_invalid(tid))
598     {
599         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
600     }
601     if (is_id_nonexisting(tid))
602     {
603         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
```

```
604         }
605         return threads[tid]->get_quantums();
606     }
```