Contents

1 Basic Test Results

```
============== Tar Content Test ==============
found README
found Makefile
tar content test PASSED!

============== logins ==============
login names mentioned in file:  tomka,alonemanuel
Please make sure that these are the correct login names.

============== make Command Test ==============
g++ -Wall -std=c++11 -g -I.   -c -o uthreads.o uthreads.cpp
g++ -Wall -std=c++11 -g -I.   -c -o thread.o thread.cpp
g++ -Wall -std=c++11 -g -I.   -c -o sleeping_threads_list.o sleeping_threads_list.cpp
ar rv libuthreads.a uthreads.o thread.h thread.o sleeping_threads_list.h sleeping_threads_list.o
a - uthreads.o
a - thread.h
a - thread.o
a - sleeping_threads_list.h
a - sleeping_threads_list.o
ranlib libuthreads.a

uthreads.cpp:31: warning: "TIMER_SET_MSG" redefined
 #define TIMER_SET_MSG "setting the timer failed."

uthreads.cpp:19: note: this is the location of the previous definition
 #define TIMER_SET_MSG "setting the timer has failed."

uthreads.cpp:36:16: warning: comma-separated list in using-declaration only available with -std=c++17 or -std=gnu++17
 using std::cout,
                ^
uthreads.cpp: In function  int get_min_id() :
uthreads.cpp:95:20: warning: comparison of integer expressions of different signedness:  int  and  std::unordered_map<int,
  for (int i = 0; i < threads.size(); ++i)
                  ~~^~~~~~~~~~~~~~~~
uthreads.cpp: In function  int uthread_terminate(int) :
uthreads.cpp:419:1: warning: control reaches end of non-void function [-Wreturn-type]
 }
 ^
In file included from thread.cpp:7:
thread.h: In constructor  Thread::Thread(void (*)(), int) :
thread.h:35:6: warning:  Thread::_id  will be initialized after [-Wreorder]
  int _id;
      ^~
thread.h:33:6: warning:    int Thread::_state  [-Wreorder]
  int _state;
      ^~~~~~
thread.cpp:33:1: warning:  when initialized here [-Wreorder]
 Thread::Thread(void (*f)(void), int id) : _id(id), _state(READY), _stack_size(STACK_SIZE), _quantums(0), func(f)
 ^~~~~~
ar: creating libuthreads.a

stderr: b'uthreads.cpp:31: warning: "TIMER_SET_MSG" redefined\n #define TIMER_SET_MSG "setting the timer failed."\n \nuthrea
make command test FAILED!

============== Linking Test ==============


Linking PASSED!

```

```
60  Pre-submission Test FAILED!
61  Check info above.
```

2 README

```
tomka, alonemanuel
Tom Kalir (316426485), Alon Emanuel (205894058)
EX: 1

FILES:
myfile.c -- a file with some code
myfile.h -- a file with some headers

REMARKS:

ANSWERS:

Assignment 1:

After running the file there are two options -

1. If you supply a single argument, the program creates a directory called 'Welcome',
and inside 'Welcome' it creates another directory called 'To', and inside of that it creates a file
called 'OS2018'.
It then opens the file and writes the following text into it:
"[username]
If you haven't read the course guidelines yet --- do it right now!
[supplied argument]"
It then closes the file and deletes it, and then goes on to delete the directories in a reverse order.

2. Else, it prints the following error:
"Error. The program should receive a single argument. Exiting."
```

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

```cpp
#ifndef SLEEPING_THREADS_LIST_H
#define SLEEPING_THREADS_LIST_H

#include <deque>
#include <sys/time.h>

using namespace std;

struct wake_up_info
{
    int id;
    timeval awaken_tv;
};

class SleepingThreadsList
{

    deque <wake_up_info> sleeping_threads;

public:

    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 add(int thread_id, timeval timestamp);

    void remove(int tid);

    /*
     * Description: This method removes the thread at the top of this list.
     * If the list is empty, it does nothing.
     */
    void pop();

    /*
     * 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 *peek();

};

#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 _state;
34     int _stack_size;
35     int _id;
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

```
1
2  #include "uthreads.h"
3  #include "thread.h"
4  #include <list>
5  #include <unordered_map>
6  #include <algorithm>
7  #include <iostream>
8  #include <stdio.h>
9  #include <signal.h>
10 #include <sys/time.h>
11 #include <memory>
12 #include "sleeping_threads_list.h"
13
14 // Constants //
15 #define SYS_ERR_CODE 0
16 #define THREAD_ERR_CODE 1
17 #define MAX_THREAD_NUM 100 /* maximal number of threads */
18 #define STACK_SIZE 4096 /* stack size per thread (in bytes) */
19 #define TIMER_SET_MSG "setting the timer has failed."
20 #define INVALID_ID_MSG "thread ID must be between 0 and "+ to_string(MAX_THREAD_NUM) + "."
21 /* External interface */
22
23
24
25 #define ID_NONEXIST_MSG "thread with such ID does not exist."
26
27 #define BLOCK_MAIN_MSG "main thread cannot be blocked."
28
29 #define NEG_TIME_MSG "time must be non-negative."
30
31 #define TIMER_SET_MSG "setting the timer failed."
32
33 #define MAX_THREAD_MSG "max number of threads exceeded."
34 // Using //
35
36 using std::cout,
37 std::endl;
38
39 using std::shared_ptr;
40
41
42 // Static Variables //
43 int total_quantums;
44
45 sigjmp_buf env[2];
46 int current_thread;
47
48 sigset_t sigs_to_block;
49
50 /**
51  * @brief map of all existing threads, with their tid as key.
52  */
53 std::unordered_map<int, shared_ptr<Thread>> threads;
54 /**
55  * @brief list of all ready threads.
56  */
57 std::list<shared_ptr<Thread>> ready_threads;
58 /**
59  * @brief the current running thread.
```

```
 */
shared_ptr<Thread> running_thread;
/**
 * @brief list of all current sleeping threads (id's).
 */
SleepingThreadsList sleeping_threads;
/**
 * @brief timers.
 */

struct itimerval quantum_timer, sleep_timer;
/**
 * @brief sigactions.
 */
struct sigaction quantum_sa, sleep_sa;


// Helper Functions //

void block_signals()
{
    sigprocmask(SIG_BLOCK, &sigs_to_block, NULL);

}

void unblock_signals()
{
    sigprocmask(SIG_UNBLOCK, &sigs_to_block, NULL);

}

int get_min_id()
{
    block_signals();

    for (int i = 0; i < threads.size(); ++i)
    {
        if (threads.find(i) == threads.end())
        {
            unblock_signals();
            return i;
        }
    }
    unblock_signals();
    return threads.size();

}

/**
 * @brief exiting due to error function
 * @param code error code
 * @param text explanatory text for the error
 */
int print_err(int code, string text)
{
    block_signals();
    string prefix;
    switch (code)
    {
        case SYS_ERR_CODE:
            prefix = "system error: ";
            break;
        case THREAD_ERR_CODE:
            prefix = "thread library error: ";
            break;
    }
    cerr << prefix << text << endl;
    if (code == SYS_ERR_CODE)
```

```
128     {
129         exit(1);    // TODO we need to return on failures, but exit makes it irrelevant
130     }
131     else
132     {
133         unblock_signals();
134         return -1;
135     }
136 
137 }
138 
139 void create_main_thread()
140 {
141     shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread());
142     threads[new_thread->get_id()] = new_thread;
143     running_thread = new_thread;
144     running_thread->increase_quantums();
145 }
146 
147 bool does_exist(std::list<shared_ptr<Thread>> lst, int tid)
148 {
149     block_signals();
150     for (std::list<shared_ptr<Thread>>::iterator it = lst.begin(); it != lst.end(); ++it)
151     {
152         if ((*it)->get_id() == tid)
153         {
154             unblock_signals();
155             return true;
156         }
157     }
158     unblock_signals();
159     return false;
160 }
161 
162 void init_sigs_to_block()
163 {
164     block_signals();
165     sigemptyset(&sigs_to_block);
166     sigaddset(&sigs_to_block, SIGALRM);
167     sigaddset(&sigs_to_block, SIGVTALRM);
168     unblock_signals();
169 }
170 
171 
172 timeval calc_wake_up_timeval(int usecs_to_sleep)
173 {
174     block_signals();
175     timeval now, time_to_sleep, wake_up_timeval;
176     gettimeofday(&now, nullptr);
177     time_to_sleep.tv_sec = usecs_to_sleep / 1000000;
178     time_to_sleep.tv_usec = usecs_to_sleep % 1000000;
179     timeradd(&now, &time_to_sleep, &wake_up_timeval);
180     unblock_signals();
181     return wake_up_timeval;
182 }
183 
184 /**
185  * @brief make the front of the ready threads list the current running thread.
186  */
187 void ready_to_running(bool is_blocking = false)
188 {
189     block_signals();
190     int ret_val = sigsetjmp(running_thread->env[0], 1);
191     if (ret_val == 1)
192     {
193         unblock_signals();
194         return;
195     }
```

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

```
264
265 void sleep_handler(int sig)
266 {
267     block_signals();
268     uthread_resume(sleeping_threads.peek()->id);
269     sleeping_threads.pop();
270     wake_up_info *last_sleeping = sleeping_threads.peek();
271     if (last_sleeping != nullptr)
272     {
273         // update sleep_timer values
274         sleep_timer.it_value.tv_sec = last_sleeping->awaken_tv.tv_sec / 1000000;
275         sleep_timer.it_value.tv_usec = last_sleeping->awaken_tv.tv_usec % 1000000;
276         if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
277         {
278             print_err(SYS_ERR_CODE, TIMER_SET_MSG);
279         }
280     }
281     unblock_signals();
282 }
283
284
285 void init_quantum_timer(int quantum_usecs)
286 {
287     quantum_timer.it_value.tv_sec = quantum_usecs / 1000000;
288     quantum_timer.it_value.tv_usec = quantum_usecs % 1000000;
289     quantum_sa.sa_handler = &quantum_handler;
290     if (sigaction(SIGVTALRM, &quantum_sa, NULL) < 0)
291     {
292         print_err(SYS_ERR_CODE, "timer initialization failed.");
293     }
294 }
295
296 void init_sleep_timer()
297 {
298     sleep_timer.it_value.tv_sec = 0;
299     sleep_timer.it_value.tv_usec = 0;
300     sleep_sa.sa_handler = &sleep_handler;
301     if (sigaction(SIGALRM, &sleep_sa, NULL) < 0)
302     {
303         print_err(SYS_ERR_CODE, "timer initialization failed.");
304     }
305
306 }
307
308
309 // API Functions //
310
311 /*
312  * Description: This function initializes the thread library.
313  * You may assume that this function is called before any other thread library
314  * function, and that it is called exactly once. The input to the function is
315  * the length of a quantum in micro-seconds. It is an error to call this
316  * function with non-positive quantum_usecs.
317  * Return value: On success, return 0. On failure, return -1.
318 */
319 int uthread_init(int quantum_usecs)
320 {
321     block_signals();
322     // quantum_usecs cannot be negative
323     if (is_time_invalid(quantum_usecs))
324     {
325         return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
326     }
327     // 1 because of the main thread
328     total_quantums = 1;
329     // init timers
330     init_quantum_timer(quantum_usecs);
331     init_sleep_timer();
```

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

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

```
 * has no effect and is not considered as an error. If no thread with
 * ID tid exists it is considered an error.
 * Return value: On success, return 0. On failure, return -1.
*/
int uthread_resume(int tid)
{
    block_signals();
    if (is_id_invalid(tid))
    {
        return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
    }
    if (is_id_nonexisting(tid))
    {
        return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
    }
    shared_ptr<Thread> curr_thread = threads[tid];
    // if thread to resume is not running or already ready
    if (!does_exist(ready_threads, tid) && !is_running_thread(tid))
    {
        ready_threads.push_back(curr_thread);
    }
    unblock_signals();
    return 0;
}


/*
 * Description: This function blocks the RUNNING thread for user specified micro-seconds (REAL
 * time).
 * It is considered an error if the main thread (tid==0) calls this function.
 * Immediately after the RUNNING thread transitions to the BLOCKED state a scheduling decision
 * should be made.
 * Return value: On success, return 0. On failure, return -1.
*/
int uthread_sleep(unsigned int usec)
{
    block_signals();
    if (is_time_invalid(usec))
    {
        return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
    }
    if (is_main_thread(running_thread->get_id()))
    {
        return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
    }
    if (sleeping_threads.peek() == nullptr)
    {
        // update sleep_timer values
        sleep_timer.it_value.tv_sec = usec / 1000000;
        sleep_timer.it_value.tv_usec = usec % 1000000;
        if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
        {
            print_err(SYS_ERR_CODE, TIMER_SET_MSG);
        }
    }
    sleeping_threads.add(running_thread->get_id(), calc_wake_up_timeval(usec));
    ready_to_running(true);
    unblock_signals();
    return 0;
}


/*
 * Description: This function returns the thread ID of the calling thread.
 * Return value: The ID of the calling thread.
*/
int uthread_get_tid()
{
```

```
536     return running_thread->get_id();
537 }
538
539 /*
540  * Description: This function returns the total number of quantums since
541  * the library was initialized, including the current quantum.
542  * Right after the call to uthread_init, the value should be 1.
543  * Each time a new quantum starts, regardless of the reason, this number
544  * should be increased by 1.
545  * Return value: The total number of quantums.
546 */
547 int uthread_get_total_quantums()
548 {
549     return total_quantums;
550 }
551
552 /*
553  * Description: This function returns the number of quantums the thread with
554  * ID tid was in RUNNING state. On the first time a thread runs, the function
555  * should return 1. Every additional quantum that the thread starts should
556  * increase this value by 1 (so if the thread with ID tid is in RUNNING state
557  * when this function is called, include also the current quantum). If no
558  * thread with ID tid exists it is considered an error.
559  * Return value: On success, return the number of quantums of the thread with ID tid.
560  *                  On failure, return -1.
561 */
562 int uthread_get_quantums(int tid)
563 {
564     if (is_id_invalid(tid))
565     {
566         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
567     }
568     if (is_id_nonexisting(tid))
569     {
570         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
571     }
572     return threads[tid]->get_quantums();
573 }
```