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

uthreads.cpp:20:16: 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:79:20: 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  void ready_to_running(bool) :
uthreads.cpp:194:27: error:  TIMER_SET_MSG  was not declared in this scope
   print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                           ^~~~~~~~~~~~~
uthreads.cpp:194:27: note: suggested alternative:  TIMER_ABSTIME
   print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                           ^~~~~~~~~~~~~
                           TIMER_ABSTIME
uthreads.cpp: In function  void sleep_handler(int) :
uthreads.cpp:262:28: error:  TIMER_SET_MSG  was not declared in this scope
    print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                            ^~~~~~~~~~~~~
uthreads.cpp:262:28: note: suggested alternative:  TIMER_ABSTIME
    print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                            ^~~~~~~~~~~~~
                            TIMER_ABSTIME
uthreads.cpp: In function  int uthread_init(int) :
uthreads.cpp:309:37: error:  NEG_TIME_MSG  was not declared in this scope
   return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
                                     ^~~~~~~~~~~~
uthreads.cpp:309:37: note: suggested alternative:  LC_TIME_MASK
   return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
                                     ^~~~~~~~~~~~
                                     LC_TIME_MASK
uthreads.cpp:319:27: error:  TIMER_SET_MSG  was not declared in this scope
   print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                           ^~~~~~~~~~~~~
uthreads.cpp:319:27: note: suggested alternative:  TIMER_ABSTIME
   print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                           ^~~~~~~~~~~~~
                           TIMER_ABSTIME
uthreads.cpp: In function  int uthread_spawn(void (*)()) :
uthreads.cpp:345:38: error:  MAX_THREAD_MSG  was not declared in this scope
   return (print_err(THREAD_ERR_CODE, MAX_THREAD_MSG));
                                      ^~~~~~~~~~~~~~
uthreads.cpp:345:38: note: suggested alternative:  MAX_THREAD_NUM
   return (print_err(THREAD_ERR_CODE, MAX_THREAD_MSG));
                                      ^~~~~~~~~~~~~~
                                      MAX_THREAD_NUM
uthreads.cpp: In function  int uthread_terminate(int) :
```

```
uthreads.cpp:372:37: error:   INVALID_ID_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:376:37: error:   ID_NONEXIST_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
                                     ^~~~~~~~~~~~~~~
uthreads.cpp: In function   int uthread_block(int)  :
uthreads.cpp:420:37: error:   INVALID_ID_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:424:37: error:   ID_NONEXIST_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
                                     ^~~~~~~~~~~~~~~
uthreads.cpp:428:37: error:   BLOCK_MAIN_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:428:37: note: suggested alternative:   CLOCK_TAI
   return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
                                     ^~~~~~~~~~~~~~
                                     CLOCK_TAI
uthreads.cpp: In function   int uthread_resume(int)  :
uthreads.cpp:461:37: error:   INVALID_ID_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:465:37: error:   ID_NONEXIST_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
                                     ^~~~~~~~~~~~~~~
uthreads.cpp: In function   int uthread_sleep(unsigned int)  :
uthreads.cpp:491:37: error:   NEG_TIME_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
                                     ^~~~~~~~~~~~
uthreads.cpp:491:37: note: suggested alternative:   LC_TIME_MASK
   return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
                                     ^~~~~~~~~~~~
                                     LC_TIME_MASK
uthreads.cpp:495:37: error:   BLOCK_MAIN_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:495:37: note: suggested alternative:   CLOCK_TAI
   return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
                                     ^~~~~~~~~~~~~~
                                     CLOCK_TAI
uthreads.cpp:504:28: error:   TIMER_SET_MSG   was not declared in this scope
    print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                            ^~~~~~~~~~~~
uthreads.cpp:504:28: note: suggested alternative:   TIMER_ABSTIME
    print_err(SYS_ERR_CODE, TIMER_SET_MSG);
                            ^~~~~~~~~~~~
                            TIMER_ABSTIME
uthreads.cpp: In function   int uthread_get_quantums(int)  :
uthreads.cpp:550:37: error:   INVALID_ID_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
                                     ^~~~~~~~~~~~~~
uthreads.cpp:554:37: error:   ID_NONEXIST_MSG   was not declared in this scope
   return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
                                     ^~~~~~~~~~~~~~~
make: *** [<builtin>: uthreads.o] Error 1

libuthreads.a NOT FOUND After "make" command
make command test FAILED!

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

g++: error: libuthreads.a: No such file or directory

Linking FAILED
Linking FAILED!
```

```
128 Pre-submission Test FAILED!
129 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

```
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

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

#ifndef EX2_THREAD_H

#include "uthreads.h"
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <memory>

#define EX2_THREAD_H
#define READY 0
#define BLOCKED 1
#define RUNNING 2

typedef unsigned long address_t;
#define JB_SP 6
#define JB_PC 7


using std::shared_ptr;

class Thread
{

private:
    static int num_of_threads;
protected:
    int _state;
    int _stack_size;
    int _id;
    int _quantums;

public:

    void (*func)(void);

    char *stack;
    sigjmp_buf env[1];
    address_t sp, pc;


    Thread(void (*f)(void) = nullptr, int id=0);

    int get_id() const
    { return _id; };

    int get_state()
    { return _state; };

    void set_state(int state)
    { _state = state; };

    int get_quantums()
    {
```

```
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>

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

// Using //

using std::cout,
std::endl;

using std::shared_ptr;


// Static Variables //
int total_quantums;

sigjmp_buf env[2];
int current_thread;

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;
/**
 * @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;

```

```
60
61  // Helper Functions //
62
63  void block_signals()
64  {
65      sigprocmask(SIG_BLOCK, &sigs_to_block, NULL);
66
67  }
68
69  void unblock_signals()
70  {
71      sigprocmask(SIG_UNBLOCK, &sigs_to_block, NULL);
72
73  }
74
75  int get_min_id()
76  {
77      block_signals();
78
79      for (int i = 0; i < threads.size(); ++i)
80      {
81          if (threads.find(i) == threads.end())
82          {
83              unblock_signals();
84              return i;
85          }
86      }
87      unblock_signals();
88      return threads.size();
89
90  }
91
92  /**
93   * @brief exiting due to error function
94   * @param code error code
95   * @param text explanatory text for the error
96   */
97  int print_err(int code, string text)
98  {
99      block_signals();
100     string prefix;
101     switch (code)
102     {
103         case SYS_ERR_CODE:
104             prefix = "system error: ";
105             break;
106         case THREAD_ERR_CODE:
107             prefix = "thread library error: ";
108             break;
109     }
110     cerr << prefix << text << endl;
111     if (code == SYS_ERR_CODE)
112     {
113         exit(1);    // TODO we need to return on failures, but exit makes it irrelevant
114     }
115     else
116     {
117         unblock_signals();
118         return -1;
119     }
120
121 }
122
123 void create_main_thread()
124 {
125     shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread());
126     threads[new_thread->get_id()] = new_thread;
127     running_thread = new_thread;
```

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

```
196     unblock_signals();
197     siglongjmp(running_thread->env[0], 1);
198 }
199 
200 shared_ptr<Thread> get_ready_thread(int tid)
201 {
202     for (std::list<shared_ptr<Thread>>::iterator it = ready_threads.begin(); it != ready_threads.end(); ++it)
203     {
204         if ((*it)->get_id() == tid)
205         {
206             return *it;
207         }
208     }
209     return nullptr;
210 }
211 
212 
213 bool is_id_invalid(int tid)
214 {
215     return ((tid < 0) || (tid > MAX_THREAD_NUM));
216 
217 }
218 
219 bool is_id_nonexisting(int tid)
220 {
221     return threads.find(tid) == threads.end();
222 }
223 
224 bool is_main_thread(int tid)
225 {
226     return tid == 0;
227 }
228 
229 bool is_time_invalid(int time)
230 {
231     return time < 0;
232 }
233 
234 bool is_running_thread(int tid)
235 {
236     return tid == running_thread->get_id();
237 }
238 
239 // Handlers //
240 void quantum_handler(int sig)
241 {
242 
243     block_signals();
244     ready_to_running();
245     unblock_signals();
246 }
247 
248 
249 void sleep_handler(int sig)
250 {
251     block_signals();
252     uthread_resume(sleeping_threads.peek()->id);
253     sleeping_threads.pop();
254     wake_up_info *last_sleeping = sleeping_threads.peek();
255     if (last_sleeping != nullptr)
256     {
257         // update sleep_timer values
258         sleep_timer.it_value.tv_sec = last_sleeping->awaken_tv.tv_sec / 1000000;
259         sleep_timer.it_value.tv_usec = last_sleeping->awaken_tv.tv_usec % 1000000;
260         if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
261         {
262             print_err(SYS_ERR_CODE, TIMER_SET_MSG);
263         }
```

```
    }
    unblock_signals();
}


void init_quantum_timer(int quantum_usecs)
{
    quantum_timer.it_value.tv_sec = quantum_usecs / 1000000;
    quantum_timer.it_value.tv_usec = quantum_usecs % 1000000;
    quantum_sa.sa_handler = &quantum_handler;
    if (sigaction(SIGVTALRM, &quantum_sa, NULL) < 0)
    {
        print_err(SYS_ERR_CODE, "timer initialization failed.");
    }
}

void init_sleep_timer()
{
    sleep_timer.it_value.tv_sec = 0;
    sleep_timer.it_value.tv_usec = 0;
    sleep_sa.sa_handler = &sleep_handler;
    if (sigaction(SIGALRM, &sleep_sa, NULL) < 0)
    {
        print_err(SYS_ERR_CODE, "timer initialization failed.");
    }

}


// API Functions //

/*
 * Description: This function initializes the thread library.
 * You may assume that this function is called before any other thread library
 * function, and that it is called exactly once. The input to the function is
 * the length of a quantum in micro-seconds. It is an error to call this
 * function with non-positive quantum_usecs.
 * Return value: On success, return 0. On failure, return -1.
*/
int uthread_init(int quantum_usecs)
{
    block_signals();
    // quantum_usecs cannot be negative
    if (is_time_invalid(quantum_usecs))
    {
        return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
    }
    // 1 because of the main thread
    total_quantums = 1;
    // init timers
    init_quantum_timer(quantum_usecs);
    init_sleep_timer();
    // set quantum timer
    if (setitimer(ITIMER_VIRTUAL, &quantum_timer, NULL))
    {
        print_err(SYS_ERR_CODE, TIMER_SET_MSG);
    }
    // create main thread
    create_main_thread();
    // init blocked signals set
    init_sigs_to_block();
    unblock_signals();
    return 0;
}


/*
 * Description: This function creates a new thread, whose entry point is the
```

```
 * function f with the signature void f(void). The thread is added to the end
 * of the READY threads list. The uthread_spawn function should fail if it
 * would cause the number of concurrent threads to exceed the limit
 * (MAX_THREAD_NUM). Each thread should be allocated with a stack of size
 * STACK_SIZE bytes.
 * Return value: On success, return the ID of the created thread.
 * On failure, return -1.
*/
int uthread_spawn(void (*f)(void))
{
    block_signals();
    if (threads.size() == MAX_THREAD_NUM)
    {
        return (print_err(THREAD_ERR_CODE, MAX_THREAD_MSG));
    }
    // create new thread
    shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread(f, get_min_id()));
    threads[new_thread->get_id()] = new_thread;
    ready_threads.push_back(new_thread);
    unblock_signals();
    return new_thread->get_id();
}


/*
 * Description: This function terminates the thread with ID tid and deletes
 * it from all relevant control structures. All the resources allocated by
 * the library for this thread should be released. If no thread with ID tid
 * exists it is considered an error. Terminating the main thread
 * (tid == 0) will result in the termination of the entire process using
 * exit(0) [after releasing the assigned library memory].
 * Return value: The function returns 0 if the thread was successfully
 * terminated and -1 otherwise. If a thread terminates itself or the main
 * thread is terminated, the function does not return.
*/
int uthread_terminate(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);
    }
    //TODO: consider an error and memory deallocation
    if (is_main_thread(tid))
    {
        exit(0);
    }
    // terminate running thread
    if (is_running_thread(tid))
    {
        threads.erase(tid);
        ready_to_running(true);
    }
        // terminate non running thread
    else
    {
        if (does_exist(ready_threads, tid))
        {
            ready_threads.remove(threads[tid]);
        }
        else
        {
            sleeping_threads.remove(tid);
        }
```

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

```
468     // if thread to resume is not running or already ready
469     if (!does_exist(ready_threads, tid) && !is_running_thread(tid))
470     {
471         ready_threads.push_back(curr_thread);
472     }
473     unblock_signals();
474     return 0;
475 }
476 
477 
478 /*
479  * Description: This function blocks the RUNNING thread for user specified micro-seconds (REAL
480  * time).
481  * It is considered an error if the main thread (tid==0) calls this function.
482  * Immediately after the RUNNING thread transitions to the BLOCKED state a scheduling decision
483  * should be made.
484  * Return value: On success, return 0. On failure, return -1.
485 */
486 int uthread_sleep(unsigned int usec)
487 {
488     block_signals();
489     if (is_time_invalid(usec))
490     {
491         return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
492     }
493     if (is_main_thread(running_thread->get_id()))
494     {
495         return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
496     }
497     if (sleeping_threads.peek() == nullptr)
498     {
499         // update sleep_timer values
500         sleep_timer.it_value.tv_sec = usec / 1000000;
501         sleep_timer.it_value.tv_usec = usec % 1000000;
502         if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
503         {
504             print_err(SYS_ERR_CODE, TIMER_SET_MSG);
505         }
506     }
507     sleeping_threads.add(running_thread->get_id(), calc_wake_up_timeval(usec));
508     ready_to_running(true);
509     unblock_signals();
510     return 0;
511 }
512 
513 
514 /*
515  * Description: This function returns the thread ID of the calling thread.
516  * Return value: The ID of the calling thread.
517 */
518 int uthread_get_tid()
519 {
520     return running_thread->get_id();
521 }
522 
523 /*
524  * Description: This function returns the total number of quantums since
525  * the library was initialized, including the current quantum.
526  * Right after the call to uthread_init, the value should be 1.
527  * Each time a new quantum starts, regardless of the reason, this number
528  * should be increased by 1.
529  * Return value: The total number of quantums.
530 */
531 int uthread_get_total_quantums()
532 {
533     return total_quantums;
534 }
535 
```

```
536 /*
537  * Description: This function returns the number of quantums the thread with
538  * ID tid was in RUNNING state. On the first time a thread runs, the function
539  * should return 1. Every additional quantum that the thread starts should
540  * increase this value by 1 (so if the thread with ID tid is in RUNNING state
541  * when this function is called, include also the current quantum). If no
542  * thread with ID tid exists it is considered an error.
543  * Return value: On success, return the number of quantums of the thread with ID tid.
544  *                 On failure, return -1.
545 */
546 int uthread_get_quantums(int tid)
547 {
548     if (is_id_invalid(tid))
549     {
550         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
551     }
552     if (is_id_nonexisting(tid))
553     {
554         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
555     }
556     return threads[tid]->get_quantums();
557 }
```