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 uthreads.cpp: In function  unsigned int get_min_id() :
23 uthreads.cpp:92:20: warning: comparison of integer expressions of different signedness:  int  and  std::unordered_map<int,
24   for (int i = 0; i < threads.size(); ++i)
25                   ~~^~~~~~~~~~~~~~~~
26 uthreads.cpp: In function  int uthread_terminate(int) :
27 uthreads.cpp:416:1: warning: control reaches end of non-void function [-Wreturn-type]
28  }
29  ^
30 In file included from thread.cpp:7:
31 thread.h: In constructor  Thread::Thread(void (*)(), int) :
32 thread.h:35:6: warning:  Thread::_id  will be initialized after [-Wreorder]
33   int _id;
34       ^~
35 thread.h:33:6: warning:    int Thread::_state  [-Wreorder]
36   int _state;
37       ^~~~~
38 thread.cpp:33:1: warning:  when initialized here [-Wreorder]
39  Thread::Thread(void (*f)(void), int id) : _id(id), _state(READY), _stack_size(STACK_SIZE), _quantums(0), func(f)
40  ^~~~~
41 ar: creating libuthreads.a
42
43 stderr: b'uthreads.cpp: In function \xe2\x80\x98unsigned int get_min_id()\xe2\x80\x99:\nuthreads.cpp:92:20: warning: compari
44 make command test FAILED!
45
46 ============== Linking Test ==============
47
48
49 Linking PASSED!
50
51 Pre-submission Test FAILED!
52 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

```
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

```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  void block_signals()
77  {
78      sigprocmask(SIG_BLOCK, &sigs_to_block, NULL);
79
80  }
81
82  void unblock_signals()
83  {
84      sigprocmask(SIG_UNBLOCK, &sigs_to_block, NULL);
85
86  }
87
88  unsigned int get_min_id()
89  {
90      block_signals();
91
92      for (int i = 0; i < threads.size(); ++i)
93      {
94          if (threads.find(i) == threads.end())
95          {
96              unblock_signals();
97              return i;
98          }
99      }
100     unblock_signals();
101     return threads.size();
102
103 }
104
105 /**
106  * @brief exiting due to error function
107  * @param code error code
108  * @param text explanatory text for the error
109  */
110 int print_err(int code, string text)
111 {
112     block_signals();
113     string prefix;
114     switch (code)
115     {
116         case SYS_ERR_CODE:
117             prefix = "system error: ";
118             break;
119         case THREAD_ERR_CODE:
120             prefix = "thread library error: ";
121             break;
122     }
123     cerr << prefix << text << endl;
124     if (code == SYS_ERR_CODE)
125     {
126         exit(1);    // TODO we need to return on failures, but exit makes it irrelevant
127     }
```

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

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

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

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

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

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

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