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:

User level threads are unseen by the kernel, thus making them much more agile and lightweight.
They give the user more control over scheduling and resource management.
A good use for user-level threads in our example is the way we make scheduling decisions -
each threads can receive a very small quantum, and due to the smart on-the-fly scheduling,
the threads are perceived to be running simultaneously.

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:

When hitting the keyboard keys to enter "shotrell", interrupts are sent to the OS, telling it that a key
was pressed. It then acts according to the handler it set for those interrupts.
When sending the command "ps -A", we can see the long list of sigactions that are done, connecting different signals
to their appropriate handlers. For instance, SIGABRT is handled by some specific handler.
At the end of the 'kill pid' statement, we see a SIGTERM signal, whose job is to terminate the process in a 'supervised' way
meaning it's threads can exit correctly.


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     bool does_exist(int tid);
36
37     /*
38      * Description: This method removes the thread at the top of this list.
39      * If the list is empty, it does nothing.
40     */
41     void pop();
42
43     /*
44      * Description: This method returns the information about the thread (id and time it needs to wake up)
45      * at the top of this list without removing it from the list.
46      * If the list is empty, it returns null.
47     */
48     wake_up_info *peek();
49
50 };
51
52 #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);
    }
}

bool SleepingThreadsList::does_exist(int tid){
    for (deque<wake_up_info>::iterator it = sleeping_threads.begin(); it!=sleeping_threads.end();++it){
        if(it->id == tid){
            return true;
        }
    }
    return false;
}

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();
}
```

```
60
61  /*
62   * Description: This method returns the information about the thread (id and time it needs to wake up)
63   * at the top of this list without removing it from the list.
64   * If the list is empty, it returns null.
65  */
66  wake_up_info* SleepingThreadsList::peek(){
67      if (sleeping_threads.empty())
68          return nullptr;
69      return &sleeping_threads.at(0);
70  }
71
```

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;
/**
```

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

std::list<int> blocked_threads;

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);

}

unsigned int get_min_id()
{

    for (unsigned int i = 0; i < threads.size(); ++i)
    {
        if (threads.find(i) == threads.end())
        {
            unblock_signals();
            return i;
        }
    }
    return (unsigned int) 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)
{
    string prefix;
    switch (code)
    {
        case SYS_ERR_CODE:
            prefix = "system error: ";
            break;
        case THREAD_ERR_CODE:
            prefix = "thread library error: ";
            break;
    }
    cout << prefix << text << endl;    // TODO change to cout
    if (code == SYS_ERR_CODE)
    {
        exit(1);    // TODO we need to return on failures, but exit makes it irrelevant
```

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

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

```
264 {
265     return ((tid < 0) || (tid > MAX_THREAD_NUM));
266 
267 }
268 
269 bool is_id_nonexisting(int tid)
270 {
271     return threads.find(tid) == threads.end();
272 }
273 
274 bool is_main_thread(int tid)
275 {
276     return tid == 0;
277 }
278 
279 bool is_time_invalid(int time)
280 {
281     return time < 0;
282 }
283 
284 bool is_running_thread(int tid)
285 {
286     return tid == running_thread->get_id();
287 }
288 
289 // Handlers //
290 void quantum_handler(int sig)
291 {
292 
293     block_signals();
294     ready_to_running();
295     unblock_signals();
296 }
297 
298 
299 void sleep_handler(int sig)
300 {
301     block_signals();
302     int tid = sleeping_threads.peek()->id;
303     sleeping_threads.pop();
304     next_sleeping();
305     if (!does_blocked_exist(blocked_threads, tid))
306     {
307         ready_threads.push_back(threads[tid]);
308 //        uthread_resume(tid);
309     }
310 
311     unblock_signals();
312 }
313 
314 
315 void init_quantum_timer(int quantum_usecs)
316 {
317     quantum_timer.it_value.tv_sec = quantum_usecs / 1000000;
318     quantum_timer.it_value.tv_usec = quantum_usecs % 1000000;
319     quantum_sa.sa_handler = &quantum_handler;
320     if (sigaction(SIGVTALRM, &quantum_sa, NULL) < 0)
321     {
322         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
323     }
324 }
325 
326 void init_sleep_timer()
327 {
328     sleep_timer.it_value.tv_sec = 0;
329     sleep_timer.it_value.tv_usec = 0;
330     sleep_timer.it_interval.tv_usec = 0;
331     sleep_timer.it_interval.tv_usec = 0;
```

```
332     sleep_sa.sa_handler = &sleep_handler;
333     if (sigaction(SIGALRM, &sleep_sa, NULL) < 0)
334     {
335         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
336     }
337 
338 }
339 
340 
341 // API Functions //
342 
343 /*
344  * Description: This function initializes the thread library.
345  * You may assume that this function is called before any other thread library
346  * function, and that it is called exactly once. The input to the function is
347  * the length of a quantum in micro-seconds. It is an error to call this
348  * function with non-positive quantum_usecs.
349  * Return value: On success, return 0. On failure, return -1.
350 */
351 int uthread_init(int quantum_usecs)
352 {
353     init_sigs_to_block();
354     block_signals();
355     // quantum_usecs cannot be negative
356     if (is_time_invalid(quantum_usecs))
357     {
358         return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
359     }
360     // 1 because of the main thread
361     total_quantums = 1;
362     // init timers
363     init_quantum_timer(quantum_usecs);
364     init_sleep_timer();
365     // set quantum timer
366     if (setitimer(ITIMER_VIRTUAL, &quantum_timer, NULL))
367     {
368         print_err(SYS_ERR_CODE, TIMER_SET_MSG);
369     }
370     // create main thread
371     create_main_thread();
372     // init blocked signals set
373     unblock_signals();
374     return 0;
375 }
376 
377 
378 /*
379  * Description: This function creates a new thread, whose entry point is the
380  * function f with the signature void f(void). The thread is added to the end
381  * of the READY threads list. The uthread_spawn function should fail if it
382  * would cause the number of concurrent threads to exceed the limit
383  * (MAX_THREAD_NUM). Each thread should be allocated with a stack of size
384  * STACK_SIZE bytes.
385  * Return value: On success, return the ID of the created thread.
386  * On failure, return -1.
387 */
388 int uthread_spawn(void (*f)(void))
389 {
390     block_signals();
391     if (threads.size() == MAX_THREAD_NUM)
392     {
393         return (print_err(THREAD_ERR_CODE, MAX_THREAD_MSG));
394     }
395     // create new thread
396     shared_ptr<Thread> new_thread = std::make_shared<Thread>(Thread(f, get_min_id()));
397     threads[new_thread->get_id()] = new_thread;
398     ready_threads.push_back(new_thread);
399     unblock_signals();
```

```
400     return new_thread->get_id();
401 }
402 
403 
404 /*
405  * Description: This function terminates the thread with ID tid and deletes
406  * it from all relevant control structures. All the resources allocated by
407  * the library for this thread should be released. If no thread with ID tid
408  * exists it is considered an error. Terminating the main thread
409  * (tid == 0) will result in the termination of the entire process using
410  * exit(0) [after releasing the assigned library memory].
411  * Return value: The function returns 0 if the thread was successfully
412  * terminated and -1 otherwise. If a thread terminates itself or the main
413  * thread is terminated, the function does not return.
414 */
415 int uthread_terminate(int tid)
416 {
417     block_signals();
418     if (is_id_invalid(tid))
419     {
420         unblock_signals();
421         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
422     }
423     if (is_id_nonexisting(tid))
424     {
425         unblock_signals();
426         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
427     }
428     //TODO: consider an error and memory deallocation
429     if (is_main_thread(tid))
430     {
431         unblock_signals();
432         exit(0);
433     }
434     // terminate running thread
435     if (is_running_thread(tid))
436     {
437         threads.erase(tid);
438         ready_to_running(true);
439     }
440         // terminate non running thread
441     else
442     {
443         if (does_exist(ready_threads, tid))
444         {
445             ready_threads.remove(threads[tid]);
446             threads.erase(tid);
447         }
448         else
449         {
450 
451             if (sleeping_threads.peek() != nullptr)
452             {
453                 int curr_sleeper_id = sleeping_threads.peek()->id;
454                 sleeping_threads.remove(tid);
455                 if (curr_sleeper_id == tid)
456                 {
457                     next_sleeping();
458                 }
459             }
460             threads.erase(tid);
461         }
462     }
463     unblock_signals();
464     return 0;
465 }
466 
467 
```

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

```
536             blocked_threads.remove(tid);
537             if (!sleeping_threads.does_exist(tid))
538             {
539                 ready_threads.push_back(curr_thread);
540             }
541
542
543
544     //        return 0;
545         }
546             unblock_signals();
547             return 0;
548     }
549
550
551     /*
552      * Description: This function blocks the RUNNING thread for user specified micro-seconds (REAL
553      * time).
554      * It is considered an error if the main thread (tid==0) calls this function.
555      * Immediately after the RUNNING thread transitions to the BLOCKED state a scheduling decision
556      * should be made.
557      * Return value: On success, return 0. On failure, return -1.
558     */
559     int uthread_sleep(unsigned int usec)
560     {
561         block_signals();
562         if (is_time_invalid(usec))
563         {
564             unblock_signals();
565             return print_err(THREAD_ERR_CODE, NEG_TIME_MSG);
566         }
567         if (is_main_thread(running_thread->get_id()))
568         {
569             unblock_signals();
570             return print_err(THREAD_ERR_CODE, BLOCK_MAIN_MSG);
571         }
572         if (usec == 0)
573         {
574             ready_to_running();
575             unblock_signals();
576             return 0;
577         }
578         if (sleeping_threads.peek() == nullptr)
579         {
580
581             // update sleep_timer values
582             sleep_timer.it_value.tv_sec = usec / 1000000;
583             sleep_timer.it_value.tv_usec = usec % 1000000;
584             if (setitimer(ITIMER_REAL, &sleep_timer, NULL))
585             {
586                 print_err(SYS_ERR_CODE, TIMER_SET_MSG);
587             }
588         }
589         sleeping_threads.add(running_thread->get_id(), calc_wake_up_timeval(usec));
590         ready_to_running(true);
591         unblock_signals();
592         return 0;
593     }
594
595
596     /*
597      * Description: This function returns the thread ID of the calling thread.
598      * Return value: The ID of the calling thread.
599     */
600     int uthread_get_tid()
601     {
602         return running_thread->get_id();
603     }
```

```
604
605 /*
606  * Description: This function returns the total number of quantums since
607  * the library was initialized, including the current quantum.
608  * Right after the call to uthread_init, the value should be 1.
609  * Each time a new quantum starts, regardless of the reason, this number
610  * should be increased by 1.
611  * Return value: The total number of quantums.
612 */
613 int uthread_get_total_quantums()
614 {
615     return total_quantums;
616 }
617
618 /*
619  * Description: This function returns the number of quantums the thread with
620  * ID tid was in RUNNING state. On the first time a thread runs, the function
621  * should return 1. Every additional quantum that the thread starts should
622  * increase this value by 1 (so if the thread with ID tid is in RUNNING state
623  * when this function is called, include also the current quantum). If no
624  * thread with ID tid exists it is considered an error.
625  * Return value: On success, return the number of quantums of the thread with ID tid.
626  *                  On failure, return -1.
627 */
628 int uthread_get_quantums(int tid)
629 {
630     if (is_id_invalid(tid))
631     {
632         return print_err(THREAD_ERR_CODE, INVALID_ID_MSG);
633     }
634     if (is_id_nonexisting(tid))
635     {
636         return print_err(THREAD_ERR_CODE, ID_NONEXIST_MSG);
637     }
638     return threads[tid]->get_quantums();
639 }
```