

Contents

1	Basic Test Results	2
2	README	3
3	Makefile	4
4	sleeping threads list.h	5
5	sleeping threads list.cpp	6
6	thread.h	8
7	thread.cpp	10
8	uthreads.cpp	11

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:94:20: 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:442:1: 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 (*)(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:94:20: 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

```
1 tomka, alonemanuel
2 Tom Kalir (316426485), Alon Emanuel (205894058)
3 EX: 2
4
5 FILES:
6 sleeping_threads_list.h -- a file with some code
7 sleeping_threads_list.cpp -- a file with some code
8 thread.h -- a file with some code
9 thread.cpp -- a file with some code
10 uthreads.cpp -- a file with some code
11
12 REMARKS:
13
14 ANSWERS:
15
16 Question 1:
17
18
19 Question 2:
20
21 Advantages:
22 1. Memory and resource management is made easier and more beneficial -
23 when a tab is closed (or minimized\hidden), its resources can
24 be freed or shared among other currently-active processes.
25 2. Bugs and crashes are focused to the specific tab (=process), thus giving
26 the browser a more stable performance that can 'survive' tab-specific errors.
27
28 Disadvantages:
29 1. As opposed to user-level threads, each process creates a lot of overhead
30 and is considered to be slow and inefficient.
31 2. Although resource management is indeed efficient in chrome's implementation,
32 but resource *use* and demand is greater, due to the need to store data for each of these
33 processes (which are not 'hidden' from the OS like user-level threads are).
34
35
36 Question 3.a:
37
38 Question 3.b:
39
40 Question 3.c:
41
42 Question 4:
43 As described in "man itimer":
44 Real time 'passes' as real-life time,
45 while virtual time passes only when the process is executing.
46 We use *virtual* time in our quantum timer - it sends a signal when a virtual quantum has passed.
47 We use *real* time in our sleep timer - it send a signal when a real time 'unit' has passed.
```

3 Makefile

```
1  CC=g++
2  CXX=g++
3  RANLIB=ranlib
4
5  LIBSRC=uthreads.cpp thread.h thread.cpp sleeping_threads_list.h sleeping_threads_list.cpp
6  LIBOBJ=$(LIBSRC:.cpp=.o)
7
8  INCS=-I.
9  CFLAGS = -Wall -std=c++11 -g $(INCS)
10 CXXFLAGS = -Wall -std=c++11 -g $(INCS)
11
12 OSMLIB = libuthreads.a
13 TARGETS = $(OSMLIB)
14
15 TAR=tar
16 TARFLAGS=-cvf
17 TARNAME=ex2.tar
18 TARSRC=$(LIBSRC) Makefile README
19
20 all: $(TARGETS)
21
22 $(TARGETS): $(LIBOBJ)
23     $(AR) $(ARFLAGS) $@ $^
24     $(RANLIB) $@
25
26 clean:
27     $(RM) $(TARGETS) $(OBJ) $(LIBOBJ) *~ *core
28
29 depend:
30     makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)
31
32 tar:
33     $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRC)
```

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

```
1  #include "sleeping_threads_list.h"
2
3  SleepingThreadsList::SleepingThreadsList() {
4  }
5
6
7  /*
8   * Description: This method adds a new element to the list of sleeping
9   * threads. It gets the thread's id, and the time when it needs to wake up.
10  * The wakeup_tv is a struct timeval (as specified in <sys/time.h>) which
11  * contains the number of seconds and microseconds since the Epoch.
12  * The method keeps the list sorted by the threads' wake up time.
13  */
14  void SleepingThreadsList::add(int thread_id, timeval wakeup_tv) {
15
16      wake_up_info new_thread;
17      new_thread.id = thread_id;
18      new_thread.awaken_tv = wakeup_tv;
19
20      if(sleeping_threads.empty()){
21          sleeping_threads.push_front(new_thread);
22      }
23      else {
24          for (deque<wake_up_info>::iterator it = sleeping_threads.begin(); it != sleeping_threads.end(); ++it){
25              if(timercmp(&it->awaken_tv, &wakeup_tv, >=)){
26                  sleeping_threads.insert(it, new_thread);
27                  return;
28              }
29          }
30          sleeping_threads.push_back(new_thread);
31      }
32  }
33
34  void SleepingThreadsList::remove(int tid){
35      for (deque<wake_up_info>::iterator it = sleeping_threads.begin(); it != sleeping_threads.end(); ++it){
36          if(it->id == tid){
37              sleeping_threads.erase(it);
38              return;
39          }
40      }
41  }
42
43  /*
44   * Description: This method removes the thread at the top of this list.
45   * If the list is empty, it does nothing.
46  */
47  void SleepingThreadsList::pop() {
48      if(!sleeping_threads.empty())
49          sleeping_threads.pop_front();
50  }
51
52  /*
53   * Description: This method returns the information about the thread (id and time it needs to wake up)
54   * at the top of this list without removing it from the list.
55   * If the list is empty, it returns null.
56  */
57  wake_up_info* SleepingThreadsList::peek(){
58      if (sleeping_threads.empty())
59          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

```
1  //
2  // Created by kalir on 27/03/2019.
3  //
4
5  #include <iostream>
6  #include "uthreads.h"
7  #include "thread.h"
8  #include <memory>
9  #define STACK_SIZE 4096 /* stack size per thread (in bytes) */
10
11 using std::cout;
12 using std::endl;
13
14
15 int Thread::num_of_threads = 0;
16
17 /* A translation is required when using an address of a variable.
18    Use this as a black box in your code. */
19 address_t translate_address(address_t addr)
20 {
21     address_t ret;
22     asm volatile("xor    %%fs:0x30,%0\n"
23                 "rol     $0x11,%0\n"
24                 : "=g" (ret)
25                 : "0" (addr));
26     return ret;
27 }
28
29 /**
30  * @brief Constructor of a thread object
31  * @param f thread function address
32  */
33 Thread::Thread(void (*f)(void), int id) : _id(id), _state(READY), _stack_size(STACK_SIZE), _quantums(0), func(f)
34 {
35     stack = new char[STACK_SIZE];
36     // if (num_of_threads)
37     // {
38     //     cout << "Creating thread!" << endl;
39     sp = (address_t) stack + STACK_SIZE - sizeof(address_t);
40     pc = (address_t) f;
41     sigsetjmp(env[0], 1);
42     (env[0]->__jmpbuf)[JB_SP] = translate_address(sp);
43     (env[0]->__jmpbuf)[JB_PC] = translate_address(pc);
44     sigemptyset(&env[0]->__saved_mask);
45     // }
46     num_of_threads++;
47 }
48
49
50 bool Thread::operator==(const Thread &other) const
51 {
52     return _id == other.get_id();
53 }
54
```

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 MAX_THREAD_MSG "max number of threads exceeded."
32 // Using //
33
34 using std::cout;
35 using std::endl;
36
37 using std::shared_ptr;
38
39
40 // Static Variables //
41 int total_quantums;
42
43 sigjmp_buf env[2];
44
45 sigset_t sigs_to_block;
46
47 /**
48  * @brief map of all existing threads, with their tid as key.
49  */
50 std::unordered_map<int, shared_ptr<Thread>> threads;
51 /**
52  * @brief list of all ready threads.
53  */
54 std::list<shared_ptr<Thread>> ready_threads;
55 /**
56  * @brief the current running thread.
57  */
58 shared_ptr<Thread> running_thread;
59 /**
```

```

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  // Helper Functions //
74
75
76
77  void block_signals()
78  {
79
80      sigprocmask(SIG_BLOCK, &sigs_to_block, NULL);
81
82  }
83
84  void unblock_signals()
85  {
86      sigprocmask(SIG_UNBLOCK, &sigs_to_block, NULL);
87
88  }
89
90  unsigned int get_min_id()
91  {
92      block_signals();
93
94      for (int i = 0; i < threads.size(); ++i)
95      {
96          if (threads.find(i) == threads.end())
97          {
98              unblock_signals();
99              return i;
100          }
101      }
102      unblock_signals();
103      return threads.size();
104
105  }
106
107  /**
108   * @brief exiting due to error function
109   * @param code error code
110   * @param text explanatory text for the error
111   */
112  int print_err(int code, string text)
113  {
114      block_signals();
115      string prefix;
116      switch (code)
117      {
118          case SYS_ERR_CODE:
119              prefix = "system error: ";
120              break;
121          case THREAD_ERR_CODE:
122              prefix = "thread library error: ";
123              break;
124      }
125      cout << prefix << text << endl;    // TODO change to cout
126      if (code == SYS_ERR_CODE)
127      {

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

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