

Contents

1	Basic Test Results	2
2	README	3
3	Makefile	4
4	thread.h	5
5	thread.cpp	6
6	uthreads.cpp	8

1 Basic Test Results

```
1  Checking logins...
2  got ['e342791191', 'archsak']
3  Make sure this is correct
4
5
6  Checking files...
7
8
9  Testing Makefile...
10 Passed basic make test
11
12
13 Compile test...
14 Passed basic compile test
15
16
17 Running basic tests:
18
19
20 Command '['./runtest']' timed out after 9.999947431031615 seconds
21 FAILED! - TIMEOUT
22
23
24 Command '['./runtest']' timed out after 9.999937523622066 seconds
25 FAILED! - TIMEOUT
26
27
28
29
30 #####
31 # One or more tests failed #
32 # see more details above  #
33 # Pre-submit test failed!  #
34 #####
```

2 README

```
1 e342791191, archsak
2 Elsa Sebagh (342791191), Aharon Saksonov(207600164)
3 EX: 2
4
5 FILES:
6 thread.cpp
7 thread.h
8 uthreads.cpp
9 makefile
10
11 ANSWERS:
12 1. The sigsetjmp function saves the current state of the thread while the siglongjmp function restores the state
13 of the thread that was saved by the sigsetjmp function. The state contains the context of the thread, and the signal mask.
14 The sigsetjmp function receives env as a parameter, which contains among other things the signal mask - a list of signals
15 to ignore. When calling siglongjmp, the state of the thread is restored to the state saved by sigsetjmp, including the
16 signal mask.
17
18 2. A server could use user level threads to handle multiple clients. The server needs to handle each request concurrently,
19 and user level threads can be used to handle each request in a separate thread. The server can create a new thread for
20 each client request, and the thread can handle the request. This allows quick context switching between threads, without
21 the overhead of processes or kernel level threads. This way the server can handle multiple clients concurrently without
22 high overhead of using the operating system.
23
24 3. Advantages: Processes assure that the program will continue to run even if one of the processes crashes. They also
25 assure a high level of security, as each process has its own memory space and cannot access the memory of other processes,
26 in this example they assure one tab cannot access the informations of another tab (passwords, credits card and such).
27 Disadvantages: Processes are slower to create and manage than threads, as they require more resources. They also require
28 more memory, as each process has its own memory space. In this example, this can be a problem when a user opens multiple
29 tabs at once as each tab will require more memory and the overall performance of the browser will decrease.
30
31 4. There are different kinds of signals and interrupts involved in the process. First, a keyboard interrupt when typing
32 in the command shell.
33 Then, the kill command is a OS interrupt. When it is executed, it sends a signal to the specified process (Shotwell).
34 The operating system manages signals and interrupts at the kernel level to handle this command.
35 When receiving the kill command, the OS sends a signal to the application's process. Then Shotwell receives the signal
36 and starts its shutdown procedure.
37
38 5. Real time is the actual time that it takes for a thread or a process to run, in the physical hardware.
39 On the other hand, virtual time is the time that the thread or process perceives that it takes to run. It depends on
40 the scheduling algorithm and the number of threads or processes running on the system. Virtual time can be different from
41 real time, as the thread or process may be waiting for other threads or processes to finish before it can run.
```

3 Makefile

```
1  # Variables
2  CXX = g++
3  AR = ar
4  CXXFLAGS = -Wall -g -std=c++11
5  ARFLAGS = rcs
6  TARGET = libuthreads.a
7  OBJECTS = uthreads.o thread.o
8
9  # Default target
10 all: $(TARGET)
11
12 # Rule to create the static library
13 $(TARGET): $(OBJECTS)
14     $(AR) $(ARFLAGS) $(TARGET) $(OBJECTS)
15
16 # Rules to compile the object files
17 uthreads.o: uthreads.cpp uthreads.h
18     $(CXX) $(CXXFLAGS) -c uthreads.cpp -o uthreads.o
19
20 thread.o: thread.cpp thread.h
21     $(CXX) $(CXXFLAGS) -c thread.cpp -o thread.o
22
23 # Clean rule to remove generated files
24 clean:
25     rm -f $(OBJECTS) $(TARGET)
26
27 .PHONY: all clean
```

4 thread.h

```
1  #ifndef THREAD_H
2  #define THREAD_H
3
4  #include "uthreads.h"
5  #include <setjmp.h>
6
7  enum State
8  {
9      READY, RUNNING, BLOCKED
10 };
11
12 class Thread
13 {
14 public:
15
16     Thread (int tid);
17     Thread (int tid, void (*entry_point) (void));
18     ~Thread ();
19     int get_tid ();
20     State get_state ();
21     thread_entry_point get_entry_point ();
22     int get_bound ();
23     char *get_stack ();
24     int get_quantum_counter ();
25     void set_state (State state);
26     sigjmp_buf *get_env ();
27
28
29
30     //quantum counters
31
32 private:
33     int _tid;
34     void (*_entry_point) (void);
35     State _state;
36     char *_stack;
37     int _bound;
38     int _quantum_counter;
39     sigjmp_buf _env;
40
41 };
42
43 #endif //THREAD_H
```

5 thread.cpp

```
1  #include "thread.h"
2  #include <iostream>
3  #include <setjmp.h>
4  #include <signal.h>
5  #include <unistd.h>
6  #include <sys/time.h>
7  #include <stdbool.h>
8
9  #ifdef __x86_64__
10 /* code for 64 bit Intel arch */
11
12 typedef unsigned long address_t;
13 #define JB_SP 6
14 #define JB_PC 7
15
16 /* A translation is required when using an address of a variable.
17    Use this as a black box in your code. */
18 address_t translate_address(address_t addr)
19 {
20     address_t ret;
21     asm volatile("xor    %%fs:0x30,%0\n"
22                 "rol     $0x11,%0\n"
23                 : "=g" (ret)
24                 : "0" (addr));
25     return ret;
26 }
27
28 #else
29 /* code for 32 bit Intel arch */
30
31 typedef unsigned int address_t;
32 #define JB_SP 4
33 #define JB_PC 5
34
35 /* A translation is required when using an address of a variable.
36    Use this as a black box in your code. */
37 address_t translate_address(address_t addr)
38 {
39     address_t ret;
40     asm volatile("xor    %%gs:0x18,%0\n"
41                 "rol     $0x9,%0\n"
42                 : "=g" (ret)
43                 : "0" (addr));
44     return ret;
45 }
46
47 #endif
48
49 // Constructor for main
50 Thread::Thread(int tid)
51 : _tid(tid), _state(READY), _quantum_counter(0)
52 {
53     _stack = new char[STACK_SIZE];
54 }
55
56 Thread::Thread(int tid, void (*entry_point)(void))
```

```

60         : _tid(tid),
61         _entry_point(entry_point), _state(READY), _bound(STACK_SIZE),
62         _quantum_counter(0)
63     {
64         _stack = new char[STACK_SIZE];
65         address_t sp, pc;
66         sp = (address_t) _stack + STACK_SIZE - sizeof(address_t);
67         pc = (address_t) _entry_point;
68         if (sigsetjmp(_env, 1) == 0) {
69             (_env->__jmpbuf)[JB_SP] = translate_address(sp);
70             (_env->__jmpbuf)[JB_PC] = translate_address(pc);
71             sigemptyset(&_env->__saved_mask);
72         } // TODO add throw error?
73     }
74
75     Thread::~Thread()
76     {
77         delete[] _stack;
78     }
79
80     int Thread::get_tid() {return _tid;}
81
82     State Thread::get_state() {return _state;}
83
84     thread_entry_point Thread::get_entry_point() {return _entry_point;}
85
86     int Thread::get_bound() {return _bound;}
87
88     char* Thread::get_stack() {return _stack;}
89
90     int Thread::get_quantum_counter() {return _quantum_counter;}
91
92     void Thread::set_state(State state)
93     {
94         if (state == RUNNING && _state == READY)
95         {
96             _quantum_counter++;
97         }
98         _state = state;
99     }
100
101     sigjmp_buf* Thread::get_env() {
102         return &_env;
103     }

```

6 uthreads.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4  #include <list>
5  #include <sys/time.h>
6  #include <bits/stdc++.h>
7
8  #include "thread.h"
9  #include "uthreads.h"
10
11 using namespace std;
12
13 int USECS_IN_SEC = 1000000;
14
15 int quantum_length;
16 int quantum_counter;
17 list<Thread *> ready_queue;
18 std::map<int, Thread *> threads;
19 std::map<int, int> sleeping_threads;
20 Thread *running_thread = nullptr;
21 priority_queue<int, vector<int>, greater<int> > indexes; // Min heap
22 struct itimerval timer = {0};
23 sigset_t signals;
24 struct sigaction sa = {0};
25
26
27 // Add counter map for blocked threads
28
29
30 void print_system_error (string error)
31 {
32     std::cerr << "system error: " << error << std::endl;
33     exit (1);
34 }
35
36 void print_library_error (string error)
37 {
38     std::cerr << "thread library error: " << error << std::endl;
39     return;
40 }
41
42 void blockTimer ()
43 {
44     if (sigprocmask (SIG_BLOCK, &signals, NULL) == -1)
45     {
46         print_system_error ("sigblock error");
47         exit (1);
48     }
49 }
50
51 void resumeTimer ()
52 {
53     if (sigprocmask (SIG_UNBLOCK, &signals, NULL) == -1)
54     {
55         print_system_error ("sigunblock error");
56         exit (1);
57     }
58 }
59
```



```

60 void switch_threads (bool is_terminating = false)
61 {
62     if (ready_queue.empty () && running_thread != nullptr)
63     {
64         running_thread->set_state (READY);
65         running_thread->set_state (RUNNING);
66         return;
67     };
68     std::cout << *(running_thread->get_env ()) << std::endl;
69     if (sigsetjmp (*(running_thread->get_env ()), 1) < 0)
70     {
71         print_system_error ("sigsetjmp error");
72         exit (1);
73     }
74     // Save current thread state
75     if (!is_terminating)
76     {
77         // Move running thread to the ready queue
78         running_thread->set_state (READY);
79         ready_queue.push_back (running_thread);
80     }
81     // Get the next thread to run
82     running_thread = ready_queue.front ();
83     ready_queue.pop_front ();
84
85     // Set the state to RUNNING and increment quantum counter
86     running_thread->set_state (RUNNING);
87
88     siglongjmp (*(running_thread->get_env ()), 1) ;
89
90     if (setitimer (ITIMER_VIRTUAL, &timer, nullptr) == -1)
91     {
92         print_system_error ("setitimer error.");
93         exit (1);
94     }
95 }
96
97 void timer_handler (int sig, bool isTerminating = false)
98 {
99     // Increment quantum counter and switch threads if needed
100    quantum_counter++;
101    vector<int> to_awake;
102    for (auto thread = sleeping_threads.begin ();
103         thread != sleeping_threads.end ());
104    {
105        thread->second--;
106        if (thread->second == 0)
107        {
108            to_awake.push_back (thread->first);
109            sleeping_threads.erase (thread->first);
110        }
111    }
112
113    for (int tid: to_awake)
114    {
115        sleeping_threads.erase (tid);
116        if (threads[tid]->get_state () == READY)
117        {
118            ready_queue.push_back (threads[tid]);
119        }
120    }
121    switch_threads (isTerminating);
122 }
123
124 void timer_handler_no_bool (int sig)
125 {
126     timer_handler (sig, false);
127 }

```

```

128
129 int uthread_init (int quantum_usecs)
130 {
131     if (quantum_usecs <= 0)
132     {
133         print_library_error ("quantum_usecs must be positive");
134         return -1;
135     }
136     for (int i = 1; i < MAX_THREAD_NUM; i++)
137     {
138         indexes.push (i);
139     }
140     quantum_counter = 0;
141     quantum_length = quantum_usecs;
142     Thread *main = new Thread (0);
143     threads[0] = main;
144     // Set up timer and signal handler
145     sa.sa_handler = &timer_handler_no_bool;
146     if (sigemptyset (&sa.sa_mask) == -1)
147     {
148         print_system_error ("sigemptyset error.");
149         exit (1);
150     }
151     sa.sa_flags = 0;
152
153     if (sigaction (SIGVTALRM, &sa, NULL) < 0)
154     {
155         print_system_error ("sigaction error.");
156     }
157
158     timer.it_value.tv_sec = quantum_usecs / USECS_IN_SEC;
159     timer.it_value.tv_usec = quantum_usecs % USECS_IN_SEC;
160     timer.it_interval.tv_sec = quantum_usecs / USECS_IN_SEC;
161     timer.it_interval.tv_usec = quantum_usecs % USECS_IN_SEC;
162
163     running_thread = main;
164     main->set_state (RUNNING);
165     // Start a virtual timer. It counts down whenever this process is executing.
166     if (setitimer (ITIMER_VIRTUAL, &timer, nullptr))
167     {
168         print_system_error ("setitimer error.");
169         exit (1);
170     }
171     // create sigset for blocking signals later on
172     if (sigemptyset (&signals) == -1)
173     {
174         print_system_error ("sigemptyset error.");
175         exit (1);
176     }
177     if (sigaddset (&signals, SIGVTALRM) == -1)
178     {
179         print_system_error ("sigaddset error.");
180         exit (1);
181     }
182
183     // Add signals somehow
184     quantum_counter = 1;
185     return 0;
186 }
187
188 int uthread_spawn (thread_entry_point entry_point)
189 {
190     blockTimer ();
191     if (!entry_point)
192     {
193         print_library_error ("entry_point must not be null");
194         resumeTimer ();
195         return -1;

```

```

196     }
197     if (threads.size () >= MAX_THREAD_NUM)
198     {
199         print_library_error ("thread limit reached");
200         resumeTimer ();
201         return -1;
202     }
203     // Create new thread
204     int next_index = indexes.top ();
205     indexes.pop ();
206     Thread *new_thread = new Thread (next_index, entry_point);
207     threads[next_index] = new_thread;
208     if (threads[next_index] != nullptr && threads[next_index]->get_state () ==
209         READY)
210     {
211         ready_queue.push_back (new_thread);
212     }
213     resumeTimer ();
214     return next_index;
215 }
216
217 int uthread_terminate (int tid)
218 {
219     fflush (stdout);
220     blockTimer ();
221     if (tid == 0 && running_thread->get_tid () == 0) // Terminating main
222     {
223         ready_queue.clear ();
224         sleeping_threads.clear ();
225         for (auto thread = threads.begin (); thread != threads.end ();)
226         {
227             if (thread->first != 0)
228             {
229                 delete thread->second;
230                 thread = threads.erase (thread);
231             }
232             else
233             {
234                 ++thread;
235             }
236         }
237         exit (0);
238     }
239     else
240     {
241         if (tid == 0)
242         {
243             print_library_error ("cannot terminate main thread");
244             resumeTimer ();
245             return -1;
246         }
247         else if (threads.find (tid) == threads.end ())
248         {
249             print_library_error ("in terminate: thread does not exist, tid: " + tid);
250             resumeTimer ();
251             return -1;
252         }
253         else
254         {
255             // Handle termination
256             Thread *thread_to_terminate = threads[tid];
257             if (sleeping_threads.find (tid) != sleeping_threads.end ())
258             {
259                 sleeping_threads.erase (tid);
260             }
261             else if (thread_to_terminate->get_state () == READY &&
262                 std::find (ready_queue.begin (), ready_queue.end (), thread_to_terminate)
263                 != ready_queue.end ())

```

```

264     {
265         ready_queue.remove (thread_to_terminate);
266     }
267     else if (running_thread != nullptr && tid == running_thread->get_tid ())
268     {
269         indexes.push (tid);
270         threads.erase (tid);
271         delete thread_to_terminate;
272         resumeTimer ();
273         timer_handler (0, true);
274         return 0;
275     }
276     threads.erase (tid);
277     indexes.push (tid);
278     delete thread_to_terminate;
279     resumeTimer ();
280     return 0;
281 }
282 }
283 }
284
285 int uthread_block (int tid)
286 {
287     blockTimer ();
288     if (threads.find (tid) == threads.end ())
289     {
290         print_library_error ("in block: thread does not exist, tid: " + tid);
291         resumeTimer ();
292
293         return -1;
294     }
295     else if (tid == 0)
296     {
297         print_library_error ("cannot block main thread");
298         resumeTimer ();
299
300         return -1;
301     }
302     // Change thread state
303     if (threads[tid] != nullptr && threads[tid]->get_state () == RUNNING)
304     {
305         sigsetjmp (*(threads[tid]->get_env ()), 1);
306         timer_handler (0, true);
307     }
308     if (std::find (ready_queue.begin (), ready_queue.end (), threads[tid]) !=
309         ready_queue.end ())
310     {
311         ready_queue.remove (threads[tid]);
312     }
313     threads[tid]->set_state (BLOCKED);
314     resumeTimer ();
315     return 0;
316 }
317
318 int uthread_resume (int tid)
319 {
320     blockTimer ();
321     if (threads.find (tid) == threads.end ())
322     {
323         print_library_error ("in resume: thread does not exist, tid: " + tid);
324         resumeTimer ();
325         return -1;
326     }
327     else if (threads[tid]->get_state () != BLOCKED)
328     {
329         resumeTimer ();
330         return 0;
331     }

```

```

332
333 // Change state to READY, add to queue
334 threads[tid]->set_state (READY);
335 // Add to ready only if it's not sleeping
336 if (sleeping_threads.find (tid) == sleeping_threads.end ())
337 {
338     ready_queue.push_back (threads[tid]);
339 }
340 resumeTimer ();
341 return 0;
342 }
343
344 int uthread_sleep (int num_quantums)
345 {
346     blockTimer ();
347     if (num_quantums <= 0)
348     {
349         print_library_error ("num_quantums must be positive");
350         resumeTimer ();
351         return -1;
352     }
353     if (uthread_get_tid () == 0)
354     {
355         print_library_error ("cannot put main thread to sleep");
356         resumeTimer ();
357         return -1;
358     }
359     sleeping_threads[uthread_get_tid ()] = num_quantums;
360     if (threads[uthread_get_tid ()] != nullptr &&
361         threads[uthread_get_tid ()]->get_state () == RUNNING)
362     {
363         timer_handler (0, true);
364     }
365     else if (threads[uthread_get_tid ()] != nullptr &&
366             threads[uthread_get_tid ()]->get_state () == READY)
367     {
368         ready_queue.remove (threads[uthread_get_tid ()]);
369     }
370     resumeTimer ();
371
372     return 0;
373 }
374
375 int uthread_get_tid ()
376 {
377     blockTimer ();
378     int id = running_thread->get_tid ();
379     resumeTimer ();
380     return id;
381 }
382
383 int uthread_get_total_quantums ()
384 {
385     blockTimer ();
386     int count = quantum_counter;
387     resumeTimer ();
388     return count;
389 }
390
391 int uthread_get_quantums (int tid)
392 {
393     blockTimer ();
394     if (threads.find (tid) == threads.end ())
395     {
396         print_library_error ("in get_quantums: thread does not exist, tid " + tid);
397         resumeTimer ();
398         return -1;
399     }

```

```
400 // Return thread quantum counter
401     int counter = threads[tid]->get_quantum_counter ();
402     resumeTimer ();
403     return counter;
404 }
405
406
407
408
409
```