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

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

2 README

21

22 23

24

25

26

27

28

29 30 31

```
e342791191, archsak
1
2
    Elsa Sebagh (342791191), Aharon Saksonov(207600164)
3
4
    FILES:
    thread.cpp
6
    thread.h
8
    uthreads.cpp
    makefile
9
10
    ANSWERS:
11
    1. The sigsetjmp function saves the current state of the thread while the siglongjmp function restores the state
12
    of the thread that was saved by the sigsetjmp function. The state contains the context of the thread, and the signal mask.
    The sigsetjmp function receives env as a parameter, which contains among other things the signal mask - a list of signals
14
15
    to ignore. When calling siglongjmp, the state of the thread is restored to the state saved by sigsetjmp, including the
    signal mask.
16
17
18
19
20
```

2. A server could use user level threads to handle multiple clients. The server needs to handle each request concurrently, and user level threads can be used to handle each request in a separate thread. The server can create a new thread for each client request, and the thread can handle the request. This allows quick context switching between threads, without the overhead of processes or kernel level threads. This way the server can handle multiple clients concurrently without high overhead of using the operating system.

3. Advantages: Processes assure that the program will continue to run even if one of the processes crashes. They also assure a high level of security, as each process has its own memory space and cannot access the memory of other processes, in this example they assure one tab cannot access the informations of another tab (passwords, credits card and such). Disadvantages: Processes are slower to create and manage than threads, as they require more resources. They also require more memory, as each process has its own memory space. In this example, this can be a problem when a user opens multiple tabs at once as each tab will require more memory and the overall performance of the browser will decrease.

4. There are different kinds of signals and interrupts involved in the process. First, a keyboard interrupt when typing

Then, the kill command is a OS interrupt. When it is executed, it sends a signal to the specified process (Shotwell). 33

34 The operating system manages signals and interrupts at the kernel level to handle this command.

When receiving the kill command, the OS sends a signal to the application's process. Then Shotwell receives the signal 35 36 and starts its shutdown procedure. 37

5. Real time is the actual time that it takes for a thread or a process to run, in the physical hardware. 38 39 On the other hand, virtual time is the time that the thread or process perceives that it takes to run. It depends on the scheduling algorithm and the number of threads or processes running on the system. Virtual time can be different from real time, as the thread or process may be waiting for other threads or processes to finish before it can run. 41

3 Makefile

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

4 thread.h

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

5 thread.cpp

```
#include "thread.h"
1
    #include <iostream>
    #include <setjmp.h>
    #include <signal.h>
4
    #include <unistd.h>
    #include <sys/time.h>
    #include <stdbool.h>
    #ifdef __x86_64__
9
    /* code for 64 bit Intel arch */
10
11
    typedef unsigned long address_t;
12
13
    #define JB_SP 6
    #define JB_PC 7
14
15
    /* A translation is required when using an address of a variable.
16
      Use this as a black box in your code. */
17
18
    address_t translate_address(address_t addr)
19
         address t ret:
20
                            \%fs:0x30,\%0\n"
21
         asm volatile("xor
           "rol $0x11,%0\n"
22
                      : "=g" (ret)
: "0" (addr));
23
24
        return ret:
25
    }
26
27
28
29
    /* code for 32 bit Intel arch */
30
    typedef unsigned int address_t;
31
    #define JB_SP 4
    #define JB_PC 5
33
34
35
    /* A translation is required when using an address of a variable.
36
37
       Use this as a black box in your code. */
    address_t translate_address(address_t addr)
38
39
40
         address_t ret;
        asm volatile("xor
                              %%gs:0x18,%0\n"
41
                     "rol
42
                              $0x9,%0\n"
                : "=g" (ret)
: "0" (addr));
43
44
45
         return ret;
46
    }
47
48
    #endif
49
50
     // Constructor for main
51
    Thread::Thread(int tid)
52
53
             : _tid(tid), _state(READY), _quantum_counter(0)
54
         _stack = new char[STACK_SIZE];
55
    }
57
    Thread::Thread(int tid, void (*entry_point)(void))
```

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

6 uthreads.cpp

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

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

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

```
196
       }
197
        if (threads.size () >= MAX_THREAD_NUM)
198
199
         print_library_error ("thread limit reached");
          resumeTimer ();
200
201
         return -1;
202
        // Create new thread
203
204
        int next_index = indexes.top ();
        indexes.pop ();
205
        Thread *new_thread = new Thread (next_index, entry_point);
206
207
        threads[next_index] = new_thread;
        if (threads[next_index] != nullptr && threads[next_index]->get_state () ==
208
                                               READY)
209
210
        {
         ready_queue.push_back (new_thread);
211
212
213
       resumeTimer ();
214
       return next_index;
     }
215
216
     int uthread_terminate (int tid)
217
218
     {
219
        fflush (stdout):
220
        blockTimer ();
        if (tid == 0 && running_thread->get_tid () == 0) // Terminating main
221
222
223
          ready_queue.clear ();
224
          sleeping_threads.clear ();
225
          for (auto thread = threads.begin (); thread != threads.end ();)
226
            if (thread->first != 0)
227
228
            {
229
              delete thread->second;
              thread = threads.erase (thread);
230
231
            }
232
            else
233
            {
              ++thread;
^{234}
           }
235
         }
236
         exit (0);
237
        }
238
239
        else
240
        {
         if (tid == 0)
241
^{242}
            print_library_error ("cannot terminate main thread");
243
244
            resumeTimer ();
           return -1;
245
246
^{247}
          else if (threads.find (tid) == threads.end ())
248
249
            print_library_error ("in terminate: thread does not exist, tid: " + tid);
250
            resumeTimer ();
           return -1:
251
         7
252
253
          else
254
          {
255
              \it Handle\ termination
            Thread *thread_to_terminate = threads[tid];
256
257
            if (sleeping_threads.find (tid) != sleeping_threads.end ())
258
              sleeping_threads.erase (tid);
259
            }
260
            else if (thread_to_terminate->get_state () == READY &&
261
                     std::find (ready_queue.begin (), ready_queue.end (), thread_to_terminate)
262
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);
             threads.erase (tid);
270
             delete thread_to_terminate;
271
272
             resumeTimer ();
             timer_handler (0, true);
273
274
             return 0;
275
276
           threads.erase (tid);
277
           indexes.push (tid);
278
            delete thread_to_terminate;
           resumeTimer ();
279
280
           return 0;
281
       }
282
     }
283
284
     int uthread_block (int tid)
285
286
       blockTimer ();
287
       if (threads.find (tid) == threads.end ())
288
289
         print_library_error ("in block: thread does not exist, tid: " + tid);
290
291
          resumeTimer ();
292
293
         return -1;
294
       else if (tid == 0)
295
296
297
         print_library_error ("cannot block main thread");
         resumeTimer ();
298
299
300
         return -1;
301
       // Change thread state
302
       if (threads[tid] != nullptr && threads[tid]->get_state () == RUNNING)
303
304
         sigsetjmp (*(threads[tid]->get_env ()), 1);
305
         timer_handler (0, true);
306
307
       if (std::find (ready_queue.begin (), ready_queue.end (), threads[tid]) !=
308
309
           ready_queue.end ())
310
         ready_queue.remove (threads[tid]);
311
312
       threads[tid]->set_state (BLOCKED);
313
       resumeTimer ();
314
315
       return 0;
316
317
     int uthread_resume (int tid)
318
319
320
       blockTimer ();
       if (threads.find (tid) == threads.end ())
321
322
323
         print_library_error ("in resume: thread does not exist, tid: " + tid);
         resumeTimer ();
324
325
         return -1;
326
       else if (threads[tid]->get_state () != BLOCKED)
327
328
         resumeTimer ();
329
         return 0;
330
331
```

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

```
// Return thread quantum counter
int counter = threads[tid]->get_quantum_counter ();
resumeTimer ();
return counter;

404
}
405
406
407
408
409
```