Contents

1 README

```
itamakatz, liavst2
Itamar Katz (555792977) , Liav Steinberg (203630090)
EX: 2

===================================
=             FILES               =
===================================

- README - this file.
- makefile
- uthreads.cpp

===================================
=            ANSWERS              =
===================================


Part 2: Theoretical Questions:
==============================

1.  Kernel level threads, unlike processes, share most of its context
    with the rest of the threads. Due to that, threads are
    fundamentally designed to work together with other threads since
    all threads have complete access to the processes heep, code,
    global variables etc. In other words, unlike processes, threads
    are not independent of one another.
    A tab in a browser could be implemented with a kernel level thread
    but it seems pointless since there is no reason to assume tabs are
    dependent one of the other and its not logical to give each tab
    the ability to access and change info of other threads.
    Besides that, a process tend to have more memory to use since by
    implementing threads, memory is needed for the PCB.

    Kernel level threads and processes have in common that they are
    not blocked when a single process or thread get blocked (unlike
    user level threads) and that they can take advantage of
    multiprocessing.

    Kernel level threads minimize context switching in comparison to
    processes and thus lowers the general overhead. In addition, since
    they share memory, kernel level threads can communication
    considerably fast and efficiently.

    On the other hand, processes tend to manage memory more
    efficiently since there is no need of a PCB for each internal
    thread. And as stated before, even though processes can cope with
    each other, unlike threads they do not have to share memory and
    information, making the process potentially autonomous and
    independent. The latter is important since there is no impact on
    other processes in case of failure, in contrast to threads.


2.  We chose to send the kill signal to "pluma" (with the pid 27469).
    The kill signal is a tool to communicate with other processes
    using a signal. by typing "kill 27469", which by default sends the
    SIGTERM signal to the process, we send the kill signal which by
    definition cannot be handled by a process which then is handled by
    the OS terminating the process.

```

3. Real time, is the physical time as we know it. Virtual time is defined by the ability to manipulate the fixed beginning of time as we wish. It can be stopped, reset, extended and done with whatever we want, since it does not represent the physical world. Real time can be used for example to measure the time a function is executed (like we did in ex1) and virtual time can be used to schedule the run of user-level threads (like we are doing in this exercise).

4a. When using fork, a child process is spawned. The child process is an exact duplicate of its parent process meaning all the data in the global variables, stack and heap is identical at the spawning point, but it is physically not the same memory, stored in different locations, thus not shared, so from that point on, none of the above is common between the processes.

4b. A pipe is an implementation of a communication channel for processes. It contains an array with only 2 slots where one slot is exclusively for writing data and the other is exclusively for reading data. The array is connected to the FDs Table in a way that two processes can communicate.
since processes are mostly autonomous and they do not share memory, the pipe is needed for complex communication between processes other than just signal communication like kill.

2 Makefile

```
CFLAGS = -g -Wall -std=c++11
TAR_NAME = ex2.tar
SOURCES = uthreads.cpp
HEADERS = uthreads.h
OSMLIB = libuthreads.a
OBJS = $(SOURCES:.cpp=.o)
EXTRA_FILES = README Makefile
TAR_FILES = $(SOURCES) $(EXTRA_FILES)

.DEFAULT_GOAL = $(OSMLIB)


all: $(OSMLIB) tar


$(OBJS): $(SOURCES) $(HEADERS)
    $(CXX) -c $(CFLAGS) $<


$(OSMLIB): $(OBJS)
    ar rcs $@ $^

tar:
    tar -cvf $(TAR_NAME) $(TAR_FILES)


clean:
    rm -f $(TAR_NAME) $(OBJS) $(OSMLIB)


.PHONY:
    all tar clean
```

3 uthreads.cpp

```cpp
using namespace std;

#include <cstdlib>
#include "uthreads.h"
#include <signal.h>
#include <sys/time.h>
#include <list>
#include <map>
#include <vector>
#include <queue>
#include <setjmp.h>
#include <iostream>
#include <memory>

#define FAILURE -1
#define SUCCESS 0
#define SYSTEM_FAILURE 1

typedef enum {READY = 1, SLEEP = 2, RUNNING = 3, BLOCKED = 4} State;

#ifdef __x86_64__
/* code for 64 bit Intel arch */

typedef unsigned long address_t;
#define JB_SP 6
#define JB_PC 7

/* 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;
}

#else
/* code for 32 bit Intel arch */

typedef unsigned int address_t;
#define JB_SP 4
#define JB_PC 5

/* 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    %%gs:0x18,%0\n"
        "rol    $0x9,%0\n"
                 : "=g" (ret)
                 : "0" (addr));
    return ret;
}

#endif
```

```
60
61  class Thread;
62
63  // segment for the timer signal
64  struct sigaction g_sa;
65
66  // timer for the slot
67  struct itimerval g_timer;
68
69  int g_current_running_thread;
70
71  // total quantums run in the current process
72  int g_total_quantums = 0;
73
74
75  // env buffer
76  vector<sigjmp_buf> g_thread_env(MAX_THREAD_NUM);
77
78  // stack for each thread
79  vector<char[STACK_SIZE]> g_stack(MAX_THREAD_NUM);
80
81  // for blocking and unblocking the timer signal
82  sigset_t g_mask_set;
83
84  // (tid:thread) map
85  map<int, unique_ptr<Thread> > g_thread_map;
86
87  // queue of unused tids. implemented as a min_heap
88  priority_queue<int, vector<int>, greater<int> >g_unused_indexes;
89
90  // a list of all the thread tids waiting to run
91  list<int> g_ready_threads;
92
93  // sleeping map. each key is the wake up time,
94  // each value is a vector containing the all the thread
95  // tids with that wake up time.
96  map<int, vector<int> > g_sleeping_thread_map;
97
98
99
100 /**
101  * prints an error when a library function fails.
102  */
103 void terror(string msg);
104
105 /**
106  * prints an error when a system call fails, and exits.
107  */
108 void serror(string msg);
109
110 /**
111  * blocks the SIGVTALRM signal
112  */
113 inline void mask_block(){
114     if (sigprocmask(SIG_BLOCK, &g_mask_set, NULL)){
115         serror("sigprocmask failed");
116     };
117 }
118
119 /**
120  * unblocks the SIGVTALRM signal
121  */
122 inline void mask_unblock(){
123     if (sigprocmask(SIG_UNBLOCK, &g_mask_set, NULL)){
124         serror("sigprocmask failed");
125     };
126 }
127
```

```
128 /**
129  * flushes all data structures used, and exits the process.
130  */
131 void general_exit(int exit_type){
132 
133     g_sleeping_thread_map.clear();
134     g_thread_map.clear();
135     g_sleeping_thread_map.clear();
136     g_ready_threads.clear();
137     g_thread_env.clear();
138     g_stack.clear();
139     mask_unblock();
140     if (sigemptyset(&g_mask_set)){
141         serror("sigemptyset failed");
142     }
143     exit(exit_type);
144 }
145 
146 /**
147  * prints an error when a library function fails.
148  */
149 void terror(string msg){
150     cerr << "thread library error: " << msg << endl;
151 }
152 
153 /**
154  * prints an error when a system call fails, and exits.
155  */
156 void serror(string msg){
157     cerr << "system error: " << msg << endl;
158     general_exit(SYSTEM_FAILURE);
159 }
160 
161 
162 
163 class Thread {
164 
165 private:
166     Thread() = delete;
167     State _state;
168     int _tid;
169     int _wakeup_time;
170     int _quantum_count;
171 
172 public:
173 
174     /**
175      * thread constructor.
176      */
177     Thread(int tid):
178         _state(READY),
179         _tid(tid),
180         _wakeup_time(0),
181         _quantum_count(0)
182     {}
183 
184     /**
185      * thread destructor
186      */
187     ~Thread(){}
188 
189     /**
190      * blocks a specific thread.
191      */
192     int block() {
193         if (_state == READY) {
194             _state = BLOCKED;
195             g_ready_threads.remove(_tid);
```

```
196             mask_unblock();
197             return SUCCESS;
198         }
199         else if(_state == RUNNING){
200             _state = BLOCKED;
201 
202             int jump = sigsetjmp(g_thread_env[_tid], 0);
203             if(!jump){
204                 check_sleepers();
205                 next_run();
206                 serror("siglongjmp failed");
207             }
208             return SUCCESS;
209         }
210         mask_unblock();
211         return SUCCESS;
212     }
213 
214     /**
215      * resumes a specific thread from the
216      * blocked state.
217      */
218     void resume() {
219         if (_state == BLOCKED){
220             _state = READY;
221             g_ready_threads.push_back(_tid);
222         }
223         return;
224     }
225 
226     /**
227      * a thread switcher. called every time the running thread
228      * is blocked / goes to sleep / the quanta ended.
229      */
230     static int next_run() {
231 
232         try {
233 
234             int front = g_ready_threads.front();
235             g_ready_threads.pop_front();
236 
237             // next thread
238             g_current_running_thread = front;
239             if (g_thread_map.find(front) != g_thread_map.end()) {
240                 g_thread_map[front]->_quantum_count++;
241                 g_thread_map[front]->_state = RUNNING;
242             }
243             g_total_quantums++;
244 
245 
246             if (setitimer(ITIMER_VIRTUAL, &g_timer, NULL)) {
247                 serror("timer failed");
248             }
249 
250             mask_unblock();
251             siglongjmp(g_thread_env[front], 1);
252         }
253         catch (const out_of_range& eer) {
254             terror(eer.what());
255             general_exit(FAILURE);
256             return FAILURE;
257         }
258     }
259 
260     /**
261      * puts a specific thread to sleep.
262      */
263     void sleep(int sleep_quantums) {
```

```
264         // not including the current quantum
265         if (_state == RUNNING) {
266 
267             // for example went to sleep at in between slots
268             // 100-101 then wake up in slot 108 (100 + 7 + 1)
269             _wakeup_time = g_total_quantums + sleep_quantums + 1;
270             _state = SLEEP;
271             g_sleeping_thread_map[_wakeup_time].push_back(_tid);
272 
273             int jump = sigsetjmp(g_thread_env[_tid], 0);
274             if(!jump){
275                 check_sleepers();
276                 next_run();
277                 serror("siglongjmp failed");
278             }
279         }
280         mask_unblock();
281         return;
282     }
283 
284     /**
285      * returns the wake up time of a specific thread.
286      */
287     int time_to_wake() {
288         if (_state == SLEEP) {
289             return (_wakeup_time - g_total_quantums);
290         }
291         return FAILURE;
292     }
293 
294     /**
295      * returns the quantum count for a specific thread.
296      */
297     inline int get_quantum_count() {
298         return _quantum_count;
299     }
300 
301     /**
302      * the handler for the signal.
303      */
304     static void scheduler(int){
305 
306         mask_block();
307 
308         check_sleepers();
309 
310         g_thread_map[g_current_running_thread]->_state = READY;
311         g_ready_threads.push_back(g_current_running_thread);
312 
313         int jump = sigsetjmp(g_thread_env[g_current_running_thread], 0);
314         if(!jump){
315             next_run();
316             serror("siglongjmp failed");
317         }
318     }
319 
320     /**
321      * goes over the sleeping map and checks who need to wake up.
322      */
323     static void check_sleepers(){
324 
325         if (g_sleeping_thread_map.find(g_total_quantums+1) ==
326             g_sleeping_thread_map.end()){
327             return;
328         }
329 
330         for(vector<int>::iterator it =
331             g_sleeping_thread_map[g_total_quantums+1].begin(); it !=
```

```
            g_sleeping_thread_map[g_total_quantums+1].end(); it++){

            g_thread_map[*it]->_state = READY;
            g_ready_threads.push_back(*it);
        }
        g_sleeping_thread_map.erase(g_total_quantums+1);
    }

};



// ================================================================== //



/*
 * Description: This function initializes the Thread library.
 * You may assume that this function is called before any other Thread library
 * function, and that it is called exactly once. The input to the function is
 * the length of a quantum in micro-seconds. It is an error to call this
 * function with non-positive quantum_usecs.
 * Return value: On success, return 0. On failure, return -1.
*/
int uthread_init(int quantum_usecs){

    if(quantum_usecs > 0){

        // init the unused indexes queue
        for (int i = 0; i < MAX_THREAD_NUM; ++i) {
            g_unused_indexes.push(i);
        }

        int tid = g_unused_indexes.top();
        g_unused_indexes.pop();

        g_thread_map[tid] = unique_ptr<Thread> (new Thread(tid));
        g_ready_threads.push_back(tid);

        // setting function where SIGVTALRM signal goes
        g_sa.sa_handler = &Thread::scheduler;
        g_sa.sa_flags = 0;

        if (sigaction(SIGVTALRM, &g_sa, NULL)) {
            serror("sigaction failed");
        }

        // initiate the time. set to one quantum
        g_timer.it_value.tv_sec = 0;
        g_timer.it_value.tv_usec = quantum_usecs;

        g_timer.it_interval.tv_sec = 0;
        g_timer.it_interval.tv_usec = quantum_usecs;

        if (sigemptyset(&g_mask_set)){
            serror("sigemptyset failed");
        }
        if (sigaddset(&g_mask_set, SIGVTALRM)){
            serror("sigaddset failed");
        }
        if (sigprocmask(SIG_SETMASK, &g_mask_set, NULL)) {
            serror("sigprocmask failed");
        }

        int jump = sigsetjmp(g_thread_env[tid], 0);
        if(!jump){
            Thread::next_run();
            serror("siglongjmp failed");
```

```
400         }
401         return SUCCESS;
402     }
403 
404     terror("non-positive quantum time");
405     return FAILURE;
406 }
407 
408 /*
409  * Description: This function creates a new Thread, whose entry point is the
410  * function f with the signature void f(void). The Thread is added to the end
411  * of the READY threads list. The uthread_spawn function should fail if it
412  * would cause the number of concurrent threads to exceed the limit
413  * (MAX_THREAD_NUM). Each Thread should be allocated with a stack of size
414  * STACK_SIZE bytes.
415  * Return value: On success, return the ID of the created Thread.
416  * On failure, return -1.
417 */
418 int uthread_spawn(void (*f)(void)){
419 
420     mask_block();
421 
422     if (g_unused_indexes.empty()){
423         // no more threads can be made (by the given limit)
424         terror("thread limit exceeded");
425         mask_unblock();
426         return FAILURE;
427     }
428 
429     int tid = g_unused_indexes.top();
430     g_unused_indexes.pop();
431 
432     g_thread_map[tid] = unique_ptr<Thread> (new Thread(tid));
433 
434     sigsetjmp(g_thread_env[tid], 0);
435     (g_thread_env[tid]->__jmpbuf)[JB_SP] = translate_address
436             ((address_t)(g_stack[tid]) + STACK_SIZE - sizeof(address_t));
437     (g_thread_env[tid]->__jmpbuf)[JB_PC] = translate_address((address_t)f);
438 
439     g_ready_threads.push_back(tid); // now it is ready to run
440 
441     mask_unblock();
442     return tid;
443 }
444 
445 /*
446  * Description: This function terminates the Thread with ID tid and deletes
447  * it from all relevant control structures. All the resources allocated by
448  * the library for this Thread should be released. If no Thread with ID tid
449  * exists it is considered as an error. Terminating the main Thread
450  * (tid == 0) will result in the termination of the entire process using
451  * exit(0) [after releasing the assigned library memory].
452  * Return value: The function returns 0 if the Thread was successfully
453  * terminated and -1 otherwise. If a Thread terminates itself or the main
454  * Thread is terminated, the function does not return.
455 */
456 int uthread_terminate(int tid){
457 
458     mask_block();
459 
460     if(tid == 0){
461         general_exit(SUCCESS);
462     }
463 
464     if (g_thread_map.find(tid) == g_thread_map.end()){
465         terror("terminate: thread not found");
466         mask_unblock();
467         return FAILURE;
```

```
468     }
469
470
471     g_thread_map.erase(tid);
472     g_unused_indexes.push(tid);
473     if (tid == g_current_running_thread){ // the thread terminated itself
474         Thread::check_sleepers();
475         Thread::next_run();
476         serror("siglongjmp failed");
477     }
478
479     g_ready_threads.remove(tid);
480     mask_unblock();
481     return SUCCESS;
482
483 }
484
485 /*
486  * Description: This function blocks the Thread with ID tid. The Thread may
487  * be resumed later using uthread_resume. If no Thread with ID tid exists it
488  * is considered as an error. In addition, it is an error to try blocking the
489  * main Thread (tid == 0). If a Thread blocks itself, a scheduling decision
490  * should be made. Blocking a Thread in BLOCKED or SLEEPING states has no
491  * effect and is not considered as an error.
492  * Return value: On success, return 0. On failure, return -1.
493 */
494 int uthread_block(int tid){
495
496     mask_block();
497
498     if(tid == 0){
499         terror("main thread cannot be blocked ");
500         mask_unblock();
501         return FAILURE;
502     }
503
504     if (g_thread_map.find(tid) == g_thread_map.end()) {
505         terror("block: thread not found");
506         mask_unblock();
507         return FAILURE;
508     }
509
510     return g_thread_map[tid]->block();
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 the RUNNING, READY or SLEEPING
517  * state has no effect and is not considered as an error. If no Thread with
518  * ID tid exists it is considered as an error.
519  * Return value: On success, return 0. On failure, return -1.
520 */
521 int uthread_resume(int tid){
522
523     mask_block();
524
525     if (g_thread_map.find(tid) == g_thread_map.end()) {
526         terror("resume: thread not found");
527         mask_unblock();
528         return FAILURE;
529     }
530
531     g_thread_map[tid]->resume();
532     mask_unblock();
533     return SUCCESS;
534
535 }
```

```
536
537 /*
538  * Description: This function puts the RUNNING Thread to sleep for a period
539  * of num_quantums (not including the current quantum) after which it is moved
540  * to the READY state. num_quantums must be a positive number. It is an error
541  * to try to put the main Thread (tid==0) to sleep. Immediately after a Thread
542  * transitions to the SLEEPING state a scheduling decision should be made.
543  * Return value: On success, return 0. On failure, return -1.
544 */
545 int uthread_sleep(int num_quantums) {
546
547     mask_block();
548
549     if (num_quantums <= 0){
550         terror("cannot sleep for negative time");
551         mask_unblock();
552         return FAILURE;
553     }
554
555     if (g_current_running_thread == 0) {
556         terror("main thread cannot sleep");
557         mask_unblock();
558         return FAILURE;
559     }
560     else{
561         g_thread_map[g_current_running_thread]->sleep(num_quantums);
562         return SUCCESS;
563     }
564
565     mask_unblock();
566     return SUCCESS;
567 }
568
569 /*
570  * Description: This function returns the number of quantums until the Thread
571  * with id tid wakes up including the current quantum. If no Thread with ID
572  * tid exists it is considered as an error. If the Thread is not sleeping,
573  * the function should return 0.
574  * Return value: Number of quantums (including current quantum) until wakeup.
575 */
576 int uthread_get_time_until_wakeup(int tid){
577
578     mask_block();
579
580     if (g_thread_map.find(tid) == g_thread_map.end()){
581         terror("wake up: thread not found");
582         mask_unblock();
583         return FAILURE;
584     }
585
586     int return_val  = g_thread_map[tid]->time_to_wake();
587     if(return_val < 0) {
588         mask_unblock();
589         return SUCCESS;
590     }
591     mask_unblock();
592     return return_val;
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     return g_current_running_thread;
602 }
603
```

```
604 /*
605  * Description: This function returns the total number of quantums that were
606  * started since the library was initialized, including the current quantum.
607  * Right after the call to uthread_init, the value should be 1.
608  * Each time a new quantum starts, regardless of the reason, this number
609  * should be increased by 1.
610  * Return value: The total number of quantums.
611 */
612 int uthread_get_total_quantums(){
613     return g_total_quantums;
614 }
615 
616 /*
617  * Description: This function returns the number of quantums the Thread with
618  * ID tid was in RUNNING state. On the first time a Thread runs, the function
619  * should return 1. Every additional quantum that the Thread starts should
620  * increase this value by 1 (so if the Thread with ID tid is in RUNNING state
621  * when this function is called, include also the current quantum). If no
622  * Thread with ID tid exists it is considered as an error.
623  * Return value: On success, return the number of quantums of the Thread with
624  * ID tid. On failure, return -1.
625 */
626 int uthread_get_quantums(int tid){
627 
628     mask_block();
629 
630     if (g_thread_map.find(tid) == g_thread_map.end()){
631         terror("get quantums: thread not found");
632         mask_unblock();
633         return FAILURE;
634     }
635 
636     mask_unblock();
637     return g_thread_map[tid]->get_quantum_count();
638 
639 }
```