Contents

1 Basic Test Results

```
Checking logins...
got ['e342791191', 'archsak']
Make sure this is correct


Checking files...


Testing Makefile...
Passed basic make test


Compile test...
Passed basic compile test


Running basic tests:


Command '['./runtest']' timed out after 9.999947431031615 seconds
FAILED! - TIMEOUT


Command '['./runtest']' timed out after 9.999937523622066 seconds
FAILED! - TIMEOUT




############################
# One or more tests failed #
# see more details above   #
# Pre-submit test failed!  #
############################
```

2 README

```
e342791191, archsak
Elsa Sebagh (342791191), Aharon Saksonov(207600164)
EX: 2

FILES:
thread.cpp
thread.h
uthreads.cpp
makefile

ANSWERS:
1. The sigsetjmp function saves the current state of the thread while the siglongjmp function restores the state
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
to ignore. When calling siglongjmp, the state of the thread is restored to the state saved by sigsetjmp, including the
signal mask.

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
in the command shell.
Then, the kill command is a OS interrupt. When it is executed, it sends a signal to the specified process (Shotwell).
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
and starts its shutdown procedure.

5. Real time is the actual time that it takes for a thread or a process to run, in the physical hardware.
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.
```

3 Makefile

```makefile
# Variables
CXX = g++
AR = ar
CXXFLAGS = -Wall -g -std=c++11
ARFLAGS = rcs
TARGET = libuthreads.a
OBJECTS = uthreads.o thread.o

# Default target
all: $(TARGET)

# Rule to create the static library
$(TARGET): $(OBJECTS)
    $(AR) $(ARFLAGS) $(TARGET) $(OBJECTS)

# Rules to compile the object files
uthreads.o: uthreads.cpp uthreads.h
    $(CXX) $(CXXFLAGS) -c uthreads.cpp -o uthreads.o

thread.o: thread.cpp thread.h
    $(CXX) $(CXXFLAGS) -c thread.cpp -o thread.o

# Clean rule to remove generated files
clean:
    rm -f $(OBJECTS) $(TARGET)

.PHONY: all clean
```

4 thread.h

```
#ifndef THREAD_H
#define THREAD_H

#include "uthreads.h"
#include <setjmp.h>

enum State
{
    READY, RUNNING, BLOCKED
};

class Thread
{
 public:

  Thread (int tid);
  Thread (int tid, void (*entry_point) (void));
  ~Thread ();
  int get_tid ();
  State get_state ();
  thread_entry_point get_entry_point ();
  int get_bound ();
  char *get_stack ();
  int get_quantum_counter ();
  void set_state (State state);
  sigjmp_buf *get_env ();



//quantum counters

 private:
  int _tid;
  void (*_entry_point) (void);
  State _state;
  char *_stack;
  int _bound;
  int _quantum_counter;
  sigjmp_buf _env;

};

#endif //THREAD_H
```

5 thread.cpp

```cpp
#include "thread.h"
#include <iostream>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <stdbool.h>

#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

// Constructor for main
Thread::Thread(int tid)
        : _tid(tid), _state(READY), _quantum_counter(0)
{
    _stack = new char[STACK_SIZE];

}

Thread::Thread(int tid, void (*entry_point)(void))
```

```
        : _tid(tid),
          _entry_point(entry_point), _state(READY), _bound(STACK_SIZE),
          _quantum_counter(0)
{
    _stack = new char[STACK_SIZE];
    address_t sp, pc;
    sp = (address_t) _stack + STACK_SIZE - sizeof(address_t);
    pc = (address_t) _entry_point;
    if (sigsetjmp(_env, 1) == 0) {
        (_env->__jmpbuf)[JB_SP] = translate_address(sp);
        (_env->__jmpbuf)[JB_PC] = translate_address(pc);
        sigemptyset(&_env->__saved_mask);
    } // TODO add throw error?
}

Thread::~Thread()
{
    delete[] _stack;
}

int Thread::get_tid() {return _tid;}

State Thread::get_state() {return _state;}

thread_entry_point Thread::get_entry_point() {return _entry_point;}

int Thread::get_bound() {return _bound;}

char* Thread::get_stack() {return _stack;}

int Thread::get_quantum_counter() {return _quantum_counter;}

void Thread::set_state(State state)
{
    if (state == RUNNING && _state == READY)
    {
        _quantum_counter++;
    }
    _state = state;
}

sigjmp_buf* Thread::get_env() {
  return &_env;
}
```

6 uthreads.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <list>
#include <sys/time.h>
#include <bits/stdc++.h>

#include "thread.h"
#include "uthreads.h"

using namespace std;

int USECS_IN_SEC = 1000000;

int quantum_length;
int quantum_counter;
list<Thread *> ready_queue;
std::map<int, Thread *> threads;
std::map<int, int> sleeping_threads;
Thread *running_thread = nullptr;
priority_queue<int, vector<int>, greater<int> > indexes; // Min heap
struct itimerval timer = {0};
sigset_t signals;
struct sigaction sa = {0};


// Add counter map for blocked threads


void print_system_error (string error)
{
  std::cerr << "system error: " << error << std::endl;
  exit (1);
}

void print_library_error (string error)
{
  std::cerr << "thread library error: " << error << std::endl;
  return;
}

void blockTimer ()
{
  if (sigprocmask (SIG_BLOCK, &signals, NULL) == -1)
  {
    print_system_error ("sigblock error");
    exit (1);
  }
}

void resumeTimer ()
{
  if (sigprocmask (SIG_UNBLOCK, &signals, NULL) == -1)
  {
    print_system_error ("sigunblock error");
    exit (1);
  }
}

```

```
void switch_threads (bool is_terminating = false)
{
  if (ready_queue.empty () && running_thread != nullptr)
  {
    running_thread->set_state (READY);
    running_thread->set_state (RUNNING);
    return;
  };
  std::cout << *(running_thread->get_env ()) << std::endl;
  if (sigsetjmp (*(running_thread->get_env ()), 1) < 0)
  {
    print_system_error ("sigsetjmp error");
    exit (1);
  }
  // Save current thread state
  if (!is_terminating)
  {
    // Move running thread to the ready queue
    running_thread->set_state (READY);
    ready_queue.push_back (running_thread);
  }
  // Get the next thread to run
  running_thread = ready_queue.front ();
  ready_queue.pop_front ();

  // Set the state to RUNNING and increment quantum counter
  running_thread->set_state (RUNNING);

  siglongjmp (*(running_thread->get_env ()), 1) ;

  if (setitimer (ITIMER_VIRTUAL, &timer, nullptr) == -1)
  {
    print_system_error ("setitimer error.");
    exit (1);
  }
}

void timer_handler (int sig, bool isTerminating = false)
{
  // Increment quantum counter and switch threads if needed
  quantum_counter++;
  vector<int> to_awake;
  for (auto thread = sleeping_threads.begin ();
       thread != sleeping_threads.end ();)
  {
    thread->second--;
    if (thread->second == 0)
    {
      to_awake.push_back (thread->first);
      sleeping_threads.erase (thread->first);
    }
  }

  for (int tid: to_awake)
  {
    sleeping_threads.erase (tid);
    if (threads[tid]->get_state () == READY)
    {
      ready_queue.push_back (threads[tid]);
    }
  }
  switch_threads (isTerminating);
}

void timer_handler_no_bool (int sig)
{
  timer_handler (sig, false);
}
```

```
int uthread_init (int quantum_usecs)
{
  if (quantum_usecs <= 0)
  {
    print_library_error ("quantum_usecs must be positive");
    return -1;
  }
  for (int i = 1; i < MAX_THREAD_NUM; i++)
  {
    indexes.push (i);
  }
  quantum_counter = 0;
  quantum_length = quantum_usecs;
  Thread *main = new Thread (0);
  threads[0] = main;
  // Set up timer and signal handler
  sa.sa_handler = &timer_handler_no_bool;
  if (sigemptyset (&sa.sa_mask) == -1)
  {
    print_system_error ("sigemptyset error.");
    exit (1);
  }
  sa.sa_flags = 0;

  if (sigaction (SIGVTALRM, &sa, NULL) < 0)
  {
    print_system_error ("sigaction error.");
  }

  timer.it_value.tv_sec = quantum_usecs / USECS_IN_SEC;
  timer.it_value.tv_usec = quantum_usecs % USECS_IN_SEC;
  timer.it_interval.tv_sec = quantum_usecs / USECS_IN_SEC;
  timer.it_interval.tv_usec = quantum_usecs % USECS_IN_SEC;

  running_thread = main;
  main->set_state (RUNNING);
  // Start a virtual timer. It counts down whenever this process is executing.
  if (setitimer (ITIMER_VIRTUAL, &timer, nullptr))
  {
    print_system_error ("setitimer error.");
    exit (1);
  }
  // create sigset for blocking signals later on
  if (sigemptyset (&signals) == -1)
  {
    print_system_error ("sigemptyset error.");
    exit (1);
  }
  if (sigaddset (&signals, SIGVTALRM) == -1)
  {
    print_system_error ("sigaddset error.");
    exit (1);
  }

// Add signals somehow
  quantum_counter = 1;
  return 0;
}

int uthread_spawn (thread_entry_point entry_point)
{
  blockTimer ();
  if (!entry_point)
  {
    print_library_error ("entry_point must not be null");
    resumeTimer ();
    return -1;
  }
  if (threads.size () >= MAX_THREAD_NUM)
  {
    print_library_error ("thread limit reached");
    resumeTimer ();
    return -1;
  }
  // Create new thread
  int next_index = indexes.top ();
  indexes.pop ();
  Thread *new_thread = new Thread (next_index, entry_point);
  threads[next_index] = new_thread;
  if (threads[next_index] != nullptr && threads[next_index]->get_state () ==
                                        READY)
  {
    ready_queue.push_back (new_thread);
  }
  resumeTimer ();
  return next_index;
}

int uthread_terminate (int tid)
{
  fflush (stdout);
  blockTimer ();
  if (tid == 0 && running_thread->get_tid () == 0) // Terminating main
  {
    ready_queue.clear ();
    sleeping_threads.clear ();
    for (auto thread = threads.begin (); thread != threads.end ();)
    {
      if (thread->first != 0)
      {
        delete thread->second;
        thread = threads.erase (thread);
      }
      else
      {
        ++thread;
      }
    }
    exit (0);
  }
  else
  {
    if (tid == 0)
    {
      print_library_error ("cannot terminate main thread");
      resumeTimer ();
      return -1;
    }
    else if (threads.find (tid) == threads.end ())
    {
      print_library_error ("in terminate: thread does not exist, tid: " + tid);
      resumeTimer ();
      return -1;
    }
    else
    {
//      Handle termination
      Thread *thread_to_terminate = threads[tid];
      if (sleeping_threads.find (tid) != sleeping_threads.end ())
      {
        sleeping_threads.erase (tid);
      }
      else if (thread_to_terminate->get_state () == READY &&
               std::find (ready_queue.begin (), ready_queue.end (), thread_to_terminate)
               != ready_queue.end ())
      {
        ready_queue.remove (thread_to_terminate);
      }
      else if (running_thread != nullptr && tid == running_thread->get_tid ())
      {
        indexes.push (tid);
        threads.erase (tid);
        delete thread_to_terminate;
        resumeTimer ();
        timer_handler (0, true);
        return 0;
      }
      threads.erase (tid);
      indexes.push (tid);
      delete thread_to_terminate;
      resumeTimer ();
      return 0;
    }
  }
}

int uthread_block (int tid)
{
  blockTimer ();
  if (threads.find (tid) == threads.end ())
  {
    print_library_error ("in block: thread does not exist, tid: " + tid);
    resumeTimer ();

    return -1;
  }
  else if (tid == 0)
  {
    print_library_error ("cannot block main thread");
    resumeTimer ();

    return -1;
  }
  // Change thread state
  if (threads[tid] != nullptr && threads[tid]->get_state () == RUNNING)
  {
    sigsetjmp (*(threads[tid]->get_env ()), 1);
    timer_handler (0, true);
  }
  if (std::find (ready_queue.begin (), ready_queue.end (), threads[tid]) !=
      ready_queue.end ())
  {
    ready_queue.remove (threads[tid]);
  }
  threads[tid]->set_state (BLOCKED);
  resumeTimer ();
  return 0;
}

int uthread_resume (int tid)
{
  blockTimer ();
  if (threads.find (tid) == threads.end ())
  {
    print_library_error ("in resume: thread does not exist, tid: " + tid);
    resumeTimer ();
    return -1;
  }
  else if (threads[tid]->get_state () != BLOCKED)
  {
    resumeTimer ();
    return 0;
  }
```

```cpp

//  Change state to READY, add to queue
  threads[tid]->set_state (READY);
//  Add to ready only if it's not sleeping
  if (sleeping_threads.find (tid) == sleeping_threads.end ())
  {
    ready_queue.push_back (threads[tid]);
  }
  resumeTimer ();
  return 0;
}

int uthread_sleep (int num_quantums)
{
  blockTimer ();
  if (num_quantums <= 0)
  {
    print_library_error ("num_quantums must be positive");
    resumeTimer ();
    return -1;
  }
  if (uthread_get_tid () == 0)
  {
    print_library_error ("cannot put main thread to sleep");
    resumeTimer ();
    return -1;
  }
  sleeping_threads[uthread_get_tid ()] = num_quantums;
  if (threads[uthread_get_tid ()] != nullptr &&
      threads[uthread_get_tid ()]->get_state () == RUNNING)
  {
    timer_handler (0, true);
  }
  else if (threads[uthread_get_tid ()] != nullptr &&
           threads[uthread_get_tid ()]->get_state () == READY)
  {
    ready_queue.remove (threads[uthread_get_tid ()]);
  }
  resumeTimer ();

  return 0;
}

int uthread_get_tid ()
{
  blockTimer ();
  int id = running_thread->get_tid ();
  resumeTimer ();
  return id;
}

int uthread_get_total_quantums ()
{
  blockTimer ();
  int count = quantum_counter;
  resumeTimer ();
  return count;
}

int uthread_get_quantums (int tid)
{
  blockTimer ();
  if (threads.find (tid) == threads.end ())
  {
    print_library_error ("in get quantums: thread does not exist, tid " + tid);
    resumeTimer ();
    return -1;
  }
```

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





```