Assignment 1: User Level Threads

Deadline: `Sunday 15th March 23:59`

Logistics

- You can do it alone for all 10 marks or in a group of two for a maximum of 8 marks. This is to encourage good programmers to do it alone and the rest to pair up and still put in a good effort. No request for a group of three will be allowed.
- There will no extension of deadline. The assignment requires some time. Please start early.
- Absolutely no sharing of code is allowed even if it is a not so important helper function. Your code will be matched against other submissions and submissions from prior years. All plagiarism cases will be forwarded to Diciplinary Committee.

Goal

What are user level threads and how they can be implemented without kernel involvement.

Interface to implement

```
void mythread_init();
int mythread_fork();
void mythread_exit();
void mythread_yield();
int mythread_join(int threadid);
```

Details

Implement this assignment on a Linux system. Either a physical linux machine or install a Linux Virtual Machine (using VirtualBox or VMWare).

Define a thread control block (TCB). The TCB contains the threadid, registers, stack, and any other information you may need. You will use the C functions getcontext, setcontext, and makecontext for saving and restoring the actual registers so you do not need to write assembly code. Read their manuals thoroughly.

In `mythread_init` , you make a TCB representing the current thread and any other initialization you may need.

In `mythread_fork` , you create a new thread (new TCB, new stack) with a copy of all registers and stack data and schedule this new thread. The new thread when returning from mythread_fork should receive a return value of 0 while the original thread should receive the threadid as return value.

In `mythread_exit` , the current thread terminates and the next ready thread is scheduled. If the last thread is exiting you call exit to terminate the program.

In `mythread_yield` , you put the current thread back on ready queue and schedule the next thread from ready queue.

In `mythread_join`, you block the current thread until thread identified by threadid terminates, put the current thread on waiting list for that thread and schedule the next thread from ready queue.

Once you have the example below working correctly, then the final step is to use `sigalarm` and implement timer based context switching. Note that it should take less than 10 additional lines of code. Also note that it should be easy to turn off timer based switching. When turned on, the `mythread_yield` function does nothing.

Example Program

```
void MyExampleThread1(int) {
  std::cout << "B" << std::endl;
  mythread_yield();
  std::cout << "E" << std::endl;
  mythread_exit();
}

void MyExampleThread2(int) {
  std::cout << "D" << std::endl;
  mythread_yield();
  std::cout << "H" << std::endl;
  mythread_exit();
}

void main() {
  mythread_init();
  std::cout << "A" << std::endl;
  int thread1 = mythread_fork();
  if (thread1==0)
    MyExampleThread1();
  std::cout << "C" << std::endl;
  int thread2 = mythread_fork();
  if (thread2==0)
    MyExampleThread2();
  std::cout << "F" << std::endl;
  mythread_join(thread1);
  std::cout << "G" << std::endl;
  mythread_join(thread2);
  std::cout << "I" << std::endl;
}
```

The output should be `ABCDEFGHI`. Once you implement timer based switching, the output can be in any order.