In this homework you are given a function `do_switch` which switches the stack pointer on a 32-bit Intel architecture CPU, and another function `setup_stack` which prepares a stack so that `do_switch` can switch to it and begin executing a specified function. Given these two pieces of code, you will write a user-space threading library named 'qthreads' which is sort of compatible with POSIX threads ('pthreads')

NOTE - the code provided works on Linux x86 systems in 32-bit mode. If you want to write the assembler for x86_64, feel free, but get everything working and tested first.

You will have to implement the following types, which should all go in qthread-impl.h (alternately you could get rid of qthread-impl.h and put them in qthread.c):

- struct qthread - you need a structure to represent a thread, with at a minimum a pointer to the allocated stack (so you can free it when the thread is terminated) and a place to save the current stack pointer when switching.
- struct qthread_mutex - a mutual exclusion lock. Note that this is pretty easy, since we're not considering any interrupts.
- struct qthread_cond - a condition variable.

Note that POSIX threads specifies a particularly annoying method of providing options ("attributes") when creating threads, mutexes, or condition variables. We will only be using a single thread attribute, detached state, so `qthread_attr_t` is defined to be an int (e.g. set it to 1 if the attribute is true) and the other attributes types are ignored. If a thread is detached it exits immediately when done; if it isn't detached it waits around until another thread calls `qthread_join` to get its return value.

You will need to write the following functions:

- `qthread_attr_init, qthread_attr_setdetachstate` - initialize an `int` to the default thread attribute state (not detached) and then set detached state on or off.
- `qthread_create` - allocate a thread control block and stack, initialize it so it's ready to call the start function, and put it on the queue of ready-to-run threads. Make sure you note if it has the detached attribute.
- `qthread_mutex_init, qthread_mutex_destroy, qthread_mutex_lock, qthread_mutex_unlock` - mutex operations. Note that the POSIX threads interface requires the caller of mutex_init to allocate the memory for the mutex, which is then initialized by your function.
- `qthread_cond_init, qthread_cond_destroy, qthread_cond_wait, qthread_cond_signal, qthread_cond_broadcast` - condition variable operations.
- `qthread_yield` - yield to the next runnable thread. Doesn't correspond to a normal pthreads operation, but you'll probably find it useful.
- `qthread_usleep, qthread_accept, qthread_read, qthread_write` - these are blocking operating system functions.

The `qthread_usleep` call is the equivalent of the POSIX `usleep` function, which sleeps for a specified number of microseconds. It can be handled by:

- recording when the thread is due to wake up and putting it on a queue of waiting threads.
- from time to time (at every thread switch isn't a bad time) check to see if any waiting timers have expired - if so, make the corresponding threads runnable.

- If all threads are waiting in `qthread_usleep`, do not busy-wait - calculate the amount of time until the first timeout will complete, and use the system `usleep` function to sleep until then.

File-based blocking calls can be handled by:

- setting the file descriptor to non-blocking mode using `fcntl()`
- try the operation - if you get a return of -1 and errno==EAGAIN, it will block, so put the thread on a wait queue somewhere and switch to another one; otherwise return whatever success or error you got back.
- if there are no threads ready to run, then you have to wait for I/O or a qthread_usleep() to expire. Calculate the timeout until the earliest wakeup, figure out all the read and write file descriptors for waiting threads, and use the select() call to wait for I/O on the files or timeout expiration.

See https://www-01.ibm.com/support/knowledgecenter/#!/ssw_ibm_i_71/rzab6/xnonblock.htm for an example of this.

The assignment comes with a makefile which builds four targets:

1. libqthread.a - this is a library of your qthread functions used by the other programs
2. philosopher - this is a solution from another class's homework that simulates the dining philosophers using threads, adapted to use qthreads.
3. test1 - this is compiled from test1.c, which is where you can put a bunch of code to test your qthreads implementation.
4. server - this is a tiny threaded web server, which serves files from the current directory on port 8080. (run 'sh mkindex.sh', then ./server, then point your non-Safari browser at 192.168.56.101:8080/index.html, or whatever the IP address of your VM is)

You are going to need a data structure for your threads, and a way of keeping lists of them. You will probably want to use either a:

- singly-linked list – see http://cslibrary.stanford.edu/103/LinkedListBasics.pdf. Note in particular the drawing on page 19, with head and tail pointers to form a queue. Also you'll need to iterate from the beginning to delete an item.
- doubly-linked list – http://c.learncodethehardway.org/book/ex32.html for an open-ended doubly-linked list; however a circular doubly-linked list is much simpler: https://en.wikipedia.org/wiki/Doubly_linked_list

(I'm a big fan of using absolutely the simplest data structure possible, which is usually an array with linear search, but that doesn't work well for this assignment.)

A hint if you use a doubly-linked list – you may want to embed the list node structure in your struct qthread, and there's no need for the 'value' pointer to be a void*, as it's always going to be a qthread pointer.

Testing

Finally, note that testing your library is an important part of the process of getting it to work. Although it's nice to have your code work with the philosopher and webserver applications, part of your grade will be based on implementing sufficient tests in test1.c or otherwise. Testing will be discussed further in class.