# 2 Exercise #2: Implement a Threading Library

In this exercise you will extend the executor service known from assignment 1 to a fully featured user-space threading library. You will implement a scheduler and basic process management facilities. In particular, the LWT library offers various synchronization mechanisms.

## 2.1 Thread Creation and Termination

Implement a simple scheduler based on a single shared queue, and let this data structure contain all `READY` threads. Scheduling thus becomes a single dequeue operation. Also implement an idle strategy, so that worker threads can wait passively when not enough LWT threads are ready.
The LWT library uses a pthread mutex for internal synchronization, and a pthread condition variable to let worker threads wait passively. Don't worry, we will use nonblocking synchronization in a later exercise.
A user-space context-switch is needed to block a LWT thread, luckily you do not need to implement it yourself. You can find the assembler code in the `pub` directory, and some example snippets that help you understand the mechanics of context switching. The assembler code uses callback functions that you have to implement. The provided examples and documentation will give you further information.
You need to provide a stack to each individual thread. Be sure that the stack is large enough ($\geq$1MiB). If the stack is too small, it will overflow and functions corrupt the data of other threads, causing them to crash. The given context functions store all relevant information on the stack, so you only need to manage a single pointer (i.e., the stack pointer) in your code.
An application can call the `lwt_begin` function, which initializes the library and starts worker threads. Besides, `lwt_begin` creates an initial LWT thread. This function only returns in error cases. On success, it converts the original thread to a worker that executes LWT threads forever. LWT does not support controlled termination of its library.
Besides, write test cases for your library. Let the test-cases run for some time to search for bugs that only occur in unlikely situations. For instance, your test-cases should tackle the following problems:

- Are all submitted threads executed correctly?
- Can threads join other threads regardless of their current state?
- Can the system handle a large number of LWT threads?
- Do thread creation and termination cause memory leaks?
- Does the library crash when no LWT thread is ready?
- Is it possible that `lwt_thread_exit` returns?

The interface documentation is intentionally vague, not all implementation details are specified. If you like, you can change the interface of the library. If you do so, please document *what* you changed and *why*.

## 2.2 Mutex and Condition Variable

When thread creation and termination work as intended, implement the mutex and condition variable interface (`lwt_mutex`, `lwt_condition`). Each of these data structures needs a list to manage the threads that wait for them. Besides, waiting for a mutex or on a condition variable is similar to waiting for thread termination.
Make sure to test these functions thoroughly. For instance, verify that mutexes support nesting.

## 2.3 Barrier

Once you have succeeded with mutexes and condition variables, implement the barrier interface (`lwt_barrier`). Except for the barrier-specific logic, the code should be relatively similar to the mutex and condition variable interface. Also write test cases that use barriers.

### Remarks:

- Thread-local variables can be created by either compiler support using the `__thread` qualifier for variables, or the Pthread functions `pthread_setspecific` and `pthread_getspecific`.
- You can find additional documentation in the provided header files.
- Submission deadline: 28.11.2016, 24:00