2 Exercise #2: Light-Weight Threading Library: LWThreads

Beginning with this exercise you will start to implement a light-weight threading library. This library will be extended in further exercises with features like deadlock detection/prevention, monitors and various kinds of synchronization primitives. Please find the API and documentation in the provided materials[1]. You will also find a C++ implementation of a simple queue data-structure. To generate the documentation, just run "doxygen config.doxy" in the extracted materials-folder.

2.1 Naive Implementation

Make yourself familiar with the provided material. Implement the API functions without any attempts to synchronize critical sections, synchronization will be done later.
Some functions are not obvious:

- `lwt_init()`: Create a thread pool of `Pthreads`, the calling thread should also join this thread pool and any thread starts executing the provided function.
- `lwt_finish()`: Terminates all `Pthreads` and cleans up resources. The main thread will not terminate but return from `lwt_init()`.
- `lwt_signal_*()`: A `lwt_signal` represents a special kind of semaphore. Only one thread can perform a `wait()` operation on such a signal and it will wait until a predefined number of signal operations were performed. `signal()` operations can be carried out in parallel by any number of threads. For this assignment `wait()` should be implemented by busy waiting.

Write a small test that runs a bunch of functions in parallel and waits for them to finish. This test should compile with your library, but will obviously fail to run correctly, because no synchronization was performed.

2.2 Synchronize

Find the critical sections in your code and synchronize appropriately. Write down in a text file (`sync.txt`) where you found critical sections and why those sections are critical. Explain your countermeasures. Most probably you will need to execute pieces of code atomically. Be creative and implement a simple spinlock based on your knowledge of atomic instructions on the x86 platform. Encapsulate the atomic instructions that you use in a separate class (or functions).
If everything works out fine, you should be able to start the execution of many functions that will be executed by (potentially) fewer system-threads in parallel and wait for them to finish.

2.3 Future

In more complex scenarios the light-weight threads may also need to synchronize their own application data. Explain why `Pthread Mutex` objects cannot be used in general for such cases and what the consequences for your library might be in the future.

2.4 Submit

Submit your solutions by creating the directory /proj/i4cs/students/`your_login`/assignment2/ All files in this folder will be collected after the submission deadline. The file `comments.txt` will be created in this directory and contains comments from the tutors. Please create a file `group.txt` with your and your partner's login if you do your assignments in a group of 2 people.

Remarks:

- Feel free to implement this library in C or C++, but note that helping functionality will only be provided in C++ classes.
- You can write atomic operations either in assembly or use the atomic builtins of your compiler. GCC offers different possibilities, the simplest is documented here:

  `https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/_005f_005fsync-Builtins.html`
- Do not change the C-API.

Submit until: 2014-11-18

[1] `https://www4.cs.fau.de/Lehre/current/V_CS/Uebungen/aufgaben/material2.tar.gz`