5 Exercise #5: Non-Blocking Synchronisation

In this exercise you will implement different non-blocking data structures and algorithms. In the end, you have a version of your LWT library that implements the API without locks.

5.1 Non-Blocking Scheduler Queue

In the previous assignments, a lock protected the scheduler's ready-queue. We can eliminate the use of this lock by implementing a multiple-producer multiple-consumer queue. The queue by Michael and Scott[1] is such a non-blocking queue. Remember to handle the ABA problem using version counters attached to all pointers. Test this data structure thoroughly before you integrate it in the LWT library.

5.2 Non-Blocking Thread Management

After changing the ready list to a non-blocking queue, remove all locks from your LWT library. Use atomic operations for all accesses to shared variables. Make sure that every thread in the ready queue is properly initialized and actually ready for execution.
Be careful accessing a thread control block. Keep in mind that threads can concurrently terminate. Besides, when multiple threads write to a shared data structure, they have to synchronize properly.
Use a `lwt_futex` for the `join()` function. If you encounter other situations where threads should wait, re-write that code to also use a `lwt_futex`.
For this part of the assignment, implement `lwt_futex_wait` as a voluntary context switch and immediately enqueue waiting threads to the ready queue. This simplification exploits the possibility of spurious wake-ups. Consequently, the `lwt_futex_wake` functions are no-ops. At this stage, the LWT library should already work as expected.

5.3 Passively Waiting Backend Threads

Use the Linux `SYS_futex` system call to let worker threads wait passively while no LWT thread is ready. Make sure that backend threads wake up as soon as work is available, but also try to avoid unnecessary `FUTEX_WAKE` operations.

5.4 LWT-Futex Implementation

When basic process management works, implement the `lwt_futex` related functions properly. While Linux futexes allow waiting on an arbitrary integer variable, LWT futexes are a dedicated data structure that combines the integer value with a waiting list. Use `PENDING` and `BLOCKED` states, and do not enqueue blocked threads to the ready list. Instead, use a futex-specific waiting list to manage blocked threads.
Make sure that the `lwt_futex_wait` function works correctly even in case of concurrent calls to wake-up functions. Besides, it is possible that multiple wake-up functions run at the same time.

5.5 Optional: Non-Blocking Semaphore Implementation

Feel free to implement a semaphore based on the `lwt_futex` mechanism. Make sure that a signal handler can safely call `V` without blocking. Furthermore, the `V` operation should call `lwt_futex_wake` only when necessary.

Remarks:

- The provided `cntref_t` data structure combines a pointer with a version counter to tackle the ABA problem. You need to compile the program with the `-mcx16` compiler option to enable the 16-byte CAS instruction.
- Exhaustive testing and debugging is recommended to find synchronisation problems you did not think of (... and that make the synchronisation problem harder than you thought).
- You can find wrapper functions to the Linux `SYS_futex` system call in the supplied material.
- Feel free to also benchmark different variants of your library.
- Deadline: 6.2.2017

[1] `https://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html`