2 Exercise #2: Simple Synchronisation in LWT

In this exercise you will extend the LWT library with additional spinlock types and some 'sleeping' synchronisation primitives. You will find a new version of LWT in the materials, as well as papers on memory barriers and ordering as well as different spinlock types.

2.1 Spinlocks

Implement the following locks based on special CPU instructions. To use these special instructions, you can write assembly code or (better) use intrinsic functions of your compiler[1]. In the materials you will already find an implementation of a 'read spinlock'.

- The most simple spinlock that uses a SWAP or CAS instruction to gain exclusive access to a critical section. The lock should perform this atomic swap as long as it does not succeed in a tight loop.
- Ticket spinlocks guarantee starvation-freedom for the threads contending on the lock. A FIFO order of the threads is ensured. Experiment by placing member variables of this lock on different cache lines.
- Ticket spinlocks are well suited for backoff strategies, as each thread can calculate how many threads acquire the lock before it can succeed. Create another version of the ticket spinlock with a backoff strategy of your choice.
- At least one 'scalable lock' of CLH, MCS or K42 type. Papers are included in the materials, but good summary of the algorithms can be found here[2]. Note that the K42 lock has only been published through a US patent.

Write micro-benchmarks to compare the different spinlock types under high and low contention. What happens when there are more threads than CPUs? Why? Come to a conclusion which type of spinlock you want to use in your version of the LWT library.

2.2 Simple Signal

Implement the simple signal type introduced in the blackboard exercise. Please find the interface declarations in `include/lwt-signal.h`. Synchronise critical sections using a spinlock (don't panic, we will also implement this primitive using non-blocking synchronisation later).
A user-space context-switch is needed to block a LWT thread, luckily the scheduler class provides a helper function for blocking operations: `lwt::Scheduler::block(void(*func)(void*), void *arg)`. This function allocates a new context, switches to this new context, prepares the old context to be executed in the future again, calls `func(arg)` and jumps to the scheduling loop, where it picks the next LWT thread to execute. You do not have to understand the details of the context-switch assembly code to implement blocking primitives in LWT.
To test your implementation, rewrite the deadlocking example from the previous assignment to use `lwt_signals` instead of pthread semaphores (hint: it should not deadlock anymore).

2.3 Barrier

Once you have succeeded with the simple signal, implement the barrier interface (`include/lwt-barrier.h`). Except for the barrier-specific logic, the interfacing to `Scheduler::block` should be very similar (easy). Now you should be able to execute the examples from the blackboard exercise with your version of LWT.

Remarks:

- Thread-local variables can be created by either compiler support using `__thread` typed variables, or the Pthread functions `pthread_setspecific` and `pthread_getspecific`.
- You can also measure the performance of `pthread_mutex` in your benchmarks.
- Intel suggests placing a `pause` instruction in busy loops to mark them as such (for the instruction fetch of the CPU pipeline). You can also experiment with this instruction, use: `asm volatile("pause \n\t");`.
- Inline assembly for a full memory barrier on x86: `asm volatile("mfence \n\t":::"memory");`.
- GCC compiler barrier: `asm volatile("":::"memory");`.
- You can create doxygen documentation using the command '`doxygen doxy.conf`' in the LWT directory.

[1] `https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/_005f_005fsync-Builtins.html`
[2] `https://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html`