# 4 Exercise #4: Non-Blocking Synchronisation

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

## 4.1 Allocator Adjustments

Concurrently allocating and freeing objects involves the risk of "use after free" situations, especially in the context of non-blocking algorithms. There are a number of approaches to circumvent these problems like garbage collection, reference counting and hazard pointers. However, these techniques are out of scope for this exercise. As the only objects that are handed around between data structures are `lwt::Thread`s it suffices to just care about the allocation of those objects. You should solve the allocation problem by implementing a different allocation strategy for the `lwt::Alloc` allocator.
Implement a non-blocking stack for the memory management of `lwt::Thread`s. If the stack is empty allocate further `lwt::Thread` objects using `new` or `malloc` and add them to the stack. If the stack is not empty remove the topmost object and return it. Freeing `lwt::Thread`s just adds the object on top of the stack. Do not forget to handle the ABA problem using pointer-counter pairs. It is advisable to encapsulate the access to those pointer-counter pairsi, their elements and operations. This gives the easy ability to exchange the implementation and to play with `DCAS` or mangled pointers and counters.

## 4.2 Non-Blocking Scheduler Queue

As you can see in the `lwt::Scheduler` implementation a single spinlock is used to synchronise accesses to the schedulers ready-queue (`enqueue()` and `dequeue()` operations). We can eliminate the use of this lock by implementing a multiple-enqueuer multiple-dequeuer queue. The queue by Michael and Scott [1] is such a non-blocking queue. Again, remember to handle the ABA problem using counters attached to all pointers.

## 4.3 Non-Blocking Simple-Signal

Invent an algorithm to implement the `lwt_signal` interface in a non-blocking way. First of all, care about the lost-update problem between the `wait` and `signal` operations. Remember that you can use a double-word compare and swap operation. After that, restructure your code to cope with the possibility of multiple CPUs that may execute the same `lwt::Thread` on the same stack at the same time: if a `signal` operation wakes up a thread while this same thread is just about to block itself in the scheduler.

## 4.4 Non-Blocking Semaphore

Implement the `lwt_sem` interface without using locks. This is the hardest task of this exercise. Use the ability to execute code on the next context after the `Scheduler::block()` call. Depending on your implementation, threads may wake up in rare cases even if its not their turn, check that again after waking up.

## 4.5 Other Non-Blocking Structures

After you have implemented the lockless semaphore, think of strategies on how to implement the other blocking primitives. You should see a general pattern there. If you like, feel free to implement all other primitives without locks. What are the software components, that your library depends on, that still use locks?

### Remarks:

- To use `__sync_bool_compare_and_swap` as a double-word compare and swap operation, you have to pass (`unsigned`) `__int128` types to the function and use `-mcx16` as a compilation argument to GCC.
- 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).
- Feel free to also benchmark different variants of your library.
- Of course it is ok to use locks in the case of an error to be able to read the error message.

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