My scalable, fair read-write lock is based on textbook MCS lock implementation which was adjusted in order not to introduce contention in a readers-only setting (by means of an SNZI object) and to account for the existence of two different kinds of nodes in its FIFO queue: readers and writers.

Every reader, before entering a critical section, first checks if there are no nodes in the MCS queue. If not, then it performs an `arrive` operation on the SNZI object of the read-write lock class and enters the CS. If the queue isn't empty, it must append itself at the end and wait for its predecessor to reset that reader's `locked` field (unless the queue became empty right after the test, hence the `if` statement in line 50 of `ReaderWriterLock.h`). While it's waiting for its predecessor, the reader issues a `depart` operation on the SNZI object in order not to block some writers ahead of it, which might be spinning on the SNZI's `query` method. Once its predecessor resets the reader's `lock` field, the reader checks whether he is last in the queue and if so, enters the CS. If some other thread appended itself at the end, the reader waits till that thread has set the reader's `next` pointer. Afterwards, it checks whether it's followed by another reader – if so, it immediately resets its `locked` field as well, so that all the readers in the chain could enter the CS almost simultaneously. They reset each other's fields, so there's no contention on a shared object, although it sometimes may take a while for all the readers to enter the CS, in case there was a long chain of such threads in the queue. In this case, this is a direct trade-off of providing a strong FIFO fairness guarantee.

Upon exiting the CS, readers perform a `depart` operation on the SNZI object and in case some other thread appended itself past the reader, while the reader was executing its CS, the reader resets its successor's `locked` field just like in the original MCS lock.

The writers behave exactly as in the original MCS lock setting, with the added condition of waiting for any readers to exit their CS; that condition is placed at the end of writer's `lock` method, in form of a loop spinning on SNZI object's indicator.

The SNZI object is implemented according to the pseudocode from the paper, with some of the fields being packed into one 64bit variable in order to provide lock-free atomic operations on it. The fields are manipulated by means of bitwise operations; one could probably provide even more efficient SNZI implementation by further fine-tuning those bitwise helper methods. The SNZI object constructs a full binary tree of given depth and all `arrive` and `depart` operations are performed on the leaf nodes whose indices are subject to modulo the number of leaves.

The chart below presents the throughput of the implemented read-write lock, as measured on 24-core Intel Xeon CPU X5680 3.33GHz: