

Use methods Acquire() and Release() of the Lock class, together with Condition variables, to implement the following version of the readers – writers problem:

Any number of readers can read at once, but a writer cannot run together with either a reader, or another writer. Neither readers nor writers get priority – instead, as soon as a writer arrives, later -arriving readers and writers are delayed, until the writer gains access to the object, and finishes. The delayed readers and writers are then run as if they had all just arrived, with their original arrival order preserved. Similarly, a sequence of writers can be executed, one at a time, until a reader arrives. Once a reader arrives, a later -arriving writer waits until all running readers finish. Readers and writers arriving after the delayed writer wait until that writer finishes, and are then executed in arrival order.

Example: R1 R2 R3 W4 R5 W6 R7 R8 W9 arrive, in numerical order. They should be allowed to execute in the order: {R1 R2 R3} W4 R5 W6 {R7 R8} W9. This execution order should be preserved, even if jobs 5..9 all arrive while W4 is running. Job Ri is a reader, Wi is a writer. Jobs in “{ }” run at the same time.

You may assume that cvar.Signal() wakes up that process that has been waiting longest on condition variable cvar, or is ignored, if no process is waiting. Don't use any synchronization primitives other than those mentioned above.

The thread scheduler uses preemptive scheduling.