**Project 3**

**Introduction:**

One of the most well-known concurrency difficulties is the Reader-Writer problem. It refers to a situation in which numerous concurrent threads read and write to a file system or any data. Design and implement a readers/writers lock using semaphores that do not starve the readers and do not starve the writers. The application must be able to prohibit data reading when an overwrite is in progress, as well as prevent reading from interrupting the faulty data under change. To satisfy the following restrictions, we must ensure that readers and writers run separate code before and after critical sections; Firstly, there can be any number of readers in the critical area at the same time. Secondly, writers must have exclusive access to the critical section, that is, no writer can enter when another thread is in CS, and no other thread can enter critical section while the writer is there.

**Code:**

First function activates the reader lock when the input thread is “r” to perform the critical section and after it’s done, the reader lock is released so that the next thread can enter the critical section. Second function activates the writer lock when the input thread is “w” to perform the critical section and after it’s done, the writer lock is released so that the next thread can enter the critical section.

Text

Description automatically generated

The code for the reader and writer helper functions:

Text

Description automatically generated

**Pseudocode:**

Semaphore Initialization:

write\_lock = Semaphore(1);

mutex = Semaphore(1);

starve = Semaphore(1);

int Count = 0;

acquire\_readlock():

wait(starve);

wait(mutex);

Count++;

if Count == 1

wait(write\_lock)

post(mutex)

post(starve)

release\_readlock():

wait(mutex)

Count --

if Count== 0

post(write\_lock)

post(mutex)

acquire\_writelock():

wait(starve)

wait(write\_lock)

release\_writelock():

post(write\_lock);

post(starve);

**Solution:**

It is critical to initialize the variables that will be accountable for implementation right away. A semaphore is built with the current number of readers in the critical part as the "count" variable. The count variable is protected by the semaphore sem\_t mutex, which prevents other threads from changing it. The second semaphore, sem\_t “write\_lock”, prevents other writers from entering the execution while someone is already inside. As a turnslite, the third semaphore sem\_t "starve" is employed. When the writer thread enters, it calls sem\_wait on the starve so that when the reader arrives, it puts itself to sleep, preventing the writer from being overwhelmed by readers and starve. When the writer quits, it knows that no more threads will be added to the execution because it was the sole one in the first place. The code for readers comprises keeping track of the number of readers in the room and incrementing and decrementing it when they enter and exit. The first person to join the room and the first person to leave must both call sem\_wait and sem\_post for write\_lock in order for the writer to know when to enter the room. When the reader enters the execution, the writer is prevented from entering the execution. When a reader enters a room that is already occupied by a writer, it will wait, just as the second writer would, and a queue will form. The third semaphore functions as a turnslight. It's there for the readers, and it allows writers to lock and unlock it after they're finished, allowing queued readers to execute. If a writer arrives with readers in tow, it will lock the turnstile first, then go to sleep because the room isn't empty. The next reader who comes in will be put to sleep right away because the turnstile is already locked, allowing the writer to enter the room first. Because turnslite was closed after the writer and no readers can now enter, the writer will enter immediately once all readers have exited the execution. The writer performs sem post on starve and unblocks the following thread, which is either a reader or a writer, after execution. However, it now allows any thread (writer or reader) to participate in the execution as shown below:

Text

Description automatically generated

**Conclusion:**

The code is simple as it is similar to the pseudocode that was given above. However, there are some of the disadvantages I perceive with this approach. Firstly, the reader's Critical Section must be safeguarded by two semaphores at the same time. If we assume that the Critical Section is fast and the Wait() is slow, having only one mutex to help us may be more beneficial. Secondly, if the writer is leaving the Critical Section, it will call Post() on the starve, which is our hunger prevention method. If there are other authors and readers scheduled, however, nothing guarantees that if the next writer was intended to come before the readers, it will really happen. It may or may not have to wait for other readers in the end, depending on the evil scheduler.