Reader Writer Problem

Pseudo code

```
class ReaderWriter
    static Semaphore read_lock := Semaphore(1, true)
    static Semaphore read_write_lock := Semaphore(1, true)
    static reader_turn := true
    static read_count := 0
end class

class Writer
    void run()
        if ReadreWriter.read_count > 0 Then
            ReadreWriter.reader_turn := false
        wait the release of read_write_lock and acquire it
        ReaderWriter.read_turn := true

        // writing is performed

        Release read_write_lock
    return
end class

class Reader
    void run()
        wait until reader_turn become true
        wait the release of read_lock and acquire it
        ReaderWriter.reader_count += 1
        if ReaderWriter.reader_count = 1 Then
            wait the Release of read_write_lock and acquire it
        Release read_lock

        // Reading is performed

        wait the release of read_lock and acquire it
        ReaderWriter.count -= 1
        if ReaderWriter.read_count = 0 Then
            release the read_write_lock
    return
end class
```

Example of real world problem :

A popular real-world problem is when multiple (reader / writer) processes want to access the same file, and we don't want to enter readers and writers at the same time to ensure that no readers read inconsistent data before the writer finishes its task, and we also don't want to enter two writers at the same time to ensure that a race condition doesn't occur.

Another real world Example :

Another popular real-world example is a ticket reservation. Many users wish to book a ticket, so the system reads the number of available tickets and displays this number to them. The system may also write to reduce the amount of available tickets when a reservation occurs so there are many readers and writers processes.

Explanation of the solution

We solve the Reader Writer problem using two Semaphores one for read operation so we can protect the reading count variable from Race condition and another for read write operation to make sure no reader and writer will access the critical section at the same time and no two writers will access the critical section at the same time also.
We used a continue reading variable that tells the system should it continue entering the read processes or it should stop them to leave a space for a write process to enter so no starvation could occur.

Continue reading mechanism :

when there is a writer want to enter we set this continue reading boolean variable to false so it prevents any new reader from entering the critical section and when the readers that already was in the critical section finish and release the lock the writer process will perform its task & allowing the readers to continue entering the critical section.

Setting Semaphore fairness to true :

we set the fairness of read lock to true so if two processes want to read l will enter the process that first come than the other process also when a reader and a writer want to acquire the lock l will give the lock to who is coming first

Deadlock Example :
If two acquires are called in the same thread before calling release, a deadlock may occur.

How we solved that :
We ensure that no other acquire is called before calling the release, so we call the release first, followed by the other acquire (if we are in the same thread).

Another example of deadlock :
If writer process didn't release the lock that will make other readers and writers not able to acquire the lock and a deadlock will happen

How we solved that :
We ensure that we release the lock, by calling the release once the writer finishes its job.

starvation example:
A common example of starvation is when readers enter the critical section and continue entering because they have the key, and because many readers processes can be in the critical section at the same time, causing the writer to starve outside; on the other hand, if we prioritize the writer so that no more readers enter when the writer exists and wants to enter the critical section, it may cause a starvation for readers.

How we solve that:
We solve the Reader Writer problem using two Semaphores one for read operation so we can protect the reading count variable from Race condition and another for read

write operation to make sure no reader and writer will access the critical section at the same time and no two writers will access the critical section at the same time also.
We used a continue reading variable that tells the system should it continue entering the read processes or it should stop them to leave a space for a write process to enter so no starvation could occur.

Continue reading mechanism :

when there is a writer want to enter we set this continue reading boolean variable to false so it prevents any new reader from entering the critical section and when the readers that already was in the critical section finish and release the lock the writer process will perform its task & allowing the readers to continue entering the critical section.

Setting Semaphore fairness to true :

we set the fairness of read lock to true so if two processes want to read l will enter the process that first come and also we set the fairness of reader writer lock to True so when a reader and a writer want to acquire the lock l will give the lock to who is coming first

Another example of starvation :

A starvation may happen if we make the reader and writer work in turns and one of them finishes before the other.

How we solved that :

We solved that by making sure that no process will hang if the other process finishes.
If the process terminates, it will have no effect on other processes, and the terminated process also doesn't have any variables that could stop other processes from execution.