# ReadWrite Lock

We discuss a common interview question involving synchronization of multiple reader threads and a single writer thread.

## Problem Statement

Imagine you have an application where you have multiple readers and multiple writers. You are asked to design a lock which lets multiple readers read at the same time, but only one writer write at a time.

## Solution

First of all let us define the APIs our class will expose. We'll need two for writer and two for reader. These are:

- acquireReadLock
- releaseReadLock
- acquireWriteLock
- releaseWriteLock

This problem becomes simple if you think about each case:

1. Before we allow a reader to enter the critical section, we need to make sure that there's no **writer** in progress. It is ok to have other readers in the critical section since they aren't making any modifications
2. Before we allow a writer to enter the critical section, we need to make sure that there's **no reader or writer** in the critical section.

```
public class ReadWriteLock {

    public synchronized void acquireReadLock() throws InterruptedException {
    }

    public synchronized void releaseReadLock() {
    }

    public synchronized void acquireWriteLock() throws InterruptedException {
    }

    public synchronized void releaseWriteLock() {
    }
}
```

Note that all the methods are synchronized on the ReadWriteLock object itself.

Let's start with the reader use case. We can have multiple readers acquire the read lock and to keep track of all of them; we'll need a count. We increment this count whenever a reader acquires a read lock and decrement it whenever a reader releases it.

Releasing the read lock is easy but before we acquire the read lock, we need to be sure that no other writer is currently writing. Again, we'll need some variable to keep track of whether a writer is writing. Since only a single writer can write at a given point in time, we can just keep a boolean variable to denote if the write lock is acquired or not. Let's translate what we have discussed so far into code.

```
public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {

        while (isWriteLocked) {
            wait();
        }

        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
    }

    public synchronized void acquireWriteLock() throws InterruptedException {
    }

    public synchronized void releaseWriteLock() {
    }
}
```

Note how we are checking in a loop whether `isWriteLocked` is true and then calling `wait()`. Also pay attention to the fact that the methods are synchronized so only one reader will be able to decrement reader in releaseReadLock.

For the writer case, releasing the lock would be as simple as setting the `isWriteLocked` variable to false but don't forget to call `notify()` too since there might be readers waiting in the `acquireReadLock()` method.

Acquiring the write lock is a little tricky, we have to check two things whether any other writer has already set `isWriteLocked` to true and also if any reader has incremented the `readers` variable. If `isWriteLocked` equals false and no reader is writing then the writer should proceed forward.

```
public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {

        while (isWriteLocked) {
            wait();
        }

        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
    }

    public synchronized void acquireWriteLock() throws InterruptedException {

        while (isWriteLocked || readers != 0) {
            wait();
        }

        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}
```

The astute reader will notice a bug in the code we have so far. Try finding it, before reading ahead !

For the impatient, note that in our `acquireWriteLock()` method we have a while loop which has `readers != 0` condition. We should remember to call `notify()` whenever any snippet of code can change the value of the `readers` variable and make the loop condition in `acquireWriteLock()` false. If we don't call notify then any thread waiting in the loop will never be woken up.

Within the `releaseReadLock()` method, we should call `notify()` after decrementing readers to make sure that any blocked readers should be able to proceed forward.

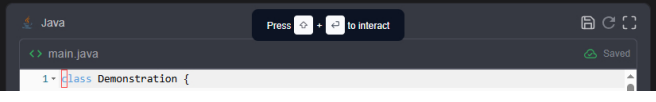
```
public class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {

        while (isWriteLocked) {
            wait();
        }

        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
        notify();
    }

    public synchronized void acquireWriteLock() throws InterruptedException {

        while (isWriteLocked || readers != 0) {
            wait();
        }

        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}
```

Note that with the given implementation, it is possible for a writer to starve and never get a chance to acquire the write lock since there could always be at least one reader which has the read lock acquired.

## Complete Code

The complete code with a test-case appears below. Run the code and examine the output messages. We start a reader and a writer thread initially. The writer blocks until the read lock is released. Also, we release the reader-lock through another reader thread.

A second writer thread is blocked forever since the first writer thread never releases the write-lock. The execution eventually times out.




The write-lock is acquired only after the read-lock is released. If you look at the output, the write lock acquisition timestamp would be between the timestamps of the statements **"read lock about to release"** and **"read lock released".** This is so because timestamps aren't granular enough and read-lock's release timestamp and write-lock's acquisition timestamp might be same.

Also - the read-lock's release statement might get printed after the write-lock's acquisition statement but that is possible if the thread tReader2 gets context-switched as soon as it releases the lock and before it gets a chance to execute the print statement.

Last but not the least, running the above test in the browser would show execution timing out. This is expected as our t1 thread is modelled as a writer thread that never releases the write-lock.


← Back lesson

Implementing Semaphore

Mark As Completed Next →

Unisex Bathroom Problem