ReadWriteLock

This lesson examines the ReadWriteLock interface and the implementing class ReentrantReadWriteLock. The lock is intended to allow multiple readers to read at a time but only allow a single writer to write.

We'll cover the following

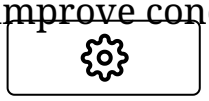
If you are interviewing, consider buying our number#1 course for Java Multithreading Interviews (https://bit.ly/2QfKXCK).

The `ReadWriteLock` interface is part of Java's `java.util.concurrent.locks` package. The only implementing class for the interface is `ReentrantReadWriteLock`. The `ReentrantReadWriteLock` can be locked by multiple readers at the same time while writer threads have to wait. Conversely, the `ReentrantReadWriteLock` can be locked by a single writer thread at a time and other writer or reader threads have to wait for the lock to be free.

ReentrantReadWriteLock#

The `ReentrantReadWriteLock` as the name implies allows threads to recursively acquire the lock. Internally, there are two locks to guard for read

and write accesses. `ReentrantReadWriteLock` can help improve concurrency



over using a mutual exclusion lock as it allows multiple reader threads to read concurrently. However, whether an application will truly realize concurrency improvements depends on other factors such as:

- Running on multiprocessor machines.
- Frequency of reads and writes. Generally, `ReadWriteLock` can improve concurrency in scenarios where read operations occur frequently and write operations are infrequent. If write operations happen often then most of the time is spent with the lock acting as a mutual exclusion lock.
- Contention for data, i.e. the number of threads that try to read or write at the same time.
- Duration of the read and write operations. If read operations are very short then the overhead of locking `ReadWriteLock` versus a mutual exclusion lock can be higher.

In practice, you'll need to evaluate the access patterns to the shared data in your application to determine the suitability of using the `ReadWriteLock`.

Fair Mode#

The `ReentrantReadWriteLock` can also be operated in the *fair mode*, which grants entry to threads in an approximate arrival order. The longest waiting writer thread or a group of longest waiting reader threads is given preference to acquire the lock when it becomes free. In case of reader threads we consider a group since multiple reader threads can acquire the lock concurrently.

Cache Example#

One common scenario where there can be multiple readers and writers is that of a cache. A cache is usually used to speed up read requests from another data source e.g. data from hard disk is cached in memory so that a request doesn't have to wait for data to be fetched from the hard disk thus saving I/O. Usually, there are multiple readers trying to read from the cache and it is imperative that the readers don't step over writers or vice versa.

In the simple case we can relax the condition that readers are ok to read stale data from the cache. We can imagine that a single writer thread periodically writes to the cache and readers don't mind if the data gets stale before the next update by the writer thread. In this scenario, the only caution to exercise is to make sure no readers are reading the cache when a writer is in the process of writing to the cache.

In the example program below we use a `HashMap` as a cache and input a single key/value pair that is periodically updated by the writer thread. From the output of the program, note that the reader threads find the value of the key updated after the acquisition of the lock by the writer thread.

```
1  import java.util.HashMap;
2  import java.util.Random;
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.Future;
6  import java.util.concurrent.locks.ReadWriteLock;
7  import java.util.concurrent.locks.ReentrantReadWriteLock;
8
9  class Demonstration {
10
11     static Random random = new Random();
12
13     public static void main( String args[] ) throws Exception {
14
15         ExecutorService es = Executors.newFixedThreadPool(15);
16
17         // cache
18         HashMap<String, Object> cache = new HashMap<>();
```

```
        HashMap<String, Object> cache = new HashMap<>();
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

        // put some data in the cache
        cache.put("key", -1);

        Runnable writerTask = new Runnable() {
            @Override
            public void run() {
                writerThread(cache, lock);
            }
```

Downgrading to Read Lock#

To showcase downgrading from a write lock to a read lock, we'll slightly modify our scenario to include a challenging requirement that readers can't tolerate stale data. We'll assume that readers have the ability to trigger an update of the cache data if it is found to be stale. A boolean flag `isDataFresh` depicts whether data in the cache is fresh or not. We'll make changes to our previous program to accommodate for these new requirements. The writer thread will now occasionally set the flag `isDataFresh` to false to indicate that the reader threads must trigger an update. The writer thread from our previous program doesn't write data to the cache anymore rather it simply sets the flag `isDataFresh` to false to force the reader threads to trigger an update.

The astute reader will realize this setup implies that a reader thread must acquire the write lock upon discovering the data is stale and initiate an update. Thus the reader thread works with both the read and the write locks. The reader thread upon finding the data is stale acquires the write lock, triggers a cache refresh, and then downgrades to the read lock to continue reading data. Note that the vice versa, i.e. upgrading a read lock to a write lock isn't allowed.

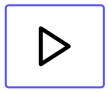
The comments in the program below explain the program flow and the

downgrading of write lock.


```
import java.util.HashMap;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class Demonstration {

    static Random random = new Random();
    static boolean isDataFresh = true;

    public static void main( String args[] ) throws Exception {
        ExecutorService es = Executors.newFixedThreadPool(15);

        // cache
        HashMap<String, Object> cache = new HashMap<>();
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

        // put some data in the cache
        cache.put("key", -1);

        Runnable writerTask = new Runnable() {
            @Override
            public void run() {
                writerThread(lock);
            }
        };

        Runnable readerTask = new Runnable() {
            @Override
            public void run() {
                readerThread(cache, lock);
            }
        };

        try {
            Future future1 = es.submit(writerTask);
            Future future2 = es.submit(readerTask);
            Future future3 = es.submit(readerTask);
            Future future4 = es.submit(readerTask);

            future1.get();
            future2.get();
            future3.get();
            future4.get();
        } finally {
            es.shutdown();
        }
    }
```

```
static void writerThread(ReadWriteLock lock) {


    for (int i = 0; i < 9; i++) {
        try {
            Thread.sleep(random.nextInt(50));
        } catch (InterruptedException ie) {
            // ignore
        }

        lock.writeLock().lock();
        System.out.println("Acquired write lock");
        isDataFresh = false;
        lock.writeLock().unlock();
    }
}

static void updateData(HashMap<String, Object> cache) {
    cache.put("key", random.nextInt(1000));
    isDataFresh = true;
}

static void readerThread(HashMap<String, Object> cache, ReadWriteLock lock) {

    for (int i = 0; i < 3; i++) {
        try {
            Thread.sleep(random.nextInt(50));
        } catch (InterruptedException ie) {
            // ignore
        }

        // acquire the read lock to check if data is fresh before
        // reading from the cache
        lock.readLock().lock();

        try {
            // check if the data is fresh
            if (!isDataFresh) {

                // release the read lock, before acquiring the write lock
                lock.readLock().unlock();

                // acquire the write lock before triggering an update
                lock.writeLock().lock();

                try {

                    // Check the flag again, the data might already have been r
                    // another writer thread.
                    if (!isDataFresh) {
                        updateData(cache);
                    }

                    // acquire read lock before releasing the write lock. This
```

```
                    // acquire read lock before releasing the write lock. This
                    // example of downgrading from write -> read lock
                    lock.readLock().lock();

                } finally {
                    lock.writeLock().unlock();
                }
            }

            System.out.println("Acquire read lock and reading key = " + cache.g

        } finally {
            lock.readLock().unlock();
        }
    }
}
```




The output of the above program is similar to the output of the previous program. Note, the acquisition of the write lock each time triggers a refresh of the `cache` as depicted by a change the key value read by the reader threads.

Reentrancy#

Finally, remember that the `ReentrantReadWriteLock` allows thread to recursively acquire the read or write lock. You must remember to unlock as many times as you lock. The sequence of locks in the snippet below will result in a deadlock since the write lock is acquired twice but released only once before attempting to acquire the read lock.

```
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock
();
        lock.writeLock().lock();
        lock.writeLock().lock();
        lock.writeLock().unlock();
        lock.readLock().lock();
```

Q Consider the code snippet below:

```
        ReentrantReadWriteLock lock = new ReentrantReadW
riteLock();
        lock.readLock().lock();
        lock.writeLock().lock();
```

What will be the outcome when a thread executes this snippet:

A) The thread will block indefinitely because read lock can’t be upgraded to write lock.

B) The thread will throw an exception because read lock can’t be upgraded to a write lock.

C) The setup is allowed and the thread will acquire both read and write locks.

Submit Answer

Reset Quiz

Mark as Completed