CS601-01/CS601-02: Lab3 Part 1

ReentrantReadWriteLock. Thread-Safe Hotel Data. HotelDataBuilder using concurrent package.

Due Date: Oct 2, 11:59pm

The goal of Lab 3 is to get experience writing multithreaded programs.
For **part 1** of the lab, you will implement your own reentrant multi-reader lock and use it to make ThreadSafeHotelData (a child class of your HotelData class from lab 1) thread-safe. You will also write HotelDataBuilder class, that can load reviews into ThreadSafeHotelData *concurrently*. For HotelDataBuilder, for this part of the lab, you will use ExecutorService and Executors from built-in Java package **java.util.concurrent.**

For this lab, you may **not** use any of the classes from `java.util.concurrent.locks.`

1. ReentrantReadWriteLock

You need to implement a *reentrant* read-write lock. Before we explain the requirements, let us introduce some terms:

- A ***reader*** is a thread holding the read lock (this thread "reads" from the shared data, does not modify it).
- A ***writer*** is a thread holding the write lock (this thread "writes" to the shared data)

The ReentrantReadWriteLock you will write should adhere to the following specifications:

- Multiple threads may become *readers* as long as there are no *writers*
- Only one thread may become a *writer* at any given time (if there are no other writers and no readers).
- A *writer* can also become a *reader* (a thread holding a write lock may also acquire the read lock).
    - The opposite is not true (a reader can **not** upgrade to a writer while holding the read lock).
- A thread only releases one lock at a time, therefore a thread may acquire a write lock then acquire a read lock. If this thread releases the write lock it will still hold the read lock.

- The lock allows both readers and writers to **reacquire read or write locks while holding the corresponding lock.** So the same thread can acquire several read locks (as long as there are no writers). While a thread is holding a write lock, it can acquire another write lock on the same object.

- The `tryAcquiringReadLock()` and `tryAcquiringWriteLock()` methods will *not* block (not calls to wait()), but rather check conditions for acquiring the lock, and if they are true, will modify info about readers or writers, and return true if the lock was acquired and false otherwise.
- The lock methods (`lockRead` and `lockWrite`) will block (call the wait() method) until the lock is acquired. For instance, lockRead method should call `tryAcquiringReadLock()` in a while loop, and if it returns false, call wait().

Use the provided test file, `ReentrantReadWriteLockTest`, to test your custom lock. You need to test your lock independently as well, the provided tests are limited.

This lock is similar to Java's built-in ReentrantReadWriteLock (I recommend you open the Java API and read its description carefully), but you may **not** use Java's built-in locks for this lab. By implementing the ReentrantReadWriteLock yourself, you will get a better understanding of how it works. Note that our custom lock has no fairness policy.

2. Thread-Safe Hotel Data

Your second task for the first part of Lab 3 is to write a class `ThreadSafeHotelData` that extends your `HotelData` class from lab 1 and makes it **thread-safe** using the ReentrantReadWriteLock described above. You should also move `loadHotelInfo` and `loadReviews` method from `HotelData` class into a different class, `HotelDataBuilder` (see the next section).

To make `ThreadSafeHotelData` thread-safe:

- Store an instance variable of class `ReentrantReadWriteLock` in `ThreadSafeHotelData.`
- Allow concurrent reads of the data structures in `ThreadSafeHotelData`. Surround the blocks of code where you read from the data structures with `lockRead/unlockRead.`
- Disallow concurrent writes and concurrent read/writes of the data structures of `ThreadSafeHotelData.` Surround the blocks of code where the data structures are modified by `lockWrite/unlockWrite.`
- Make sure you unlock in the `finally()` block.

3. HotelDataBuilder

Your third task for part 1 of the lab is to write a new class, `HotelDataBuilder` that loads hotel data into ThreadSafeHotelData from json files and processes each review file concurrently. For part 1 of lab 3, `HotelDataBuilder` should contain the following:

- an instance variable of type `ThreadSafeHotelData`
- two constructors:

```
    public HotelDataBuilder(ThreadSafeHotelData data)
    public HotelDataBuilder(ThreadSafeHotelData data, int numThreads)
```

Inside the constructors, assign a value to your instance variable of type `ThreadSafeHotelData` and use ExecutorService and Executors from java.util.concurrent package to create a pool of threads (1 in the first case, and `numThreads` in the second case).

- the method `loadHotelInfo` that reads a single json file with general hotel information and adds this info into ThreadSafeHotelData. (Note that you should **move** this method from `HotelData` (or `ThreadSafeHotelData`) to `HotelDataBuilder`). This method is **not** going to be multithreaded.

- `loadReviews(Path dir)` that recursively traverses a given directory with reviews and ***concurrently*** processes all JSON files found, adding reviews to the `ThreadSafeHotelData`. You need to create **a new Runnable job for each json review file,** that will parse the file, create Reviews and add them to ThreadSafeHotelData. **Hint**: you need to create an inner class in HotelDataBuilder that implements Runnable and that processes a single Json file in the run() method. After you create an instance of this Runnable task for each json file with reviews, "submit" it to the ExecutorService variable, so that one of the threads from the pool of threads will start executing it.
  Note: in lab1, loadReviews method was in `HotelData` class, now you need to move it to HotelDataBuilder and make it multi-threaded.
  Note that your input folder for lab 3 contains **reviewsLargeSet** subfolder which contains larger review files for each hotel.

- `printToFile(String filename)` that waits until all the runnable tasks have finished executing (using methods shutdown() and awaitTermination() of ExecutorService), and calls printToFile method of class `ThreadSafeHotelData`.
-
- An inner class that represents a Runnable task of parsing a single json review file (mentioned in loadReviews - see above).
- Other variables and methods as needed (such as Logger et.)

You are required to use Executors and ExecutorService for this class, but are **not** allowed to use any other classes from the built-in concurrent package and not allowed to use third party libraries (apart from JsonSimple, log4j2 and jUnit). If there is a class you would like to use, ask the instructor first.

You are required to use Logger (log4j2) for debugging your code. The corresponding jars are in the **lib** folder of your starter code.

Your code should pass all tests in:
- **HotelBuilderTest**
- **ReentrantReadWriteLockTest**

for this part of the lab.

The WorkQueue class that has been provided with the starter code, is not used in this part of the lab, but do not delete it - you will need it for part 2.

Javadoc Comments

You are required to add javadoc comments to your code (above each class and above each public method). Please also add some comments within the methods, where the logic is not obvious (such as in `tryAcquiring...` and `lock...` methods in `ReentrantReadWriteLock`).

Submission: Your lab3 part 1 should be submitted to your private username-lab3 repo. Please keep pushing your code to github as you work on the lab. I need to see at least 5 meaningful commits in github, otherwise we will assign a 0 for the lab.

The instructor provides several tests for the lab, but you are responsible for doing your own thorough testing before submitting the lab (I recommend writing your own tests for that).