CS 251 2020 Outlab 8C - Java

C for Concurrency

Our Outlabs probably don't deserve to be named like iPhones, because they're patched and updated more often than Androids. Watch out for updates in this colour. This exercise is entirely graded, and will contribute 50% to your Outlab-8 score. The other half will come from the graded Basic Java tasks which have been released earlier. To avoid confusion, **basic Java** and **concurrency Java** exercises are **two different submissions** to be made from the Moodle account of the teammate with the lexicographically smallest roll number.

The Complete Package (50 points)

In this exercise, we will explore concurrency in Java by incrementally building a rudimentary reader-writer application. At each step, you will write a class, and these classes will be compiled into a package called `readwrite`. The first line in each of your source files should be:

```
package readwrite;
```

Within a package, you don't need to explicitly import anything to instantiate an object or invoke a method of another class. To quote the excellent Oracle Java Documentation: "To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages."

In your submission directory, create a new directory called `readwrite`. All your java source code files go here.

We will provide you `FinalTester.java`, `w1.txt`, `w2.txt`, `r1.txt`, `r2.txt` to test the correctness of the synchronisation. Put the java file in the package folder, and keep the text files in the outer submission directory. You can also write your own java test files, but you needn't submit them.

It's best to compile and run the files from outside the package directory.

```
# javac readwrite/*.java
```

Suppose you want to execute the main method which was in the file `Foo.java`.

```
# java readwrite.Foo
```

Please follow the naming conventions for classes and methods strictly in order to ensure smooth evaluation.

Part A: The Underlying Data Structure (15 points)

Implement the class `Tree` in, you guessed it, `Tree.java`. This is a binary search tree that holds integer entries. With the DSA exposure you've had, this is now bread and butter. While Java is very close to C++ syntactically, a key conceptual difference is that Java uses **reference based semantics** in comparison to C++ value based semantics with supplementary pointers on top. In Java, you're implicitly playing with pointers all the time, and this makes implementing a BST in Java a lot more fun. Don't be lazy, try doing it yourself first! :)

By convention, the tree should be initialised to a single node with nothing written to it. The method `public void write(int)` will write a value to the tree. Small values go left, big values go right, no duplicate entries may the tree have.

The method `public int read(int)` will search the tree for the argument provided. If found, return the argument itself. If not, return the last value encountered while traversing the tree searching for the argument.

For example, if you write 56, 52, 55, 47 in that order and then read 53, you get 55. If you change the order of writes to 55, 52, 47, 56 and then read 53, you get 52. Given an order of writes, the reads are deterministic and will indicate the correctness of your implementation.

Credit for this part will be awarded independent of the synchronization. You needn't worry about reading from a tree that doesn't have any entries yet: in fact, in the next part, you will guarantee that it doesn't happen.

Part B: Synchronization (20 points)

Please do read the following background before proceeding with the actual task. The code for this is very short and reasonably accessible, but you must know what you're doing.

In the bigger picture, you will have multiple threads trying to read and write to the tree you just built. You do *not* want someone to read while the tree is being written into, or worse, have two threads write together. The logic for the data structure is in place, now you want to *protect the integrity of the data.*

The access to shared data is termed as a **critical section**. We use a construct called a **lock** to protect access to a critical section. The idea is, a lock can only be held by one thread, and a thread must acquire a lock to enter a critical section, and release it immediately once it's done.

Java does provide the **synchronized** keyword to qualify methods or blocks of code. This implicitly declares a lock associated with the calling object of the class. Feel free to read more about it in the excellent Oracle Java Documentation.

For those of you who opened the link, congratulations, you are a connoisseur of Java. For those of you who are here to enjoy the show, the crux is, “When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.”

This strategy is somewhat coarse, and we’d like more refined control. Java knows it too, and provides us with **locks and conditional variables**. Conditional variables, along with complementary booleans, give us a mechanism to block and awaken threads depending on some condition or state.

Conditional variables are associated with a lock. When a thread decides to `await()` on a Conditional variable `cv`, it must do so with a lock held. The implementation releases the lock from the thread, and reacquires it when it resumes execution on a `signal()` on `cv` from another thread. Refer to the excellent Oracle Java Documentation for how to use locks and conditional variables in your code.

In particular, note the while loop while awaiting. The while loop is compulsory, for reasons you will fully appreciate in your OS course next year. (Over?)simplified: you slept because the state wasn’t right for you to execute. Now that you’ve woken up, you better **verify** that the state is indeed right for you to resume.

Although you could try this task with just the synchronized keyword, it’s easier once you get the hang of locks and conditional variables.

Here’s what you have to do.

Implement the `ProtectedTree` class in `ProtectedTree.java`. This has a `private Tree` associated with it, and will be initialised as follows:

```
ProtectedTree ptree = new ProtectedTree(new Tree());
```

`ProtectedTree` also has the public read and write methods with the same signature as their `Tree` counterparts. Here’s what it’ll look like:

```
public void write(int value){
     //sync logic
     System.out.println(“WE”);
     this.tree.write(value);
     System.out.println(“WX”);
     //sync logic
}
```

```
public int read(int value){
     //sync logic
     int answer = this.tree.read(value);
     if (answer == value)
          System.out.println("RS");
     else
          System.out.println("RF");
     //sync logic
     return answer;
}
```

The print statements are important for evaluation, retain them in your code! As evident in the stub code, you have to print WE WX regardless of whether the "write" actually modified the tree. Define a successful read as one where the returned value is equal to the query. You are allowed to read only if there have been more writes than successful reads so far. If not, go to sleep and wait for the signal. You could think of this as the unbounded buffer problem. For simplicity, assume that we will make at least as many write requests as read requests, so you don't need to worry about those deadlocks.

In this way, your synchronized code, when run by multiple threads, will produce a trace of WE, WX and RF, RS on the terminal. Your code will get credit if the trace has the following properties:

1. All requests are accounted for. If there are a total of w writes and r reads, the trace should have precisely 2w + r lines
2. The first line should be a WE.
3. Any WE must immediately be followed by a WX.
4. In any prefix of the resulting trace, there must be at least as many (WE WX) as RS.

In the next part, we will code these threads up, but you will be awarded credit for this part independent of that. You can write simple programs yourself to verify that your trace satisfies the given conditions.

Part C: Threads (15 points)

Java is an Object Oriented Language. A class can implement an interface. An interface is the equivalent of a C++ abstract class with virtual functions/methods. In this part, we will implement the `ReaderWriter` class that `implements` the `Runnable` interface. This, of course, will be the `ReaderWriter.java` file. This is the signature of the default constructor:

```
public ReaderWriter(String FILENAME, ProtectedTree ptree,
boolean iswriter);
```

You need to describe what it does in `public void run()`. You have to read the file called `FILENAME`, which has integers on each line. Depending on the instantiation, you will either read or write each integer to the `ProtectedTree` associated with the instance.

It is important to note that these operations throw exceptions (handle them as you please), and Java requires you to protect them with try-catch blocks. You'll figure out when the compiler complains :P

In `FinalTester.java`, notice the line

```
Thread w1 = new Thread(new ReaderWriter("w1.txt", ptree, true));
```

There is a `ProtectedTree ptree` somewhere in memory, and we're giving objects pointers to that. `Thread w1` will run an instance of the `ReaderWriter` class: that is, open `w1.txt`, read each line and get the integer, and write the integer to `ptree.` Similarly `Thread r1` will open `r1.txt`, read each line, get the integer, and read the integer from `ptree`. We have already checked the correctness of the reads and writes, for this part, you don't need to do anything with the integer value that is returned.

You will be awarded credit for this part if the threads you design produce a valid trace as described earlier.

Submission Instructions

You have to submit `Tree.java`, `ProtectedTree.java` and `ReaderWriter.java`. They should be in the `readwrite` directory under your submission directory.
Name your submission directory `<rollno1>-<rollno2>-outlab8C` (C is uppercase) where the lexicographically smaller roll number comes first, eg.
`17D070059-180070035-outlab8C`

This is the expected directory structure for your submission directory:

```
.
|-- readwrite
|   |-- ProtectedTree.java
|   |-- ReaderWriter.java
|   `-- Tree.java
`-- references.txt
```

Compress it into a tarball strictly with the command `tar -zcvf`
Eg,
`tar -zcvf 17D070059-180070035-outlab8C.tar.gz 17D070059-180070035-outlab8C`

Submit this tarball from the Moodle account of the teammate with the lexicographically smallest roll number. A wrong directory structure makes it inconvenient for us to grade and attracts a 30% penalty for you and makes everyone unhappy. Please be careful.

The submission deadline Basic Java tasks is **Sunday, November 8, 23:59**. **The deadline for 8C is Thursday, November 12, 11:59**. The links will be open for another 6 hours after the deadlines, the standard 2^H% penalty per hour applies component wise. Assume you got everything right but submitted basic in time but concurrency 2 minutes late, you score 50 and 49 respectively.

Concluding Remarks

This exercise is inspired by the structure of this helpful online demo. Of course, you are allowed to borrow from it as long as you cite in your `references.txt`, but we believe that it will be more instructive if you have a go at it yourself first. While the problem statement is verbose, most of it is background and instructions to streamline the evaluations, and the code isn't as long as the size of the PS suggests.

No one:
Literally no one:
Not a single soul:
Mihir: excellent Oracle Java Documentation

Tutorialspoint, javatpoint and jenkov are pretty good too.