Assignment 3

(may be done by a team of at most two students)
Assigned: Thursday, October 4
Due: Monday, October 15 (11:59 pm)

Part 1: Concurrent BST Insertion

Lecture 11 describes the problem of concurrent insertion into a binary search tree (BST). Posted on Piazza is a file `ParTreeInsert.java` whose `main` method creates and starts five threads, each of which inserts five random integers into a tree, and finally prints out all values in the tree. The class `Tree` in this file defines the standard `insert` method but the problem of using this method in conjunction with threads is that some of the inserted values get dropped from tree due to improper synchronization – as illustrated in Lecture 11.

A preliminary solution to this problem is to declare `insert` as a Java 'synchronized' method. Doing so will solve the problem of dropped values, but this solution sacrifices concurrency because values are now inserted sequentially into the tree.

Your task in Part 1 is to write a method in class `Tree` called `insert_par(n)` which preserves the basic logic of the standard `insert`, but permits concurrent insertion of values into the tree, with no dropped values. Concurrent insertion into disjoint subtrees as well as concurrent insertion at different nodes along any path from the root to a leaf should be allowed. To meet these objectives:

1. Do not declare the method `insert_par(n)` as a synchronized method.
2. Define two synchronized methods in class `Tree`, called `lock()` and `unlock()`, using which `insert_par(n)` can ensure that only one thread at a time is accessing any `Tree` object.
3. Call the `lock()` and `unlock()` methods from `insert_par(n)` in such a way that any `Tree` object is locked for as short a duration as possible.
4. Define `lock()` and `unlock()` using Java's `wait-notify` constructs. Hint: Their definitions are similar to other `wait-notify` examples discussed in the lectures.

Run `ParTreeInsert.java` replacing the call on `insert` in class `InsertNums` by `insert_par` and proceed as follows:

1. Check the JIVE object diagram to make sure that it does not have any dropped values, and check the console output to make sure that the printed values are in ascending order.
2. Bring up the JIVE Search window, choose 'Object Created' option, and enter 'Tree' for the class name. There should be 26 entries in the Search Results window for the given test case.
3. Step through the search results one by one, observing the object diagram (with the 'Stacked' option) at each step, until you locate at an object diagram showing 'maximal concurrency', i.e., where there is a maximum number of active threads in disjoint subtrees and also a maximum number of active threads at different nodes along a path in the tree. There could be more than one such diagram; choose any 'maximal concurrency' object diagram.
4. Save the chosen object diagram from step 3 in a file called `A3_obj.png`.

***What to Submit*.** Prepare a top-level directory named *A3_Part1_UBITId1_UBITId2* if the assignment is done by a team of two students; otherwise, name it as *A3_Part1_UBITId* if the assignment is done solo. (Order the *UBITId*s in alphabetic order, in the former case.) In this directory, place your `ParTreeInsert.java` and `A3_obj.png`. Compress the directory and submit the compressed file using the `submit_cse522` command. Only one submission per team is required.

Part 2: Readers-Writers with Write Priority

Lecture 12 discusses the Readers-Writers problem, a classic example in the study of concurrency control. There are two types of concurrent threads, reader threads (readers) and writer threads (writers), and they concurrently access a database. Readers execute the database 'read' operation while writers execute the database 'write' operation. The basic requirement of concurrency control is that a 'read' operation may be executed concurrently by two or more readers, but a 'write' operation should not be executed concurrently with any 'read' or any 'write' operation.

An important variant of the basic problem is the Readers-Writers problem with Write Priority. Here, when a writer tries to access the database and is made to wait because of one or more active readers, all subsequent read requests are delayed until this writer gets to access the database. Active readers are not pre-empted but are allowed to complete their read operations before the writer begins its operation. Every waiting writer takes precedence over every waiting reader regardless of the order of their arrival.

Posted on Piazza is a file `ReadersWriters.java` containing a complete implementation of the Readers-Writers problem with Write Priority. This program is written with `wait-notify` constructs. Your task in Part 2 is to translate all `synchronized` methods and `wait-notify` constructs in terms Java `Semaphores`, using the methodology outlined in Lecture 13. Name the translated program as `RW_Semaphore.java`. In developing your solution:

1. Use one semaphore for translating synchronized methods.
2. Use one semaphore for waiting readers.
3. Use one semaphore for waiting writers.
4. Implement the `notifyAll` operation by performing release(s) on the appropriate semaphore.

Install the State Diagram plugin posted on Piazza and run `RW_Semaphore.java` as follows:

1. Run the program to completion and check that `data` field in object `Database:1` has the value 55555 for the given test case.
2. Save the Execution Trace in a file called `RWS.csv` and load this file into the State Diagram. Set the Canvas Dimension as 1200 x 1200 at the bottom of the diagram.
3. From the dropdown menu, choose the variables `Database:1->r` and `Database:1->w`. Draw the State Diagram and check that the basic requirement of concurrency control for the problem has been met, i.e., in every state of the diagram, the following condition is true: `(w = 1 → r = 0) /\ (r > 0 → w = 0) /\ (w = 0 \/ w = 1)`. Export the diagram to a file called `A3_state1.png`.
4. From the dropdown menu, choose the variables `Database:1->r`, `Database:1->w`, `Database:1->wr` and `Database:1->ww`. Draw the State Diagram and check that the

Writers priority condition has been met, i.e., if ww > 0 in some state then either r = 0 in that state or the value of r should monotonically decrease and reach 0 in some future state and should remain at 0 until ww becomes 0 at a subsequent future state. Export the diagram to a file called `A3_state2.png`.

5. Finally, from the dropdown menu choose the variables `Database:1->r`, `Database:1->w`, and `Database:1->data`. Draw the State Diagram and export it to a file called `A3_state3.png`. This diagram also helps you check the Writers priority condition.

***What to Submit*.** Prepare a top-level directory named *A3_Part2_UBITId1_UBITId2* if the assignment is done by a team of two students; otherwise, name it as *A3_Part2_UBITId* if the assignment is done solo. (Order the *UBITId*s in alphabetic order, in the former case.) In this directory, place `RW_Semaphore.java`, `RWS.csv`, `A3_state1.png`, `A3_state2.png` and `A3_state3.png`. Compress the directory and submit the compressed file using the `submit_cse522` command. Only one submission per team is required.

End of Assignment 3