Assignment 4 Part 2

(may be done by a team of at most two students)

Due: Friday, November 15 (11:59 pm)

Part 2: Readers-Writers with Writers Priority

Lecture 17 discussed 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 Writers Priority. Here, when a writer (w1) tries to access the database and is made to wait because either a read or write is in progress, all subsequent read requests are delayed until this writer (w1) gets to access the database. In other words, a waiting writer takes precedence over every waiting reader regardless of the order of their arrival. Note: Active (i.e., running) readers are not pre-empted but are allowed to complete their read operations before the waiting writer begins its operation.

Posted on Piazza is a file `ReadersWriters.java` containing a complete implementation of the Readers-Writers problem with Writers 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 17 slide 30, i.e.,

1. Use one semaphore (named `s1`) for translating synchronized methods.

2. Use one semaphore (named `s2_r`) for waiting readers and another semaphore (named `s2_w`) for waiting writers.

3. Implement `notifyAll` by performing release(s) on the appropriate semaphores.

Before running `ReadersWritersSemaphore.java`, install the *State Diagram with Property Checker plugin* posted on Piazza – installation instructions in separate file called `StateDiagrams.pdf` – and proceed as follows:

1. Run the program to completion and check that the `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 *Property Checker* – instructions given in `StateDiagrams.pdf`.

3. From the dropdown menu, choose the fields
   `Database:1->r`, `Database:1->w`, `Database:1->ww`
   Enlarge the Canvas Dimension (using the text boxes at bottom of diagram), and draw.

4. In the *Abbreviations* text box enter:

```
Database:1->r = r, Database:1->w = w, Database:1->ww = ww
```

5. In the *Properties* textbox, enter:

```
G [ (w == 1 -> r == 0) &&
    (r > 0 -> w == 0)  &&
    (w == 0 || w == 1) &&
    (r > 0 && ww > 0 -> r’ <= r)
  ]
```

- The first three conjuncts express the basic readers-writers concurrency control policy, and the last conjunct expresses the writers-priority condition.
- The variable `r’` refers to the value of `r` in the next state, and the condition states that the number of running readers must monotonically decrease when there are waiting writers.
- The outermost `G` specifies that this property must hold globally, i.e., in all states.

6. Press Validate and look for the message *“All properties satisfied.”* below the Properties textbox. If this message does not appear, look for states highlighted in red in the state diagram, as they will help determine the cause of failure. (The topmost state is always red-highlighted when a G property fails.) Use this information to correct your program.

7. When all properties are satisfied, export the diagram to a file called `A4_state1.svg`.

8. Reset previous choices and, again, from the dropdown menu, choose the fields
   `Database:1->r`, `Database:1->w`, `Database:1->data`.

9. Draw the State Diagram and export it to a file called `A4_state2.svg`. This diagram helps visually check the Writers priority condition – the diagram should slope to the right and end with a long tail of read operations.

***What to Submit*.** Prepare a top-level directory named *A4_Part2_UBITId1_UBITId2* if the assignment is done by a team of two students; otherwise, name it as *A4_Part2_UBITId* if the assignment is done solo. (Order the *UBITId*s in alphabetic order, in the former case.) In this directory, place `ReadersWritersSemaphore.java`, `RWS.csv`, `A4_state1.svg`, and `A4_state2.svg`. Compress the directory and submit the compressed file using the `submit_cse522` command. Only one submission per team is required.

End of Assignment 4 Part 2