SCHOOL OF COMPUTING, QUEEN'S UNIVERSITY

CISC324: Operating Systems

Lab 3

January 25, 2020

This lab may be your first time programming exercise in which you will be writing concurrent programs that use semaphores in Java. Java programming language provides you with a package `java.util.concurrent.semaphore`, in order to use semaphore operations. You will learn how to create and initiate semaphores and use them in both styles: process execution ordering (waiting style) or for critical section (mutual exclusion style). You will also learn how to use the primitives `acquire()` and `release()` that have been discussed in the lecture.

Objective

In this lab, you are asked to understand and execute the Java code which is provided to you. This Java code is an implementation for the readers and writers classical synchronization problem with readers having priority. The readers and writers are basically implemented by their respective threads. In the readers and writers problem, a shared data structure, usually a file or a database, is accessed by reader threads and writer threads. For efficiency, many readers should be allowed to read simultaneously. For correctness, only one writer is allowed to write into the data structure at a time, all other threads (readers and writers) must wait. There exists three possible solutions. Each solution defines a kind of scheduling policy to decide which thread should first have access to the data structure as described below:

- **Readers have priority.** Once a reader has started reading, all newly-arriving readers are allowed to start reading (multiple readers can read at a time). A waiting writer must wait until all readers have finished. This means that a starvation may occur in cases where there is a steady stream of newly-arriving readers. You have been given the Java code that implements this solution:

  `MainMethod.java`: Main Java class that creates reader and writer threads.

  `Reader.java`: Reader Java class that defines the reader thread.

  `Writer.java`: Writer Java class that defines the writer thread.

  `Synch.java`: This Java class declares the semaphores (mutex and wrt) and counter (readcount) used to synchronize readers and writers.

  `RandomSleep.java`: This Java class defines a "doSleep" method for suspending a thread.

- **Writers have priority.** Anytime there is a writer waiting, no new readers are allowed to start. In case there is a steady stream of newly-arriving writers, the readers may starve.
- **A starvation-free solution.** To avoid starvation it is necessary for readers to give priority to writers, and vice versa. In a busy system, this results in one batch of readers, followed by one writer, followed by all the readers that arrived in the meantime, followed by another writer etc.

Your task *Modify and adapt the given Java code that implements the Readers-having-priority solution so that it becomes a Java code that implements the writer-having-priority. Below are some directions that express the requirements of the new solution.*

Directions

Here are the synchronization requirements for "giving writers priority":

- Writers must have exclusive access: if a writer is active, no other writer or reader may start.
- Readers are active in parallel: if reader(s) are active, and no writer is waiting, a new reader is allowed to start.
- Once a reader or writer becomes active, it is allowed to finish. There is no way to interrupt active readers when a writer arrives: the writer has to wait until all these readers finish. The last reader that finishes should wake up the waiting writer.
- If a new reader arrives when a writer is waiting or a writer is active, the reader must wait.
- When a writer finishes, it wakes up the next waiting writer (if there is one), or it wakes up all the waiting readers (if there are any).
- When a reader finishes, it checks whether it is the last reader. If so, it wakes up the next waiting writer (if there is one).

Hints

1. In the new solution there will be two queues. The first waiting queue is for the readers who arrived when a writer was writing or when a writer was waiting. The second waiting queue is for the writers who arrived when at least one reader was reading the data structure or when a writer was writing on the data structure.
2. You need to have multiple counters to count how may readers and writers are active (reading/writing) and/or waiting (in the queue).
3. Counters are shared variables among readers and writer i.e., they are subject to race condition and hence should be protected when manipulated at a given time by a given thread.
4. For a reader to read, it has to check whether there are waiting writers or an active writer already there. If none is there, it starts reading. Then, when it finishes, if it is the last reader, it checks whether a writer is waiting to wake it up. However, for a writer to write, it has to check whether some readers are reading or a writer is writing. Then, when it finishes writing, it wakes up a writer if there is any, else it wakes up all waiting readers if there are any.

What should be submitted

- Place your codes (`MainMethod, Reader, Writer, and Synch`) in one folder.
- Your codes should be well commented. You must add comments to explain why you have used a particular variable and what each piece of code performs.
- Run your program for a certain number of readers and writers and save your output in a text file (copy and past). Perform at least three different runs (three different output files). Do not run your program for more than 10 readers and writers.
- Use a ReadMe.txt file to explain the settings that you have used to produce each output file (e.g., for 10 readers and 10 writers, the outputs are in file out10.txt).
- Place everything inside one folder and compress the file using zip.
- A well annotated (or commented) code with a correct output will get 100% of the mark. A correct output with no annotation gets 75% of the mark. For outputs that contain errors, but the annotation claims that there are no errors (or there is no annotation) gets 50%.