Coursework CSC8016

Giacomo Bergami

12th of February, 2024

Use Case Scenario

We want to implement a forum platform (`Blog`), where users are opening topic threads for discussions (`createNewTopicThread`) and posting some comments (`addPostToThreadId`). When a topic thread is opened, there might be no messages available. We also freely assume that all the users are also moderators, thus allowing to remove specific threads as a whole (`removeTopicThreadById`). We do not consider the possibility of removing single messages within a topic thread. In addition to this, we might also freely assume that each user might retrieve all the available topic threads names sorted by published date (`getAllTopics`) as well as their IDs (`getAllTopicIDs`); a user can also retrieve all messages from a specific topic thread (`getAllMessagesFromTopic`). Furthermore, the user might be also want to retrieve the latest available update event within the server (`pollForUpdate`) if available and otherwise wait; an eponymous method also accepting the previous message update as an argument will only return if a new event being different from the one being provided as an argument is effectively generated by one of the users using the platform. When required, the users receive the feedback on their operation's status through either a boolean (true for successful, false for error or unsuccessful) or a `TopicUpdates` message. The latter class shall not be changed by the student.

We might freely assume that such platform acts as a monitor, where requests are handled through method invocation. For synchronising read and write operations over the server, the students shall only use the `ReadWriteMonitorMultiRead` to guarantee the main synchronisation mechanism.

In this scenario, the users should be allowed to perform all of the aforementioned operations concurrently with no race conditions. Using 2PL transactions or ensuring the fairness over reader and writing threads is not strictly required.

Assumptions

- In a realistic example, communications happen through restful HTTP requests, and the `Blog` is actually an HTTP server. In this module, we don't require that. We can freely assume that each user is mimicked by one single thread. We assume they directly exploit such an interface (no FrontEnd or HTTP requests are required!)
- The student shall keep the same package structure, as the tests will assume to access the classes within the `uk.ncl.CSC8016.jackbergus.coursework.project3` package. The students might consider extending the tests for ensure the code correctness. Still, the overall work will be assessed through our provided code, and not theirs!

- The students shall freely assume that the main server, `Blog`, will be an advanced monitor, where an instance of the `ReadWriteMonitorMultiRead` is going to provide the required `ReentrantLock` mechanism.
- The student might try to implement such a server as a (multi)`Thread` service via a thread pool to gain extra points, but this is not strictly required. In this case, the student might freely assume that the thread pool is started by calling the `Blog` constructor, and one single thread will stop the thread pool if there is any inactivity for more than 30 seconds. Such pooled threads might receive the client's method invocation as a sequence of "client messages" through shared variables, where requests need to be handled FIFO. The client will return from its method invocation once the message is handled successfully by one of the threads. Still, this should require not changing the strict assumption over the API being provided, which should not be by any means changed.
- We assume that no user registration, logging, or ban, is given. Additional features going beyond the specifications within the marking scheme will not be evaluated.
- The server should allow the creation of multiple topics with the same name and different and incremental ID **as far as different timestamps are given to distinguish them**. In order to maximise seriality and concurrency requirements, the students might investigate *optimistic protocols*, thus allowing the concurrent posting across different topic, as this will give the chance to gain extra points.

Submission Requirements

1. To help us with the marking, the students should update the `studentID` method so to return a string corresponding to their student id. This will help us expediting the marking using our automated tool.
2. `Blog` and `ReadWriteMonitorMultiRead` should be finalised, as the current implementation does not pass the provided tests! The latter class should be used within the former as a main reentrant locking mechamism for the `Blog` monitor (or thread-pool server).

   The `TopicUpdates` class
3. The student shall submit the code as a zipped *Maven* project via `File > Export > Project to Zip file...` with **no** *jar* and *classes*. The source code will be recompiled from scratch, and no pre-compiled jar/class is going to be run.
4. If you want to use an external Java library, please consider the following:
   - The Java library should be explicitly described as a `<dependency>` in the `pom.xml` file, and should only access the libraries from the default *Maven Central Repository*.
   - A library might provide single concurrency mechanisms primitives, but not ready-made solutions already composing those: semaphores, monitors, locks, just logs, thread barriers, thread pools, passing le baton mechanisms are allowed. Code reuse from the exercises and examples seen in class is permitted.
   - Systems completely solving the coursework for you are **strictly prohibited**: e.g., any kind of (data) management system having concurrency control (ensuring safe concurrent thread access to any data representation) and supporting concurrent transactions (implementing any kind of transaction protocol, either pessimistic or optimistic) **must be avoided**, as they both implement commit/aborts and thread-safe operations on data.

- None of the (direct or indirect) dependencies of the coursework should rely on external servers or processes to run or to be installed.
- The solution should **not** include external jar files.
- If unsure whether the solution might be exploited, please ask before submitting.

5. Attached to the source code, please provide a short report motivating the compliance of the source code to each point and sub-point of the marking scheme. Providing such report in form of comments in the implementation is also fine. New classes might be created for supporting the implementation, but existing classes should be neither renamed or moved to a different package.

Marking Scheme

The marking scheme is capped at **100%**.

- Single-Thread Correctness **[+50%]**

  +4%: I cannot interact with a blog thread if this was not previously created..

  - ■ You should not be able to remove a non-existing topic thread.
  - ■ You cannot post a comment over a non-existing topic thread.
  - ■ You cannot read the posted messages for a non-existing topic thread.

  +16%: I can always interact with a topic thread that was previously created.

  - ■ You can create a new topic thread.
  - ■ You can post a message within an existing topic thread.
  - ■ You can correctly read all the messages in their order of appearance.

  +4%: I am correctly handling the thread closure for an existing topic thread.

  +7%: I am correctly handling the `pollForUpdate` method where, if successful requests are always fired before polling for events, should always return the most recent event available.

  - ■ You only retrieve the latest update event.
  - ■ You can correctly detect all the required interaction events of the users over the platform.

  +3%: If no user performs any suitable action, `pollForUpdate` should pause indefinitely.

  +10%: I am correctly handling the `set` method from `ReadWriteMonitorMultiRead` (please see the documentation of this method for additional information concerning its expected behaviour).

  +6%: I am correctly handling the `get` method from `ReadWriteMonitorMultiRead` (please see the documentation of this method for additional information concerning its expected behaviour).

- Multi-Threaded Correctness **[+50%]**

  +7%: The concurrent creation of different topics is handled correctly.

  - ■ Creating multiple topic threads does not raise deadlocks.

    - All the topics requests being sent are fulfilled and available in the exact same order of creation.

  +9%: The concurrent posting of multiple messages within multiple topic threads is handled correctly.

    - Posting multiple messages across topics or within the same topics does not result into deadlocks.
    - All the messages being sent are fulfilled and available in the exact order of posting.

  +12%: We have one querying user, one moderator user, and eight answering user. Each of those are simulated as distinct threads. The *querying thread* posts the topic and publishes the first comment within the thread; after receiving exactly 8 answers from the answering threads, the querying user posts a thank you message. The *answering threads* are waiting for the first message from the querying user to be posted, after which they provide a reply within the thread. The *moderator thread* will close all the topics reaching a number of 10 comments.

  +12%: While having only one single user running and one subscriber to receive the updates from the website, no interference occurs, and all the perceived events actually match the expected results.

- Advanced Features (capped at maximum 10%):
  - **[+1%]** The program allows to visually determine the correctness of the operations performed by the threads (e.g., terminal prints, engaging with extra tests not provided to the students, or graphical user interfaces).
  - **[+1%]** Any Java library imported via `pom.xml` 'not violating the 3rd Submission Requirement. The following libraries will not be considered, as already provided in the given source code as dependencies: `annotations` from `org.jetbrains`, and `jansi` from `org.fusesource.jansi`.
  - **[+1%]** Either the methods to be implemented or the extended tests exploit Java's concurrent collections.
  - **[+1%]** The student correctly uses `ReentrantLocks` and `Conditions` within the `ReadWriteMonitorMultiRead` class.
  - **[+3%]** The `ReadWriteMonitorMultiRead` provides a satisfactory extension of the classical `ReadWriteMonitor` by not only considering the number of services waiting to read and write a specific latest message update, but also waiting to have a new event being written different from the previously-obtained one. Furthermore, such class is actually used as the only way to deal with read-write operations within `Blog`, which are also handled correctly.
  - **[+1%]** Thread pools are used to either handle multiple requests from multiple users, or start multiple threads within the extended tests.
  - **[+5%]** The Blog service is emulated realistically as a separate thread accepting restful requests; this requires the Blog to handle requests one at a time through a queue; Still, the student shall not change the API interface as currently provided.