Lab 4: The Readers-Writers Problem

Adriaan Jacobs
adriaan.jacobs@kuleuven.be

Jonas Vinck
jonas.vinck@kuleuven.be

Novemeber 30, 2022

HINT: Read this *before* asking questions.

1 Introduction

So far in our examples, the thread that read data from a shared resource, also altered it in some way. In the shared buffer from the first lab, the consumers removed entries from the buffer. When transferring money in the banking example, one account was decremented and the other incremented. When allocating or freeing memory, internal bookkeeping had to be updated. Another common problem with concurrency in parallel computing, is when two kinds of accessing threads take on other roles: readers and writers. A reader will only read from a shared resource, thus not altering it. In contrast, a writer will write to the shared resource, altering it. This means that multiple readers can be allowed to access the resource in parallel. A writer, however, requires that it is the only thread accessing the resource, with even readers not being allowed access.

In this lab, we will be looking at a scenario where multiple reader threads and multiple writer threads attempt to access a shared resource concurrently. In fact, the so called *readers-writers problem* can be divided into three version. We will address these versions in this lab and build a better solutions with every step.

2 Code Skeleton

We provide a code Skeleton for this lab exercise that will take care of most things for you. To make things easier, all files you shouldn't have to change are part of the same `dontreadme/` directory. The only file that really matters for your solution is `Proxy.java`, and potentially `RWConfig.java`.

The system centers around an abstract representation of a *shared resource*. Two operations are defined on this resource: *read* and *write*. Further, several reader and writer threads are spawned, which will continuously read from and write to the shared resource, respectively. For sake of simplicity, and comparability between the three versions, we assume that a reader thread only reads and a writer thread only writes. In the Introduction we already mention some conditions that need to be guaranteed to avoid corrupting the resource or the flow of the application using it. Our implementation checks that these conditions hold and, do not worry, will tell you when you mess up. However, you will also have to take these conditions into account for your synchronization. The essential conditions are as follows.

- Multiple readers are allowed access in parallel.
- Only one writer is allowed at any time.
- Readers and writers cannot access the resource simultaneously.

`RWConfig.java` contains configurations for the code skeleton. The configuration options and their meanings are listed in Table 1. Most of these values are, admittedly, chosen somewhat arbitrarily. However, it is generally safe to assume that writing to a resource will take longer than reading from it. You could change these values, or we could afterwards, to stress your solutions differently.

Option	Description
PERFORMANCE	Disables debug printing. For more info, see `Util.java`.
RUN_DURATION_MS	Duration of the simulation in milliseconds.
N_READERS	Number of reader threads that will be spawned.
READER_WAITING_MIN	Lower bound for the duration of a single read.
READER_WAITING_MAX	Upper bound for the duration of a single read.
READER_PROCESSING_MIN	Lower bound for the wait after a read.
READER_PROCESSING_MAX	Upper bound for the wait after a read.
N_WRITERS	Number of writer threads that will be spawned.
WRITER_WAITING_MIN	Lower bound for the duration of a single write.
WRITER_WAITING_MAX	Upper bound for the duration of a single write.
WRITER_PROCESSING_MIN	Lower bound for the wait after a write.
WRITER_PROCESSING_MAX	Upper bound for the wait after a write.

Table 1: Readers-writers config options

`Proxy.java` is where you will be implementing your synchronization solution. It functions, as the name suggests, as a proxy for the shared resource. The actual shared resource is not synchronized in any way, thus it requires synchronization before and after the calls to `read()` and `write()`. To do this, the readers and writers wrap them in four proxy methods: `preRead()`, `preWrite()`, `postRead()`, and `postWrite()`. These methods should house your synchronization logic and the singleton `Proxy` object can hold any variables you use in your implementation.

Solving the issues in this lab might be technically possible using basic synchronized blocks, but it will likely be significantly more complex than if you were to use other Java synchronization primitives. Among others, we can recommend you look at `AtomicInteger`, `ThreadLocal`, `Lock`, and `ReadWriteLock` when implementing your solutions.

`RWLogger.java` is our own small logger implementation. `debugf(String message, Object... values)` offers the option to only print the message of the configuration does not specify that we're running in performance mode. Similarly, you could enable or disable features based on custom configurations you can add to `RWConfig.java`.

3 Running the Framework

The entire simulation is started by simply executing the `main()` in the code root directory. At startup, it will first print your configuration to the console, then spawn and start all reader and writer threads immediately after. The configuration defines the time the simulation will run at least. After the specified time has elapsed, the main thread will collect the data and print it to the console. The total amount of reads and writes, as well as the individual reads and writes of each thread, are printed out alongside the actual runtime in nanoseconds. These results are determined in a way that is inherently racy, i.e. not thread safe, but this should never cause any real issues. Even then, if the simulation runs for long enough, the errors due to race conditions become insignificant.

4 The First Readers-Writers Problem

The first solution you might think of is the same strategy we have often employed for the other labs. Enforcing mutual exclusion. We encourage you to quickly try this. you'll see that the old method of declaring synchronized blocks no longer works in this framework. You can enforce mutual exclusion still, using locks, but the further you go through this lab, the less efficient it will prove to be. As a first step, we could increase the throughput based on the fact that readers do not alter the data, and can thus be allowed to access the data in parallel. If this is the case, it might seem foolish to make readers wait if another reader is currently reading. Augment the synchronization so that no reader has to wait while the shared resource is currently being read. This is called *reader preference*.

5 The Second Readers-Writers Problem

You'll quickly experience why this first solution will lead to issues. A writer will remain blocked from accessing the shared resource for as long as threads keep reading from it. If readers access in quick enough succession, such that they always overlap with previous readers, the writers will starve. One could argue that updates to shared resources are most likely critically important to the system, thus a writer should never have to wait longer than is necessary. Augment the synchronization to reflect this so called *writer preference*.

6 The Third Readers-Writers Problem

Again, you'll quickly see why the second solution is also problematic. As with the first solution, if enough writers request access to the shared resource that they can always immediately take over from the previous writer, the readers will starve. The ideal, but yet most difficult, property to enforce is *fairness*. How you implement this is hard to say, thus you have full creative freedom here. Augment the synchronization with the condition that no thread should ever starve.

7 Report

A written report is due by the evening before the next lab session, **23:59 December 13, 2022**. Or earlier, we're not stopping you. In this report, write about any design decisions you made while solving the synchronization issues, as well as any interesting findings. Try to limit the report, conciseness is key! In addition, add all Java source files for this lab as a `.zip`. Add your solutions for the three readers-writers problems in separate directories in the `.zip` file. If you followed the separation already enforced in the skeleton code and did not change files other than `Proxy.java` and `RWConfig.java`, it suffices to only hand in those files. You can send in the report and source files by email.