TU Munich, Department of Informatics
Chair XXV: Data Science and Engineering
Prof. Dr. Thomas Neumann

Exercise for the *Practical Course: Systems Programming in C++* Summer Term 2021

Michael Freitag, Moritz Sichert ({freitagm,sichert}@in.tum.de)
https://db.in.tum.de/teaching/ss21/c++praktikum

Sheet Nr. 08 (due on 08.07.2021 at 23:59)

Create a fork of the Git repository at `https://gitlab.db.in.tum.de/cpplab21/sheet08`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1

(30 points)

In this exercise you will add multi-threading support to a given data structure. The project scaffold for this exercise already contains a working *multimap* which is implemented based on a chaining hash table.

Data Structure While you do not have to implement the data structure yourself, an understanding of its inner workings is required to add proper multi-threading support. A multimap differs from a regular hash table in that it allows multiple entries with the same key. In order to keep things simple, our implementation supports only 32-bit integers as keys and values.

Internally, our multimap maintains a vector of buckets in the same way as a regular chaining hash table. Each bucket contains a pointer to the beginning of a linked list which contains all `(key, value)` entries which fall into the same bucket. Since existing entries are never overwritten, we can simply insert new entries at the beginning of this linked list without having to traverse the entire chain.

The main operations of our multimap are as follows.

- `insert`: Insert a `(key, value)` pair into the hash table. The bucket index is determined based on the key, and the entry is inserted at the beginning of the respective chain.
- `findFirst`: Returns an iterator pointing to the first entry with a given `key`. If no such entry is found, returns the past-the-end iterator of the multimap.
- `findNext`: Given an iterator, returns an iterator pointing to the next entry with the same `key`. If no such entry is found, returns the past-the-end iterator of the multimap.

Furthermore, our multimap supports iteration over all entries through the standard `begin` and `end` functions.

Requirements Your job in the following subtasks is to make the `insert` and `findFirst` functions thread-safe. Your thread-safe version of these functions should fulfill the following requirements.

- `insert`: The `insert` function must be thread-safe with respect to concurrent invocations. That is, if multiple threads invoke the `insert` function in parallel, you have to ensure that the multimap deterministically contains all entries that have been inserted in parallel. The order in which entries are inserted does not have to be deterministic.
- `findFirst`: The `findFirst` must be thread-safe with respect to concurrent insertions. That is, it must be possible that one thread invokes the `findFirst` function while another thread concurrently invokes the `insert` function. If this situation occurs, you have to ensure that the `findFirst` function does not see the multimap in an inconsistent state.

The `begin` and `end` functions, as well as the iterator operations do *not* have to be thread-safe. Furthermore, the `findNext` function does not have to be explicitly thread-safe, as insertions can only occur at the beginning of a chain. Nevertheless, depending on your implementation, you may have to adjust code in these operations as well, e.g. to account for changed data types.

We implemented test cases that check the thread-safeness of the `insert` function. Depending on your system, you may have to build the tests in release mode in order to observe any synchronization issues. Since the thread-safe `findFirst` function does not necessarily behave deterministically, we cannot provide any test cases that check the thread-safeness of this function.

(a) Make the `insert` and `findFirst` functions of the `MutexMultiMap` class thread-safe by using mutexes.

(b) Make the `insert` and `findFirst` functions of the `AtomicMultiMap` class thread-safe by using atomic data types and operations. You can use the `parallel_ht_benchmark` executable to compare the performance of the mutex-based and atomic-based implementations. You should build this executable in release mode to obtain meaningful performance numbers.