Concurrent Data Structures
Lab Assignment 2

Parosh Aziz Abdulla
Sarbojit Das
Fridtjof Stoldt

December 5, 2023

Deadline: 2023–12–25
Please, submit your program and potentially instructions to run them.

1 Instruction for the lab

1.1 Lab Template

There is a code template for this lab. You are not required to use the template, but it is highly recommended as the tasks are defined using the template. The `README.md` file contains a quick overview of the contained files.

1.2 Compilation

The lab template contains a `makefile` file that can be used to build the project. The file has been tested on Ubuntu with g++ v11.4.0, but it should also work on macOS and other Unix-based systems. You can also manually invoke the following `g++` command in the `src` folder.

```
g++ -c main.cpp -o a.out -Wall
```

This will generate an `a.out` file.
Your final submission should compile and run on the Linux lab machines from Uppsala University[1]. If your submission requires a different command for compilation, please document it in your report.

1.3 Run Instructions

The created binary takes the task number as the first argument. For example `./a.out 1` will run the first task.

1.4 Memory Management

C++ uses manual memory management. In concurrent programs, this can cause problems if one thread uses a pointer to access memory freed by another thread. If it is possible, try to free all the memory you allocate. Otherwise, you can skip the freeing if you can explain why freeing might be unsafe because memory management is not part of this course. In lab 1, it should be possible to implement all data structures without leaking memory.

[1] See `https://www.it.uu.se/datordrift/maskinpark/linux` (Accessed 2023-11-17) for more information

2 The `Set` abstract data type

Abstract Data Types (ADTs) are mathematical objects that allow us to specify the expected behaviours of implementations of common data structures such as sets, queues, and stacks. In this lab assignment, you will implement different versions of the `Set` data type and compare performance. The `Set` data type used in this lab has the following methods:

1. `add(elem) -> bool`:

   If `elem` is not in the set, it will be added, and `true` is returned; otherwise `false`.

2. `rmv(elem) -> bool`:

   If `elem` is in the set, it will be removed, and `true` is returned; otherwise `false`.

3. `ctn(elem) -> bool`:

   If `elem` is in the set `true` is returned, otherwise `false`.

Task 1 Optimistic synchronization Implement a concurrent list-based set with *optimistic synchronization* in a file named `optimistic_set.hpp`.
The code you need to modify has been marked with \\ `A01:` comments. You can test your implementation using the command `./a.out 1`.

Task 2 Lazy synchronization Implement a concurrent list-based set with *lazy synchronization* in a file named `lazy_set.hpp`.
The code you need to modify has been marked with \\ `A02:` comments. You can test your implementation using the command `./a.out 2`.

Task 3 Fine-Grained Set In task 4, you will compare the performance of different synchronization mechanisms. The fine-grained set from lab 1 should be included in the experiment. For this, copy your Fine-Grained set implementation from the previous lab into the `fine_set.hpp` file, and remove all code related to monitoring. The constructor of the `FineSet` should take no arguments. You can test this copy using the command `./a.out 3`.

Task 4 Experiment
Benchmark the set implementations from task 1, 2 and 3. For this, you should first use the command `make bench` to build the project with `-O3` optimizations and then run `./a.out 4`.
The benchmark will perform the following experiments:

- For each value $i = 10$, 50, and 90, a benchmark will run operations such that $i\%$ are `ctn()` operations. From the remaining operations, 90% will be `add()` and 10% will be `rmv()` operations. For instance, for $i = 60$, we have 60% `ctn()`, 36% `add()`, and 4% `rmv()` operations.
- For each $n = 2, 4, 8, 16$, and 32 the benchmark is performed with n worker threads. Each worker thread will perform 500 operations on the shared data structure.
- The code also tests different value ranges. First, the values are from $0, 1, \ldots 8$ and then $0, 1, \ldots 1028$.

`./a.out 4` will print the measured time in milliseconds.
Depict the results in tables and graphs, where the x-axis is the number of threads, and the y-axis is the throughput. Explain the tables and curves.

3 The Multiset abstract data type

A multiset is a generalization of a set where a given element may occur multiple times in the multiset. An example of a multiset (over the set of integers) is [2; 2; 2; 3; 7; 7]. Here, the number of occurrences of 2, 3, and 7 is 3, 1, and 7, respectively. A multiset can also be viewed as a special case of a stack or a queue where the order of the elements is not relevant. For this task, you are asked to implement a concurrent multiset that supports the following functions:

1. `add(x) -> bool`:

   Adds element x to the set and returns `true`

2. `rmv(x) -> bool`:

   If `x` is in the set, it will be removed, and `true` is returned; otherwise `false`.

3. `ctn(elem) -> int`:

   Returns the count of element `x` (how many instances of x there are in the multiset).

Task 5 Fine-Grained Multiset Implement a concurrent multiset using Fine-Grained synchronization in `fine_multiset.hpp`. All code you might need to modify has been marked with \\ `A05:` comments. Identify the linearization policy and document it in the report.
As in the previous lab, test your implementation by inserting the events into the `EventMontior`. The rest of the testing code is already in the `task_5()` function of the lab template. You can run the code of `task_5()` using the command `./a.out 5`.