Concurrent Data Structures
Lab Assignment 1

Parosh Aziz Abdulla
Sarbojit Das
Fridtjof Stoldt

November 17, 2023

Deadline: 2023–12–02
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 check if the implementation is safe for concurrent usage. 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 Standard non-concurrent set An operation is defined as a pair of a method and an argument to the method. An operation together with the output can be written as a tuple, such as `(add, 6, false)` when the operation `(add, 6)` returns `false`. In the given lab template, these are called `Events`. From a given state s, an event is allowed if the operation yields the expected output. For instance, assume we have the set $s = \{3, 7, 9\}$. The program allows the events $(\texttt{add}, 5, \texttt{true})$ and $(\texttt{add}, 7, \texttt{false})$ from s, but not $(\texttt{add}, 8, \texttt{false})$.
The lab template implements the set abstract datatype, where the state is an instance of C++ `std::set`. In this task, you are supposed to test the implementation of the set abstract datatype by running event sequences. Run task 1 of the lab template using the command `./a.out 1` and inspect the output. Make sure that you understand the code of `task_1(...)`. Then, add a longer sequence of 10+ instructions that are allowed in the `task_1` function of `main.cpp`.

Task 2 A simple Set Implement a simple `Set` abstract datatype in the `simple_set.hpp` file, where the state is a linked list. The code you need to modify has been marked with \\ `A02:` comments. You can also add testing and debugging code. Please do not use any synchronization mechanism for the set. You need to test your implementation. For this, insert all performed operations with their output into an event sequence shared across all threads. This is already done in the `EventMonitor` in the template. A separate thread is used to monitor the sequence and check each event is allowed. If an operation is not allowed, a message will be printed. Make sure to keep the `this->monitor.add(...)` code for monitoring.
You can run the code for task 2 using the command `./a.out 2`. The single-threaded version should process all 200 operations just fine. The multi-threaded version will fail, which is expected.

Task 3 A Coarse-Grained Set Implement a `Set` using a linked list that uses coarse-grained locking internally to allow concurrent access. This implementation should be done in `coarse_set.hpp`. The code you need to modify has been marked with \\ `A03:` comments.
For this data structure, you also want to test our implementation. For this, you need to identify the linearization points and insert the corresponding event into the `EventMontior`. The rest of the testing code is already present in the `task_3()` function of lab template. You can run the code of `task_3()` using the command `./a.out 3`.
Hint: You might be able to reuse parts from task 2.

Task 4 A Fine-Grained Set Implement a `Set` using a linked list, that uses fine-grained locking internally to allow concurrent access. This implementation should be done in `fine_set.hpp`. All code you might need to modify has been marked with \\ `A04:` comments.
Test your implementation like the previous task. Please identify the linearization points and insert the corresponding event into the `EventMontior`. The rest of the testing code is already present in the `task_4()` function of lab template. You can run the code of `task_4()` using the command `./a.out 4`.
Hint: You might be able to reuse parts from task 2.