Concurrent Data Structures
Lab Assignment 3

Parosh Aziz Abdulla
Sarbojit Das
Fridtjof Stoldt

December 18, 2023

Deadline: 2024-01-12
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.

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

2 The `Stack` abstract data type

The *Stack* abstract data type has the following methods:

1. `push(x) -> int`:

   Adds the element `x` to the top of the stack and returns `true` (1).

2. `pop() -> int`:

   Removes the top-most stack element and returns it. If the stack is empty, the special value `-1` is returned.

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

   Returns the size of the stack, meaning how many elements are currently in the stack.

Task 1 Treiber Stack (40 points) Implement a stack using the treiber algorithm in the file named `treiber_stack.hpp`.
Identify the linearization policy and document it in your report.
The code you need to modify has been marked with \\ `A01:` comments. As in the previous lab, test your implementation by inserting the events into the `EventMontior`. You can test your implementation using the command `./a.out 1`.

3 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 2 A Lock-Free Set (40 points) The course covered how *compare-and-set (cas)* operations can be used to implement concurrent data structures, without using locks. For this task, you're asked to implement a lock-free set, based on the data structure described in chapter 9.8 of the course book[2]. A template for this data structure is available in the file named `lock_free_set.hpp`.
The book calls the data structure `LockFreeList`, but it already has the properties of a set, mainly that every item can only be in the set once. The book uses a `AtomicMarkableReference` class to store a flag inside a pointer value. The lab template provides the `AtomicPtrAndFlag` class, which implements the same behavior in c++.
Identify the linearization policy and document it in your report.

[2] *The Art of Multiprocessor Programming* by Maurice Herlihy and Nir Shavit

The code you need to modify has been marked with \\ `A02:` comments. Your implementation will be benchmarked in the next task, so you don't need to insert any events into a sequential queue. You can test your implementation using the command `./a.out 2`. If you get totally stuck on this implementation, you can also get points for explaining what you've done and what the problem is, along with possible solutions.

Motivation:

- Several new concurrent data structures, like concurrent heaps and priority queues, use lock-free skiplists internally. Skiplists distribute their items into sub lists to allow for quick search and access. The skiplist implementation, described in chapter 14.4 of the course book, is based on the lock-free set implemented in this task. You will not be asked to implement a skiplist, but we wanted to share, why this data structure, in particular, is interesting and relevant.
- This task also shows that you're able to understand descriptions of concurrent data structures and can implement them yourself.

Task 3 Benchmarking (20 Points)
Benchmark the set implementation from task 2. For this, you should first use the command `make bench` to build the project with `-O3` optimizations and then run `./a.out 3`.
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 3` will print the measured time in milliseconds. (The benchmark is the same, as the one used in the previous lab)
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. You can also compare this data structure to the benchmarks from the previous lab.
Give a recommendation, under which circumstances this set performs well, or under which another one, like the `LazySet` or `OptimisticSet` might be preferable based on the benchmark.