Concurrent Algorithms and Data Structures
Lab Assignment 2

Parosh Aziz Abdulla
Fridtjof Stoldt
Tage Johansson

December 2, 2024

Deadline: 2023-12-18

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 (Using `make`). 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++ -Wall -lpthread -std=c++20 main.cpp
```

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 explain in your report when freeing would be unsafe.

[1]See `https://www.it.uu.se/datordrift/maskinpark/linux` (Accessed 2024-10-25) for more information

1.5 A Note about Atomic Types

It is undefined behavior to read from and write to the same value from multiple threads simultaneously. One solution is to use a mutex to ensure exclusive access to a section of the program. You will practice this; however, in some cases, we may only want to perform a single read or write operation on a variable, and using a mutex in such situations can introduce unnecessary overhead.

In these cases, you should use the built-in `std::atomic<>` type. For reference, see this complete documentation on cppreference.

Essentially, all you need to do is replace assignments like `p->next = c` with `p->next.store(c)`. Similarly, when reading from an atomic variable, you need to use the `load()` method, such as `p->next.load()`.

2 The `Set` Abstract Data Type

Abstract Data Types (ADTs) are mathematical objects that allow us to specify the expected behaviors of implementations of common data structures such as sets, queues, and stacks. 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 Set (15 Points) Implement a concurrent linked-list-based set with *optimistic synchronization* in the `optimistic_set.hpp` file.

The code you need to modify has been marked with \\ `A01:` comments. You can test your implementation using the command `./a.out 1`.

You should use std::atomic for variables that are simultaneously read from and written to. (See 1.5) If your implementation leaks memory, remember to document this in your report. (See 1.4)

Task 2 Lazy Synchronization Set (15 Points) Implement a concurrent linked-list-based set with *lazy synchronization* in the `lazy_set.hpp` file.

The code you need to modify has been marked with \\ `A02:` comments. You can test your implementation using the command `./a.out 2`.

You should use std::atomic for variables that are simultaneously read from and written to. (See 1.5) If your implementation leaks memory, remember to document this in your report. (See 1.4)

Task 3 Fine-Grained Set (0 Points) In Task 5, 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 (30 Points)

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 $0, 1, \ldots 1028$.

The benchmark will print the measured time in milliseconds.

In your report:

- Include the resulting tables
- Add graphs that compare the data structures with each other, regarding their performance.

  The x-axis should be the number of threads. The y-axis should be the throughput.
- The percentages of `ctn()` operations and range of input values will affect the performance. Explain how these elements affect the throughput of different data structures.

Task 5 Evaluation (10 Points)

In your report, describe under which circumstances a which synchronization mechanism might be the better choice over the alternatives. You should give reference your results from Task 5 where applicable.

In this task, you can optionally also include an example of when Course-Grained synchronization might be the best choice. You're allowed to use knowledge from other courses to motivate your answer.

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 2, 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 6 Fine-Grained Multiset (30 Points) Implement a concurrent multiset using Fine-Grained synchronization in `fine_multiset.hpp`. All code you might need to modify has been marked with `\\ A06:` comments.

Identify the linearization policy and document it in your 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_6()` function of the lab template. You can run the code of `task_6()` using the command `./a.out 6`.

4 Submission Instructions

The submission in Studium should have two files:

1. Your report as a PDF document.

   All text should be written on a computer. Graphs can be drawn digitally or by hand.

   If you have any feedback or suggestions regarding the assignment, you're welcome to include them in your report as well. This will not affect the grading of your submission.

2. A ZIP file with your programmed solution.

   If you're using the provided template, please zip the entire `lab0*` directory.

Each student must create their own individual solution. You are allowed to discuss your work with others, but you shouldn't share code.