Concurrent Data Structures Lab Assignment 3

Bernat Xandri Zaragoza

1. **Task 1:** Treiber Stack.

I implemented the Treiber stack following the code from the slides, and found a weird result. The code seems to work except for one instruction, where it fails. I tried implementing other tasks to help me debug the code, as well as adding verbose statements, but it seems to work correctly, as far as I could see. The error seems to arise from the comparation with the reference stack, but the Treiber stack seems to execute all the orders correctly, at least as far as I could see.
I also tried making a second implementation, based on the code provided in the book, instead of the slides, running into the same results. Given this situation, I think the code is correct but I'm making a mistake when logging the events in the test set. Therefore, I don't know how to fix it.

The linearization policy for my code would be always at the atomic Compare-And-Swap(CAS), which in C++ is done with a specific function applied to the checked object.
For the push method, the linearization point is the line 58, when the CAS operation successfully changes the `top` pointer to point to the new node. This is the exact moment when the new element is considered to be part of the stack.
For the pop method, the linearization point is in the line 87, when the CAS operation successfully updates the `top` pointer to point to the next node in the stack, therefore deleting the previous top.

2. **Task 2:** Lock-Free Set.

I implemented this algorithm using the template provided in the book.
For this implementation, the linearization policy would be:
For the add method, the linearization point would be the line 177, where the CAS method on the pred->next is successfully called. This is where the new node is effectively inserted into the set.
For the rmv method, it would be in the line 205, with the CAS. This is where the node is deleted from the list.
For the ctn it would be the line 225, where each node is being read.

3. **Task 3:** Benchmarking.

I tried to run the benchmarking. I had to update the makefile to make sure that it ran in C++17, since otherwise it was giving me errors. After this, I ran into the issue of getting a

Segmentation fault. When trying to debug this error using gdb, I get this message:

```
(gdb) bt
#0  __GI___libc_free (mem=0xde91) at malloc.c:3102
#1  0x000000000800666d in __gnu_cxx::new_allocator<OpWeights<SetOperator> >::deallocate (this=0x8023178,
    __p=<optimized out>) at /usr/include/c++/9/ext/new_allocator.h:119
#2  std::allocator_traits<std::allocator<OpWeights<SetOperator> > >::deallocate (__a=..., __n=<optimized out>,
    __p=<optimized out>) at /usr/include/c++/9/bits/alloc_traits.h:469
#3  std::_Vector_base<OpWeights<SetOperator>, std::allocator<OpWeights<SetOperator> > >::_M_deallocate (
    this=0x8023178, __n=<optimized out>, __p=<optimized out>) at /usr/include/c++/9/bits/stl_vector.h:351
#4  std::_Vector_base<OpWeights<SetOperator>, std::allocator<OpWeights<SetOperator> > >::~_Vector_base (
    this=0x8023178, __in_chrg=<optimized out>) at /usr/include/c++/9/bits/stl_vector.h:332
#5  std::vector<OpWeights<SetOperator>, std::allocator<OpWeights<SetOperator> > >::~vector (this=0x8023178,
    __in_chrg=<optimized out>) at /usr/include/c++/9/bits/stl_vector.h:680
#6  OpGenerator<SetOperator>::~OpGenerator (this=0x8023170, __in_chrg=<optimized out>) at src/monitoring.hpp:151
#7  bench::run_config<LockFreeSet> (set_name=<optimized out>, config=...) at src/bench.hpp:84
#8  0x0000000008003203 in bench::benchmark_set<LockFreeSet> (set_name=0x800b08e "LockFreeSet") at src/bench.hpp:34
#9  task_3 () at src/main.cpp:164
#10 0x0000000008002ab6 in main (argc=<optimized out>, argv=<optimized out>) at src/main.cpp:245
```

Which makes me think that the issue is in the benchmarking code, and the memory allocation of a vector in the generator module. Since this is not part of the code that I wrote, I tried to fix it but it feels a bit over my head, so didn't succeed. I would appreciate an explanation on why this issue happens.

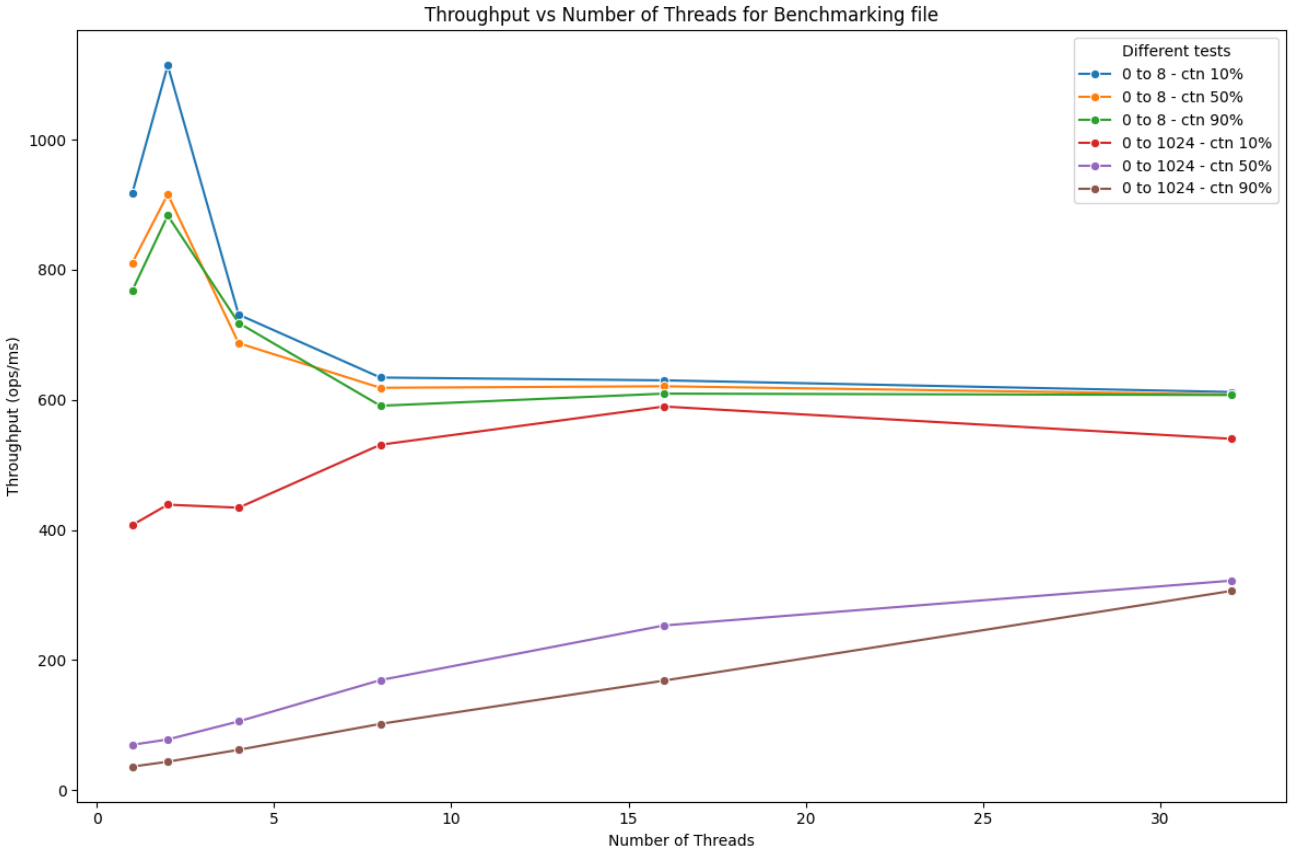
FEEDBACK

The table with the throughput calculated is as following:

values	ctn %	add [%]	rmv [%]	threads	time [ms]	ops	throughput
0..8	10	81	9	1	0.5449	500	917.599560
0..8	10	81	9	2	0.8977	1000	1113.95789
0..8	10	81	9	4	2.7361	2000	730.967435
0..8	10	81	9	8	6.3071	4000	634.205895
0..8	10	81	9	16	12.7000	8000	629.921260
0..8	10	81	9	32	26.1450	16000	611.971696
0..8	50	45	5	1	0.6169	500	810.504134
0..8	50	45	5	2	1.0920	1000	915.750916
0..8	50	45	5	4	2.9109	2000	687.072727
0..8	50	45	5	8	6.4678	4000	618.448313
0..8	50	45	5	16	12.8862	8000	620.819171
0..8	50	45	5	32	26.3149	16000	608.020551
0..8	90	9	1	1	0.6511	500	767.931193
0..8	90	9	1	2	1.1321	1000	883.314195
0..8	90	9	1	4	2.7849	2000	718.158641
0..8	90	9	1	8	6.7700	4000	590.841950
0..8	90	9	1	16	13.1260	8000	609.477373
0..8	90	9	1	32	26.3440	16000	607.348922
0..1024	10	81	9	1	1.2271	500	407.464754
0..1024	10	81	9	2	2.2791	1000	438.769690
0..1024	10	81	9	4	4.6069	2000	434.131412
0..1024	10	81	9	8	7.5342	4000	530.912373

0..1024	10	81	9	16	13.5669	8000	589.670448
0..1024	10	81	9	32	29.6267	16000	540.053398
0..1024	50	45	5	1	7.1772	500	69.665050
0..1024	50	45	5	2	12.8459	1000	77.845850
0..1024	50	45	5	4	18.9363	2000	105.617254
0..1024	50	45	5	8	23.6528	4000	169.113171
0..1024	50	45	5	16	31.6079	8000	253.101282
0..1024	50	45	5	32	49.7002	16000	321.930294
0..1024	90	9	1	1	13.8362	500	36.137090
0..1024	90	9	1	2	22.9080	1000	43.652872
0..1024	90	9	1	4	32.2732	2000	61.970923
0..1024	90	9	1	8	39.2949	4000	101.794380
0..1024	90	9	1	16	47.5239	8000	168.336353
0..1024	90	9	1	32	52.2603	16000	306.159743

Once we obtain these values, we can make a representation for this data, in order to be able to actually extrapolate the results into some general conclusions (I did this with Python and not in C++, since I already uninstalled my environment for C++):



We can observe that this set performs well with the 0 to 8 tests, but quickly gets a worse performance when using 0 to 1024. Therefore, we could say that for examples with high concurrency, we could assume that LazySet or OptimisticSet might be preferable based on the benchmark