

CS 142 Home Work #2

Timofey Malko

October 30, 2023

lib.rs

```
1 mod stack;
2 mod atomic_stack;
3 mod mutex_stack;
4 mod verify;
5
6 pub const N: usize = 1_000_000;
```

stack.rs

```
1 use super::N;
2
3 #[repr(C)]
4 #[derive(Debug)]
5 pub struct Stack {
6     head: usize,
7     data: [i32; N],
8 }
9 // A naive implementation of a stack
10 // its mixed with C compatability
11 impl Stack {
12     #[no_mangle]
13     pub extern "C" fn new_stack() -> Self {
14         Self {
15             head: 0,
16             data: [i32::MAX; N],
17         }
18     }
19
20     #[no_mangle]
21     pub extern "C" fn peak(&self) -> i32 {
22         self.data[self.head - 1]
23     }
24
25     #[no_mangle]
26     pub extern "C" fn push(&mut self, el: i32) {
27         self.data[self.head] = el;
28         self.head += 1;
29     }
30
31     #[no_mangle]
32     pub extern "C" fn pop(&mut self) -> i32 {
33         self.head -= 1;
34         self.data[self.head]
35     }
36 }
37
38 #[no_mangle]
39 pub extern "C" fn verify_stack(stack: &Stack, total: usize) -> bool {
40     super::verify::verify(&stack.data, stack.head, total)
41 }
```

```

42
43 #[cfg(test)]
44 mod tests {
45     use super::*;
46
47     #[test]
48     fn it_works() {
49         let mut stack = Stack::new_stack();
50         stack.push(1);
51         assert_eq!(stack.peak(), 1);
52         stack.push(4);
53         assert_eq!(stack.peak(), 4);
54         assert_eq!(stack.pop(), 4);
55         assert_eq!(stack.pop(), 1);
56     }
57 }

```

atomic_stack.rs

```

1 use super::N;
2 use std::sync::atomic::{AtomicUsize, Ordering};
3
4 #[derive(Debug)]
5 pub struct AtomicStack {
6     head: AtomicUsize,
7     data: [i32; N],
8 }
9
10 // Like a naive stack, but with atomic value for the head, utilizing fetch_add operation for
11 // thread safety
12 impl AtomicStack {
13     fn new() -> Self {
14         Self {
15             head: AtomicUsize::new(0),
16             data: [i32::MAX; N],
17         }
18     }
19
20     fn peak(&self) -> i32 {
21         self.data[self.head.load(Ordering::Acquire) - 1]
22     }
23
24     fn push(&mut self, el: i32) {
25         self.data[self.head.fetch_add(1, Ordering::SeqCst)] = el;
26     }
27
28     fn pop(&mut self) -> i32 {
29         self.data[self.head.fetch_sub(1, Ordering::SeqCst) - 1]
30     }
31 }
32
33 // A compatibility stuff for C
34 mod c_compat {
35     use super::*;
36
37     #[no_mangle]
38     pub extern "C" fn atomic_stack_new() -> *mut AtomicStack {
39         Box::into_raw(Box::new(AtomicStack::new()))
40     }
41
42     #[no_mangle]
43     pub extern "C" fn atomic_stack_drop(ptr: *mut AtomicStack) {
44         if ptr.is_null() {
45             return;
46         }
47         unsafe {

```

```

47         let _ = Box::from_raw(ptr);
48     }
49 }
50
51 #[no_mangle]
52 pub extern "C" fn atomic_stack_peak(ptr: *mut AtomicStack) -> i32 {
53     let stack = unsafe {
54         assert!(!ptr.is_null());
55         &mut *ptr
56     };
57     stack.peak()
58 }
59
60 #[no_mangle]
61 pub extern "C" fn atomic_stack_push(ptr: *mut AtomicStack, elem: i32) {
62     let stack = unsafe {
63         assert!(!ptr.is_null());
64         &mut *ptr
65     };
66     stack.push(elem)
67 }
68
69 #[no_mangle]
70 pub extern "C" fn atomic_stack_pop(ptr: *mut AtomicStack) -> i32 {
71     let stack = unsafe {
72         assert!(!ptr.is_null());
73         &mut *ptr
74     };
75     stack.pop()
76 }
77
78 #[no_mangle]
79 pub extern "C" fn verify_atomic_stack(stack: &AtomicStack, total: usize) -> bool {
80     crate::verify::verify(&stack.data, stack.head.load(Ordering::Acquire), total)
81 }
82 }
83
84 #[cfg(test)]
85 mod tests {
86     use super::*;
87
88     #[test]
89     fn it_works() {
90         let mut stack = AtomicStack::new();
91         stack.push(1);
92         assert_eq!(stack.peak(), 1);
93         stack.push(4);
94         assert_eq!(stack.peak(), 4);
95         assert_eq!(stack.pop(), 4);
96         assert_eq!(stack.pop(), 1);
97     }
98 }

```

mutex_stack.rs

```

1 use std::sync::Mutex;
2
3 use super::N;
4
5 #[derive(Debug)]
6 pub struct MutexStack {
7     head: Mutex<usize>,
8     data: [i32; N],
9 }
10
11 // A stack with head wrapped in a mutex

```

```

12 // Exactly like naive stack but the need to unlock head for each operation
13 impl MutexStack {
14     fn new() -> Self {
15         Self {
16             head: Mutex::new(0),
17             data: [i32::MAX; N],
18         }
19     }
20
21     fn peak(&self) -> i32 {
22         self.data[*self.head.lock().unwrap() - 1]
23     }
24
25     fn push(&mut self, el: i32) {
26         let mut head = self.head.lock().unwrap();
27         self.data[*head] = el;
28         *head += 1;
29     }
30
31     fn pop(&mut self) -> i32 {
32         let mut head = self.head.lock().unwrap();
33         *head -= 1;
34         self.data[*head]
35     }
36 }
37
38 // A compatability stuff for C
39 mod c_compat {
40     use super::*;
41
42     #[no_mangle]
43     pub extern "C" fn mutex_stack_new() -> *mut MutexStack {
44         Box::into_raw(Box::new(MutexStack::new()))
45     }
46
47     #[no_mangle]
48     pub extern "C" fn mutex_stack_drop(ptr: *mut MutexStack) {
49         if ptr.is_null() {
50             return;
51         }
52         unsafe {
53             let _ = Box::from_raw(ptr);
54         }
55     }
56
57     #[no_mangle]
58     pub extern "C" fn mutex_stack_peak(ptr: *mut MutexStack) -> i32 {
59         let stack = unsafe {
60             assert!(!ptr.is_null());
61             &mut *ptr
62         };
63         stack.peak()
64     }
65
66     #[no_mangle]
67     pub extern "C" fn mutex_stack_push(ptr: *mut MutexStack, elem: i32) {
68         let stack = unsafe {
69             assert!(!ptr.is_null());
70             &mut *ptr
71         };
72         stack.push(elem)
73     }
74
75     #[no_mangle]
76     pub extern "C" fn mutex_stack_pop(ptr: *mut MutexStack) -> i32 {

```

```

77         let stack = unsafe {
78             assert!(!ptr.is_null());
79             &mut *ptr
80         };
81         stack.pop()
82     }
83
84     #[no_mangle]
85     pub extern "C" fn verify_mutex_stack(stack: &MutexStack, total: usize) -> bool {
86         crate::verify::verify(&stack.data, *stack.head.lock().unwrap(), total)
87     }
88 }
89
90 #[cfg(test)]
91 mod tests {
92     use super::*;
93
94     #[test]
95     fn it_works() {
96         let mut stack = MutexStack::new();
97         stack.push(1);
98         assert_eq!(stack.peak(), 1);
99         stack.push(4);
100         assert_eq!(stack.peak(), 4);
101         assert_eq!(stack.pop(), 4);
102         assert_eq!(stack.pop(), 1);
103     }
104 }

```

verify.rs

```

1 use itertools::Itertools;
2 use rand::Rng;
3 use std::cmp::Ordering;
4 use std::collections::HashMap;
5 use std::fs::create_dir_all;
6 use std::fs::File, io::Write;
7
8 use super::N;
9
10 /// A verification function checks if an array have all numbers from 0 to its size
11 /// Logs missing or duplicated values
12 ///
13 pub fn verify(data: &[i32; N], head: usize, total: usize) -> bool {
14     let mut data = data[0..total].to_vec();
15     data.sort();
16
17     let counts: HashMap<&i32, usize> = data.iter().counts();
18     let mut outliers = counts.iter().filter(|(&x)| x != 1).collect_vec();
19     outliers.sort_by(|&x, &y| x.0.cmp(y.0));
20
21     if !outliers.is_empty() { //catches majority of the errors
22         let mut i = 0;
23         let missing = (0..total as i32).into_iter().filter_map(|num| loop {
24             match data[i].cmp(&num) {
25                 Ordering::Less => {
26                     i += 1;
27                 }
28                 Ordering::Equal => {
29                     i += 1;
30                     break None;
31                 }
32                 Ordering::Greater => {
33                     break Some(num);
34                 }
35             }
36         })

```

```

36     }).collect_vec();
37     create_dir_all("./Log").unwrap();
38     let mut file = File::create(format!(
39         "./Log/FaileLog{}.txt",
40         rand::thread_rng().gen_range(0..100)
41     ))
42     .unwrap();
43     file.write_fmt(format_args!(
44         "head:␣{}\nOutliers\n{:?}\nMissing\n{:?}\n---\n",
45         head, outliers, missing
46     ))
47     .unwrap();
48 }
49 outliers.is_empty()
50 }

```

bench_stack.cpp

```

1  #include "stack_lib.h"
2  #include <chrono>
3  #include <cstdlib>
4  #include <iostream>
5  #include <thread>
6
7  uintptr_t global_limit;
8
9  //MACROS left the chat
10 //The same functions are repeated for simple, atomic, and mutex stack because of my poor
    choices
11
12 #pragma region SimpleStack
13 // pushes specific numbers into stack
14 void push_numbers(stacks::Stack *stack, int16_t offset, int16_t step) {
15     for (int i = offset; i < global_limit; i += step) {
16         stacks::push(stack, i);
17     }
18 }
19
20 // bench simple stack for a single thread
21 size_t bench_single(size_t num_benches, bool verbose) {
22     size_t avrg_duration = 0;
23     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
24         auto stack = stacks::new_stack();
25         auto start = std::chrono::high_resolution_clock::now();
26         push_numbers(&stack, 0, 1);
27         auto stop = std::chrono::high_resolution_clock::now();
28         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
29
30         if (verbose) { //skip some print
31             std::cout << (stacks::verify_stack(&stack, global_limit) ? "Correct" : "Smth␣is␣
wrong") << std::endl;
32         }
33         avrg_duration += duration.count();
34     }
35     return avrg_duration / num_benches;
36 }
37
38 // bench simple stack for a double thread
39 size_t bench_thread(size_t num_benches, bool verbose) {
40     size_t avrg_duration = 0;
41     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
42         auto stack = stacks::new_stack();
43         auto start = std::chrono::high_resolution_clock::now();
44
45         std::thread th1(push_numbers, &stack, 0, 2);
46         std::thread th2(push_numbers, &stack, 1, 2);

```

```

47     th1.join();
48     th2.join();
49
50     auto stop = std::chrono::high_resolution_clock::now();
51     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
52
53     if (verbose) {
54         std::cout << (stacks::verify_stack(&stack, global_limit) ? "Correct" : "Smth is
wrong") << std::endl;
55     }
56     avrg_duration += duration.count();
57 }
58 return avrg_duration / num_benches;
59 }
60 #pragma endregion SimpleStack
61
62 #pragma region MutexStack
63 // pushes specific numbers into stack
64 void push_mutex_numbers(stacks::MutexStack *stack, int16_t offset, int16_t step) {
65     for (int i = offset; i < global_limit; i += step) {
66         stacks::mutex_stack_push(stack, i);
67     }
68 }
69
70 // bench mutex stack for a single thread
71 size_t bench_mutex_single(size_t num_benches, bool verbose) {
72     size_t avrg_duration = 0;
73     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
74         auto stack = stacks::mutex_stack_new();
75         auto start = std::chrono::high_resolution_clock::now();
76         push_mutex_numbers(stack, 0, 1);
77         auto stop = std::chrono::high_resolution_clock::now();
78         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
79
80         if (verbose) {
81             std::cout << (stacks::verify_mutex_stack(stack, global_limit) ? "Correct" : "
Smth is wrong") << std::endl;
82         }
83         avrg_duration += duration.count();
84
85         mutex_stack_drop(stack);
86     }
87     return avrg_duration / num_benches;
88 }
89
90 // bench mutex stack for a double thread
91 size_t bench_mutex_thread(size_t num_benches, bool verbose) {
92     size_t avrg_duration = 0;
93     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
94         auto stack = stacks::mutex_stack_new();
95         auto start = std::chrono::high_resolution_clock::now();
96
97         std::thread th1(push_mutex_numbers, stack, 0, 2);
98         std::thread th2(push_mutex_numbers, stack, 1, 2);
99         th1.join();
100        th2.join();
101
102        auto stop = std::chrono::high_resolution_clock::now();
103        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
104
105        if (verbose) {
106            std::cout << (stacks::verify_mutex_stack(stack, global_limit) ? "Correct" : "
Smth is wrong") << std::endl;
107        }
108        avrg_duration += duration.count();

```

```

109         mutex_stack_drop(stack);
110     }
111     return avrg_duration / num_benches;
112 }
113
114 #pragma endregion MutexStack
115
116 #pragma region AtomicStack
117 // pushes specific numbers into stack
118 void push_atomic_numbers(stacks::AtomicStack *stack, int16_t offset, int16_t step) {
119     for (int i = offset; i < global_limit; i += step) {
120         stacks::atomic_stack_push(stack, i);
121     }
122 }
123
124 // bench atomic stack for a single thread
125 size_t bench_atomic_single(size_t num_benches, bool verbose) {
126     size_t avrg_duration = 0;
127     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
128         auto stack = stacks::atomic_stack_new();
129         auto start = std::chrono::high_resolution_clock::now();
130         push_atomic_numbers(stack, 0, 1);
131         auto stop = std::chrono::high_resolution_clock::now();
132         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
133
134         if (verbose) {
135             std::cout << (stacks::verify_atomic_stack(stack, global_limit) ? "Correct" : "
136             Smth is wrong") << std::endl;
137         }
138         avrg_duration += duration.count();
139     }
140     return avrg_duration / num_benches;
141 }
142
143 // bench atomic stack for a double thread
144 size_t bench_atomic_thread(size_t num_benches, bool verbose) {
145     size_t avrg_duration = 0;
146     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
147         auto stack = stacks::atomic_stack_new();
148         auto start = std::chrono::high_resolution_clock::now();
149
150         std::thread th1(push_atomic_numbers, stack, 0, 2);
151         std::thread th2(push_atomic_numbers, stack, 1, 2);
152         th1.join();
153         th2.join();
154
155         auto stop = std::chrono::high_resolution_clock::now();
156         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
157
158         if (verbose) {
159             std::cout << (stacks::verify_atomic_stack(stack, global_limit) ? "Correct" : "
160             Smth is wrong") << std::endl;
161         }
162         avrg_duration += duration.count();
163     }
164     return avrg_duration / num_benches;
165 }
166 #pragma endregion AtomicStack
167
168 int main(int argc, char *argv[]) {
169     size_t num_benches;
170     bool verbose = true;
171
172     // multiple ways to get input
173     if (argc >= 3) {

```



```

172     num_benches = (size_t)atoi(argv[1]);
173     global_limit = (uintptr_t)atoi(argv[2]);
174     verbose = false;
175 } else {
176     std::cin >> num_benches;
177     std::cin >> global_limit;
178 }
179
180 global_limit = stacks::N < global_limit ? stacks::N : global_limit;
181
182 //Perform benches
183 auto single = bench_single(num_benches, verbose);
184 auto thread = bench_thread(num_benches, verbose);
185 auto mutex_single = bench_mutex_single(num_benches, verbose);
186 auto mutex_thread = bench_mutex_thread(num_benches, verbose);
187 auto atomic_single = bench_atomic_single(num_benches, verbose);
188 auto atomic_thread = bench_atomic_thread(num_benches, verbose);
189
190 if (verbose) {
191     std::cout << "Simple" << std::endl;
192     std::cout << "SINGLE_THREAD_Avg_duration:\t" << single << std::endl;
193     std::cout << "DOUBLE_THREAD_Avg_duration:\t" << thread << std::endl;
194     std::cout << "Mutex" << std::endl;
195     std::cout << "SINGLE_THREAD_Avg_duration:\t" << mutex_single << std::endl;
196     std::cout << "DOUBLE_THREAD_Avg_duration:\t" << mutex_thread << std::endl;
197     std::cout << "Atomic" << std::endl;
198     std::cout << "SINGLE_THREAD_Avg_duration:\t" << atomic_single << std::endl;
199     std::cout << "DOUBLE_THREAD_Avg_duration:\t" << atomic_thread << std::endl;
200 } else {
201     std::cout << single << "\t" << thread << "\t" << mutex_single << "\t" << mutex_thread
202 << "\t" << atomic_single << "\t" << atomic_thread << std::endl;
203 }
204 return 0;
}

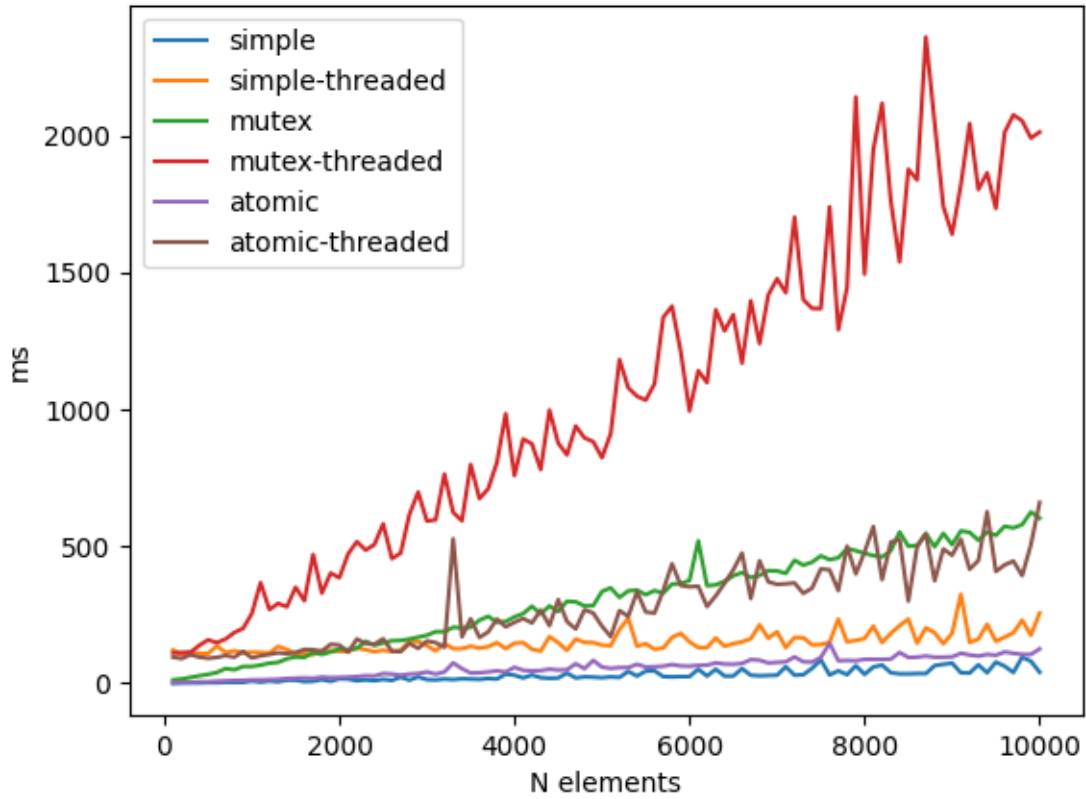
```

plot.py

```

1 import matplotlib.pyplot as plt
2 import subprocess
3
4 labels = ["simple", "simple-threaded", "mutex", "mutex-threaded", "atomic", "atomic-threaded"]
5 data = []
6 x = []
7
8 # benches for multiple values of N
9 for i in range(1, 101):
10     print(i, "%")
11     data.append([int(x) for x in (subprocess.run(["./atomic", str(5), str(i*100)],
12         capture_output=True).stdout.split())])
13     x.append(i*100)
14
15 plt.plot(x, data)
16 plt.legend(labels)
17 plt.axis
18 plt.xlabel("N_elements")
19 plt.ylabel("ms")
20 plt.draw()
21 plt.savefig('plot.png', dpi=100)

```



As expected single threaded cases have better results compared to relative double threaded cases. The simple approach provides the best results, though it results in incorrect states of the stack in multithreaded approach (see logs). The worst results are from mutex-stack which proves that it has greatest overhead. For the small N atomic is similar in performance to simple stack. As N increases the atomic shows some overhead mostly in threaded scenario and there is no errors (see lack of logs).