CS 142 Home Work #2

Timofey Malko

October 30, 2023

lib.rs

```rust
mod stack;
mod atomic_stack;
mod mutex_stack;
mod verify;

pub const N: usize = 1_000_000;
```

stack.rs

```rust
use super::N;

#[repr(C)]
#[derive(Debug)]
pub struct Stack {
    head: usize,
    data: [i32; N],
}
// A naive implementation of a stack
// its mixed with C compatability
impl Stack {
    #[no_mangle]
    pub extern "C" fn new_stack() -> Self {
        Self {
            head: 0,
            data: [i32::MAX; N],
        }
    }

    #[no_mangle]
    pub extern "C" fn peak(&self) -> i32 {
        self.data[self.head - 1]
    }

    #[no_mangle]
    pub extern "C" fn push(&mut self, el: i32) {
        self.data[self.head] = el;
        self.head += 1;
    }

    #[no_mangle]
    pub extern "C" fn pop(&mut self) -> i32 {
        self.head -= 1;
        self.data[self.head]
    }
}

#[no_mangle]
pub extern "C" fn verify_stack(stack: &Stack, total: usize) -> bool {
    super::verify::verify(&stack.data, stack.head, total)
}
```

```rust

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let mut stack = Stack::new_stack();
        stack.push(1);
        assert_eq!(stack.peak(), 1);
        stack.push(4);
        assert_eq!(stack.peak(), 4);
        assert_eq!(stack.pop(), 4);
        assert_eq!(stack.pop(), 1);
    }
}
```

atomic˙stack.rs

```rust
use super::N;
use std::sync::atomic::{AtomicUsize, Ordering};

#[derive(Debug)]
pub struct AtomicStack {
    head: AtomicUsize,
    data: [i32; N],
}

// Like a naive stack, but with atomic value for the head, utilizing fetch_add operation for
    thread safety
impl AtomicStack {
    fn new() -> Self {
        Self {
            head: AtomicUsize::new(0),
            data: [i32::MAX; N],
        }
    }

    fn peak(&self) -> i32 {
        self.data[self.head.load(Ordering::Acquire) - 1]
    }

    fn push(&mut self, el: i32) {
        self.data[self.head.fetch_add(1, Ordering::SeqCst)] = el;
    }

    fn pop(&mut self) -> i32 {
        self.data[self.head.fetch_sub(1, Ordering::SeqCst) - 1]
    }
}

// A compatability stuff for C
mod c_compat {
    use super::*;

    #[no_mangle]
    pub extern "C" fn atomic_stack_new() -> *mut AtomicStack {
        Box::into_raw(Box::new(AtomicStack::new()))
    }

    #[no_mangle]
    pub extern "C" fn atomic_stack_drop(ptr: *mut AtomicStack) {
        if ptr.is_null() {
            return;
        }
        unsafe {
```

```
            let _ = Box::from_raw(ptr);
        }
    }

    #[no_mangle]
    pub extern "C" fn atomic_stack_peak(ptr: *mut AtomicStack) -> i32 {
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.peak()
    }

    #[no_mangle]
    pub extern "C" fn atomic_stack_push(ptr: *mut AtomicStack, elem: i32) {
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.push(elem)
    }

    #[no_mangle]
    pub extern "C" fn atomic_stack_pop(ptr: *mut AtomicStack) -> i32 {
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.pop()
    }

    #[no_mangle]
    pub extern "C" fn verify_atomic_stack(stack: &AtomicStack, total: usize) -> bool {
        crate::verify::verify(&stack.data, stack.head.load(Ordering::Acquire), total)
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let mut stack = AtomicStack::new();
        stack.push(1);
        assert_eq!(stack.peak(), 1);
        stack.push(4);
        assert_eq!(stack.peak(), 4);
        assert_eq!(stack.pop(), 4);
        assert_eq!(stack.pop(), 1);
    }
}
```

mutex˙stack.rs

```
use std::sync::Mutex;

use super::N;

#[derive(Debug)]
pub struct MutexStack {
    head: Mutex<usize>,
    data: [i32; N],
}

// A stack with head wraped in a mutex
```

```rust
// Exactly like naive stack but the need to unlock head for each operation
impl MutexStack {
    fn new() -> Self {
        Self {
            head: Mutex::new(0),
            data: [i32::MAX; N],
        }
    }

    fn peak(&self) -> i32 {
        self.data[*self.head.lock().unwrap() - 1]
    }

    fn push(&mut self, el: i32) {
        let mut head = self.head.lock().unwrap();
        self.data[*head] = el;
        *head += 1;
    }

    fn pop(&mut self) -> i32 {
        let mut head = self.head.lock().unwrap();
        *head -= 1;
        self.data[*head]
    }
}

// A compatability stuff for C
mod c_compat {
    use super::*;

    #[no_mangle]
    pub extern "C" fn mutex_stack_new() -> *mut MutexStack {
        Box::into_raw(Box::new(MutexStack::new()))
    }

    #[no_mangle]
    pub extern "C" fn mutex_stack_drop(ptr: *mut MutexStack) {
        if ptr.is_null() {
            return;
        }
        unsafe {
            let _ = Box::from_raw(ptr);
        }
    }

    #[no_mangle]
    pub extern "C" fn mutex_stack_peak(ptr: *mut MutexStack) -> i32 {
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.peak()
    }

    #[no_mangle]
    pub extern "C" fn mutex_stack_push(ptr: *mut MutexStack, elem: i32) {
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.push(elem)
    }

    #[no_mangle]
    pub extern "C" fn mutex_stack_pop(ptr: *mut MutexStack) -> i32 {
```

```
        let stack = unsafe {
            assert!(!ptr.is_null());
            &mut *ptr
        };
        stack.pop()
    }

    #[no_mangle]
    pub extern "C" fn verify_mutex_stack(stack: &MutexStack, total: usize) -> bool {
        crate::verify::verify(&stack.data, *stack.head.lock().unwrap(), total)
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        let mut stack = MutexStack::new();
        stack.push(1);
        assert_eq!(stack.peak(), 1);
        stack.push(4);
        assert_eq!(stack.peak(), 4);
        assert_eq!(stack.pop(), 4);
        assert_eq!(stack.pop(), 1);
    }
}
```

verify.rs

```
use itertools::Itertools;
use rand::Rng;
use std::cmp::Ordering;
use std::collections::HashMap;
use std::fs::create_dir_all;
use std::{fs::File, io::Write};

use super::N;

/// A verification function chechks if an array have all numbers from 0 to its size
/// Logs missing or duplicated values
///
pub fn verify(data: &[i32; N], head: usize, total: usize) -> bool {
    let mut data = data[0..total].to_vec();
    data.sort();

    let counts: HashMap<&i32, usize> = data.iter().counts();
    let mut outliers = counts.iter().filter(|(_, &x)| x != 1).collect_vec();
    outliers.sort_by(|x, y| x.0.cmp(y.0));

    if !outliers.is_empty() { //catches majority of the errors
        let mut i = 0;
        let missing = (0..total as i32).into_iter().filter_map(|num| loop {
            match data[i].cmp(&num) {
                Ordering::Less => {
                    i += 1;
                }
                Ordering::Equal => {
                    i += 1;
                    break None;
                }
                Ordering::Greater => {
                    break Some(num);
                }
            }
```

```
        }).collect_vec();
        create_dir_all("./Log").unwrap();
        let mut file = File::create(format!(
            "./Log/FaileLog{}.txt",
            rand::thread_rng().gen_range(0..100)
        ))
        .unwrap();
        file.write_fmt(format_args!(
            "head: {}\nOutliers\n{:?}\nMisssing\n{:?}\n---\n",
            head, outliers, missing
        ))
        .unwrap();
    }
    outliers.is_empty()
}
```

bench˙stack.cpp

```
#include "stack_lib.h"
#include <chrono>
#include <cstdlib>
#include <iostream>
#include <thread>

uintptr_t global_limit;

//MACROS left the chat
//The same functions are repeated for simple, atomic, and mutex stack because of my poor choises

#pragma region SimpleStack
// pushes specific numbers into stack
void push_numbers(stacks::Stack *stack, int16_t offset, int16_t step) {
    for (int i = offset; i < global_limit; i += step) {
        stacks::push(stack, i);
    }
}

// bench simple stack for a single thread
size_t bench_single(size_t num_benches, bool verbose) {
    size_t avrg_duration = 0;
    for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
        auto stack = stacks::new_stack();
        auto start = std::chrono::high_resolution_clock::now();
        push_numbers(&stack, 0, 1);
        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

        if (verbose) { //skip some print
            std::cout << (stacks::verify_stack(&stack, global_limit) ? "Correct" : "Smth is wrong") << std::endl;
        }
        avrg_duration += duration.count();
    }
    return avrg_duration / num_benches;
}

// bench simple stack for a double thread
size_t bench_thread(size_t num_benches, bool verbose) {
    size_t avrg_duration = 0;
    for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
        auto stack = stacks::new_stack();
        auto start = std::chrono::high_resolution_clock::now();

        std::thread th1(push_numbers, &stack, 0, 2);
        std::thread th2(push_numbers, &stack, 1, 2);
```

```
        th1.join();
        th2.join();

        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

        if (verbose) {
            std::cout << (stacks::verify_stack(&stack, global_limit) ? "Correct" : "Smth is
    wrong") << std::endl;
        }
        avrg_duration += duration.count();
    }
    return avrg_duration / num_benches;
}
#pragma endregion SimpleStack

#pragma region MutexStack
// pushes specific numbers into stack
void push_mutex_numbers(stacks::MutexStack *stack, int16_t offset, int16_t step) {
    for (int i = offset; i < global_limit; i += step) {
        stacks::mutex_stack_push(stack, i);
    }
}

// bench mutex stack for a single thread
size_t bench_mutex_single(size_t num_benches, bool verbose) {
    size_t avrg_duration = 0;
    for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
        auto stack = stacks::mutex_stack_new();
        auto start = std::chrono::high_resolution_clock::now();
        push_mutex_numbers(stack, 0, 1);
        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

        if (verbose) {
            std::cout << (stacks::verify_mutex_stack(stack, global_limit) ? "Correct" : "
    Smth is wrong") << std::endl;
        }
        avrg_duration += duration.count();

        mutex_stack_drop(stack);
    }
    return avrg_duration / num_benches;
}

// bench mutex stack for a double thread
size_t bench_mutex_thread(size_t num_benches, bool verbose) {
    size_t avrg_duration = 0;
    for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
        auto stack = stacks::mutex_stack_new();
        auto start = std::chrono::high_resolution_clock::now();

        std::thread th1(push_mutex_numbers, stack, 0, 2);
        std::thread th2(push_mutex_numbers, stack, 1, 2);
        th1.join();
        th2.join();

        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);

        if (verbose) {
            std::cout << (stacks::verify_mutex_stack(stack, global_limit) ? "Correct" : "
    Smth is wrong") << std::endl;
        }
        avrg_duration += duration.count();
```

```
109
110         mutex_stack_drop(stack);
111     }
112     return avrg_duration / num_benches;
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" : "
    Smth␣is␣wrong") << std::endl;
136         }
137         avrg_duration += duration.count();
138     }
139     return avrg_duration / num_benches;
140 }
141
142 // bench atomic stack for a double thread
143 size_t bench_atomic_thread(size_t num_benches, bool verbose) {
144     size_t avrg_duration = 0;
145     for (uintptr_t i = 0; i < num_benches; i++) { // average over multiple runs
146         auto stack = stacks::atomic_stack_new();
147         auto start = std::chrono::high_resolution_clock::now();
148
149         std::thread th1(push_atomic_numbers, stack, 0, 2);
150         std::thread th2(push_atomic_numbers, stack, 1, 2);
151         th1.join();
152         th2.join();
153
154         auto stop = std::chrono::high_resolution_clock::now();
155         auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
156
157         if (verbose) {
158             std::cout << (stacks::verify_atomic_stack(stack, global_limit) ? "Correct" : "
    Smth␣is␣wrong") << std::endl;
159         }
160         avrg_duration += duration.count();
161     }
162     return avrg_duration / num_benches;
163 }
164 #pragma endregion AtomicStack
165
166 int main(int argc, char *argv[]) {
167     size_t num_benches;
168     bool verbose = true;
169
170     // multiple ways to get input
171     if (argc >= 3) {
```

```
        num_benches = (size_t)atoi(argv[1]);
        global_limit = (uintptr_t)atoi(argv[2]);
        verbose = false;
    } else {
        std::cin >> num_benches;
        std::cin >> global_limit;
    }

    global_limit = stacks::N < global_limit ? stacks::N : global_limit;

    //Perform benches
    auto single = bench_single(num_benches, verbose);
    auto thread = bench_thread(num_benches, verbose);
    auto mutex_single = bench_mutex_single(num_benches, verbose);
    auto mutex_thread = bench_mutex_thread(num_benches, verbose);
    auto atomic_single = bench_atomic_single(num_benches, verbose);
    auto atomic_thread = bench_atomic_thread(num_benches, verbose);

    if (verbose) {
        std::cout << "Simple" << std::endl;
        std::cout << "SINGLE␣THREAD␣Avrg␣duration:␣\t" << single << std::endl;
        std::cout << "DOUBLE␣THREAD␣Avrg␣duration:␣\t" << thread << std::endl;
        std::cout << "Mutex" << std::endl;
        std::cout << "SINGLE␣THREAD␣Avrg␣duration:␣\t" << mutex_single << std::endl;
        std::cout << "DOUBLE␣THREAD␣Avrg␣duration:␣\t" << mutex_thread << std::endl;
        std::cout << "Atomic" << std::endl;
        std::cout << "SINGLE␣THREAD␣Avrg␣duration:␣\t" << atomic_single << std::endl;
        std::cout << "DOUBLE␣THREAD␣Avrg␣duration:␣\t" << atomic_thread << std::endl;
    } else {
        std::cout << single << "␣" << thread << "␣" << mutex_single << "␣" << mutex_thread
    << "␣" << atomic_single << "␣" << atomic_thread << std::endl;
    }
    return 0;
}
```

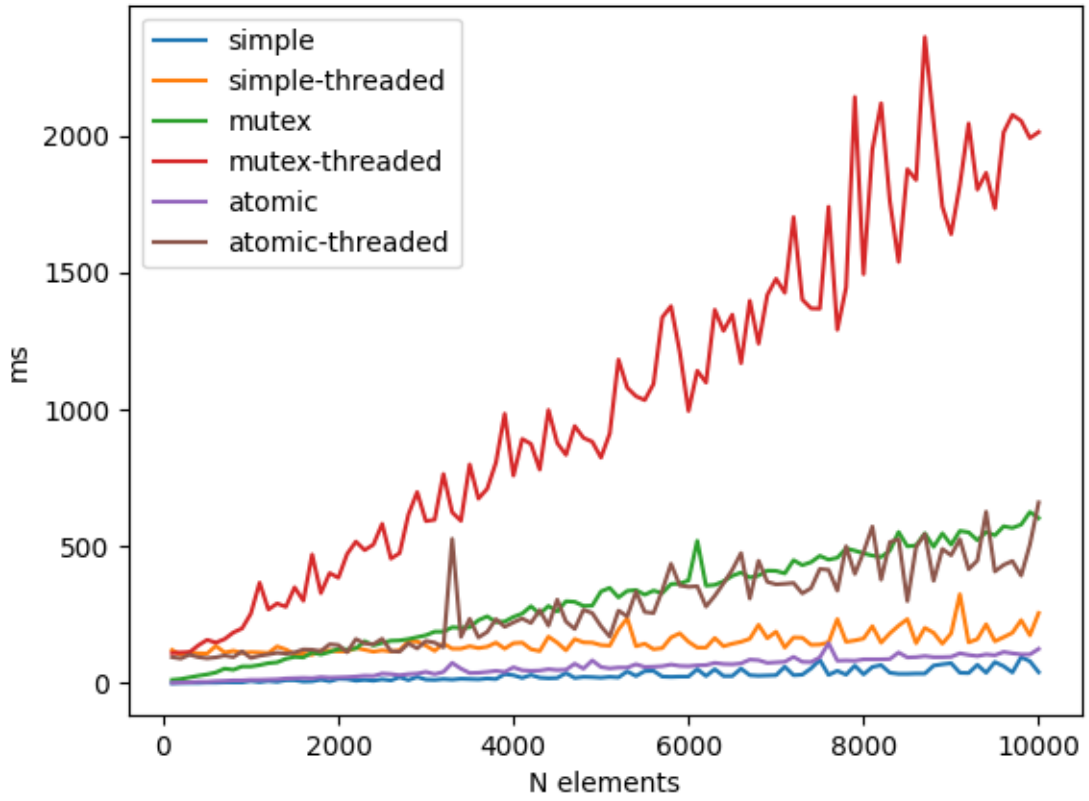
plot.py

```
import matplotlib.pyplot as plt
import subprocess

labels = ["simple", "simple-threaded", "mutex", "mutex-threaded", "atomic", "atomic-threaded
    "]
data = []
x = []

# benches for multiple values of N
for i in range(1, 101):
    print(i, "%")
    data.append([int(x) for x in (subprocess.run(["./atomic", str(5), str(i*100)],
    capture_output=True).stdout.split())])
    x.append(i*100)

plt.plot(x, data)
plt.legend(labels)
plt.axis
plt.xlabel("N␣elements")
plt.ylabel("ms")
plt.draw()
plt.savefig('plot.png', dpi=100)
```



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