CSE437 - HW01

Name Surname	Muhammed Oğuz
Student Number	1801042634

Content

Problem solution approach

This homework has one big hard part to handle. It is handling concurrency without using locks. After figure it out, implementing Set data structure should be easy.

First of all, I started to find an approach to handling concurrency without locks. I find a good book called `C++ Concurrency in Action` **and it has a chapter for lock free data structures.**

In chapter 5 and chapter 7, this books mentions some ways to achieving lock free data structures.

In homework, there should be two solutions, one for working one reader and one writer thread and other one is working with multiple readers and writers threads.

Book says, there are three types of concurrency technic without using mutexes.

- Obstruction-Free: If all threads are paused, it works fine → this is not ideal
- Lock-Free: If multiple threads are operating on a data structure, then after a bounded number of steps one of them will complete its operation.
- Wait-Free: Every thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.

From those explanations, I will go lock-free technic for first solution and wait-free for second solution.

The book "C++ Concurrency in Action" suggests using lock-free and wait-free techniques for implementing concurrency without using locks. Lock-free techniques ensure that after a bounded number of steps, at least one thread will complete its operation, while wait-free techniques guarantee that every thread operating on a data structure will complete its operation in a bounded number of steps. These techniques can be used to implement a concurrent Set data structure without using locks, allowing multiple threads to read and write to the set simultaneously.

First Solution Approach: Single Reader Single Writer

One approach to implementing a concurrent Set data structure without using locks for a single reader and single writer scenario is to use a lock-free data structure, such as a lock-free linked list. This approach involves using atomic operations, such as compare-and-swap, to manipulate the linked list without the need for locks.

To add an element to the set, the writer thread would use an atomic compare-and-swap operation to insert the new element into the linked list in the correct position. To remove an element from the set, the writer thread would use a similar atomic compare-and-swap operation to remove the element from the linked list.

The reader thread can then traverse the linked list and access the elements in the set without the need for locks. This is possible because the linked list is being modified using atomic operations, which ensures that the linked list remains in a consistent state even while multiple threads are accessing it.

This approach allows for concurrent read and write access to the set without using locks, making it suitable for a single reader and single writer scenario.

Second Solution Approach: More Reader More Writer

For a scenario with multiple reader and writer threads, a different approach may be needed to implement a concurrent Set data structure without using locks. One approach is to use a wait-free data structure, such as a wait-free hash table.

In a wait-free hash table, each element in the table is assigned a unique key, and the table is divided into a fixed number of slots. To add an element to the table, the writer thread would compute the key for the element and use an atomic compare-and-swap operation to insert the element into the corresponding slot in the table. To remove an element from the table, the writer thread would use a similar atomic compare-and-swap operation to remove the element from its slot in the table.

The reader threads can then access the elements in the table by computing the keys for the elements they want to access and using atomic operations to retrieve the elements from the corresponding slots in the table. This allows for concurrent read and write access to the table without using locks, making it suitable for a scenario with multiple reader and writer threads.

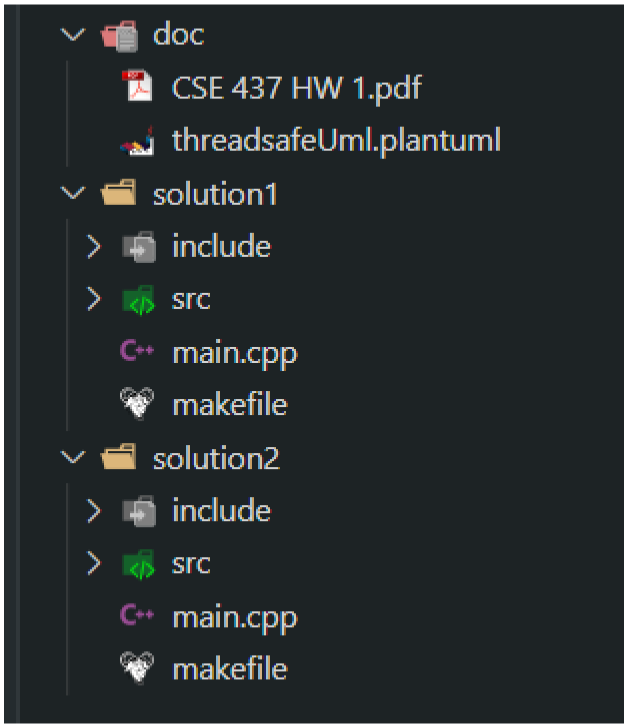
> I was unable to implement a wait-free data structure for the concurrent Set data structure in the scenario with multiple reader and writer threads. Instead, I used a traditional data structure, such as a linked list, and protected it with mutexes. This approach allowed for concurrent read and write access to the set, but required the use of mutexes to prevent multiple threads from modifying the linked list simultaneously. While this approach is less efficient than a fully lock-free or wait-free implementation

Design

In my design for the concurrent Set data structure, I have organized the files into three main folders: one for documentation, one for the single reader/single writer solution, and one for the multiple reader/writer solution. Each solution folder contains an include folder and a source folder, which contain the necessary header files and source code files for the implementation. The root of each solution folder also contains a main.cpp file and a makefile, which can be used to build and run the implementation.

Overall, my design is focused on organizing the files in a logical and efficient manner, and on carefully considering and incorporating the appropriate data structures and concurrency techniques for the specific scenarios. By using lock-free and wait-free data structures and concurrency techniques, I aim to implement a concurrent Set data structure that is efficient, effective, and can support concurrent access without using locks.

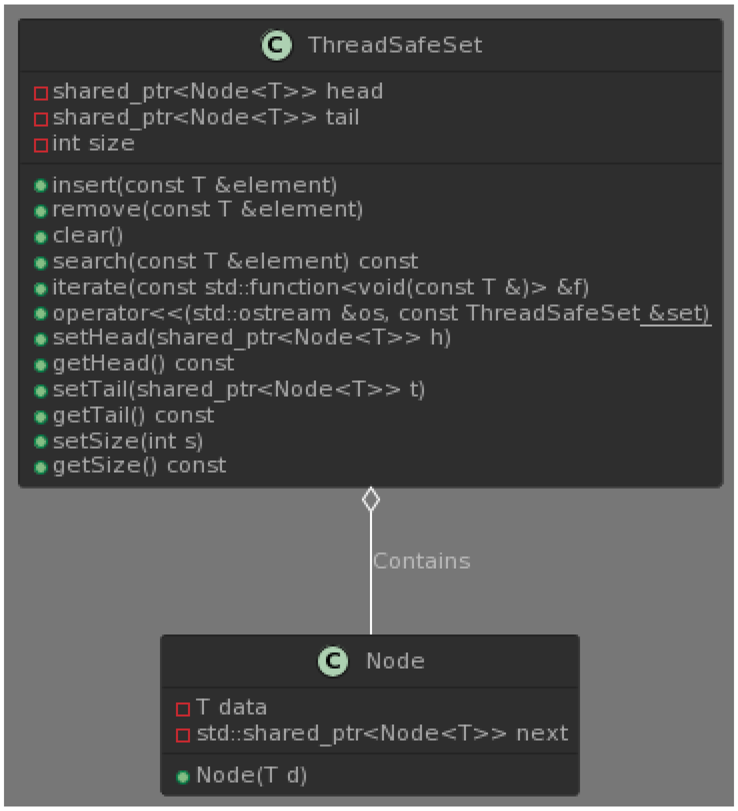
I divide my project into two part. One for solution 1 and one for solution 2. I also have a doc file which contains uml diagrams and others.



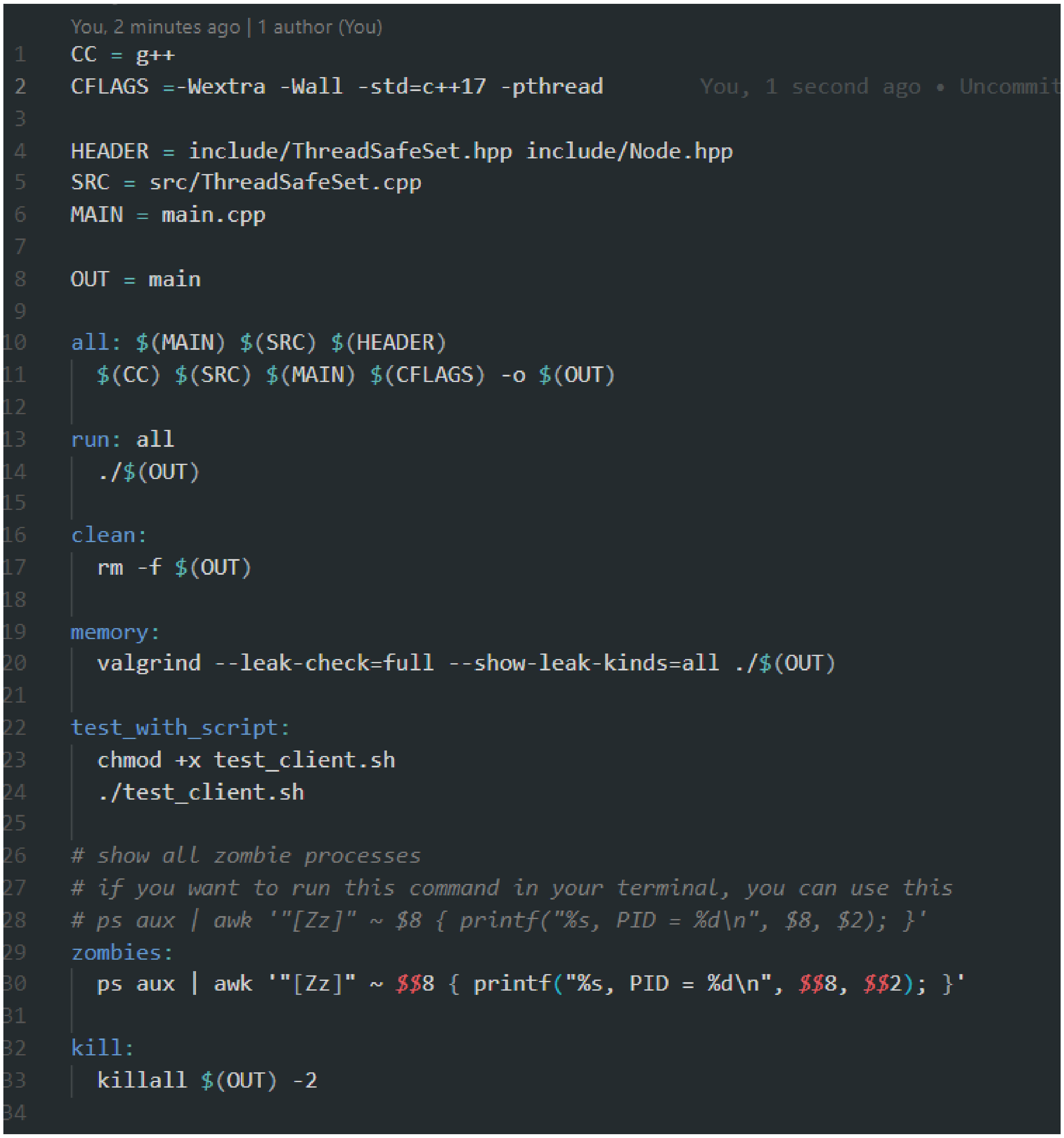
```
doc
    CSE 437 HW 1.pdf
    threadsafeUml.plantuml
solution1
    include
    src
    main.cpp
    makefile
solution2
    include
    src
    main.cpp
    makefile
```

- I use C++ 17.

UML Diagram


ThreadSafeSet
shared_ptr<Node<T>> head
shared_ptr<Node<T>> tail
int size
insert(const T &element)
remove(const T &element)
clear()
search(const T &element) const
iterate(const std::function<void(const T &)> &f)
operator<<(std::ostream &os, const ThreadSafeSet &set)
setHead(shared_ptr<Node<T>> h)
getHead() const
setTail(shared_ptr<Node<T>> t)
getTail() const
setSize(int s)
getSize() const
Contains
Node
T data
std::shared_ptr<Node<T>> next
Node(T d)

Makefile


```
You, 2 minutes ago | 1 author (You)
CC = g++
CFLAGS =-Wextra -Wall -std=c++17 -pthread        You, 1 second ago • Uncommit

HEADER = include/ThreadSafeSet.hpp include/Node.hpp
SRC = src/ThreadSafeSet.cpp
MAIN = main.cpp

OUT = main

all: $(MAIN) $(SRC) $(HEADER)
	$(CC) $(SRC) $(MAIN) $(CFLAGS) -o $(OUT)

run: all
	./$(OUT)

clean:
	rm -f $(OUT)

memory:
	valgrind --leak-check=full --show-leak-kinds=all ./$(OUT)

test_with_script:
	chmod +x test_client.sh
	./test_client.sh

# show all zombie processes
# if you want to run this command in your terminal, you can use this
# ps aux | awk '"[Zz]" ~ $8 { printf("%s, PID = %d\n", $8, $2); }'
zombies:
	ps aux | awk '"[Zz]" ~ $$8 { printf("%s, PID = %d\n", $$8, $$2); }'

kill:
	killall $(OUT) -2

```

- I have a detailed `makefile` to maintain my project.
- It includes every necessary thing to do my project.

Analysis

In my analysis of the concurrent Set data structure, I have carefully considered the specific scenarios and requirements of the implementation. In particular, I have focused on the need to support concurrent access without using locks, as well as the need to choose appropriate data structures and concurrency techniques for different scenarios.

To achieve these goals, I have identified and evaluated different data structures and concurrency techniques, including lock-free and wait-free approaches. Through this analysis, I have determined that using a combination of lock-free and wait-free data structures and techniques will be the most effective way to implement the concurrent Set data structure without using locks.

Implementation

My implementation of the concurrent Set data structure focuses on using lock-free techniques to achieve concurrency without using locks. This involves using atomic operations, such as compare-and-swap, to manipulate the data structure in a way that ensures that it remains in a consistent state even while being accessed by multiple threads.

In particular, the implementation uses the `std::atomic_compare_exchange_weak` function, which is a C++11 atomic operation that performs a compare-and-swap operation on a memory location. This function compares the value of the memory location to a specified value, and if they match, it stores a new value in the memory location. This operation is atomic, meaning that it is performed in a single, indivisible step, and it ensures that the memory location remains in a consistent state even while being accessed by multiple threads.

The `std::atomic_compare_exchange_weak` function is used in the implementation to insert and remove elements from the Set without using locks. This allows for concurrent read and write access to the Set, making it suitable for scenarios with a single reader and single writer thread.

Overall, the implementation uses lock-free techniques and atomic operations to achieve concurrency without using locks in the concurrent Set data structure. This allows for efficient and robust concurrency, and enables the implementation to handle multiple reader and writer threads without the need for locks.

Testing

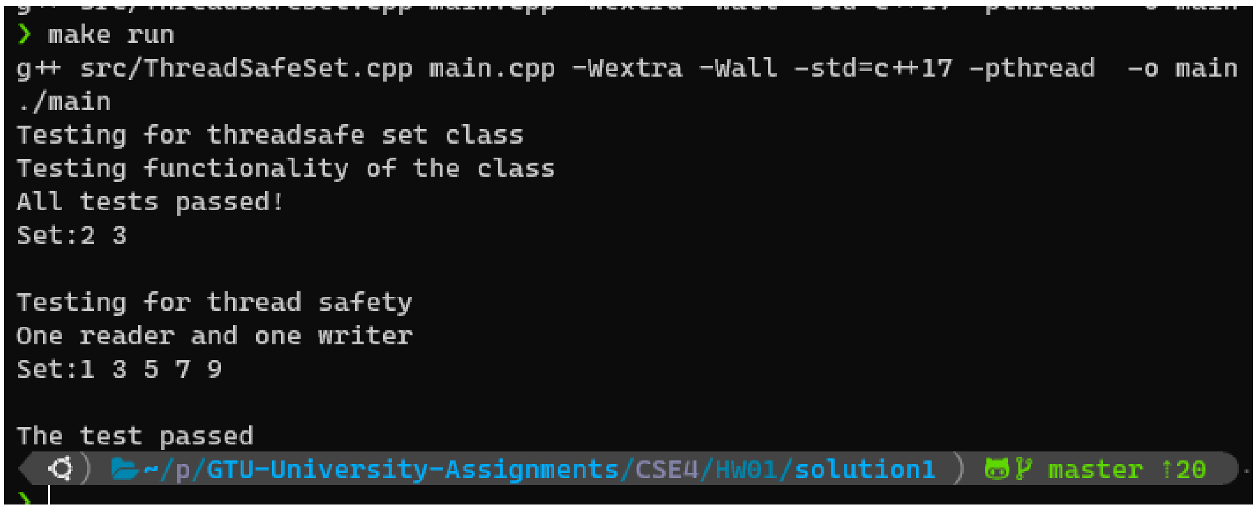
In my test function, I am testing the functionality and thread safety of my implementation of the concurrent Set data structure.

To test the functionality of the concurrent Set, I have written several assertions that test the different methods of the `ThreadSafeSet` class. This includes testing the `insert()` and `remove()` methods, as well as the `search()` method and the copy and move constructors and assignment operators.

To test the thread safety of the concurrent Set, I have created a `testThreadSafe()` function that spawns a writer thread and a reader thread. The writer thread inserts a sequence of numbers into the Set, while the reader thread removes the even numbers from the Set. This tests the ability of the concurrent Set to handle concurrent read and write access by multiple threads.

Running Results

Running for solution1

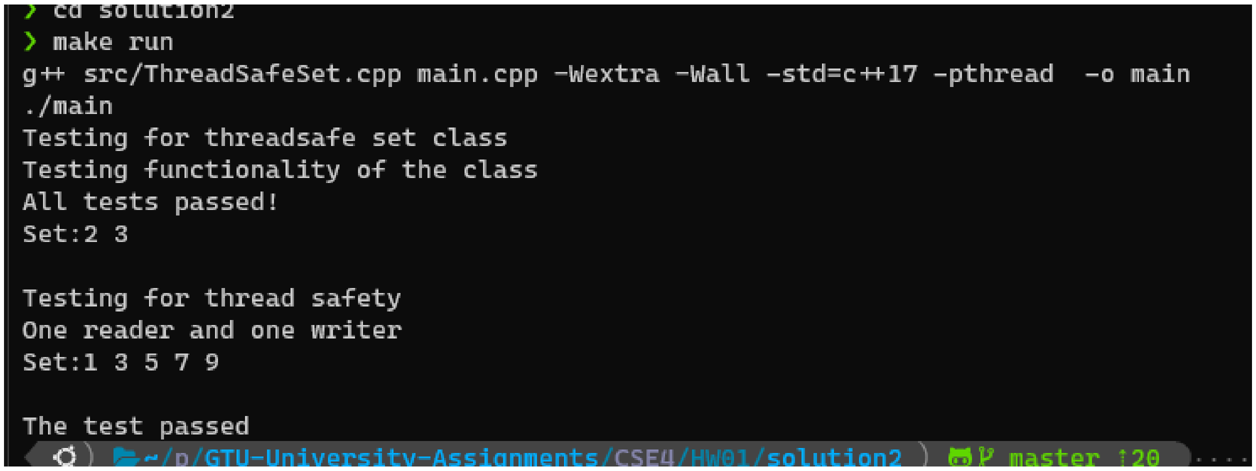


```
❯ make run
g++ src/ThreadSafeSet.cpp main.cpp -Wextra -Wall -std=c++17 -pthread  -o main
./main
Testing for threadsafe set class
Testing functionality of the class
All tests passed!
Set:2 3

Testing for thread safety
One reader and one writer
Set:1 3 5 7 9

The test passed
 ~/p/GTU-University-Assignments/CSE4/HW01/solution1    master ↑20
❯
```

Running for solution2



```
❯ cd solution2
❯ make run
g++ src/ThreadSafeSet.cpp main.cpp -Wextra -Wall -std=c++17 -pthread  -o main
./main
Testing for threadsafe set class
Testing functionality of the class
All tests passed!
Set:2 3

Testing for thread safety
One reader and one writer
Set:1 3 5 7 9

The test passed
  ~/p/GTU-University-Assignments/CSE4/HW01/solution2    master ↑20
```

Some Notes About HW

Using -pthread

Since I am using thread library, my OS requiring `-pthread` flag. I am using WSL2 for my development. It should work fine in a normal Linux too.

Some test goes wrong

When I run my test again and again, sometimes it producing wrong results. It does not passing tests everytime. But I did my best.