CS3211 Assignment 1 Report

Hubertus Adhy Pratama Setiawan - A0200816R
Simon Julian Lauw - A0196678A

Architecture

The matching engine maintains a separate order book for each symbol and operation (buy/sell). At the highest level, a hash table stores the mapping of symbols to its corresponding order book. Inside each order book, the orders are sorted based on their price appropriately. The matching engine also maintains a mapping of order id to pointer to the order object for order cancellation. The architecture can be seen in figure 1.

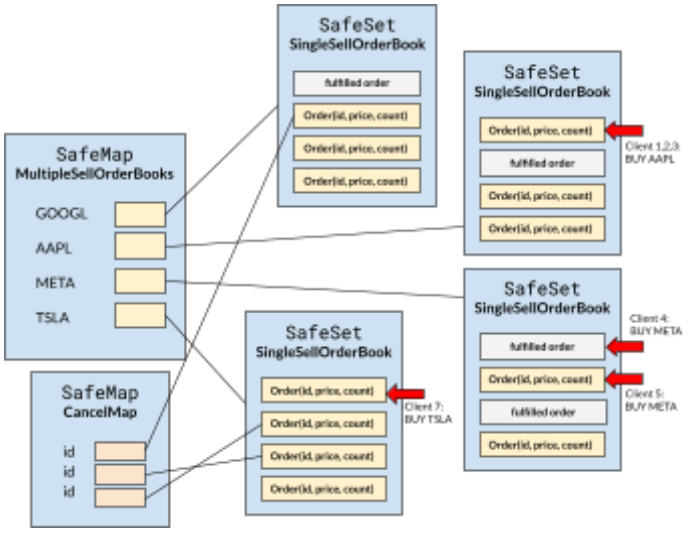


Figure 1. Architecture of the matching engine (sell order book). The buy order book is analogous.

Data Structure(s)

Data structures used to implement the matching engine are

1. Hash Table (to store mapping of symbols to order book and order id to the order itself)
2. Priority Queues implemented using red-black tree (to implement the order book).

Since C++ standard containers do not provide a strong thread-safety guarantee, we implemented our own **`SafeMap`** and **`SafeSet`**. **`SafeMap`** and **`SafeSet`** wrap the standard container (**`std::unordered_map`** or **`std::set`**) together with **`std::shared_mutex`** to ensure exclusive access during write operations while allowing concurrent access during read operations. The implementation can be found in **`SafeMap.hpp`** and **`SafeSet.hpp`**.

Synchronisation Primitives

A. Lightswitches to ensure exclusive access to one side (buy/sell)

To ensure no buy operation is executed during sell operations and vice versa, *lightswitches* [1] are used. *Lightswitch* is built on top of mutexes. The first thread to enter the critical section locks the shared mutex. Other threads of the same type (i.e. buy/sell) will be able to enter the critical section, while the threads of the other type will wait and try to acquire the shared mutex. The last thread leaving the critical section unlocks the shared mutex. The implementation can be found in **`lightswitch.hpp`**.

B. Synchronising concurrent access to the same Order object inside order book

Since multiple threads can access the same order book at the same time and hence the potentially the same order, a mutex is added to each **`Order`** object stored in the order book. When a thread tries to write to the **`Order`** object, it will need to lock the mutex inside the order object before performing the write operation.

Level of Concurrency

The level of concurrency achieved by the matching engine is **phase-level concurrency**.

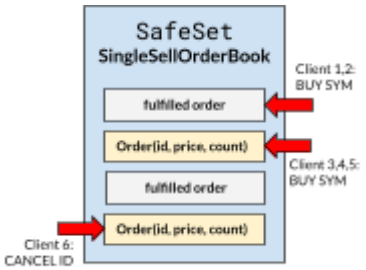
1. Orders of different symbols can execute concurrently.
2. A buy/sell order can execute concurrently with another order of the same type. Orders from different sides (buy and sell) cannot execute concurrently.
3. Cancel orders can execute concurrently with all orders.

A. Instrument-level concurrency

As each symbol has its own orderbook, orders of different symbols can execute concurrently. Moreover, **`MultipleOrderBooks`** can be read by multiple threads concurrently and the exclusive access to it happens during the write operations, which only happens if a new symbol is observed.

B. Multiple operations of the same side within each instrument

During a buy/sell phase (i.e. multiple buy/sell happening concurrently), the same order book can be traversed by multiple threads concurrently as allowed by **`SafeSet`**. Each thread executes any matching order if any, locking the mutex inside the **`Order`** object to prevent data race. Moreover, fulfilled and cancelled orders are not removed from the order book but the count is set to 0, therefore filled and cancelled orders will stay within the order books.



C. Cancel orders

During the execution of cancel order, the thread needs to access the **`CancelMap`** to obtain **`Order`** object, acquire the **`Order`** object lock and then set the count field of the **`Order`** object to zero. This operation can be executed concurrently with any other operations.

Testing Methodology

A. Data Structures

`SafeMap` and **`SafeSet`** were tested by running multiple threads to read and write to the data structure simultaneously. For simplicity, data race detection was done by checking the size of the Set/Map at the end of insertion. First, standard containers were used to ensure that data races can be properly simulated. Then, the standard containers were replaced with their corresponding thread-safe versions. The code was compiled and run three times with different flags: no sanitizer (and run using valgrind), thread sanitizer, and address sanitizer. The implementation can be found in **`safemap_test.cpp`** and **`safeset_test.cpp`**.

```
SUMMARY: ThreadSanitizer: data race /home/e0407797/cs3211-assignment-1-e0389164_e0407797/safeset_test.cpp:23:18
in int_reader(std::set<int, std::less<int>, std::allocator<int>>&)
==================
a.tsan: safeset_test.cpp:54: void int_check(): Assertion `s.size() == NUM_WRITERS * NUM_ITEMS' failed.
./scripts/safeset_test.sh: line 11: 15528 Aborted                 (core dumped) ./a.tsan > /dev/null

running ASAN
a.asan: safeset_test.cpp:54: void int_check(): Assertion `s.size() == NUM_WRITERS * NUM_ITEMS' failed.
./scripts/safeset_test.sh: line 17: 15556 Aborted                 (core dumped) ./a.asan > /dev/null

e0407797@soctf-pdc-012:~/cs3211-assignment-1-e0389164_e0407797$
```

*Figure 2. Ensuring that data race was properly simulated by using **`std::set`** before using **`SafeSet`***

B. Engine

The matching engine was tested using both handwritten and auto-generated test cases. The handwritten test cases simulate the given test cases but with multiple clients. Some scenarios such as multiple cancels are also covered. The auto-generated test cases are used to ensure that the engine can handle multiple connections simultaneously. The engine was tested using 40 clients, 500 symbols, and 25 million random orders. The script to generate the test case can be found in **`scripts/test.py`**.

References

[1] A. B. Downey, “Classical synchronization problems,” in The Little Book of Semaphores, Green Tea Press, 2016, p. 70.