Members: Sean Chen Zhi En (E0978468), Ryad Alla (E1329504)

Description of data structures used

The order matching engine utilises several key data structures to manage orders and ensure efficient execution:

1. **std::multiset<Order> buyOrders, sellOrders**: This is used for both buyOrders and sellOrders to maintain orders in a sorted sequence. The multiset is chosen for its ability to sort orders according to a custom criterion automatically, helping us to match buy and sell orders efficiently based on their prices. It automatically sorts elements with the “<” operator. Also, as opposed to the std::set, it allows for duplicate elements as some orders may have the same price, hence allowing fairness.
2. **std::map<int, std::multiset<Order>::iterator> orderMap**: This maps order IDs to their corresponding iterator in the buyOrders or sellOrders multisets. This allows for fast access, insertion, and deletion operations based on order ID.
3. **std::map<int, int> executionIds**: This keeps track of the execution ID for each order, facilitating the generation of unique execution IDs for matched orders. Therefore, by keeping the execution ID, we accurately track partially filled orders.

Synchronisation primitives used

The code employs mutexes as the primary synchronisation primitive to manage concurrent access to shared data:

1. **std::mutex orderMutexesMutex**: This mutex is used to protect access to the orderMutexes map. The map itself contains a mutex for each order, identified by the order's ID. Since the map may be accessed by multiple threads simultaneously, it's necessary to ensure that only one thread can modify the map at a time, either by adding a new mutex when a new order is received or accessing an existing mutex for order processing.
2. **std::map<uint32_t, std::mutex> orderMutexes**: This map holds a mutex for each order based on the order's ID. This allows for fine-grained locking on a per-order basis. When an order is processed, its corresponding mutex in this map is locked, ensuring that no other thread can process commands for this specific order simultaneously.

Locking Mechanisms

std::lock_guard<std::mutex>: This is a lock guard that simplifies mutex locking and unlocking. When an instance of std::lock_guard is created, it automatically locks the mutex passed to its constructor. When the std::lock_guard object goes out of scope, its destructor automatically releases the lock. This mechanism is used twice in the connection_thread function:

1. It is used when locking orderMutexesMutex to safely access and modify the orderMutexes map.
2. It is also used immediately after locking the specific mutex associated with an order ID. This ensures that access to any shared resource or critical section related to that order is synchronised across threads.

Explanation of level of concurrency

The level of concurrency achieved in the code is **instrument-level concurrency**.

Why instrument-level concurrency?
This was achieved by using a std::mutex for each order ID, stored in the orderMutexes map. This approach allows for fine-grained locking on a per-order basis. This means that operations on orders with different IDs can proceed in parallel, as long as they do not access shared data structures that require synchronisation.

Why not order-level concurrency?
While we used a separate mutex for each order ID, if multiple orders for the same instrument are processed, they would unfortunately likely contend for access to the same data structures (buyOrders, sellOrders, orderMap). As such, even if individual orders are locked by their IDs, operations that need to access or modify shared order books for matching or updating orders are still required to serialise their operations.

Explanation of testing methodology

Our objective was to test some of the edge cases by feeding our engine with ".in" files. In keeping with the assignment's directives, we were able to create some additional test cases covering a situation where we had more threads and instruments. We started by testing the basic functionalities of our engine with the test cases provided in the GitHub repository. Specifically, we tested the cancellation of orders, partial fills and cancelled orders that were eventually rejected because the order was already filled or cancelled previously. Additionally, we tested the case where 2 orders with the same asking price were processed. In this scenario, we ensured that our engine takes the order with the earlier timestamp.

As our goal is to maximise concurrency, we started introducing more threads and instruments. With the help of a Python script, we generated test cases with up to 40 threads running 60,000 orders concurrently on up to 300 different instruments. Some orders were partially filled while others were cancelled.

Through these tests, we verified that our matching engine was able to process all the orders accurately. Hence, this shows that the basic functionalities of our engine is working as intended and that our engine can support multiple threads running concurrently.