Gautham Kailash (E0959106)
Khoo Wui Hong (E0934117)

CS3211
Assignment 1: Exchange matching engine in C++

1) Data Structures for Order Tracking

To manage orders within the exchange matching engine, we have employed a combination of std::unordered_map and std::map data structures, alongside std::queue for maintaining order priorities by time.

The main orderbook structure, `instrument_to_instrument_struct`, is an unordered_map that maps every instrument name (e.g. AAPL) to an `instrument_struct`. The `instrument_struct` contains two std::maps to maintain buy and sell orders respectively, as well as a std::mutexes `instrument_mutex` to protect access to the maps. The separation of buy and sell maps for each instrument allows for more granular control and quicker access during the matching process.

Both std::maps in the `instrument_struct` map a uint32_t value (the price of the instrument) to an std::queue that stores buy / sell orders in chronological order. Every price value received by the matching engine is stored as a price level in the std::map. Using maps ensures that the price levels are sorted in increasing or decreasing order of price, which enables easier access to the best price. We use queues as this allows us to simply append to the back of the queue and pop from the front of the queue, which allows us to obtain the earliest order at that price.

We also employ an unordered_map, `order_id_map`, which allows us to easily access each order by its id, so that we can get information required to delete the order (such as its price and its instrument).

Additionally, we employ a std::unordered_map<std::string, std::unordered_map<uint32_t, uint32_t>> `instrument_to_id_map` to track the count of shares for each order by instrument and order ID. This secondary map serves as a direct lookup to facilitate quick updates and checks on the remaining share count for orders, crucial for handling partial matches and cancellations efficiently. Essentially, the maps in `instrument_struct` serve to identify all the information about the order except the count of shares, and count of shares are tracked in and edited in the inner map of `instrument_to_id_map`.

2) Synchronisation primitives used

We used unique_locks to ensure that in the event that we forgot to unlock them, they would automatically unlock them when the scope exited. We still did manual unlocking to assist us with reading our code and debugging.

3) Level of concurrency

We have achieved instrument level concurrency. Inside `instrument_struct`, there is a single mutex. This means that every instrument has a mutex tagged to it, and as such, whenever any

operations are done for the buy and sell orders of the instrument, that mutex will first be locked, and then unlocked upon completion. This allows for execution of different instruments to execute concurrently, although at any point in time only one order per instrument is being executed. E.g. “Buy GOOG, Sell TSLA, BUY AAPL” can occur at the same time, but not “Buy GOOG, Sell GOOG, Sell TSLA, Buy AAPL”.

Most of the time, the concurrency level will be at instrument level concurrency.

Situations where the concurrency level decreases:

1. In lines 72-76 and lines 194-197 when handling a buy and sell order respectively, we lock the entire order book when we are getting references to the `instrument_struct` and the inner unordered_map of `order_id` to `count`.

2. In lines 90-94, 111-115, 173-176 and lines 212-215, 232-236, 294-297 when handling a buy and sell order respectively, we also lock a global mutex `order_id_map_mutex` as we add it to the unordered_map `order_id_map`.

3. In lines 305-316, when handling a cancel order, we also lock a global mutex `order_id_map_mutex` as we read from the unordered_map `order_id_map`.

4. In lines 318-322, we lock the entire order book when we are getting references to the `instrument_struct` and the inner unordered_map of `order_id` to `count`.

4) Testing Methodology

We generated multiple test cases with multiple threads and ran them (Upwards to 100 threads and 4 million orders). We also ran the same test cases multiple times with a bash script. Manual tracing and compilation with threadsanitizer was also used. We also verified that accessing the value of the outermost unordered_map (E.g. `instrument_struct`} would not cause data races.
https://stackoverflow.com/questions/62253972/is-it-safe-to-reference-a-value-in-unordered-map