Jacob Stein and Marek Pinto
CS3211 - Parallel and Concurrent Programming
Assignment 1 - Concurrent Stock Engine

Data Structures

`Orderbook` - `Orderbook` is a custom class with a vector as an attribute. The vector contains tuples that each represent an order of the form `<int price, int count, int orderId, int executionId>`. Each vector and by extension `Orderbook` contains either the buy or sell orders for a given ticker. Orderbook also has custom accessor and mutator methods for the contents of the vectors. The `Orderbooks` enable concurrency because they come in a tuple of size two to represent the buy and sell orders for a ticker. This is a form of data separation that allows for concurrent access to the buy and sell orders for a stock.

`instrumentMap` - `instrumentMap` is a hashmap created with `Engine`, so only one exists, and it is accessed by all threads in the program. instrumentMap maps each stock ticker to a tuple of length two containing a buy orderbook in index 0, and sell orderbook in index 1. `instrumentMap` enables concurrency because it separates the buy and sell books for each ticker, and thus allows access to and matching between buy and sell orders of the same ticker.

`orders` - *`orders`* is an unordered map that is created for each client. Whenever the client submits a buy or sell order, it is added to `instrumentMap`, and a mapping is created from the id to its `orderbook` in `instrumentMap`. Whenever a client submits a cancel request, it only executes if there is a mapping in their `order` map (meaning the client created the order) and the order is still in its `orderbook` (it has not been canceled or matched yet). *`orders`* enables concurrency because it prevents cross-thread cancellations, which could incur data races if they are executed.

Explanation of Concurrency

We enable the concurrent execution of orders from multiple clients by compartmentalizing data based on ticker and status as a buy or sell order. Since `instrumentMap` maps each ticker to its own tuple of `Orderbooks`, threads can concurrently access and execute orders with different tickers. Furthermore, since each stock has a separate `Orderbook` for its buy and sell orders, orders of different types that share a ticker can be accessed and matched concurrently. In the event that matchable orders are executed concurrently, we add new orders to the book immediately before they look for a match so that they can match with other active orders executing concurrently.

We used mutexes to isolate reads and writes to the `Orderbooks` and to the `instrumentMap` under certain circumstances. `Orderbook` has accessor methods for the vector, and mutators for the vector and the count and execution ID of individual orders. These operations are protected by `lock_guard`s which end within the method definition to limit the area and time of isolation. We also use `lock_guard`s within engine.cpp to limit access to `instrumentMap` when accessing it to update *`orders`*, accessing a specific `Orderbook`, or adding a tuple of `Orderbooks` when a new key (stock) is encountered. The process of finding a match for a given buy or sell order is mutexed by type in a switch statement. A buy order and sell order searching can happen concurrently, but not two buys or two sells.

Our engine achieves the second kind of ***Phase-Level Concurrency***. Orders for different instruments can execute concurrently because the pertinent data is stored in separate tuples accessed via hash map. Orders of the *same* instrument and *different* types can be executed because separate `Orderbooks` are maintained for each instrument. Orders of the *same* instrument and *same* type cannot be executed concurrently because they are stored in the same `Orderbook` vector, which would likely lead to a data race and undefined behavior. Thus, our engine only supports Phase-Level Concurrency of the second type, and not the first.

Testing

We began by testing basic functionality against the provided test cases, which highlighted several pointer and logical errors that we patched. We passed the basic cases and then moved on to manual testing with multiple threads. In an environment with 4 threads, our engine was able to perform cross-thread full and partial matching. The engine also maintained correct ordering of matching based on pricing and timestamp. The engine also only allowed cancellations for orders produced within the same thread.

We then moved on to creating complex test cases using a Python script *generate_test_cases.py*. We generated 100 test files to mimic complex testing cases. We also created two more categories of tests: *medium* (up to 4 clients) and *mediumHard* (up to 20 clients). Below are parameters for our complex test cases.

- Random stock ticker chosen from a group of length 428
- 40 clients
- Random number of orders in range [1000, 50000]
- Random order type, both buy and sell with a probability of ½
- Random assignment of client with an equal probability for all clients
- Random price in range [100, 2000]
- Random count between [10, 1000]

With larger test cases, we found our engine performs logically correct, but outputs in an incorrect order. When two matchable orders are submitted concurrently, they are immediately added to our data structure before searching so that they can find each other. The actual output for “OrderAdded” is only given after potential executions are performed, as we want to avoid mistakenly adding a resolved active order. As such, the first of the two active orders to execute will find the other active order and perform an OrderExecuted, which throws a ‘resting order not in book’ error since “OrderAdded” has not yet been called despite the resting order being in the data structure. The “OrderAdded” is called after the execution, which leads to the correct logic outputted with incorrect timestamps. We were unable to solve this error with our time and resource constraints, and it has a higher probability of occurring with larger test cases, so our engine struggles with *mediumHard* and *complex* cases because of this output error. This error does, however, confirm that we achieved phase-level concurrency because it would not occur otherwise.