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 instrument. Orderbook also has custom accessor and mutator methods for the contents of the vectors. The `Orderbooks` enable concurrency because they are separated into a buy `Orderbook` and sell `Orderbook` for each instrument. This is a form of data separation that allows for concurrent access to the buy and sell orders for an individual stock, and avoids data races when matching concurrently.

`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 instrument to a tuple of length 3 containing shared pointers to a buy orderbook, sell orderbook, and a corresponding mutex. `instrumentMap` enables concurrency because it separates the buy and sell books for each instrument, and thus allows access to and matching between buy and sell orders of the same instrument.

`orders` - `orders` is an unordered map that is created for each client. While `instrumentMap` uses instruments as keys, `orders` stores a mapping from an id to the corresponding `orderbook` where that order is stored. This is useful for cancellation requests, as it allows us to access the corresponding Orderbook for a given id to remove the order from the book. 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 instrument and status as a buy or sell order. Since `instrumentMap` maps each instrument to its own tuple of `Orderbooks`, threads can concurrently access and execute orders with different instruments.

We used mutexes to isolate reads and writes to the `Orderbooks` and to the `instrumentMap` under certain circumstances. We decided that all synchronization would be done from the Engine class for simplicity. This is possible because of the splitting of Orderbooks by instrument and type, so that each Orderbook only needs to be accessed by one thread at a time. We use a `unique_lock` within engine.cpp to limit access to `instrumentMap`. However, instrumentMap is only modified when a new instrument is encountered, and only accessed to update `orders`, access a specific `Orderbook`, or obtaining the pointer to the Orderbook mutex. These actions only happen around once per input cycle, and we use limited scopes to ensure that the instrumentMutex does not significantly reduce concurrency. The process of finding a match for a given buy or sell order is done through the handleOrder function, which sets a unique_lock on the mutex which is stored in the instrumentMap function and corresponds to that mutex. Therefore, orders on different instruments can happen concurrently, but orders on the same instrument are serialized.

Our engine achieves ***Instrument-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 cannot be executed concurrently because they are locked by the mutex corresponding to that instrument, and are therefore serialized. Our engine could be effectively converted to *Phase-Level Concurrency* of the second type by using two mutexes, one for each type of Orderbook. However, when we tested this, it led to issues with a buy and sell coming in at the same time, so we decided to serialize orders on the same instrument to avoid data races.

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 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 instrument chosen from a group of length 428
- 40 clients
- Random number of orders in range [1000, 50000]
- Random order type, both buy, sell, and cancel with a probability of ⅓ each
- Random assignment of client with an equal probability for all clients
- Random price in range [100, 2000]
- Random count between [10, 1000]

We first made an engine that worked at phase-level concurrency. However, with larger test cases, we found our engine performed correctly logically, but outputted in an incorrect order. When two matchable orders were submitted concurrently, we decided to immediately add them to our data structure before searching so that they could find each other. As such, the first of the two active orders to execute found the other active order and performed an "OrderExecuted", which threw a 'resting order not in book' error since "OrderAdded" had not yet been called despite the resting order being in the data structure.

In order to debug these issues, we used ThreadSanitizer, as well as printing to std:cerr to identify the order in which different parts of our program were executed. We also created our own timestamp system (with a counter starting at 0 and incrementing whenever a timestamp was printed), so that we could more easily track the execution order and fix issues with orders being added to the book out of order.

We then rewrote our code to its new form, putting all synchronization within the engine class. We also used a single mutex for both buy and sell orderbooks to avoid the issue of matching buys and sells that came in at the same time. By imposing these concurrency restraints, we were able to achieve 100% accuracy across all concurrent test cases (medium, mediumHard, and complex).