CS3211 Assignment 1

- Albert Sutiono (A0219773U)
- Daffa Fathani Adila (A0219822A)

Description of data structures used (eg. linked list, priority queue, skip list)

We used two main data structures in this assignment.

a. **std::set<Order>** (with proper comparator) to represent our **buySet** and **sellSet**. In particular, **buySet** and **sellSet** have their own comparator according to the priority criteria of **BUY** and **SELL Order**.
b. **std::unordered_map<std::string, Instrument> m_instrument_map** to help with Instrument Level concurrency. In particular, this connects the **instrument string** to the **Instrument object**. With this, we can put the synchronisation measures for processing **Order** inside each **Instrument object** and thus, the different queries of different **Instrument** can run concurrently.
c. **std::unordered_map<uint32_t, pair<std::string, OrderType>> m_id_map** to help with Cancel Order. The key is **order_id** and the value is **pair** of **(instrument, OrderType).** With incoming cancel order, we can get the initial order's instrument and type, which can be used to obtain the corresponding **Instrument Object** and be processed.

Relevant Classes/Structs created:

a. **Order and CancelOrder**: classes to represent the incoming orders. Mainly serves as data holder.
b. **Instrument**: class to represent one particular instrument object. **Instrument** holds **buySet** and **sellSet**. The member functions also serve to do all the processing of a particular order to the corresponding **buySet** and **sellSet.**
c. **Orderbook**: class to represent the main orderbook (used directly in **Engine)**. **Orderbook** holds **m_instrument_map** and **m_id_map** and acts as a higher layer encapsulating **Instrument**.
d. **ResultWrapper**: wrapper class to hold the results to be printed to **Engine's cout**. Works mainly as just a **std::vector** wrapper of **IResult object** (holds the data relevant to printing).

Synchronisation primitives used (eg. atomics, mutexes, hand-over-hand locking)

a. **std::Mutex**: Mutex is used to ensure atomic operations of data structures. For example, set insertions and accesses as well as increment/decrement counter variable. Additionally, mutex is also used for critical section, i.e. acquiring mutex to enter the critical section. In our model, only one thread can execute matching orders at any point of time, and is guaranteed by using a combination of mutexes and unique_lock.
b. **Std::condition_variable**: In our model, we adapted the readers-writers problem using conditional variable. At any point of time, more than one orders of one type (either Buy, Sell, or Cancel) can be processed, while the other types of order wait. When there is no more orders of that type to be processed, it will notify all the waiting orders using std::condition_variable.

Explanation of *level of concurrency* (order level, instrument level, etc.). Must match with the explanation of synchronisation primitives.

Phase Order: (Instrument-level + some concurrency among orders of same type)

For the **instrument level**, we deliberately designed our classes as such (**Orderbook** holding multiple **Instrument objects** and the logic to process each process is inside the **Instrument**). This means incoming queries can be designated to different **Instrument objects** to be processed concurrently. Notice that in the **Orderbook Object**, we use **mutex** to maintain **m_instrument_map** and **m_id_map's** invariants. However, processing of the query itself is managed by **Instrument**.

Cancel Order for each instrument must be run exclusively.

The more complicated part is to **enable concurrency among orders of the same type** (some BUYs or some SELLs orders) to be processed concurrently. WLOG, let's take multiple BUY orders. Notice that an **active order that can be matched** (mainly deals with popping **sellSet)** can be processed together with **active orders that cannot be matched** (only deals with inserting to buySet). Essentially, every buy order will either wait to be matched, or directly added into the orderbook. Every incoming buy order will peek the best sell order (atomically). If it can match the best Sell, then it will try to acquire the execution mutex, otherwise, it will be added to the buy orderbook (atomically).

To ensure atomicity when concurrently accessing/inserting elements, we make sure to only lock mutex when immediately inserting/reading from the **buySet** or **sellSet**. However, most of the processing logic can be run in parallel (matching orders and inserting orders to the orderbook).

For correctness, **BUY orders** and **SELL orders** must run **exclusively** from each other. However, **multiple BUY orders must be able to run concurrently**. Therefore, **conditional variable** (Inspired by **Reader-Writer Pattern**) is used.

Testing methodology

We wrote a script to generate test cases by specifying the number of threads, number of buy, sell, and cancel orders, and number of different instruments. We tested the program using various combinations of test case (1 - 40 threads, 100 - 100,000 buys, sells, and cancels, 1 - 10000 instruments) and used the grader to verify the correctness. The script can be found under the /scripts sdirectory.

Although there are warnings from thread sanitiser (lock ordering inversion), it can be observed that deadlock will not occur as we have separated the execution of different order types (buy, sell, and cancel).