CS3211 Assignment 1 Report

Team

Yeat Nai Jie (A0219955N/E0550613)	Brendan Lau Siew Zhi (A0219938M/E0550596)

Data Structures

OrderMap

Instantiated within `Engine` and contains two `unordered_map` s.

- `instrument_map` - maps the instrument string to `InstrumentOrderBook`
- `order_instrument_map` - maps order id to instrument string, used when cancelling an order.

InstrumentOrderBook

Contains two `HeadNode` objects, one for `buy_head` and one for `sell_head`, essentially acting as the head node for the buy order book and sell order book (linked list) respectively. One `InstrumentOrderBook` is created for each instrument. We perform buy and sell matching, as well as cancelling on these linked lists. In addition, both linked lists are designed to remain sorted throughout operations in order to facilitate order matching. This is done by inserting `Node` objects at the correct position when new orders are added

HeadNode

Contains a `Node` object, which acts as the head node for the buy/sell order book in `InstrumentOrderBook`. Two `HeadNode` objects are created per `InstrumentOrderBook`, one for the instrument's buy order book, and one for its sell order book,

Node

Contains an `Order` object and a pointer to the next `Node` in the linked list for buy/sell order books

Order

Contains information on a resting order in buy/sell order books.

Synchronisation primitives

We enable concurrency between orders for different instruments by having seperate `InstrumentOrderBook` instances for each distinct instrument, so commands for different instruments can be carried out in isolation from each other.

To enable concurrency within each `InstrumentOrderBook`, our solution makes use of fine grained locks in each node in the linked lists for each instrument. For maximum concurrency, we use `shared_mutex` s instead of normal mutexes. Our implementation is explained using the examples below:

> ***NOTE:*** At the beginning of each buy or sell, we instantiate two `unique_lock` s on control mutexes in `sell_head` and `buy_head`. This is to introduce a layer of randomisation with regards to the ordering of different orders of the same instrument. Without this randomisation, we observed that some threads could be blocked for extended periods e.g. for the sequence of orders `S` `B` `S` `S` `S` `S` `S` `S` ... the `B` order could be blocked for a long time leading to timeouts. We also lock the corresponding control mutexes when performing a cancel.

Example: single buy

1. When a buy order is received, 'tryExecuteBuy' is called, and we instantiate a `unique_lock` on `sell_head` and a `shared_lock` on `buy_head`.
    - the reason why we have use a `unique_lock` on `sell_head` is to ensure that there are no sell orders for the same instrument is currently in the midst of finding matching orders, since any thread calling `tryExecuteSell` would have a `shared_lock` on `sell_head` when matching orders, which would block the `tryExecuteBuy` thread.
2. Then, we unlock this `unique_lock` on `sell_head` and traverse through the `sell_head` linked list to look for potential matching orders, locking nodes before attempting to access it and releasing them after we are done checking the node
3. If no matching orders are available, we unlock the `shared_lock` on `buy_head` and traverse through the `buy_head` linked list to insert the order at its correct position in the sorted linked list, locking nodes before accessing it and releasing once we are done checking.

Example: single sell

Similar to the above single buy example, the same idea is used here, but we instantiate a `shared_lock` on `sell_head` and a `unique_lock` on `buy_head` instead.

Example: multiple buys(same instrument)

Multiple buys received in succession from multiple different threads for the same instrument can be processed concurrently, i.e. another buy order can begin processing before the preceding one has been completed.

1. When a buy order is received, `tryExecuteBuy` is called, and we instantiate a `unique_lock` on `sell_head` and a `shared_lock` on `buy_head`.
2. Then, we release the `unique_lock` on `sell_head` and traverse through the `sell_head` linked list to look for potential matching orders, locking nodes before attempting to access it and releasing them after we are done checking the node.
3. During this stage, buy orders from different threads can be carried out concurrently once the `unique_lock` on `sell_head` is released by the initial thread.
4. Similar to the single buy example above, if no matching orders are available, the `shared_lock` on `buy_head` is released and we traverse through the `buy_head` linked list to insert the order into the linked list.
5. Only after all threads executing buy orders release the `shared_lock` on `buy_head` can sell orders start being processed as sell orders need a `unique_lock` on `buy_head`.

Example: multiple sells(same instrument)

Similar to the above multiple buys example, the same idea is used here, but we instantiate a `shared_lock` on `sell_head` and a `unique_lock` on `buy_head` instead.

Example: one buy and one sell(same instrument)

Suppose a buy order comes in followed immediately by a sell order

1. When the buy order is received, `tryExecuteBuy` is called, and we instantiate a `unique_lock` on `sell_head` and a `shared_lock` on `buy_head`.
2. When the sell order is received, `tryExecuteSell` is called, and we instantiate a `shared_lock` on `sell_head` and a `unique_lock` on `buy_head`. However, the thread calling this will be blocked since both mutexes in `sell_head` and `buy_head` have been locked by the buy order
3. Then, we unlock `sell_head` and start traversing through the `sell_head` linked list to look for matching orders, locking nodes before attempting to access it and releasing them after we are done checking the node.
    - When `sell_head` is unlocked, it'll be acquired using a `shared_lock` by the sell order thread
4. If no matching order is available, the `shared_lock` on `buy_head` is released and we traverse through the `buy_head` linked list to insert the order into the linked list
    - When 'buy_head' is unlocked, it'll be acquired using a `unique_lock` by the sell order thread
5. At this point, the sell order will start being processed since it has acquired both mutexes in `sell_head` and `buy_head`
6. The sell order thread will unlock `buy_head` and traverse through the `buy_head` linked list to look for matching order, locking nodes before attempting to access it and releasing them after it is done checking the node (similar to step 3).
7. If no matching order is available, it unlocks `sell_head` and traverses through the `sell_head` linked list to insert the order into the linked list (similar to step 4)

Example: cancel order

1. When a cancel command is received, we get the associated instrument string from `order_instrument_map` in `OrderMap`
2. Then, `tryCancel` is called, and we instantiate a `unique_lock` on `sell_head`
3. Then, we release the lock on `sell_head` and traverse through the `sell_head` linked list to look for the order to cancel, locking nodes before accessing them, and releasing after we are done checking the nodes.
4. If the order is not found in the `sell_head` linked list, we instantiate a `unique_lock` on `buy_head` and traverse the `buy_head` linked list to look for the order
5. The order is cancelled if possible once it is found.

Level of concurrency

As explained above, our data structures enable multiple orders (buys or sells) to execute concurrently for each instrument. Hence, our implementation is a "Phase-level concurrency" solution. This is because for each instrument, multiple buys or sells (but not both) can execute concurrently.

Testing

We used a python script to generate large test cases randomly, with the largest test case having 160000 order IDs and 80 threads.