CS3211

(Parallel and Concurrent Programming)

Assignment 1 Report

By

Name	Student ID
Koh Jia Cheng	A0249132J
Muqaffa Al-Afham Bin Kamaruzaman	A0249221L

Date of Submission: 20th Feb 2022

1. Implementation & Data Structure

For Assignment 1: Exchange matching engine in C++, our implementation involves three basic data structure in order to handle concurrency mechanism, which is as follows:

1. OrderBook Class

```
class OrderBook{
  private:
    std::mutex orderbook_mutex;
    std::map<int, std::string> order_to_instrument;
    std::map<std::string, OrderList*> instrument_map;
  public:
    void printOrderBook();
    std::string getInstructmentByID(int order_ID);
    OrderList* getOrderList(int order_id, std::string instrument_name, bool is_cancel);
};
```

Fig 1.1. (Code of the OrderBook class)

All incoming threads' requests are handled through the *OrderBook* object. In order to perform any match/cancel operation on an instrument, the thread must first gain access to *OrderBook* through orderbook_mutex and then obtain *OrderList* object. Additionally, acquisition and relinquish of the mutex is handled within *getOrderList()* method with the use of *scope lock*.

Furthermore, our implementation utilise the Map structure, order_to_instrument, to keep track of orders and its respective OrderList. Each pairing is inserted every time an input type is 'Buy' or 'Sell'. Lastly, the printOrderBook() method outputs resting orders from each OrderList, which is used for testing and debugging. The method is called with the input 'P'.

2. OrderList

```
class OrderList{
  private:
    Order* b_head = NULL; // Sorted Descending (Buy)
    Order* s_head = NULL; // Sorted Ascending (Sell)
    std::string instrument;
    std::mutex instrument_mutex;
    std::map<int, Order*> resting_orders;
  public:
    void printOrders();
    void matchOrder(Order* order, std::chrono::microseconds::rep input_time_stamp);
    void cancelOrder(int order_ID, std::chrono::microseconds::rep input_time_stamp);
    void insertSellOrder(Order* order, std::chrono::microseconds::rep input_time_stamp);
    void insertBuyOrder(Order* order, std::chrono::microseconds::rep input_time_stamp);
    OrderList(std::string instrument_name): instrument(instrument_name){};
};
```

Fig 1.2. (Code of the OrderList class)

As mentioned previously, a threads must first gain access to the *OrderBook* object to obtain the *OrderList* object, implying that operation for *OrderBook* is sequential.

The *OrderList* object is where the concurrency mechanism takes place. Each *OrderList* object represent a list of order for a particular instrument, for example all 'GOOG' orders are in one *OrderList* whereas all 'BABA' orders are in another *OrderList*. Consequently, each unique *OrderList* object has its own *instrument_mutex*, allowing **multiple** unique instruments' order to be executed simultaneously. Acquisition and relinquish of the mutex is done with the use of *scope lock*.

A new *OrderList* is initialised if there is no instrument found from *OrderBook's getOrderList()* method. To access any attributes and methods within *OrderList* object, the thread must acquire the *instrument_mutex*. Lastly, 'Buy' and 'Sell' orders are stored in a doubly linked list structure, referenced through the pointer *b_head* and *s_head* respectively.

3. **OrderList's Doubly Linked List**

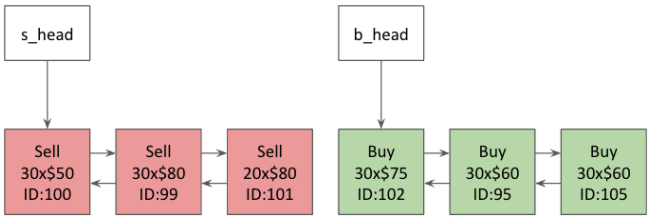

Fig 1.3. (Sorted Doubly Linked List)

In the Doubly Linked List structure, 'Sell' orders are stored in ascending order, whereas 'Buy' orders are stored in descending order. As a result, active 'Sell' orders can simply match against the head of 'Buy' orders which is the highest buying price. Similarly, for active 'Buy' orders, it will be matched against the lowest selling price. This eliminates the need to traverse the entire link list to search for the best deal.

During the matching process, any fulfilled resting order will be discarded and removed from the Map *resting_orders*, and if the active order has not been fully matched, it will be inserted into its respective linked list and *resting_orders*. Lastly, earlier orders will be inserted closer to the head, meaning any resting orders that are executed will always be those that were inserted earlier, as matches are done from the head.

2. Concurrent Execution

As mentioned above, our implementation enables concurrent execution of orders of different instruments. Each *OrderList* can perform execution concurrently with another *OrderList*, whereas executions performed in the *OrderList* is sequential as it would require the mutex *instrument_mutex*. The reason being that only one thread is allowed to scan through the Linked List of Orders of *OrderList* at any point of time. In addition, our implementation added the mutex *print_mutex,* to allow thread safe output environment. Lastly, fulfilled/cancelled orders are freed, preventing any memory leakage.

3. Testing

In order to test our program's concurrency mechanism and reliability, our group wrote three types of basic bash script:

1. **test_order.sh**
   The script executes six concurrent threads, each sending 'Buy' and 'Sell' orders into the client program. In short, the script sends a total of 12,000 orders, with a mix of instruments. Lastly, all 12,000 orders should be fully matched, leaving 0 resting order at the end, which we input 'P' to print out the resting orders.

2. **test_random.sh**
   The script is identical as test_order.sh, except the *Price* and *Size* for each order are set randomly, ranging from 25 to 100.

3. **test_cancel.sh**
   The script executes three concurrent threads, two sending in 'Buy' and 'Sell' orders to the client program, while the other sends 'Cancel' order.

Our program managed to finish executing all three test cases without any errors.

4. References

1. (2022). std::scoped_lock. Retrieved from cppreference: https://en.cppreference.com/w/cpp/thread/scoped_lock
2. (2022). std::map. Retrieved from cppreference: https://en.cppreference.com/w/cpp/container/map
3. Mark Reed. (2013, May 12). How do I provide input to a C program from bash? Retrieved from StackOverFlow: https://stackoverflow.com/questions/16508817/how-do-i-provide-input-to-a-c-program-from-bash