Description of Data Structures

We have used two linked lists to simulate a Sell Order book and a Buy Order book. Each individual node of these linked lists can be locked individually, and we have opted to lock the nodes of each linked list in the same order each time when matching, inserting and cancelling in order to eliminate deadlocks.

Both of the linked lists that we use as order books have a dummy head and dummy tail node. The dummy head/tail allows *head_* and *tail_* pointers to point to something even when the linked list is empty (hence preventing null access errors). When an insertion is done in either of the Linked Lists, it is done in-place (the linked list is sorted).

A boolean value *sort_asc* determines whether the linked list is used for storing resting buy orders (*sort_asc* = true) or for storing resting sell orders (*sort_asc* = false), and is initialised during construction.

Each node of the linked list has a mutex, an *input* struct and an *execution ID*, as well as a pointer to the next node. The *execution ID* of a node is incremented whenever it is matched as a resting order.

Explanation of Concurrency Support

By allowing each node of both linked lists to be locked individually, we have allowed multiple orders of the same type to traverse the same linked list simultaneously, resulting in greater concurrency.

When traversing down a linked list, a pair of mutexes are used to lock access to the two nodes currently being read/written to. (the first node to be locked is *head_*, and the second node to be locked is *head_->next*). This ensures that there will be no data races (two threads try to read/write to the same node or it's neighbouring node). A *std::swap* operation and manipulation of *Node* pointers is used to ensure that this pair of mutexes are able to traverse down the linked list without any deadlocks, and to ensure that the locking of both these mutexes happens in the same order every time, which is crucial in order to prevent deadlocks. This process is repeated until the targeted node is reached or the dummy *tail_* node is discovered.

In order to ensure that two incoming orders that would be matched with each other are **NOT** added to the order book together, a "Switch-Mutex" is used.

1. Whenever an order comes in, we check whether it is of the same type of orders that is currently executing (concurrently)
2. If it is the same type, then the "Switch-Mutex" is NOT acquired, and this order is allowed to execute concurrently.
3. If it is not of the same type, then the "Switch-Mutex" is acquired, and this order must be executed sequentially with the rest of the RESTING orders.

This enforces a “happens before” relationship between the insertion of a buy order and the first match of a sell order and vice versa.

In actuality, we could remove the “Switch-Mutex” altogether to maximise the concurrency of the engine, but we found that this would compromise correctness of the engine as there is a probability that a matching buy and sell order would not see each other when matching, and insert themselves into their respective order books. As a result we have chosen to stick with the current implementation.

To compare types between the current order and the previous order, the atomic<*input_type*> *lastOrderType* is used. We have decided to use *mem_order_acquire* for loads and *mem_order_release* for stores of this atomic variable as each store of this variable in one thread needs to synchronise with the load of the same variable in another thread.

For cancellation orders and resting orders which are fully matched, we have decided to set the quantity of these orders to 0 instead of deleting them from the list. This is to ensure that no memory leaks/errors will be caused due to deletion of nodes.

Finally, whenever an order is added, an order is executed, or an order is cancelled, it needs to print to standard output. In order to do this with no data races, we have implemented a *print_mutex* which is taken whenever the program wants to print to standard output, and unlocked after the printing is done.

Description of Testing

1. Test cases (simulating threads) were created and placed into different folders.
2. Scripts were created for each test case to emulate different clients sending out commands on their own threads.

All test cases can be found within the files on GitHub, with test1 referencing files within test1_files and so on. Note that the script files have to be in the correct directory as the engine and client output to work (for submission purposes, we have also consolidated all of them in a *scripts* folder).

Additionally, code was compiled using Thread Sanitizer to detect and rectify data races.