Data Structure used:

InstrumentList => a struct containing a map, to map instrument to a its own instrument order book InstrumentOrder, with a mutex lock for the map accessing to prevent adding 2 same instrument concurrently if ever. Along with a vector to keep track of the names of instruments that have been allocated its own order list.

InstrumentOrder => a struct containing 2 BSOrderList, which are the buy and sell order list for the specific instrument. It also has the methods to call the BSOrderList to perform the necessary buy/sell/cancel actions on the order list for its own instrument.

BSOrderList => a struct containing a Linked List of OrderNodes which contains DataOrder, which is the buy/sell ordering being stored in a resting state. It also uses has atomics and methods to access them to ensure mutual exclusion for access the itself(order list) and also the methods for accessing the itself(order list).

At any one time, only one client's active order should be access an instrument's order book so as to prevent multiple buy/sell trying to match and execute with the same sell/buy.

And, prevent the situation where one buy and one sell order which would match, ends up trying to match and fail to match then append to their own buy and sell order list and end up not matching.

Thus, when a matching action is done, both buy and sell order list for an instrument has to be locked to prevent data race, race condition and locked in the same order for sell, buy, cancel actions to prevent deadlocks.

The InstrumentList map would also lock when adding a mapping so as to prevent 2 separate threads from trying to add the same map and override one another or some other undefined result, thus a data race.

Level of Concurrency: Partial Phase-level (?)

In an attempt to not have to lock every single OrderNode in the Linked List in every instrument order list, I attempted a semaphore implementation for the add_order action, which is the 2nd half of a buy/sell action, which consists of the matching/try_execute + add_order action(if matching fails or if there is remainder). Because adding orders to an instrument order list can be done in parallel as long as an additional node is appended to the linked list, the only locking and mutually exclusive portion would be the addition of new dummy node and change the last node to point to this new dummy node. Afterwards, the filling in of the OrderData of an OrderNode can be done simultaneously by multiple threads of multiple clients in parallel. Thus, I have a diff_action_type atomic<int> which is incremented during access, locking a different action type from accessing the order list(ie. prevent buy action from accessing sell order list when sell order list's diff_action_type is incremented), but at the same time, when the atomic<int> diff_action_type is >= 0, fellow same type actions can share this "lock" by incrementing it(ie. other sell action type can further increase the diff_action_type of the sell order list and access the sell order list, thus if an instrument's sell order list diff_action_type = 10, then 10 threads are running the sell action on this instrument at once, and decrement it afterwards using compare and exchange afterwards) Then for a buy action to access a sell order list, it would require this diff_action_type to be 0 which means no sell action is acting on it currently, and it can further go to -1 to lock others out and "acquire the lock" of diff_action_type. Preventing other buy, sell and cancel action from accessing this sell order list.

Thus for at least half of the operation of a buy/sell action, multiple of them can run at the same time, but matching cannot occur at the same time due to a linked list level lock, which is not as finely grained as a node level locking. Though this design is more unique and does not make each node have a lock though!

The Compare and exchange is done using acquire and release also to ensure the add_order or buy/sell actions are visible to other threads who then access it after them.

Cancel order requires the buy or sell list of the instrument it is checking currently to be locked so that no other active orders can try to match it and also to prevent the add_order to join the linked list to the OrderNode that is being cancelled and removed and thus cause possible use after free and disjoint the linked list.

Testing Methodology:

Scripts that generate test cases and batch files which has scripts to run grader against the test cases repeatedly to check for errors are written. One batch file script is just to run against a single fixed test file repeatedly. The other batch file script generates a test case and runs it with the grader and engine repeatedly and this whole sequence repeats with newly generated test cases repeatedly to check empirically for any issues.

Specific test cases also such as, single instrument and multi threaded access to check issues regarding instrument-level concurrency is done also to ensure no timeout due to deadlocks or other issues.