Lee Ee Jian A0183105N; Leow Jit Yong A0182998A

Concurrent Data Structures

1.1 Concurrent HashMap	1.2 Concurrent LinkedList
Supports generic key-value pairs. HashMap consists of an array of buckets, and an array of mutices for each bucket. Each bucket consists of a singly linked list of HashNodes whose keys hash to the same bucket. Each HashNodes stores the key-value pairs. The concurrent hashmap allows for: (A) shared reading to allow threads to check if a particular key exists in a bucket and get the corresponding value (B) exclusive writing to a bucket when a thread wants to add a new key-value pair.	As mentioned, every Instrument has its own OrderBook. Each OrderBook stores its own SellBook, BuyBook and mutex. The mutex must be obtained before traversing the SellBook or BuyBook. The Sellbook and Buybook are both concurrent, sorted singly linked lists of OrderNode. Each OrderNode has its own mutex, and stores a list of Orders of the same price that are sorted by time. Each OrderNode maintaining its own lock allows for our implementation to support more fine-grained concurrency, because active orders only need to acquire the locks for matching OrderNodes for execution, leaving the unneeded OrderNodes unlocked

Table 1

Supporting Concurrent Executions using Locking

The first level of concurrency is at the instrument level. Engine has a member `orderBooks` HashMap that stores the Instrument (key) - OrderBook (value) pairs. Thus, threads executing orders for different instruments can happen concurrently.

The second level of concurrency is at OrderBook level. By having a separate BuyBook and SellBook within an OrderBook, we allow threads to work on them concurrently, with more fine-grained support within each buy/sell book via their OrderNodes.

For two threads executing orders on the same OrderBook, the overarching idea is for the thread to first acquire all relevant locks required for their operation before it can start its execution logic. For matching of orders, we need to acquire all the locks for OrderNodes that we need to traverse and for Add and Cancel orders, we only need to acquire the head of the corresponding Buy/Sell book's lock before we start.

We use the following approach the to support concurrent execution on the same OrderBook.

2.1 Process Buy/Sell Order	2.2 Matching Order	2.3 Adding Order	2.4 Cancel Order
a. Lock OrderBook mutex b. Traverse and acquire all required locks from orderNodes in the corresponding buy/sell book. c. If it would be a partial order after matching those OrderNodes from (b), lock the head of the buy/sell book that the active order will be added to. d. Unlock the OrderBook mutex e. Perform MatchOrder (part 2.2), followed by Add (part 1.3) if it's a partial order in step (c) .	a. Traverse the vector of orders from matching OrderNode from the buy/sell book, then match and remove them. b. Unlock OrderNode lock acquired from part (2.1b) before traversing to the next OrderNode. c. Repeat steps a & b until the active order's count == 0 or there are no more resting orders to be matched.	a. Traverse the buy/sell book using the "Hand-over-Hand" locking mechanism to find existing OrderNode to add into, or insert a new OrderNode into the linked list to add the order into. b. Unlock the OrderNode that we added into after the add operation is complete.	a. Traverse the buy/sell book using the "Hand-over-Hand" mechanism to search for the OrderNode where the order to be cancelled resides in. b. If found, traverse the OrderNode's vector of orders and find the order to be cancelled, using its order id. Else, cancellation is rejected. c. Unlock the locked OrderNode before returning.

Table 2

Note: “Hand-over-Hand” locking mechanism: Acquire lock for next node before releasing lock on current node.

Shown below are some of the cases where operations from 2 threads work on the same instrument and BuyBook / SellBook and can achieve concurrency.

MatchOrder & MatchOrder	S1 can be fully matched by BuyBook, B1 can be fully matched by SellBook S1 can be matching with BuyBook and unlocks OrderNodes required by S2. After S2 acquires all required OrderNodes, S2 can start matchOrder too, running concurrently with S1.
MatchOrder & AddOrder	S1 (can be fully matched) matching with BuyBook, B1 can start traversal to find Add location by locking OrderNodes that S1 unlocks → B1 will never be visible to S1 when it’s matching. If S1 cannot be fully matched, S1 can either start matching with BuyBook after B1 unlocks BuyBook head, or B1 can only start traversing after S1 unlocks SellBook head.
MatchOrder & CancelOrder	S1 matching with BuyBook, C1 can start traversal of BuyBook to find OrderNode by locking OrderNodes that S1 unlocks.
AddOrder & AddOrder	S1 traversing SellBook to find Add location, S2 can start traversal of SellBook by locking OrderNodes that S1 unlocks.
AddOrder & CancelOrder	S1 traversing SellBook to find Add location, C1 can start traversing SellBook by locking OrderNodes unlocked by S1.
CancelOrder & CancelOrder	C1 traversing SellBook to find Add location, C2 can start traversing SellBook by locking OrderNodes unlocked by S1.

Table 3

Testing

Generator.py	Test.sh	Checker.py
Helps to create and distribute randomly generated orders to desired number of input files based on the user-specified number of threads.	Spawns a process that calls ./client socket for the number of threads specified, reading from a generated input file each.	Test the correctness of single-threaded execution using the input to client and output from engine. We sort the output from the engine by their output time and read it sequentially, maintaining a state of the OrderBooks and active orders at any point in time. This helps us determine if each output is valid for the maintained state. For example, we check if the execution of an active order is matching with the best resting order and update their remaining count. The active order’s remaining count will be used to check if it is correct when the active order is getting added as resting order. We also check if cancel operations are correctly rejected or accepted. Finally, we maintain a count to ensure that every input order is executed.

Table 4 – Scripts for Testing

We make use of Generator.py and test.sh to run multiple clients at the same time. For single-threaded execution, we used Checker.py to ensure correctness.

For smaller number of input orders, we manually printing out every OrderNode in the OrderBook and checking that the state of the OrderBooks are consistent after executing each order
For larger number of input orders, we ensure that there is no segmentation fault, the number of lines of output is greater than the number of inputs and all inputs are executed.

For multi-threaded executions, Checker.py is not able to check for execution correctness because simply checking the consistency of the OrderBook for executions sorted by timestamp is not sufficient.

For example, thread A that is performing matchOrder will only traverse nodes that are visible to itself when it was acquiring the OrderNodes’ locks previously. During the traversal, another thread B can add a new order with better price concurrently (see MatchOrder & AddOrder example above), which would not and should not be considered by thread A. However, checker.py may not be aware of such concurrent operations since the output time of B’s AddOrder can be before A completes its MatchOrder execution. As a result, Checker.py will assume that B’s AddOrder happened first. Hence, validating the execution output without knowledge of when the mutexes are acquired by different threads was not sufficient in our case.