CS3211 Assignment 1 Report

Program Overview and Data Structures

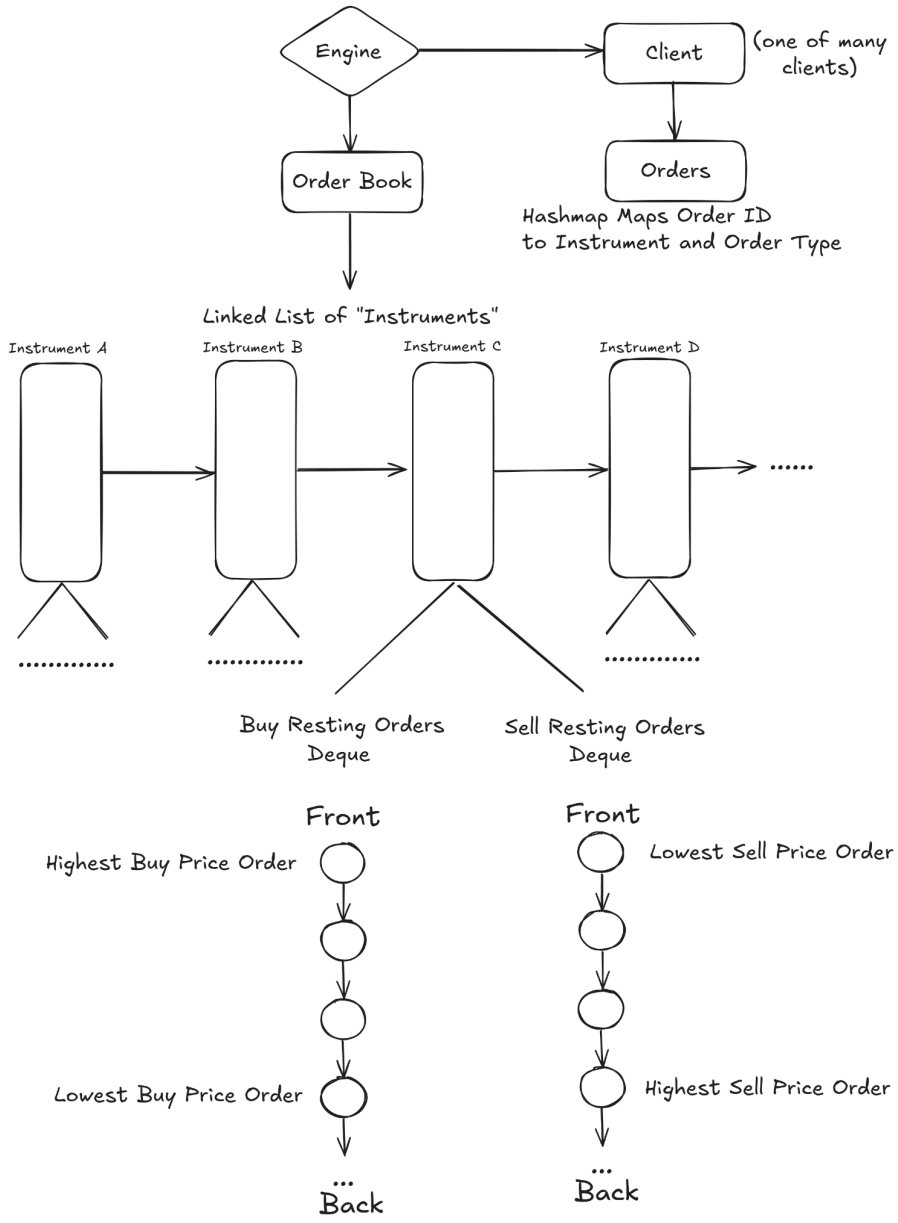

Figure 1: Overview of Program

Figure 1 illustrates a high level overview of the program. `engine.cpp` stores an instance of `OrderBook` which stores a **concurrent linked list** with hand-over-hand locking. Each node of the list holds an instance of `Instrument`, which has two **deques** for Buy and Sell Orders.

Each client thread also holds a **hashmap** that maps the order ID to instrument and order type, to track resting orders that it sent in and are allowed to cancel.

Program Concurrency

Overall, the program allows for the first type of phase-level concurrency (most of the time), where orders from the same instrument and type (e.g. multiple buys) can execute concurrently.

Phase-level Concurrency

Concurrency between orders with different instruments is enabled by giving each connection thread a reference to the corresponding instrument, which then independently processes their order. This reference is retrieved concurrently by hand-over-hand locking and traversal of the instruments linked list, with the use of mutexes for each node of the linked list.

Within each instrument, concurrency between orders of the same type is achieved by solving the single-lane bridge problem. The integer variables `buyActive` and `sellActive` are used with condition variables to keep track of the number of threads accessing the instrument and their intentions. If either count is greater than 0, only threads with orders of the same type are allowed to concurrently process their order while others have to wait.

To illustrate this, we look at an example of an incoming **buy** order for a particular instrument:

1. **Waiting for Sell Orders to Complete:**
   The buy order first locks the `accessMutex` and waits while `sellActive` > 0 to ensure no sell orders are currently being processed. It then increments the `buyActive` counter (preventing sell orders from continuing) and releases the `accessMutex`, allowing other buy orders to proceed concurrently.
2. **Matching Against Existing Sell Orders:**
   Next, the buy order acquires the `sellOrdersMutex` to safely access the `sellOrders` deque. It inspects the first element (lowest price sell order), to attempt a match. If a match is possible, the resting sell order is updated accordingly (and removed from the deque if necessary).

   The `sellOrdersMutex` is then released to allow other buy orders to be processed while the info about the order's execution is printed. Step 2 is repeated while there was a successful match against a resting order and the buy order's count > 0.
3. **Adding to the Buy Order Book:**
   If the incoming buy order isn't completely fulfilled after repeating step 2, it then acquires the `buyOrdersMutex` to append the remaining portion of the buy order to the `buyOrders` deque. After adding the order, the deque is re-sorted by descending price and by timestamp to maintain the correct priority.

   The `buyOrdersMutex` is then released to allow other buy orders to be processed while the info about the order's addition to the book is printed.
4. **Notifying Sell Orders:**
   Finally, the buy order locks the `accessMutex` again to decrement the `buyActive` counter. If no buy orders remain active (i.e., `buyActive` becomes zero), it notifies all threads waiting on the sell condition variable (`sellCond`), thereby allowing any pending sell orders to proceed.

The process for an incoming sell order is analogous, but with the roles of buy and sell, and sorting direction by price reversed.

During steps 2 and 3, the program temporarily decreases to instrument-level concurrency while the mutexes for the respective order deques are being held, as only one thread is allowed to access and match against the best resting order. This downtime is minimized by only locking the respective mutexes for the shortest time possible, releasing it before printing to the standard output, to allow for concurrent processing as much as possible.

Canceling Orders

Depending on the type of the order, the corresponding mutex is acquired before searching through the respective deque for the matching order and removing it. Similarly, the program temporarily decreases to instrument-level concurrency while this is ongoing.

Testing Methodology

Besides the basic test cases provided to us, harder test cases were created with the aid of ChatGPT. The test cases were created to test the program in increasing complexity.

1. Fully Matching Orders
2. Partial Matching Orders
3. Partial Fulfillment with Cancellation
4. Multiple Instruments, Independent Concurrency
5. Phase-Level Concurrency (Multiple Buys vs. Single Sell)
6. Multiple Instruments with Phase-Level Concurrency
7. Interleaved Orders with Sleep Delays
8. Multi-Instrument Stress Test (10 Clients)
9. Large Single-Instrument Queue with Multiple Price Levels
10. 40 Clients, All Fully Matching
11. 40 Clients, Mixed Full & Partial Matches with Cancellations

Thread Sanitizer and Address Sanitizer were utilized to test the correctness of the program alongside the created test cases. A bash script is created to run the test cases all at once for convenience.

Appendix

AI Tools Declaration

ChatGPT was used to aid in generating the test cases and debugging Valgrind.

Links to ChatGPT prompts:

- Test cases generation, test case generation queries are near the end of the ChatGPT session

  https://chatgpt.com/share/67cbfc15-ea30-800a-b708-6f053aaf95d4

- Valgrind, Helgrind, Thread Sanitizer and Address Sanitizer debugging

  https://chatgpt.com/share/67cbfe04-1b2c-800a-a401-580034d83675

  Download Link to .txt file of the full prompts

  https://drive.google.com/file/d/1tcDJWHsVUgiam3hPUqAXbLAuDdEwT9jJ/view?usp=sharing

  For this ChatGPT link, the original thread/session had an image and ChatGPT did not allow me to share it due to privacy reasons. We copied and pasted all the conversations and asked ChatGPT to summarize it. A download link to a .txt file that contains all of the original questions and answers, but it is pretty messy.

  ChatGPT was mainly used to debug the Valgrind version on the Lab machines and try to understand why Valgrind was complaining and realised that it was a version control issue. We have contacted Professor Cristina about this and she is aware of this issue.