Team: e0774645_e1324860

Description of Data Structures

The data structures used in our implementation can be organised into 3 user defined classes.

OrderBooks class

We represent the order book with the *OrderBooks* class. In the class, we maintain two unordered_maps:

1. A mapping from instrument name to an instance of *ListOfOrders* for each instrument.
2. A mapping from order ID to instrument name for each order, to get the instrument name of an order in a cancel order.

A mutex is also maintained to synchronize read and writes to these unordered_maps.

ListOfOrders class

We represent the list of orders for each instrument with the ListOfOrders class. In the class, we maintain two priority queues and an unordered_map:

1. A priority queue of resting buy *Order*.
2. A priority queue of resting sell *Order*.
3. A mapping from order ID to *Order*, to get an order quickly when cancelling an order.

The priority queue is ordered according to price-time priority rule. A mutex is also maintained to synchronize read and writes to these data structures.

Orders class

We represent a resting order with the Orders class. The class contains fields related to an order.

Synchronisation Primitives + Level of Concurrency

To synchronise between threads, our program made use of std::mutex and RAII(std::lock_guard).

All readable and editable attributes are kept with private modifiers to prevent any unwanted usage such as a thread reading/editing the data without acquiring the lock. For a thread to read or write to our data structures, it can only do so by making use of the exposed public methods. This helps us ensure the correct behaviour, as the public methods are implemented to first acquire the lock of the mutex. After the method call ends, the mutex that is wrapped in a std::lock_guard that unlocks automatically once out of scope of the method (as per RAII).

There are two mutexes defined in the program. One for OrderBooks (only 1 instance in the entire application) and one for ListOfOrders (each instrument has one instance of ListOfOrders).

There is no implementation in the program where a thread attempts to obtain both the OrderBooks mutex and ListOfOrders mutex at the same time. This helps prevent a deadlock as it is not possible to create a cycle of dependencies between two threads where each thread is trying to obtain a mutex locked by the other thread.

When matching or cancelling an order, a thread will hold the ListOfOrders mutex for the full operation. The matching method will also add any of the remaining active order into its

ListOfOrders. This ensures that at any point in time, only one command for a instrument can read and write to its ListOfOrders.

The program has Instrument-level concurrency. Since each ListOfOrders (corresponding to one instrument) has its own mutex, the execution of orders for different instruments can execute concurrently.

Testing Methodology

To test the program, first we ran the program with the various debugging tools (e.g. Valgrind, Address Sanitizer, Thread Sanitizer) to help identify any concurrency bugs such as memory leaks.

Secondly, we also created 3 custom test scripts apart from the grader scripts given. This can be found in the scripts folder. These scripts make use of the grader to test more than 1 thread.

Specifically, we tested running 2, 4, 8 threads at the same time (since 8 is a common core count for high end CPUs).

A variety of tests are also represented in the test cases, such as cancelling an order, fulfilling buy and sell orders, etc.