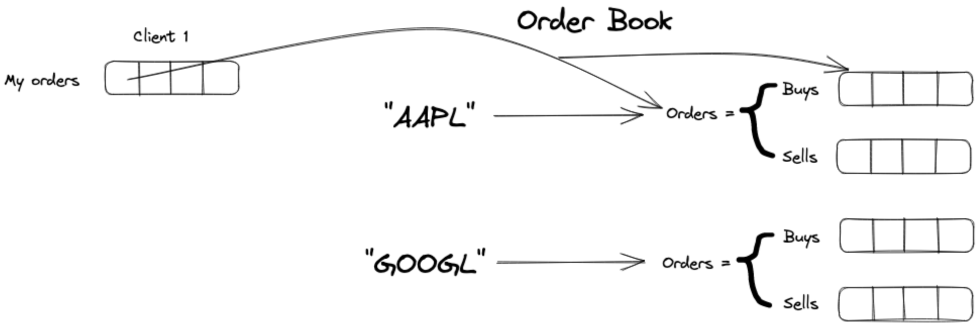
Description Of Data Structures Used

1. `Order`. Wrapper class around the given `ClientCommand` struct with 2 additional information: `time` and `execution_id`. We overrode the operators `==`, `<` and `>` to define equality and comparison.
2. `Buys` and `Sells`. 2 types of `ordered_set` containing `Order` structs where the ordering corresponds to the assignment's description for Buy side and Sell side orders.
3. `Orders`. Used to hold all `Order` of the same instrument.
    a. `Buys`: set of buy side `Order`s (as previously explained)
    b. `Sells`: set of sell side `Order`s (as previously explained)
    c. 4 mutexes. For our current implementation, only 1 of the mutex is important, which is `instr_mtx` (instrument mutex). The other 3 would have been to facilitate our phase level concurrent implementation.
4. `order_book`. Implemented as `unordered_map`.
    a. Key = `String` of instrument's name
    b. Value = `shared_ptr` to this instrument's `Orders`
    c. This groups orders of the same instrument together and separates the orders for different instruments which leads us to achieve instrument-level concurrency as explained later on
5. `my_orders`. Each client stores information about its own orders so that it can cancel only its own later on. Implemented as `unordered_map`. This is thread local so no synchronization is needed
    a. Key = `int`. It's the `order_id` from `ClientCommand`
    b. Value = `pair<shared_ptr<Order>, shared_ptr<Orders>>`
        i. First pointer points to the `Order` object of the `order_id`
        ii. Second pointer points to the `Orders` in which `Order` can be located

Here is a simple illustration to visualize the above:



Synchronization Primitives Used

<u>Atomics</u>

`timestamp` is an atomic unsigned integer. It is shared among all threads hence we need it to be atomic. We used the default memory order `memory_order_seq_cst` for our operations on it since it has to be completely consistent for all outputs.

<u>Mutexes</u>

1. `oob_mutex`: a `shared_mutex` to implement reader-writer synchronization for `order_book`
2. `instr_mtx`: a `mutex` contained in each `Orders` struct. It synchronizes accesses to the struct's `Buys` and `Sells`.

No mutex is needed to synchronize accesses to `my_orders` as it is a thread local storage and is only accessed by that thread so it will not lead to any race conditions.

<u>How we use the mutexes</u>

`oob_mutex`:

The first step of processing an order is to check if its corresponding `Orders` in the `order_book` exists. This is a read operation. If it doesn't exist yet, this order must have been the first order for this instrument so we will have to insert an entry into `order_book`. This is a write operation. If it exists, we can get a `shared_ptr` to the `Orders` object and proceed with executing the order (`instr_mtx` section).

This results in a reader-writer problem which we solved using `oob_mutex`. As it is a shared mutex, we will use `shared_lock` for read operations and `unique_lock` for write operations.

Using `unique_lock` will only allow 1 writer access which temporarily reduces the level of concurrency we can have. Hence, through fine-grained locking, we ensure that the `unique_lock` will be released as soon as the write operation is completed. Thus allowing for more concurrent reads again immediately after.

`instr_mtx`:

Each `Orders` has an `instr_mtx` which synchronizes access to its attributes, `Buys` and `Sells`. When processing an order of any type (buy/sell/cancel), it has to first acquire this `instr_mtx` before it can be processed. It will then release it after it has finished processing, allowing for the next order to be processed.

Explanation Of Level Of Concurrency

Level of Concurrency: Instrument-Level Concurrency

We achieve instrument level concurrency through our data structures and our synchronization primitives.

`order_book` groups orders for the same instrument as explained previously.

`oob_mutex` allows multiple concurrent read operations to `order_book` to access its corresponding `Orders`. Since each `Orders` has an `instr_mtx` and multiple concurrent accesses to different `Orders` are allowed, we reach instrument-level concurrency as orders for different instruments can execute concurrently but orders for the same instrument are serialized by `instr_mtx` which allows only 1 order for each instrument to be processed at any time.

Explanation Of Testing Methodology

Testing Overall

We also built custom_runner.cpp to simulate concurrent clients (uses `std::barrier` to wait for all threads to be created before running tests). Build by running `clang++ -g -O3 -Wall -Wextra -pedantic -Werror -std=c++20 -pthread custom_runner.cpp` in scripts/custom_runner folder.

Run via `./a.out ../../socket <paths to input files>`

Small Testcase (scripts/custom_runner/small_multi/)

2 clients. 1 made 2 buy orders (c1.in), the other made 2 sell orders (c2.in)

We used custom_runner (./a.out) to run them simultaneously.. We passed the grader's correctness test with no issues from TSAN and ASAN.

Large Testcase (scripts/custom_runner/random_buys_sells_cancels)

We used order_gen_random_buy_sell_cancel.py to generate random orders for 8 clients. 2 instruments, 4 clients for each instrument. The python script lets us vary the proportion of buy to sell to cancel orders, and lets us vary the price within a range.

We ran the grader format version of the test case (random_bsc_combined.in) and passed the correctness test.

We ran them using custom_runner (random_c0_0.in … random_c7_7000.in). Helgrind reported issues for the large testcase but we believe this is due to the large number of writes to the stdout buffer while the OS is trying to flush it. This is supported by the fact that the medium version of this testcase (scripts/custom_runner/random_buys_sells_cancels_medium/random_c0_0.in … random_c3_300.in) and other test cases results in no Helgrind issues.

Testing `oob_mutex`

1. Testcase: ./grader engine < scripts/concurrent-sell_then_concurent_buy_medium.in
    a. First, 20 threads send sell orders for unique instruments to simulate concurrent reading and writing to `order_book`. There are 727 unique instruments
    b. Then, the same 20 threads send matching buy orders for each instrument to simulate concurrent reading to `order_book`
2. Testcase: ./grader engine < scripts/concurrent-buy_then_concurent_sell_medium.in . Similar to above but in the other way around

We ran this 5-10 times with and without the locks for `oob_mutex` with TSAN (TSAN_OPTIONS="history_size=1 force_seq_cst_atomics=1") and ASAN to detect issues like data races and use-after-free. Without, data race issues were reported. With lock, no issues.

Testing `instr_mtx`

1. Testcase: ./grader engine < scripts/instr_concurr_test_medium.in
    a. 8 clients, 4 clients per instrument. For each instrument, 2 clients send buy orders and other 2 clients send sell orders. This simulates multiple orders trying to be executed concurrently which should be prevented by `instr_mtx`

We ran this similarly to before and found issues without the lock. No issues with the lock.