**Description of Data Structures Used**

---

SafeHashmap:
`SafeHashMap` encapsulates an `unordered_map`. It allows multiple readers to call functions that read from the `unordered_map`. However, only 1 writer can modify the `unordered_map` at any time. `SafeHashMap` uses a `mutex` and a `semaphore` to maintain this invariant.

SingleLaneBridge:
`SingleLaneBridge` allows multiple buy orders and cancel buy orders in, or multiple sell orders and cancel sell orders, but not >= 1 buy/cancel buy and >= 1 sell/cancel sell orders in the `OrderBook` at any time. `SingleLaneBridge` uses a `mutex` and a `conditional_variable` to maintain this invariant.

OrderBook:
`OrderBook` stores resting (buy/sell) orders in 2 `std::map`. This allows us to access the orders with the 'best' price first. `OrderBook` as a mutex to protect `std::map` when adding a new resting order.

Timer:
A global struct to take a timestamp. It is just a static atomic int counter being incremented each time Timer::getTime() is called.

Key:
Struct containing the immutable info of a resting order namely: price, time of being added into OrderBook, order_id, and which side (buy/sell) the order is. This allows `std::map` to sort the resting orders in decreasing 'best' price.

Store:
Struct containing `count` i.e the remaining order count and `eid`, the next free eid to use when a match occurs

AtomicNode:
Simply a wrapper around `atomic<Store>`. Needed because for `std::map<Key, T>.insert(value_type&&)`, T needs to be copy constructible.

**Synchronisation primitives used**

---

atomic<T>, mutex, lock_guard, unique_lock, conditional_variable, counting_semphore

## Phase Level Concurrency

**Methodology**

Our implementation achieves phase-level concurrency by allowing multiple buy orders and buy cancels or multiple sell orders and sell cancels of the same instrument to execute simultaneously. This is achieved by leveraging C++ atomic instructions.

Consider the case where multiple threads are executing the `matchBuy` function call for the same instrument (similar reasoning can be analogously applied to multiple sell threads executing concurrently). Every thread will traverse the resting sell orders beginning from the best available price. Multiple buy threads are allowed to attempt to match with the same resting sell order; each of these threads will repeatedly do the following:

- Atomically load the structure containing information regarding the available quantity and execution id of the said resting sell order. If the quantity is empty (meaning this resting order has already been fully filled), the thread exits the repeat and moves onto the next best available resting order.
- Create a copy of the loaded quantity and execution id values and make changes to the copy according to the matching behavior. The new values are `{count - c, execution_id + 1}`, here c is the quantity of orders matched.
- Take a timestamp here.
- Attempt to `compare_exchange_strong` using the loaded original value as the expected value and the updated values as the new replacement.
- If the `compare_exchange_strong` succeeds, this means that no other thread matched this resting order since the time the structure was loaded. This indicates a successful match between the active order of the current thread and this resting order. The thread can exit the loop.
- If the `compare_exchange_strong` fails, this means that some other thread matched this resting order since the time the structure was loaded. The local updates made by the current thread are discarded and the attempt is repeated.

Even though the threads may repeat the above loop more than once, they are nonetheless guaranteed to either break out due to zero remaining quantity or successfully execute the `compare_exchange_strong` atomic instruction. Evidently, within a group of buy/sell commands for the same instrument, each thread is actively executing instructions and moving forward in its execution rather than being indefinitely blocked or stalled, fitting the CS3211 course's definition of concurrent progress.

Firstly, there is no ABA problem, as execution id only increases and count only decreases So if compare_exchange is successful, it means no other threads match with the resting order.

Secondly, all our atomic instructions are `seq_cst`. Hence successful matches to a resting order are in the following order: `load(), timestamp(), compare_exchange(), load(), timestamp(), compare_exchange()`. Failed matches do not affect the

quantity and execution id. Each block of `load()`, `timestamp()`, `compare_exchange()` is called by 1 thread. This means that the order of execution_id matches the order of timestamps. Thirdly, if a thread moves to the next elem, it must have seen all the successful matches done before. When it takes a timestamp when matching with the next order, it will be after the timestamps for the previous resting order, as our atomic operations are all `seq_cst`. Similar reasoning when a thread sees no more suitable resting orders and then adds a resting order.

Furthermore, for cancel orders, we allow them to enter the matching state together with their respective types (cancelling buy resting orders can be executed concurrently with buy orders and cancelling sell resting orders can be executed concurrently with sell orders). However, the actual removal from the orderbook is done serially with other modifications (see next section).

**Drawback Considerations**

Because mutual exclusion is used when modifying the `buyBook` and `sellBook` of each instrument, adding to or deleting from the orderbook will be done serially. Therefore, while multiple threads of buy / sell are matching with resting orders, the actual updates are done one by one.

**Testing Methodology**

---

We wrote a script, `scripts/gen_test.py` to generate large test cases for the grader

We also wrote test cases for `SingleLaneBridge` to test its correctness. We created multiple threads that called `enterBuy()`, `leaveBuy`, `enterSell()`, `leaveSell()` and `printf` statements and manually verified that no orders from the opposite side can be in the 'critical section' at any time. The test cases are called `Test1-SLB.cpp` and `Test2-SLB.cpp`.

We compiled `engine` with `ThreadSanitizer` and `AddressSanitizer` to check for data races and invalid memory accesses/memory leaks respectively.