Data Structure	Remarks
`OrdersMap`	Maps instrument name → `Order`.
`OrderBooksMap`	Maps instrument name → `OrderBook`
`OrderBook`	- Each `OrderBook` keeps track of resting buy and sell orders for a given instrument using a Buy `Book` and Sell `Book` - Creates and reads from a Channel of `Order` (`OrdersChn`) where clients pass their orders that need processing to.
`Book`	`Book` stores and organizes its orders by maintaining a linked list of `OrderNode`s sorted by price.
`OrderNode`	Each `OrderNode` represents a given price and consists of a list of orders for that given price, sorted by their completion time when they are added.
`Order`	Abstraction for each individual `Order`
`Message`	A common DS to manage all output types (ie. Cancel, Execute, Add). Contains corresponding formatters and takes in their required arguments to be printed as our program output.

Above is a summary of data structures we used to keep track of orders & manage our output. We will be referencing them below as we provide more details on our implementation. Our ConcurrentHashMap is self-implemented, utilising the Read-Writer No Starve implementation from Lecture 8 slides 10-12. In our implementation below, all locks that we used are implemented using a buffered channel of size 1.

Implementation Details

We will be providing a high level idea on how our implementation works and how we make use of channels and goroutines to enable concurrency & support concurrent execution of orders coming from multiple parallel clients.

1. Client Initialisation and `Order` parsing

Each time Engine accepts a new client connection, a goroutine is spawned to handle orders from the client. To identify each client uniquely, every client is assigned a unique ID. Each client must obtain a lock before being able to read and update the id global variable. Upon obtaining an id, the engine can then process incoming orders from the client.

1. For Buy/Sell orders, the Engine will look up `OrderBooksMap` based on the instrument specified in the `Order` to get the `OrderBook` for the instrument. If the `OrderBook` does not already exist, a new `OrderBook` is first created for the new instrument. The `Order` is then written into the `OrdersChn` of the `OrderBook`.

2. For Cancel orders, the Engine first performs certain validations. Firstly, the Engine looks up `OrdersMap` based on the orderId specified in cancel `Order` to get the actual `Order`. If it does not exist, the cancel is invalid and rejected. Secondly, the Engine checks that the client ID stored in the actual `Order` matches that of the client that sent the cancel `Order`. If it does not match, the cancel is invalid and rejected. If these validations pass, the Engine processes the cancel `Order` similar to Buy/Sell `Order` as mentioned above.

2. How the `OrderBook` works

Whenever an `OrderBook` is instantiated, it spawns a new worker goroutine which constantly reads for incoming orders written to `OrdersChn`. This allows for concurrency as different `OrderBook` can operate independently & concurrently.

Upon receiving the orders, it will perform either 1. `ProcessCancelOrder`(received cancel `Order`) or 2. `ProcessActiveOrder` (received Buy/Sell `Order`). Both `ProcessCancelOrder` and `ProcessActiveOrder` are spawned as a goroutine. This enables greater concurrency for multiple non-conflicting orders of the same instrument.

It is important to note that each `OrderBook` has a lock and each `OrderNode` also has its own lock as well.

2.1 ProcessCancelOrder

No need to acquire `OrderBook`'s lock. Traverses the `OrderNode` (locking them in a Hand-over-Hand mechanism) to find `OrderNode` where `Order` resides before deleting it.

2.2 ProcessActiveOrder

2.2.A. Match/Execute Orders

Each `Order` will acquire `OrderBook`'s lock before it proceeds to acquire required locks of `OrderNode`s it needs for matching, execution and adding of itself into the book if needed. Then, it will release the global lock. Every execution operation is spawned as a goroutine. This helps to increase concurrency and achieve greater concurrency in 2 ways:

1. Multiple execution/add operations working on different `OrderNode`s can happen concurrently.
2. Facilitates quicker release of the `OrderBook`'s lock to allow another `Order` to enter and acquire required `OrderNode`s' lock if available. The acquired `OrderNode`'s lock serves as a critical section and will be released within the spawned goroutine itself once the operation is completed.

2.2.B. Add Orders

Similarly for add operations, we first acquire the lock of the head `OrderNode` of the `Book` that it will be added to, before spawning a goroutine to perform the actual add operation. The add operation traverses the `OrderNode`s using the Hand-over-Hand mechanism to find the position to add the new `Order` to.

3. Managing Output

Here we discuss how we serialise the outputs, manage their output timestamp and execution Id.

3.1 Serialisation - Use of channels of channels

Similar to the tutorial approach, we utilise the concept of promises and implement it using a global channel of channels. We spawn a goroutine in main.go which constantly reads from this channel of channels and prints their messages in the same order that is read. Each operation that produces an output will create an output channel of `Message` and write it into the global channel of channels.

To serialise outputs across multiple orders operating on the same `OrderBook` / `OrderNode`, we write the output channel into the global channel of channels only after acquiring the relevant lock needed to complete its operation (ie. execute, add, cancel). This ensures that the order of execution of the `Order` is the same as the order in which the output channel is passed to the global channel of channels, which will then be read and printed in that same order.

3.2 Output Timestamp & Execution Id

As we have established that the channel of channels of outputs are read in the correct order by a single goroutine (no race/data conditions), we simply get the current timestamp when we read from the output channel and pass it to the message before printing. Similarly for the Execution Id, we increment & update the Execution Id to pass to the output message when it is required.

Go Patterns

We employed the following go patterns identified from Lecture 7, slide 2 in our implementation:

Fan-out, Fan-in	For-select Loop	Preventing goroutine leaks
The execution of orders is fanned out at multiple stages. Execution is first fanned out at instrument level as each `OrderBook` has its own worker goroutine. Next, within each `OrderBook`, executions are further fanned out at `Order` level as we spawn a goroutine to process every `Order`. The output of each `Order` will then be fanned into the single channel of channels which serialises the output.	1. Utilised by the worker goroutine of each `OrderBook` to read incoming orders from clients. 2. Utilised by the worker goroutine that serialises & prints output messages by reading from the channel of channels.	The context created in main.c is passed to all workers. Each worker goroutine checks for ctx.Done() case in it's for-select loop. Whenever the context is cancelled, the workers are also able to terminate and release resources.

Testing

We made use of the same generator & checker script as A1. We printed the thread ID in the output for Add & Cancel orders for testing and updated the checker to identify the thread ID of both the client calling the cancel `Order` and the client who created the `Order` to be cancelled.

The checker script first parses & sorts the output based on the output time. It then maintains the state of the orders based on previous output orders. It ensures that each `Order` operation is valid based on this maintained state.

We tested up to 10 clients running concurrently with a total of 1million randomly generated orders (ie. buy, sell, cancel) evenly distributed between the clients. We have 5 different instruments with a price range within $5 difference for the orders. This is to get a high rate of overlaps between different orders to test the concurrency.