Assignment 1: Exchange matching engine in C++

In this assignment, you will implement an exchange matching engine, using various concurrency mechanisms, in C++.

1 Matching engine

A matching engine is a component that allows the matching of buy and sell orders inside an exchange. When an exchange receives a new order, it will first try to match this order against existing orders, and, in case it cannot match it, will store it in an order book, so that it can potentially match it later. If an order is added to the book, we call that order 'resting'.

Your exchange will match orders using the *price-time* priority rule. This rule for matching two orders on an exchange is expressed using the following conditions – which must all be true for the matching to happen:

- The *side* of the two orders must be different (i.e. a buy order must match against a sell orders or vice versa)
- The *instrument* of the two orders must be the same (i.e. an order for "GOOG" must match against another order for "GOOG")
- The *size* of the two orders must be greater than zero
- The *price* of the buy order must be *greater or equal* to the price of the sell order
- In case multiple orders can be matched, the order with the earliest arrival time will be matched first. This is to ensure fairness towards the orders that have waited the longest.

Orders can be partially matched in case the size of the other order is lower. Consequently, orders can be matched multiple times.

1.1 Functionality

In the commands below, the following terms are used:

- Order ID: a unique ID across all orders
- Instrument: the instrument i.e. symbol of the order, up to 8 characters
- Price: the price of the order
- Count: the size of the order

Your engine should handle these input commands:

- New buy or sell order
  - Arguments: Order ID, Instrument, Price, Count (Size)
- Cancel order
  - Arguments: Order ID

In response to input commands, the following output actions can be done:

- Order added to order book
  - Arguments: Order ID, Instrument, Price, Count (Size), Side (buy or sell), Timestamp when the Order ID is received, Timestamp when the action is completed.
- Order executed
  - Arguments: Resting order ID, New order ID, Order execution ID, Price, Count (Size), Timestamp when the New order ID is received, Timestamp when the action is completed
  - Order execution ID should be an ID starting from 1, incrementing serially each time a particular resting order is matched.
    - e.g. if there is a resting order with ID 123, and it is matched three times, then there should be executions with (Resting order ID, Order execution ID) (123, 1), (123, 2), and (123, 3).
- Order deleted
  - Arguments: Order ID, Cancel state, Timestamp when the cancel request is received, Timestamp when the action is completed
  - Cancel state is either Accepted (A), if the order is deleted, or Rejected (R), if the order was already deleted or fully filled

When a new order is created, the engine should first try to match it against one or more existing (resting) orders. When that happens, the exchange matching engine should first write the execution message for those matches, and then followed by the messages for any new entries added to the order book, if the new order is not fully filled.

Examples below are given in the input format of the provided client program, and the output format as required (both described below), except that the timestamps required in the output are omitted for clarity.

▶ **Example 1: New order fully matches against two resting orders**

▶ **Example 2: New order partially matches against two resting orders and then is added**

▶ **Example 3: Orders cancelled**

The matching engine receives connections and commands from multiple clients at the same time. The commands received from a client must be executed (committed) in the order sent by the client (no re-ordering is acceptable on the commands sent by a client).

1.2 Requirements

You should design and implement appropriate concurrent data structures with appropriate synchronisation mechanisms that will ensure the correctness of your matching engine's results. Aim to write an implementation that enables the maximum possible concurrency and parallelism e.g. by making locks fine-grained, by using lock-free techniques, etc. **The focus of the assignment is on maximising concurrency, and not simply raw performance.**

Please note: the provided skeleton already handles input/output as required, but a description of the I/O protocol is provided below anyway.

You will implement a server in C++ that accepts the above commands from clients over Unix domain stream sockets. Your server application should accept a single argument, which is the path to the Unix domain socket to listen on i.e. it should be invoked like `./engine path/to/socket`.

The server should be able to accept an unbounded number of connections, and an unbounded number of commands from each connection.

Each input command will be of the format described by this C structure:

```
enum input_type { input_buy = 'B', input_sell = 'S', input_cancel = 'C' };

struct input {
  enum input_type type;
  uint32_t order_id;
  // the following fields are present but ignored for a cancel command
  // i.e. all commands are the same size
  uint32_t price;
  uint32_t count;
  // up to 8 chars + null terminator
  char instrument[9];
};
```

In response to commands, actions should be written to the server's standard output in the following format:

- Order added
  - `<Buy/Sell> <Order ID> <Instrument> <Price> <Count> <Timestamp received> <Timestamp completed>`
  - The `Buy/Sell` field should be `B` if the order is a buy order, and `S` if it is a sell order.
  - The `Timestamp received` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the `Order ID` was received by the server.
  - The `Timestamp completed` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the `Order ID` has finished recording in the order book and this action was sent for printing.
  - The timestamps should be used relative to each other, and they do not represent the absolute time. Using these timestamps you may compute the elapsed time for processing a command.
- Order executed
  - `E <Resting order ID> <New order ID> <Execution ID> <Price> <Count> <Timestamp received> <Timestamp completed>`
  - The `Timestamp received` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the `New order ID` was received by the server.
  - The `Timestamp completed` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the execution has finished and this action was sent for printing.
- Order deleted
  - `X <Order ID> <Accepted?> <Timestamp received> <Timestamp completed>`
  - The `Accepted?` field should be `A` if the cancel is accepted, or `R` if the cancel is rejected.
  - The `Timestamp received` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the cancel command was received by the server.
  - The `Timestamp completed` is the monotonic clock (`CLOCK_MONOTONIC`) at the time when the cancel command was executed and this action was sent for printing.

As mentioned earlier, the provided skeleton already implements the above I/O specification for you, with an implementation using normal synchronous I/O functions and one thread per connection, which is intended to be simple and to encourage a concurrent implementation that can achieve speedup via *parallelism* rather than simply by raw performance (such as by using asynchronous I/O, etc.). You are free to modify the I/O to suit your own needs while adhering to the protocol described above, with the restriction that **each client connection must be handled on a unique thread** (as is done by the provided I/O implementation).

1.3 Provided skeleton

The following files are provided.

Files you will likely need to modify:

- `engine.cpp`: An empty `Engine` implementation that creates a new thread when a connection is received, and then simply reads commands and dumps them to standard output. You will likely need to modify `Engine::ConnectionThread`.
- `engine.hpp`: Declarations for `Engine`.
- `Makefile`: GNU makefile. If you add any new source files, add them to the `SRCS` variable declaration on line 9.

Files you should not need to modify:

- `io.cpp`: Provided I/O implementation, C++ part
- `main.c`: Main function and provided I/O implementation, C part
- `io.h`: Shared declarations for I/O
- `client.c`: A client that reads commands from standard input in the format described below and sends them using the prescribed protocol to the server.

To build, simply run `make`. You are free to change, add or remove any files as you wish. Make sure to update the `Makefile` if you do so, so that we are able to build your program.

To run the engine, run e.g. `./engine socket`, as described above. The server will run and bind to the Unix domain socket at `./socket`. Then you can use the provided client to send commands to the engine, described next.

1.3.1 Provided client

There is a provided client in `client.c`. To run it, run e.g. `./client socket`. The client will read commands from standard input in the following format:

- Create buy order
  - `B <Order ID> <Instrument> <Price> <Count>`
- Create sell order
  - `S <Order ID> <Instrument> <Price> <Count>`
- Cancel order
  - `C <Order ID>`

There are some example input files `*.input` provided.

You are encouraged to write your own clients that e.g. generate commands programmatically, etc., in order to test your implementation more thoroughly.

2 Submission

Your program must be written in C++20. If you modify or write your own I/O, you can write that in C, but the actual engine/data structures/etc. must be in C++20. You may not use any libraries aside from the C++ standard library and POSIX libraries.

Your program must work on the lab machines provided Parallel Computing Lab at COM1-B1-02, which are running Ubuntu 20.04. Check the appendix for further details about how to use the lab machines.

With your submission, include a writeup as a PDF of up to two A4 pages including the following:

- a description of your data structure(s) used to keep track of the orders. Explain how your data structures enable the concurrency.
- an explanation of how you support the concurrent execution of orders coming from multiple parallel clients. This description is essential for our grading, as such it should match your implementation and help us understand
- a description of how you tested your code with multiple clients and orders. Feel free to add any scripts or test case file(s) to your submission, under folder `scripts`

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized. Cite your references or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.).

Submit your assignment before the deadline under LumiNUS Files. Each group must submit ONE zip archive named with your student number(s) (`A0123456Z.zip` if you worked by yourself, or `A0123456Z_A0173456T.zip` if you worked with another student) containing the following files and folders.

Your zip archive must contain **only** the following:

- all source and header files that comprise your program
  - Please include files from the skeleton that are required by your program, even if you do not modify them.
- a `Makefile` whose default target produces an `engine` binary conforming to the requirements described above
  - The provided `Makefile` conforms to this.
- the 2-page writeup as a PDF named `A0123456Z_report.pdf` or `A0123456Z_A0173456T_report.pdf`
- an optional folder, named `scripts`, containing any additional scripts and testcases that you might have used for the assignment.

Do not include compiled binaries or object files in your submission archive.

It is required to use a GitHub repository for your team. Access **GitHub Classroom** at this link to create such a repository. Name your team `A0123456Z` (if you work by yourself) or `A0123456Z_A0173456T` (if you work with another student). We have enabled automatic checks for these repositories to ensure that we can compile and run your code after submission. You may check the status of these checks under Actions for your repository. Check the appendix for further details about GitHub Classroom.

Submit your zip archive to the **Assignment 1 Submissions** folder before **Fri, 25 Feb, 2pm**.

All FAQs for this assignment will be answered here. If there are any questions you have that are not covered by the FAQ document, please post on the LumiNUS Forum.

Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade.

A penalty of 5% per day (out of your grade) will be applied for late submissions.

3 Appendix

3.1 Accessing the lab machines

Before using the SoC compute services (i.e. computer cluster nodes, login to Sunfire, etc), you **should enable SoC Computer Cluster from your MySoC Account page**.

To access the lab machines, ssh into ***Sunfire*** using the command below. Enter your ***SoC account and password*** when prompted.

`> ssh your_soc_account_id@sunfire.comp.nus.edu.sg`

Next, from Sunfire, ssh into the lab machines using the command below. ***Enter your NUSNET id (e012…) and password*** published in LumiNUS Gradebook when prompted.

`> ssh your_NUSNET_id@hostname` (# `hostname` might be `soctf-pdc-004.comp.nus.edu.sg`)

Once entered, you should be looking at the shell of the remote lab machine, and you will be placed at the home directory of your account.

Note that you should use your SoC account and password to access Sunfire, but use your NUSNET id and password published under LumiNUS Gradebook to login to the lab machines.

To login without inputting a password, configure SSH key-based authentication. Use `ssh-keygen` and `ssh-copy-id` to copy the public key to the remote node.

3.2 List of machines

- `soctf-pdc-001` - `soctf-pdc-008`: Xeon Silver 4114 (10 cores, 20 threads)
- `soctf-pdc-009` - `soctf-pdc-016`: Intel Core i7-7700 (4 cores, 8 threads)
- `soctf-pdc-018` - `soctf-pdc-019`: Dual-socket Xeon Silver 4114 (2*10 cores, 40 threads)
- `soctf-pdc-020` - `soctf-pdc-021`: Intel Core i7-9700 (8 cores, 8 threads)
- `soctf-pdc-022` - `soctf-pdc-023`: Intel Xeon W-2245 (8 cores, 16 threads)

Direct login is enabled to all lab machines, but some nodes are managed by Slurm Workload Manager. You can use Slurm to submit your batch jobs.

- Direct login (no Slurm)
  - `soctf-pdc-001` - `soctf-pdc-004`,
  - `soctf-pdc-010` - `soctf-pdc-012`,
  - `soctf-pdc-018`,
  - `soctf-pdc-020`,
  - `soctf-pdc-022`,
  - `soctf-pdc-023`
- Direct login, and managed with Slurm as well:
  - `soctf-pdc-005` - `soctf-pdc-009`,
  - `soctf-pdc-013` - `soctf-pdc-016`,
  - `soctf-pdc-019`,
  - `soctf-pdc-021`

Updated Slurm guide can be found here.

To view the usage of the lab machines, you can use this Telegram bot: `@cs3210_machine_bot`. Simply type `/start` to get a real time status update for all machines.

3.3 GitHub Classroom

Visit the assignment link.

The first member of the team to access this link will be prompted to accept the assignment that gives your team access to the new repository. Create a new team by typing `A0123456Z` or `A0123456Z_A0173456T`, using the student numbers of the students forming a team. Note that the naming convention must be followed strictly, e.g. capitalisation, dash, and spacing. Use the same name for your LumiNUS submission for easy identification.

The other member in the team will be able to see an existing team with your team id under "Join an existing team" section.

The repository is automatically populated with the skeleton code provided for this assignment.