Name: Jack Lu
Last Update: 11/14/2020
Email: jianlu2001@yahoo.com
C++ standard: C++17
GibHub full source codes: https://github.com/jacklu2030/TradingEngine-Ultra-Low-Latency

Attempt to Implement Ultra-Low Latency Trading Engine using C++17

This trading system contains three major components: 1) Market Data Engine, 2) Order Book/Matching Engine, 3) Post-Trade Engine (architecture diagram at end of this doc)

1. Data Structure and order matching logic

A) At the top level is a map with key=ccyPair, value=Limit object. Each Limit object represents one specific limit price of either buying or selling. Each Limit object holds a duque containing orders of the same limit price in sequence order.

B) An Order Book is created for each ccyPair. Within each order book, there are two BST trees: one with highest buying limit price on top, one with lowest selling limit price on top.

C) Upon entry of each order from the market event pipe or stream, it’s inserted into the ring buffer, which is dequeued int the trading engine, where the highest top buying Limit object is matched against the lowest limit selling price at the top of the buying/selling BST trees. Orders within the top Limit objects are matched against each other in the sequence of entry, results are printed to console.

D) At the end, all unmatched orders or partially filled orders from all Limit objects are collected, sorted in entry sequence, and printed to console.

Note, a LMAX Disruptor style ring buffer is adopted to pass market events / orders from producer to consumers. LMAX ring buffer reuses pre-allocated memory, which is better than a queue where objects are created and thrown away once consumed.

2. Performances

The number n below is the number of currently active limit prices in the market (it should be fairly small)

A) execute an order with brand new Limit price: O(log(n))
B) execute an order whose limit price currently exists in the order book: O(1)
C) update an order: O(1)
D) Cancel an order: O(1)
E) Perform order book analysis: O(log(n))
F) Obtain highest bid / lowest ask orders: O(1)

3. Thread safety

A) Consumer thread pool is created, but it may race trade executions concurrently, trade execution may finish in different order than their entry sequential order. So, using single thread with thread affinity on CPU core may be a good option.

B) A custom class “ThreadSafeQueue” is created to pass trade orders from producer to consumer. Lockless algorithm is adopted instead of using thread wait/notify.

C) Spinlock is used instead of mutex lock on relatively short tasks for faster performance. With small thread pool(or even single thread consumer), thread contentions happen much less frequent

D) One ccyPair per mutex is adopted, lock scope is limited to order execution per ccyPair, per Limit price.

E) Memory false sharing at CPU L1/L2 caches is optimally minimized when accessing Order objects sequentially in ring buffer by a single consumer thread

F) Below is how consumer thread pool size can be adjusted:

```
 size_t consumer_thread_pool_size = 1;
OrderConsumer::getInstance().startConsumerThreadPool(consumer_thread_pool_size);
```

4. Memory usage

This project totally avoids allocating memory on the heap, it uses stack frame for fast performance. Taking advantage of high performance frame memory is the #1 design priority. LMAX Disruptor ring buffer is the only one pre-allocated on heap with fixed size.

5. Test run the trading engine, review trading results.

A) How to test run on linux

```
cat sample_input.txt | ./run.sh
```

- Docker container of Debian:10 will be created by the command above, then C++ codes are compiled and deployed, and run;
- Modify input stream data in sample_input.txt (format: unique_order_ID, buy/sell, ccyPair, shares, price)

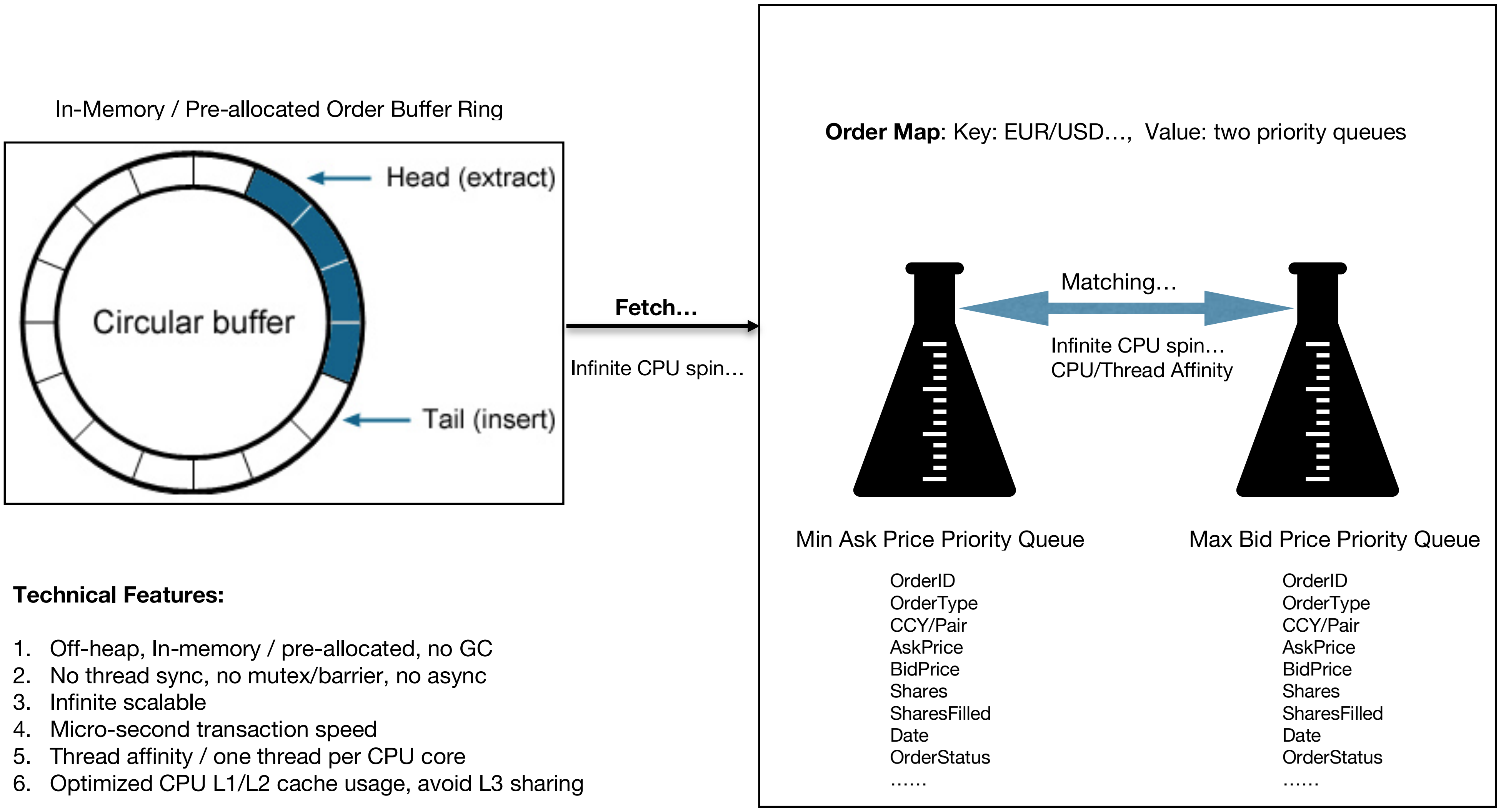
B) Review trade results – all traded are listed. At the end, un-filled / partially-filled orders in the order book is dumped to screen.

```
***** Trades filled / partially-filled -- Mon Dec 21 20:06:07 2020

TRADE BTCUSD abe14 12345 5 10000
TRADE BTCUSD abe14 13471 2 9971
TRADE ETHUSD plu401 11431 5 175

***** Dump of Order Book Content -- Mon Dec 21 20:06:07 2020

zod42 SELL BTCUSD 2 10001
13471 BUY BTCUSD 4 9971
11431 BUY ETHUSD 4 175
45691 BUY ETHUSD 3 180
```

Please modify the input data stream in sample_input.txt, and verify the results to match what you have expected.

Order Book / Order Matching Engine

Designer: Jack Lu
Date: Sept. 2020



Technical Features:

1. Off-heap, In-memory / pre-allocated, no GC
2. No thread sync, no mutex/barrier, no async
3. Infinite scalable
4. Micro-second transaction speed
5. Thread affinity / one thread per CPU core
6. Optimized CPU L1/L2 cache usage, avoid L3 sharing