Matching Engine Specification

Summary

A price-time priority limit order matching engine is responsible for determining when buy and sell orders transact (see Definitions below if unclear).

High frequency trading has pushed the limits of legacy matching engine implementations at the NYSE, CME, and other major exchanges. Proprietary trading groups can benefit from an internal matching engine to trim execution costs. Electronic market makers need to locally reconstruct the order book to determine when to provide liquidity. At the core of all these is the matching engine you will be writing.

Implement the interface provided in engine.h in a file engine.c for your entry. An example engine.c is provided. Rename or delete it before writing your own.

Language: C (gcc 4.4.5)

Starter Code: match.zip
Includes: API interface, autotester, makefile, simple example solution, type definitions, assumptions.

Starter Code Descriptions

engine.h

init()
Called once before the first orders are submitted. Initialize your data structures here. Set the next order id to 1.

destroy()
Called once after the last order is submitted. Free dynamically allocated memory here and cleanup. Calling init() after destroy() should completely reset the matching engine.

limit()
Called when a trader submits a new order. Check whether the order crosses with any existing orders. If so, call execution. If not, queue the order up with the others that have been submitted. Return the next order id and then increment it. The first order id returned is 1.

cancel()
Remove an order from the queue, identified by the order id returned from limit().

execution()
Report that a trade occured. Called twice whenever two orders cross, once with the buyer's name and side, and also with the seller's name and side. The two calls to execution may be in any order.

types.h

t_orderid
  Order identification numbers which are returned by limit(). Used to uniquely identify the order for the sake of cancellation.

t_price
  Price of a share. In the range 0-65536, and interpreted as divided by 100. Eg the range is 000.00-655.36. Eg the price 123.45 = 12345; the price 23.45 = 2345; the price 23.4 = 2340

t_size
  Number of shares.

t_side
  Boolean representing whether a bid (0) or ask (1).

t_order
  Order submitted to the matching engine containing symbol, trader name, side, price, and size.

t_execution
  Execution report sent to trader informing them of the symbol, side, price, and size of the transaction.

limits.h

MAX_PRICE
  Assume limit() will never be called with an order whose price field is greater than this number.

MIN_PRICE
  Assume limit() will never be called with an order whose price field is less than than this number.

MAX_LIVE_ORDERS
  Assume there will be no more than this number of uncrossed orders sitting on the book at any time.

STRINGLEN
  Assume the trader names and order and execution symbols will be this long.

Makefile

all
  build the implementation in engine.c and the test script. Then call ./test to run the auto testing script.

clean
  remove engine.o, executables, and emacs backups

derived.h+c

Not required. These are provided to show you how one can derive interesting order types that are offered by many exchanges from the basic limit() and cancel() api.

engine.c + double_link_list.c

An example implementation that is not optimized. It treats the bid and ask queues as doubly linked lists. This makes limit() O(n), cancel() O(n) and in addition the constant factors, ignored by big-O, are quite high.

test.c

Script to automatically test your implementation's logic.
After implementing engine.c, call:
$ make all
$ ./test
Fix the errors before submission or your entry will be disqualified. Tests are incremental so they can be used to check implementation progress.

score.c

Script to automatically test your implementation's speed. Shows how the scoring script works and what you should optimize for. This will be run on an isolated core (isolcpus=1).
After implementing engine.c, call:
$ make all
$ taskset 2 ./score
Try to minimize both the mean and the standard deviation of the latency.

score_feed.h

Simulated order and cancel data feed for use with score.c. This is an example of what the official scoring dataset will look like. Orders with price = 0 correspond to cancels with orderid = size.

Scoring

Submit *only* your engine.c file by emailing it to maxdama at berkeley.edu with the subject line: QuantCup1:<Displayed Name>. I will verify <Displayed Name> with the email address you registered. Your registration information must be accurate to be scored. You may submit additional code but not submit modified versions of any of the provided code except for engine.c. For example one may submit engine.c and double_link_list.c since it does not require modification of Makefile.

Minimum Requirements to be Considered:
1) Implement all the matching engine logic
2) Compile without errors on Ubuntu 10.10 and run using dedicated core: isolcpus=1 and cpu affinity
3) Pass all the logic tests in the autotester script
4) Include comment explaining overall architecture and main optimizations

Score Formula = mean(latency) + sd(latency), as reported by ./score.
./score will use a different score_feed.h than the one provided to discourage over-optimization.
The entry with the lowest score wins.

Disqualification:
You may be disqualified, without the judge providing justification nor entertaining appeals, for any perceived attempt to cheat or otherwise game the contest.

Detailed information about judging platform

Suggestions

Data Structure:
How to Build a Fast Limit Order Book (copy) from WK Selph's HowToHFT Blog
Cache Optimization:
Building a HPC Financial Exchange Video

Definitions

Price-Time: At any time there are multiple buyers waiting for someone to come sell shares at an acceptable price. Price-time priority means that the buyers will get filled in order of who offers the best price, and if the prices tie, then by who has been waiting the longest. The same prioritization applies to the sellers who submitted orders that did not fill immediately. Price-time is how the NYSE, NASDAQ, CME, and other major exchanges work.

Matching Engine: the computer program at the heart of an exchange which (1) stores all the outstanding limit orders (this data structure is called an "order book" or just "book"), (2) accepts new orders, (3) determines if new orders cross with any orders already on the book, (4) notifies both sides that their orders have been executed if it does cross, (5) queues orders in the book if they do not cross, (6) accepts order cancellation requests.

Bid: buy limit order
Ask: sell limit order

Cross: a bid and an offer having mutually acceptable prices. A cross results in a trade occurring and shares being transferred. For example a bid for 100 shares at $100 on the book and a newly arriving ask for 50 shares at $100 dollars cross at the price of $100, resulting in the trade of 50 shares for $100. The same trade would occur if the newly arriving ask has a price of $99, or even less.

Execution: short for the execution report which is sent to a trader notifying them that a trade has occurred, how many shares have been transacted, and at what price. Executions are always sent in pairs, one to the buyer and one to the seller.

Latency: the time a computer process takes from start to finish. Can be broken down into matching engine, connection, network stack, data processing, and decision logic latency. In this challenge you will be optimizing matching engine latency only.

Jitter: roughly the standard deviation of latency. High jitter is bad because trades will nondeterministically be delayed.

FAQ

Q: I assume that writing code in multiple files or some other language, compiling them, and sticking the source and a giant assembly dump in engine.c wouldn't be appreciated?
A: Correct. Code must be portably written in pure C.

Q: limits.h has the following: #define MAX_LIVE_ORDERS 65536. MAX_TRADERS and MAX_SYMBOLS must therefore be <= 65536... what would
be a realistic upper limit for these?
A: MAX_TRADERS = 50. MAX_SYMBOLS = 1.