COL106: Data Structures, I Semester 2016-17

Assignment 1

A multithreaded stock exchange

August 20, 2016

The main goal of this assignment is that we get comfortable with JAVA. You will get a chance to use class hierarchy, File I/O, Exception handling, Thread programming, Inter-thread synchronization etc. Please read the entire assignment carefully, many times over.

In this assignment we will create a innovative "no-cost" stock exchange, which receives buy and sell orders and pockets a profit while matching them. The main idea is to receive time-bound requests, which are matched with other requests so as to maximize the "spread" between matched requests, where spread is defined as the difference between the price the seller wants to offer a stock at and the price a buyer is willing to pay. The exchange keeps the spread.

For example, if A is ready to buy 1 unit of stock X at price p, while B is prepared to sell X at q, $q \leq p$, both orders can be satisfied while the exchange earns Rs. $p - q$.

The requests are given to you in an input file. The log of transactions is your output. The file formats are specified below (%x is the format, the string in parenthesis is a pointer to its explanation for it).

%d(T0) %s(Name) %d(Texp) %s (Type) %d(Qty) %s(Stock) %d(Price) %b(Partial)

The input file contains number of requests, one per line. %d is an integer, %b is a boolean (taking values T and F) and %s is a string. The field names are explained below: T0: The time at which the order is placed. The time is in seconds from the beginning of the simulation. Assume orders are sorted by T0 in the file.
Name: Name of the person placing the order.
Texp: The number of seconds that the order remains valid.
Type: Can be buy or sell (case insensitive).
Qty: The number of stocks to sell or purchase.
Stock: The name of the stock for sell or purchase.
Price: The per unit price bid.
Partial: Is partial order acceptable. If the value is F, the entire Qty stocks must be sold or bought. Otherwise, partial order satisfaction is acceptable. For implementation purposes, a non-partial order must be satisfied in a single transaction. (The order it matches may or may not be partial, but if it is not: both must fully satisfy each other.)

Your program must handle the errors in input file format as exception. These must be caught and reported. Your program will consist of three threads: Order Generation Thread, Exchange thread and Cleanup thread.

The responsibility of the Order thread is to read the order file and push the order into a queue of orders at the designated time. If it reads an order after it has expired, it discards that order and does not insert it into the queue.

The responsibility of the Exchange thread is to read the order from the queue and attempt to execute it if it has not expired. It maintains two lists of orders, one for all the buy orders and one for all the sell orders. A new sell order should be matched with one of the buy orders (both should be un-expired) and vice versa. The thread attempts to maximize its profit – so it picks the order(s) that maximizes the the number of stocks transacted times the spread.

Note that maximising this objective function, number of stocks multiplied by the spread, is a computationally hard problem. So, you may use the following algorithm which gives an approximate solution that is easy to compute:

- Suppose n units of a stock X are being offered at price s.
- Set $r = n$ (r denotes the number of remaining units).
- While $r > 0$ and there are unexpired buy requests for X left
  - Pick the buy order b that is asking for at most r units X such that the price it offerers, $p_b > s$ and is the maximum amongst all buy orders that are asking at most r units of X. If a buy order is asking for more than r units but we are allowed to partially fulfil it then all r units are allocated to that order.

It is possible that this algorithm only partially fulfils an order that can possibly be completely fulfilled but we will ignore this problem and treat a sell order that is not able to allocate all its units using this algorithm as a partially fulfilled order (and hence clear it only if we are allowed to satisfy it partially).

If an incoming order is fully satisfied, it is not added to any list. An order (or more) from the list may get fully satisfied by an incoming order – the exchange thread does not remove it from the list, but relies in the clean-up thread to clean it up. The exchange is also greedy, and after any search through its list of pending order, if an order (or more) can be satisfied at this instant, it will execute it rather than wait hoping for a more profitable order to show up later.

The clean-up thread simply removes any expired or satisfied orders from the two lists kept by the exchange thread.

The threads produce outputs in files called, respectively, order.out, exchange.out and cleanup.out (all in the same directory). They simply log their activities. The formats for the three files are as follows. Order thread simply logs the details of the order it places on the queue:

%d(Time) %d(T0) %s(Name) %d(Texp) %s(Type) %d(Qty) %s(Stock) %d(Price) %b(Partial)

Time: Time when this order was enqueued.

In addition, the exception handler in the Order thread tags a line in its log file for each exception it encounters, naming the exception. This prepends the keyword EXCEPTION followed by the rest of the line. It must also discard such erroneous input lines and not generate any transaction for exchange. The Exchange thread logs its order de-queueing activities and transaction activities. When both partial transaction and queue activities happen, the transaction is logged first. When multiple orders are satisfied together, the order that came in first is logged first.

%c(Log) %d(Time) %d(Status) %d(T0) %s(Name) %d(Texp) %s (Type) %d(Qty) %s(Stock) %d(Price) %b(Partial)

Log: It is a character, P if the log is for a request being added to its purchase list, S if it's added to the sale list, and T if it is a transaction.
Status: The number of units transacted.
The last line in Exchange thread log is the total profit the exchange made.

Time is when the order's processing was completed. Note that the process should not be started unless the current time (in seconds) is strictly less than the deadline for participating orders.

Cleanup thread logs the order it deletes from the lists. The format is the same as Order thread's format. Time is the time when the order's removal was initiated.

The simulation ends when the Order thread has no more entries in its file and the cleanup thread has cleaned up both Buy and Sell lists.

The tar-ball of supporting files, "assignment1.tar.gz", contains four java files:

- checker.java: This is the main checker file. It will call the stub functions that you write. DO NOT make any modifications to this file.
- Stock.java: This file should contain the implementation of I/O. You can add extra methods to this file.
- Exchange.java: This file should focus on order matching.
- Test.java: This file should contain the implementation of Threads.

The tar-ball "assignment1.tar.gz" also contains four sample inputs that you can use to test your code on.

Grading: This assignment is designed to test your understanding of multi-threading. If the assignment is implemented without threads then the maximum marks obtained will be 40. A parallel implementation is necessary to be eligible for full marks.

How to run the assignment:
```
$ ... Download assignment1 tar-ball ...
$ tar -xvf assignment1.tar.gz
$ cd assignment1 ... Make suitable changes to Exchange.java, stock.java, and test.java ...
$ make
```
If the make command is not working, replace it with following commands:
```
$ javac Stock.java
$ javac Exchange.java
$ javac Test.java
$ javac checker.java
$ java checker
```