Class: CSCI144
Name: Ahmed Almansor
Date: 11/28/2016
Prof. Dr. Ming Li,

Automated Stock Trading Project Report

This project uses multithreads to simulate trading transactions in a stock market. First, a stock market will be created using random stock names and then assign random prices to the stocks available in the market created. The Prices rang are randomly generated between 5 and 50 dollars. I used a separate threads when assigning the prices to the stocks, so I can change the prices whenever I want with out generating new stocks so the user does not get confused by new stocks every time he/she enters the market, also to help hold the original prices of the stocks, therefore, selling decisions can easily made by the user. The prices will be changing accordingly to the 10000 transactions.

Then a user is randomly chosen from a list of users to enter the market. He/she will be assigned a random balance as a budget, which can be used as funding. This will implement the first bonus part, the user has limited funds, and this budget will be to control the buying decision. The user will not be able to buy if he/she cannot afford it.

Moreover, I used the given variables X, Y, and Z to define the user's decisions. In this part, I implemented the second bonus part by checking the balance of the user every time a user wants to buy. My implementation was that the user cannot buy if he/she can't afford it. X and Y are also controlling the rate of change of the stocks prices.

X and Y were used as selling decisions. However, Z was used for the buying decision. Z's algorithm is based on the following formula:

```
Z = (1 - (s.price*q)/balance);
```

In explaining this formula: I used the current price of the randomly generated stock and the quantity available for the same stock to check if the user's balance is enough to fund the transaction, buying available stock. The value yields from the calculation is decimal, however, I needed the decision to be percentage of the users budget. Therefore, I subtracted 1, so it yields percentage of the user's balance available. After that, I controlled the buying decision by a condition statement to make buying decision wiser, so the user does not buy if he/she started to loose money from buying all stocks that he/she can afford. If the user can afford the buying that does not mean it is a good decision. The formula I used was as follows:

```
if(balance/originalBalance >= 1.25)
      Z *= 0.5;
else {
      Z *= balance/originalBalance;
}
```

As shown in the above code, if the guy is making money then buy more, but if not don't just buy and waste your money! I then generated a random number between 40 and 200 to make it even wiser for the buying decision. I chose the range from 40 to 200 so it can be less chance for the number to be less than 100 everytime. Then, I compare it with Z%. If the random number is less than the Z% then the buying decision returns true. The formula worked excellent and the user was making a lot of money after I added this part. Furthermore, X was set as the alarm for the user to sell whenever the price of the stock is being elevated so the user can make money. On the other hand, Y is the flag that will make the selling decision whenever the price of the stock is dropped. The algorithm used in both decisions is according to these two formulas:

X= ownedStocks[i].price*(1+X) <= s.price
Y= ownedStocks[i].price*(1-Y) >= s.price

X will be random values that fall between 0.15 and 0.25. Let's assume that X's random number is 0.12, this value will be added to 1 because we want to know how much the original purchased price is lower than the current price. After that this values is multiplied by the purchased price. In this case if the purchased price is $12 it will be multiplied by 1.12, which is slightly increase of the original price, then this value will be compared with the current price of the same stock. The condition statement will return true whenever current price is higher than the purchased price by the randomly generated value of X. On the other hand, Y will be a random number as well but it fall between 0.6 to 0.9, safe decision so he/she sells if the price start to drop a lot! Therefore it will return true when ever the current price is dropped a certain percentage based on the random number generated by the random function. Let's say Y is 0.7 it will be subtracted by 1 it will be 0.3 that means when the price is dropped by 3% sell the stock.
In all cases the selling and buying will recalculate the balance of the user and add or subtract money.

Summary & Conclusion:

This project is a multi-threading project that simulates "automated stock trading" where it keeps generating buying/selling transactions threads alternatively. A stock is randomly selected by user from a list of stocks created by a market function that will generate 300 random stocks and a thread that assign random prices between $5 and $50. A buy decision is made with a probability of Z%. On the other hand, current prices of the purchased stocks is randomly checked and the selling decision is based on X and Y. X will sell if price has increased by X% and will also sell if the price is dropped by Y%. The user in this project will have limited fund randomly assigned between the values $1000 to $5000.Moreover, this project contain a server thread which keeps track of the balance and the stock information, such as number of shares owned and its original prices. The following format was used to display the buying transaction, the stock bought, the price and the quantity, when the buying thread is created and excited.

```
ACTION STOCK_NAME NUM_SHARES PRICE_PER_SHARE
```

This project has also a selling thread that will sell purchased stocks. The user's balance is calculated as, revenue from sale - cost from purchase. The server thread will print out the balance and overall yielding on the screen periodically every 5 seconds. The "chrono" library was used to accomplish the waiting time. In every thread I used the following lines to accomplish the waiting:

```
	chrono::seconds waitTime(10);
    this_thread::sleep_for(waitTime);
```

Each buying/selling thread will wait for 2 seconds and then simulate the transaction. This also helped maintain many concurrent threads. Therefore the total wait time is 9 seconds. The wait time is necessary so the threads don't overlap each other and therefore, run efficiently. Locks as well, help in maintaining the complexity of multi threading programs. Because it allows one task at a time.