Due before:	Deadline posted on canvas

CSE-381: Systems 2

Homework #5

Maximum Points: 40

Submission Instructions

This homework assignment must be turned-in electronically via Canvas CODE plug-in. Ensure your C++ source code is named `MUID_homework5.cpp`, where `MUID` is your Miami University Unique ID (e.g., `raodm_homework5.cpp`). Ensure your program compiles without any warnings or style violations. Ensure you thoroughly test operations of your program as indicated. Once you have tested your implementation, upload the following onto Canvas:

1. The 1 C++ source file developed for this homework.

General Note: Upload each file associated with homework individually to Canvas. Do not upload archive file formats such as zip/tar/gz/7zip/rar etc.

Objective

The objective of this homework is to develop a custom "stock market" web-server to:

- Understand the use of multiple detached threads.
- Understand the use of sleep-wake up approach using `std::unique_lock` and `std::condition_variable`.
- Understand the use of synchronization constructs with object-oriented design
- Continue to gain experience with HTTP protocol and working with sockets.
- Demonstrate expertise with straightforward string processing.
- Continue to gain familiarity with automated functional testing.

Grading Rubric:

The program submitted for this homework **must pass necessary base case test(s) in order to qualify for earning any score at all**. Programs that do not meet base case requirements will be assigned zero score!
Program that do not compile, have a method longer than 25 lines, or just some skeleton code will be assigned zero score.

- **5 points**: Reserved overall structure, organization, **conciseness**, documentation, variable-names, etc. in the next page. **If your methods are not concise points will be deducted**.
- **-1 Points**: for each warning generated by the compiler (warnings are most likely sources of errors in C++ programs)

NOTE: Violating CSE programming style guidelines is a compiler error! Your program should not have any style violations.

Due before:	Deadline posted on canvas

Starter code

A short starter code is supplied in `homework5.cpp` to streamline testing. You are also supplied with a `Stock.h` file to encapsulate information pertaining to a stock. You may reuse code from previous homework and exercises as you see fit.

For testing, a custom client is supplied in `stock_client.cpp` along with input text files. **You should not be modifying the `stock_client.cpp` or the supplied input text files.** You are expected to separately compile and run this program for testing your custom web-server as shown further below.

Despite all the testing, it still takes a human to review a program to decide if a program is correctly multithreaded. Hence, the even if your program passes all the tests in the CODE plug-in it could still be incorrectly multithreaded. Consequently, double-check your solution to ensure you are correctly multithreading your program in order to earn full points for this homework.

Responses

The responses from your server should be in the following standard HTTP format, with each HTTP header lines only terminated with a `"\r\n"`. The message at the end of the headers does not have any newline characters.

```
HTTP/1.1 200 OK
Server: StockServer
Content-Length: 36
Connection: Close
Content-Type: text/plain

Stock msft created with balance = 10
```

Note: The `Content-Length` corresponds to the length of the message in the HTTP response. The message should not have any newline characters in it.

Requirements

In this homework, you are expected to develop a simple multithreaded stock market web-server that operates using standard HTTP requests and responses. The stock market consists of a collection of stocks stored as an `unordered_map`, with the stock's name as the key. You are supplied with a `Stock.h` file that contains the definition for a `Stock` class. Each stock is identified by a unique name (a `std::string`). Each stock has a `balance` field that indicates number of available stocks.

Your stock market web-server is expected operate as follows:

1. The server should be developed using the supplied `runServer` (see starter code) as the top-level method. This method should loop forever and accept connections from web-clients.
2. For each connection, it must use a separate **detached thread** for processing the request from the user. Each request will be a standard HTTP GET request in the form (with each line terminated with a `"\r\n"`):

```
GET /TransactionInfo HTTP/1.1
Host: localhost:4000
Connection: close
```

Where, `TransactionInfo` is a HTTP-query string in the form: "`trans=`**`cmd`**`&stock=StockName&amount=amt`". The **`cmd`** indicates type of operation to perform. The stock and amount information are optional. The expected operation and responses (or outputs) for each command is shown below.

Note: The commands below assume that stock name is `0x01`. However, the name can be different/vary. **So do not hardcode stock name to `0x01`**.

TransactionInfo (all on 1 line)	Description of required operation	Expected msg from server in HTTP response
`trans=create&stock=0x01&amount=10`	Add stock (to `unordered_map`) with `balance`=amount if stock with same name does not exist. Note: Stock creation requests will be performed in single-threaded mode only by `stock_client.cpp`.	`Stock 0x01 created with balance = 10` *or* `Stock 0x01 already exists`
`trans=buy&stock=0x01&amount=5`	Subtract specified `amount` from stock's `balance`, if stock exists. Optional feature: If sufficient balance is unavailable, `wait` until stocks are sold.	`Stock 0x01's balance updated` *or* `Stock not found`
`trans=sell&stock=0x01&amount=20`	Add specified `amount` to stock's `balance`, if stock exists.	`Stock 0x01's balance updated` *or* `Stock not found`
`trans=status&stock=0x01`	Return stock's `balance`, if stock exists.	`Balance for stock 0x01 = 24` `Stock not found`

Note: The message "`Stock not found`" is returned if the specified stock name does not exist (in the `unordered_map stockMap`). The response is always in the format shown earlier but the specific message in the response will vary as shown above.

Multithreading Notes

- The web-server must use 1 detached thread per client connection
- Don't forget to read (and ignore) request headers from the client. Otherwise your server will not operate correctly with the web-browser or any standard web-clients.
- Note that, stock creation requests will be performed in single-threaded mode only by `stock_client.cpp`. However, rest of the operations will be performed in multithreaded mode.
- Ensure you appropriately lock & unlock each Stock prior to operating on it to avoid race conditions.
- Do not perform I/O in the critical section. Keep critical sections as short as possible.
- All changes to a stock must be completed before sending response to client.

Due before:	Deadline posted on canvas

Basic Testing

Run your web-browser, in the debugger to help troubleshoot issues. Next, you can test your web-server using the following URLs, after changing the **port number**:

- `http://os1.csi.miamioh.edu:12345/trans=create&stock=msft&amount=20`
- `http://os1.csi.miamioh.edu:12345/trans=buy&stock=msft&amount=10`
- `http://os1.csi.miamioh.edu:12345/trans=sell&stock=msft&amount=5`
- `http://os1.csi.miamioh.edu:12345/trans=status&stock=msft`

Functional testing

A custom multithreaded test client is supplied along with this homework for testing your server. You will need to compile the tester program from a terminal once as shown below:

```
$ g++ -g -Wall -std=c++14 stock_client.cpp -o stock_client -lboost_system -lpthread
```

Base case [10 points] -- Will be strictly enforced

The base case tests require the server to operate correctly in single threaded mode. The base case is relatively simple string processing. To add new stock entries you can use the following simple approach:

```
std::string stock = "msft";
sm::stockMap[stock].name    = stock;
sm::stockMap[stock].balance = 10;
```

The base case testing should be conducted as shown below, assuming your server is listening on port 6000.

```
$ ./stock_client base_case_req.txt 6000
```

Note: On correct operation, the client generates the following output:

```
Finished block #0 testing phase.
Testing completed.
```

Optional Feature: Simple multithreading case [10 points]

Once the base case is operating correctly, multithreading is relatively straightforward using detached threads. See prior exercises or lecture videos for working with detached threads. Ensure you lock/unlock the mutex with a given stock to maximize concurrency without race conditions. **In this feature, you are guaranteed that a stock will always have sufficient balance when buying a stock**. The multithreading testing should be conducted as shown below, assuming your server is listening on port 6000.

```
$ ./stock_client mt_test_req.txt 6000
```

Note: On correct operation, the client will generate the following output:

```
Finished block #0 testing phase.
Finished block #1 testing phase.
Finished block #2 testing phase.
Finished block #3 testing phase.
Finished block #4 testing phase.
Finished block #4 testing phase.
Testing completed.
```

Optional Feature: Multithreading + wait, when insufficient balance [7 points]

Unlike in the previous case, this feature expects you to handle situations where a stock's balance is insufficient – *i.e.*, the client wants to buy 100 stocks, but the `balance` is only 50. If this situation occurs, the thread must use a sleep-wake up solution (using the condition variable associated with the stock) to wait until sufficient `balance` is available to fulfil the `buy` request. This optional feature can be tested as shown below, assuming your server is listening on port 6000.

```
$ ./stock_client mt_wait_test_req.txt 6000
```

Note: On correct operation, the client will generate the following output:

```
Finished block #0 testing phase.
Finished block #1 testing phase.
Finished block #2 testing phase.
Finished block #3 testing phase.
Finished block #4 testing phase.
Finished block #4 testing phase.
Testing completed.
```

Optional Feature: Limit maximum number of detached threads [8 points]

Points will be assigned for this feature only if the above 3 features are operating correctly! In the previous multithreading features the maximum number of detached threads were not limited (to keep the problem simple). Unlimited threading leaves the server highly susceptible to Denial-of-Service (DOS) attacks, which is a big cybersecurity issue. Hence, in this feature you are required to limit the maximum number of detached threads to be fewer than the `maxThreads` parameter to the `runServer` method. Here are a couple of tips to implement this feature:

1. Add an atomic variable (within the `sm` namespace) to count number of threads. Increment and decrement this variable appropriately
2. Add a condition variable (within the `sm` namespace). In `runServer` method wait until the number of detached threads is fewer than `maxThreads`.

This optional feature can be tested as follows. First, run your server in `NetBeans` with command-line arguments `6000 3` – *i.e.*,

```
$ ./homework5 6000 3
```

Next, from a separate terminal run the `stock_client` as:

```
$ ./stock_client mt_wait_test_req.txt 6000
```

Note: On correct operation, the client will generate the following output:

```
Finished block #0 testing phase.
Finished block #1 testing phase.
Finished block #2 testing phase.
Finished block #3 testing phase.
Finished block #4 testing phase.
Expected thread count matched (2)
Finished block #5 testing phase.
```

```
Expected thread count matched (2)
Finished block #6 testing phase.
Expected thread count matched (1)
Finished block #7 testing phase.
Finished block #8 testing phase.
Finished block #9 testing phase.
Expected thread count matched (4)
Finished block #10 testing phase.
Expected thread count matched (7)
Testing completed.
```

Submit to Canvas

This homework assignment must be turned-in electronically via Canvas. Ensure your C++ source files are named appropriately. Ensure your program compiles (without any warnings or style errors) successfully. Ensure you have tested operations of your program as indicated. Once you have tested your implementation, upload the following onto Canvas:

- The 1 C++ source file named with the convention you modified for this homework. Ensure your source file is named with the convention `MUID_homework5.cpp`, where `MUID` is your Miami University Unique ID (e.g., `raodm_homework5.cpp`).

Upload all the necessary C++ source files to onto Canvas independently. Do not submit zip/7zip/tar/gzip files. Upload each file independently.