CSE-381: Systems 2

Homework #7

Due: Wednesday Oct 21 2020 before 11:59 PM

Email-based help Cutoff: 5:00 PM on Tue, Oct 20 2020

Maximum Points: 25

Submission Instructions

This homework assignment must be turned-in electronically via Canvas. Ensure your C++ source code is named `MUID_hw7.cpp`, where `MUID` is your Miami University Unique ID. 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.

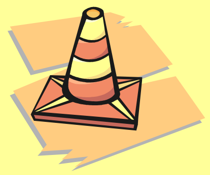
General Note: Upload each file associated with homework individually to Canvas. Do not upload archive file formats such as zip/tar/gz/7zip/rar etc.

Objective

The objective of this homework is to:

- Understand the use of multiple threads
- Develop a custom multithreaded web server.
- Understand the use of critical sections created using a `std::mutex` and `std::lock_guard`.

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.

- **Base case [10 points]:** No multithreading needed
- **Extra function [10 points]:** Multithreading and using `std::lock_guard`
- **Documentation & Conciseness [5 points]:** Overall structure, organization, **conciseness**, documentation, variable-names, etc. **If your methods are not concise points will be deducted**.

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

Starter code

A short starter code is supplied in `homework7.cpp` to streamline testing. You may reuse code from previous homework and labs as you see fit.

However, for testing a custom multithreaded test client is supplied in `bank_client.cpp` along with input files. **You should not be modifying the supplied input text files. You should not add `bank_client.cpp` to your NetBeans project**. You are expected to separately compile and run the supplied `bank_client.cpp` 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\r\n
Server: BankServer\r\n
Content-Length: 19\r\n
Connection: Close\r\n
Content-Type: text/plain\r\n
\r\n
Account 0x01 created
```

Note: The `Content-Length` corresponds to the length of the message (e.g., – "`Account 0x01 created`" has 19 characters) in the HTTP response. This message should not have any newline characters in it.

Requirements

In this homework, you are expected to develop a simple multithreaded banking application that operates using standard HTTP requests and responses. The bank consists of a collection of accounts. Each account is identified by a unique account ID (a `std::string`) and its associated balance (`double`). **This information must be stored as a `std::unordered_map`, with the account ID serving as the key**.

The bank web-server is expected operate as follows:

1. The bank server should be developed by suitably implemented the supplied `runServer` as the top-level method. Similar to other web-servers, this method should loop forever and accept connections from web-clients.
2. For each connection, it should 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 standard query string in the form: `"trans=cmd&acct=AccountID&amount=amt"`. The **`cmd`** indicates type of operation to perform. The account and amount information are optional. The expected operation for and output for each command is shown below.

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

TransactionInfo (all in 1 line)	Description of required operation	Expected msg from server in HTTP response
`trans=reset`	Clear out all accounts	`All accounts reset`
`trans=create&acct=0x01`	Create account only if account does not exist.	`Account 0x01 created` *or* `Account 0x01 already exists`
`trans=credit&acct=0x01&amount=20.25`	Add specified amount to account, if account exists	`Account balance updated` *or* `Account not found`
`trans=debit&acct=0x01&amount=20.25`	Subtract specified amount to account, if account exists	`Account balance updated` *or* `Account not found`
`trans=status&acct=0x01`	Return account balance with exactly 2 decimal places via: `ss << std::fixed << std::precision(2) << 20.25;`	`Account 0x01: $0.00` / `Account not found`
Note: The message "`Account not found`" is returned if the specified account ID does not exist. The response is always in the format shown earlier but the specific message in the response will vary as shown above.		

Notes & Suggestions

- Use the Bank class in starter code to encapsulate all of the data. You can pass a reference to a shared bank object to various methods in your program.
- Processing requests is straightforward. **But don't forget to read request headers from the client**. Otherwise your server will not operate correctly with the web-browser or any standard web-clients.
- It may be easier to use a `std::istringstream` to extract information after replacing `'&'` and `'='` with a space. Of course, you already did something this in Homework #2.
- **The web-server must use 1 detached thread per client connection.**
- You may suitably use a `std::mutex` variable inside the `Bank` class (see starter code) to avoid race conditions.
- Ensure checking for bank account and operating on accounts is performed within the same critical section. Otherwise you will experience a race condition.
- Do not perform I/O in the critical section. Keep critical sections as short as possible.
- All changes to the bank account must be completed before sending response to client.

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=reset`
- `http://os1.csi.miamioh.edu:12345/trans=create&acct=test1`
- `http://os1.csi.miamioh.edu:12345/trans=credit&acct=test1&amount=10.50`
- `http://os1.csi.miamioh.edu:12345/trans=debit&acct=test1&amount=1.25`
- `http://os1.csi.miamioh.edu:12345/trans=status&acct=test1`

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 bank_client.cpp -o bank_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 essentially just simple string processing. The base case testing should be conducted as shown below, assuming your server is listening on port 6000.

```
$ ./bank_client base_case_req.txt 6000
```

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

```
Finished block #0 testing phase.
Finished block #1 testing phase.
Testing completed.
```

Multithreading case [10 points]

Once the base case is operating correctly, multithreading is relatively straightforward using detached threads. However, ensure you use critical sections (using `std::lock_guard` will make your life easier) when performing various operations. The multithreading testing should be conducted as shown below, assuming your server is listening on port 6000.

```
$ ./bank_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.
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 you developed for this homework

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