Bank Service | Sabin Raj Tiwari | CMSC 621 Project 1

Make

If there are already compiled files in the directory, first run:

```
make clean
```

Then, to build the program, run:

```
make compile
```

This will create two excecutable files: `server` and `client`

Run

To run the program, first run the server program with the port and the records filename, like so:

```
server 3000 ./Records.txt
```

Once the server is running and waiting for requests, run the client by passing three arguments (address, port, timestep, and filename):

```
client localhost 3000 0.1 ./Transactions.txt
```

The logs from the programs will be saved in the logs directory. Each client process will have its own program log file and an associated transactions log file. The server has one log file. There are local timestamps in the log files.

Design

The overall design of the code was based on basic client-server communication functionality. The client reads in the transaction input file and writes each of those to the socket. While the server waits for a client connection to be made and the transactions to be sent over. Once the connection is made by the client, the server reads in the transaction data first, tries to perform the transaction, and sends a response if the transaction was successful or not. The client waits for a response after sending the transaction to see if it was successful or not.

There were some tradeoffs that had to be made. For example, I was not able to modularize the

classes as much as possible. There are few duplicate code that could be made into functions and classes to make the program code much cleaner and efficient. I started adding a text based user interface that would allow to create custom queries from the `Client` program when a filename was not passed in the arguments. Because of lack of time and the fact that I was spending a lot of time debugging the UI, I had to remove that feature. Now the `Client` only expects a transactions file with a proper list of transactions like so: `<time> <account> <type> <amount>`. I also wanted to use a lot of C++11 functionality like std::threads and other helpful functions but decided to stick to a lot of C functionality and used C++ only when needed. Making the program more C++ oriented with libraries and more class declarations would have made the code much shorter and efficient.

Threading and Synchronization

The `Server` program has threading capabilities for each of the `Client` request that comes through. Each of the worker threads will check if the account exists that the transactions will be performed on. The checking of the account is handled by threads as well. Before reading the `std::vector<Record>`, there is locking so that there aren't updates being peformed at the same time reads are being performed. If the account exists, `pthread_cond_t` will be used to wait until the `Record.is_locked` value is `0`. Once the value is `0`, the `Record.is_locked` will be set to `1`, the transaction will be performed based on `Transaction.type`, and finally the `Record.is_locked` will be set to `0` as well as the the `pthread_cond_t` will be used to signal the condition. There is also a thread for the `Server` program's internal interest accumulator that keeps running as long as the `Server` is running and performs the interest transaction at set intervals. After each transaction the data is written to the file and the locking and synchronization to the file are handled in the same manner as the `Record` object.

Main Files

`Server: server.cpp`

- The `Server` program first initializes the `std::vector<Record>` using the records input file. If anything fails here (IO error, invalid record input,etc.) the program will exit. Then `Server` intializes the socket based on the information provided as arguments and then binds the socket to the address. The main loop of `Server` listens for a client request, creates a thread to handle a request, and closes the connection once the client sends call to end. Each thread, once spawned, will wait to read data from the client, converts the buffer to `Transaction` object, performs the transaction, and send a response (`Record.balance` if successful, `-1` if account not found, and `-2` if insufficient funds). Those steps keep happenning in a loop until the client sends "finish" as the data. `Server` uses `Logger` to create a log file and write to `std::cout`. The log file will be `logs/_server_log_file.txt`. `Server` program contains the `struct` objects that store the socket and thread information. Those are used to transfer the data to the thread functions. `Server` also uses the `accumulate_interest()` function in a thread to update the records at fixed intervals.
- The `Server` program has the following functions:

```
std::string i_to_s(int value);
/* Get the string value from an int. */

std::string m_to_s(double value);
/* Get the string value for money. */
```

```
Record* get_record_by_id(int id);
/* Checksif the account with the id exists and returns the Record. */

int save_records();
/* Saves the records to the file and returns 1 if success full. */

void load_records(std::string filename);
/* Load all the records from the file in the records vector. */

double perform_transaction(Record* record, Transaction* transaction);
/* Write data for the transaction to the record. Handles locking of the recor
d when multiple write access is attempted. */

void *accumulate_interest(void *args);
/* Accumulates interest in a fixed interval on all the records in the vector.
 */

void *client_request(void *args);
/* Handle the client request and perform the transaction. Send response to th
e client with success or failure. */

void *wait_for_connection(void *args);
/* Wait for connection to come in from the client. */
```

`Client: client.cpp`

- The `Client` program first initializes the socket information using the arguments that are passed. If successful getting the host using the arguments, the `Client` will call the `batch_transactions()` method with the filename of the transactions file. The transactions will be run based on the timestamps associated with them in the file. The connection is maintained until all the transactions are sent and replies are received. `Client` uses `Logger` to create two log files and write to `std::cout`. The log files will be `logs/[pid]_client_log_file.txt` for the program log and `logs/[pid]_transactions_log_file.txt` for the transaction log.
- The `Client` program has the following functions:

```
std::string i_to_s(int value);
/* Get the string value from an int. */

std::string m_to_s(double value);
/* Get the string value for money. */

int connect_to_server(struct sockaddr_in server_address);
/* Create a connection to the server and return the socket file descriptor. *
/

float batch_transactions(struct sockaddr_in server_address, std::string f
ilename);
/* Read the transactions from a file and perform multiple transactions. */
```

Helper Classes

There are 3 helper classes that help `Client` and `Server` perform their functions.

`Record`: `record.h` and `record.cpp`

- The `Record` class is the conversion of the contents of the records file. It contains the following attributes: `int account`, `std::string name`, `double balance`, `int is_locked`, `pthread_mutex_t lock`, and `pthread_cond_t cond`. The first three attributes are data from the file. `is_locked` is used as a condition variable to prevent multiple write access at the same time. `lock` and`cond` use the `is_locked` attribute to make threads wait for the `Record` to be accessible.

`Transaction`: `transaction.h` and `transaction.cpp`

- The `Transaction` class is the conversion of the contents of the transactions file. It contains the attributes: `int time`, `int account`, `std::string type`, and `double amount`. All the attributes are the data from the file.
- The `Transaction` class has the following functions:

```
int is_valid();
/* Returns 1 if the transaction data can be parsed successfully and has valid
 data else returns 0. */

void reset(std::string transaction);
/* Resets the transaction data using a string with transaction data. */
```

`Logger`: `logger.h` and `logger.cpp`

- The `Logger` class is a utility class that can be used to create log files. It contains one attribute `std::ofstream log_file` that is setup using a filename passed in the constructor. `Logger` will clear any file that already exists when it is initialized.
- The `Logger` class has the following functions:

```
void log(std::string message);
/* Logs a message to the cout and the log file. Includes a \n at the end. */

void close();
/* Closes the file stream. */
```
