

**UNIVERSITY OF MARYLAND BALTIMORE COUNTY**

**DEPARTMENT OF COMPUTER SCIENCE**

**ADVANCED OPERATING SYSTEMS**

**CMSC 621**

**PROJECT 1**

**SUBMITTED BY:**

**SUSHMITHA MANJUNATHA**

**ABSTRACT**

In this project, I have tried to implement a Centralized Multi-user Concurrent bank Account Manager. Mainly the system has two main components: The Bank server and The Clients. The Bank server program services online requests for account manipulations and maintains all customer records correctly. The Clients are the customers and uses the services of the bank server to update bank accounts. They mainly perform two operations: Withdrawal of an amount from account or Deposit of an amount to an account. The server is so designed to handle multiple requests simultaneously. This is implemented using threads. Shared simultaneous access has been handled using mutex functions. The whole system has been implemented by using C++ and has been tested on linux machines. The system successfully accepts the client requests and performs the requested operation. It also handles multiple requests simultaneously and takes care of the critical issue of shared simultaneous access. Finally it sends an appropriate message to the clients along with the available balance. Also timing measurements have been done and average time to perform each transaction is measured. Finally, a graph is plotted to show the average time when number of clients increase and also when request rate increases.

**SYSTEM DESIGN**

**Description of the functions used:**

**In-built functions used:**

**argc -**  It is the number of arguments passed into the program from the command line, including the name of the program.

**argv[]** - The array of character pointers is the listing of all the arguments provided in the command line.

**stderr** -Standard error is an output stream typically used by programs to output [error messages](https://en.wikipedia.org/wiki/Error_message) or diagnostics. It is a stream independent of standard output and can be redirected separately.

**gethostbyname()** – This function returns a structure of type *hostent* for the given host *name*. Here *name* is either a hostname or an IPv4 address in standard dot notation.

**atoi** - Convert string to integer.

**atof** - Convert string to double.

**sockaddr\_in** -  The basic structure for all syscalls and functions that deal with internet addresses.

**socket(domain,type,protocol)** - The function socket() creates an endpoint for communication and returns a [file descriptor](https://en.wikipedia.org/wiki/File_descriptor) for the socket. socket() takes three arguments: domain, type and protocol.

**AF\_INET**- Indicates network protocol [IPv4](https://en.wikipedia.org/wiki/IPv4) (IPv4-only).

**SOCK\_STREAM** – It is reliable stream-oriented service .

**bind()** - It assigns a socket to an address. When a socket is created using socket(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts. bind() takes three arguments:

sockfd, a descriptor representing the socket to perform the bind on.

my\_addr, a pointer to a sockaddr structure representing the address to bind to.

addrlen, a socklen\_t field specifying the size of the sockaddr structure.

Bind() returns 0 on success and -1 if an error occurs.

**listen()** – After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (SOCK\_STREAM, SOCK\_SEQPACKET). listen() requires two arguments:

sockfd, a valid socket descriptor.

backlog, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

[**ifstream**](http://www.cplusplus.com/ifstream) **-** Stream class to read from files

[**fstream**](http://www.cplusplus.com/fstream) **-** Stream class to both read and write from/to files.

**htons**() - This function converts the unsigned short integer *hostshort* from host byte order to network byte order.

**filename.c\_str()***-* If *filename* is a variable of type *string*, we can obtain the corresponding C-style string by calling the function *filename.c\_str().*

**bzero(buff,n)** - The bzero() function erases the data in the *n* bytes of the memory starting at the location pointed to by buff, by writing zeroes (bytes containing '\0') to that area.

**sprint(char \*str ,const char \*format)** - Composes a string with the same text that would be printed if *format* was used on [printf](http://www.cplusplus.com/printf), but instead of being printed, the content is stored as a *C string* in the buffer pointed by *str*.

**write(int *fd*, const void \**buf*, size\_t *count*) -** It writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

**read(int *fd*, void \**buf*, size\_t *count*) -** attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

**sockfd -** It is the socket descriptor returned by socket(). serv\_addr is pointer to struct sockaddr that contains information on destination IP address and port. addrlen is set to sizeof(struct sockaddr)

**sleep() –** It makes the calling thread sleep until specified seconds have elapsed or a signal arrives which is not ignored.

**accept()** - When an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using the accept() function. The accept() function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

sockfd, the descriptor of the listening socket that has the connection queued.

cliaddr, a pointer to a sockaddr structure to receive the client's address information.

addrlen, a pointer to a socklen\_t location that specifies the size of the client address structure passed to accept(). When accept() returns, this location indicates how many bytes of the structure were actually used.

The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs.

**pthread\_attr\_init(&tattr)-** This function initializes the thread attributes object pointed to by *attr* with default attribute values.

**pthread\_attr\_setdetachstate((&tattr, int detachstate)** – This function sets the detach state attribute of the thread attributes object referred to by t*attr* to the value specified in *detachstate*. The detach state attribute determines whether a thread created using the thread attributes object t*attr* will be created in a joinable or a detached state.

**pthread\_create()** - This function starts a new thread in the calling process. The new thread starts execution by invoking *start\_routine*(); *arg* is passed as the sole argument of *start\_routine*().

**pthread\_mutex\_init(&mutex,NULL)** - This function shall initialize the mutex referenced by *mutex* with attributes specified by*attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

**pthread\_mutex\_lock(&mutex)** - The mutex object referenced by *mutex* is locked by calling *pthread\_mutex\_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

**pthread\_mutex\_unlock(&mutex)** - This function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute.

**User-defined functions used:**

**void deposit(arguments) –**  Takes records structure , transactions structure , buffer containing details of transactions requested and number of accounts as arguments. It performs the requested deposit operation by adding the requested amount to the previous balance available. It also handles multiple requests and synchronization issues. After performing the operation, it sends a success message to the client along with the available balance and exits.

**void withdrawal(arguments)-** This function takes records structure , transactions structure , buffer containing details of transactions requested and number of accounts as arguments. It performs the requested withdrawal operation by first checking if the requested withdrawal amount is available in the account. If insufficient balance is present, it sends an appropriate error message to the requesting client. If sufficient funds are available, it subtracts the withdrawal amount from the previous available balance and send the client an appropriate success message along with the available balance and exits.

**Void GetAccount(arguments)-**This function is used to get the details of account by giving account number as user input. It fetches complete record that belongs to that account and writes it to a file by using ofstream.

**Structures Used:**

**records -**  It stores the records of the accounts in the bank-server. It has attributes like account-number, customer-name and balance.

**transactions -**  It stores the information of transactions requested by the clients. It has attributes like timestamp, account-number, operation requested i.e. deposit or withdrawal, and requested amount.

**How to run the code:**

Server side

1. g++ -o server Bankserver.cpp -lm -lpthread
2. ./server <portnumber> <Records.txt>

Client part:

1. g++ -o newclient2C.cpp
2. ./client <hostname> <portnumber> <timestamp> <Transactions.txt>

Ex: ./client 127.0.0.1 7777 0.4 Transaction.txt

**Server program**

1.Server program takes port number and Records.txt file as command line arguments.

2.Server has 2 structures records and transaction to store the data read from file.

3.Creates a socket and initializes the socket structures then binds to the given port number.

4.Then keeps listening for the clients to connect.

5.For each of the client a thread will be created and condition handler function is called.

6.Executes the operation mentioned by the client transaction record (operations are **deposit**-depositing money to account, **withdraw**-withdrawing money, **Getaccountinfo**-To fetch the account information in a file called Record1.txt by giving account number as in put .)

7.Writes back to the client after the transaction is successful with previous and updated balance amount.

8.Keeps waiting for the next transaction to process.

**Client program**

1.Client program takes Server name, Port number, Timestamp and Transaction.txt as command line arguments.

2.Creates the socket and initializes the socket structures for server and client.

3.connects to the server.

4.Sends the transactions record one by one to server for processing. Transaction is done based on timestamp.

5.Reads the acknowledgement sent back from the server.

**Process Synchronization**:

**Timestep** is used to handle synchronization. Timestamp obtained from command line argument will be compared with each of the timestamp of transaction record given in the file. If timestamp of transaction record is more than or equal to current time those records will be sent for processing by server and records having lesser timestamp will have to wait until its timestamp becomes equal to the current timestamp for getting executed. For making the threads wait I have used Sleep() function which calculates the wait time by this way

sleep((timestamp-initialtimestep)\*sec);

Once the initialtimestep value becomes equal to the timestamp value that thread will be sent for processing by server.

In this way synchronization of threads are handled.

**Functionalities of client program:**

i) Issues withdrawal or deposit requests.

ii) For ease of testing and to make experiments bigger, I have made our clients to issue requests at fixed time intervals.

iii) A client can read an input file for transaction information and perform accordingly.

**Functionalities of server program:**

I)Is able to accept multiple concurrent customer requests (i.e. multi-threaded).

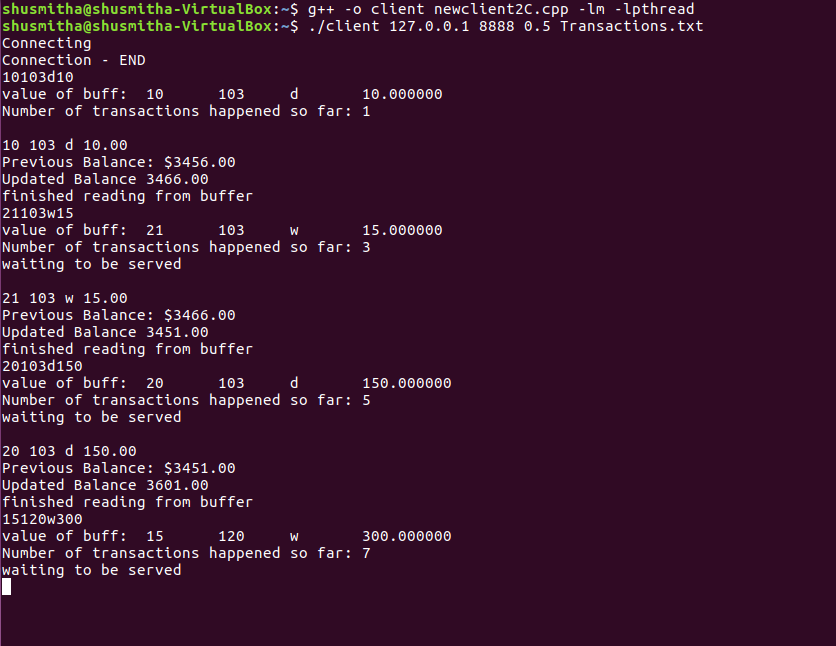
ii) It provides locking/protection for access to an account records during shared access.

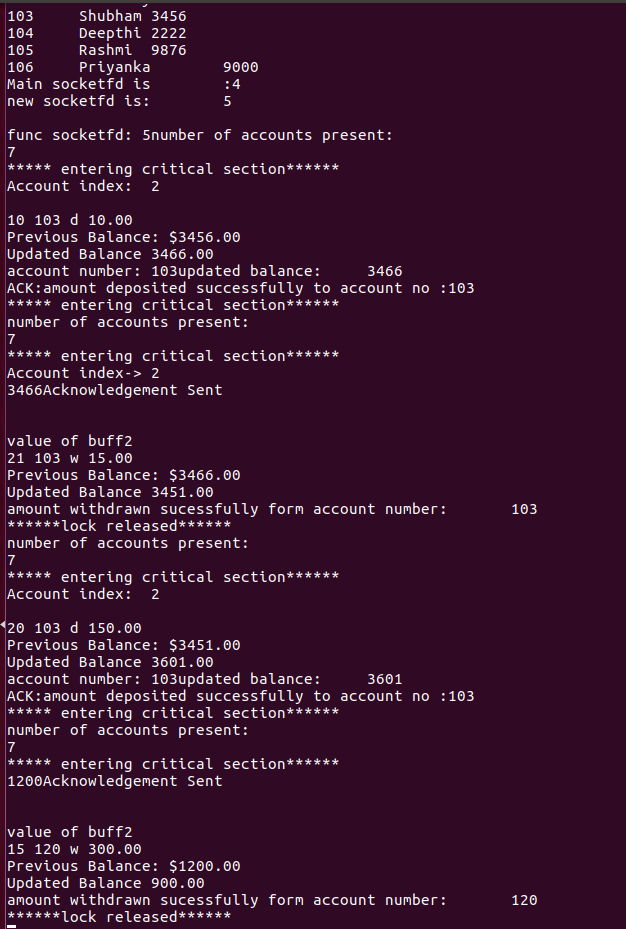
iii) Maintains correctness of records at each record, (i.e., allow withdrawal from an account only if it has sufficient funds.

Possible improvements in the program:

1.Server can be added with more functionalities such as add a new account.

2.For each client authentication can be done at server before serving.

OUTPUT: client



Server output generated