Networks Lab Assignment 2
Date of Submission: 27th February, 2013

Problem 3
Calendar Server

Name: Amogh Tolay
Roll No. 10010110
Dept of CSE
IIT, Guwahati

Setting up the code and executing the code:

Server Side:

1. Unzip the folder named server.
2. Run the Makefile by running the command '**`make`**'
3. One server executable will be created named **`serverAllModes`**.
4. Please ensure that there is a folder name as specified in the FOLDER_NAME macro defined in returnCodes.h. This is the folder calendar by default and it already exists. So no need to do anything.
5. This file takes as command line arguments the values:
    1. Port Number (This can be any value greater that 1024 and less than approximately 65000)
    2. The mode in which the server needs to be run:
        1. Integer 1 corresponds to the iterative mode in which the server runs in an infinite loop waiting for server to connect. This supports just 1 client.
        2. Integer 2 corresponds to the multi-threaded mode in which the server keeps listening on the socket, and then creates a new process the moment a new client comes. It supports multiple simultaneous connections.
        3. Integer 3 corresponds to the mode using **`select()`** function. This supports multiple clients and loops through each client.

    Thus a sample command for running the server in iterative mode would be:
    `./serverAllModes 7000 1`

Client Side:

1. Unzip the folder named client on the client machine.
2. Run the Makefile by running the command **`make`** in this directory.
3. One client executable called client will be created.
4. The command line arguments for client are as follows:
    1. hostname: This denotes the IP address (or name of machine) of the server
    2. username: This denotes the username of the person who wishes to use the calendar server
    3. command: The supported commands are add, get, update, remove, getall
    4. arguments: The argument size is different for different commands.

    An example of various commands are as follows:

```
1. ./client localhost 7000 amogh add 031613 0800 0930 Lab
2. ./client localhost 7000 amogh update 031613 0800 1000 Lab
3. ./client localhost 7000 amogh get 031513
4. ./client localhost 7000 martin add 031713 0900 1130 Shopping
5. ./client localhost 7000 martin update 031713 0900 1230
   Shopping
6. ./client localhost 7000 martin getall
7. ./client localhost 7000 amogh2 remove 031713 0900
8. ./client localhost 7000 amogh get 031513 0800
```

Note: Date is in the format MMDDYY and time is in HHMM format.

Sailient Features:

1. The server supports events which span across multiple days also. (This wasn't required according to problem statement but was implemented anyways)
2. The user details are stored in a text file, and thus the calendar entries are permanent across multiple runs of the server.
3. Server supports 3 modes as asked in the question.
4. There is minimum processing at the client side, so the issue of security is handled (even if the client sends a bad request, it does not crash the server)
5. The port is freed as soon as the server terminates. That is, freeing up of ports is being done. So, the same port can be reused again without waiting for any time.
6. The signal SIGINT (caused when we press Ctrl-C) is caught and appropriate processing takes place. That is, the server gracefully exits when Ctrl-C is pressed.
7. Proper date and time checking of events is done. That is, invalid dates and times will be reported by the server.
8. The code is very modular. File handling (dealing with updates, reading or writing from the file) is done in another file. So the network part of the server and the file handling part of the server is done in 2 files, increasing modularity.
9. Error checking is being done properly.
10. The errors or even success statements are very verbose. The client is informed of each step the server took and where there was an error, if any. This not only helps the user to resolve the error, but also can be used by server administrators to debug the code.
11. User's file is automatically created when the first event is added, and automatically removed when the last event is removed.
12. All expired events of all users are removed when any command is executed on the server.
13. No past event can be entered in the file. Also no event can have startTime < endTime.
14. Getall query is supported for all 3 types (iterative, multi-process and select() based)

Storage of events in file:

Each user has a unique file (named ./calendar/calendar_username)which is created in the calendar folder of the server side folder. The name of the file is the username. The file contains the following details:

```
startTimeStamp endTimeStamp EventName
```

I've converted the date and time (both start and end times) into time_t times. That is, for any date and time, it stores a unique unsigned integer (which is the number of seconds since Epoch time (1st January, 1970, 00:00:00) and is of the type that the time() function of C++ returns.

The time_t form of storing date time is used because then its easier to check events that have past the present time. Its also easy to check errors like endTime < startTime etc.
This also ensures that the event can span multiple days.

For eg. When the user performs this operation:
`/client localhost 7000 amogh add 031613 0800 0930 Lab`
The entry in the file (filename is calendar_amogh) is stored as this:
`1363401000 1363406400 Lab`
where 1363401000 corresponds to 16th March, 2013 0800 hrs and 136346400 corresponds to the same date and time 0930.

Description of Code:
There are basically 5 files and 2 makefiles for each client and server sides.
The files are as follows:
`fileReadWrite.h`: This is the file with all function prototype declarations
`returnCodes.h`: This file is the file which denotes the strings that have to be sent over the server as macros. This is used by both client and server.
`fileReadWrite.cpp`: This is the file that is used for reading and writing of the user's file. All the functions (add, get, update etc.) are ultimately implemented in this file as this is the only file that reads/writes to the file.
`serverAllModes.cpp`: This is the file that handles the connections between server and client and supports three modes of operation.
`client.cpp`: This is the client side file that deals with sending the query, and then displaying the output.

Steps that take place in the execution of any query:

1. The server is set up and listens on the given port for incoming clients.
2. When the client arrives, the connection is accepted. In the case of multiple processes, a new process is created to handle all queries with this client using **fork()** call. If single iterative server is being run, then the server simply executes the query by calling **`execQuery()`** function as declared in **`fileReadWrite.cpp`**. If **`select()`** mode is being run, then the server executes the same function, but with alternating each of the clients that is connected.
3. In the **`select()`** based mode, there is a list of clients that are connected and the server keeps polling and checking if any of the client has anything to write.
4. The query is parsed in the function **`execQuery()`** and parsed using parse() function. It returns all arguments as a vector set of strings. Then, **`execQuery()`** calls the appropriate function to execute the query and returns a string which is the output of this query. This string is then passed on to the server, and subsequently on the client.

5. For the **`getall`** function, the query just sends the number of events of the user to the client. Then, the client keeps checking and iterating and sends the request for next entry in every 2 secs. This query is executed and the nth line is returned.
6. The code has proper inline documentation and comments, so the code is easily readable and understandable.

The file handling code is divided into two sections, one set of helper functions (like valid date checking, parsing etc.) and the other set being the actual functions that are used. These functions call these helper functions repeatedly.

Test Cases:

A set of test cases is attached with the code. These test cases can be run and other testcases can also be tested. They are listed below also. Each of these test cases when run in order from top to bottom should give the outputs as follows:

All these queries are executed in order, if the order is changed, result might be different.

Query should be successful in these cases:

```
./client localhost 7000 amogh add 031513 0900 1100 Test1
./client localhost 7000 amogh add 031513 1300 1400 Meeting1
./client localhost 7000 amogh add 031513 1600 1830 Class
./client localhost 7000 amogh add 031613 0800 0930 Hospital
./client localhost 7000 amogh add 031513 1400 1500 NetworksClass
./client localhost 7000 amogh add 031613 1500 1600 Birthday
./client localhost 7000 amogh update 031613 0800 1000 ENT
./client localhost 7000 amogh get 031513
./client localhost 7000 amogh get 031513 0900
./client localhost 7000 amogh getall
./client localhost 7000 amogh remove 031513 0900
./client localhost 7000 amogh add 031513 0800 0930 Compilers
```

Query should fail in the following case:

```
./client localhost 7000 amogh add 031513 0800 0930 LectureClash
./client localhost 7000 amogh update 031713 0800 0830 Update
./client localhost 7000 amogh add 021713 0800 0900 pastDate
./client localhost 7000 amogh add 031713 0900 0800 durationNegative
./client localhost 7000 amogh remove 031513 0900
./client localhost 7000 amogh update 031513 1600 1830 Class
```

Any other cases can be checked. All required queries are supported by my implementation.