Problem Statement 2

[Online Bill Manager]

In this assignment, you will design an online application that can be used to manage bills. For the purpose of this assignment, bills are to be represented as text files with multiple records (one record per line). The server can serve multiple clients using either fork() or select() methods. Each client can upload a particular bill to the server, and the server can perform certain operations on the bill and return the modified bill to the client on success.

Valid Bill:

A text file will be considered as a valid bill if it strictly adheres to the specified format. There will be a single record per line, and each record is a tab-separated 3-tuple:

`<date> [TAB] <item_name> [TAB] <price>`

Date will only be in DD.MM.YYYY format. You need to ensure that the date is a valid date.

Connection Phase:

The server creates a socket and listens for client connections on a specific port. The client should send a request for connection to the server, and an appropriate reply should be returned based on success or failure.

- ***Successful Connection:*** If successful, the server acknowledges the same to the client, which then can send commands to the server.
- ***Unsuccessful Connection:*** If unsuccessful, an error message is returned to the client.

The server should handle multiple clients at the same time (max. 5 concurrent client connections). If 5 clients are already connected to the server, any new connection request should be denied.

Server Functionalities:

After successful connection, a client can now access different functionalities by sending queries to the server. For any type of operation, the file must be uploaded to the server while specifying the query, and the server should check whether the uploaded text file is a valid bill or not.

- – ***Sorting a bill***

A client can upload a bill and ask the server to sort the bill in only ascending order based on either of the 3 fields: date, item_name (sort lexicographically) and price.

– – ***Merge two sorted bills***

A client can upload two sorted bills and ask the server to merge them (sorted). The server should check if the individual files are sorted or not. If not, return an error message. The field (on which the bills are sorted) will be mentioned by the client.

– – ***Similarities in two bills***

A client can upload two bills (may be unsorted) and ask the server to find **all pairs of similar records** in the two bills. A pair of records is said to be similar if at least one of the three fields (date, item_name, price) are the same.

– – ***Internal functionalities***

Apart from the above three, the server should have an internal functionality to check if a bill is valid or not, and also to check if a bill is sorted along a given field or not.

- **Successful Operation:** For sort and merge sorted, the server should return the modified bill file to the client. For similarity, the server should return all pairs of similar records in a file. This will be of the format

```
<date1> [TAB] <item_name1> [TAB] <price1> [TAB] | [TAB]
<date2> [TAB] <item_name2> [TAB] <price2>
```

- **Unsuccessful Operation:** The server should return an error message to the client based on the type of error encountered.

At server side (VARIATION – Using fork):

- Create a socket, and listen for client connections.
- Once a client connects successfully, use fork() to create a server subprocess dedicated to handling that client
- The client will send queries as described above, the subprocess should process the query, perform the requested operations and return the desired output on success and appropriate error message on failure.
- **The server should print the message (query) it receives from the client verbatim**
- Repeat the last step till the client disconnects, at which point exit this subprocess.

- The main server process (which listens for new connections) should run infinitely.

At server side (VARIATION – Using select):

- Create a socket, and listen for client connections.
- Once a client connects successfully, it can send requests to the server. The server should be waiting for any client request using select()
- The client will send queries as described above, and once the server detects a client request, it goes on to serve that request before again returning to wait for further requests
- **The server should print the message (query) it receives from the client verbatim**
- Repeat the last two steps till infinity

At client side:

- Connect to server
- Take command input from user, and send appropriate query message to the server (pass the command and the contents of the file)
- Retrieve server outputs and display to user
- Repeat the last two steps till the user terminates, and disconnects from the server.

User Commands: To be entered at client-side input (strictly following the given format)

1. `/sort <filename> <field>`**:** Sort the bill stored at filename, using the field axis (field can be either of D/N/P indicating date, item_name and price respectively), and store the result at filename
2. `/merge <filename1> <filename2> <filename3> <field>:` Merge two bills stored at filename1 and filename2, both of which should be sorted along field axis, and store the resulting bill at filename3
3. `/similarity <filename1> <filename2>:` Find out all pairs of similar records in the two bills stored at filename1 and filename2 (may be unsorted). Here, there is nothing to store, the desired outputs should be printed to the terminal.
4. `/exit:` Terminate connection with the server

   All the different file names indicated here refer to client-side files, i.e., these files must be present at the client side computer.

Design Guide:

All implementation details are left to you, given that you follow the requirements mentioned above. However, you can refer to this design guide if you wish to.

1. First, start the server and wait for client connections
2. When a new client is started, it first tries to establish a connection with the server
    a. If the connection is successful, the server creates a subprocess to handle this particular client (if using fork()), and sends a welcome message to the client.
    b. If unsuccessful (for example, if 5 active connections are already present), an appropriate error message should be sent by the server to the client.
3. A client that has connected successfully now waits for user input (user enters commands as mentioned above)
4. The user enters a command, and the client parses it
    a. If the command is invalid, the client should reject it
    b. If the command is valid, the client now needs to construct a message (query) to send to the server
    c. The syntax of this message is left to you, however, you should construct these messages such that the command requested and the required parameters can be understood by the server
    d. For example, for the sort command, the client can read the entire contents of the file and pass it as part of the message
    e. Since the file can contain newlines, you should handle this appropriately when sending and receiving messages
    f. For the other commands, you need to demarcate the contents of the two files
5. Server receives the message constructed by the client
    a. It prints the message verbatim
    b. It parses the requested command and the file contents
    c. Ensure that the parsed contents should exactly reflect the nature of the file, i.e., if you apply any kind of operation to handle newlines, that should be reversed at the server side again
    d. It performs the desired operation on the file contents
    e. It then returns a message containing the modified file contents to the user (maybe using similar techniques you used when sending the file contents from client to server)
    f. If there is some error, e.g., uploaded file is not a valid bill, or they are not sorted in case of merge sorted, appropriate error message should be sent to the client.
6. Client receives the output from the server
    a. For sort and merge sorted commands, the client should appropriately parse the server side message to get the modified file contents and then save these to the desired files. (same file for sort, filename3 for merge sorted)

    b. For similarity command, the client should display the results on the terminal in the format specified.
7. When a client exits, the server should note this so that it can allow more connections.

Submission Instructions:

- You should have two files for server **"server_fork.c"** and **"server_select.c"** (implementation using fork and select respectively) and one for the client, named as and **"client.c"**.
- Save this in a folder named in the format: **<Roll No.>_CL2_A2**. Compress this folder to tar.gz format, creating a compressed file **<Roll No.>_CL2_A2.tar.gz**. Upload this compressed file to moodle. *Example: If your roll no. is 21CS60R05, the folder should be* 20CS60R05_CL2_A2*, and the compressed file should be* 21CS60R05_CL2_A2.tar.gz.
- Not adhering to these instructions can incur a penalty.

Marking Scheme:

Using fork()	**15**
Using select()	**15**
sort	**20**
merge	**15**
similarity	**20**
Design Decisions and Error Handling (Incl. checking validity and sorted)	**15**