Your submission archive (only .zip archives accepted) must include:

1. A **makefile** with the following rules callable from the project directory:
   (a) `make server` to build **server**
   (b) `make client` to build **client**
   (c) `make clean` to delete any executable or intermediary build files

   There is a 10% penalty for any deviation from this format. Without this makefile or something very similar, your project will receive a 0.
2. Correct C\C++ source code for an application implementing a server with behavior described below for **server**. The file(s) can be named anything you choose, but must be correctly broken into header(.h) and source(.cc) files.
3. Correct C\C++ source code for an application implementing a client with behavior described below for **client**. The file(s) can be named anything you choose, but must be correctly broken into header(.h) and source(.cc) files.
4. A README.md file describing your project. It should list and describe:
   (a) The included files,
   (b) Their relationships (header/source/etc), and
   (c) The classes/functionality each file group provides.

Note that all files above must be your original files. Submissions will be checked for similarity with all other submissions from both sections.

Overview

This project explores usage of the IPC in the form of Unix domain sockets. Your assignment is to create a client/server pair to process large text files to find and transmit lines of text requested.

In general, your task is to create a client which uses a Unix domain socket to send the name of a desired file along with requested lines of text to a server over an agreed-upon Unix domain socket. The server will respond by sending the desired lines from the file. The client will

1. Process the lines as individual equations,
2. For each equation, the client prints
   (a) The correct line number from the file,
   (b) The equation, itself, and
   (c) The equation's calculated value,

   followed by new line characters.

Server

The server should be started with a single command-line argument—the name of the domain socket it will build and monitor, i.e.,

```
./server domain_socket_name
```

If the server does not execute as indicated, the project will receive **0 points**.

1. Build a Unix Domain Socket using the provided file name in the argument and begin listening for client connections
   - A server simulate hosting $n-1$ clients, where n is the number of execution contexts on the machine it is executing.[1] [2]
     - The `get_nprocs_conf` function can be used to determine n.
   - After the `accept` function returns, the server must write
     `"SERVER STARTED"`
     `" MAX CLIENTS: 7"`
     to its terminal's standard output stream (use `std::cout` if using C++), where 7 is the number of execution contexts - 1.
2. Accept incoming connections from clients,
   - When a client connects, the server must write
     `" CLIENT CONNECTED"`
     to its terminal's standard output stream.
3. For each client connection, a server must acquire n strings from the Domain Socket,
   (a) One, the name and path to a file (relative to the project directory)
   - When the path is determined, the server must write "PATH: " followed by the provided path name on a line to the terminal's standard output stream, e.g.
     `" PATH: dat/equations_1.txt"`

   (b) Two, a set of line numbers corresponding to the lines requested from the file.

   (c) The server must print the string
   `" Lines: 1, 3, 95, ..., n"`
4. Using the path to open the indicated file
   (a) Use Domain Socket to send "INVALID FILE" instead of file lines if file cannot be opened or read
   (b) Close the client's connection, and
   (c) Wait for next connection
5. Parse the file, line-by-line, and send the requested lines to the client via the domain socket.
   (a) Use Domain Socket to send "INVALID LINE NO 834" if a line does not exist in a file,
   (b) Close the client's connection, and
   (c) Wait for next connection
6. After parsing the file and writing all lines to the client, the server must print the number of bytes (characters) of text lines transmitted to the client:
   - `BYTES SENT: 745`

   Ensure bytes/chars are counted on the server as will be on the client. The **client must calculate the same value** by counting the number of characters it is sent from the text files.
7. The server need not exit

[1] An execution context is a CPU core (or virtual core using techniques like hyper-threading).
[2] Note that you do not actually have to thread your server, only build the groundwork for future threading.

Cont.

Client

The client should start with a single command-line argument—./client—followed by:

1. The name of the Unix Domain Socket hosted by the server
2. The name and path of the text file, relative to the root of the project directory which should be searched
3. A set of numbers corresponding to lines in the file name provided.

If the client does not executed as described above, your project will receive **0 points**.

The client is used to connect to the `text-server`, pass along file, and search string information, count the number of lines of text returned, and print each line to the standard output stream of its terminal.

The client must behave as follows

1. Using the domain socket file name, opens a socket to an existing server
   - When a server accepts the client connection, the client must write
     `"SERVER CONNECTION ACCEPTED"`
     to the standard output stream of its terminal
2. Sends the text file name and path to the server
3. Sends the file lines
4. Reads all data sent from server, calculating the indicated equations from the file. For example:

   ```
   ./client socket_name 191 2 17
   line 2:  3 + 4 x 6 = 27
   line 17:  8 + 7 + 3 x 4 = 27
   line 191:  4 x 9 - 3 - 3 - 3 = 27
   ```

5. Writes the number of bytes received from the server to the standard output stream of its terminal, e.g.,
   `BYTES RECEIVED: 152`
   This number will depend on how much data you send from the server, e.g., do you send new line characters (\n) or do you delineate in some other way? However you choose to calculate it, make sure you use the same method between your client and server. The numbers much match which compared.
6. Terminates by returning 0 to indicate a nominative exit status

Cont.

Notes

You are provided two test files of varying length to test your code; check the dat directory in the provided directory:

- `dat/bankloan1.csv`, 223B.
- `dat/bankloan2.csv`, 950KB

References

- Manual with example code
- A simple makefile tutorial. Colby University.
- Beej's Quick Guide to GDB. Beej, 2009.
- Getting Started With Unix Domain Sockets. MATT LIM, 2020.
- An Introduction to Linux IPC. Kerrisk, 2013.

Grading

Grading is based on the performance of both the client and server. Without both working to some extent you will receive 0 POINTS. There are no points for "coding effort" when code does not compile and run.

The portions of the project are weighted as follows:

1. **makefile**: 10%
2. **text-server**: 40%
3. **text-client**: 40%
4. **README.md**: 10%

The End.