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

1. A **makefile** with the following rules callable from the project directory:
   (a) `make text-server` to build **text-server**
   (b) `make text-client` to build **text-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 **text-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 **text-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 task is to create a client/server pair to process large text files to find and transmit lines of text containing given search strings.

In general, your task is to create a client which uses a Unix Domain Socket to send the name of a desired CSV file along with search terms to the server. The server will find all lines containing all or part of the terms and return each one using the Domain Socket. The client will format the lines and print each one, line-by-line (each line will end in a newline char).

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.,

```
./text-server csv_server_socket
```

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

The server must perform as follows:

1. Build a Unix Domain Socket using the provided file name in the argument and begin listening for client connections
   - A server should host $n - 1$ 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"`
     `"\tMAX CLIENTS: 7"`
     to its terminal's STDLOG (use `std::clog` and `std::endl` from `iostream` 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 `STDLOG`.[3]
3. For each client connection, a server must acquire n strings from the Domain Socket,
   (a) 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 STDLOG, e.g.
     `PATH: dat/dante.txt`

   (b) A set of operator delimited search terms and operators; when the search tokens are parsed, the server must write
   - "Operation: " followed by "n/a", "AND", or "OR", depending on the operators discovered or their lack
   - "SEEKING: " followed by a comma-delimited list of tokens to the terminal's STDLOG, e.g.
     `SEEKING: default, industrial`
   - For example,
     - Input "default" would result in
       `OPERATION: n/a`
       `SEEKING: default`
     - Input "default + industrial" would result in
       `OPERATION: OR`
       `SEEKING: default, industrial`
     - Input "default x industrial" would result in
       `OPERATION: AND`
       `SEEKING: default, industrial`
4. Using the path to open the indicated file

[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.
[3] Use `std::clog` and `std::endl` from `iostream` if using C++.

Cont.

   (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 any line containing the provided search expression to the client via the domain socket

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

Cont.

Client

The client should start with a single command-line argument—./text-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. The search expression for which the file will be parsed, e.g.,
   - ./text-client csv_server_sock default
   - ./text-client csv_server_sock default + industrial
   - ./text-client csv_server_sock default x industrial
   - ./text-client csv_server_sock default x professional x 41

Note that you do not need implement mixed operation expressions, e.g. `default + professional x 41`, though you should test for this, print "Mixed boolean operations not presently supported," to STDERR, and exit with return value 2.

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 STDOUT of its terminal.

The client must behave as follows

1. Using the domain socket file name, opens a socket to an already-existing server
   - When a server accepts the client connection, the client must write
     `"SERVER CONNECTION ACCEPTED"`
     to the STDLOG of its terminal
2. Sends the text file name and path to the server
3. Sends the search expression to the server
4. Reads all data sent from server, writing the lines of text to STDOUT, line-by-line
   - The client must begin each line of text from the server with a count starting with 1 and must use a tab character (\t) after the count, e.g.,
     `./text-client csv_server_sock dat/bankloan1.csv 34 + repay`
     results in

     ```
     1\t1, industrial, 34, 2.96, repay
     2\t6, industrial, 61, 2.52, repay
     3\t7, professional, 37, 1.50, repay
     4\t8, professional, 40, 1.93, repay
     ```

     Note that the \t should be an actual tab character.
5. Writes the number of bytes received from the server to the STDLOG 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`, 432B.
- `dat/bankloan2.csv`, 1.9KB

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.