Group: (Rachel)^3

(Rachel, Renae, and Tracy)

I. Message formats

Messages:

- "LIST"
  - Client: used to request a list of files that the server has
  - Server: used to respond to a LIST message from a client with a list of files
- "PULL"
  - Client: used to request that the server send files
- "PUSH"
  - Client: used to transfer files from Client to Server when the user enters "sync"
  - Server: used to respond to PULL messages to transfer requested files to the Client
- "BYE!"
  - Client: used to indicate to the Server to end the connection

II. Structures

- `header`: forms the beginning of each message
  - Fields:
    - `type`: a 4-character string specifying the type of message; can be "LIST", "PUSH", "PULL", or "BYE!"
    - `length`: indicates the number of data items included in the message
      - 0 for LIST requests from the client and BYE! messages, as they don't include other data
      - for PULL and LIST messages, this will be the number of `file_name` structs in the packet (the number of files being requested/returned)
      - for PUSH messages, this will be the number of `push_file` structs in the packet (also the number of files being sent)
- `file_name`: used by the server to respond to LIST messages and by the client to request files
  - Fields:

        - `filename`: the name of the file being listed/requested
        - `hash`: the SHA256 hash of the file (to detect duplicate files)
- `push_file`: used by both client and server to transfer files
    - Fields:
        - `size`: the size (in bytes) of the file immediately following this struct in the packet
        - `name`: the name of the file immediately following this struct in the packet

III. Implementation

1. Client

a) Command-line arguments: path of directory that contains music files, server IP address/name and port number

Usage:
./Project4Client -d <directory containing music files> [-s <server IP/hostname>:<port>]

b) When run, the client opens a connection with the server and displays a menu to the user. The user can enter four possible commands to standard input:

1. list:
    - sends LIST request to the server
    - reads in the server's reply, containing a list of file names and hashes (as an array of `struct file_names`)
    - prints to `stdout` the list of files on the server
        - if any files in the server's list have the same hash as files on the client, the client prints "(duplicate content with [file name on client])" after the file name
2. diff:
    - sends a LIST message to the server and reads in the reply
    - prints to `stdout` a list of files that the client has and the server doesn't, and a list of files that the server has and the client doesn't
    - if files are found that have the same content as a file on the server but a different name, the program prints this information for the user and gives the user the option of renaming the local file to match the name of the file on the server
3. sync:

- sends a LIST message to the server and reads in the reply, comparing the hashes of the server's reply with the hashes of the client's files
- files with the same content as a file on the server but with a different name are handled in the same way as in `diff`: the program prints a message indicating which files have identical content but different names and asks if the user wants to rename the local file to match the name on the server
- sends a PULL message to request the files that the server has and the client doesn't
  - excludes files that are duplicates of files the client already has
- reads in the server's PUSH response and writes those files to its music directory
- sends the server a PUSH message containing the files that are not on the server

4. bye!:
   - sends a BYE! message to the server, closes the connection, and exits the program

2. Server

a) Command-line arguments: port and the name of the file in which to write client information

to run: ./Project4Server -p <port> -l <log file name>

b) When run, the server sets up to receive connections. When a client connects, it accepts the connection and creates a thread for that client's session using `pthreads()`. As it runs, it stores information about the messages it receives from the client and the files that it knows the client has (based on the information it receives from the messages). Each client is identified by its IP address.

We chose to use `pthreads()` because it allows the server to accept connections from multiple clients on a single port. This way, the server doesn't have a set limit on the number of clients it can handle, and the client code doesn't have to handle connecting to different ports to avoid conflicts, so the clients can be kept a little simpler and don't need to be configured so that they all connect to different ports. This also means that the server can accept many clients to a single port instead of having to use many ports.

The server then waits to receive a message. It first reads in the bytes of the `header` at the beginning of the packet and determine the type of the message:

1. LIST
   - the server adds to its log that it received a list request from this client
   - it gets a list of `.mp3` files in its directory and creates a `file_name` struct for each containing the file's name and the hash of its contents

- it constructs a header with type LIST and sets length to be the number of files in its directory, then sends this to the client

2. PULL
    - the server adds to its log that it received a pull request and includes a list of files that it is sending to the client; it adds the files being sent to its running list of files that are on that client
    - it checks its directory for each file; if the file is found, it creates a `push_file` struct for that file specifying the file's name and size (in bytes), then writes the file's contents to the packet after
    - it sends the `push_file` structs, followed by the file contents, in a PUSH message to the client
3. PUSH
    - the server adds to its log that it received a PUSH message and includes a list of the files it received from the client; it also adds these files to the list it keeps of the files on that client
    - for each file, it first reads the `push_file` struct, and uses that information to read in the file and write it to the current directory
4. BYE!
    - the server closes the connection with the client and writes the log and file information from the client to the log file
    - a boolean flag `log_file_open` (that is global to the server) keeps track of whether the log file is being written; if it is, all other processes wait until it is free to write, to avoid race conditions between the threads