CS 392 Spring 2024 Systems Programming — Project

Preliminary

Concepts you need to understand for this assignment:

- File I/O, both low- and high-level;
- The steps of establishing socket connections;
- Multiplexed I/O.

Your task is to write a Trivia game application as shown in class. The server acts as the host who sends questions to the players, and the clients are the players.

1 Part 1: The Server (70 pts)

1.1 Task 1: Establish the Server with `getopt` (10 pts)

We will start writing the server first, where the first step is to parse arguments from command line using `getopt()`. It is very similar to `getopts` utility we used in bash script.

There are four possible flags, all of which are optional for the program:

```
Usage: ./server [-f question_file] [-i IP_address] [-p port_number] [-h]

  -f question_file      Default to "question.txt";
  -i IP_address         Default to "127.0.0.1";
  -p port_number        Default to 25555;
  -h                    Display this help info.
```

Above is also an example output of the help message. Note that `./server` in the help message **must not** be hardcoded – it has to be replaced by the actual name of the server program. If an unknown flag is passed, please print the following message and exit with failure:

```
Error: Unknown option '-<c>' received.
```

where `<c>` is replaced by the actual unknown flag.

After parsing the arguments, you will follow the steps to create a server socket, bind it, and listen. Please limit the maximal number of connections to 2 or 3 when listening (for your own good). During any step if there's an error, you must use `perror()` to print out the error, and exit with 1. If `listen()` succeeds, print the following message:

```
Welcome to 392 Trivia!
```

1.2 Task 2: Read the Question Database (10 pts)

Once the server has been established, we can create our question database before the game starts. This is accomplished by reading a plain text file, which has the format as follows:

```
Which Game of Thrones character is known as the Young Wolf?
Robb_Stark Arya_Stark Sansa_Stark
Robb_Stark

What city hosted the 2000 Olympic Games?
Tokyo Beijing Sydney
Sydney
```

For each question entry, the first line is the question itself, and the second line the three possible options, and the third line the answer. Between every two question entries there's an empty line. Notice that if the answer has a space in it, there's always an underscore, so a white space is always used to separate three options.

We use the following structure to store each question entry:

```
struct Entry {
    char prompt[1024];
    char options[3][50];
    int  answer_idx;
};
```

where `prompt` is the question itself, `options` are the three options, and `answer_idx` is the index to the correct answer in `options`.

The function that reads the question database is declared as follows:

```
int read_questions(struct Entry* arr, char* filename);
```

where `arr` is an array of question entries and `filename` the path to the question database file. The function returns the actual number of question entries read from the file.

You can assume there's no more than 50 questions, so a hardcoded `struct Entry` array of 50 is acceptable.

1.3 Task 3: Accepting Players (20 pts)

Now is the time to accept new players!

When the maximal number of players have been reached, you should print the following message on screen:

```
Max connection reached!
```

and close the new player's file descriptor.

If there's still empty spot for players, print the following message on screen:

```
New connection detected!
```

and send out the following message to the player:

```
Please type your name:
```

After you receive the name sent by the player, you will print the following message on screen:

```
Hi <NAME>!
```

where `<NAME>` is replaced by the name sent from the player. You don't need to check duplicates; it's ok to assume each player will put in a different name.

For each player, you will use the following structure to store their information:

```
struct Player {
    int  fd;
    int  score;
    char name[128];
};
```

where `fd` is the file descriptor assigned to this player when connected to the server, `score` the score of this player, and `name` the player's name.

Once the last player has typed their name and you received it in the server, you will print out the following message on screen:

```
The game starts now!
```

If a player quits their program, print the following message on screen:

```
Lost connection!
```

1.4 Task 4: Start the Game! (30 pts)

The game starts now! You will print one question and the three options to both the screen and each of the player's terminal.

On the screen, the question and the options are formatted as follows:

```
Question <QUESTION_NO>: <THE_QUESTION>
1: <OPTION_1>
2: <OPTION_2>
3: <OPTION_3>
```

where `<QUESTION_NO>` starts from 1.

On each player's terminal, the question should be formatted as follows:

```
Question <QUESTION_NO>: <THE_QUESTION>
Press 1: <OPTION_1>
Press 2: <OPTION_2>
```

```
Press 3: <OPTION_3>
```

Once a player pressed one of the numbers, we apply rewards or penalty to the player. The rule is, if a player answered the question right, they get one point; otherwise they get -1. Regardless of the option the player chose, you will display the correct answer on the screen, and also broadcast it to all the players. Then you can move on to the next question.

When all the questions have been answered, display the winner with the following message on the screen:

```
Congrats, <WINNER>!
```

Then close all the connections, and exit the server program.

2 Part 2: The Client (30 pts)

2.1 Task 5: Parse Arguments and Connect to the Server (10 pts)

The client (player) side is relatively easier. The first task is still to parse the flags, all of which are optional for the program:

```
Usage: ./client [-i IP_address] [-p port_number] [-h]

  -i IP_address         Default to "127.0.0.1";
  -p port_number        Default to 25555;
  -h                    Display this help info.
```

Again, `./client` in the help message **must not** be hardcoded – it has to be replaced by the actual name of the client program. If an unknown flag is passed, please print the following message and exit with failure:

```
Error: Unknown option '-<c>' received.
```

where `<c>` is replaced by the actual unknown flag.

After parsing the arguments, you will follow the steps to create a socket and connect it to the server.

The function is declared as follows:

```
void parse_connect(int argc, char** argv, int* server_fd);
```

2.2 Task 6: Enter the Game (20 pts)

Once connected to the server, the server will send out the message to let the player type their name (see Part 1, Task 3). The player types their name, and wait for the game to start. During the game, if another player answered the question first, the server will send out the correct answer to all the players, and start the next question. If the current player knows the answer, they should act fast and press 1, 2, or 3, before the correct answer shows up.

After all questions have been asked, the server cuts the connection to all the players, and the player program can exit.

Hint: the client side also needs to implement multiplexed I/O, because after the question is shown to the player, there are two possible ways to move on to the next question: either the current player presses 1, 2, or 3 to answer the question, or another player answers the question first and the server sends out the correct answer. Therefore, you need to monitor two file descriptors to see which one is ready first: `stdin`, or the server's file descriptor.

3 Grading

This project has less restraints and is more of a free-style programming. As long as you are handling different cases rationally (*e.g.,* exiting with useful error messages if an error occurs), and the program is a well-designed software, you will get the points. So be creative! If you are not sure about something, always ask.

However, we do deduct points for the following items:

- **-100:** the code does not compile through the Makefile, or executes with run-time error;
- **-100:** the code is generated by AI;
- **-100:** the code does not use socket and multiplexed I/O;
- **-30:** memory leak through valgrind (only "definitely lost" category);
- **-10:** no pledge and/or name in C file.

Earlybird Extra Credit: 2% of extra credit will be given if the homework is finished two days before the deadline. For specific policy, see syllabus.

Deliverable

Submit the following files on Canvas. No need to zip.

(1) `Makefile`;
(2) `server.c`;
(3) `client.c`.