Assignment 4: A multi-player word game

Due Apr 4 by 4pm   **Points** 10

First task: Makefile

We have provided starter code for the Makefile. Completing it is your first task and the starter code will not compile until you complete these tasks.

To avoid port conflicts when testing your programs on teach.cs, you will use `PORT` (see below) to specify a port number. Select that number using the same algorithm we used for lab 10: take the last four digits of your student number, and add a 5 in front. For example, if your student number is 1008123456, your port would be 53456. Using this base port, you may add 1 to it as necessary in order to use new ports (for example, the fictitious student here could also use 53457, 53458, 53459, 53460). Sometimes, when you shutdown your server (e.g. to compile and run it again), the OS will not release the old port immediately, so you may have to cycle through ports a bit.

- In `wordsrv.c`, replace `x` with the port number on which the server will expect connections (this is the port based on your student number):

  ```
  #ifndef PORT
    #define PORT y
  #endif
  ```

- Then, in `Makefile`, replace `y` with your student number port plus 1:

  ```
  PORT= x;
  ```

Now, if you type `make PORT=53456` the program will be compiled with `PORT` defined as `53456`. If you type just `make`, `PORT` will be set to `y` as defined in the makefile. Finally, if you use `gcc` directly and do not use `-D` to supply a port number, it will still have the `x` value from your source code file. This method of setting a port value will make it possible for us to test your code by compiling with our desired port number. (Read the Makefile. It's also useful for you to know how to use `-D` to define macros at command line.)

Word guessing game server

For assignment four, you will write a word guessing game server. The game is similar to the game you wrote in assignment two, but with some key differences. In particular, the server chooses the word before the players start guessing and doesn't change the word during a round. The game is played by one or more

players who take turns guessing letters.

Players will connect using `nc`. When a new player connects, the server will send them "Welcome to our word game. What is your name?". After they input an acceptable name that is not equal to any existing player's name and is not the empty string, they are added to the game, and all active players (if any) are alerted to the addition (e.g., "Karen has just joined.") If they input an unacceptable name, they should be notified and asked again to enter their name.

With your server, players can join or leave the game at any time. A player leaves the game by exiting/killing `nc`.

New players who have not yet entered their names are stored in a linked list, and each new player is added to the front of the list (see `add_player`). Players in this list do not get turns, and do not receive any messages about the status of the game.

When a player has entered their name, they are moved from the new players list to the head of the list of active players that is stored in the `game_state struct`.

Adding a player to the active list in the `game_state struct` must not change whose turn it is, unless this is the first player to join the game. When someone leaves, if it is their turn, the turn has to pass to the next player in the list, not to the top of the list. When the last player leaves, the variable that stores the pointer to the player with the next turn is set to `NULL`.

For each turn, or to newcomers moved to the active list, you display the game state in a simple text format. The output should look something like this:

```
***************
Word to guess: --u--s
Guesses remaining: 3
Letters guessed:
u s z
***************
```

Then, prompt the player whose turn it is with the query "Your guess?", and tell everyone else (for example) "It's Jen's turn."

The player is expected to type a single lowercase letter, followed by the enter (or return) key. Note that because we are running `nc` with the `-c` or `-C` option it will send the network newline (`'\r\n'`). If the user enters anything else, such as multiple characters or a character that is not a lowercase letter between `'a'` and `'z'`, the guess is invalid. You should tell the player this (the exact format of this message is up to you) and ask them again for their guess. An invalid guess does not decrement the number of guesses.

Once the player makes a valid guess, tell everyone what guess that player made. (The exact format of this

message is up to you.)

If the player guesses a letter that has not already been guessed and that letter is in the word, the player gets to guess again. Prompt the player whose turn it is with the query "Your guess?", and once again tell everyone else (for example) "It's Jen's turn."

The game ends in two situations. First, it ends when a player guesses the last hidden letter. Announce to the player, for example, "Game over! You win!" and tell everyone else, for example, "Game over! Karen won!" Second, the game ends when the players have zero guesses remaining. In that case, tell everyone "No guesses left. Game over."

Once the game has ended (for either reason), it should restart with a new word to guess. The player who was supposed to guess next, gets the first turn.

Be prepared for the possibility that the player drops the connection (disconnects) when it is their turn to guess. Furthermore, you must notice user input or dropped connections even when it isn't that player's turn. If the player types something other than a blank line when it is not their turn, tell them "It is not your turn to guess." For a blank line, you can say "It is not your turn to guess" or you can ignore it, whichever you prefer.

How does the server tell when a client has dropped a connection? When a client terminates, the socket is closed. This means that when the server tries to write to the socket, the return value of `write` will indicate that there was an error, and the socket is closed. Similarly, if the server tries to read from a closed socket, the return value from the `read` call will indicate if the client end is closed.

As players connect, disconnect, enter their names, and make moves, the server should send to `stdout` a brief statement of each activity. (Again the format of these activity statements is up to you.) Remember to use the network newline in your network communication throughout, but not in messages printed on `stdout`.

You are allowed to assume that the user does not "type ahead" -- if you receive multiple lines in a single `read()`, for this assignment you do not need to do what is usually required in working with sockets of storing the excess data for a future input.

However, you can't assume that you get the entire player name or guess in a single `read()`. For the input of the player name or guess, if the data you read does not include a network newline, you must loop to get the rest of the input.

Sample Interactions

To help you better understand the game play, we provide three sets of sample interactions. The messages displayed may vary, but behaviour of your program should be consistent with these interactions.

Please note that these interactions do not cover all possible scenarios and your program must take other

situations into account. For example, the number of players can vary, and players can connect and disconnect at any time, including when it is a player's turn.

In these interactions, our server logs show that we always read the word from index 0, since the dictionary used contained only one word for demonstration purposes. With a larger dictionary, we'd expect that index number to vary.

- Interaction 1: **game1_client_jen.txt** 📄, **game1_client_karen.txt** 📄, **game1_server.txt** 📄
    - player Jen connects, then enters name
    - player Karen connects, then enters name
    - players make guesses (some correct, some incorrect, one out of turn) until player Karen wins
    - a new game starts
    - player Jen disconnects
    - player Karen disconnects
    - server terminated
- Interaction 2: **game2_client_jared.txt** 📄, **game2_client_mark.txt** 📄, **game2_client_anisha.txt** 📄, **game2_client_stathis.txt** 📄, **game2_server.txt** 📄
    - player Jared connects, enters name and guesses
    - during Jared's turn, player Mark connects and enters name
    - during Jared's turn, player Anisha connects and enters name
    - when Jared's turn ends, player Anisha guesses; players Mark and Jared disconnect during Anisha's turn
    - the game ends with no winner
    - a new game starts
    - player Anisha guesses once
    - player Anisha disconnects
    - player Stathis connects and enters name
    - server terminated
- Interaction 3: **game3_client_jingyi.txt** 📄, **game3_client_tudor.txt** 📄, **game3_server.txt** 📄
    - player Jing Yi connects, but does not enter name yet
    - player Tudor connects, enters name
    - player Tudor guesses incorrectly; player Tudor's turn again
    - during player Tudor's turn, player Jing Yi enters name
    - when player Tudor's turn ends, it becomes player Jing Yi's turn
    - server terminated

Testing

To use `nc`, type `nc -C hostname yyyyy` (use lowercase -c on Mac), where `hostname` is the full name of the

machine on which your server is running, and `yyyyy` is the port on which your server is listening. If you aren't sure which machine your server is running on, you can run `hostname -f` to find out. If you are sure that the server and client are both on the same server, you can use `localhost` in place of the fully specified host name.

To test if your partial reads work correctly, you can send a partial line (without the network newline) from `nc` by typing Ctrl-D.

Marking

The TAs will be reading the output of your program (rather than using a script to match it against expected output). This means that your message do not need to exactly match the ones in this handout. However, your messages will be read by a person, so you want to make your output meaningful and readable.

Your code may be evaluated on:

- **Code style and design**: At this point in your programming career you should be able to make good choices about code structure, use of helper functions, variable names, comments, formatting, etc.
- **Memory management**: your programs should not exhibit any memory leaks. Use `valgrind` to check your code.
- **Error-checking**: library and system call functions (even `malloc`!) can fail. Be sure to check the return values of such functions, and terminate your program if anything bad happens.
- **Warnings**: your programs should not cause any warnings to be generated by `gcc -Wall`.

Reminder

Your program must compile on `teach.cs` using `gcc` with the `-Wall` option and should not produce any error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Submission

We will be looking for the following files in the a4 directory of your repository:

- Makefile
- wordsrv.c
- socket.c
- socket.h

- gameplay.c
- gameplay.h

No other files will be accepted. Do not commit `.o` files or executables to your repository.