

Assignment 4

Due Apr 8 by 4pm **Points** 10

Assignment 4: Twitter Server

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 `twerver.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 x
#endif
```

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

```
PORT=y;
```

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.)

Twitter Server (twerver)

For assignment four, you will write a simple version of a Twitter server. When a new user connects, the server will send them "Welcome to CSC209 Twitter! Enter your username: ". After they input an acceptable name that is not equal to any existing users's username and is not the empty string, they are added to Twitter, and all active users (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, users can join or leave Twitter at any time. A user leaves by issuing the quit command or exiting/killing `nc`.

New users who have not yet entered their names are added to the head of a linked list of new users (see `add_client`). Users in this list cannot issue commands, and do not receive any announcements about the status of Twitter (e.g., announcements clients connecting, disconnecting, etc.).

When a user has entered their name, they are removed from the new clients list and added to the head of the list of active clients. Once a user becomes active, they can issue commands. The possible commands are:

- follow username
 - When this user follows username, add username to this user's following list and add this user to username's list of followers. Once this user follows username, they are sent any messages that username sends.
 - A user can follow up to `FOLLOW_LIMIT` users and a user can have up to `FOLLOW_LIMIT` followers. If either of those lists have insufficient space, then this user cannot follow username, and should be notified of that.
 - If unsuccessful (e.g., username is not an active user), notify the user who issued the command.
- unfollow username

- If this user unfollows username, remove username from this user's following list, and remove this user from username's list of followers. Once this user unfollows username, they are no longer sent any messages that username sends.
- If unsuccessful (e.g., username is not an active user), notify the user who issued the command.
- show
 - Displays the previously sent messages of those users this user is following.
- send <message>
 - Send a message that is up to 140 characters long (not including the command send) to all of this user's followers.
 - You may assume all messages are between 1 and 140 characters.
 - If a user has already sent `MSG_LIMIT` messages, notify the user who issued the command and do not send the message.
- quit
 - Close the socket connection and terminate.

If a user issues an invalid command or enters a blank line, tell them "Invalid command".

When a user leaves, they must be removed from all active clients' followers/following lists.

Be prepared for the possibility that a client drops the connection (disconnects) at any time. 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 clients connect, disconnect, enter their names, and issue commands, 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 client 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 name or command in a single `read()`. For the input of the name or command, if the data you read does not include a network newline, you must get the rest of the input from subsequent calls on `read()`.

The server should continue running even if all clients have disconnected.

Sample Interactions

To help you better understand our Twitter network, 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 users can vary, and users can connect and disconnect at any time.

- Interaction 1: [user Ruiqi](#) , [user Mit](#) , [server log](#) 
 - user Ruiqi connects, but does not enter name yet
 - user Mit connects, enters name
 - Mit disconnects
 - Ruiqi enters name
 - server terminated
- Interaction 2: [user Stathis](#) , [user Caroline](#) , [server log](#) 
 - user Stathis connects, then enters name
 - user Caroline connects, then enters name
 - Caroline issues an invalid command
 - Caroline sends a message
 - Stathis follows Caroline
 - Stathis runs show
 - Caroline sends another message
 - Caroline disconnects
 - Stathis runs show

- server terminated
- Interaction 3: [user Ibrahim](#), [user Rhys](#), [user Sean](#), [server log](#)
 - user Ibrahim connects, then enters name
 - user Rhys connects, then enters name
 - Rhys follows Ibrahim
 - user Sean connects, then enters name
 - Sean follows Rhys
 - Ibrahim follows Rhys
 - Ibrahim sends a message
 - Ibrahim disconnects
 - Sean unfollows Rhys
 - server terminated

Testing

To use `nc`, type `nc -C hostname yyyy` (use lowercase `-c` on Mac), where `hostname` is the full name of the machine on which your server is running, and `yyyy` 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 the 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. Since your output will be read by a human, make sure that it is 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. For any malloc calls, you should have corresponding free calls. If the server program terminates with a signal, the free calls won't necessarily be executed, but they should be present in the 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 program 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 or warnings. 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 file in the a4 directory of your repository:

- twerver.c

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