Quiz Server in C Programming Language with POSIX Threads and Multiplexing

Final Project, Part 3

What is working?

- Works with GUI client provided on Moodle
- Users can create groups with the specified properties such as Groupname, Quiz topic, desired size of the group. After this client is not able to create any other group or join it before he cancels the created group. This client socket's file descriptor is recorded in the Group structure as admin_sock. The OPENGROUPS|topic|name|size|curr command from server is sent to client with information about the all open groups with all fields that should be in the group and, additionally, the current size of the group. SENDQUIZ\r\n is sent if the group is created successfully and BAD\r\n if there is a group with a such name because duplicates are not allowed or if he specifies the desired size to be less or equal to 0.
- After SENDQUIZ\r\n command client chooses the text file with question formatted according to the protocol (assumed that client is well-behaved). The text with the text size without CRLF is sent to the server, which reads it in multiple reads before the quiz is not fully read. Size here is needed because text file can be of very large volume and server will need to read the whole text. Text is parsed into questions and answers and the Quiz structure is filled with these values, while this structure itself is added to the structure of the Group.
- Users can join groups where the current size is smaller than the desired one. If join was successful, OK\r\n is sent; otherwise, if the group is already full or there are some other problems preventing join BAD\r\n is sent. If successful, the number of current users in the group is increased, and client is added to the array of Client structures.
- When the number of clients in the groups is increased to be equal to the desired one, the new thread is created by the last client and the index of the current group is passed as argument to the thread. The quiz is immediately started sending questions and waiting for a timeout to get an answer (I have configured the timeval structure of select() system call to 60 seconds to wait while fd changes the state). According to my implementation, multiplexing should occur here to identify which clients are able to send an answer; however, this implementation was not feasible to be accomplished successfully. I'll explain it in the next section.
- There is a function find_group(char *join_group) that looks through the global array of groups and returns the index of the group if it is there and -1, otherwise. Also, there is open_groups() function that returns a response to GETOPENGROUPS query of client. I have used this functionality in many parts of the code, so I've decided to make them functions.

What is not working?

There are some bugs that could potentially make some quiz functionality not available, i.e. answers are not read from clients[1]. The reason behind this is that I have created not feasible design. In my design the same sockets that are stored in the array of clients in group are accessed by 2 threads simultaneously, what creates an undefined behavior. The solution for this was to reorganize the code in the way that multiplexing is made in the main thread and only Quiz is read in another thread because this operation is heavy enough to block the main thread. In that implementation I could use FD_SET and FD_CLR to add and remove socket from one place to another. Now, because it was not possible to accomplish given the scope of available time, the answers of clients from quiz_thread are sent to run_thread, which, actually, created the quiz_thread.

Deviations

During the implementation I have changed some of data structures (Quiz structure remained without changes). This is illustrated below.

Part 1 Submission	**Final Part Submission**
struct Client { **char name[256];** **int score;** **int is_admin;** **int is_member;** **int is_in_quiz;** **Int socket_fd;** **} client;**	**typedef struct Client {** **char name[256];** **int score;** **int group_index;** **int is_in_quiz;** **int socket_fd;** **} Client;**
typedef struct Group { **char group_name[256];** **char quiz_topic[256];** **int des_size;** **Client *clients;** **pthread_mutex client_num;** **pthread_cond_t last_client;** **int current_size;** **Quiz quiz;** **} Group;**	**typedef struct Group {** **char group_name[256];** **char quiz_topic[256];** **int des_size;** **int quest_num;** **Client clients[1010];** **int admin_sock;** **pthread_mutex_t client_num;** **int current_size;** **struct Quiz quiz;** **} Group;**

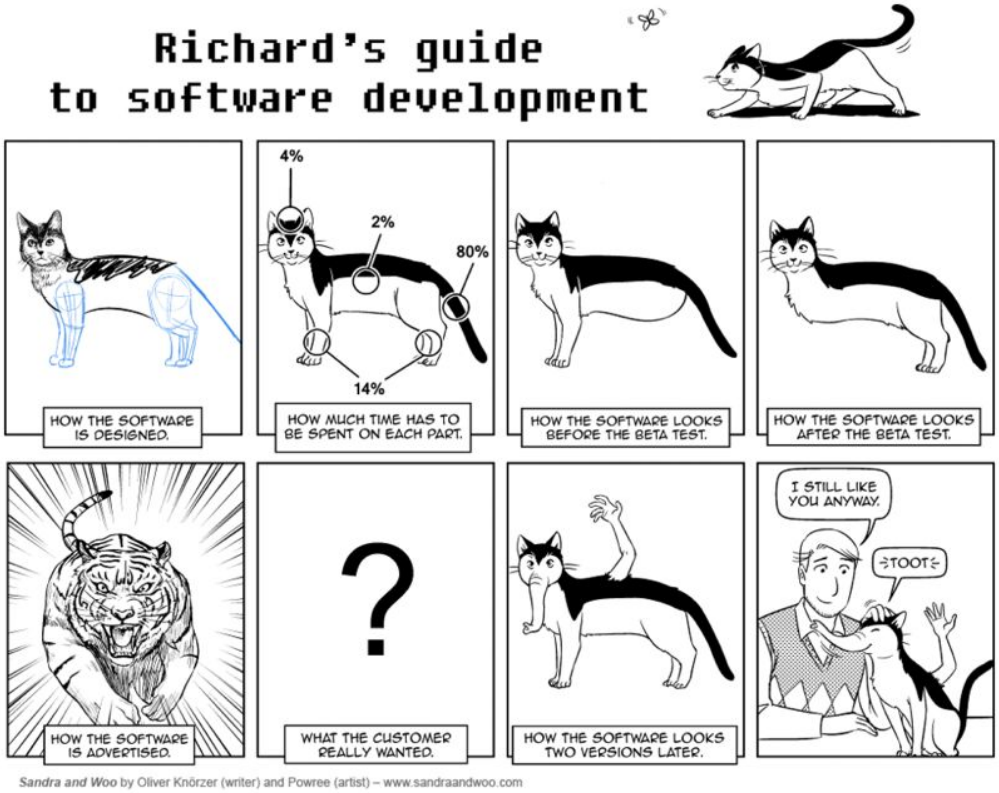
[1] It was not possible to finish scoring functionality due to the technical debt accumulated during the development process. However, the implementation should be similar to Homework 4.

The main changes:

- I have removed conditional variable from the Group structure because it is not useful in this server as there should be no busy waiting, which will block, for example, Admin of the group from canceling until there are enough clients and conditional variable broadcasts.
- I have added *quest_num* variable to Group to make it easier to traverse through all questions of the quiz when sending them to clients, and I have added *admin_sock* to it because it seems to be a better decision to store it as a distinct variable instead of traversing through the list of all clients checking value of *is_admin* (this field was removed from Client).
- Finally, I have removed *is_member* variable and added *group_index* to Client. Variable is_member turned out to be unnecessary since only joined clients are added to the group, while variable group_index made it easier to work on the implementation of LEAVE function.

<u>What I have learned/understood from this project?</u>

- My implementation of Quiz Server turned out to be a little bit over-complicated and not feasible to accomplish. So, I have learned that all parts of Software Development life cycle are important and only combined together and iterated for plenty of times they can be helpful. I have understood that my initial design was not well thought; therefore, even if my final project kind of satisfies the initial one, it doesn't work properly.



(*Software Development Life Cycle* 2018)

- It is better to write well-formatted and readable code from the beginning because it helps to understand the code later. For example, in the previous submission I have written that I have lost in amongst 500+ lines of the code from the previous homework, which could, probably, be partially used in this one. Previously, I have made relatively big projects in Python, where the well-formatted code is a requirement to compile the program but I have understood that it also can help to work with the code. Especially if the codebase is big.
- How to write multithreaded servers, multiplexing concept and the overall concept of threads. Now, I want to add multithreaded behavior to my previous project, while multiplexing was covered in 3 of 5 courses I am taking now, so it was interesting to learn how it is implemented at the low level.
- Enough time should be given for testing of implementation and for work with techniques that haven't been used before. I have encountered the problem with my multiplexing server design only in the end, what left me without enough time to rewrite the code because of finals.

References

Software Development Life Cycle. 2018. Image. https://goo.gl/ADzwbx