Fancy Quiz Server
Part III. Final Report

Explain your key data items and data structures, and how your server is going manage them to deliver the quizzes to the groups, and give good, fair service to the clients. Focus on technical details.

As I intended in my first Architecture and Design mixing of threads and multiplexing was decided be used in this project. Multiplexing via select() system call was used firstly for accepting new clients and I took the example code of echoserver as a starting point and added more features. Furthermore, instead of creating just one thread per group I created two threads:

a) for receiving the quiz from the group creator (could be the long operation):

```
status1 = pthread_create(&thread, NULL, getQuiz, (void *)fd);
```

b) for starting the quiz only when the needed number of clients arrive:

```
if(groups[leadsock]-> currSize == groups[leadsock]->size){
 status2 = pthread_create(&thread, NULL, startQuiz, (void *)leadsock);
}
```

This second thread is created in the body of if statement where JOIN method is handled since each time new client arrives and joins the group, the current size of that group is incremented.
Also when new group is created leader's socket is removed from the global file descriptors set and added later to the local file descriptors set in each group where the quiz is going. The same is with the sockets of the clients who joined groups.

```
FD_CLR( fd, &afds );
// lower the max socket number if needed
if ( nfds == fd+1 )
    nfds--;
 }
```

It should be mentioned that one of the problems I struggled after the first quiz test was that leader who tried to read the quiz the second time was blocked. I found that mistake was in not removing the file descriptor of the leader from the main thread when passing it sendQuiz thread. That was fixed later.

Data structures:

```
typedef struct{
    int mysock;
    int score;
    char name[30];
    char groupName[30];
    int isLeader;
}client;

typedef struct{
```

```
    char topic[30];
    int size;
    int currSize;
    char groupName[30];
    client *members[1024];
    char *lines[1010];
    int numberOfQuestions;
    int answ[128];
}group;
```

My main data structures did not change much I just added more fields, namely

- `int answ[128];` – to group structure in order to set the flag if someone answered first for the given question;

To store all groups and clients pointer to arrays were created:

```
client *clients[1010];
group *groups[32];
```

Each time when a new group or client is added the memory is dynamically allocated for each element using socket number, which is a file descriptor as an index:

```
clients[fd] = (client*)malloc(sizeof(client));
groups[fd] = (group*)malloc(sizeof(group));
```

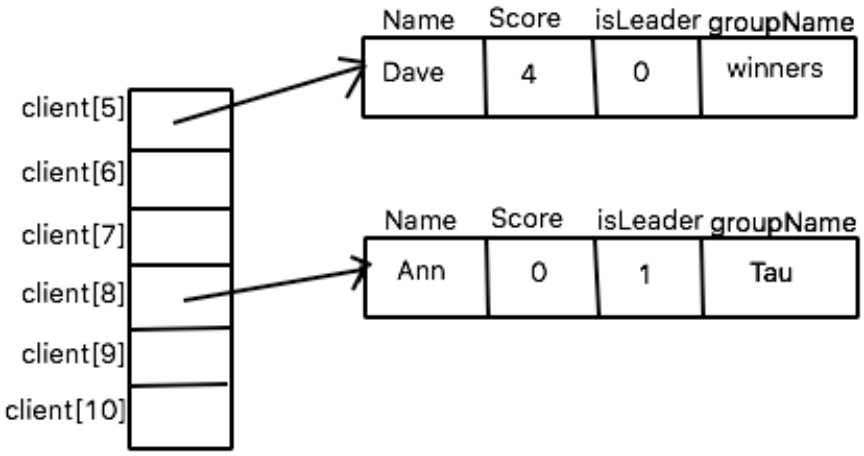

Figure 1. Array diagram for clients

As it is illustrated in Figure 1 each element in clients[ ] points to one client which has attributes: Name, Score, Leader flag and the name of the group to which the client belongs.

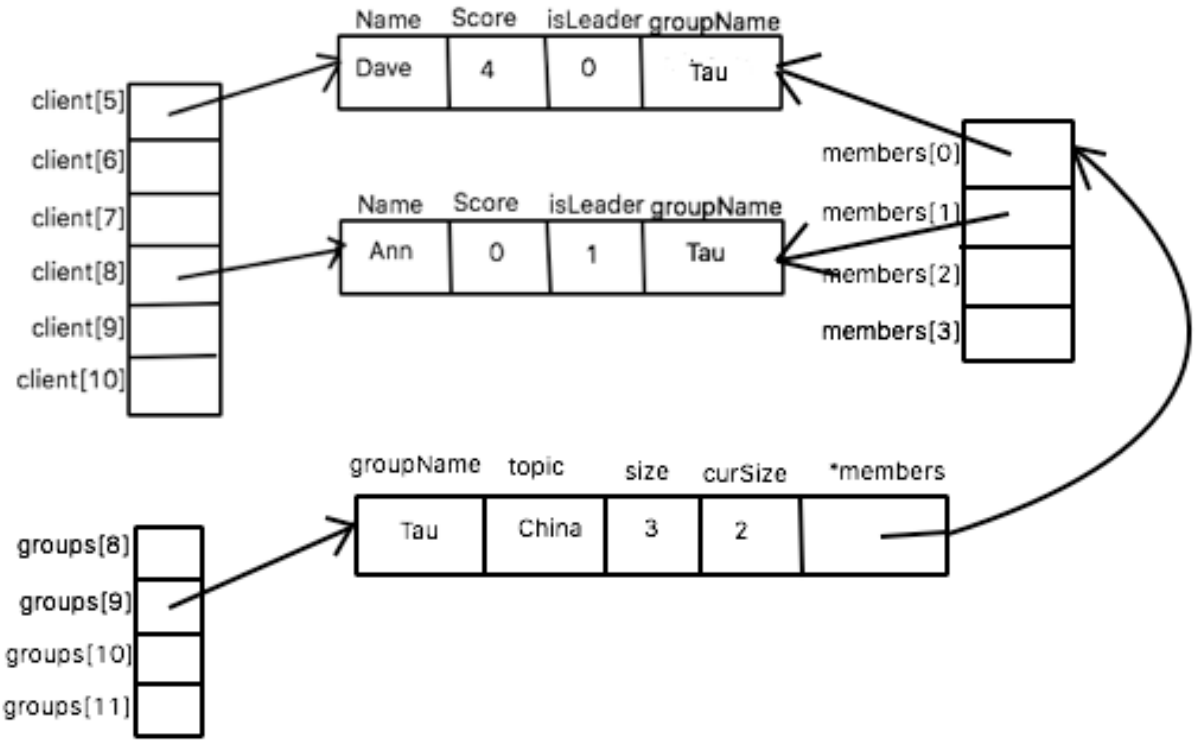

Figure 2. Array diagram for groups/clients

The similar diagram is shown in Figure 2 for groups[ ] array but the last attribute of the groups has client type and it is an array of pointers to the same memory location as from clients[ ] array.

These array diagrams did not change in overall just new fields are added.

Limitations

1) I allocated 32 spaces in memory for groups however for index in array socket number was used, which will work only for 29 groups and only if they will come in order. The last assumption is not always true, therefore it is better to create some ID for each group or to use instead of array linked list as a basic structure.

2) New members of the group were stored in the array `*members[1024]` and the index for the array was the current size of the group. However, this approach will result in gaps if some clients will leave the group. The simple solution would be just skip these members while iterating through the array of members.

Commands from client:

When the client connects to the server the new element in clients[ ] is allocated. Then the server is expecting the answer from the client. Handling of different commands was done within if-else blocks, comparing the first token in the client answer's string to existing commands. The list of implemented commands:

1) GROUP command.

Firstly the third token in the string will be checked to ensure that the groupname is not already in use. Here findByGroupname() function is called, which iterates through the array of groups and return -1 if the groupname is used by other group. If it not there, the client (leader) creates a new element in groups[ ] array, adds topic, groupname and groupsize. Current groupsize is set to 1. Thread is started to receive the quiz from the leader.

2) JOIN command

Client provides name, groupname to which it wants to join and score attribute is set to zero. At the same time a new client should be pointed by the element in members[ ] in that group. To find the group I call here function findByGroupname(), which will be searching from the groups[ ] the group with that name and return -1 if no matching group is found. BAD message is sent. Otherwise, it will return the socket number of the group leader. Before updating the current size and adding groupname attribute to the client. If the currSize == size then no other clients can be accepted and again BAD message is sent.

```
    int findByGroupname(char *grname){
    printf("Join??");
    fflush( stdout );
     for(int i = 0; i < 32; i++){
         if(groups[i] == NULL){ //skip not used socket
             continue;
         }
         if(strcmp(groups[i]->groupName, grname) == 0){
             printf("Found!\n");
             fflush( stdout );
             return i;
     }
}
     return -1;
}
```

As it was mentioned earlier when the current size of the group the client just joined achieves group's maximum size, new thread is started to play a quiz in that group preliminary removing corresponding file descriptors from the set.

3) GETOPENGROUPS command

```
void listGroups(int lssock);
```

The list of all open groups is send to that socket iterating through groups[ ] array.

4) LEAVE command

```
void leaveGroup(int csock);
```

When the above function is called groupname and score attributes of the client are cleaned Current size will be decremented. To find the member findByGroupname() is again called. Before cleaning operations I checked if the client is leader using isLeader flag, since as stated in the specifications LEAVE command is not allowed for the group creator.
In my implementation this message cannot be sent during the quiz.

5) CANCEL command

If the quiz has not yet started group[fd] will be cleaned where fd is a socket of the leader. Server responses with OK only when it is the group leader sending this command and if the currentSize of the group is not equal to the size of the group.

6) QUIZ command

Each sending quiz operation is handled inside the thread. When the message SENDQUIZ is sent each thread expects that the client will start to send the quiz probably part by part. Therefore client's response will be read in the loop until the quiz size will be reached. Before the text of the quiz its size sent in the message QUIZ|size|text.....

I will have a variable remSize, which is the quiz's size minus the length of the part of the quiz text sent by the client for the first time.

```
int remSize;

        if(strcmp(splitted[2], NULL) == 0){
            remSize = quizSize - strlen(splitted[2]);
            printf("%s\n", splitted[2]);
            fwrite(splitted[2] , 1 , strlen(splitted[2]) , fp );
        }else{
            remSize = quizSize;
        }
```

So while remSize is greater than zero server will read from client and each time write the client's response to the temporary file. The name of temporary file is made of string concatenated with the leader' socket number to make the file unique for each thread.

```
      sprintf(filename, "myquiz%d.txt", leaderSock);
```

Afterwards as it was done in the previous homework by fgets the content of the temporary file was written into array of lines, which is one of the fields of the leader. The number of questions also was calculated relying on the format of the quiz and the field numberOfQuestions of the group also was updated. Temporary file is removed from the directory before thread exists and leader's socket is returned to the file descriptors set.

```
if (remove(filename) == 0)
    printf("Deleted successfully");
else
    printf("Unable to delete the file");
pthread_mutex_lock( &mutex );
FD_SET(leaderSock, &afds );
pthread_mutex_unlock( &mutex );

    pthread_exit( NULL ); //exit thread
```

Sending the quiz

Concurrency inside the quiz was provided this time by using multiplexing, i.e. the new set of descriptors to hold the participating clients was created and the timer value was set to 1 minute.

Questions are sent one by one. This is how it is done:

a) Elements of the *lines[] are read until some empty element which marks the end of the question-answer block. String q is created to store one block;
b) After that empty line answerID for that question is stored in answ string, which is needed later to determine the winner of the question.
c) Question-answer block is written to the sockets of all members and their sockets are added to the local file descriptors set.
d) Using memcopy copy of the file descriptors set was populated with member's socket. After the select call I went through all ready descriptors reading their answers and adjusting scores comparing the answerID of each client with the correct answer. The winner message is sent to all clients and leader.
e) The above procedure is repeated numberOfQuestions time.

To make a synchronized quiz was a problem that I faced because each time when one of the clients answered server immediately proceeded to the next question without waiting other clients' responses. This issue was solved by wrapping the whole multiplexing procedure into while loop which forces the server to call select over and over again until answerCount is not equal to current group size. However, another problem occurred since the next question should be given if timeout occurs. To break the loop in the case of timeout setCount variable was created firstly initialized to 1 to enter the loop.

```
int setCount = 1;
```

Then in the body of while loop after select its value each time is set to 0 and I go through all members and if someone in the ready set I will increment the value of setCount.

```
setCount = 0;
        for(int i = 0; i < groups[leaderSock]->currSize; i++){
            int fd = groups[leaderSock]->members[i]->mysock;
            if(FD_ISSET(fd, &trfds)){
                setCount++;
            }
        printf("Set count: %d\n\n", setCount);
        fflush(stdout);

        if(setCount == 0){
            break;
        }
```

While loop will happen until the second condition is satisfied, i.e. as long as setCount is greater than 1. If after timeout nobody in the ready set the loop will be broken and server proceeds to the next question.

Action on timeout

According to specifications if some player does not answer within 1 minute, it should be kicked out of the group. This part was not implemented and those clients just allowed answering to the next question. I tried to implement this functionality by adding this for loop to check who is in the ready set however this cause leaving of the client who answers the question:

```
            for(int i = 0; i < groups[leaderSock]->currSize; i++){
                int fd = groups[leaderSock]->members[i]->mysock;
                if(FD_ISSET(fd, &trfds) == 0 ){
```

```
                printf("You are kicked out!");
                fflush(stdout);

                pthread_mutex_lock( &mutex );
                leaveGroup(fd);
                pthread_mutex_unlock( &mutex );

                FD_CLR(fd, &tafds );
                pthread_mutex_lock( &mutex );
                FD_SET(fd, &afds);
                pthread_mutex_unlock( &mutex );
                }
        }
```

At the end of the quiz RESULT message is sent to all members and the leader.

Messages sent by the server:

7) ENDGROUP message

This message is sent by the server to the clients in the group if leader sends CANCEL message or when the quiz has ended. Group[ ] item is cleaned and as well as groupname and score attribute for all clients who were in this group. Pointer is to that group is freed. Most importantly all members' and leader's file descriptors have to be put back to the main FD_SET so they again can create or join groups.

8) OPENGROUPS message

This command is just reply for GETOPENGROUPS command and is sent immediately when the new client is created by calling listGroups function.

Note:

In all places where either main thread or other threads try to access global variables the critical section was protected with mutex locks.

Overall limitations:

a. LEAVE functionality was not implemented for the clients who did not answer within one minute.
b. The program was not tested with large amount of clients and the large amount groups taking quiz simultaneously.
c. The program was not tested for the cases when clients leave the program during the quiz.
d. Multiple creating and joining also was not tested.