Lab Assignment 3. Inter-process Communication

You have to change the current functionality of your "coordinator.c" program such that to transform it in a server providing SF-file services to more users (clients) simultaneously. Consequently, you have to make the following changes in your program:

1. (5 min) After reading and validating its CONFIG_FILE, create a pipe named CONNECTION_REQUEST_PIPE.
2. (40 min) The authenticate() function will be called in an infinite loop, but instead of reading the username from STDIN, the CONNECTION_REQUEST_PIPE is used.In order to be able to differentiate between clients and to know how much bytes a username contains, the request messages sent by the clients to the server must comply the following structure:

```
struct connection_request_msg {
   uint32_t client_pid;     // used to differentiate between clients; could be used to establish unique names of client pipes
   char username[MAX_USERNAME_SIZE];
   char client_response_pipe[MAX_PIPE_NAME];
       char client_request_pipe[MAX_PIPE_NAME];
};
```

3. (15 min) Any message the "coordinator.c" previously displayed on STDOUT must now be written on each client's response pipe, whose name is taken from the request message. We refer to such a pipe in the followings as CLIENT_RESPONSE_PIPE. The CLIENT_RESPONSE_PIPE should normally be created by the client before sending the request. For instance, if the authentication fails, the error message must be written back to the client on its CLIENT_RESPONSE_PIPE.
4. (60 min) If the authentication succeeds, the message "SUCCESS: Authentication succeeded!" must be written back to the client on its CLIENT_RESPONSE_PIPE. Then, the coordinator must create a new process to deal with the next requests, i.e. commands, sent by the connected client. That process will execute the loop that reads a command line and handles the corresponding command on the user's SF-format files until EXIT command is given. Though the command line must not be read anymore from the STDIN but from another client specific pipe, called CLIENT_REQUEST_PIPE. Such a pipe must also be (like CLIENT_RESPONSE_PIPE) unique for each different client and should normally be created by the client before sending the connection request.
5. In order to control the communication between a client and the corresponding process at the server site, the messages exchanged on the CLIENT_REQUEST_PIPE and CLIENT_RESPONSE_PIPE should comply the following structures, respectively:

```
struct client_request_msg {
   uint32_t cmd_line_size;
   uint8_t  cmd_line[1];
};
struct client_response_msg {
   uint32_t response_size;
   uint8_t  response[1];
};
```

   The idea is that every message consists in just a stream of bytes (normally a string of characters) preceded by a 4-byte integer indicating the size of the following byte stream. The process sending a message must calculate the size of the byte stream, write it on the pipe and then write the corresponding stream. Similarly, the process receiving messages from a pipe must first read a 4-byte integer and after that the number of bytes indicated by the read integer. **NOTE:** you are free, if you want and consider more appropriate for your application, to use another message structure like, NULL-terminated strings

of characters. However, if you decide on such an alternative, you must describe it clearly as comments in your program.

6. (30 min) In order to avoid race conditions on the code executed by processes that handle the clients on the server site, a new connecting client is not allowed to authenticate with a username already authenticated for another active client. In order to manage that, your server must keep track of currently authenticated clients. The data structure used for this must be protected against concurrent accesses and changes by a global semaphore acting as a lock.

Besides changes that must be performed on the "coordinator.c" program, you must also implement a "client.c" program to contain the code executed by a client. The "client.c" must contain code to satisfy the following functionality:

1. (5 min) Creation of CLIENT_REQUEST_PIPE and CLIENT_RESPONSE_PIPE named pipes.
2. (10 min) Reading the username for authentication from STDIN and sending of the connection request on the "CONNECTION_REQUEST_PIPE".
3. (5 min) Getting back the authentication response from server on the CLIENT_RESPONSE_PIPE and displaying it on the screen for the client user.
4. (30 min) Reading commands from the command line, sending them on the CLIENT_REQUEST_PIPE, getting responses from the CLIENT_RESPONSE_PIPE and displaying them on the screen.

IMPORTANT NOTES

1. **YOU MUST UPLOAD BOTH your (UPDATED) "coordinator.c" file and the NEW client.c file as TWO SEPARATE FILES, NOT AS AN ARCHIVE!**
2. It is supposed to use your solutions to the previous lab assignments. If you did not provide solutions for the previous assignments or those you provided did not covered all required functionality, you can call functions that simply return a message consisting in their name and supposed functionality.
3. **Your programs must successfully compile with a command line like "gcc -Wall -Werror -lpthread coordinator.c -o server"** and **"gcc -Wall -Werror client.c -o client", respectively to be further evaluated.** The "-Wall" option says the compiler to report anything that could be wrong from its point of view, while the "-Werror" says it to report any warning as an error and do not accept it. As a consequence, your program must compile without any error and warning at all in order to be admitted for evaluation.
4. As usually, your programs must check the results of ALL functions used in relation to pipes (e.g. mkfifo, open, read, write) to be sure they were executed successfully.
5. You can automate your client programs' execution by redirecting their STDIN to text files prepared to contain the commands normally given from the keyboard. This way you can easily simulate concurrent clients. This is actually the way we will test your solutions.
6. Extra credit (3 points) will be given to those of you implementing network (socket) communication instead of pipes.