1 Introduction

In this homework, we implement a remote login server with similarities to sshd as well as to internet relay chat. The server listens to TCP port 8080, and accepts several types of commands from the user. In addition to introducing socket programming and threads, the command language and its constraints introduces a somewhat elaborate set of states and transitions to be modeled.

1.1 /name *word*

Sets the public name of the user to *word*, and responds appropriately.

```
/name jakob
Name set to "jakob".
```

If the user changes their name while they are subscribed to a channel, alert the other users to this fact.

```
User "jakob" changed their name to "eriksson".
```

1.2 /join *word*

Sets *word* as the channel that the user is subscribed to, responds appropriately, including the number of members in the channel. Also, alerts other members to the new arrival. For example,

```
/join uic
Channel set to "uic". There are 857 other members.
```

and

```
** "jakob" has joined the channel
```

Users may not join a channel without first setting their name. If they try, refuse and respond appropriately. Users may not join a channel if they are already subscribed to a channel. If they try, refuse and respond appropriately.

1.3 /leave

Leaves the channel.

```
/leave
Channel set to "uic". There are 857 other members.
```

and separately

```
** "jakob" has left the channel
```

1.4 /execute *command...*

Executes the command provided, including any arguments provided and prints the output of the command. The first word is the executable, the rest are arguments: no redirection operators or other shell functionalty expected. If the command fails, alert the user to this fact.

The user terminal waits until command has finished, and forwards any user input to the command, and any application output to the user. For example when running `cat`, any input from the user is printed back as soon as the user hits enter.

If the command "/break" is issued by the user during the execution of a command, the execution is stopped, using Child::kill(). The server continues running as before, and /break has no effect except while running an /execute command.

For this part, you will need to simultaneously read input from the command and listen for input, and /break on the socket from the user. Since we do not allow any crate dependencies, note that rust does not support nonblocking reads from the command output. Work around this by having a separate thread read the command output, and using a channel to send the output to the main thread. Channels can be read in a nonblocking fashion with try_recv(). In my solution, I use the same design for reading from the socket.

Ensure that other users are able to continue using their terminal uninterrupted while one user is executing a command.

1.5 /exit

Closes the connection.

1.6 *any other input*

Input not preceded by `/` is considered a chat message, and relayed to all users that have joined the same channel as the user, preceded by the user's name and the ">" character and a space. For example,

```
jakob> hi, is anyone there?
```

Nothing except chat messages are relayed on the chat channel.

```
/execute sleep 10
/execute ls
```

here, if the user enters the `ls` command before the sleep has concluded, we would still expect it to run (after the sleep). In the below example,

```
/execute sleep 10
/break
/execute ls
```

if /break is issued before sleep has concluded, we would expect the sleep to be terminated, then `ls` to run. Finally, if /break is pressed after sleep has finished, everything `ls` would still run. No commands other than /break should have any effect during the execution of a program. Make sure stdin still works correctly.

1.7 Suggestions

Key to correct behavior is that the server spawns a separate thread for each received connection, such that all users may interact with the server in parallel. Don't try to use any global variables - the lack of lazy_static or other dependencies will anyway make this extremely difficult. Instead, capture any state a new thread will need when it is spawned. std::sync::Arc will be your friend.

Watch out that you don't hold any locks (MutexGuard) while running commands - this would prevent other users from acquiring the lock while the program is running.

The set of possible states and inputs is fairly large. Think carefully about how you want to model this. My solution uses a single match statement over a tuple, where the first element of the tuple is the command. This made the code quite readable.

1.8 Grading criteria

For this assignment, we will use no external dependencies. We will not depend on *lazy_static*. The program must never panic - be careful to check your errors, and don't use unwrap() except where you are sure it is safe. Grading will be done by connecting to your server with multiple clients, issuing commands and monitoring the output.

1.9 Double-check your submission

Before turning in your homework, verify that your program builds and runs correctly after downloading the submitted version, and that it works the way you expect it to.

1.10 turn-in instructions

In your Cargo.toml, please include your name and UIC email address in the authors field, to identify your submission. Submission is via github classroom. Use this link to create your turn-in repository `https://classroom.github.com/a/N3rf4ciG`.