Project 1:

Com S 352

REMOTE SHELL USING SOCKET PROGRAMMING

100 points

Due: Friday, October 12, 2018, 11:59pm

This project consists of writing a C program for a remote shell system. It involves a Server and a Client process. The server process resides on another machine on the network and the user provides commands to the server via a client process on a local machine. The client sends the commands to the server, the server executes them, and sends the output back to the client.

The goal of this project is to practice some system calls and process management by the operating system. In addition, it reinforces the operating system support for client server programming using sockets. You will write a basic remote shell in which a user specifies an arbitrary shell command to be executed, and it is sent over socket connection and executed on a remote server.

(Note: use connection-oriented communication (**socket programming using TCP**))

You will write a server and a client program:

Server: The server will run on the remote machine.

- It will bind to a **TCP** socket at a port known to the client and waits for a Connection Request from Client.
- When it receives a connection, it forks a child process to handle this connection. **The Server must handle multiple clients at a time**.
- The parent process loops back to wait for more connections.
- The child process executes the given shell command (received from the client), returning all **stdout** and **stderr** to the client. (Hence, the server will **not** display the output of the executed command)
- The server can assume that the shell command does not use **stdin**.

Client: The client will run on the local machine.

- From the command line, the user will specify the host (where the server resides) and the command to be executed.
- The client will then connect to the server via a TCP socket.
- The Client sends the command to the server.
- The client will display any output received from the server to the **stdout**.
- After displaying the output, the client waits for next command from the user.
- The client will **not** close/exit until the user enters “quit” command.

Creating a Child Process

At Server side, for each incoming client, a child process is forked which executes the command received from the client.

The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family.

Note:(The example below is one way of doing this, you are welcome to do it your own way!)

This will require parsing the command into separate tokens and storing the tokens in an array of character strings, say **args**. For example, if the command entered at the prompt is:

ps -ael

the values stored in the args array are:

args[0] = "ps"

args[1] = "-ael"

args[2] = NULL

This args array will be passed to the execvp() function, which has the following prototype:

execvp(char *command, char *params[]);

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args).

Remember: In this project, the command is entered on Client!

Submission

- You should use C to develop the code.
- You need to turn in electronically by submitting a zip file named: Firstname_Lastname_Project1.zip.
- **Source code must include proper documentation to receive full credit (you will lose 10% of your score, if the code is not well documented).**
- All projects require the use of a **make file** or a certain script file (accompanying with a **readme file** to specify how to use the script/make file to compile), such that the grader will be able to compile/build your executable by simply typing "make" or some simple command that you specify in your readme file.
- **Source code must compile and run correctly on the department machine "pyrite", which will be used by the TA for grading. If your program compiles, but does not run correctly on pyrite, you will lose 15% of your score. If your program doesn't compile at all on pyrite, you will lose 50% of your score (only for this issue/error, points will be deducted separately for other errors, if any).**
- You are responsible for thoroughly testing and debugging your code. The TA may try to break your code by subjecting it to bizarre test cases.
- You can have multiple submissions, but the TA will grade only the last one.

Start as early as possible!

****This page is only for Educational Purposes, no use in the project****

Some Details on Shell:

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt **osh>** and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

osh> cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c), and then create a separate child process that performs the command.

A C program that provides the general operations of a command-line shell is supplied in Figure 3. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as shouldrun equals 1; when the user enters "quit" at the prompt, your program will set shouldrun to 0 and terminate.

```
#include <stdio.h>
#include <unistd.h>
#define MAX LINE 80                    /* The maximum length command */
int main(void)
{ char *args[MAXLINE/2 + 1];                    /* command line arguments */
 int shouldrun = 1;                  /* flag to determine when to exit program */
while (shouldrun) {
        printf("osh>");
        fflush(stdout);
/**
After reading user input, the steps are:
(1) fork a child process using fork()
(2) the child process will invoke execvp()
*/
} return 0; }
```

Figure 3 Outline of simple shell.