CSE3100 - Introduction to System Programming
Assignment #6, Due: 11/01/17

The purpose of this assignment is to discover how to author a client-server applications that communicate over sockets (network connections). To make this interesting, we are going to replicate the venerable `rsh` command that was available on the original UNIX systems. `rsh` stands for *remote shell*. Namely, it is a pair of executable acting as a server (to create new shells) and a client that is used to access the server hosted on a remote system. For simplicity's sake, we are going to run the client and the server on the same machine, but your code should work just as well on a remote server.

We are working with network code. It is quite easy to reach out to the machines of your peers or to create vulnerabilities on servers by running code that does not use authentication. There is a reason why `rsh` is no longer favored as a good solution for remote shell (these days we all use `ssh`). So while this is a good learning opportunity, the short software pieces you will produce *are not production ready and you should not deploy them on real systems!* You would have to produce a feature-full version with strong authentication to do so. The focus is on understanding the mechanics of client-server communications over sockets. Not the virtues of good authentication. That's for a cryptography or network security class! [Highly recommended!].

There are clearly two parts to complete: the server and the client. Good luck!

Exercise 1. ***Server (50 points)*** *Your server should be accessible on* `localhost` *and port* `8025`*. You should use, for your implementation, the IPv4 stack and the TCP (byte stream) transport layer. Your server should listen for inbound request to talk and, upon reception of such a request, spawn a child process responsible for running the* `bash` *shell (this is your default shell and is available under* `/usr/bin/bash`*). Recall that sockets are full-duplex and bi-directional. A shell script should be started for an interactive session (option* `-i` *for* `bash`*) and it should be a login shell (option* `-l` *for* `bash`*). Note that a shell uses all three standard stream* `STDIN,STDOUT,STDERR` *and that* `STDERR` *is used to print the prompt.*
Clearly, the spawned child process should take care of the necessary re-directions for `bash` *to work properly. Finally, the parent process (the* `rshd` *daemon) should collect the dead processes regularly to fully release resources back to the OS).*

Note that the remote shell daemon you are creating here is not meant to provide more advanced capabilities like authentication or terminal emulation. It is only suitable for simple line oriented commands such as those uses in the pipeline of the previous homework.

Your server should be named `rshd`*. You can execute it from a shell at the command line and it will* listen *for inbound client requests. If implemented correctly, your daemon should be inter-operable with the* `rsh` *client of any of your peers!*

Exercise 2. ***Client (50 points)*** *Your client should take as input the name of the host you wish to contact. Clearly, for testing purposes, you would invoke the client with the argument* `localhost` *to contact the server you have running in another terminal on your UITS VM. Your client should be called* `rsh` *and take the name of the target host as an argument on the command line. The client is pretty straightforward. It should setup a socket to contact the server on port* 8025 *and its task should be to obtain from the standard inputs the commands to be forwarded to the remote shell and display on its standard output the messages coming back from the remote shell. The sole delicate point lies in how to choreograph the two activities (reading from the standard input and reading from the socket representing the remote shell). A working implementation should be able to show you the prompt sent by the remote shell and submit simple commands such as* `ls, cd, pwd, cat, echo , ...` *to the remote shell and display the results on its local standard output. The entire solution (server and client) can be done in 140 lines of code total! This time around, you receive an empty folder and you must produce* everything *(*`Makefile` *and source code for the two programs). Make sure you adhere to the naming directives in this document so that the automated grading can build and use your executables!*

Enjoy!