COMP 530 INTRODUCTION TO OPERATING SYSTEMS

Fall 2017
Kevin Jeffay

Homework 5, October 18

Due: 11:15 AM, November 1

Building a Simple Distributed Shell

For this assignment you will extend your program from Homework 2 (the simple shell) to work as a distributed client-server program. You'll build a new "distributed shell" that will allow you to execute shell commands on a remote machine and have the results of the remote execution (the output of the remotely executed command) displayed on the local machine. To do this, you will write two programs: a "shell client" program and a "shell server" program. Functionally, your client and server programs should interact so as to be equivalent to a correct HW2 program except for the fact that the commands input to your shell execute on a remote machine. (Note: if your program for HW2 had errors, you should fix them before working on this assignment.)

Your client program will take two command line arguments. Argument one is the DNS name of the host system where your server program will be running (*e.g.*, *classroom.cs.unc.edu*). Argument two is the number of the "welcoming" port at which your server program will be accepting incoming socket connections (*e.g.*, 6789). Your server program will take one command line argument — the number of the "welcoming" port at which your server program will be accepting incoming socket connections (*e.g.*, 6789). (Note: to help avoid conflicting requests for the same port from other COMP 530 programs, your server can use 10000 + the integer represented by the last 4 digits of your UNC PID as its port number.)

Your client program will:

- Establish a socket connection to your server program.
- Output a prompt to *stdout*.
- Read a line of input from *stdin* (a line here is a series of characters terminated by a newline). The line of input is the command line to be executed on the server host system.
- Send the input line (including the newline) to the server program using the connected socket.
- Receive the lines of output generated to *stdout* on the server host system from the execution of the command. These lines are sent from the server program using the connected socket. (A line here is a series of characters terminated by a newline.) The client will output the received lines to *stdout*.
- After all the *stdout* lines have been sent by the server, the server will send a response line that indicates both the end of output generated from the remote command execution and the command terminating status. In the case of any error detected by your server program, the response line, or the client's summary of the response line should be printed to *stdout*. (That is, the client doesn't have to print the literal response line — it can print some more readable version of it.) Note that if the error detected by the server was one that prevented the execution of the command on the server, then the client will receive only a single response line generated by the server.

  If the response line indicates that no errors occurred on the server, the client should only print the command output sent from the server and not print the response line.
- After receiving all the command *stdout* lines and/or the server response line, your client should output a new prompt and continue to read input lines from *stdin*, send them to the server, and receive the next

round of output and response line. This should be repeated until end-of-file is reached on *stdin*. Once end-of-file is reached the client program closes the connected socket and terminates.

Your server program will:

- Accept a connection from your client program.
- Dynamically redirect *stdout* to a temporary file created so the program executed in the command will have its *stdout* go into a file. The system call *freopen*() is to be used to redirect *stdout* to this file. The temporary file should be given a name of the form "tmpxxxxx" where xxxxx is the process id of the server process. Use the system call *getpid*() to get the process id value and the library function *sprintf*() to format it as a string that can be concatenated to the string "tmp."
- Receive a line of input (the "command line") from the client using the connected socket (a line here is a series of characters terminated by a newline).
- Fork a new process.
- The parent process will wait for the termination of the child process.
- The child process will parse the line of input (the "command line") into a set of words ("arguments"). Words are defined as sequences of printable characters separated by whitespace.
- The child process will interpret the first word in the set (the first argument) as a filename and will *exec* the file, passing the remaining arguments (if any) to *exec* as the "argv" parameter (the "arguments" parameter) for the program being *exec*'ed.
- After the child terminates, the parent opens the temporary file that holds the redirected *stdout* and reads the lines (terminated by newlines) until EOF. For each line read before EOF, the server sends the line (including the newline character) to the client program using the connected socket. The system calls *fopen*() and *read*() can be used for file I/O.
- The parent always also sends a response line to the client program using the connected socket. The response line should indicate that the end of redirected *stdout* has been reached and give the status returned by the termination of the child process (see WIFEXITED, and WEXITSTATUS macros used with the system call *waitpid*()).
- Any error conditions generated by *fork*() or *execvp*() (or any other systems call made in the server) should result in the generation of an appropriate response line sent to the client using the connected socket. Note that there is always one response line (either one for the end of redirected *stdout* and child completion status, or one for a system call error) sent to the client for each input command line. The specific format of the response line should be chosen to be informative and (with high probability) not generated as a line to *stdout* by the executed program.
- After sending all the output required from executing the input command, your server should continue to read the next input line from the client using the connected socket and fork a child process to handle it. This should be repeated until end-of-file is reached on the socket it is reading. Once end-of-file is reached the server program terminates. Before terminating it should delete the temporary file used for *stdout* redirection using the system call *remove*().
- Note that the processing done by the child process of the server is exactly the same as for Homework 2. The major change in the parent process of the server is to handle the sending of redirected *stdout* and a response line to the client using socket communications instead of *stdin* and *stdout*.

Details

Write two C programs on Linux that perform the above client and server functions, respectively. For socket operations you will be using the simple socket "object"/abstract data type to be described in a video posted on the class website. The implementation of this ADT is in an object module named *libsocket.o* and an include file named *Socket.h*. These files can be obtained from the course web page. There will also be example programs using all the system calls needed for this assignment on the course web page. Be sure to test your programs by

running them on *different* host systems. That is, when testing your program, start a shell on one machine (*e.g.*, *classroom.cs.unc.edu*) and execute your server program. Start a shell on a second machine (*e.g.*, *snapper.cs.unc.edu*) and execute your client program. Be sure to watch out for zombie processes you accidently create on either machine!

Extra Credit

Extend your server program to use the "daemon-service" model described in the video for this assignment. In this model, multiple clients can be serviced concurrently by the service. To implement this, the server main process is a simple loop that accepts incoming socket connections, and for each new connection established, uses *fork*() to create a child process that is a new instance of the server process as described above. This child (server) process will handle the socket input and output for the single client program that established that socket connection. Each new instance of the server process will execute your code for all the steps described above for the server program (including using *fork*() to create its own child processes to parse the input and invoke *execvp*()), except it will not accept any incoming socket connections. It uses its copy of the connected socket created when the daemon process accepts an incoming connection. There are two somewhat subtle points concerning the daemon process: (1) since there is no defined stopping criterion, it must be terminated by some action such explicitly killing it (kill -9), and (2) because the child processes it creates may terminate at arbitrary times and because the daemon should block only to wait for incoming connections, it should use a non-blocking version of the *waitpid*() system call after each fork to clean up any terminated "zombie" processes. An example of how this is done is included in the example programs on the course web page.

Bonus Credit

If you did not complete the first two extra credit portions of Homework 2, here is a second chance. Implement them as part of the processing in your server's child process. They are restated below.

For extra credit, have your child process test to see whether or not the file name to passed to *execvp*() is a valid filename before calling it.

A more challenging extension is to use the *execv*() system call instead of *execvp*(). The primary difference here is that you will have to read the shell environment variable PATH and search the directories in the PATH to determine the appropriate directory where the command is stored. (Read the man pages for *stat*() and *getenv*())

Grading

Place copies of your C program(s) in your *~/submissions/HW5* directory. Name your client program HW5client.c and your server HW5server.c. As always, after you submit your homework, *do not modify your files*. If you want to keep working on the assignment, do so on a copy. Send mail to *jacobdf@cs.unc.edu* when your program is ready for grading. Your distributed program will be tested with the same command line inputs that were used for Homework 2.