CSC155/290

Homework #2
(Assigned: 9/13/2013)
(Due: 9/23/2013)

I. Overview

Implement a simple UNIX **shell.** The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

Your shell should run in interactive mode and display a prompt (any string of your choosing). The user of the shell will type in a command at the prompt.

To exit, your shell terminates when it sees the `exit` command on a line (i.e. A user types 'exit' to quit – the `exit()` system call will be useful here).

II. Program Specifications

Your shell is basically a loop: it repeatedly prints a prompt, parses the input, executes the command specified on that line of input, and waits for the command to finish, if it is in the foreground. This is repeated until the user types "exit" or ends their input.

Design your shell so that it creates a new process for each new command.

Some hints...

1. Parsing

For reading lines of input, use `fgets().` To open a file and get a handle with type `FILE *` use `fopen().` Be sure to check the return code of these routines for errors! (If you see an error, remember that the routine `perror()` is useful for displaying the problem.) You may find the `strtok()` routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or ...).

2. Executing Commands

Look into `fork(), execvp(),` and `wait/waitpid().` The `fork()` system call creates a new process. After this point, two processes will be executing within your code. You will be able to differentiate the child from the parent by looking at the return value of `fork;` the child sees a 0, the parent sees the `pid` of the child.

You will note that there are a variety of commands in the `exec` family; **for this project, you must use** `execvp().` Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

```
int main(int argc, char *argv[]);
```

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

```
foo 205 535
```

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

Important: the list of arguments must be terminated with a NULL pointer; that is, argv[3] = NULL. Carefully check that you are constructing this array correctly.

The `wait()/waitpid()` system calls allow the parent process to wait for its children. Read the man pages for more details.

III. Test, Test, Test!

Try to break your own code!

IV. GitHub (required)

Use GitHub during your development process.

V. What To Turn In to Blackboard

Once you have have your code fully tested and working, submit the following to Blackboard:

1. Makefile
2. .c source files and any .h header files
3. README (explaining what your code does, how to compile it and how to run it)