CS 3305: Operating Systems
Department of Computer Science
The University of Western Ontario
Programming Assignment 1
Winter 2010

Purpose

The goals of this assignment are the following:

- Get experience with the fork(), execlp(), pipe() and dup2() system calls
- Understand how a shell works
- Learn more about how operating systems are structured
- Gain more experience with the C programming language

Part I: Specification for Shell Program

In this part of the assignment you are to implement a basic shell. A shell is a command line interpreter that accepts input from the user and executes programs on behalf of the user based on the commands that the user inputs. The line that the user enters commands on is referred to as the *command line*. The shell repeatedly prints a prompt on the command line, waits for the user to enter commands and executes programs. You are to write a simple Unix-like shell in the C programming language that has these features:

- If your name is xyz then the prompt should be the string *xyz>*
- You may assume that the user enters commands correctly.
- Your shell must support pipes but does not need to support redirection or background processes. In other words, you should be able to handle commands with "|" but it is not necessary to handle commands that consist of the symbols "<", ">", or "&". Your shell does not need to support any special characters like "*" or "?". Thus you do not need to support commands such as "ls *.c". Your shell must support any number of command line arguments, for example the user could type "ls -al foo.c blah.h".
- Your shell should look for commands in the */bin* directory.
- A *built-in* command is a command that changes the state of the shell or requests information about the shell's state. You should support the following two built-in commands:
    - quit: This command is used to terminate the shell.
    - history: This command is used to print a list of the last 10 commands issued; If there are fewer then 10 commands then print out all previously issued commands.
- Your shell should look for commands (except for built-in commands) in the */bin* directory. In the case of an error you should print an error message and quit.
- Your shell must support multiple pipes. However, you may assume that you have a constant defined that specifies the maximum number of commands allowed.
- Your code must work in Minix.

Part II: Hints

1. You will need to make use of system calls such as *execvp()*, *fork()*, *pipe()*, *wait()* and *dup2()*. The functionality of these system calls is similar for MINIX and Unix-based systems. The man pages are an excellent starting point. If you follow the Resources link in the CS 3305 web page, you will find links to several useful references. There are many more on the web. Example code is provided on the Assignments web page of the CS 3305 web page.
2. Assume the user enters "ls –lt | sort | more". Your shell would create a process for each command. A pipe must be created (using the pipe command) between the first and second process and the second and third process. Creating a pipeline between two processes is relatively simple (you have sample code for this), but the building of a multiple command pipeline is more difficult. An excellent discussion of the considerations needed for developing a multiple command pipeline can be found at: http://www.cse.ohio-state.edu/~mamrak/CIS762/pipes_lab_notes.html
3. You may use any of the system calls found in the *exec()* family. My own solution uses *execvp()*. The main challenge of calling *execvp()* is to build the argument list correctly. If you use *execvp*, remember that the first argument in the array is the name of the command itself, and the last argument must be a null pointer. There are several code examples that show how to construct the arrays needed for *execvp*.
4. A shell needs a command-line parser. The parser breaks down the string representing the command into constituent parts. These parts are referred to as tokens. To read a line from the user, you may use *gets()*. The parser you implement may use *scanf()*, *strtok()* or any other suitable C library functions. Parsing requires these steps:
    a. Read a line from standard input. You may use the *gets()* system call.
    b. A *token* is a sequence of non-whitespace characters that is separated from other tokens by whitespace characters. For each command line, you should form an array of tokens. This can be done using the *strtok()* system call. Example code is provided that takes a string and provides the tokens in an array.
    c. You should determine commands by analyzing the array created in the previous step. Any tokens between pipe signs are considered to be part of the same command.
    d. You may assume that there is whitespace before and after "!". Note that in most shells this is not necessary, but if you make this assumption parsing is easier.
5. Do not try the "big bang" approach to programming. Develop your code in incremental stages. One order of stages is the following:
    a. Write a shell that can fork single program invocations (with no arguments) and wait for their completion.
    b. Develop the command line parser.
    c. Extend the shell developed earlier to be able to handle arguments.
    d. Add support for pipes
6. Start early!