CPE 435: OPERATING SYSTEMS LABORATORY.

Lab03

Building a custom Linux shell.

Submitted by: Dan Otieno.

Date of Experiment: 01/27/23.

Report Deadline: 02/03/23.

Demonstration Deadline: 02/03/23.

Introduction

The goal for this lab was to understand how to build a Linux shell, applying the concepts of piping, forking and I/O redirection.

Theory

Users interact with an operating system using the command line. Commands are entered in the terminal or console as characters separated by a space, which are then handled and processed by the operating system. According to an article by Computer Hope, ["….While each variant has subtle differences, command-line interpreters all accept text via a keyboard input through a command-line interface using a command prompt. Once the command is entered, the CLI translates it into functions that the operating system understands. The operating system then returns output to the user or performs a task based on the information it receives."] - Source. The purpose of this lab was to write and implement a program that runs a custom shell, receives, and executes shell commands. A few theory topics to explore are explained below:

- Shells:
    - A shell is a program that acts between the user and the operating system. When a user enters commands at the command line prompt, in the form of keyboard characters, the shell takes those commands and sends them to the operating system for processing. According to Wikipedia, ["In computing, a shell is a computer program that exposes an operating system's services to a human user or other programs. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system."]

- Strtok():
    - This is a function defined in the c standard library that uses a delimiter to break a string variable into tokens. The strtok() functions takes two arguments, a string, and a delimiter, with char data type pointers to both parameters. When executed, the function returns a pointer to the first byte of the first token in the string, and a NULL pointer when there are no more tokens left.

- Dup() and dup2():
    - Dup() and dup2() are both system calls that make copies of (duplicate) file descriptors. Both the newly created duplicate and the original file can be used interchangeably. Dup() differs with dup2() in terms of the specific file descriptor used; while dup() will automatically take the next lowest numbered unused file descriptor, dup2() allows for user specification on which file descriptor to take for new.

- Pipe:

- The best definition of a pipe, found online, is ["A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '|'."] - Source. Piping allows for using the output of a command as the input of a subsequent command, as in the form of a "flow" of a pipe.

- execvp():
  - This is part of the exec() group of functions in an operating system. When called, they replace the image of the current process with a new process image. Execvp() also provides a pointers array that point to null-terminated strings representing a list of arguments for a new program. If an error has occurred, this function returns a value. The arguments for this function are a a string character containing the name of the file to be executed, and the second argument is a pointer to an array of strings, and when the function is called, the file given by the first argument will be loaded into the caller's address space and over-write the program there, while the second argument will get the program and start executing it.

Results & Observation

Assignment 1:

Description:

The goal for this assignment was to use C/C++ to write a program that executes a custom shell, accepts commands, and returns the expected outputs for those commands. The shell would execute a loop, prompts a user for entry, reads the characters entered, and then incorporates processes like piping, input/output redirection to execute the operating system processes and return expected outputs.

Program Output:

```
dpo0002prac@DESKTOP-140DFI9:/mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_3$ ./lab3
MyBash$: ls -a
 .                   lab03_demo1.c   lab03_demo3.c     lab03_demo5     lab3code.c     Study_Lab03.pptx
 ..                  lab03_demo2     lab03_demo3.exe   lab03_demo5.c   lab3_out.png   testlab
'Hi, I am an echo.'  lab03_demo2.c   lab03_demo4       Lab03.pdf       lab3test2.c    testlab.c
 lab03_demo1         lab03_demo3     lab03_demo4.c     lab3            labfn.txt      today
MyBash$: date > date.txt
MyBash$: cat date.txt
Fri Feb  3 03:06:16 PM CST 2023
MyBash$: ls | sort
date.txt
Hi, I am an echo.
lab03_demo1
lab03_demo1.c
lab03_demo2
lab03_demo2.c
lab03_demo3
lab03_demo3.c
lab03_demo3.exe
lab03_demo4
lab03_demo4.c
lab03_demo5
lab03_demo5.c
Lab03.pdf
lab3
lab3code.c
lab3_out.png
lab3test2.c
labfn.txt
Study_Lab03.pptx
testlab
testlab.c
today
MyBash$: who | wc -l
0
MyBash$: who | wc -l > a.txt
MyBash$: cat a.txt
0
MyBash$: exit
dpo0002prac@DESKTOP-140DFI9:/mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_3$
```

In conclusion, this was a great introduction to understanding how a shell executes commands. And the various concepts involved. I was able to better understand redirection, but I will need more information to grasp piping.

Appendix

Assignment 1 code.

```
/*
DAN OTIENO
CPE 434-01
LAB 3
Exercise 1
*/
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
```

```
#include <sys/wait.h>
#include <sys/stat.h>
#define L_MAX 100

int fWait = 1;
int fRun = 1;

void fnIn(char *myFileName)
{
        int fIn = open(myFileName, O_RDONLY);
        dup2(fIn, 0);
        close(fIn);
}

void fnOut(char *myFileName)
{
        int fOut = open(myFileName, O_WRONLY | O_TRUNC | O_CREAT, 0644);
        dup2(fOut, 1);
        close(fOut);
}
void fnRun(char *myArgs[])
{
        pid_t pid;

        if(strcmp(myArgs[0], "exit") != 0)
        {
                pid = fork();

                if(pid < 0)
                {
                        fprintf(stderr, "Child process failed");
                }
                else if(pid == 0)
                {
                        execvp(myArgs[0], myArgs);
                }
                else
                {
                        if(fWait)
                        {
                                waitpid(pid, NULL, 0);
                        }
                        else
                        {
                                fWait = 0;
                        }
                }
                fnIn("/dev/tty");
        fnOut("/dev/tty");
        }

        else
        {
                fRun = 0;
        }
}
void fnpipe(char *myArgs[])
```

```
{
        int fds[2];
        pipe(fds);

        if(pipe(fds) == -1)
        {
                printf("Error executing piping process");
                exit(0);
        }

        dup2(fds[1], 1);
        close(fds[1]);

        //printf("myArgs = %s\n", *myArgs);

        fnRun(myArgs);

        dup2(fds[0], 0);
        close(fds[0]);
}

char * fnToken(char *input)
{
    int i;
    int j = 0;

    char *token = (char *)malloc((L_MAX * 2) * sizeof(char));

    // add spaces around special characters
    for (i = 0; i < strlen(input); i++)
    {
        if (input[i] != '>' && input[i] != '<' && input[i] != '|')
        {
            token[j++] = input[i];
        }
        else
        {
            token[j++] = ' ';
            token[j++] = input[i];
            token[j++] = ' ';
        }
    }
    token[j++] = '\0';

    // add null to the end
    char *end;
    end = token + strlen(token) - 1;
    end--;
    *(end + 1) = '\0';

    return token;
}

int main(void)
{
        char *myArgs[L_MAX];
```

```
        while (fRun)
        {
                printf("MyBash$: ");
                fflush(stdout);
                char input[L_MAX];

                // Get user input
                fgets(input, L_MAX, stdin);

                char *ftoken; //Tokens.

        ftoken = fnToken(input);

        if(ftoken[strlen(ftoken) - 1] == '&')
        {
                fWait = 0;
                ftoken[strlen(ftoken) - 1] = '\0';
        }
         char *myTemp = strtok(ftoken," ");
         int counter = 0;

                while(myTemp)
                {
                        if(*myTemp == '<')
                        {
                                fnIn(strtok(NULL, " "));
                        }
                        else if(*myTemp == '>')
                        {
                                fnOut(strtok(NULL, " "));
                        }
                        else if(*myTemp == '|')
                        {
                                myArgs[counter] = NULL;
                                fnpipe(myArgs);
                                counter = 0;
                        }
                        else
                        {
                                myArgs[counter] = myTemp;
                                counter++;
                        }

                        myTemp = strtok(NULL, " ");
                }

                myArgs[counter] = NULL;

                fnRun(myArgs);
        }

        return 0;
}
```