

Table of Contents:

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Mission Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Survey of existing work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4 Proposed solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

Evaluation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Appendix (Code) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

**Introduction**

**Mission Statement:**

The implementation of our own Linux shell to replicate a particular set of the functionality of the Bourne Again Shell distributed with Linux and GNU operating systems.

**Problem Statement:**

The initial problems presented in our project were few, but grew to be more numerous and complex as time progressed. Our goal was to implement Linux BASH using our own C code; without the use of system(). Topics that seemed to give us some trouble was implementing the lexical analyzer/ tokenizer/ parser and multiple command manipulation. Our motivation for the project was to use our newly acquired skills in C programming and in Linux Operating Systems to demonstrate our understanding of the Linux Bourne Again Shell.

**Survey of Existing Work:**

There are a lot of existing implementations of creating your own Linux shell in C, all which vary in degree of programming sophistication and level of expertise. With our group consisting of programmers that are inexperienced in the C language, we wanted to not use code that was far above our level of knowledge. Very few of the examples we found on the web were able to implement memory protection, or piping which is necessary for the “*grep*” command in Linux. Others were able to implement compilation and execution of code without using high level code which at our current skill level would be impractical of us to attempt to implement.

Quite a few versions found online used method wrappers to be pseudo helper functions for other methods and used a fairly difficult amount of double pointers (that point to the first index of a memory address) to accomplish the same thing we did in a more elegant way than our implementations. The most difficult part of learning from existing examples is finding a way to implement the same commands differently than what was presented.

**Proposed solution:**

As stated above, our main obstacle was tokenizing the command line input strings. The steps we chose to follow in order to overcome this problem was initializing a buffer to hold the finished tokens and then dynamically allocating that buffer using “*malloc*”. The choice of having a dynamically allocated buffer came from the fact that we wanted to be able to accept an arbitrarily long input, specifically for long directories or for executing multiple commands. After the buffer was established we separated the input string of characters by collecting all strings in between any break, space, and/or end line. We defined the space boundary delimiters as “\t\r\n” and stored each resulting token in a temporary variable using “*strtok*” which breaks up a string into a series of tokens using the delimiters we defined. Finally we added each token to the dynamically allocated buffer, resulting in a tokenized array of character arrays.

Now that we have a tokenized string of characters from the command line, we need to give it semantics, or meaning behind each user created command. In order to link each command, we created methods to give each command an intended functionality. We essentially used a complex series of nested if statements and nested while loop to control the ability for the user to input multiple commands at the same time via separating them each with “|”. Each separated command will have its own independent process and for our purposes we included “|” as a command which creates a process itself to handle the next command to the right of it. Initially we wanted the parsing functionality to be in its own method, but in order to simplify the passing of arguments and pointers we opted to just include it in the main loop.

After we tackled tokenizing and parsing the string of characters in the format of our grammar, we were simply left with the creation of our functions. Our intent was to use a methodology as low level possible, so we are not relying on the “*system()*” command in C which would use the actual shells process instead of the user handled processes in our shell. We were able to implement the commands we intended to manipulate without the system() command except for ps.

**Implementation:**

To begin our project, we needed a way to tokenize all the incoming input from the command line and link them to our commands. Once our shell is running or implemented, the Bourne Again Linux shell would not be able to catch, or stop Our Shell from being able to catch the incoming commands. We also needed to parse the incoming commands and give them semantics that would build the grammar of our input standard for the shell we were creating.

Once the commands were tokenized and parsed in a way that we could manipulate them properly, we needed to determine the types of commands that we wanted to implement. Starting at the very basic of things like changing a directory(cd). For each type of command we dissected what the actual Linux shell commands wanted to implement via “*strace*” command in Linux and system calls. After that, we dealt with incoming signals with signal handlers, and formatted the output and structure of the code we created.

For signal handling we used C’s simple “*SIG\*\*\**” code to identify the signals being passed through the command line. To do this, we created a signal handler function along with signal specific methods to perform the command requested. All we had to do was install the signal handlers at the start of our main method with the “*signal()*” call. The two signals we decided to catch and handle are sigInt, for process termination, and sigQuit, for process termination and dump core.

All of our forking was controlled by the parsing process in order to make sure that there weren’t any processes floating around due to child creations for each command. It seemed more logical to us that after we have determined what a command was and to which method should implement that command, we would create a new process to perform that method call. This allowed us to circumvent the need of using the “*exec()*” family of system calls. We initially wanted to use exec() system call and actually tried to apply it in our program but we realized the command only worked with Linux Bash files so it would not execute with our implemented commands.

The commands we were able to fully implement were help, cd, ls, “|”(execute multiple commands), clear, and exit. We were also able to open a file in any directory.

The change directory command (cd) was implemented by using two linux functions; getenv(), and chdir(). To change a user’s directory we pass the path requested to the cd method, which then passes the path getenv() to retrieve the actual environment path with will be passed to chdir() which actually completes the directory change. We also added our own variation to cd by using the character ‘h’ to change directory to home instead of linux implementation of /home.

The list directory command(ls) was implemented by passing the current directory to the ls method which is passed to opendir() linux function which contains a list of all directories and files in a given directory. After opendir() we loop through all the files and print them to the screen using printf().

The pipe command is nothing similar to the linux version of piping. Our implementation of piping was concatenating multiple commands and executing them in the order received. This functionality was implemented by creating 2 child processes, one the first command and the other for the second command after the “|” symbol. This process also works by checking the token after the pipe symbol, compares it all the different commands, and calls the proper functions to execute. The pipe process continues until there are not more commands including “|”.

The clear command was implemented through the use of printf. Only prints ANSI escape codes. The method uses printf to print “\033[H\033[J”. The string represents some escape characters line feed and backspace. In short it states clear screen(“\033”), backspace(“[H”), to top/ linefeed(“[j”).

The exit command was implemented by using Linux kill() function which took the parentid and SIGKILL as parameters. This command exits program and exits the terminal.

The last functionality that we implemented was the open command, which mimics Linux cat command. This command takes as an input the filename you wish to open then uses fopen to open the file in readonly mode and prints each character in that file.

**Evaluation Results**

The only commands we were unable to implement using our own code was the “ps” and “grep” commands. There were a lot more involved in the ps command than we imagined and it required going through the proc files, determining which processes were running, and a lot more that we did not get to before the end of our project. We also sought to do a lot more with file manipulation such as writing to and editing in the terminal but we did not get too far. As for evaluating our results, our code functioned as if it was the given linux shell used in ubuntu. Every command was tested and compared to the output that we had expected. Overall, all of the implemented code except for what was stated above worked perfectly.

**Conclusion**

At the conclusion to our project, we have completed most of our goals with the exception of “ps” and “grep” however we have accomplished the majority of our program requirements we set for ourselves. By writing our own shell we have obtained a much stronger grasp of the many details and facets of manipulating processes, traversing a file system, and our level of understanding of the C programming language.

**References**

<http://www.cs.cornell.edu/Courses/cs414/2004su/homework/shell/shell.html>

<https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf>

<http://web2.clarkson.edu/class/cs444/labs/lab01/Writing_Your_Own_Shell.html>

<http://wiki.bash-hackers.org/scripting/terminalcodes>

**Appendix (code)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#define BUFFER\_SIZE 2048

#define DELIMS " \t\r\n"//Set of deliminators for the lexical anylizyer

int SIZE = 0;

char \*path;

//Help function that displays all commands

void help(void){

printf("HELP MENU: \n");

printf("1: cd - Change Directory := cd <path> or cd to return to home directory \n");

printf("2: grep - Search for directory or file. := grep <word to search for> <directory> \n");

printf("3: ps - Shows all current proccesses := ps \n");

printf("4: ls - Lists contents of the directory := ls \n");

printf("5: | - Pipes multiple commands together with independant processes := cd | ls \n");

printf("6: open - opens a given file := <fileName> <path>\n");

printf("7: clear - Clears the prompt := clear\n");

printf("8: exit - exits the shell := exit \n");

kill(getpid(), SIGTERM);

}

char \*lexical\_Anylyzer(char \*inLine){

int size = BUFFER\_SIZE;//Starting buffer size

int pos = 0;//Index for tokenList

char \*\*tokenList = malloc(size \* sizeof(char\*));//Allocates memory for a char list, \*\*tokenList is essentially a 2D char array

char \*token;//This will point to the the result of strtok

SIZE = 0;

if (!tokenList){//Memory allocation verification

perror("ERROR: ");

printf("\n");

exit(EXIT\_FAILURE);

}

token = strtok(inLine, DELIMS);//Breaks up the input according to the deliminators DELIMS

while(token != NULL){//traverses the list of char arrays

tokenList[pos] = strdup(token);//Deepcopies the string tokens into tokenList

pos++;

token = strtok(NULL, DELIMS);

SIZE++;

}

tokenList[pos] = NULL;

return tokenList;

}

//handles the inturrupt signal

void handler\_SIGINT(int signal) {

int status;

pid\_t pid;

pid = wait(&status);

}

void SIGhandler(int sig){

switch (sig) {

case SIGINT:

{

printf("PID: %d \n", getpid());

printf("SIGINT recieved\n");

break;

}

case SIGQUIT:

{

printf("PID: %d \n", getpid());

printf("SIGQUIT: Exiting Shell. GOODBYE\n");

exit(EXIT\_SUCCESS);

}

//default:

// other signal types

}

}

//Need to add memory protection to /proc/

//working on it now

void ps(void){

FILE \*fp;

DIR \*dirp;

struct dirent \*proc;

char \*line;

char allProc[BUFFER\_SIZE];

size\_t length = 0;

dirp = opendir("/proc/");

if(dirp == NULL){

perror("Failed To Open Process Directory.\n");

}else{

strcpy(allProc, "/proc/");

fp = fopen(allProc, "r");

if(fp != NULL){

getline(&line, &length,fp);

}

while ((proc = readdir(dirp) != NULL)){

printf("PID: %d \n", proc->d\_name);

}

fclose(fp);

}

kill(getpid(), SIGTERM);

}

void clear(void){

printf("\033[H\033[J");

}

//Lists the conetents of the current working directory

void ls(char \*path){

DIR \*dp;

struct dirent \*dirp;

dp=opendir(path);

while((dirp = readdir(dp))!= NULL){

printf("%s\n", dirp->d\_name);

}

closedir(dp);

kill(getpid(), SIGTERM);

}

// to open small file

void openFile(char \*path){

FILE \*fp;

char nextchar;

printf("in openfile\n: %s\n", path);

fp = fopen(path, "r"); // read only

if(fp == NULL){

perror("error");

exit(EXIT\_FAILURE);

}

do{

nextchar = fgetc(fp);// get next character

if(feof(fp)){

break;

}

putchar(nextchar);

//printf("%c",nextchar);

}

while(1);{

fclose(fp);

}

kill(getpid(), SIGTERM);

}

//Changes directory

void cd(char \*path){

printf("path : %s\n", path);

if (strcmp(path, "h") == 0)

{

char \*home = "/home";

chdir(home);

printf("Working in this directory: %s\n",get\_current\_dir\_name());

}

else {

chdir(path);

printf("Working in this directory: %s\n",get\_current\_dir\_name());

}

}

int main(int argc, char \*argv[]) {

char \*input = NULL;

char \*\*tokens;//Holds the tokenized string

char \*cmd;

pid\_t childPID;//pid for the child process

pid\_t wpid; //pid for the parent to wait for child

int i = 0;

int status;

int flag = 1;

//Finds the current working directory

if ((path = (char \*)malloc((size\_t)BUFFER\_SIZE)) != NULL){

getcwd(path, (size\_t) BUFFER\_SIZE);

printf("CWD: %s ",path);//Prints current working directory

printf("\n");

} else{

perror("ERROR: ");

printf("\n");

exit(EXIT\_FAILURE);

}

//These install the signal handlers and handles errors

if (signal(SIGINT, SIGhandler) == SIG\_ERR)

{

printf("\ncan't catch SIGINT\n");

}

if (signal(SIGQUIT, SIGhandler) == SIG\_ERR)

{

printf("\ncan't catch SIGQUIT\n");

}

while(flag == 1){

printf("OSH $");

//size\_t is an unsigned datatype that can hold ANY array index

ssize\_t sizeOfBuffer = 0;//Address of the variable that holds the size of the input buffer

//getLine() allocates memory to sizeOfBuffer automatically

if(getline(&input, &sizeOfBuffer, stdin) == -1){

printf("OSH $");

}//Gets input from user, stores it in buffer

tokens = lexical\_Anylyzer(input);

//int size = sizeof(tokens)/sizeof(char);//Size is the number of commands entered

for(i = 0; i < SIZE; i++){

//if statements to determine command tokens semantics

if(strcmp(tokens[i],"exit") == 0){

kill(getpid(), SIGKILL);

exit(1);

}else if(strcmp((tokens[i]),"help") == 0){

childPID = fork();

if (childPID == 0){

// Child process

help();

} else if (childPID < 0) {

// Error forking

perror("lsh");

}else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp((tokens[i]),"open") == 0){

childPID = fork();

if (childPID == 0){

// Child process

i++;

path = tokens[i];

openFile(path);

} else if (childPID < 0) {

// Error forking

perror("lsh");

} else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp((tokens[i]),"cd") == 0){

i++;

path = tokens[i];

cd(path);

} else if(strcmp(tokens[i],"ls") == 0){

childPID = fork();

if (childPID == 0){

// Child process

path = get\_current\_dir\_name();

ls(path);

} else if (childPID < 0) {

// Error forking

perror("lsh");

} else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp(tokens[i],"ps") == 0){

childPID = fork();

if (childPID == 0){

printf("ps");

// Child process

//system("ps");

ps();

} else if (childPID < 0) {

// Error forking

perror("lsh");

} else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp(tokens[i],"clear") == 0){

childPID = fork();

if (childPID == 0){

// Child process

clear();

} else if (childPID < 0) {

// Error forking

perror("lsh");

} else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if (strcmp(tokens[i],"|") == 0){

childPID = fork();

pid\_t childPID2, wpid2;

if (childPID == 0) { i++;

// Child process

//Determines which command is used with for the second new process if(strcmp((tokens[i]),"cd") == 0){

i++; path = tokens[i];

cd(path);

}else if(strcmp((tokens[i]),"help") == 0){ childPID2 = fork();

if (childPID2 == 0) {

help(); } else if (childPID2 < 0) {

// Error forking

perror("lsh");

} else {

// Parent process waits for child to finish and returns if child has stopped

do {

wpid2 = waitpid(childPID2, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp((tokens[i]),"ls") == 0){

childPID2 = fork();

if (childPID2 == 0) {

// Child process

path = get\_current\_dir\_name();

ls(path);

} else if (childPID2 < 0) {

// Error forking

perror("lsh");

} else {

// Parent process waits for child to finish and returns if child has stopped

do {

wpid2 = waitpid(childPID2, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp(tokens[i],"ps") == 0){

childPID2 = fork();

if (childPID2 == 0) {

ps();

} else if (childPID2 < 0) {

// Error forking

perror("lsh");

} else {

// Parent process waits for child to finish and returns if child has stopped

do {

wpid2 = waitpid(childPID2, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

}else if(strcmp((tokens[i]),"open") == 0){

childPID2 = fork();

if (childPID2 == 0) {

i++;

path = tokens[i];

openFile(path);

} else if (childPID2 < 0) {

// Error forking

perror("lsh");

} else {

// Parent process waits for child to finish and returns if child has stopped

do {

wpid2 = waitpid(childPID2, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

} else if(strcmp((tokens[i]),"exit") == 0){

kill(getpid(), SIGKILL);

exit(1);

}else if(strcmp(tokens[i],"clear") == 0){

childPID2 = fork();

if (childPID2 == 0) { clear();

}else if (childPID2 < 0) {

// Error forking

perror("lsh");

}else{

//Parent process waits for child to finish and returns if child has stopped

do{

wpid2 = waitpid(childPID2, &status, WUNTRACED);

}while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

}

}else if (childPID < 0) {

// Error forking

perror("lsh");

} else{

// Parent process waits for child to finish and returns if child has stopped

do{

wpid = waitpid(childPID, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

}else if(tokens[i] == NULL){

printf("Invalid Number of Arguments\n");

}

}

}

free(input);

free(tokens);

return 0;

}