Implementation of basic shell interpreter

Authors: D.SAI LOHITH REDDY(19BCE0450), BHARGAV(19BCE0322), ARYAN GUPTA(19BCE2134)

Co-author: prof Deepa.k

***Abstract*— A Shell interpreter is the piece of a computer working framework that comprehends and executes orders that are entered intuitively by a person or from a program. The most conventional feeling of the term shell implies any program that clients utilize to type orders. A shell conceals the details of the fundamental working framework and deals with the specialized subtleties of the working framework portion interface, which is the least level, or "inner-most" part of most working frameworks.**

WHAT WILL THE SHELL DO?

- **Input/output Redirection**
- **Accept user input and parse it into executable commands**
- **Pipes**
- **Job Control**
- **Execute system calls (linux).**

I. INTRODUCTION

The shell content is a PC framework intended to be constrained by a UNIX shell. The language of the shell content is viewed as a content language. A Unix shell is an order line interpreter or shell that gives an order line UI for Unix-like working frameworks. The shell is both an interactive command language and a scripting language, and is utilized by the working framework to control the execution of the framework utilizing shell contents.

In Unix-like working frameworks, clients normally have numerous options of order line interpreters for interactive sessions. At the point when a client signs into the framework intuitively, a shell program is naturally executed for the length of the meeting. The kind of shell, which might be altered for every client, is ordinarily put away in the client's profile, for instance in the neighborhood password document or in a dispersed setup framework, for example, NIS or LDAP; be that as it may, the client may execute some other accessible shell intelligently.

By completing this project, we aim to achieve and understanding of the Linux shell (bash) framework and the practical use of system calls such as fork and concepts such as pipelining.

Literature Survey:

A Unix shell is a command-line interpreter or shell that provides a command line user interface for Unix-like operating systems. The shell is both an interactive command language and a scripting language, and is used by the operating system to control the execution of the system using shell scripts.

The first Unix shell was the Thompson shell, written by Ken Thompson at Bell Labs, and released with Unix versions 1 through 6 from 1971 to 1975. Although the modern standard is simple, it introduces many basic functions that are common in the future. Unix shell, including pipes, simple control structures using if and goto, and filename wildcards.

Although it is not currently in use, it can still be used as part of some ancient UNIX systems.

C shell, csh, written by Bill Joy, is a graduate student at the University of California at Berkeley and widely distributed with BSD Unix. Languages, including control structures and expression syntax, are modeled in C. The .C shell also introduces a number of features for interactive work, including history and editing mechanisms, aliases, directory stacks, waveform symbols, cd path, job control, and path. hope.

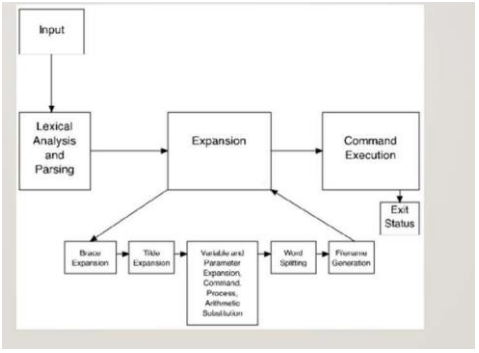
The most generic sense of the term shell means any program that users employ to type commands. A shell hides the details of the underlying operating system and manages the technical details of the operating system kernel interface, which is the lowest-level, or "inner-most" component of most operating systems.

A UNIX shell is a summon dialect mediator, the main role of which is to decipher order lines wrote at a terminal into framework activities. The shell itself is a system through which different projects are conjured. Albeit there are a few distinctive UNIX shells, among

the Bourne shell and the Korn shell, the one most every now and again utilized on Blake inside Berkeley

The shell has some inherent capacity, it can be executed directly, but most orders you enter are like this Causes the shell to execute programs outside the shell. This separates the shell from the other order Translator, the reason is that it is just a client project, and it is just a component Summon different projects.either, but C/ C++ do. In this project, we will design and implement Memory Leak Detector (MLD) tool for C programs, easily extendible to C++ as well.

II. BLOCK DIAGRAM



Overview of the Work

Problem description:

The projects aims to implement a basic shell using the C language, the shell is capable of executing all Linux system calls though the use of execvp() function. The shell also has built-in features such as pipelining (Upto 2 pipes max) and cd command

Software Requirements:

Any UNIX based operating system such as OSX, Ubuntu, Arch Linux, etc.

Hardware Requirements:

Recommended minimum specification specified by Ubuntu:

1. 2 GHz dual core processor
2. 2 GB RAM (system memory)
3. 25 GB of hard-drive space (or USB stick, memory card or external drive)
4. VGA capable of 1024x768 screen resolution.

Implementation

Description of Modules/Programs

int main()

Main function of the C code calls function shell_loop which begins the execution of the shell program.

```
void shell_loop()
void shell_loop()
{
char *cmd;
size_t cmd_size = 32;
cmd = (char *)malloc(cmd_size * sizeof(char));
welcome();
while(1){
printf("$:");
getline(&cmd,&cmd_size,stdin); /*gets the command from user*/
fflush(stdin);
parse_cmd(cmd);
make_shell_wait(n);

n=0;
}
return;
}
```

Main shell loop that takes commands from user till exit(0) is passed.Commands from user are taken in through cmd. While(1) is an infinite loop that prints the shell output till exit(0) is passed, and call the parse_cmd function to parse the command inputted by the user.

```
Void parse_cmd(char * cmd)
void parse_cmd(char * cmd)
{
/*Variables to count no of pipes */
Int
 pipe1=0,pipe2=1;
cmd=skipWS(cmd);
char *pipe_handler=strchr(cmd,'|'); //Finds pipes in the command while(pipe_handler!=NULL)
{
*pipe_handler='\0';          pipe1=run(cmd,pipe1,pipe2,0);
cmd=pipe_handler+1;          pipe_handler=strchr(cmd,'|');
pipe2=0;
}
pipe1=run(cmd,pipe1,pipe2,1); make_shell_wait(n);
n=0;
}
```

The parse_cmd function finds the '|' and assigns it to the pipe_handler pointer

Then using a while loop to execute the pipes(till the penultimate one) and finally the last command.

The function calls for run() for executing the commands.

```
int run(char *cmd,int pipe1,int pipe2,int pipe3);
int run(char *cmd,int pipe1,int pipe2,int pipe3)
{
int i=0;
cmd=skipWS(cmd);
char
*pipe_handler=strchr(cmd,' '); while(pipe_handler!=NULL)
/* If options then*/
{
*pipe_handler='\0';
args[i]=cmd; i++;
cmd=skipWS(pipe_handler+1); pipe_handler=strchr(cmd,' ');
}
if(cmd[0]!='\0') /* Code to handle if no options are present */
{
args[i]=cmd;                    pipe_handler=strchr(cmd,'\n');
pipe_handler[0]='\0';
i++;
}
args[i]=NULL;
if(args[0]!=NULL)
{
if (strcmp(args[0], "exit")==0)
exit(0);
else if(strcmp(args[0], "cd")==0)
{
func_cd(args[1]);
return 0;
}
else n++;
return exec(pipe1,pipe2,pipe3);
}
return 0;
}
```

The function first skips whitespaces using skipWS() function and then parses the command by splitting the commands by finding the ' space ' between the command arguments, during each split the arguments are put into args[] array and the exec function is called ,return value is passed back into parent function.

```
int exec(int pipe1,int pipe2,int pipe3)
int exec(int pipe1,int pipe2,int pipe3)
{
pid_t pid;
int pipefd[2]; pipe(pipefd); pid=fork();
if(pid==0)
{
if (pipe1 == 0 && pipe2 == 1 && pipe3 == 0)
{
dup2( pipefd[1], STDOUT_FILENO );
}
else if (pipe1 != 0 && pipe2 == 0 && pipe3 == 0)
{
```

```
dup2(pipe1, STDIN_FILENO);
dup2(pipefd[1], STDOUT_FILENO);
}
else {
dup2( pipe1, STDIN_FILENO);
}
if (execvp( args[0], args) == -1) printf("Could not run command!\n");
}

if(pipe1!=0) close(pipe1);
close(pipefd[1]);

if(pipe3==1) close(pipefd[0]);
return pipefd[0];
}
```

The function which executes are the arguments in contained in args[] array, this is achieved through the system call execvp(), and using a fork() to execute the command and return control back to the program.
Pipe() function is used to create the pipes, and dup2 to copy the file descriptors and thus successfully opening transferring output from one command to another.

Source Code:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include<sys/wait.h>
#define MAX 100 /*Max no of arguments */

int n=0; /* Stores the number of commands to be executed */
static int count=0;
char* args[MAX];

void shell_loop();
void parse_cmd(char * cmd); char* skipWS(char *s);
void func_cd(char *path); void make_shell_wait(); void welcome();
int run(char *cmd,int pipe1,
int pipe2,int pipe3);
int exec(int pipe1,int pipe2,int pipe3);
int main()
{
printf("Type \'exit\' to exit\n"); shell_loop();
return 0;
}
void shell_loop()
{
char *cmd;
size_t cmd_size = 32;
cmd = (char *)malloc(cmd_size * sizeof(char));
welcome();
while(1){
printf("$:");
getline(&cmd,&cmd_size,stdin); /*gets the command from user*/
fflush(stdin);
pharse_cmd(cmd);
make_shell_wait(n);
 n=0;
}
return;
}
void parse_cmd(char * cmd)
{
/*Variables to count no of pipes */
int pipe1=0,pipe2=1;
cmd=skipWS(cmd);
char *pipe_handler=strchr(cmd,'|'); //Finds pipes in the command while(pipe_handler!=NULL)
{
*pipe_handler='\0';
pipe1=run(cmd,pipe1,pipe2,0);
cmd=pipe_handler+1;
pipe_handler=strchr(cmd,'|');
pipe2=0;
}

pipe1=run(cmd,pipe1,pipe2,1); make_shell_wait(n);
n=0;
}
int run(char *cmd,int pipe1,int pipe2,int pipe3)
{
int i=0;
cmd=skipWS(cmd);
char *pipe_handler=strchr(cmd,' ');
while(pipe_handler!=NULL) /* If options then*/
{
*pipe_handler='\0'; args[i]=cmd;
i++;
cmd=skipWS(pipe_handler+1);
 pipe_handler=strchr(cmd,' ');
}
if(cmd[0]!='\0') /* Code to handle if no options are present */
{
args[i]=cmd;                    pipe_handler=strchr(cmd,'\n');
pipe_handler[0]='\0';
i++;
}
args[i]=NULL;

if(args[0]!=NULL)
{
if (strcmp(args[0], "exit")==0)
exit(0);
 else if(strcmp(args[0], "cd")==0)
{
func_cd(args[1]);
 return 0;
}
else n++;
return exec(pipe1,pipe2,pipe3);
}
return 0;
}
int exec(int pipe1,int pipe2,int pipe3)
{
pid_t pid;
int pipefd[2]; pipe(pipefd); pid=fork(); if(pid==0)
{
if (pipe1 == 0 && pipe2 == 1 && pipe3 == 0) { dup2( pipefd[1], STDOUT_FILENO );
}
else if (pipe1 != 0 && pipe2 == 0 && pipe3 == 0) {
dup2(pipe1, STDIN_FILENO);
dup2(pipefd[1], STDOUT_FILENO);
}
```

```
else {
dup2( pipe1, STDIN_FILENO);
}
if (execvp( args[0], args) == -1) printf("Could not run command!\n");
}
if(pipe1!=0) close(pipe1);
close(pipefd[1]);
if(pipe3==1)

close(pipefd[0]);
return pipefd[0];
}
char* skipWS(char *s)
{
while(isspace(*s)){ if(isspace(*s))
++s;
}
return s;
}
void make_shell_wait()
{
int i; for(i=0;i<n;i++)
{
wait(NULL);
}
}
void func_cd(char *path)
{
chdir(path);
}
void welcome()
{
printf("╦        ╦ ╦╔═╗  ╔═╗╔═╗  ╔═╗╦═╗╔═╗ ╦╔═╗╔═╗╔╦╗\n");
printf("╚╗╔╝╠╣ ║    ║  ║ ║╚═╗  ╠═╝╠╦╝║  ║ ║║╣ ║  ║ \n");
printf("   ╚╝   ╩ ╩    ╚═╝╚═╝ ╩    ╩╚═╚═╝╚╝╚═╝╚═╝ ╩ \n");
```

Test cases:

Standalone-commands:

pwd: Shows current working directory date:Prints date

cd: Changes working directory

Commands with option:

ls- la: expanded ls function uname -r: prints version number Touch [file]: create a file.

Commands with pipe:

ls | grep test.txt | wc:

print wc of serach “test.txt”. cat test.txt | uniq:

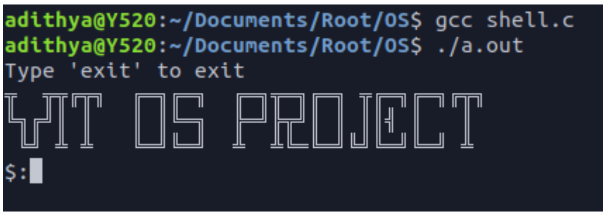
print unique data from test.txt

Execution snapshots:

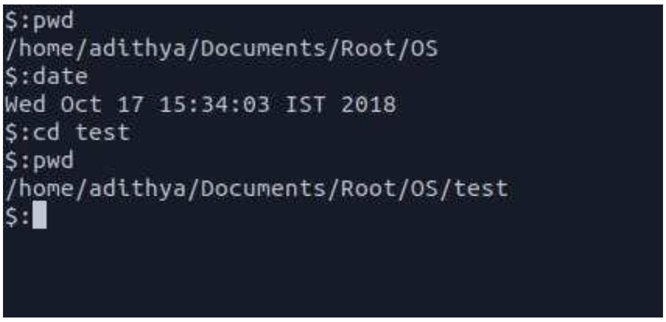
Note: The implementation was done on my friend Aditya’s PC since at that moment a Linux system wasn’t available to us.

Initial execution:

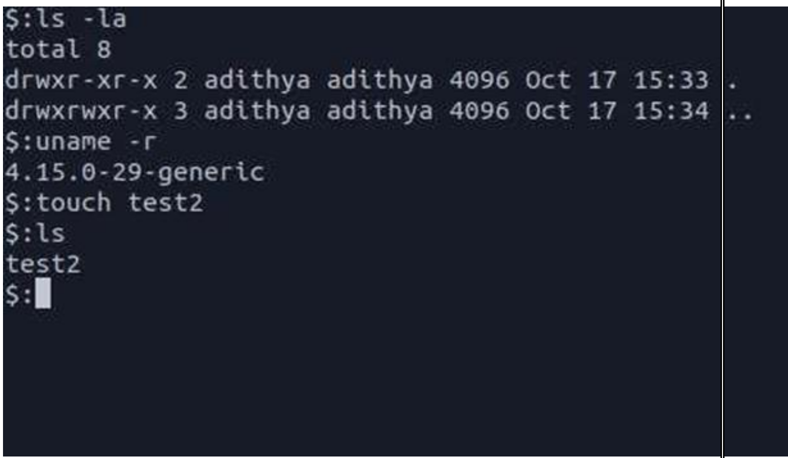
Test Case 1:



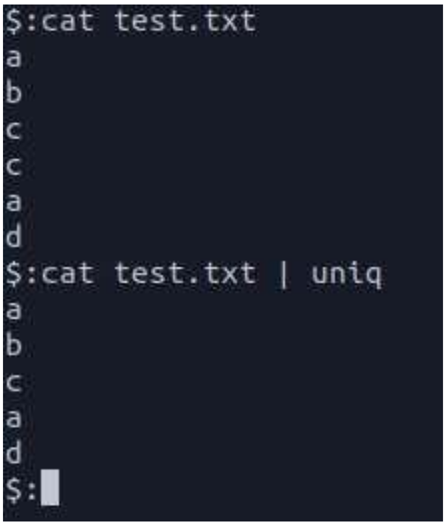
Test Case 2:



Test Case 3:



Test case 4:



Conclusion and Future Directions:

After all the rigorous sittings and hours of exhausting labor by all the team members, the project is finally completed and hopefully shaped in a very satisfying approach.

The application (Shell Interpreter) stands out in most of its performances needs and conjointly encompasses a very friendly and self- explaining user interface.

Currently the shell does not support I/O redirection and only a maximum of 3 pipes are currently supported thus in the future. These features are to be implement.

References

- Stephen G Kochan and Patrick Wood ,UNIX Shell Programming, ; 3rdEdition(2003). Retrieved from https://Unix-Shell-Programming-Stephen-Kochan/dp/0672324903

- Mark Bates,Conquering the Command Line (UNIX and LINUX commands for developers)(2007),Retrieved from http://conqueringthecommandline.com/

- Morris I. Bolsky and David G.Korn ; The New KornShell, 2nd Edition(1996) Retrieved from https://New-KornShell-Command-Programming-Language/dp/0131827006/
- Shell Scripting Recipes: A Problem-Solution Approach- Chris F.A Johnson 2015
- Concept and applications of UNIX- SumitabhaDas 2011