IMPLEMENTATION OF UNIX SHELL

NIT ANDHRA PRADESH

UNDER THE GUIDANCE OF

Mr. Prajwal Pralhad Panzade

Report By
JAI SHANKAR
411529
NAVEEN REDDY
411525
April 5,2018

Contents

1 INTRODUCTION

In the architecture of Unix, the Shell Layer exists on top of the Kernel Layer and below the Application Layer. Basically Shell is an Interpreter that interprets the commands entered by the user and produces the output. Hence it is also known as Command Line Interpreter. It is an abstraction or a way to communicate to the underlying kernel and Hardware. Most Popular shell programs in UNIX are:

Table 1: Popular Shells.

Shell	**Description**
sh	Shell Program. The original shell program in UNIX.
csh	C Shell. An improved version of sh.
tcsh	A version of Csh that has line editing
ksh	Korn Shell. The father of all advanced shells.
bash	The GNU shell. Takes the best of all shell programs. It is currently the most common shell program.

The input can be either in batch mode or Interactive mode. In Interactive mode, the user enters some predefined commands along with arguments and gets the output. A simple command is a sequence of optional variable assignments followed by blank-separated words and re directions, and terminated by a control operator. The first word specifies the command to be executed, and is passed as argument 0. The remaining words are passed as arguments to the invoked command. The return value of a simple command is its exit status, or 128+n if it is terminated by signal n. The commands entered will be first searched by the Shell in the $PATH environment variable and gets executed if the corresponding binary file is found else it displays "no command found". Apart from predefined commands of Unix, other commands can be executed if the command's corresponding binary file is included in the $PATH Environment Variable.

2 IMPLEMENTATION

The Shell implementation is divided into three parts.

2.1 Parser

It is the component that takes the input, removes unnecessary spaces or tabs, checks for pipe filters, separates the input to command and arguments and then sends them to the Executor Part.

2.2 Executor

Starting processes is the main functionality of the Shell. There are only two ways by which we can create processes.

2.2.1 Init

This is the process that is executed once the kernel is loaded and initialised. Init runs for the entire life time until the computer is ON and manages the other processes of the computer.

2.2.2 Fork

This is the one that is useful for the shell The Executor uses fork() to create a child process, execvp() System call to execute the command and then the shell waits for the return using wait() system call. If there are pipes in the command, then it uses the duplicate function dup2() to interchange the file descriptors back and forth between stdin/stdout and the command's file descriptor so that the output from the one command is passed as input to the next command and so on. Additionally, it will redirect the stdin,stdout and stderr if there are any re-directions. The exec() system call cannot be used as the problem is that it doesn't return once called. Hence, we are using a variant of exec(), i.e execvp().

2.3 Shell Subsystems

Among the many shell subsystems, the following are the most important

2.3.1 Environment Variables

Expressions of the form $VAR are expanded with the corresponding environment variable. Also the shell can set, expand and echo environment variables.

2.3.2 Wild Cards

Shell offers regular expressions in grep and extended regular expressions in egrep.

2.3.3 History

Stores the commands in a file either valid or invalid which can be seen using the command history or !. For this subsystem,the readline library from the GNU is used.

2.4 Flow of Functions and System calls

sh compatible shell is implemented in c language along with library libreadline-dev .The following are the functions and system calls used.

2.4.1 int getcwd(dir,size)

This system call stores the absolute path of the current working directory into dir. This dir is used as prompt for the user to enter a command.

2.4.2 char* getenv("USER")

This returns the user name of the user who is using the shell.

2.4.3 char* readline(prompt)

This function reads the commands from the stdin.

2.4.4 void fileprocess()

This is the function that stores the commands entered by the user in an array. This is executed when the input is nonempty and not beginning with '!'. The array is passed on to filewrite().

2.4.5 void filewrite()

This takes the array and writes them to a file named 'history.txt'.

2.4.6 int with_pipe_execute()

This function is the initial function which is called for checking the all the command after initial pre processing . It passes the processed output to function split.

2.4.7 int split(char *cmd_exec, int input, int first, int last)

This function is responsible for splitting of command and passing it to command function.

2.4.8 static int command(int input, int first, int last, char *cmd_exec)

This does the major part of the program. It checks for various possibilities of commands. The types of commands that are checked are as under:

- Internal commands: pwd and cd
- echo commands, setting and getting environment variables
- redirection handler
- PIPE
- External commands

It make use of various functions like tokenise_redirect_input_output,tokenise_redirect_input,tokenise_redirect_output which internally calls tokenise_commands() for tokenization.

2.5 Helper Functions

2.5.1 pid = fork()

This function creates a child process of the current process and returns the pid of the process to the parent process and 0 to the child process. The command() function is called only when it is the child process.

2.5.2 ret = waitpid(pid, &status, 0)

This function is used by the parent process so that the parent waits for the child to complete its execution and return.

2.5.3 execvp(args[0], args)

The child process uses this call to execute the command 'args[0]' using the corresponding arguments 'args'.

2.5.4 openhelp()

This function gets executed when the user enters help and opens the help of the shell that is built.

2.5.5 void clear_variables()

This function is executed after each command has completed its execution and clears variables like buffer,flags and other global variables.

2.5.6 void history_execute_with_constants()

This function gets executed when the command's first character is '!' and then searches in the history for the corresponding command number.

2.5.7 void change_directory()

This function is used when the user enters 'cd' command. This function in return calls the chdir() system call.

2.5.8 echo_calling(char *echo_val)

This function is explicitly called when the user enter the echo command followed by its arguments.

2.5.9 void set_environment_variables()

This function is used for the setting/modifying the environment variables.

2.5.10 Tokenise functions

These are the functions used to split the input into command and corresponding arguments, piped commands into individual commands, removing empty spaces.

2.5.11 void sigintHandler(int sig_num)

This is used for handling signal interrupt

2.5.12 free(char *)

This frees the space required for the input command.

3 CONCLUSION

The Shell program is implemented in C-Language using the readline library from the GNU. Almost all the commands of the bash shell have been implemented including environment variables, history and auto completion. Both Interactive and Batch Mode inputs can be given to the shell. Also, defensive programming has been used so that there are no occurrences of circumstances like core dump,hanging indefinitely or premature termination. Help and man pages can also be accessed.