Shell

A REPORT ON PACKAGE SUBMITTED BY

KRISHNAN S G

18PT19

SADHAM HUSSIAN

18PT29

APRIL 2020

OPERATING SYSTEMS

18XT44

DEPARTMENT OF APPLIED MATHEMATICS

AND COMPUTATIONAL SCIENCES

PSG
College of Technology

Contents

Contents **2**

1 Introduction **3**

1.1 What is a shell? 3

1.2 Shell in UNIX architecture? 4

1.3 Responsibilities of Shell 5

1.3.1 Program Execution 5

1.3.2 I/O redirection 5

1.3.3 Pipeline hookup 5

1.3.4 Variable and filename substitution 6

1.3.5 Environment control 6

1.4 Lifetime of shell 6

1.4.1 Initialize 6

1.4.2 Interpret 6

1.4.3 Terminate 7

2 Description **8**

3 System calls **9**

3.1 getcwd() 9

3.2 getuid() 10

3.3 getpwuid() . . . 10
3.4 chdir() . . . 11
3.5 fork() . . . 11
3.6 execvp() . . . 12
3.7 dup() . . . 13
3.8 dup2() . . . 13

4 Tools and Technologies **15**
4.1 GCC . . . 15
4.2 GNU Debugger . . . 15
4.3 VCS . . . 15

5 Workflow **16**
5.1 The Loop . . . 16
5.1.1 Read . . . 16
5.1.2 Parse . . . 17
5.1.3 Execute . . . 18
5.2 Commands . . . 19
5.2.1 Internal commands . . . 19
5.2.2 External commands . . . 20
5.3 Input-Output redirection . . . 20
5.4 Pipes . . . 21
5.5 Background Process . . . 21

6 Result and Discussion **22**
6.1 Summary . . . 22
6.2 Future enhancement . . . 22

7 Conclusion **24**

Chapter 1

Introduction

1.1 What is a shell?

A shell (also known as "command-line interpreter" or CLI) is a special application which interacts with the user and the system, thus creating an interface between the two. Fig. 1.1 shows how a shell enables the user to interact with the system (kernel).

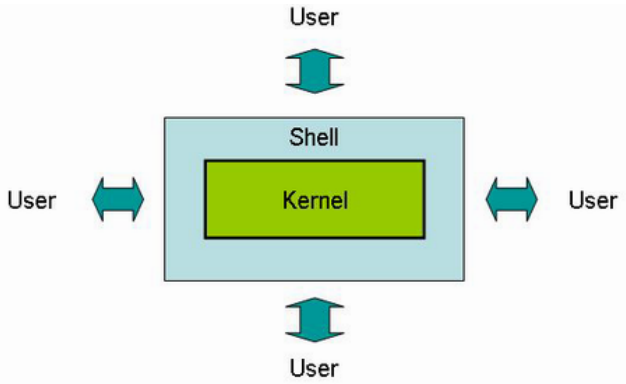

Figure 1.1: Shell

1.2 Shell in UNIX architecture?

A Shell provides you with an interface to the Unix system [1] . It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Fig. 1.2 illustrates how the kernel [2] is closest to the hardware, utility programs like shell on top of the kernel and application programs on the outer layer interacting with the user.

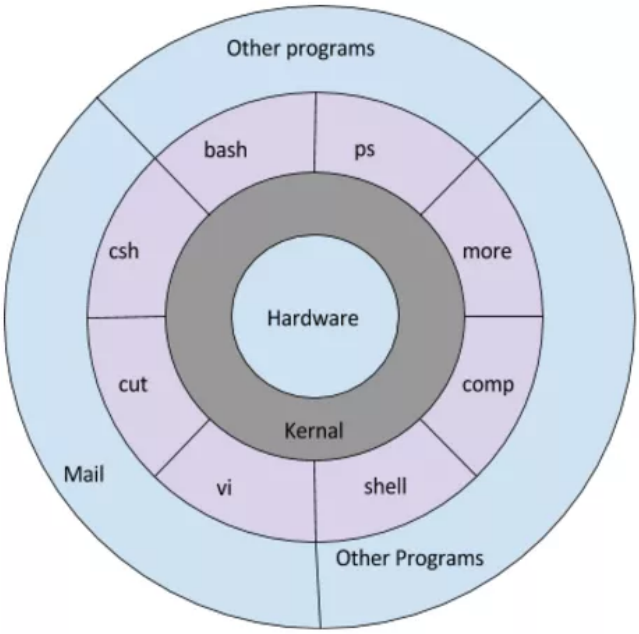

Figure 1.2: Shell in UNIX Architecture

1.3 Responsibilities of Shell

1.3.1 Program Execution

The shell is responsible for the execution of all programs that you request from your terminal.

Each time you type in a line to the shell, the shell analyzes the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format:

program-name argument1 argument2 . . . .

The shell uses special characters to determine where the program name starts and ends, and where each argument starts and ends. These characters are collectively called **whitespace characters**, and are the space character, the horizontal tab character, and the end-of-line character.

1.3.2 I/O redirection

Shell takes care of input and output redirection on the command line. It scans the command line for the occurrence of the **special redirection characters** $<$, $>$, or **».**

ls > output.txt

The shell recognizes the special output redirection character $>$ and takes the next word on the command line as the name of the file that the output is to be redirected to.

1.3.3 Pipeline hookup

Shell scans the command line looking for redirection characters, it also looks for the **pipe character** | . For each such character that it finds, it connects the

standard output from the command preceding the | to the standard input of the one following the | . It then initiates execution of both programs.

*who	wc -l*

1.3.4 Variable and filename substitution

Like any other programming language, the shell lets you assign values to variables. The shell also performs filename substitution on the command line. In fact, the shell scans the command line looking for filename substitution characters $*$, ?, or [...] before determining the name of the program to execute and its arguments.

1.3.5 Environment control

The shell provides certain commands that let you customize your environment. Your environment includes your home directory, the characters that the shell displays to prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

1.4 Lifetime of shell

1.4.1 Initialize

In this step, the shell would read and execute its configuration files. These change aspects of the shell's behavior.

1.4.2 Interpret

Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them. The commands are then parsed, tokenized and then executed either in foreground or background.

1.4.3 Terminate

After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.

Chapter 2

Description

To design and implement a program that acts like a simple shell in C language and over the UNIX/Linux operating system, The program must follow strictly the following specifications and requirements.

1) Parser and tokenizer
2) Command execution
3) I/O redirection
4) Handle pipes
5) Background process
6) Implement two internal commands: globalusage and averageusgage.

Chapter 3

System calls

3.1 getcwd()

Name

getcwd - get the path name of the current working directory

Synopsis

include <unistd.h>

char * getcwd(char * *buf*, size_t *size*);

Description

- The *getcwd*() function shall place an absolute pathname of the current working directory in the array pointed to by *buf*, and return *buf*.
- The pathname copied to the array shall contain no components that are symbolic links.
- The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd*() is unspecified.

Return value

Upon successful completion, *getcwd*() shall return the *buf* argument. Otherwise, *getcwd*() shall return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by *buf* are then undefined.

3.2 getuid()

Name

getuid - get a real user ID

Synopsis

include <unistd.h>

uid_t getuid(void);

Description

The *getuid*() function shall return the real user ID of the calling process.

Return value

The *getuid*() function shall always be successful and no return value is reserved to indicate the error.

3.3 getpwuid()

Name

getpwuid, getpwuid_r - search user database for a user ID

Synopsis

include <pwd.h>

struct passwd * getpwuid(uid_t *uid*);

Description

- The *getpwuid*() function shall search the user database for an entry with a matching *uid*.
- The *getpwuid*() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
- Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid*(). If *getpwuid*() returns a null pointer and *errno* is set to non-zero, an error occurs.

Return value

- The *getpwuid*() function shall return a pointer to a **struct passwd** with the structure as defined in *<pwd.h>* with a matching entry if found.

- A null pointer shall be returned if the requested entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

3.4 chdir()

Name

chdir - change working directory

Synopsis

include <unistd.h>

int chdir(const char * *path*);

Description

The *chdir*() function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '/'.

Return value

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, the current working directory shall remain unchanged, and errno shall be set to indicate the error.

3.5 fork()

Name

fork - create a new process

Synopsis

include <unistd.h>

pid_t fork(void);

Description

The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

- The child process shall have a unique process ID.

- The child process ID also shall not match any active process group ID.
- The child process shall have a different parent process ID, which shall be the process ID of the calling process.
- The child process shall have its own copy of the parent's file descriptors. Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
- The child process shall have its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.

Return value

- Upon successful completion, fork() shall return 0 to the child process and shall return the process ID of the child process to the parent process.
- Both processes shall continue to execute from the fork() function.
- Otherwise, -1 shall be returned to the parent process, no child process shall be created, and errno shall be set to indicate the error.

3.6 execvp()

Name

execvp - execute a file

Synopsis

include <unistd.h>

int execvp(const char * *file*, char * const *argv*[]);

Description

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file unistd.h. There are many members in the exec family which are shown below with examples.

- execvp : Using this command, the created child process does not have to run the same program as the parent process does. The exec type system calls allow a process to run any program files, which include a binary executable or a shell script.

Return value

If one of the exec functions returns to the calling process image, an error has occurred; the return value shall be -1, and errno shall be set to indicate the error.

3.7 dup()

Name

dup - duplicate an open file descriptor

Synopsis

include <unistd.h>

int dup(int *fildes*);

Description

The dup() function provides an alternative interface to the service provided by fcntl() using the F_DUPFD command. The call dup(fildes) shall be equivalent to:

fcntl(fildes, F_DUPFD, 0);

Return value

Upon successful completion a non-negative integer, namely the file descriptor, shall be returned; otherwise, -1 shall be returned and errno set to indicate the error.

3.8 dup2()

Name

Dup2 - duplicate an open file descriptor

Synopsis

include <unistd.h>

int dup2(int fildes, int fildes2);

Description

- The dup2() function shall cause the file descriptor fildes2 to refer to the same open file description as the file descriptor fildes and to share any locks, and shall return fildes2.
- If fildes2 is already a valid open file descriptor, it shall be closed first, unless fildes is equal to fildes2 in which case dup2() shall return fildes2 without closing it.
- If the close operation fails to close fildes2, dup2() shall return -1 without changing the open file description to which fildes2 refers.
- If fildes is not a valid file descriptor, dup2() shall return -1 and shall not close fildes2. If fildes2 is less than 0 or greater than or equal to { OPEN_MAX} , dup2() shall return -1 with errno set to [EBADF].

Upon successful completion, if fildes is not equal to fildes2, the FD_CLOEXEC flag associated with fildes2 shall be cleared. If fildes is equal to fildes2, the FD_CLOEXEC flag associated with fildes2 shall not be changed.

Return value

Upon successful completion a non-negative integer, namely the file descriptor, shall be returned; otherwise, -1 shall be returned and errno set to indicate the error.

Chapter 4

Tools and Technologies

4.1 GCC

GNU GCC compiler was used to compile and create executable for the shell.

4.2 GNU Debugger

The debugger was used to detect and resolve bugs during the development stage.

4.3 VCS

We have used Git as our version control system to manage and collaborate on the project.

Chapter 5

Workflow

After understanding the requirements and specifications mentioned in the problem statement, we have implemented our own version of shell for linux systems called "**skerl**" .

Figure 5.1: skerl

5.1 The Loop

The implemented skerl shell has a basic loop which does the following actions:

5.1.1 Read

Accepts inputs from the standard input using getline function. When exit command is encountered the shell terminates. The history command shows the history and updates the history file. The following code snippet explains how the command is read from the user.

```
char *input_command = NULL;
ssize_t input_command_buffer = 0;
getline(&input_command, &input_command_buffeer, stdin);
if (strcmp(input_command,"exit")==0)
    add_command_history(input_command);
    exit;
if (strcmp(input_command,"history")==0)
    add_command_to_history(input_command);
else
    add_command_to_history(input_command);
parse(input_command);
```

5.1.2 Parse

The main responsibility of the parser is to parse the command string into a program and arguments. The parse functionality also tokenizes the string.

The following code snippet explains how the command is parsed and tokenized.

```
char **skerl_split_command(char *input_command, int *count)
{
    char **tokens = malloc(buffer_size * sizeof(char *));
    token = strtok(input_command, SKERL_TOKEN_DELIMITER);
    while (token != NULL)
    {
        tokens[position] = token;
        position++;

        if (position >= buffer_size)
        {
            buffer_size += SKERL_TOKEN_BUFFERSIZE;
            tokens = realloc(tokens, buffer_size * sizeof(char *));
        }
        token = strtok(NULL, SKERL_TOKEN_DELIMITER);
    }
    tokens[position] = NULL;
    return tokens;
}
```

```
char **command = skerl_split_command(input_command, &token_count);
int type = 0;
for (int i = 1; i < token_count; i++)
{
    if ((strcmp(command[i],''>'') == 0) || (strcmp(command[i],''>>'') == 0))
    {
        type = 1;
        output_redirection(command, i);
    }
    else if (strcmp(command[i],''<'') == 0)
    {
        type = 1;
        input_redirection(command, i);
    }
    else if (strcmp(command[i],''|'') == 0)
    {
        type = 1;
        execute_pipe(command, i);
    }
}
if (type == 0)
    skerl_execute(command);
```

5.1.3 Execute

From the tokens generated by the parser, The executer identifies the tokens.

The executor has two main responsibilities, executing internal commands and creating child processes to execute external commands. For processes identified as background processes the executor does not wait for the child process to complete, instead return the **pid** and continue with the loop.

The following code snippet explains how the command is executed.

```
for (int i=0; i<skerl_total_builtin_command(); i++)
{
     if (strcmp(command[0], builtin_command[i]) == 0)
         return (*execute_builtin_command[i])(single_command);
}
return skerl_execute_external_command(command);
```

```
int skerl_execute_external_command(char **single_command, int background)
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
    {
        if (execvp(single_command[0], single_command) == -1)
        {
            perror("skerl");
        }
        exit(EXIT_FAILURE);
    }
    else
    {
        if (!background)
            waitpid(pid, &status, WUNTRACED);
        else
        // return pid
    }
}
```

5.2 Commands

The commands a shell accepts and parses can be classified into internal and external based on how the command is executed.

5.2.1 Internal commands

Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable and also no process needs to be spawned for executing it.

Internal commands supported by skerl:

cd, pwd, help, exit, globalusage and averageusage

Implementation of globalusage and averageusage

1) globalusage

   Reads .usage.log file and counts the overall number of commands executed since the first use.

/home/user$ globalusage
skerl usage: 23 commands; current session: 6 commands

2) averageusage

   Reads .usage.log file and shows the avg no. of commands executed since the first use.

/home/user$ averageusage
skerl average usage: 5 commands; current usage: 26.01%

5.2.2 External commands

Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in the PATH variable and also a new process has to be spawned and the command gets executed.

<Add code to which executes external commands. Only the important part>

5.3 Input-Output redirection

Redirections in skerl are achieved by using the **dup** and **dup2** system call. The following code snippet explains how output redirection is implemented.

```
int saved_stdout = dup(1);
output_file = open(filename, O_WRONLY);
dup2(output_file, 1);
skerl_execute(command, 0);
dup2(saved_stdout, 1);
close(saved_stdout);
```

The dup2 system call is used to replace stdout with the file descriptor of the filename provided. Hence all outputs generated are written using the newly created file descriptor instead of stdout.

The dup system call is used to duplicate the stdout(1) so as to restore the original stdout after the redirection operation is performed.

5.4 Pipes

Pipes in skerl are handled by creating a temporary file descriptor to hold the output of the previous command. The input for the command after pipe(|) is read from the temporary file descriptor.

5.5 Background Process

To execute a process in background in the skerl shell, the command is suffixed with & to inform the shell to execute the process in background and return it's **pid**

Chapter 6

Result and Discussion

6.1 Summary

The following features have been implemented in **skerl** shell.

1) Parser and tokenizer
2) Command execution
3) I/O redirection
4) Handle pipes
5) Background process
6) History
7) Implement two internal commands: globalusage and averageusgage.

6.2 Future enhancement

The current implementation of skerl has all basic features expected in a shell, but as we all know there is no end to software creation and innovation. The following list of enhancements can be made:

1) Display better prompt messages with colors.
2) Provide alias for commands.

3) A better version of history.

4) Add more internal commands.

Chapter 7

Conclusion

This report summarizes the concepts related to shell and it's working. Also as per the requirements mentioned in the problem statement we have created our own shell –**skerl** for linux systems.

Bibliography

[1] W. Richard Stevens Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Ed. by Addison-Wesley. Vol. 3. 2013.

[2] Silberschatz, Gavin, and Gagne. *Operating System Concepts*. Ed. by Wiley. Vol. 3. 2011.