*A project on*

**Implementation of a shell containing commands similar to ls, grep, mkdir, wc, cmp and a command to produce zombie process only using system call**

**By :**

Amrithesh (01JST18CS008)

Vidyasagar M S (01JST18CS166)

Vishal Rajkumar Naik (01JST18CS171)

Vishwanath (01JST18CS172)

(CS 4th Semester, Section: C)



**Department of Computer Science**

**JSS MAHAVIDYAPEETHA**

**JSS SCIENCE & TECHNOLOGY UNIVERSITY, Mysuru - 570006**

*(formerly Sri Jayachamarajendra College of Engineering)*

**Implementation of a shell containing commands similar to ls, grep, mkdir, wc, cmp and a command to produce zombie process only using system call**

**Abstract**

A shell is software that provides an interface for an operating system's users to provide access to the kernel's services. On Unix-based or Linux-based operating systems, a shell can be invoked through the shell command in the command line interface, allowing users to direct operations through computer commands, text or script. Shells also exist for programming languages, providing them with autonomy from the operating system and allowing cross-platform compatibility.

Implementing an own shell consists of several steps. First of all, it should tell users about its features. It should read the command provided through terminal. It should add the command to the history for easy access. It should identify commands to be executed. Piping and improper space aligning should be taken care if present. Commands can be executed by fork and exec system calls. Zombie is a child process which has completed its execution. It cannot be killed since it is already dead. Only way to eliminate is by killing its parent process. So, we need to create another program for the zombie process.

|  |  |
| --- | --- |
| **TABLE OF CONTENTS** | **PAGE NO** |
| 1. **Introduction** | 3 |
| **1.1 Working of ls command** | 4 |
| **1.2 Working of grep command** | 5 |
| **1.3 Working of mkdir command** | 6 |
| * 1. **Working of wc command** | 7 |
| **1.5 Working of cmp command** | 7 |
| **1.6 Working of zombie process in Linux** | 8 |
| 1. **Design and Implementation** | 12 |
| **2.1 Implementation** | 12 |
| 1. **Testing/Result Analysis** | 21 |
| 1. **Conclusion and future enhancements** | 22 |
| 1. **References** | 22 |

**1. Introduction**

If you are using any major operating system you are indirectly interacting to **shell**. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal.

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

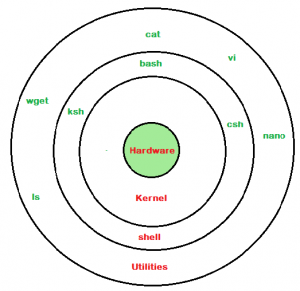


Fig. 1.0 Linux shell

Shell is broadly classified into two categories –

* + Command Line Shell
  + Graphical shell

**Command Line Shell**

Shell can be accessed by user using a command line interface. A special program called **Terminal** in Linux/macOS or **Command Prompt** in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user.

**Graphical Shells**

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every action.

There are several shells are available for Linux systems like –

* + [BASH (Bourne Again Shell)](https://en.wikipedia.org/wiki/Bash_(Unix_shell)) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
  + [CSH (C Shell)](https://en.wikipedia.org/wiki/C_shell) – The C shell’s syntax and usage are very similar to the C programming language.
  + [KSH (Korn Shell)](https://en.wikipedia.org/wiki/Korn_shell) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built in functions.

**1.1 Working of ls command**

The **ls** command is used to list the directory contents in Linux. This is a very common task for every Linux power users and system administrators. In this article, I am going to show you how to use the ls command in Linux. So, let’s get started.

### Listing Directory Contents with ls:

You can list the contents of your current working directory with ls. This is the most common usage of ls. To list the contents of your current working directory, run ls as follows. As you can see, the directory contents of my current working directory (which is the login user’s HOME directory by default) is listed.

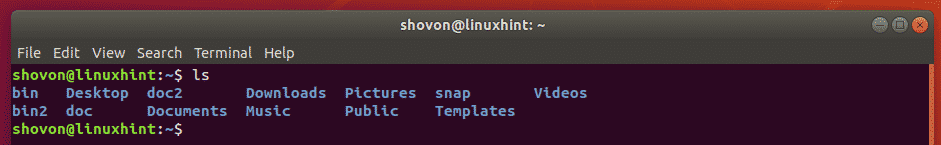


Fig. 1.1 ls command

You can also list the contents of another directory using the full/absolute path or relative path of that directory. For example, let’s say you want to list the contents of the **/etc** directory. To do that, run ls as follows.

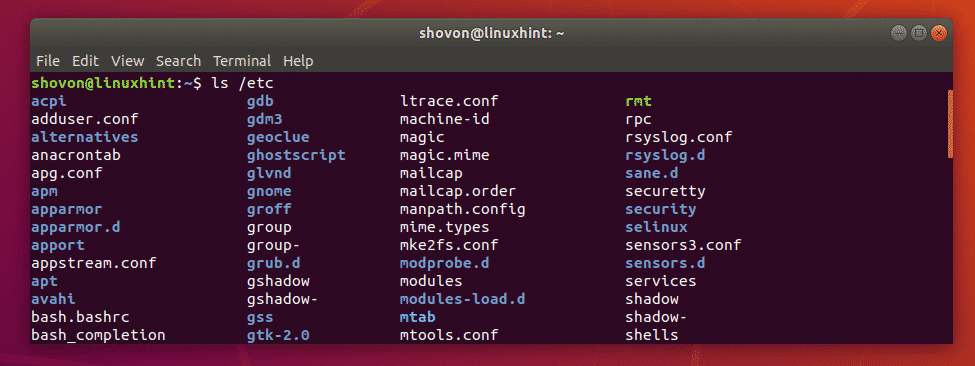


Fig. 1.2 ls command implementation

**1.2 Working of grep command**

The grep command which stands for “global regular expression print,” processes text line by line and prints any lines which match a specified pattern. The grep command is used to search text or searches the given file for lines containing a match to the given strings or words. By default, grep displays the matching lines. Use grep to search for lines of text that match one or many regular expressions, and outputs only the matching lines. Grep is considered to be one of the most useful commands on Linux and Unix-like operating systems. **grep** is a powerful file pattern searcher in **Linux**. If it is not installed on your system, you can easily install it via your package manager (**apt-get** on **Debian/Ubuntu** and **yum**on**RHEL/CentOS/Fedora**). For installing grep in your system, please use the following command.

$ sudo apt-get install grep         #Debian/Ubuntu

$ sudo yum install grep             #RHEL/CentOS/Fedora

grep searches the named input FILEs (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given PATTERN. By default, grep prints the matching lines. In addition, three variant programs egrep, fgrep and rgrep are available. egrep is the same as grep -E. fgrep is the same as grep -F. rgrep is the same as grep -r. Direct invocation as either egrep or fgrep is deprecated, but is provided to allow historical applications that rely on them to run unmodified. The grep has no limits on input line length other than available memory, and it can match arbitrary characters within a line. If the final byte of an input file is not a newline, grep silently supplies one. Since newline is also a separator for the list of patterns, there is no way to match newline characters in a text.

**Using grep for search files**

To search /etc/passwd file for the user harry, enter the following command.

$ grep harry /etc/passwd

Sample outputs:

harry:x:1000:1000:harry,,,:/home/harry:/bin/ksh

**1.3 Working of mkdir command**

You can create new folders and directories in [Linux](https://www.lifewire.com/beginners-guide-to-linux-4090233) using the [command line](https://www.lifewire.com/what-is-a-command-line-interpreter-2625827). The [command](https://www.lifewire.com/what-is-a-command-2625828) to create directories is mkdir. Below is a look at how to create directories with mkdir. We'll also look at the switches used with mkdir and the proper [syntax](https://www.lifewire.com/what-is-syntax-2626014) required to make the command work correctly.

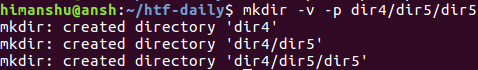


Fig. 1.3 mkdir command

The -p argument is useful, when you don't want to make parent directories manually.

If you want to create a directory containing several subdirectories, or a directory tree, using the command line in Linux, generally you have to use the mkdir command several times. However, there is a faster way to do this.

Let’s say we’ve created a directory called htg, and want to create four subdirectories within it. In a normal situation, we’d use the mkdir command to create the htg directory. Then, we’d need the cd command to change to the new htg directory and, finally, we we’d use the mkdir command again four times to create the four subdirectories.

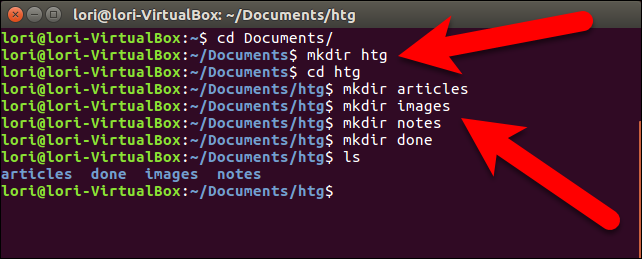


Fig. 1.5 mkdir implementation

.To create a new directory with multiple subdirectories you only need to type the following command at the prompt and press Enter .

mkdir -p htg/{articles,images,note,done}

**1.4 Working of wc command**

The **wc** (**word count**) command in Unix/Linux operating systems is used to find out number of **newline count**, **word count**, **byte and characters** count in a files specified by the file arguments. The syntax of **wc** command as shown below.

$ wc [options] filenames

The following are the options and usage provided by the command.

**wc -l** : Prints the number of lines in a file.

**wc -w** : prints the number of words in a file.

**wc -c** : Displays the count of bytes in a file.

**wc -m** : prints the count of characters from a file.

**wc -L** : prints only the length of the longest line in a file

* 1. **Working of cmp command**

**cmp** command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

* When cmp is used for comparison between two files, it reports the location of the first mismatch to the screen if difference is found and if no difference is found *i.e* the files compared are identical.
* cmp displays no message and simply returns the prompt if the the files compared are identical.

**Syntax:**

**cmp [OPTION]... FILE1 [FILE2 [SKIP1 [SKIP2]]]**

SKIP1, SKIP2 & OPTION are optional

and FILE1 & FILE2 refer to the filenames.

The syntax of cmp command is quite simple to understand. If we are comparing two files then obviously, we will need their names as arguments (*i.e. as FILE1 & FILE2 in syntax*). In addition to this, the optional SKIP1 and SKIP2 specify the number of bytes to skip at the beginning of each file which is zero by default and OPTION refers to the options compatible with this command about which we will discuss later on.

**cmp** Example**:**As explained that the cmp command reports the byte and line number if a difference is found. Now let’s find out the same with the help of an example. Suppose there are two files which you want to compare one is file1.txt and other is file2.txt:

**$cmp file1.txt file2.txt**

If the files are not identical:the output of the above command will be :

**$ cmp file1.txt file2.txt**

**file1.txt file2.txt differ: byte 9, line 2**

/\*indicating that the first mismatch found in

two files at byte 20 in second line\*/

If the files are identical: you will see something like this on your screen:

**$ cmp file1.txt file2.txt**

**$ \_**

**/\*indicating that the files are identical\*/**

**1.6 Working of zombie process in linux**

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child’s exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.

A diagram that demonstrates the creation and termination of a zombie process is given as follows:

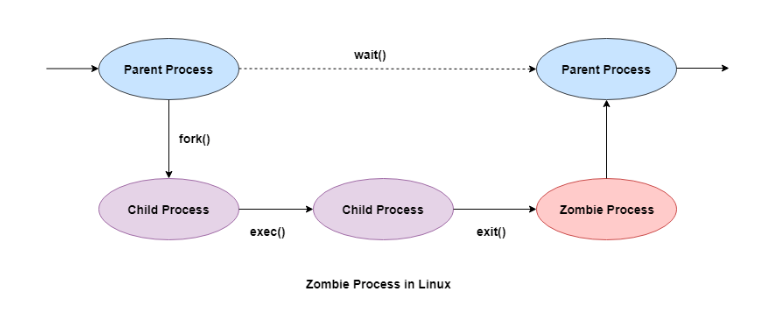


Fig. 1.6 Zombie process

**Salient points of Zombie Processes**

Some of the salient points related to zombie processes are as follows:

1. All the memory and resources allocated to a process are deallocated when the process terminates using the exit() system call. But the process’s entry in the process table is still available. This process is now a zombie process.
2. The exit status of the zombie process zombie process can be read by the parent process using the wait() system call. After that, the zombie process is removed from the system. Then the process ID and the process table entry of the zombie process can be reused.
3. If the parent process does not use the wait() system call, the zombie process is left in the process table. This creates a resource leak.
4. If the parent process is not running anymore, then the presence of a zombie process indicates an operating system bug. This may not be a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system such as running out of process table entries.
5. The zombie processes can be removed from the system by sending the SIGCHLD signal to the parent, using the kill command. If the zombie process is still not eliminated from the process table by the parent process, then the parent process is terminated if that is acceptable.

**Dangers of Zombie Processes**

Zombie processes don't use any system resources but they do retain their process ID. If there are a lot of zombie processes, then all the available process ID’s are monopolized by them. This prevents other processes from running as there are no process ID’s available.

The presence of zombie processes also indicates an operating system bug if their parent processes are not running anymore. This is not a serious problem if there are a few zombie processes but under heavier loads, this can create issues for the system.

**Killing Zombie Processes**

Zombie process is terminated by killing the parent process. Zombie processes can also be killed by sending the SIGCHLD signal to the parent, using the kill command. This signal informs the parent process to clean up the zombie process using the wait() system call. This signal is sent with the kill command. It is demonstrated as follows:

kill -s SIGCHLD pid

# **fork() in C**

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process. It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.  
Zero: Returned to the newly created child process.  
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

**Exec in C**

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed. More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().

The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process, we are just replacing a process with another process in exec. If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process. PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point. Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. execl
2. execle
3. execlp
4. execv
5. execve
6. execvp

It should be noted here that these functions have the same base exec followed by one or more letters. These are explained below:

e: It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

l: l is for the command line arguments passed a list to the function

p: p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

v: v is for the command line arguments. These are passed as an array of pointers to the function.

Syntaxes of exec family functions:

int execl(const char\* path, const char\* arg, …)  
int execlp(const char\* file, const char\* arg, …)  
int execle(const char\* path, const char\* arg, …, char\* const envp[])  
int execv(const char\* path, const char\* argv[])  
int execvp(const char\* file, const char\* argv[])  
int execvpe(const char\* file, const char\* argv[], char \*const envp[])

The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

**2. Design and Implementation**

After a command is entered, the following things are done:

1. Command is entered and if length is non-null, keep it in history.
2. Parsing : Parsing is the breaking up of commands into individual words and strings
3. Checking for special characters like pipes, etc is done
4. Checking if built-in commands are asked for.
5. If [pipes](https://www.geeksforgeeks.org/pipe-system-call/) are present, handling pipes.
6. Executing system commands and libraries by [forking](https://www.geeksforgeeks.org/fork-system-call/) a child and calling [execvp](https://www.geeksforgeeks.org/exec-family-of-functions-in-c/).
7. Printing current directory name and asking for next input.

For keeping history of commands, recovering history using arrow keys and handling autocomplete using the tab key, we will be using the readline library provided by GNU.

**2.1 Implementation**

To install the readline library, open the terminal window and write

sudo apt-get install libreadline-dev

It will ask for your password. Enter it. Press y in the next step.

* Printing the directory can be done using getcwd.
* Getting user name can be done by getenv(“USER”)
* Parsing can be done by using strsep(“”). It will separate words based on spaces. Always skip words with zero length to avoid storing of extra spaces.
* After parsing, check the list of built-in commands, and if present, execute it. If not, execute it as a system command. To check for built-in commands, store the commands in an array of character pointers, and compare all with strcmp().  
  Note: “cd” does not work natively using execvp, so it is a built-in command, executed with chdir().
* For executing a system command, a new child will be created and then by using the execvp, execute the command, and wait until it is finished.
* Detecting pipes can also be done by using strsep(“|”).To handle pipes, first separate the first part of the command from the second part. Then after parsing each part, call both parts in two separate new children, using execvp. Piping means passing the output of first command as the input of second command.
  1. Declare an integer array of size 2 for storing file descriptors. File descriptor 0 is for reading and 1 is for writing.
  2. Open a pipe using the pipe() function.
  3. Create two children.
  4. In child 1->
  5. Here the output has to be taken into the pipe.
  6. Copy file descriptor 1 to stdout.
  7. Close file descriptor 0.
  8. Execute the first command using execvp()
  9. In child 2->
  10. Here the input has to be taken from the pipe.
  11. Copy file descriptor 0 to stdin.
  12. Close file descriptor 1.
  13. Execute the second command using execvp()
  14. Wait for the two children to finish in the parent.

// C Program to design a shell in Linux

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<readline/readline.h>

#include<readline/history.h>

#include<sys/stat.h>

#define MAXCOM 1000 // max number of letters to be supported

#define MAXLIST 100 // max number of commands to be supported

// Clearing the shell using escape sequences

#define clear() printf("\033[H\033[J")

// Greeting shell during startup

void init\_shell()

{

clear();

printf("\n\n\n\nwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww");

printf("\n\n\n\t----Welcome to the shell----");

printf("\n\n\n\n\t-Press help for the instructions-");

printf("\n\n\n\nwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww");

char\* username = getenv("USER");

printf("\n\n\nUSER is: @%s\n\n", username);

printf("Please wait...\n\n");

sleep(5);

clear();

}

// Function to take input

int takeInput(char\* str)

{

char \*buf;

buf = readline("\n>>>");

if (strlen(buf) != 0) {

add\_history(buf);

strcpy(str, buf);

return 0;

} else {

return 1;

}

}

// Function to print Current Directory.

void printDir()

{

char cwd[1024];

getcwd(cwd, sizeof(cwd));

printf("\nDir: %s", cwd);

}

// Function where the system command is executed

void execArgs(char\*\* parsed)

{

// Forking a child

pid\_t pid = fork();

if (pid == -1) {

printf("\nFailed forking child..");

return;

} else if (pid == 0) {

if (execvp(parsed[0], parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

// Function where the piped system commands is executed

void execArgsPiped(char\*\* parsed, char\*\* parsedpipe)

{

// 0 is read end, 1 is write end

int pipefd[2];

pid\_t p1, p2;

if (pipe(pipefd) < 0) {

printf("\nPipe could not be initialized");

return;

}

p1 = fork();

if (p1 < 0) {

printf("\nCould not fork");

return;

}

if (p1 == 0) {

// Child 1 executing..

// It only needs to write at the write end

close(pipefd[0]);

dup2(pipefd[1], STDOUT\_FILENO);

close(pipefd[1]);

if (execvp(parsed[0], parsed) < 0) {

printf("\nCould not execute command 1..");

exit(0);

}

} else {

// Parent executing

p2 = fork();

if (p2 < 0) {

printf("\nCould not fork");

return;

}

// Child 2 executing..

// It only needs to read at the read end

if (p2 == 0) {

close(pipefd[1]);

dup2(pipefd[0], STDIN\_FILENO);

close(pipefd[0]);

if (execvp(parsedpipe[0], parsedpipe) < 0) {

printf("\nCould not execute command 2..");

exit(0);

}

} else {

// parent executing, waiting for two children

wait(NULL);

}

}

}

// Help command builtin

void openHelp()

{

clear();

puts("\n\t\t\t---Welcome to the shell help---\n"

"\n\n\nList of Commands supported:"

"\n\n\t>HELLO"

"\n\t>LS"

"\n\t>GREP"

"\n\t>MKDIR"

"\n\t>WC"

"\n\t>CMP"

"\n\t>UNDEAD"

"\n\t>EXIT"

"\n\n>all other general commands available in UNIX shell"

"\n>pipe handling"

"\n>improper space handling");

return;

}

void executels(char\*\* parsed){

pid\_t p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("/bin/ls",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

void executewc(char\*\* parsed){

pid\_t p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("/usr/bin/wc",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

void executecmp(char\*\* parsed){

pid\_t p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("/usr/bin/cmp",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

void executegrep(char\*\* parsed){

pid\_t p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("/bin/grep",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

void executemkdir(char\*\* parsed){

pid\_t p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("/bin/mkdir",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

void executezombie(char\*\* parsed){

int p1;

p1=fork();

if (p1 == -1) {

printf("\nFailed forking child..");

return;

} else if (p1 == 0) {

if (execv("./undead",parsed) < 0) {

printf("\nCould not execute command..");

}

exit(0);

} else {

// waiting for child to terminate

wait(NULL);

return;

}

}

// Function to execute builtin commands

int ownCmdHandler(char\*\* parsed)

{

int NoOfOwnCmds = 10, i, switchOwnArg = 0;

char\* ListOfOwnCmds[NoOfOwnCmds];

char\* username;

ListOfOwnCmds[0] = "EXIT";

ListOfOwnCmds[1] = "cd";

ListOfOwnCmds[2] = "help";

ListOfOwnCmds[3] = "HELLO";

ListOfOwnCmds[4] = "MKDIR";

ListOfOwnCmds[5] = "LS";

ListOfOwnCmds[6] = "WC";

ListOfOwnCmds[7] = "CMP";

ListOfOwnCmds[8] = "GREP";

ListOfOwnCmds[9] = "UNDEAD";

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

if (strcmp(parsed[0], ListOfOwnCmds[i]) == 0) {

switchOwnArg = i + 1;

break;

}

}

switch (switchOwnArg) {

case 1:

printf("\nGoodbye\n");

exit(0);

case 2:

chdir(parsed[1]);

return 1;

case 3:

openHelp();

return 1;

case 4:

username = getenv("USER");

printf("\nHello %s."

"\nPlease dont crash the system."

"\nHave a nice time.\n",

username);

return 1;

case 5:

printf("\n");

executemkdir(parsed);

return 1;

case 6:

printf("\n");

executels(parsed);

return 1;

case 7:

printf("\n");

executewc(parsed);

return 1;

case 8:

printf("\n");

executecmp(parsed);

return 1;

case 9:

printf("\n");

executegrep(parsed);

return 1;

case 10:

printf("\n");

executezombie(parsed);

return 1;

default:

break;

}

return 0;

}

// function for finding pipe

int parsePipe(char\* str, char\*\* strpiped)

{

int i;

for (i = 0; i < 2; i++) {

strpiped[i] = strsep(&str, "|");

if (strpiped[i] == NULL)

break;

}

if (strpiped[1] == NULL)

return 0; // returns zero if no pipe is found.

else {

return 1;

}

}

// function for parsing command words

void parseSpace(char\* str, char\*\* parsed)

{

int i;

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

parsed[i] = strsep(&str, " ");

if (parsed[i] == NULL)

break;

if (strlen(parsed[i]) == 0)

i--;

}

}

int processString(char\* str, char\*\* parsed, char\*\* parsedpipe)

{

char\* strpiped[2];

int piped = 0;

int i;

piped = parsePipe(str, strpiped);

if (piped) {

parseSpace(strpiped[0], parsed);

parseSpace(strpiped[1], parsedpipe);

} else {

parseSpace(str, parsed);

}

if (piped==0){

if(ownCmdHandler(parsed))

return 0;

return 1;

}

else

return 1 + piped;

}

int main()

{

char inputString[MAXCOM], \*parsedArgs[MAXLIST];

char\* parsedArgsPiped[MAXLIST];

int execFlag = 0;

init\_shell();

while (1) {

// print shell line

printDir();

// take input

if (takeInput(inputString))

continue;

// process

execFlag = processString(inputString,

parsedArgs, parsedArgsPiped);

// execflag returns zero if there is no command

// or it is a builtin command,

// 1 if it is a simple command

// 2 if it is including a pipe.

// execute

if (execFlag == 1)

execArgs(parsedArgs);

if (execFlag == 2)

execArgsPiped(parsedArgs, parsedArgsPiped);

}

return 0;

}

For zombie command:

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<stdlib.h>

void main()

{

int p;

if((p=fork())==0)

{

exit(0);

}

else

{

printf("Zombie process has been successfully created.\n\n");

printf("Process status:\n\n");

system("ps -au");

printf("\n");

printf("Please wait....\n");

sleep(10);

}

}

**3. Testing /Result and Analysis**

Output of the above program is given below.

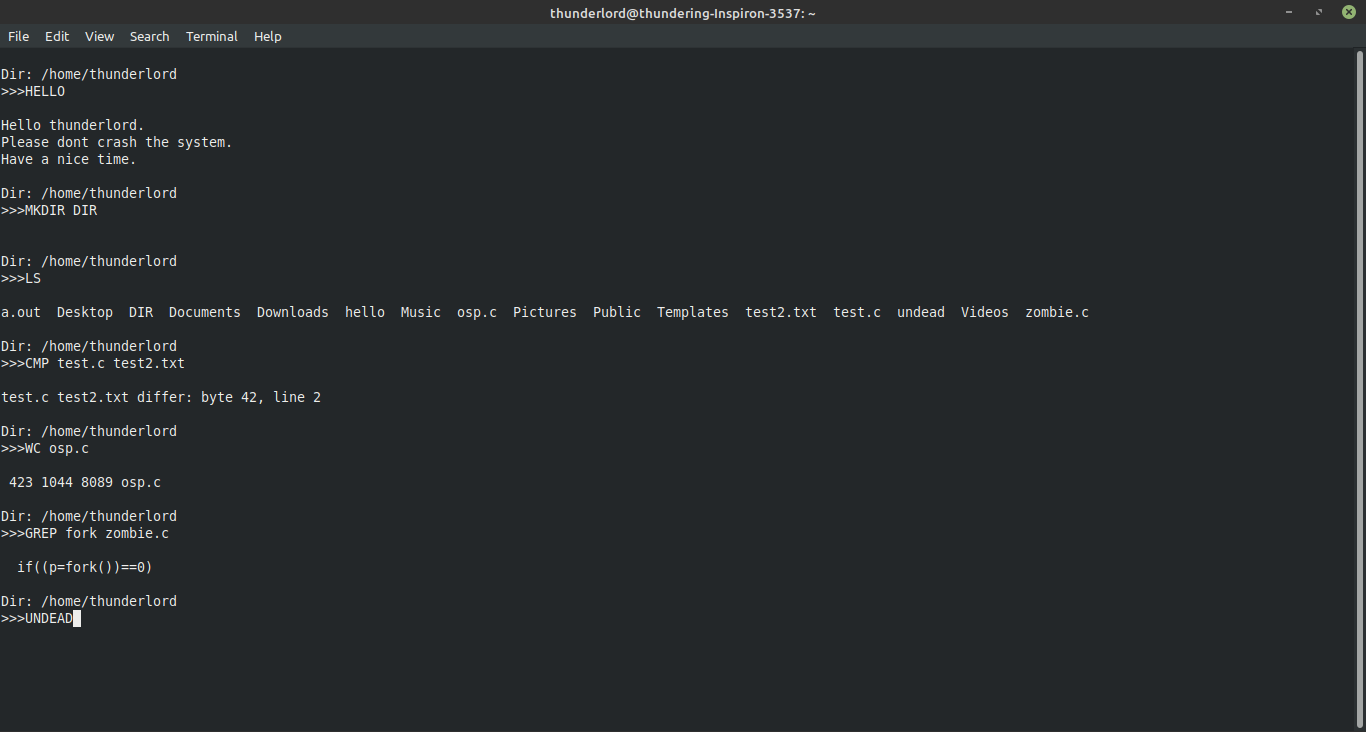
****

Fig. 3.0 Output of shell (1)

This Shell can run all the user defined commands as well as general commands of UNIX shell. Using fork and exec system calls we can implement any commands present in Unix shell.

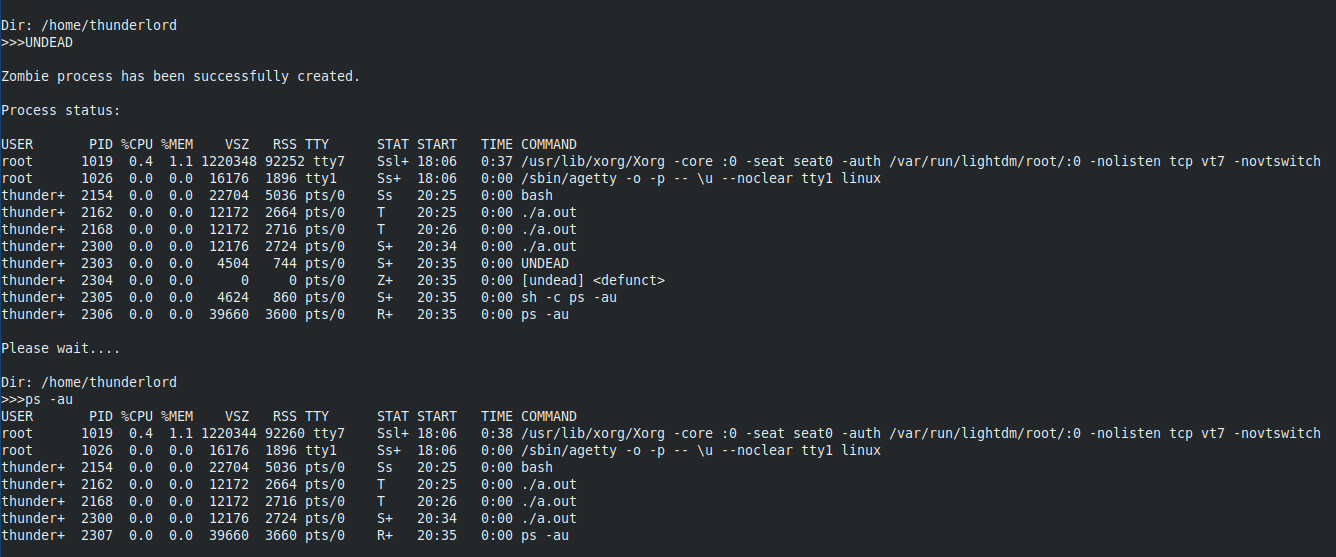


Fig. 3.1 Output of shell (2)

Zombie process can be terminated when its parent finishes execution. So, we have to write separate program for zombie process. Using fork and exec system calls we can implement a command for zombie process as shown above. We can see process with STAT Z is created which will be removed after parent complete execution.

**4. Conclusion and future enhancement**

Shell is implemented and results were analyzed. Shell can execute commands similar to ls, mkdir, grep, wc, cmp and a command to produce zombie process. It handles improper space alignments as well as all general command in UNIX shell. Currently piping is enabled for general commands of Unix system. Shell can be enhanced so that piping can be applied to own commands. Graphics and delay can be applied to make shell look attractive.

**5. References**

<https://www.geeksforgeeks.org/making-linux-shell-c/>

<https://www.google.com/>

<https://www.youtube.com/>