**IMPLEMENTATION OF NEW SHELL AND ITS FUNCTIONS**

J Component Project Report for the course

**OPERATING SYSTEMS CSE2005**

BY

AVULA ROHITHA 17BLC1007

K. MADHU SWAPNIKA 17BLC1024

AARTHI G 17BLC1166

A project report submitted to

**DR. MANAS RANJAN PRUSTY**

SCHOOL OF ELECTRONICS ENGINEERING

in partial fulfilment of the requirements for the course of

**OPERATING SYSTEMS CSE2005**



Vellore Institute of Technology (VIT) CHENNAI

Vandalur – Kelambakkam Road

Chennai – 600127

June 2020

**BONAFIDE CERTIFICATE**

Certified that this project report entitled “**IMPLEMENTATION OF NEW SHELL AND ITS FUNCTIONS”** is a bonafide work of **AVULA. ROHITHA (17BLC1007) , KONENI MADHU SWAPNIKA (17BLC1024)** and **AARTHI .G (17BLC1166)** who carried out the Project work under my supervision and guidance.

**DR. MANAS RANJAN PRUSTY**

Assistant Professor(sr.)

School of Computer Science and Engineering(SCSE)

VIT University, Chennai

Chennai – 600 127.

**ACKNOWLEDGEMENT**

We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Dr. Manas Ranjan Prusty,** Assistant Professor (Sr.), School of Computer Science and Engineering (SCSE, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to **Dr. Sivasubramanian A** Dean of the School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for her unstinting support.

We express our thanks to our Programme Chair**, Dr.Thiripurasundari D** for her support throughout the course of this project.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

**ABSTRACT**

The main purpose of this project is that we make our own shell script and include our own functions into it. We have used C code program for implementing shell and its functions. The main functions that we have coded are listed as below:

* cddir – Change Directory
* help – Lists down implemented functions
* wipe – clears the screen
* makedir – creates directory
* sysdate – displays current system data and time
* open – opens a file
* copy – copies a file
* delete – deletes a file
* deldir – deletes a particular directory
* rname – renames the file
* cwd – returns current working directory
* list – returns a list of all directories
* hist – returns command history
* file – creates a file
* exit – exits the shell
* virstat – virtual memory statistics
* cpu – display cpu information and architecture
* cpuusage – reports processor related statistics
* statio – input/output statistics for devices and partitions
* process – display user processes
* meminfo – shows RAM usage
* interr – shows number of interrupts
* dskmem – shows disk space usage
* iptraf – IP LAN monitor
* version – displays the version of OS
* memfrag – used to identify memory fragmentation issues
* iomem – system memory for every physical address
* locks – displays files locked by the kernel
* calender–displays the calendar of the current month
* factor–displays the factors of a number
* export–set environment variable
* unset–unset environment variable
* jobs–list all current jobs
* fg–put a job to foreground
* bg–put a job to background
* kill–kill a job

Implementing a self-created Unix Shell in C. By creating this shell, we can add various functions of our choice and make the shell as versatile as possible. The lifetime of a shell is quite simple and can be explained in three main terms:

* **Initialize:** In this step, a typical shell read and execute its configuration files. These changes aspects of the shell’s behavior.
* **Interpret:** Next, the shell reads commands from stdin (which could be interactive, or a file) and executes them.
* **Terminate:** After its commands are executed, the shell executes any shutdown commands, frees up any memory, and terminates.
* We are actually going to use them for the basis of our shell. Our shell won’t have any configuration files, and there won’t be any shutdown command. We can just call the looping command and terminate later on.
* The first few lines are just declarations and then using do-while looping because it is more convenient for checking status variable, we check the value once before executing.

**INTRODUCTION:**

If we are using any major operating system then we are indirectly interacting to shell. Ubuntu, Linux Mint or any other Linux distribution, interacts to shell every time we use terminal. Three important terminologies in shell scripting are:

* Kernel
* Shell
* Terminal

**Kernal:**

The kernel is a computer program that is the core of a computer’s operating system, with complete control over everything in the system. It manages following resources of the Linux system:

* + File management
  + Process management
  + I/O management
  + Memory management
  + Device management etc.

It is often mistaken that Linus Torvalds has developed Linux OS, but actually he is only responsible for development of Linux kernel.

Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

**Shell:**

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.

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. Working with command line shell is bit difficult for the beginners because it’s hard to memorize so many commands. It is very powerful; it allows user to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called batch files in Windows and Shell Scripts in Linux/macOS systems.

**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 actions.

**Shell Subsystems:**

**Environment Variables:** Expressions of the form ${VAR} are expanded with the corresponding environment variable. Also, the shell should be able to set, expand and print

environment vars.

**Wildcards:** Arguments of the form a\*a are expanded to all the files that match them in the local directory and in multiple directories.

**Subshells:** Arguments between `` (backticks) are executed and the output is sent as input to

the shell.

**Shell Scripting**

Usually shells are interactive that mean, they accept command as input from users and execute them. However, some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

* Shell Keywords – if, else, break etc.
* Shell commands – cd, ls, echo, pwd, touch etc.
* Functions
* Control flow – if..then..else, case and shell loops etc.

**Why do we need shell scripts?**

There are many reasons to write shell scripts –

* To avoid repetitive work and automation
* System admins use shell scripting for routine backups
* System monitoring
* Adding new functionality to the shell etc.

**Advantages of shell scripts:**

* The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
* Writing shell scripts are much quicker
* Quick start
* Interactive debugging etc.

**Disadvantages of shell scripts:**

* Prone to costly errors, a single mistake can change the command which might be harmful
* Slow execution speed
* Design flaws within the language syntax or implementation
* Not well suited for large and complex task
* Provide minimal data structure unlike other scripting languages. etc

**LITERATURE SURVEY / RELATED WORKS:**

The shell is a program where users can type commands. With the shell, it’s possible to invoke complicated programs like climate modelling software or simple commands that create an empty directory with only one line of code. The most popular Unix shell is Bash (the Bourne Again Shell — so-called because it’s derived from a shell written by Stephen Bourne). Bash is the default shell on most modern implementations of Unix and in most packages that provide Unix-like tools for Windows.

Using the shell will take some effort and some time to learn. While a GUI presents you with choices to select, CLI choices are not automatically presented to you, so you must learn a few commands like new vocabulary in a language you’re studying. However, unlike a spoken language, a small number of “words” (i.e. commands) gets you a long way, and we’ll cover those essential few today.

The grammar of a shell allows you to combine existing tools into powerful pipelines and handle large volumes of data automatically. Sequences of commands can be written into a script, improving the reproducibility of workflows.

In addition, the command line is often the easiest way to interact with remote machines and supercomputers. Familiarity with the shell is near essential to run a variety of specialized tools and resources including high-performance computing systems. As clusters and cloud computing systems become more popular for scientific data crunching, being able to interact with the shell is becoming a necessary skill. We can build on the command-line skills covered here to tackle a wide range of scientific questions and computational challenges.

**EXISTING WORK / SYSTEM:**

It is limited to the creation of shell without implementation of own os related functions.

**PROPOSED WORK / SYSTEM:**

To implement a new shell using C programming language. And to implement new OS related functions.

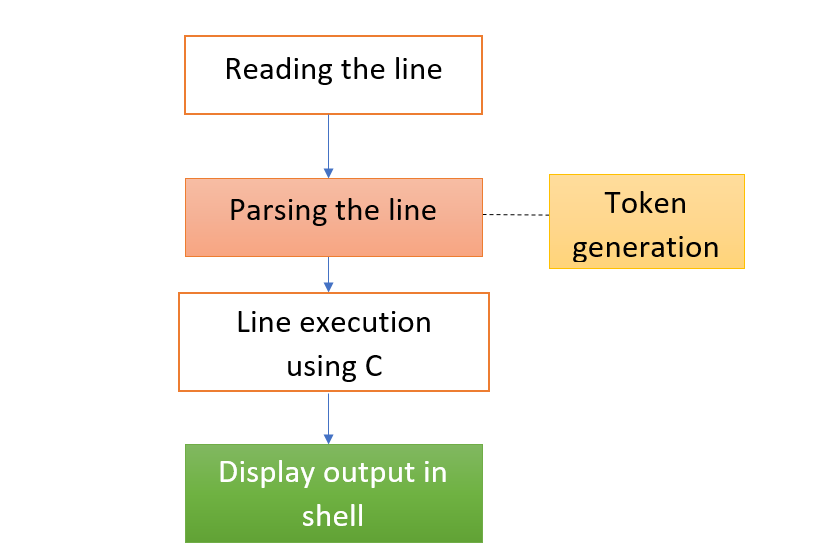
The various functions which are not available in the terminal are added in our shell in the form of C functions.

**WORKING MODULES AND FLOW CHART/ SYSTEM DESIGN OF PROPOSED WORK:**

**MODULES:**

1. **READING THE LINE**
2. **PARSING THE LINE**
3. **EXECUTING THE LINE**
4. **VALIDATING FUNCTIONS**

**ARCHITECTURE:**



**DESCRIPTION OF EACH MODULES WITH APPROPRIATE DIAGRAM AND EXPLANATION:**

1. **Reading the line**

You can’t simply allocate a block and hope they don’t exceed it. Instead, you need to start with a block, and if they do exceed it, reallocate with more space. This is a common strategy in C, and we’ll use it to implement lsh\_read\_line().

1. **Parsing the line**

Now, we need to parse that line into a list of arguments. We’re going to make a glaring simplification here, and say that we won’t allow quoting or backslash escaping in our command line arguments. Instead, we will simply use whitespace to separate arguments from each other. So, the command echo "this message" would not call echo with a single argument this message, but rather it would call echo with two arguments: "this and message".

1. **Executing the line**

There are only two ways of starting processes on Unix. The first one (which almost doesn’t count) is by being Init. You see, when a Unix computer boots, its kernel is loaded. Once it is loaded and initialized, the kernel starts only one process, which is called Init. This process runs for the entire length of time that the computer is on, and it manages loading up the rest of the processes that you need for your computer to be useful.

1. **Validating functions**

We are building terminal functions as well as customised functions scripted in C code. These functions also include individual help on what they perform and its corresponding syntax.

COMPLETE PROGRAM:

CODE-1:

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <a.out.h>

#include <time.h>

#include <dirent.h>

#include <fcntl.h>

char cwd[1024];

int check;

time\_t rawtime;

char c;

FILE \*fptr;

FILE \*source,\*target;

char ch;

int i=0;

static char history[4][25];

static unsigned history\_count = 0;

char lsfile[20]="file ";

void addhistory(char \*\*command )

{

strcpy(history[i],command[0]);

i++;

}

int lsh\_cddir(char \*\*args);

int lsh\_help(char \*\*args);

int lsh\_exit(char \*\*args);

int lsh\_makedir(char \*\*args);

int lsh\_sysdate(char \*\*args);

int lsh\_deldir(char \*\*args);

int lsh\_rname(char \*\*args);

int lsh\_cwd(char \*\*args);

int lsh\_list(char \*\*args);

int lsh\_hist(char \*\*args);

int lsh\_virstat(char \*\*args);

int lsh\_calender(char \*\*args);

int lsh\_dskmem(char \*\*args);

int lsh\_interr(char \*\*args);

char \*builtin\_str[] = {

"cddir",

"help",

"exit",

"makedir",

"sysdate",

"deldir",

"rname",

"cwd",

"list",

"hist",

"virstat",

"calender",

"dskmem",

"interr",

};

int (\*builtin\_func[]) (char \*\*) = {

&lsh\_cddir,

&lsh\_help,

&lsh\_exit,

&lsh\_makedir,

&lsh\_sysdate,

&lsh\_deldir,

&lsh\_rname,

&lsh\_cwd,

&lsh\_list,

&lsh\_hist,

&lsh\_virstat,

&lsh\_calender,

&lsh\_dskmem,

&lsh\_interr

};

int lsh\_num\_builtins()

{

return sizeof(builtin\_str) / sizeof(char \*);

}

int lsh\_interr(char \*\*args)

{

addhistory(args);

system("cat /proc/interrupts");

return 1;

}

int lsh\_dskmem(char \*\*args)

{

addhistory(args);

if( args[1]==NULL)

system("df");

else

printf("lsh:no argument required\n");

return 1;

}

int lsh\_virstat(char \*\*args)

{

addhistory(args);

if(args[1]==NULL)

system("vmstat");

else

printf("lsh:No argument required\n");

return 1;

}

int lsh\_hist(char \*\*args)

{

addhistory(args);

int j=i-1,k=0,m;

if(j>4)

{

for(k=0;k<4;k++)

{

m=i;

printf("[%d]%s \n",k+1,history[k]);

m++;

}

}

else

{

while(k<=j)

{

printf("[%d]%s \n",k+1,history[k]);

k++;

}

}

return 1;

}

int lsh\_list(char \*\*args)

{

addhistory( args );

int i=0,j=1;

struct dirent \*de;

DIR \*dr = opendir(".");

if(dr == NULL)

{

printf("Could not open current directory" );

return 0;

}

while((de = readdir(dr)) != NULL)

{

if (i<4)

{

printf("[%d]%s\t ", j++,de->d\_name);

i++;

}

else

{

i=0;

printf("\n [%d]%s\t ",j++,de->d\_name);

}

}

printf("\n");

closedir(dr);

return 1;

}

int lsh\_cwd(char \*\*args)

{

addhistory(args);

if(getcwd(cwd,sizeof(cwd))!=NULL)

printf("Current Working Directory: %s\n",cwd);

else

printf("error");

return 1;

}

int lsh\_rname(char \*\*args)

{

addhistory(args);

int i,fd1,fd2;

char \*file1,\*file2,buf[2];

file1=args[1];

file2=args[2];

printf("%s renamed to %s \n",file1,file2);

fd1=open(file1,O\_RDONLY,0777);

fd2=creat(file2,0777);

while(i=read(fd1,buf,1)>0)

write(fd2,buf,1);

remove(file1);

close(fd1);

close(fd2);

return 1;

}

int lsh\_deldir(char \*\*args)

{

addhistory(args);

int check;

char c;

if(args[1]!=NULL)

{

printf("Really want to delete[Y/N]\n");

c=getchar();

if(c=='Y'||c=='y')

{

check = rmdir(args[1]);

if(!check)

printf("Directory deleted : %s\n",args[1]);

else

{

printf("Unable to remove directory Or directory does not exist\n");

}

}

}

else

printf("Need an argument\n");

return 1;

}

int lsh\_makedir(char \*\*args)

{

addhistory(args);

if(args[1]==NULL)

{

printf("lsh:needs an argument\n");

}

else

{

check=mkdir(args[1]);

if(!check)

printf("Directory created\n");

else

printf ("Error creating directory\n");

}

return 1;

}

int lsh\_sysdate(char \*\*args)

{

addhistory(args);

if(args[1]==NULL)

{

struct tm \* timeinfo;

time ( &rawtime );

timeinfo = localtime ( &rawtime );

printf("Current local time and date: %s\n", asctime (timeinfo));

}

return 1;

}

int lsh\_cddir(char \*\*args)

{

addhistory(args);

if(args[1] == NULL)

{

fprintf(stderr, "lsh: expected argument to \"cddir\"\n");

}

else

{

if(chdir(args[1]) != 0)

{

perror("lsh");

}

else

{

if(getcwd(cwd, sizeof(cwd)) != NULL)

fprintf(stdout, "Current working dir: %s\n", cwd);

else

perror("getcwd() error");

}

}

return 1;

}

int lsh\_help(char \*\*args)

{

addhistory(args);

int i;

if(args[1]==NULL)

{

printf("Use the help command for information on functions as :\n help --<function\_name>\nfor the list of functions : use 'functions' command\n");

}

else if(strcmp(args[1],"--cddir")==0)

{

printf("cddir: changes the current working directory of the calling process to the directory specified in path.\n USAGE:- cddir <PATH>\n");

}

else if(strcmp(args[1], "--wipe")==0)

{

printf("wipe: clear the terminal screen.\n USAGE:- wipe \n" );

}

else if(strcmp(args[1],"--makedir")==0)

{

printf("makedir: create the DIRECTORY(ies), if they do not already exist.\n USAGE:- makedir <ARGUMENT>\n");

}

else if (strcmp(args[1],"--sysdate")==0)

{

printf("sysdate: print the system date and time\n USAGE: sysdate <NO ARGUMENT>\n");

}

else if (strcmp(args[1],"--open")==0)

{

printf("open: print on the standard output.\nUSAGE: open <FILENAME> \n");

}

else if (strcmp(args[1],"--copy")==0)

{

printf("copy: copy contents of one file to other. \nUSAGE: copy <FILE1><FILE2>\n");

}

else if (strcmp(args[1],"--delete")==0)

{

printf("delete: delete the file \nUSAGE: delete <FILENAME>\n");

}

else if (strcmp(args[1],"--rname")==0)

{

printf("rname: rename the file \nUSAGE: rname <FILENAME> \n");

}

else if (strcmp(args[1],"--cwd")==0)

{

printf("ced: prints the current working library\nUSAGE: cwd <NO ARGUMENTS> \n");

}

else if (strcmp(args[1],"--list")==0)

{

printf("list: it shows the directory contents \nUSAGE: list <NO ARGUMENTS> \n");

}

else if (strcmp(args[1],"--hist")==0)

{

printf("hist: prints the previous commands implemented in LSH\n hist<NO ARGUMENTS> \n");

}

else if (strcmp(args[1],"--file")==0)

{

printf("file: prints the type of the file \nUSAGE: file <FILENAME> \n");

}

else if (strcmp(args[1],"--vmstat")==0)

{

printf("vmstat: shows the virtua lmemory statistics \nUSAGE: vmstat /<NO ARGUMENT> \n");

}

else if (strcmp(args[1],"--factor")==0)

{

printf("factor: shows the factor of a number\nUSAGE: factor <NUMBER> \n");

}

else if (strcmp(args[1],"--cpu")==0)

{

printf("cpu: display information on CPU architecture\nUSAGE: cpu -a , cpu-i \n");

}

else if (strcmp(args[1],"--cpuusage")==0)

{

printf("cpuusage: report processors related statistics\nUSAGE: cpuusage <NO ARGS> \n");

}

else if (strcmp(args[1],"--statio")==0)

{

printf("statio: input/out‐put statistics for devices and partitions\nUSAGE:statio <NO ARGS> \n");

}

else if (strcmp(args[1],"--meminfo")==0)

{

printf("meminfo: shows the RAM usage\nUSAGE: meminfo <NO ARGUMENTS> \n");

}

else if (strcmp(args[1],"--process")==0)

{

printf("process: display User processes\nUSAGE: process <NO ARGUMENTS> \n");

}

else if (strcmp(args[1],"--interr")==0)

{

printf("interr: number of interrupts per x86 Archietecture \n USAGE:interr <NO ARGS>\n");

}

else if (strcmp(args[1],"--version")==0)

{

printf("version: shows the Linux version\nUSAGE: version \n");

}

else if (strcmp(args[1],"--dskmem")==0)

{

printf("dskmem:Shows the disk space Usage\nUSAGE: dskmem \n");

}

else if (strcmp(args[1],"--iptraf")==0)

{

printf("iptraf: IP LAN monitor \nUSAGE: iptraf <NO ARGS> \n");

}

return 1;

}

int lsh\_exit(char \*\*args)

{

addhistory(args);

return 0;

}

int lsh\_launch(char \*\*args)

{

pid\_t pid, wpid;

int status;

pid = fork();

if(pid == 0)

{

if (execvp(args[0], args) == -1)

{

perror("lsh");

}

exit(EXIT\_FAILURE);

}else if (pid < 0)

{

perror("lsh");

}

else

{

do

{

wpid = wait(pid, &status, WUNTRACED);

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

}

return 1;

}

int lsh\_execute(char \*\*args)

{

int i;

if(args[0] == NULL)

{

return 1;

}

for (i = 0; i < lsh\_num\_builtins(); i++)

{

if (strcmp(args[0], builtin\_str[i]) == 0)

{

return (\*builtin\_func[i])(args);

}

}

return lsh\_launch(args);

}

#define LSH\_RL\_BUFSIZE 1024

char \*lsh\_read\_line(void)

{

int bufsize = LSH\_RL\_BUFSIZE;

int position = 0;

char \*buffer = malloc(sizeof(char) \* bufsize);

int c;

if (!buffer)

{

fprintf(stderr, "lsh: allocation error\n");

exit(EXIT\_FAILURE);

}

while (1)

{

c = getchar();

if (c == EOF || c == '\n')

{

buffer[position] = '\0';

return buffer;

}

else

{

buffer[position] = c;

}

position++;

if (position >= bufsize)

{

bufsize += LSH\_RL\_BUFSIZE;

buffer = realloc(buffer, bufsize);

if (!buffer)

{

fprintf(stderr, "lsh: allocation error\n");

exit(EXIT\_FAILURE);

}

}

}

}

#define LSH\_TOK\_BUFSIZE 64

#define LSH\_TOK\_DELIM " \t\r\n\a"

char \*\*lsh\_split\_line(char \*line)

{

int bufsize = LSH\_TOK\_BUFSIZE, position = 0;

char \*\*tokens = malloc(bufsize \* sizeof(char\*));

char \*token;

if (!tokens)

{

fprintf(stderr, "lsh: allocation error\n");

exit(EXIT\_FAILURE);

}

token = strtok(line, LSH\_TOK\_DELIM);

while (token != NULL)

{

tokens[position] = token;

position++;

if (position >= bufsize)

{

bufsize += LSH\_TOK\_BUFSIZE;

tokens = realloc(tokens, bufsize \* sizeof(char\*));

if(!tokens)

{

fprintf(stderr, "lsh: allocation error\n");

exit(EXIT\_FAILURE);

}

}

token = strtok(NULL, LSH\_TOK\_DELIM);

}

tokens[position] = NULL;

return tokens;

}

void lsh\_loop(void)

{

char \*line;

char \*\*args;

int status;

do

{

printf("> ");

line = lsh\_read\_line();

args = lsh\_split\_line(line);

status = lsh\_execute(args);

free(line);

free(args);

}while (status);

}

int main(int argc, char \*\*argv)

{

lsh\_loop();

return EXIT\_SUCCESS;

}

CODE-2:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <signal.h>

#include <pwd.h>

#include <glob.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#define NR\_JOBS 20

#define PATH\_BUFSIZE 1024

#define COMMAND\_BUFSIZE 1024

#define TOKEN\_BUFSIZE 64

#define TOKEN\_DELIMITERS " \t\r\n\a"

#define BACKGROUND\_EXECUTION 0

#define FOREGROUND\_EXECUTION 1

#define PIPELINE\_EXECUTION 2

#define COMMAND\_EXTERNAL 0

#define COMMAND\_EXIT 1

#define COMMAND\_CD 2

#define COMMAND\_JOBS 3

#define COMMAND\_FG 4

#define COMMAND\_BG 5

#define COMMAND\_KILL 6

#define COMMAND\_EXPORT 7

#define COMMAND\_UNSET 8

#define STATUS\_RUNNING 0

#define STATUS\_DONE 1

#define STATUS\_SUSPENDED 2

#define STATUS\_CONTINUED 3

#define STATUS\_TERMINATED 4

#define PROC\_FILTER\_ALL 0

#define PROC\_FILTER\_DONE 1

#define PROC\_FILTER\_REMAINING 2

#define COLOR\_NONE "\033[m"

#define COLOR\_RED "\033[1;37;41m"

#define COLOR\_YELLOW "\033[1;33m"

#define COLOR\_CYAN "\033[0;36m"

#define COLOR\_GREEN "\033[0;32;32m"

#define COLOR\_GRAY "\033[1;30m"

const char\* STATUS\_STRING[] = {

"running",

"done",

"suspended",

"continued",

"terminated"

};

struct process {

char \*command;

int argc;

char \*\*argv;

char \*input\_path;

char \*output\_path;

pid\_t pid;

int type;

int status;

struct process \*next;

};

struct job {

int id;

struct process \*root;

char \*command;

pid\_t pgid;

int mode;

};

struct shell\_info {

char cur\_user[TOKEN\_BUFSIZE];

char cur\_dir[PATH\_BUFSIZE];

char pw\_dir[PATH\_BUFSIZE];

struct job \*jobs[NR\_JOBS + 1];

};

struct shell\_info \*shell;

int get\_job\_id\_by\_pid(int pid) {

int i;

struct process \*proc;

for (i = 1; i <= NR\_JOBS; i++) {

if (shell->jobs[i] != NULL) {

for (proc = shell->jobs[i]->root; proc != NULL; proc = proc->next) {

if (proc->pid == pid) {

return i;

}

}

}

}

return -1;

}

struct job\* get\_job\_by\_id(int id) {

if (id > NR\_JOBS) {

return NULL;

}

return shell->jobs[id];

}

int get\_pgid\_by\_job\_id(int id) {

struct job \*job = get\_job\_by\_id(id);

if (job == NULL) {

return -1;

}

return job->pgid;

}

int get\_proc\_count(int id, int filter) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

int count = 0;

struct process \*proc;

for (proc = shell->jobs[id]->root; proc != NULL; proc = proc->next) {

if (filter == PROC\_FILTER\_ALL ||

(filter == PROC\_FILTER\_DONE && proc->status == STATUS\_DONE) ||

(filter == PROC\_FILTER\_REMAINING && proc->status != STATUS\_DONE)) {

count++;

}

}

return count;

}

int get\_next\_job\_id() {

int i;

for (i = 1; i <= NR\_JOBS; i++) {

if (shell->jobs[i] == NULL) {

return i;

}

}

return -1;

}

int print\_processes\_of\_job(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

printf("[%d]", id);

struct process \*proc;

for (proc = shell->jobs[id]->root; proc != NULL; proc = proc->next) {

printf(" %d", proc->pid);

}

printf("\n");

return 0;

}

int print\_job\_status(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

printf("[%d]", id);

struct process \*proc;

for (proc = shell->jobs[id]->root; proc != NULL; proc = proc->next) {

printf("\t%d\t%s\t%s", proc->pid,

STATUS\_STRING[proc->status], proc->command);

if (proc->next != NULL) {

printf("|\n");

} else {

printf("\n");

}

}

return 0;

}

int release\_job(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

struct job \*job = shell->jobs[id];

struct process \*proc, \*tmp;

for (proc = job->root; proc != NULL; ) {

tmp = proc->next;

free(proc->command);

free(proc->argv);

free(proc->input\_path);

free(proc->output\_path);

free(proc);

proc = tmp;

}

free(job->command);

free(job);

return 0;

}

int insert\_job(struct job \*job) {

int id = get\_next\_job\_id();

if (id < 0) {

return -1;

}

job->id = id;

shell->jobs[id] = job;

return id;

}

int remove\_job(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

release\_job(id);

shell->jobs[id] = NULL;

return 0;

}

int is\_job\_completed(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return 0;

}

struct process \*proc;

for (proc = shell->jobs[id]->root; proc != NULL; proc = proc->next) {

if (proc->status != STATUS\_DONE) {

return 0;

}

}

return 1;

}

int set\_process\_status(int pid, int status) {

int i;

struct process \*proc;

for (i = 1; i <= NR\_JOBS; i++) {

if (shell->jobs[i] == NULL) {

continue;

}

for (proc = shell->jobs[i]->root; proc != NULL; proc = proc->next) {

if (proc->pid == pid) {

proc->status = status;

return 0;

}

}

}

return -1;

}

int set\_job\_status(int id, int status) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

int i;

struct process \*proc;

for (proc = shell->jobs[id]->root; proc != NULL; proc = proc->next) {

if (proc->status != STATUS\_DONE) {

proc->status = status;

}

}

return 0;

}

int wait\_for\_pid(int pid) {

int status = 0;

waitpid(pid, &status, WUNTRACED);

if (WIFEXITED(status)) {

set\_process\_status(pid, STATUS\_DONE);

} else if (WIFSIGNALED(status)) {

set\_process\_status(pid, STATUS\_TERMINATED);

} else if (WSTOPSIG(status)) {

status = -1;

set\_process\_status(pid, STATUS\_SUSPENDED);

}

return status;

}

int wait\_for\_job(int id) {

if (id > NR\_JOBS || shell->jobs[id] == NULL) {

return -1;

}

int proc\_count = get\_proc\_count(id, PROC\_FILTER\_REMAINING);

int wait\_pid = -1, wait\_count = 0;

int status = 0;

do {

wait\_pid = waitpid(-shell->jobs[id]->pgid, &status, WUNTRACED);

wait\_count++;

if (WIFEXITED(status)) {

set\_process\_status(wait\_pid, STATUS\_DONE);

} else if (WIFSIGNALED(status)) {

set\_process\_status(wait\_pid, STATUS\_TERMINATED);

} else if (WSTOPSIG(status)) {

status = -1;

set\_process\_status(wait\_pid, STATUS\_SUSPENDED);

if (wait\_count == proc\_count) {

print\_job\_status(id);

}

}

} while (wait\_count < proc\_count);

return status;

}

int get\_command\_type(char \*command) {

if (strcmp(command, "exit") == 0) {

return COMMAND\_EXIT;

} else if (strcmp(command, "cd") == 0) {

return COMMAND\_CD;

} else if (strcmp(command, "jobs") == 0) {

return COMMAND\_JOBS;

} else if (strcmp(command, "fg") == 0) {

return COMMAND\_FG;

} else if (strcmp(command, "bg") == 0) {

return COMMAND\_BG;

} else if (strcmp(command, "kill") == 0) {

return COMMAND\_KILL;

} else if (strcmp(command, "export") == 0) {

return COMMAND\_EXPORT;

} else if (strcmp(command, "unset") == 0) {

return COMMAND\_UNSET;

} else {

return COMMAND\_EXTERNAL;

}

}

char\* helper\_strtrim(char\* line) {

char \*head = line, \*tail = line + strlen(line);

while (\*head == ' ') {

head++;

}

while (\*tail == ' ') {

tail--;

}

\*(tail + 1) = '\0';

return head;

}

void mysh\_update\_cwd\_info() {

getcwd(shell->cur\_dir, sizeof(shell->cur\_dir));

}

int mysh\_cd(int argc, char\*\* argv) {

if (argc == 1) {

chdir(shell->pw\_dir);

mysh\_update\_cwd\_info();

return 0;

}

if (chdir(argv[1]) == 0) {

mysh\_update\_cwd\_info();

return 0;

} else {

printf("aarthi: cd %s: No such file or directory\n", argv[1]);

return 0;

}

}

int mysh\_jobs(int argc, char \*\*argv) {

int i;

for (i = 0; i < NR\_JOBS; i++) {

if (shell->jobs[i] != NULL) {

print\_job\_status(i);

}

}

return 0;

}

int mysh\_fg(int argc, char \*\*argv) {

if (argc < 2) {

printf("usage: fg <pid>\n");

return -1;

}

int status;

pid\_t pid;

int job\_id = -1;

if (argv[1][0] == '%') {

job\_id = atoi(argv[1] + 1);

pid = get\_pgid\_by\_job\_id(job\_id);

if (pid < 0) {

printf("aarthi: fg %s: no such job\n", argv[1]);

return -1;

}

} else {

pid = atoi(argv[1]);

}

if (kill(-pid, SIGCONT) < 0) {

printf("aarthi: fg %d: job not found\n", pid);

return -1;

}

tcsetpgrp(0, pid);

if (job\_id > 0) {

set\_job\_status(job\_id, STATUS\_CONTINUED);

print\_job\_status(job\_id);

if (wait\_for\_job(job\_id) >= 0) {

remove\_job(job\_id);

}

} else {

wait\_for\_pid(pid);

}

signal(SIGTTOU, SIG\_IGN);

tcsetpgrp(0, getpid());

signal(SIGTTOU, SIG\_DFL);

return 0;

}

int mysh\_bg(int argc, char \*\*argv) {

if (argc < 2) {

printf("usage: bg <pid>\n");

return -1;

}

pid\_t pid;

int job\_id = -1;

if (argv[1][0] == '%') {

job\_id = atoi(argv[1] + 1);

pid = get\_pgid\_by\_job\_id(job\_id);

if (pid < 0) {

printf("aarthi: bg %s: no such job\n", argv[1]);

return -1;

}

} else {

pid = atoi(argv[1]);

}

if (kill(-pid, SIGCONT) < 0) {

printf("aarthi: bg %d: job not found\n", pid);

return -1;

}

if (job\_id > 0) {

set\_job\_status(job\_id, STATUS\_CONTINUED);

print\_job\_status(job\_id);

}

return 0;

}

int mysh\_kill(int argc, char \*\*argv) {

if (argc < 2) {

printf("usage: kill <pid>\n");

return -1;

}

pid\_t pid;

int job\_id = -1;

if (argv[1][0] == '%') {

job\_id = atoi(argv[1] + 1);

pid = get\_pgid\_by\_job\_id(job\_id);

if (pid < 0) {

printf("aarthi: kill %s: no such job\n", argv[1]);

return -1;

}

pid = -pid;

} else {

pid = atoi(argv[1]);

}

if (kill(pid, SIGKILL) < 0) {

printf("aarthi: kill %d: job not found\n", pid);

return 0;

}

if (job\_id > 0) {

set\_job\_status(job\_id, STATUS\_TERMINATED);

print\_job\_status(job\_id);

if (wait\_for\_job(job\_id) >= 0) {

remove\_job(job\_id);

}

}

return 1;

}

int mysh\_export(int argc, char \*\*argv) {

if (argc < 2) {

printf("usage: export KEY=VALUE\n");

return -1;

}

return putenv(argv[1]);

}

int mysh\_unset(int argc, char \*\*argv) {

if (argc < 2) {

printf("usage: unset KEY\n");

return -1;

}

return unsetenv(argv[1]);

}

int mysh\_exit() {

printf("Goodbye!\n");

exit(0);

}

void check\_zombie() {

int status, pid;

while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED|WCONTINUED)) > 0) {

if (WIFEXITED(status)) {

set\_process\_status(pid, STATUS\_DONE);

} else if (WIFSTOPPED(status)) {

set\_process\_status(pid, STATUS\_SUSPENDED);

} else if (WIFCONTINUED(status)) {

set\_process\_status(pid, STATUS\_CONTINUED);

}

int job\_id = get\_job\_id\_by\_pid(pid);

if (job\_id > 0 && is\_job\_completed(job\_id)) {

print\_job\_status(job\_id);

remove\_job(job\_id);

}

}

}

void sigint\_handler(int signal) {

printf("\n");

}

int mysh\_execute\_builtin\_command(struct process \*proc) {

int status = 1;

switch (proc->type) {

case COMMAND\_EXIT:

mysh\_exit();

break;

case COMMAND\_CD:

mysh\_cd(proc->argc, proc->argv);

break;

case COMMAND\_JOBS:

mysh\_jobs(proc->argc, proc->argv);

break;

case COMMAND\_FG:

mysh\_fg(proc->argc, proc->argv);

break;

case COMMAND\_BG:

mysh\_bg(proc->argc, proc->argv);

break;

case COMMAND\_KILL:

mysh\_kill(proc->argc, proc->argv);

break;

case COMMAND\_EXPORT:

mysh\_export(proc->argc, proc->argv);

break;

case COMMAND\_UNSET:

mysh\_unset(proc->argc, proc->argv);

break;

default:

status = 0;

break;

}

return status;

}

int mysh\_launch\_process(struct job \*job, struct process \*proc, int in\_fd, int out\_fd, int mode) {

proc->status = STATUS\_RUNNING;

if (proc->type != COMMAND\_EXTERNAL && mysh\_execute\_builtin\_command(proc)) {

return 0;

}

pid\_t childpid;

int status = 0;

childpid = fork();

if (childpid < 0) {

return -1;

} else if (childpid == 0) {

signal(SIGINT, SIG\_DFL);

signal(SIGQUIT, SIG\_DFL);

signal(SIGTSTP, SIG\_DFL);

signal(SIGTTIN, SIG\_DFL);

signal(SIGTTOU, SIG\_DFL);

signal(SIGCHLD, SIG\_DFL);

proc->pid = getpid();

if (job->pgid > 0) {

setpgid(0, job->pgid);

} else {

job->pgid = proc->pid;

setpgid(0, job->pgid);

}

if (in\_fd != 0) {

dup2(in\_fd, 0);

close(in\_fd);

}

if (out\_fd != 1) {

dup2(out\_fd, 1);

close(out\_fd);

}

if (execvp(proc->argv[0], proc->argv) < 0) {

printf("aarthi: %s: command not found\n", proc->argv[0]);

exit(0);

}

exit(0);

} else {

proc->pid = childpid;

if (job->pgid > 0) {

setpgid(childpid, job->pgid);

} else {

job->pgid = proc->pid;

setpgid(childpid, job->pgid);

}

if (mode == FOREGROUND\_EXECUTION) {

tcsetpgrp(0, job->pgid);

status = wait\_for\_job(job->id);

signal(SIGTTOU, SIG\_IGN);

tcsetpgrp(0, getpid());

signal(SIGTTOU, SIG\_DFL);

}

}

return status;

}

int mysh\_launch\_job(struct job \*job) {

struct process \*proc;

int status = 0, in\_fd = 0, fd[2], job\_id = -1;

check\_zombie();

if (job->root->type == COMMAND\_EXTERNAL) {

job\_id = insert\_job(job);

}

for (proc = job->root; proc != NULL; proc = proc->next) {

if (proc == job->root && proc->input\_path != NULL) {

in\_fd = open(proc->input\_path, O\_RDONLY);

if (in\_fd < 0) {

printf("aarthi: no such file or directory: %s\n", proc->input\_path);

remove\_job(job\_id);

return -1;

}

}

if (proc->next != NULL) {

pipe(fd);

status = mysh\_launch\_process(job, proc, in\_fd, fd[1], PIPELINE\_EXECUTION);

close(fd[1]);

in\_fd = fd[0];

} else {

int out\_fd = 1;

if (proc->output\_path != NULL) {

out\_fd = open(proc->output\_path, O\_CREAT|O\_WRONLY, S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);

if (out\_fd < 0) {

out\_fd = 1;

}

}

status = mysh\_launch\_process(job, proc, in\_fd, out\_fd, job->mode);

}

}

if (job->root->type == COMMAND\_EXTERNAL) {

if (status >= 0 && job->mode == FOREGROUND\_EXECUTION) {

remove\_job(job\_id);

} else if (job->mode == BACKGROUND\_EXECUTION) {

print\_processes\_of\_job(job\_id);

}

}

return status;

}

struct process\* mysh\_parse\_command\_segment(char \*segment) {

int bufsize = TOKEN\_BUFSIZE;

int position = 0;

char \*command = strdup(segment);

char \*token;

char \*\*tokens = (char\*\*) malloc(bufsize \* sizeof(char\*));

if (!tokens) {

fprintf(stderr, "aarthi: allocation error\n");

exit(EXIT\_FAILURE);

}

token = strtok(segment, TOKEN\_DELIMITERS);

while (token != NULL) {

glob\_t glob\_buffer;

int glob\_count = 0;

if (strchr(token, '\*') != NULL || strchr(token, '?') != NULL) {

glob(token, 0, NULL, &glob\_buffer);

glob\_count = glob\_buffer.gl\_pathc;

}

if (position + glob\_count >= bufsize) {

bufsize += TOKEN\_BUFSIZE;

bufsize += glob\_count;

tokens = (char\*\*) realloc(tokens, bufsize \* sizeof(char\*));

if (!tokens) {

fprintf(stderr, "aarthi: allocation error\n");

exit(EXIT\_FAILURE);

}

}

if (glob\_count > 0) {

int i;

for (i = 0; i < glob\_count; i++) {

tokens[position++] = strdup(glob\_buffer.gl\_pathv[i]);

}

globfree(&glob\_buffer);

} else {

tokens[position] = token;

position++;

}

token = strtok(NULL, TOKEN\_DELIMITERS);

}

int i = 0, argc = 0;

char \*input\_path = NULL, \*output\_path = NULL;

while (i < position) {

if (tokens[i][0] == '<' || tokens[i][0] == '>') {

break;

}

i++;

}

argc = i;

for (; i < position; i++) {

if (tokens[i][0] == '<') {

if (strlen(tokens[i]) == 1) {

input\_path = (char \*) malloc((strlen(tokens[i + 1]) + 1) \* sizeof(char));

strcpy(input\_path, tokens[i + 1]);

i++;

} else {

input\_path = (char \*) malloc(strlen(tokens[i]) \* sizeof(char));

strcpy(input\_path, tokens[i] + 1);

}

} else if (tokens[i][0] == '>') {

if (strlen(tokens[i]) == 1) {

output\_path = (char \*) malloc((strlen(tokens[i + 1]) + 1) \* sizeof(char));

strcpy(output\_path, tokens[i + 1]);

i++;

} else {

output\_path = (char \*) malloc(strlen(tokens[i]) \* sizeof(char));

strcpy(output\_path, tokens[i] + 1);

}

} else {

break;

}

}

for (i = argc; i <= position; i++) {

tokens[i] = NULL;

}

struct process \*new\_proc = (struct process\*) malloc(sizeof(struct process));

new\_proc->command = command;

new\_proc->argv = tokens;

new\_proc->argc = argc;

new\_proc->input\_path = input\_path;

new\_proc->output\_path = output\_path;

new\_proc->pid = -1;

new\_proc->type = get\_command\_type(tokens[0]);

new\_proc->next = NULL;

return new\_proc;

}

struct job\* mysh\_parse\_command(char \*line) {

line = helper\_strtrim(line);

char \*command = strdup(line);

struct process \*root\_proc = NULL, \*proc = NULL;

char \*line\_cursor = line, \*c = line, \*seg;

int seg\_len = 0, mode = FOREGROUND\_EXECUTION;

if (line[strlen(line) - 1] == '&') {

mode = BACKGROUND\_EXECUTION;

line[strlen(line) - 1] = '\0';

}

while (1) {

if (\*c == '\0' || \*c == '|') {

seg = (char\*) malloc((seg\_len + 1) \* sizeof(char));

strncpy(seg, line\_cursor, seg\_len);

seg[seg\_len] = '\0';

struct process\* new\_proc = mysh\_parse\_command\_segment(seg);

if (!root\_proc) {

root\_proc = new\_proc;

proc = root\_proc;

} else {

proc->next = new\_proc;

proc = new\_proc;

}

if (\*c != '\0') {

line\_cursor = c;

while (\*(++line\_cursor) == ' ');

c = line\_cursor;

seg\_len = 0;

continue;

} else {

break;

}

} else {

seg\_len++;

c++;

}

}

struct job \*new\_job = (struct job\*) malloc(sizeof(struct job));

new\_job->root = root\_proc;

new\_job->command = command;

new\_job->pgid = -1;

new\_job->mode = mode;

return new\_job;

}

char\* mysh\_read\_line() {

int bufsize = COMMAND\_BUFSIZE;

int position = 0;

char \*buffer = malloc(sizeof(char) \* bufsize);

int c;

if (!buffer) {

fprintf(stderr, "aarthi: allocation error\n");

exit(EXIT\_FAILURE);

}

while (1) {

c = getchar();

if (c == EOF || c == '\n') {

buffer[position] = '\0';

return buffer;

} else {

buffer[position] = c;

}

position++;

if (position >= bufsize) {

bufsize += COMMAND\_BUFSIZE;

buffer = realloc(buffer, bufsize);

if (!buffer) {

fprintf(stderr, "aarthi: allocation error\n");

exit(EXIT\_FAILURE);

}

}

}

}

void mysh\_print\_promt() {

printf(COLOR\_GREEN "%s" COLOR\_NONE " in " COLOR\_YELLOW "%s" COLOR\_NONE "\n", shell->cur\_user, shell->cur\_dir);

printf(COLOR\_CYAN "aarthi>" COLOR\_NONE " ");

}

void mysh\_print\_welcome() {

printf("Welcome to my shell\n");

}

void mysh\_loop() {

char \*line;

struct job \*job;

int status = 1;

while (1) {

mysh\_print\_promt();

line = mysh\_read\_line();

if (strlen(line) == 0) {

check\_zombie();

continue;

}

job = mysh\_parse\_command(line);

status = mysh\_launch\_job(job);

}

}

void mysh\_init() {

struct sigaction sigint\_action = {

.sa\_handler = &sigint\_handler,

.sa\_flags = 0

};

sigemptyset(&sigint\_action.sa\_mask);

sigaction(SIGINT, &sigint\_action, NULL);

signal(SIGQUIT, SIG\_IGN);

signal(SIGTSTP, SIG\_IGN);

signal(SIGTTIN, SIG\_IGN);

pid\_t pid = getpid();

setpgid(pid, pid);

tcsetpgrp(0, pid);

shell = (struct shell\_info\*) malloc(sizeof(struct shell\_info));

getlogin\_r(shell->cur\_user, sizeof(shell->cur\_user));

struct passwd \*pw = getpwuid(getuid());

strcpy(shell->pw\_dir, pw->pw\_dir);

int i;

for (i = 0; i < NR\_JOBS; i++) {

shell->jobs[i] = NULL;

}

mysh\_update\_cwd\_info();

}

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

mysh\_init();

mysh\_print\_welcome();

mysh\_loop();

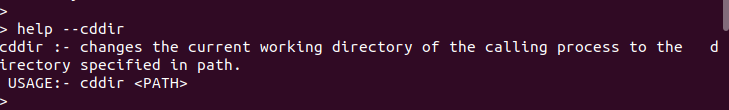
return EXIT\_SUCCESS;

}

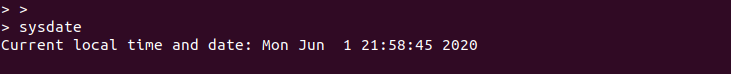
SCREEN SHOTS OF OUTPUT:

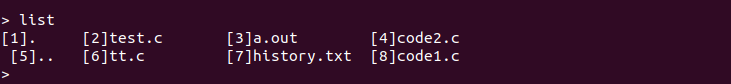


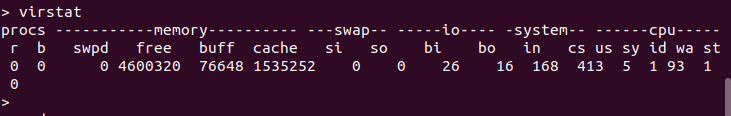


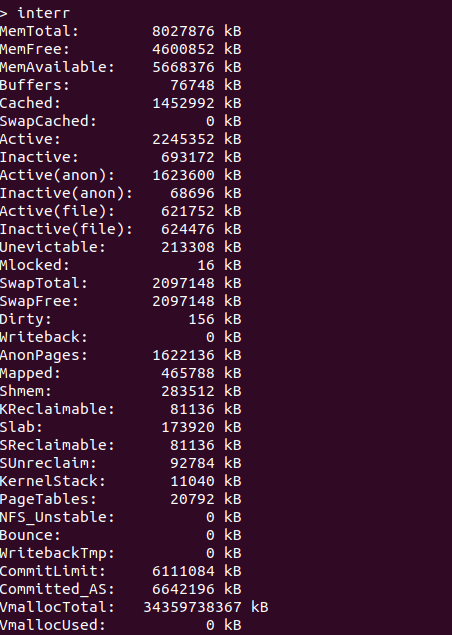


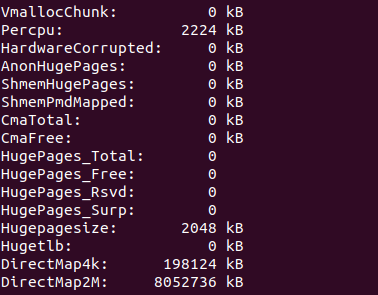


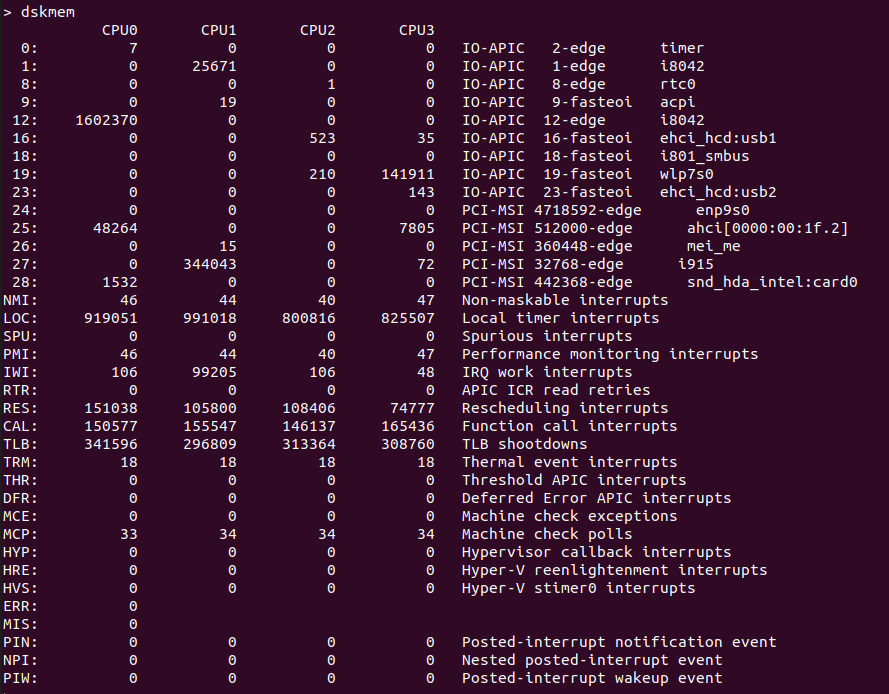


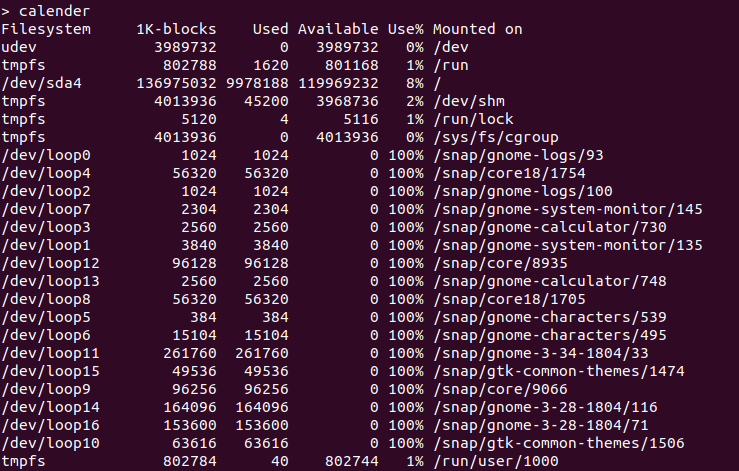




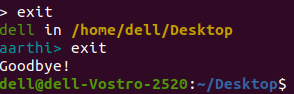












CONCLUSION:

Finally, we wrap this project and conclude by implementing all the customized terminal functions were scripted and executed successfully.

FUTURE WORK:

* Wider range of functions can be included in the shell
* Correction of mis-performing functions could be rectified
* Lines of code can be reduced by shifting to better performing languages like python, java, etc.

References:

* https://brennan.io/2015/01/16/write-a-shell-in-c/
* <http://pubs.opengroup.org/onlinepubs/9699919799/functions/getline.html>
* http://www.binarytides.com/category/distros/linux-mint/
* https://ss64.com/bash/
* http://brainstorms.in/pdf/commonds.pdf
* A book by O'Reilly, GNU CoreUtils