Assignment 08 | Operating System CE-092

Assignment submission for Operating System subject week 8.

nevilparmar24@gmail.com

Task 1:

Implement the linux shell in C.

This code can be viewed on the shared github gist file , If you are facing difficulty in reading such long code inside pdf.
https://gist.github.com/nevilparmar11/6c617636944a7cdb1774ed87343624ef

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <signal.h>
#include <ctype.h>

#define RL_BUFF_SIZE 1024
#define TK_BUFF_SIZE 64
#define TOK_DELIM " \t\r\n\a"
```

```c
char *clr[2] = {"clear", NULL};

//ANSI Color codes
#define RED          "\033[0;31m"
#define YELLOW       "\033[0;33m"
#define CYAN         "\033[0;36m"
#define GREEN        "\033[0;32m"
#define BLUE         "\033[0;34m"
#define INVERT       "\033[0;7m"
#define RESET        "\e[0m"
#define BOLD         "\e[1m"
#define ITALICS      "\e[3m"

// Function Prototypes
void history_input(char **, char *);
void pipe_history_input(char *);
void printtokens(char **);
void get_dir(char *);
void signalHandler();
int shell_cd(char **);
int shell_exit(char **);
int shell_help(char **);
int shell_grep(char **);
int shell_launch(char **);
int shell_execute(char **);
int history_line_count();
int shell_history();
int shell_pipe(char **);
int args_length(char **);
char **split_line(char *);
char *read_line();
char *trimws(char *);           //trim leading and
trailing whitespaces
char **split_pipes(char *);
char *get_hist_file_path();

/*
* Function Definition Begins Here
*/

// array of builtin function pointers
int (*builtin_funcs[])(char **) = {&shell_cd,
&shell_help, &shell_exit, &shell_history, &shell_grep,
&args_length };

// string array of builtin commands for strcmp() before
invoking execvp()
char *builtin_str[] = { "cd",  "help", "exit" ,
"history", "grep", "sizeof" };

// return the size of the builtin array
int builtin_funcs_count()
{
    return sizeof(builtin_str) / sizeof(char *);
}


void pipe_history_input(char *line)
{
    FILE *history_file = fopen(get_hist_file_path(),
"a+");
    fprintf(history_file, "%d. %s\n",
history_line_count(), line);
```

```
    fclose(history_file);
}

void history_input(char **args, char *d)
{
    FILE *history_file = fopen(get_hist_file_path(),
"a+");
    int j = 0;
    fprintf(history_file, "%d. ",
history_line_count());
    while(args[j] != NULL)
    {
        if(j > 0)
            fputs(d, history_file);
        fputs(args[j], history_file);
        j++;
    }
    fputs("\n", history_file);
    fclose(history_file);
}

char *trimws(char *str)
{
    char *end;
    while(isspace((unsigned char) *str)) str++;
    if(*str == 0)
        return str;
    end = str + strlen(str) - 1;
    while(end > str && isspace((unsigned char) *end))
end--;
    *(end+1) = 0;
    return str;
```

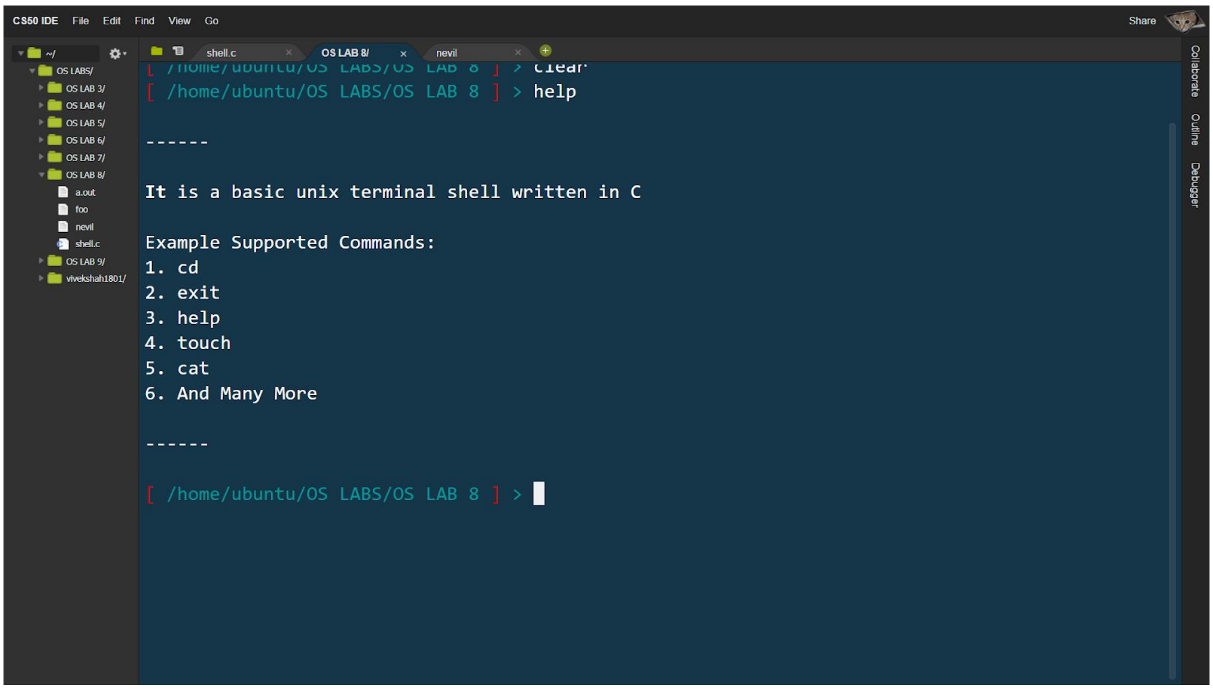
```
}


char **split_pipes(char *input)
{
    char *p = strtok(input, "|");
    char **s = malloc(1024*sizeof(char *));
    int i = 0;
    while(p != NULL)
    {

        s[i] = trimws(p);
        i++;
        p = strtok(NULL, "| ");
    }
    s[i] = NULL;
    i=0;
    while(s[i] != NULL)
    {
        i++;
    }
    return s;
}

int args_length(char **args)
{
    int i = 0;

    while(args[i] != NULL)
    {
        i++;
    }
    return i;
}

int shell_pipe(char **args)
{
    /*saving current stdin and stdout for restoring*/
    int tempin=dup(0);
    int tempout=dup(1);
    int j=0, i=0, flag=0;
    int fdin = 0, fdout;

    for(j =0; j<args_length(args); j++)
    {

        if(strcmp(args[j], "<") == 0)
        {
            fdin=open(args[j+1], O_RDONLY);
            flag += 2;
        }
    }

    if(!fdin)
        fdin=dup(tempin);
    int pid;
    for(i=0; i<args_length(args)-flag; i++)
    {
        char **rargs = split_line(args[i]);
        dup2(fdin, 0);
        close(fdin);
        if(i == args_length(args)-3 &&
strcmp(args[i+1], ">") == 0)
        {
            if((fdout = open(args[i+1], O_WRONLY)))
                i++;
        }
        else if(i == args_length(args)-flag-1)
            fdout = dup(tempout);
        else
        {
            int fd[2];
            pipe(fd);
            fdout = fd[1];
            fdin = fd[0];
        }

        dup2(fdout, 1);
        close(fdout);


        pid = fork();
        if(pid == 0)
        {
            execvp(rargs[0], rargs);
            perror("error forking\n");
            exit(EXIT_FAILURE);
        }

        wait(NULL);
    }

    dup2(tempin, 0);
    dup2(tempout, 1);
    close(tempin);
    close(tempout);
    return 1;
}

char *get_hist_file_path()
{
    static char file_path[128];
    strcat(strcpy(file_path, getenv("HOME")),
"/.shell_history");
    return file_path;
}

int shell_history()
{
    FILE *fp = fopen(get_hist_file_path(), "r");
    int ch, c, line_num = 1;
    char line[128];
    char prev_comm[128];
    char **args=NULL;
    if(!fp)
        fprintf(stderr, RED "shell: file not found"
RESET "\n");
    else
    {
        putchar('\n');
        while((c = getc(fp)) != EOF)
        {
            putchar(c);
        }
    }
    printf( "\n" INVERT " <0>: Quit    <#line>: Execute
command    <-1>: clear history" RESET "\n\n: ");
```

```
    scanf("%d", &ch);
    getchar();
    fseek(fp, 0, SEEK_SET);
    if(isdigit(ch) != 0)
    {
        printf("please enter a numerical choice\n");
    }
    else if (ch == 0)
    {
        fclose(fp);
        return 1;
    }
    else if(ch == -1)
    {
        fclose(fp);
        fp = fopen(get_hist_file_path(), "w");
        fclose(fp);
        return shell_execute(clr);
    }

    else
    {

        while((fgets(line, 128, fp)) != NULL)
        {
            if(line_num == ch)
            {


                strcpy(prev_comm, &line[3]);
                int p = 0, flag = 0;
                fclose(fp);
                while(prev_comm[p] != '\0')
                {
                    if(prev_comm[p] == '|')
                    {
                        flag = 1;
                        break;
                    }
                    p++;
                }
                if(!flag)
                {
                    args = split_line(prev_comm);
                    return shell_launch(args);
                }
                else
                {
                    args = split_pipes(prev_comm);
                    return shell_pipe(args);
                }

            }
            else
                line_num++;

        }
    }
    return 1;
}

int history_line_count()
{
    FILE *fp = fopen(get_hist_file_path(), "r");
    int c;
    int numOfLines = 1;
    do
    {
        c = getc(fp);
        if(c == '\n')
        {
            numOfLines++;
        }
    }while(c != EOF);
    return numOfLines;
}

void signalHandler()
{
    signal(SIGINT, signalHandler);
    getchar();
}

int shell_execute(char **args)
{
    pid_t cpid;
    int status;
    cpid = fork();

    if(cpid == 0)
    {
        if(execvp(args[0], args) < 0)
            printf("shell: command not found: %s\n",
args[0]);
        exit(EXIT_FAILURE);
    }
    else if(cpid < 0)
        printf(RED "Error forking" RESET "\n");
    else
    {
        waitpid(cpid, &status, WUNTRACED);
    }
    return 1;

}

int shell_launch(char **args)
{
    int i = 0;
    if(args[0] == NULL)
    {
        return 1;
    }
    else if(strcmp(args[0], "history") != 0 &&
strcmp(args[0], "exit") != 0 && strcmp(args[0],
"clear") != 0)       //excluding the history command
    {
        history_input(args, " ");
//storing cmds in history
    }
    for(i = 0; i<builtin_funcs_count(); i++)
    {
        if(strcmp(args[0], builtin_str[i]) == 0)
        {
            return (*builtin_funcs[i])(args);
        }
    }
    return shell_execute(args);

}

int shell_grep(char **args)
{
    FILE *fp = NULL;
    int flag = 0;
    char temp[512];
    int line_num = 1;
    if(args[0] != NULL && strcmp(args[0], "grep") == 0)
    {
        if(args[1] != NULL && args[2] != NULL)
        {
            fp = fopen(args[2], "r");
            while((fgets(temp, 512, fp)) != NULL)
            {
                if(strstr(temp, args[1]))
                {
                    printf("%d. %s", line_num, temp);
                    flag = 1;
                }
                line_num++;
            }
            fclose(fp);
        }
        else
        {
            fprintf(stderr, RED "shell: grep requires
two params, " ITALICS "PATTERN" RESET RED " and " RED
ITALICS "FILE" RESET "\n");
        }
```

```c
    }
    if(flag == 0)
        printf("No matches were found \n");
    return 1;
}

int shell_help(char **args)
{
    if(args[0] != NULL && strcmp(args[0], "help") == 0)
    {
        fprintf(stderr,"\n------\n"
                BOLD "\nIt " RESET "is a basic unix
terminal shell written in C \n"
                "\nExample Supported Commands:\n1.
cd\n2. exit\n3. help\n4. touch\n5. cat\n6. And Many
More"
                "\n\n------\n\n");
    }
    return 1;
}


int shell_exit(char **args)
{
    return 0;
}

void get_dir(char *state)
{
    char cwd[1024];
    if(getcwd(cwd, sizeof(cwd)) != NULL)
    {
        if(strcmp(state, "loop") == 0)
            printf(RED "[ " RESET CYAN "%s" RESET RED "
] " RESET, cwd);
        else if(strcmp(state, "pwd") == 0)
            printf("%s\n", cwd);
    }
    else
    {
        printf("%sgetcwd() error%s", RED, RESET);
    }
}


int shell_cd(char **args)
{
    if(args[1] == NULL)
    {
        fprintf(stderr, "%sShell: Please enter a path
to cd%s\n", YELLOW, RESET);
    }
    else
    {
        if(chdir(args[1]) > 0)
        {
            perror("Shell");
        }
    }
    return 1;
}

char **split_line(char *line)
{
    int buffsize = TK_BUFF_SIZE, position = 0;
    char **tokens = malloc(buffsize*sizeof(char*));
    char *token;

    if(!tokens)
    {
        fprintf(stderr, "%sShell: Allocation
error%s\n", RED, RESET);
        exit(EXIT_FAILURE);
    }
    token = strtok(line, TOK_DELIM);
    while(token != NULL)
    {
        tokens[position] = token;
        position++;

        if(position>=buffsize)
        {
            buffsize += TK_BUFF_SIZE;
            tokens = realloc(tokens,
buffsize*sizeof(char*));

            if(!tokens)
            {
                fprintf(stderr, "%sShell: Allocation
error%s\n", RED, RESET);
                exit(EXIT_FAILURE);
            }
        }

        token = strtok(NULL, TOK_DELIM);
    }
    tokens[position] = NULL;

    return tokens;
}

void printtokens(char **tokens)
{
    int i = 0;
    while(tokens[i] != NULL)
    {
        printf("%s\n", tokens[i]);
        i++;
    }
}

char *read_line()
{
    int buffsize = RL_BUFF_SIZE;
    int position = 0;
    char *buffer = malloc(sizeof(char) * buffsize);
    int c;

    if(!buffer)
    {
        fprintf(stderr, "%sShell: Allocation
error%s\n", RED, RESET);
        exit(EXIT_FAILURE);
    }

    while(1)
    {
        c  = getchar();
        if (c == EOF || c == '\n')
        {
            buffer[position] = '\0';
            return buffer;
        }
        else
        {
            buffer[position] = c;
        }
        position++;

        if (position >= buffsize)
        {
            printf("Overflow buffer....allocating more memory\n"); //test
            buffsize += RL_BUFF_SIZE;
            buffer = realloc(buffer, buffsize);

            if(!buffer)
            {
                fprintf(stderr, "%sShell: Allocation error%s\n", RED, RESET);
                exit(EXIT_FAILURE);
            }
        }
    }
}

void loop()
{
    char *line;
    char **args;
    int status=1, i = 0, flag = 0;


    do{
        get_dir("loop");
        printf(CYAN "> " RESET);
        line = read_line();
        flag = 0;
        i = 0;
        while(line[i] != '\0')
        {
            if(line[i] == '|')
            {
                flag = 1;
                break;
            }
            i++;
        }
        if(flag)
        {
                pipe_history_input(line);
                args = split_pipes(line);
                status = shell_pipe(args);
        }
        else
        {
            args = split_line(line);
            status = shell_launch(args);
        }
        free(line);
        free(args);
    }while(status);
}

int main(int argc, char **argv)
{
    // Contiue asking for commnad till the user
interrupt and exists the process
    loop();
    return EXIT_SUCCESS;
}
```

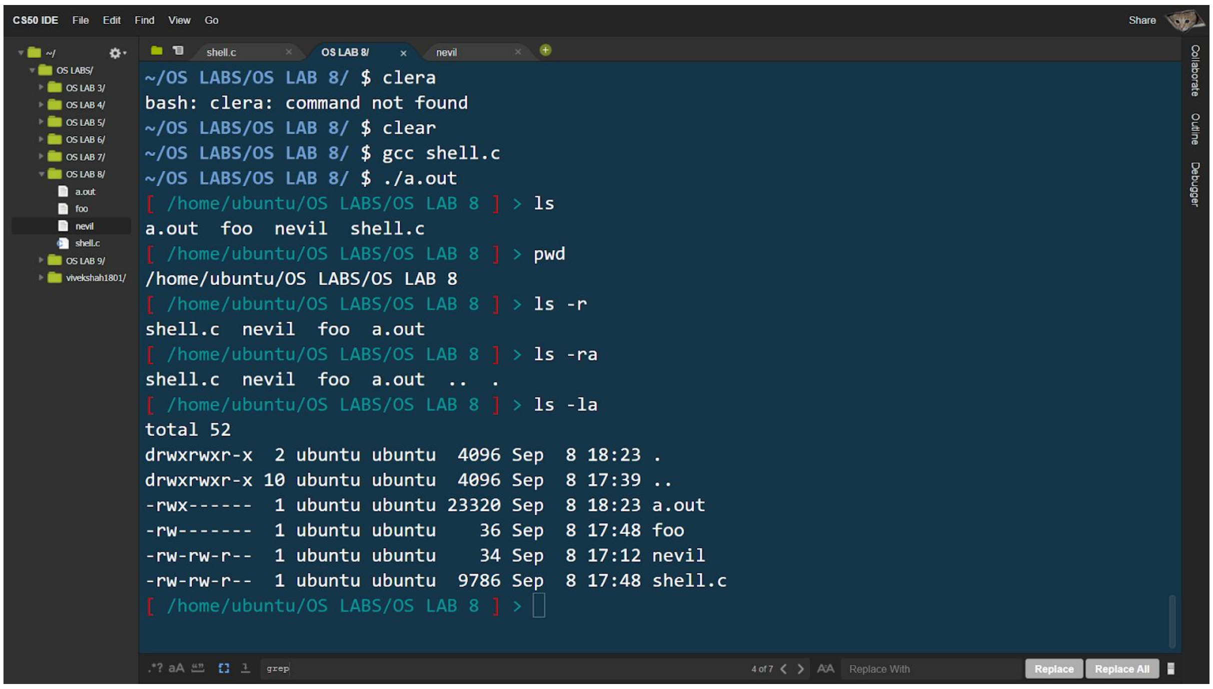
Output:

Let' s start by executing the first help command of the shell.



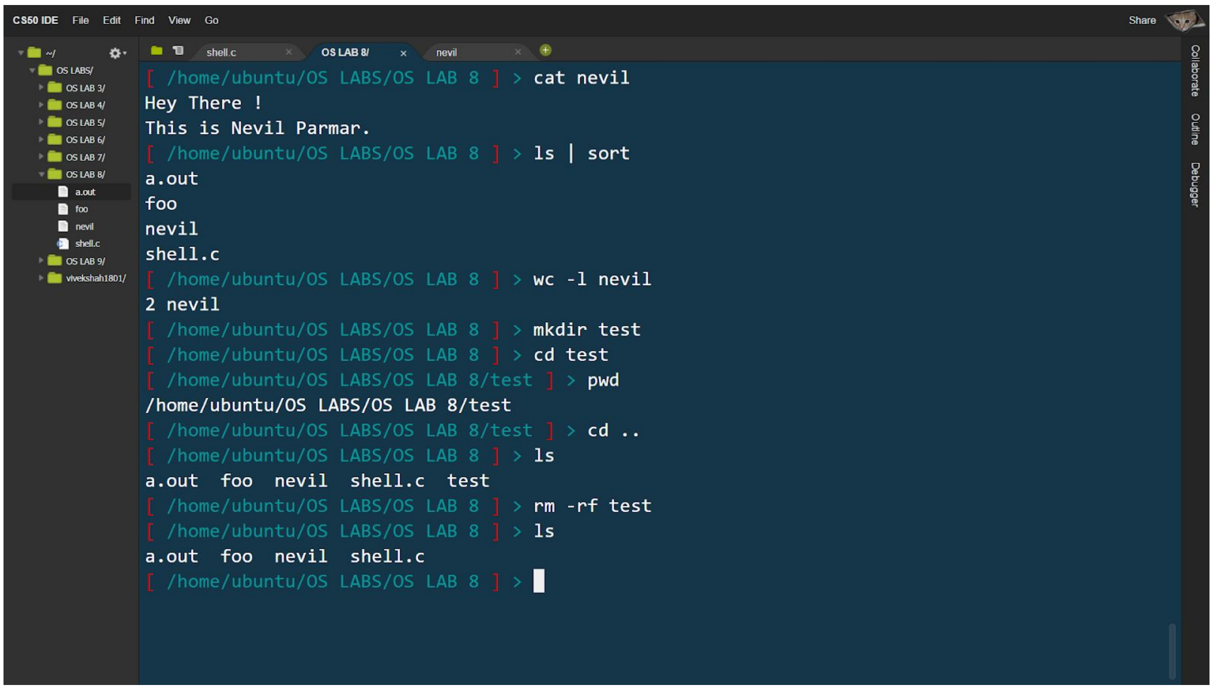
```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > clear
[ /home/ubuntu/OS LABS/OS LAB 8 ] > help

------

It is a basic unix terminal shell written in C

Example Supported Commands:
1. cd
2. exit
3. help
4. touch
5. cat
6. And Many More

------

[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Support for all the basic command which can be run on the terminal with /without any arguments



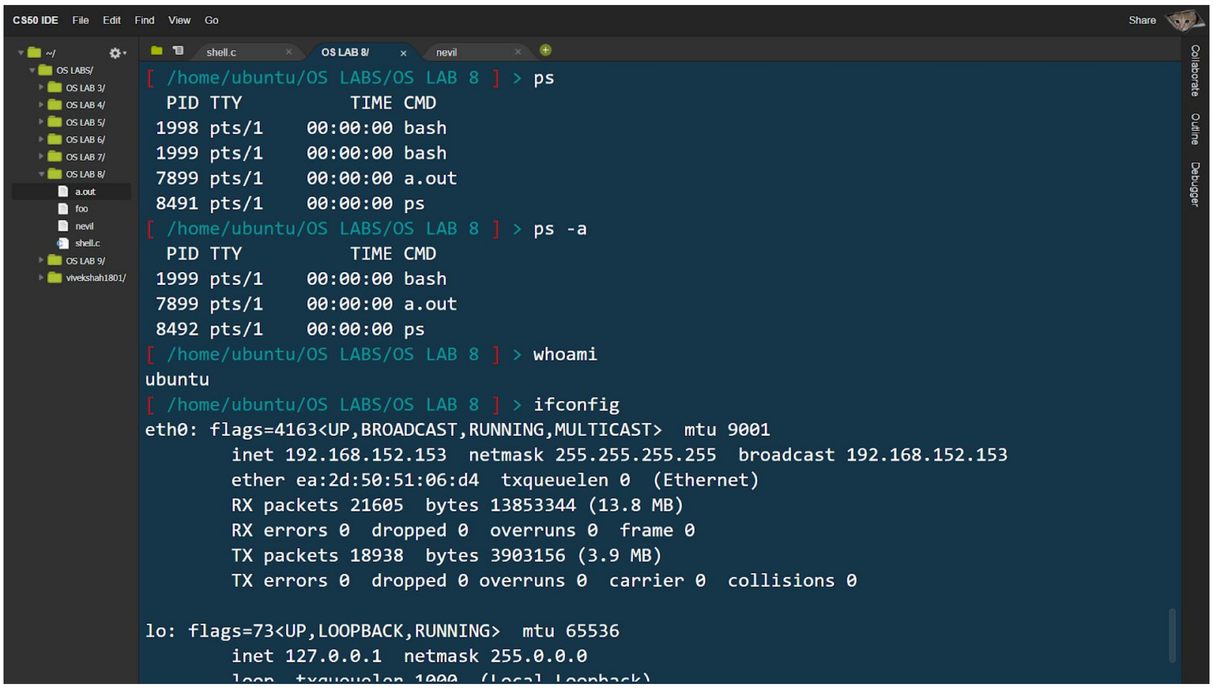
```
~/OS LABS/OS LAB 8/ $ clera
bash: clera: command not found
~/OS LABS/OS LAB 8/ $ clear
~/OS LABS/OS LAB 8/ $ gcc shell.c
~/OS LABS/OS LAB 8/ $ ./a.out
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls
a.out  foo  nevil  shell.c
[ /home/ubuntu/OS LABS/OS LAB 8 ] > pwd
/home/ubuntu/OS LABS/OS LAB 8
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls -r
shell.c  nevil  foo  a.out
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls -ra
shell.c  nevil  foo  a.out  ..  .
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls -la
total 52
drwxrwxr-x  2 ubuntu ubuntu  4096 Sep  8 18:23 .
drwxrwxr-x 10 ubuntu ubuntu  4096 Sep  8 17:39 ..
-rwx------  1 ubuntu ubuntu 23320 Sep  8 18:23 a.out
-rw-------  1 ubuntu ubuntu    36 Sep  8 17:48 foo
-rw-rw-r--  1 ubuntu ubuntu    34 Sep  8 17:12 nevil
-rw-rw-r--  1 ubuntu ubuntu  9786 Sep  8 17:48 shell.c
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Support for the | operator in shell.
It also supports multiple | in the command.

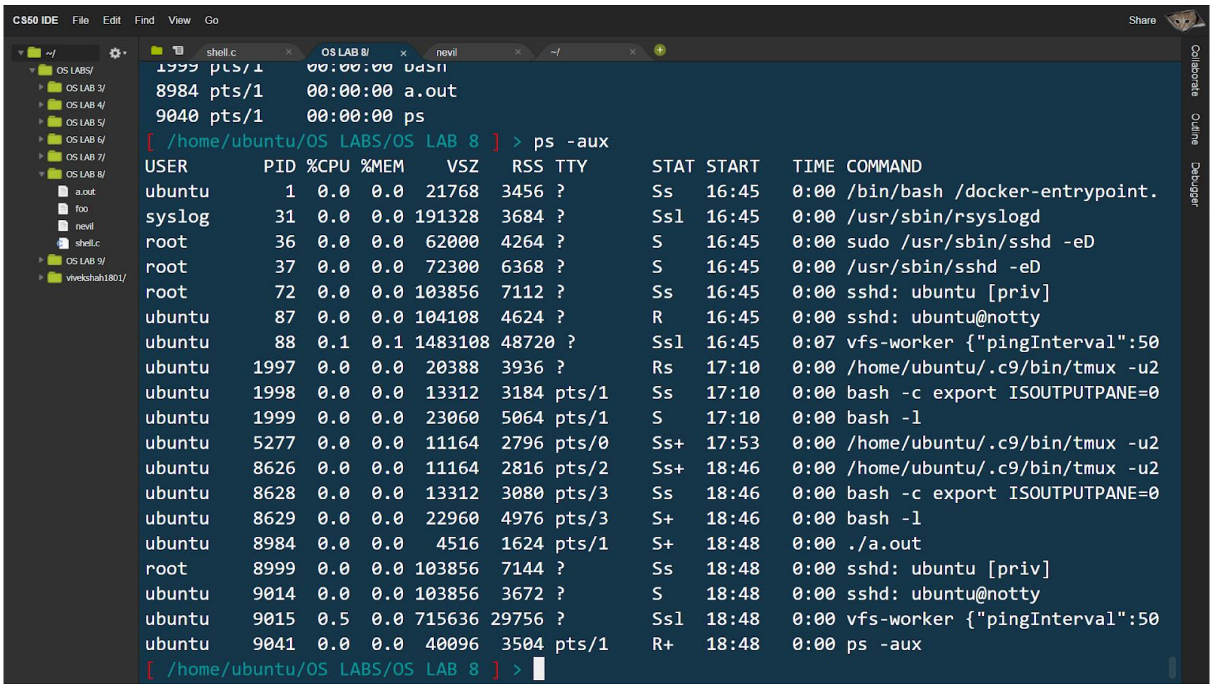


```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > cat nevil
Hey There !
This is Nevil Parmar.
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls | sort
a.out
foo
nevil
shell.c
[ /home/ubuntu/OS LABS/OS LAB 8 ] > wc -l nevil
2 nevil
[ /home/ubuntu/OS LABS/OS LAB 8 ] > mkdir test
[ /home/ubuntu/OS LABS/OS LAB 8 ] > cd test
[ /home/ubuntu/OS LABS/OS LAB 8/test ] > pwd
/home/ubuntu/OS LABS/OS LAB 8/test
[ /home/ubuntu/OS LABS/OS LAB 8/test ] > cd ..
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls
a.out  foo  nevil  shell.c  test
[ /home/ubuntu/OS LABS/OS LAB 8 ] > rm -rf test
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ls
a.out  foo  nevil  shell.c
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Few more examples to demonstrate the working of commonly used commands in linux.



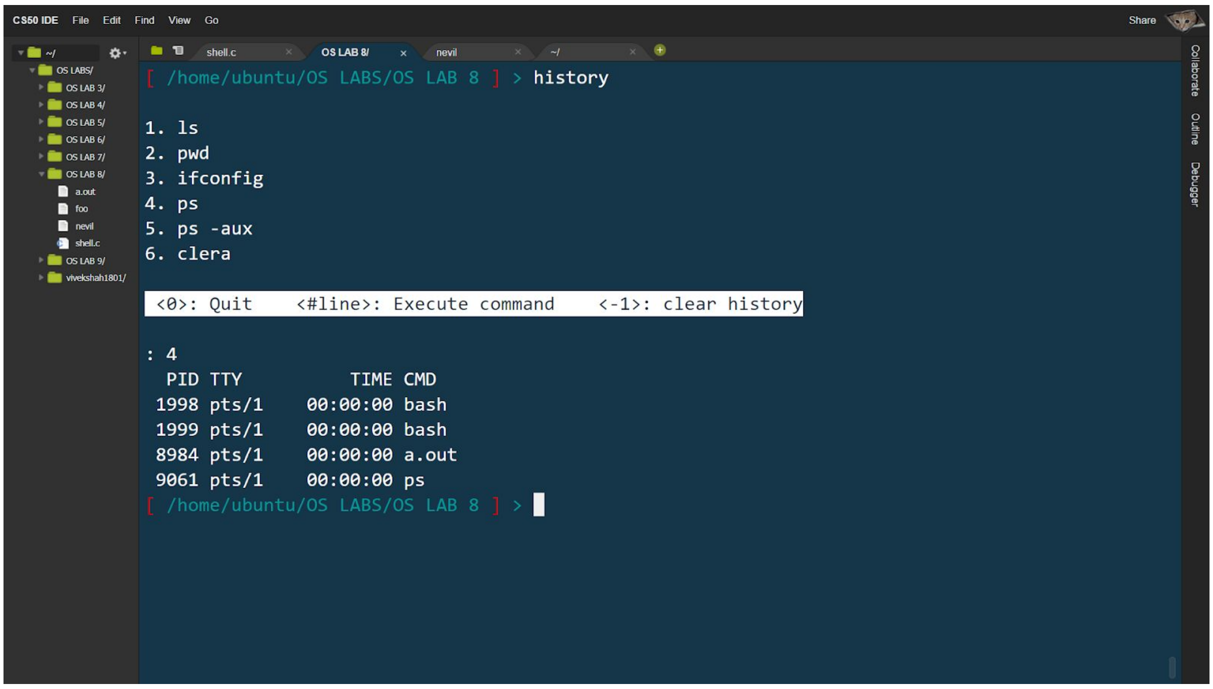
```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ps
  PID TTY          TIME CMD
 1998 pts/1    00:00:00 bash
 1999 pts/1    00:00:00 bash
 7899 pts/1    00:00:00 a.out
 8491 pts/1    00:00:00 ps
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ps -a
  PID TTY          TIME CMD
 1999 pts/1    00:00:00 bash
 7899 pts/1    00:00:00 a.out
 8492 pts/1    00:00:00 ps
[ /home/ubuntu/OS LABS/OS LAB 8 ] > whoami
ubuntu
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 192.168.152.153  netmask 255.255.255.255  broadcast 192.168.152.153
        ether ea:2d:50:51:06:d4  txqueuelen 0  (Ethernet)
        RX packets 21605  bytes 13853344 (13.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18938  bytes 3903156 (3.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
```



```
 8984 pts/1    00:00:00 a.out
 9040 pts/1    00:00:00 ps
[ /home/ubuntu/OS LABS/OS LAB 8 ] > ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ubuntu        1  0.0  0.0  21768  3456 ?        Ss   16:45   0:00 /bin/bash /docker-entrypoint.
syslog       31  0.0  0.0 191328  3684 ?        Ssl  16:45   0:00 /usr/sbin/rsyslogd
root         36  0.0  0.0  62000  4264 ?        S    16:45   0:00 sudo /usr/sbin/sshd -eD
root         37  0.0  0.0  72300  6368 ?        S    16:45   0:00 /usr/sbin/sshd -eD
root         72  0.0  0.0 103856  7112 ?        Ss   16:45   0:00 sshd: ubuntu [priv]
ubuntu       87  0.0  0.0 104108  4624 ?        R    16:45   0:00 sshd: ubuntu@notty
ubuntu       88  0.1  0.1 1483108 48720 ?       Ssl  16:45   0:07 vfs-worker {"pingInterval":50
ubuntu     1997  0.0  0.0  20388  3936 ?        Rs   17:10   0:00 /home/ubuntu/.c9/bin/tmux -u2
ubuntu     1998  0.0  0.0  13312  3184 pts/1    Ss   17:10   0:00 bash -c export ISOUTPUTPANE=0
ubuntu     1999  0.0  0.0  23060  5064 pts/1    S    17:10   0:00 bash -l
ubuntu     5277  0.0  0.0  11164  2796 pts/0    Ss+  17:53   0:00 /home/ubuntu/.c9/bin/tmux -u2
ubuntu     8626  0.0  0.0  11164  2816 pts/2    Ss+  18:46   0:00 /home/ubuntu/.c9/bin/tmux -u2
ubuntu     8628  0.0  0.0  13312  3080 pts/3    Ss   18:46   0:00 bash -c export ISOUTPUTPANE=0
ubuntu     8629  0.0  0.0  22960  4976 pts/3    S+   18:46   0:00 bash -l
ubuntu     8984  0.0  0.0   4516  1624 pts/1    S+   18:48   0:00 ./a.out
root       8999  0.0  0.0 103856  7144 ?        Ss   18:48   0:00 sshd: ubuntu [priv]
ubuntu     9014  0.0  0.0 103856  3672 ?        S    18:48   0:00 sshd: ubuntu@notty
ubuntu     9015  0.5  0.0 715636 29756 ?        Ssl  18:48   0:00 vfs-worker {"pingInterval":50
ubuntu     9041  0.0  0.0  40096  3504 pts/1    R+   18:48   0:00 ps -aux
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

We can even execute a history command in this shell.
It internally manages the history of the commands in the .bash_history text file, which is created inside the HOME folder of the system.

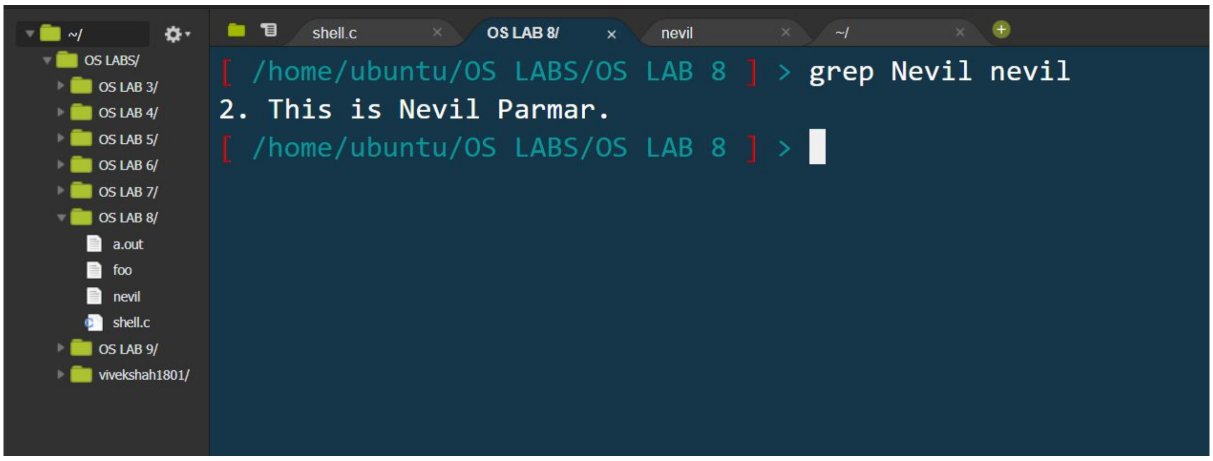
We can see here, **ps** command from the history is executed by giving **4** as an input.



```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > history

1. ls
2. pwd
3. ifconfig
4. ps
5. ps -aux
6. clera

 <0>: Quit    <#line>: Execute command    <-1>: clear history

: 4
  PID TTY          TIME CMD
 1998 pts/1    00:00:00 bash
 1999 pts/1    00:00:00 bash
 8984 pts/1    00:00:00 a.out
 9061 pts/1    00:00:00 ps
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Support for the Custom grep command.
Syntax :
Grep <Pattern> <FILE>

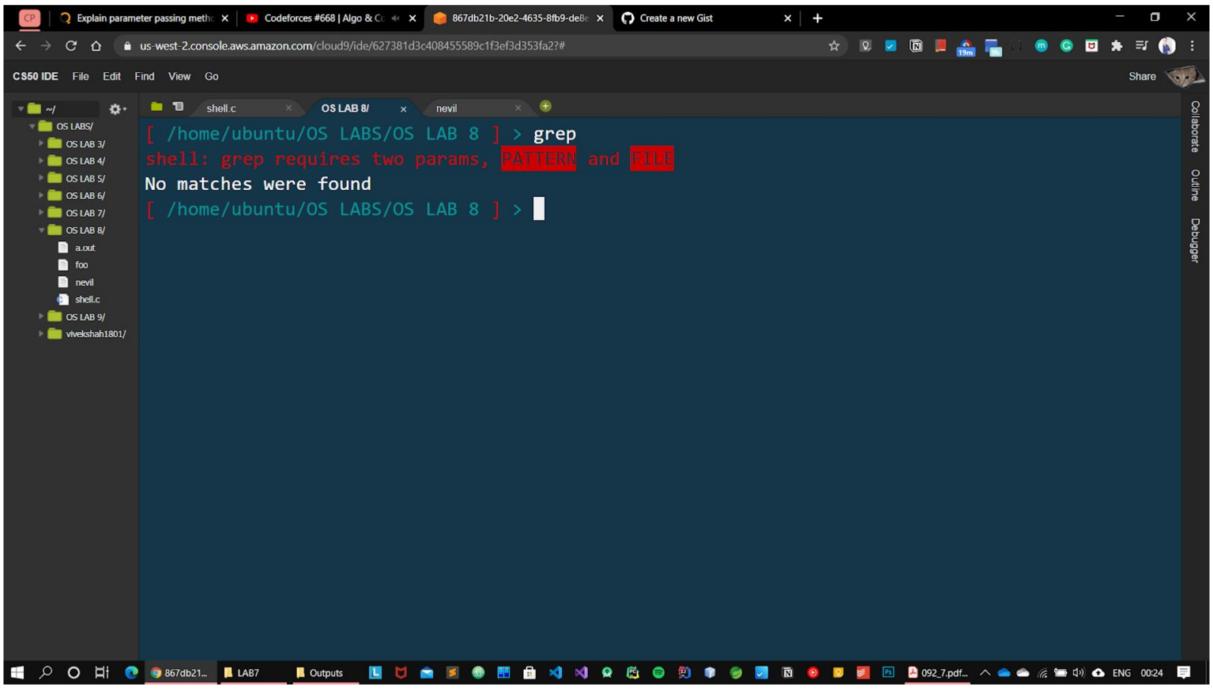
Here we are trying to find the pattern "**Nevil**" inside a file named "**nevil**".
And when it finds the matches for the pattern , it displays the whole line on the console.



```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > grep Nevil nevil
2. This is Nevil Parmar.
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Example to demonstrate the error handling portion of the code.
As mentioned above, we are accepting <PATTERN> and <FILE> for the grep command, which does not match with the below command in the screenshot and so it throws an error on the console.

Also most of the edge cases are covered in the code , but due to the length of the pdf and infeasibility I have mentioned only bullet points in the pdf.



```
[ /home/ubuntu/OS LABS/OS LAB 8 ] > grep
shell: grep requires two params, PATTERN and FILE
No matches were found
[ /home/ubuntu/OS LABS/OS LAB 8 ] >
```

Nevil Parmar
CE-092
https://nevilparmar.me