OPERATING SYSTEMS
CS250
ASSIGNMENT - II

PIYUSH PANCHOLI
12041050
piyushpancholi@iitbhilai.ac.in

REPORT :

The program made us create our own shell, we used the system calls fork() , execvp() , exec(), wait(), pipe() , exit() to make shell commands execute such as used fork and exec to execute ls command , furthermore for some direct commands like cd and exit we have used their primitive library chdir which is mentioned in the assignment itself.

The instructions to create the shell is given in the README.md file.

The structure of code is as follows :

> include all the standard libraries required
> predeclared functions
> all the functions needed to execute different commands such as :

Part A :

Int main()
This is the skeleton of the program which execute at first and activates an infinite time running loop to boost the shell .

removeWhiteSpace();
this function to avoid white space characters from the input command
(Referenced from https://github.com/csabagabor/Basic-Shell-implementation-in-C)

process_execute();
use of fork() and exec() to create child process from the parent and executing them

```c

int process_execute(char **args)
{
    // Forking a child
    pid_t pid = fork();
    if (pid == -1) {
        printf("\nFailed forking child..");
        return -1;
    } else if (pid == 0) {
        if (execvp(args[0], args) < 0) {
            printf("\nCould not execute command.. \n");
            return -1;
        }
        exit(0);
    } else {

        wait(NULL);
        return 1;
    }
}
```

execute_command();

this is main executing body of function which executes simple multi argument commands
Here we have defined some direct commands likewise exit , cd , help .and implemented a loop which first checks these commands one by one , if none of these presents then it by default executes the command.

take_command();
This function takes the input command and uses delimiters to parse the commands into individual commands ,it uses delimiters like space , \n , \t etc which are defined by char c .

tokenize_command();
It creates a array in which all the commands are stored in the form of tokens which are later executed one by one.it further removes whitespace characters from the input line.furthermore , i have used strtok() method is used to delimit the line by a delimiter.

command_split();
It takes the input line by user as input and splits the arguments by space , so that each argument is executed as per the shell. Here we have also used malloc() function to allocate memory for array .

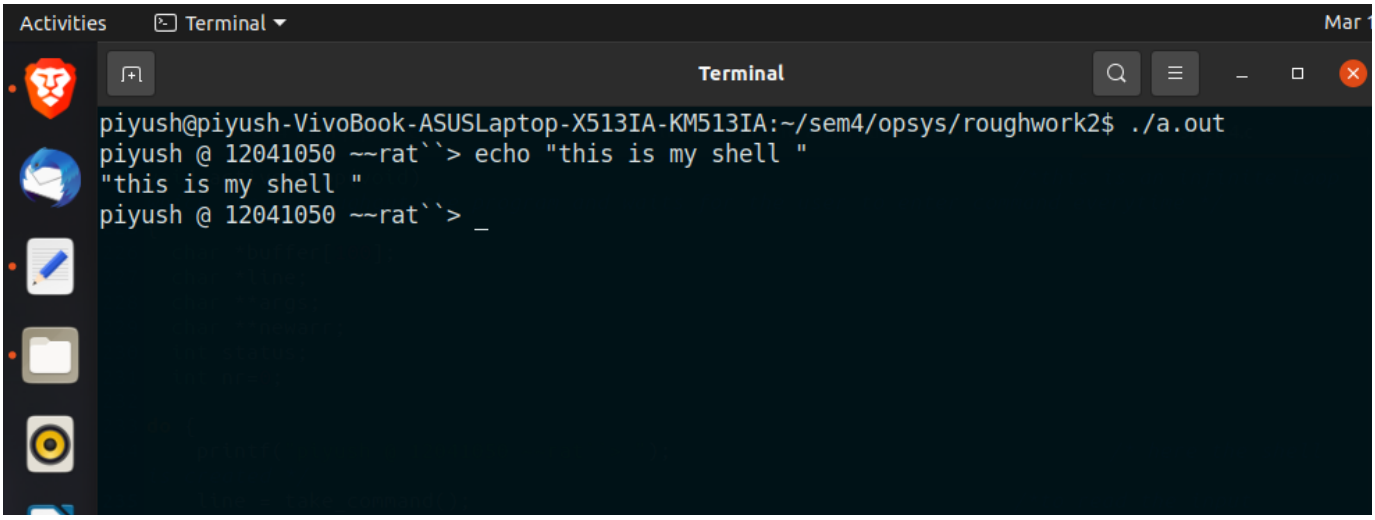
```
////////////////////////////////////////////////////////////////////////////////
                                                                    /* here we are using character c as delimiter for segmenting string into various small snippets of commands */
char **command_split(char *line, const char *c )
{
  int bufsize = 64, position = 0;
  char **tokens = malloc(bufsize * sizeof(char*));
  char *token, **tokens_backup;

                                                                    //strtok for splitting a string by some delimiter " " , such as space , pipe
        token = strtok(line,c);
        while (token != NULL) {
                tokens[position] = token;
                position++;

        if (position >= bufsize) {
                bufsize += 64;
                tokens_backup = tokens;
                tokens = realloc(tokens, bufsize * sizeof(char*));
        if (!tokens) {
                free(tokens_backup);
                fprintf(stderr, " allocation error\n");
                exit(EXIT_FAILURE);
        }}
        token = strtok(NULL,c);
        }
        tokens[position] = NULL;
        return tokens;
        }

```

active_loop();
This is an infinite loop which runs throughout the program and waits for the user to enter command every time. First we have initiated shell in this :



strchr(); functions checks if the character presents in the input line or not
This function just checks for single character
strpbrk(); this function used in && to check for multi character presence

We have implemented a block for redirection here , one block for and && here , one for piping , and else block for all the basic commands .

Part B :

do_pipe();
This function checks first checks the presence of pipe in the input string , then creates an array of commands separated by pipe , later on we have initiated the pipe for maximum 10 piped to be present in the fle descriptor array , we can increase the number of pipes by changing the size of file descriptor.

We have used pipe() system call : which takes the input and then pushes the output itself by default

```c
////////////////////////////////////////////////////////////////////////////////
                                                                    /* function to perform piping '|' in the input commands , it supports upto maximum 10 commands */
                                                                    /* user can increase the piping limit in the function descripter array [10][2]*/
void do_pipe(char** line,int nr){
        if(nr>10) return;

        int fd[10][2],i,pc;
        char *argv[100];

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

                tokenize_command(argv,&pc,line[i]," ");
                if(i!=nr-1){
                        if(pipe(fd[i])<0){
                                perror("pipe creating was not successfull\n");
                                return;
                        }
                }
                if(fork()==0){//child1
                        if(i!=nr-1){
                                dup2(fd[i][1],1);
                                close(fd[i][0]);
                                close(fd[i][1]);
                        }

                        if(i!=0){
                                dup2(fd[i-1][0],0);
                                close(fd[i-1][1]);
                                close(fd[i-1][0]);
                        }
                        execvp(argv[0],argv);
                        perror("invalid input ");
                        exit(1);                                    //exit when exec is not successful
                }
                //parent
                if(i!=0){//second process
                        close(fd[i-1][0]);
                        close(fd[i-1][1]);
                }
                wait(NULL);
        }}


```

&& and

I have implemented the && functionality of commands in the main function itself , and line by line execution is described in code itself .

```c

//program to run && in shell
        else if(strpbrk(line,"&&")){                        /*searches multichar && in the input */
                int i =0;                                   /*delimiter '&&' to separate into individual commands  */
                args=command_split(line,"&&");

                newarr=command_split(args[0]," ");
                status=execute_command(newarr);
         while (status != -1) {                             //it checks if the output of previous commands is -1 ,if its -1 the

                if (i<=sizeof(args)-1){                     // checks the size of array having commands separated by &&
                        newarr=command_split(args[i+1]," ");  // splits that command into individual commands
                        status=execute_command(newarr);     // executes each command using the same command execute function
                        i=i+1;
                        free(newarr);
                        }
                else{break;}
        }       }
```

Part C :

Redirection implementation :

\>

It has been mentioned in the code itself how redirection has been implemented . function to perform redirection '>' in the commands , only OUTPUT redirection is implemented as asked in assignment . we have used dup system call in this for execution .

```c
///////////////////////////////////////////////////////////////////////////
                                                                         /* function to perform
redirection '>' in the commands , only OUTPUT redirection is implemented as asked in assignment
*/
void Redirect_command(char** buf,int nr){
        int pc,fd;
        char *argv[100];
        removeWhiteSpace(buf[1]);
        tokenize_command(argv,&pc,buf[0]," ");
        if(fork()==0){
{               fd=open(buf[1],O_WRONLY);      }
                if(fd<0){
                        perror("cannot open file\n");
                        return; }
{               dup2(fd,1);             }
                execvp(argv[0],argv);
                perror("invalid input ");
                exit(1);                                                 //exit when exec is not
successful
        }
        wait(NULL);
}
```

/*****

The code for part C is the extended form of part A and Part B as mentioned in the assignment , so for the sake of reader convenience ,I have put the same file in place of A and B file.Hopefully this is proper and considerable for you .

*****/

Possibility for error :

Malloc size allocation is made keeping in mind the input contents to be small .

Redirection is not creating the file itself as it generally does in the actual shell , it was not demanded in the assignment so the file need to be created before inputting some data into it via redirection .

To create file use :

touch <filename>

To redirect data in file use :

Ls > <filename>

References:

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

https://www.albany.edu/~csi402/pdfs/handout_13.4.pdf