**NAME: ARKA DEBNATH**

**ROLL NO.: 001610501007**

**SECTION: A1**

**PROGRAM TO CREATE YOUR OWN SHELL**

The program creates a custom shell as per the following specifications:

**newdir [directory name] -** creates a new directory

**content [filename]-** Display contents of a file

**info [filename] -** Display owner name, last modified, size of a file

**editfile <optional filename> -** open the file in vi or open vi for new file

**exitjubcse -** Exit the shell

The arrays contain the names of the custom functions and their pointers respectively.



* **void sh\_loop()**

Function responsible for starting the shell-loop.

It gets the username from the environment variable “USER” and the current working directory using ***getcwd()***.



Then it prints the prompt inside a do-while loop until the EXIT\_CODE is reached. Each line of input is read and parsed by using the ***sh\_read\_line()*** and ***sh\_parse()*** functions respectively. Then the line is executed using ***sh\_execute()***.



* **char\* sh\_read\_line()**

Function responsible for reading each line of input

It makes use of ***getline()*** to take input and return it.



* **char\*\* sh\_parse(char\*)**

Function responsible for parsing the input.

It accepts a string, splits the input string using ***strtok(),*** and return an array of strings tokens[] such that tokens[0] is the command and the rest are arguments.

*//Parse the line*

char\*\* sh\_parse(char\* line)

{

int bufsize=TOK\_BUFFSIZE;

int position=0;

char\*\* tokens=malloc(bufsize\* **sizeof**(char\*)); *//Array to store all the tokens*

char\* token; *//Each token*

*//Check for memory allocation error*

**if**(!tokens)

{

printf("jubcseIII: allocation error");

exit(1);

}

*//Tokenize the line*

token=strtok(line, TOK\_DELIM);

**while**(token)

{

tokens[position++]=token; *// Assign each token to an element in the array of tokens*

*//Handle buffer overflow case*

**if**(position>=bufsize)

{

bufsize+=TOK\_BUFFSIZE;

tokens=realloc(tokens,bufsize\***sizeof**(char\*)); *//Reallocate memory with extended buffer*

*//Check for memory allocation error*

**if**(!tokens)

{

printf("jubcseIII: allocation error");

exit(1);

}

}

token = strtok(NULL,TOK\_DELIM);

}

tokens[position]=NULL;

**return** tokens;

}

* **int sh\_launch(char\*\*)**

This function is used to execute system commands.

In order that the current shell keeps on running even when another process is started we use the **fork()** function in **unistd.h** to create a child process and execute the command in the child process as now tow processes are running independently .

The function also checks for **&** for background processes and sets a background flag to 1.



**fork()** returns 0 to child process, -1 for fork failure and a non-zero value to parent process.

1. If the returned **pid\_t** is -1 then fork failure and appropriate message is displayed.
2. If the returned value is 0 i.e. we are in the child process, here, the current process image is replaced by another new process image. If any system built-in command is called or any of the custom command requires any built-in command; this function is called with the parsed string array as argument. The **int** **execvp(char\* file, char\* constargv[])** function does the task of starting the new process. The first argument is the filename, and the second argument is the entire array of arguments. **execvp()** returns negative for error in executing command.
3. If the returned value is non-zero, i.e. parent process then if the background flag is not set the parent process will wait for child process to finish, else it will not wait.

*//Function to start a builtin process*

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

{

*//Create a child process and call functions*

pid\_t pid=fork();

int background=0;

**if**(args[2]!=NULL && strcmp(args[2],"&")==0)

{

background=1;

args[2]=NULL;

}

**if**(background!=1 && args[1]!=NULL && strcmp(args[1],"&")==0)

{

background=1;

args[1]=NULL;

}

**if**(pid==-1)

{

*//In case of failure to fork a chaild*

printf("forking child failed**\n**");

**return** 0;

}

**else**

**if**(pid==0 && background==1)

{

*//If it is the child process and is background*

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

int x = open("/dev/null", O\_RDWR); *// Redirect input, output and stderr to /dev/null*

dup(x);

**if**(execvp(args[0],args)==-1)*//For invalid command*

{

printf("jubcseIII: no such file or command**\n**");

**return** 0;

}

kill(getpid(),SIGINT);

}

**else**

**if**(pid==0 && background!=1)

{

*//If it is the child process*

*//execute command*

**if**(execvp(args[0],args)==-1)*//For invalid command*

{

printf("jubcseIII: no such file or command**\n**");

**return** 0;

}

}

**else**

**if**(background!=1)

{

*//For parent process wait for child to terminate*

wait(NULL);

**return** 0;

}

**return** 1;

}

* **int sh\_launch\_custom(char\*\*,** **int (\*func)(char\*\*))**

This function is used to execute system commands.

In order that the current shell keeps on running even when another process is started we use the **fork()** function in **unistd.h** to create a child process and execute the command in the child process as now tow processes are running independently .

The function also checks for **&** for background processes and sets a background flag to 1.

**fork()** returns 0 to child process, -1 for fork failure and a non-zero value to parent process.

1. If the returned **pid\_t** is -1 then fork failure and appropriate message is displayed.
2. If the returned value is 0 i.e. we are in the child process, and call the function which is passed as function pointer.
3. If the returned value is non-zero, i.e. parent process then if the background flag is not set the parent process will wait for child process to finish, else it will not wait.

*//Function to start a custom process will take the fucntion pointer*

int sh\_launch\_custom(char\*\* args,int (\*func)(char\*\*))

{

*//Create a child process and call functions*

pid\_t pid=fork();

int background=(args[2]!=NULL && strcmp(args[2],"&")==0)?1:0; *//Sets the background flag*

background=(background!=1 && args[1]!=NULL && strcmp(args[1],"&")==0)?1:0;

**if**(pid==-1)

{

*//In case of failure to fork a chaild*

printf("forking child failed**\n**");

**return** 0;

}

**else**

**if**(pid==0 && background==1)

{

*//execute command*

**if**((\*func)(args)==0)*//For invalid command*

{

printf("jubcseIII: error executing command**\n**");

**return** 0;

}

}

**else**

**if**(pid==0 && background!=1)

{

*//If it is the child process*

*//execute command*

**if**(func(args)==0)*//For invalid command*

{

printf("jubcseIII: error executing command**\n**");

**return** 0;

}

}

**else**

**if**(background!=1)

{

*//For parent process wait for child to terminate*

wait(NULL);

**return** 0;

}

**return** 1;

}

* **int sh\_execute()**

Function for checking and calling appropriate function to execute commands.

1. If no command is entered return.
2. Otherwise check if this command is among the custom commands and call ***sh\_launch\_custom()*** the corresponding function pointer as argument.
3. If it is not a custom command call ***sh\_launch().***

*//Function to execute*

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

{

int i;

**if**(args[0]==NULL) *// no command entered*

**return** 1;

**else**

{

*//Comapare if the command is equal to any of the nuiltin commands*

**for**(int i=0;i<5;i++)

{

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

{

**if**(i==1)

**return** editfile(args);

**if**(i==4)

{

kill(0,SIGTERM);

exit(0);

}

**return** sh\_launch\_custom(args,builtin\_func[i]);

}

}

}

**return** sh\_launch(args);

}

* **int newdir(char\*\*)**

This is the custom function responsible for creating a new directory

1. It first checks if a valid argument is given to it.

**if**(args[1]==NULL) *//If no arguments are given*

{

printf("jubcseIII: newdir requires exactly one argument**\n**");

**return** 0;

}

1. If a directory name is supplied it checks if it exists and uses ***mkdir()*** function to create it if the directory does not exist.

*//Function to make a new directory*

int newdir(char\*\* args)

{

**if**(args[1]==NULL) *//If no arguments are given*

{

printf("jubcseIII: newdir requires exactly one argument**\n**");

**return** 0;

}

**else**

{

**if**(mkdir(args[1],0700)!=0)

{

printf("%s directory already exists**\n**",args[1]);

**return** 0;

}

}

printf("directory %s was created**\n**",args[1]);

**return** 1;

}

* **int editfile(char\*\*)**

1. This function calls the ***sh\_launch()*** after changing ***args[0]=”vim”.***

*//Function to edit a file*

int editfile(char\*\* args)

{

args[0]="vim";

*//As a new process will run we have to fork it so call launch*

**return** sh\_launch(args);

}

* **int content(char\*\*)**

1. At first it checks if valid number of arguments are given if not displays error message.
2. Open the file and check if it exists, if not display error.
3. If the file exists read the file line by line and print it.
4. Close the file

*//Function to print contents of a file*

int content(char\*\* args)

{

**if**(args[1]==NULL) *//If no arguments are given*

{

printf("jubcseIII: content requires exactly one argument**\n**");

**return** 0;

}

**else**

{

*//Read file and display it*

FILE \*fptr;

char\* filename,c;

filename=args[1];

*// Open file*

fptr = fopen(filename, "r");

**if** (fptr == NULL)

{

printf("jubcseII: no such file **\n**");

**return** 0;

}

*// Read contents from file*

c = fgetc(fptr);

**while** (c != EOF)

{

printf ("%c", c);

c = fgetc(fptr);

}

fclose(fptr);

printf("**\n**");

}

**return** 1;

}

* **int info(char\*\*)**

1. At first it checks if valid number of arguments are given if not displays error message.
2. Open the file and check if it exists, if not display error.
3. If the file exists then fetch the required info.
4. The **realpath(char\*,char\*)** returns the absolute path of the file
5. The **struct stat** returned by **stat()** is in **sys/stat.h** header file. The various members of the struct are used to get the info. **st\_size** gives the size of file in bytes, **st\_uid** gives owner of file, and **st\_mtime** gives the last modified date of file.
6. Close the file.

*//Function to print info about a file*

int info(char\*\* args)

{

**if**(args[1]==NULL) *//If no arguments are given*

{

printf("jubcseIII: info requires exactly one argument**\n**");

**return** 0;

}

**else**

{

*//If file does not exist*

FILE \*file;

**if** (!(file = fopen(args[1], "r")))

{

printf("info:file does not exist**\n**");

**return** 0;

}

fclose(file);

*//Get the required info about the file*

*//Get path of file*

char actualpath [PATH\_MAX+1];

char \*ptr;

ptr = realpath(args[1], actualpath);

*//Get size of file*

**struct** stat st;

stat(args[1], &st);

int size = st.st\_size;

*//Get last modified of a file*

char date[10];

strftime(date, 20, "%d-%m-%y", localtime(&(st.st\_mtime)));

*//Get owner of file*

uid\_t owner=st.st\_uid;

**struct** passwd \*pwd;

pwd = getpwuid(owner);

printf("Absolute path: %s**\n**",ptr);

printf("Size of file: %d bytes**\n**",size);

printf("Last Modified: %s**\n**",date);

printf("Owner: %s**\n**", pwd->pw\_name);

date[0]=0;

}

**return** 1;

}

* **int main(int, char\*[])**

The main function checks the time and displays an appropriate prompt, then call **sh\_loop().**

int main(int argc, char **const** \*argv[])

{

clear();

*//Display welcome message*

printf("================ Welcome to JUBCSEIII ===============**\n**");

printf("Hi! Good ");

time\_t currTime=time(NULL);

**struct** tm \*cuTime=localtime(&currTime);

int hour=cuTime->tm\_hour;

**if**(hour>=4 && hour<12)

printf("Morning**\n**");

**else**

**if**(hour>=12 && hour<=17)

printf("Afternoon**\n**");

**else**

**if**(hour>17 && hour<=23)

printf("Evening**\n**");

**else**

printf("Night**\n**");

printf("======================================================**\n**");

*// Call the shell loop*

sh\_loop();

**return** 0;

}