

NAME – SOULIB GHOSH

ROLL – 0016105010 47

GROUP – A2

SUBJECT – OS ASSIGNMENT 1 PROBLEM 6

JUBCSEIII linux shell

In the begining, the shell will display welcome message according to the time of the day (. time and localtime functions are used to obtain the time of the day)

```
time_t s;
struct tm* current_time;
s=time(NULL);
current_time=localtime(&s);
int hour=current_time->tm_hour;
```

The commands to be used by JUBCSEIII are:

1. **newdir directoryname:** creates a new directory
2. **editfile [filename]:** the file will be opened in vi editor if filename is given, otherwise the vi editor for a new file will be opened
3. **content filename:** prints the contents of a file on the screen
4. **info filename:** displays the following info of a file:
 - full path of the file
 - size of the file
 - last modification date
 - name of the creator
5. **exitbcse:** quits the shell

List of the functions are:

1. char* manual_commands[]={ "editfile", "newdir", "content", "info", "exitbcse" };
2. void shell_function(void);
3. int execute_func(char** args,int count);
4. int custom_command(char** args,int (*func)(char**,int),int count);
5. int system_command(char** args,int count);
6. int newdir(char** args,int count);
7. int editfile(char** args,int count);
8. int content(char** args,int count);
9. int info(char** args,int count);
10. int exitbcse(char** args,int count);
11. int (*manual_functions[]) (char**,int)={&editfile,&newdir,&content,&info,&exitbcse};

The details of the functions are given below.

1) void shell_function(void):

This function runs an infinite loop until `exitbcse` command is entered. This function prompts the user by `BCSEIII` and accepts input and tokenizes it using `strtok`, counts number of tokens and calls `execute_func` with the tokens.

```
void shell_function(void){
    while(1){
        printf("BCSE !!! ");
        char line[100];
        fgets(line,100,stdin);
        //printf("Input : %s\n",line);

        char** words=malloc(100*sizeof(char*));

        int count=0;
        char *token;
        token=strtok(line," \t\r\n");
        while(token!=NULL){
            words[count]=token;
            //printf("%d %s\n", count,words[count]);
            count++;
            token=strtok(NULL," \t\r\n");
        }

        words[count]=NULL;
        //for(int i=0;i<count;i++)
        //    printf("%d %s\n",i,words[i] );
        //clear the line
        int val=execute_func(words,count);
        if(val==EXITCODE) return;
        memset(line,0,sizeof(line));
    }
}
```

The arrays contain the name and pointers of the custom functions of `JUBCSEIII` respectively.

```
char* manual_commands[]={ "editfile", "newdir", "content", "info", "exitbcse" }

int (*manual_functions[]) (char**,int)={ &editfile, &newdir, &content, &info, &exitbcse }
```

2) int execute_func(char** args,int count):

This functions checks and calls appropriate function for a command. `count` denotes the number of tokens entered by the user. If no command is entered, it returns 0. Else it checks if the entered command is a custom command and it calls `custom_command`. Else it calls `system_command`.

```

int execute_func(char** args,int count){
    if(args[0]==NULL){
        return 0;
    }
    else{
        int i;
        //printf("%s\n",args[0]);
        for(i=0;i<5;i++){
            if(strcmp(args[0],manual_commands[i])==0){
                return custom_command(args,manual_functions[i],count);
            }
        }
        return system_command(args,count);
    }
    return EXITCODE;
}

```

3) int system_command(char** args,int count):

This function is to run the system commands. It checks for & to ensure whether it is a background job and sets the background flag as 1. We use the fork() function from unistd.h to create a child process and execute the command in the child process as now two processes are running independently . fork() returns 0 to child process, -1 for fork failure and a non-zero value to parent process.

- If fork() returns -1, then an appropriate message is shown and returned from there
- If pid is 0, then we are in child process. If any built in command is entered or any custom command is called which requires built in command call, this function is called with tokenized words and number of words entered. 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 -1 for error in executing command.
- If the return value is not 0, i.e., parent process. Then if the background flag is set, it will not wait for the child process to terminate else it will wait.

```

int system_command(char** args,int count){
    printf("inside system_command()\n");
    for(int i=0;i<count;i++){
        printf("%s\n",args[i] );
    }
    pid_t pid=fork();
    int background=0;
    if(args[count-1]!=NULL&&strcmp(args[count-1],"&")==0){
        background=1;
        args[count-1]=NULL;
    }
    else if(count>=2&&args[count-2]!=NULL&&strcmp(args[count-2],"&")==0){
        background=1;
        args[count-2]=NULL;
    }
    if(pid==-1){
        printf("forking failed\n");
    }
}

```

```

        return 0;
    }
    //printf(":%s:\n", args[0]);
    if(pid==0&&background==0){
        if(execvp(args[0],args)==-1)                //-1 if failure
        {
            printf("wrong input commands\n");
            return 0;
        }
    }
    else if(pid==0&&background){
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);
        int x = open("/dev/null", O_RDWR);
        dup(x);
        if(execvp(args[0],args)==-1)                //-1 if failure
        {
            printf("wrong input commands\n");
            return 0;
        }
        kill(getpid(),SIGINT);
    }
    else if(background!=1){
        wait(NULL);
        return 0;
    }
    return 1;
}

```

4) int custom_command(char** args,int (*func)(char**,int),int count):

This function is to execute custom commands. Along with the argument list and count, the pointer to the manual function to be executed is also passed as parameter. Work of this function is similar to the `system_command()` function.

```

int custom_command(char** args,int (*func)(char**,int),int count){
    pid_t pid=fork();
    int background=0;
    if(args[count-1]!=NULL&&strcmp(args[count-1],"&")==0){
        background=1;
        //args[count-1]=NULL;
    }
    else if(count>=2&&args[count-2]!=NULL&&strcmp(args[count-2],"&")==0){
        background=1;
        //args[count-2]=NULL;
    }
    if(pid==-1){
        printf("forking failed\n");
        return 0;
    }
    if(pid==0&&background==0){

```

```

        if(func(args,count)==-1){
            printf("wrong input commands\n");
            return 0;
        }
    }
    else if(pid==0&&background){
        if(func(args,count)==-1){
            printf("wrong input commands\n");
            return 0;
        }
    }
    else if(background!=1){
        wait(NULL);
        return 0;
    }
    return 1;
}

```

5) int editfile():

This function runs `system_command()` after replacing the `args[0]` with “vi”.

```

int editfile(char** args,int count){
    args[0]="gedit";
    return system_command(args,count);
}

```

6) int newdir():

This requires exactly one argument (directory name). The number of argument is checked first. The `mkdir()` function is called to create a directory with read and write permission.

```

int newdir(char** args,int count){
    if(count!=2){
        printf("newdir requires exactly one argument\n");
        return 1;
    }
    if(mkdir(args[1],0770)!=0){
        printf("directory already exists\n");
        return 1;
    }
    printf("directory created\n");
    return 0;
}

```

7) int content():

This method prints the contents of a file. First it checks the number of arguments and the existence of the file. Then it reads the file and prints its content in the screen.

```

int content(char** args,int count){

```

```

    if(count!=2){
        printf("newdir requires exactly one argument\n");
        return 1;
    }

    FILE *fptr;
    char* filename,c;
    filename=args[1];
    // Open file
    fptr = fopen(filename, "r");
    if (fptr == NULL){
        printf("cannot open %s \n",filename);
        return 0;
    }

    while ((c = fgetc(fptr)) != EOF){
        printf ("%c", c);
    }

    fclose(fptr);
    printf("\n");
    return 1;
}

```

8) int info():

This first checks the number of arguments passed and existence of the file. It prints the absolute path of the file using realpath(). The size, last modification date and owner of the file is obtained by using stat: st_size, st_mtime and st_uid.

```

int info(char** args,int count){
    if(count!=2){
        printf("newdir requires exactly one argument\n");
        return 1;
    }
    if(access(args[1],F_OK)==-1){
        printf("Does not exist\n");
        return 1;
    }
    char actualpath[300];
    char *ptr;
    ptr=realpath(args[1],actualpath);
    printf("Path : %s\n",ptr );

    struct stat st;
    stat(args[1],&st);
    int size=st.st_size;
    ptr=ctime(&st.st_mtime);

    printf("Size : %d\n", size);
    printf("Last access : %s",ptr);
}

```

```
    uid_t owner=st.st_uid;
    struct passwd *pwd;
    pwd = getpwuid(owner);
    printf("Owner : %s\n",pwd->pw_name);

    return 0;
}
```

9) int exitbcse():

This function uses kill() to end the running shell.

```
int exitbcse(char** args,int count){
    printf("Inside exitbcse\n");
    kill(0,SIGTERM);
    exit(0);
    return EXITCODE;
}
```