Making Basic Linux Shell

-Parth Singh

2018356

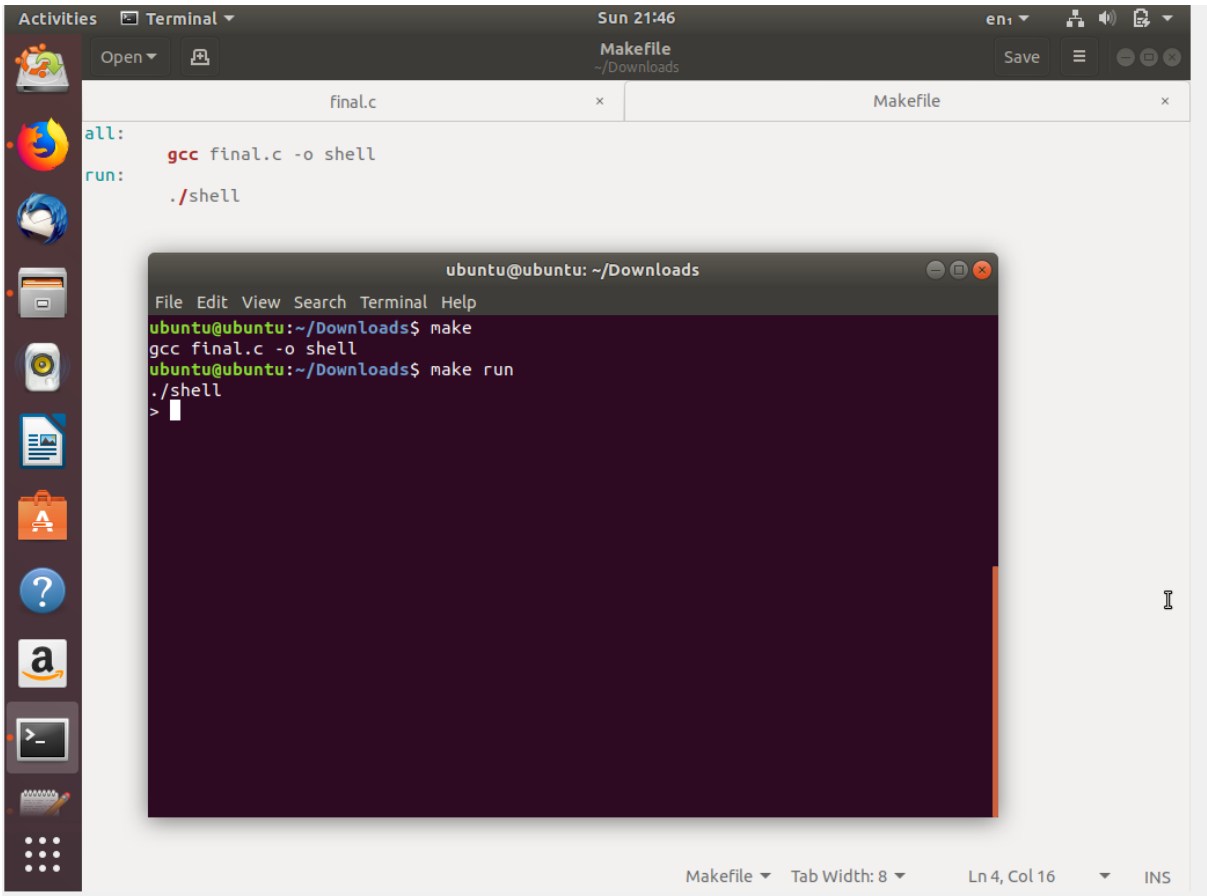
Giving a small summary what we have done in this assignment:

- Google how to make a shell and using the YouTube videos we got to learn how to “c” code internal command shell and external.
- Then we made a “c” code that has to be used in ubuntu terminal.
- Now, before opening the terminal we googled MakeFile, using this we separated different functions in different files and called it in a single file.
- Then you have to handle different errors and after handling it your C code will be done.
- Now just open that code in ubuntu to open the shell.
- Remember first Make a separate file(MAKEFILE) that will open the shell in ubuntu.

Rubrics for the assignment:

1. Successful Compilation using MakeFile:(5marks)

   Here first we made a Makefile that inputs our final.c and runs it.



Thus successful compilation.

2. Use of system calls like – fork(), execl() family of system calls, wait() family of system calls to handle external commands : (10 points)

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

#include<dirent.h>

#include<errno.h>

#include<fcntl.h>

#define GREEN "/x1b[92m"

#define BLUE "/x1b[94m"

#define DEF "/x1b[0m"

#define CYAN "/x1b[96sm"

#define LSH_RL_BUFSIZE 1000

#define LSH_TOK_BUFSIZE 100

#define LSH_TOK_DELIM " \t\r\n\a"

//Function Declarations for builtin shell commands

int lsh_cd(char **args);

int lsh_exit(char **args)

/*

  List of builtin commands, followed by their corresponding functions.

 */

char *builtin_str[] = {

  "cd",

  "exit"

};


int (*builtin_func[]) (char **) = {

  &lsh_cd,

  &lsh_exit

};


int lsh_num_builtins() {
```

```
  return sizeof(builtin_str) / sizeof(char *);
}

/*
  Builtin function implementations.
*/
int lsh_cd(char **args)
{
  if (args[1] == NULL) {
    fprintf(stderr, "lsh: expected argument to \"cd\"\n");
  } else {
    if (chdir(args[1]) != 0) {
      perror("lsh");
    }
  }
  return 1;
}


int lsh_exit(char **args)
{
  return 0;
}
void printDir()
{
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    printf("\nDir: %s", cwd);
}
/* copy one file to another */
void function_cp(char* file1, char* file2)
{
    FILE *f1,*f2;
    struct stat t1,t2;
    f1 = fopen(file1,"r");
    if(f1 == NULL)
    {
        perror("+--- Error in cp file1 ");
```

```
        return;
    }
    f2 = fopen(file2,"r");// if file exists
    f2 = fopen(file2,"ab+"); // create the file if it doesn't exist
    fclose(f2);

    f2 = fopen(file2,"w+");
    if(f2 == NULL)
    {
        perror("Error in cp file2 ");
        fclose(f1);
        return;
    }
    char cp;
    while((cp=getc(f1))!=EOF)
    {
        putc(cp,f2);
    }

    fclose(f1);
    fclose(f2);
}



/* Just a fancy name printing function*/
void nameFile(struct dirent* name,char* followup)
{
    if(name->d_type == DT_REG)         // regular file
    {
        printf("%s%s%s",BLUE, name->d_name, followup);
    }
    else if(name->d_type == DT_DIR)    // a directory
    {
        printf("%s%s/%s",GREEN, name->d_name, followup);
    }
    else                               // unknown file types
    {
        printf("%s%s%s",CYAN, name->d_name, followup);
```

```
    }
}



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)
  {
    // Read a character
    c = getchar();
    if (c == EOF)
     {
      exit(EXIT_SUCCESS);
    } else if (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);
      }
    }
  }
}



char **lsh_split_line(char *line)
{
  int bufsize = LSH_TOK_BUFSIZE, position = 0;
  char **tokens = (char**)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;
}
```

```
int lsh_launch(char **args)
{
  pid_t pid = fork();
  int status;
  if (pid == 0)
  {
    // Child process
    if (execvp(args[0], args) == -1)
    {
      perror("lsh");
    }
    exit(EXIT_FAILURE);
  }
  else if (pid < 0)
  {
    // Error forking
    perror("lsh");
  }
  else
  {
    // Parent process
    do
    {
      waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
  }
  return 1;
}
int lsh_execute(char **args)
{
  int i;
  if (args[0] == NULL)
  {
    // An empty command was entered.
    return 1;
  }
  //----------------------------->>>>>>>>> 5th function takes to lsh_num_builtins which is limit for i variable in for loop
  for (i = 0; i < lsh_num_builtins(); i++)
```

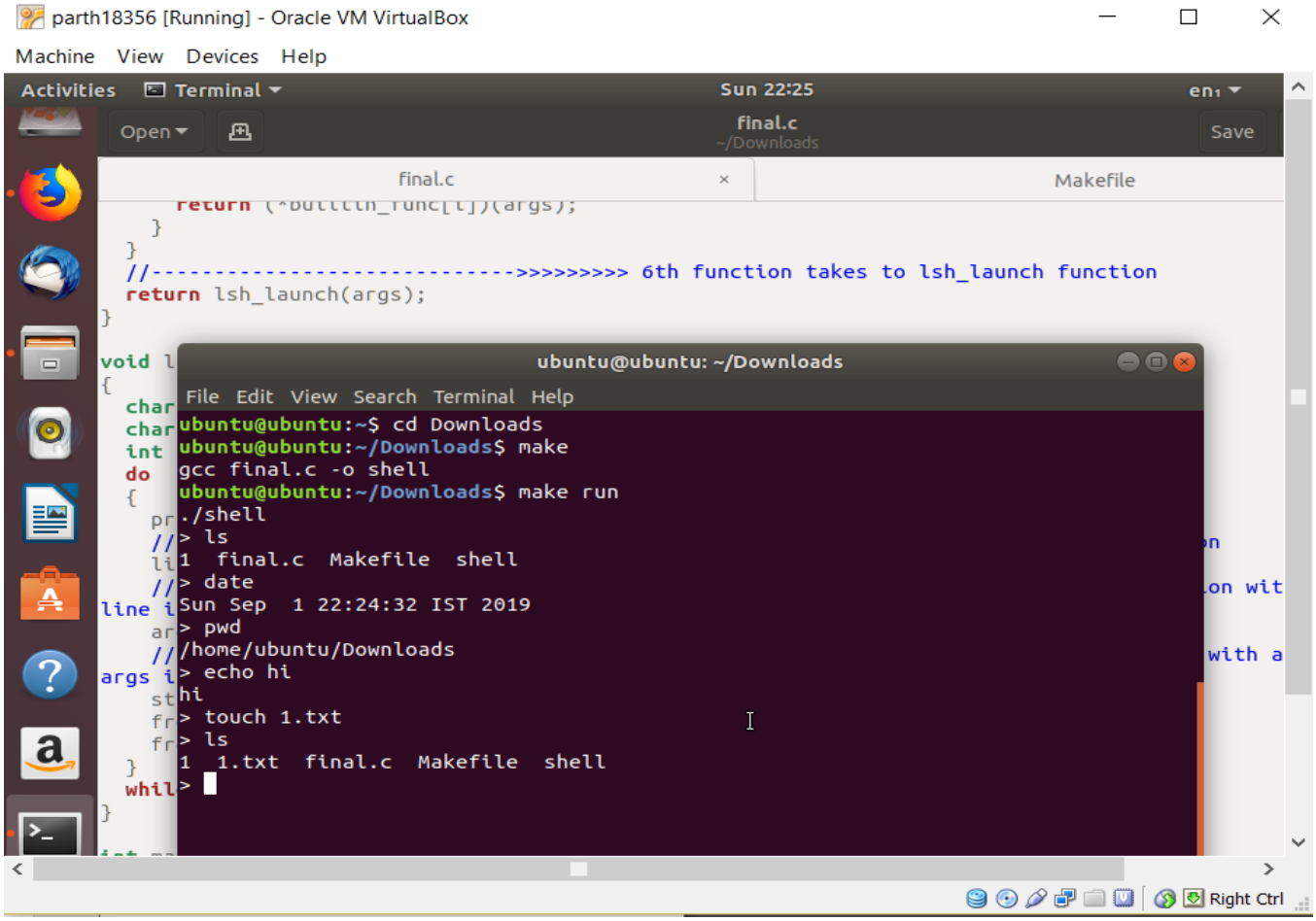
```
  {
                    //----------------------------->>>>>>>>> goes to builtin function i.e builtin_str[]
    if (strcmp(args[0], builtin_str[i]) == 0)
    {
    //----------------------------->>>>>>>>> goes to builtin function i.e builtin_func[]
      return (*builtin_func[i])(args);
    }
  }
  //----------------------------->>>>>>>>> 6th function takes to lsh_launch function
  return lsh_launch(args);
}

void lsh_loop(void)
{
  char *line;
  char **args;
  int status;
  do
  {
    printf("> ");
   /* 2nd function takes to lsh_read_line function*/
    line = lsh_read_line();
    /* 3rd function takes to lsh_split_line function with argument line i.e 2nd function output*/
    args = lsh_split_line(line);
    /* 4th function takes to lsh_execute function with argument args i.e 3rd function output*/
    status = lsh_execute(args);
    free(line);
    free(args);
  }
  while (status);
}

int main(int argc, char **argv)
{
  //------------------------------->>>>>>>>> 1st Function takes to lsh_loop function
  lsh_loop();
  // Perform any shutdown/cleanup.
  return EXIT_SUCCESS;
```

}

Use of all the system calls like 1). fork() , 2).execl() and all other system commands are being used by me in my code.

3. Description of the systems, commands to execute and test the program and the assumptions that you made – 5 points.

Here starts the main thing the description,working,test cases and the errors.



Here we tested:

- ls command that tells what that directory contains.
- date mentions the date.
- pwd tells the current directory in which I am working.
- echo prints the message which I entered along with a gap with echo.
- Touch made a new empty text file.

Functioning of my shell:

- My shell takes the input from the user through my lsh_read_line() function.
- Then it checks the inputs length
- If length is 0 then it will not do anything.
- Else it parse the line by separator ‘ ‘ (space).
- After that it checks the command and then it goes to the next function i.e. lsh_execute.
- Then the command output is displayed on the screen.

Assumptions:

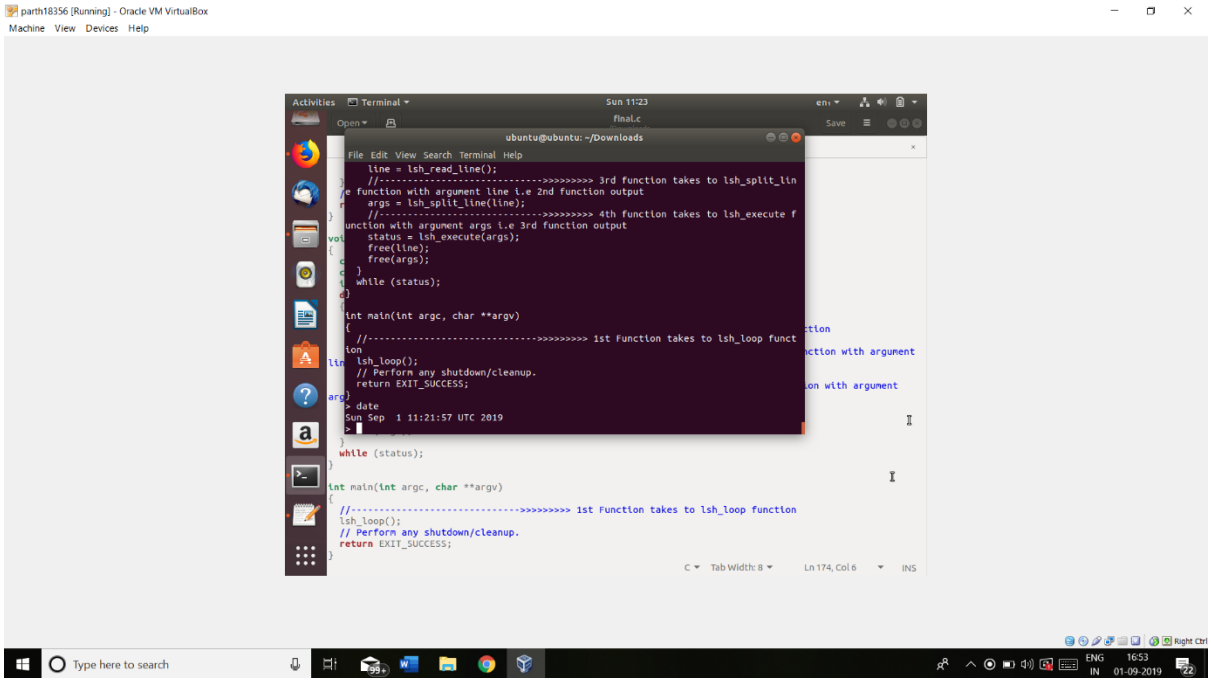
- The user does not use the concept of pipeline.
- The user does not want to import any module.
- The user does not want the shell to be advance shell where every module is preinstalled in the memory.

There were many errors that came while I was coding and testing it in ubuntu .

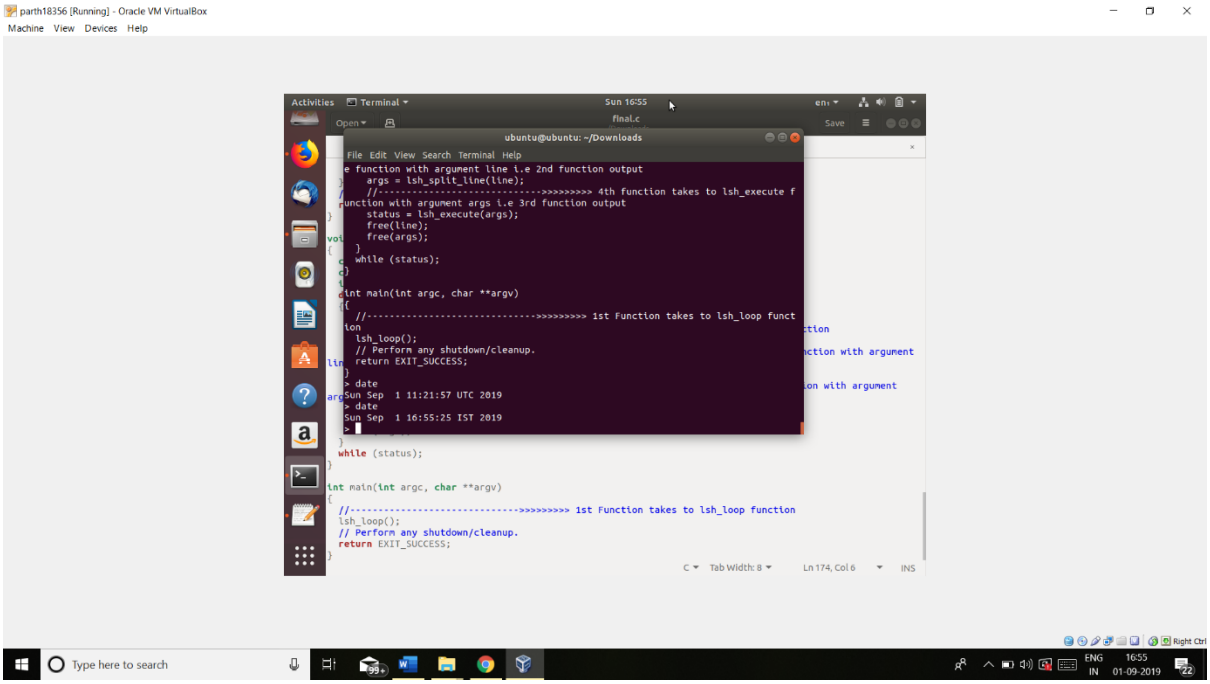
The code errors were sorted but some shell errors were difficult to sort .

Some shell errors are:

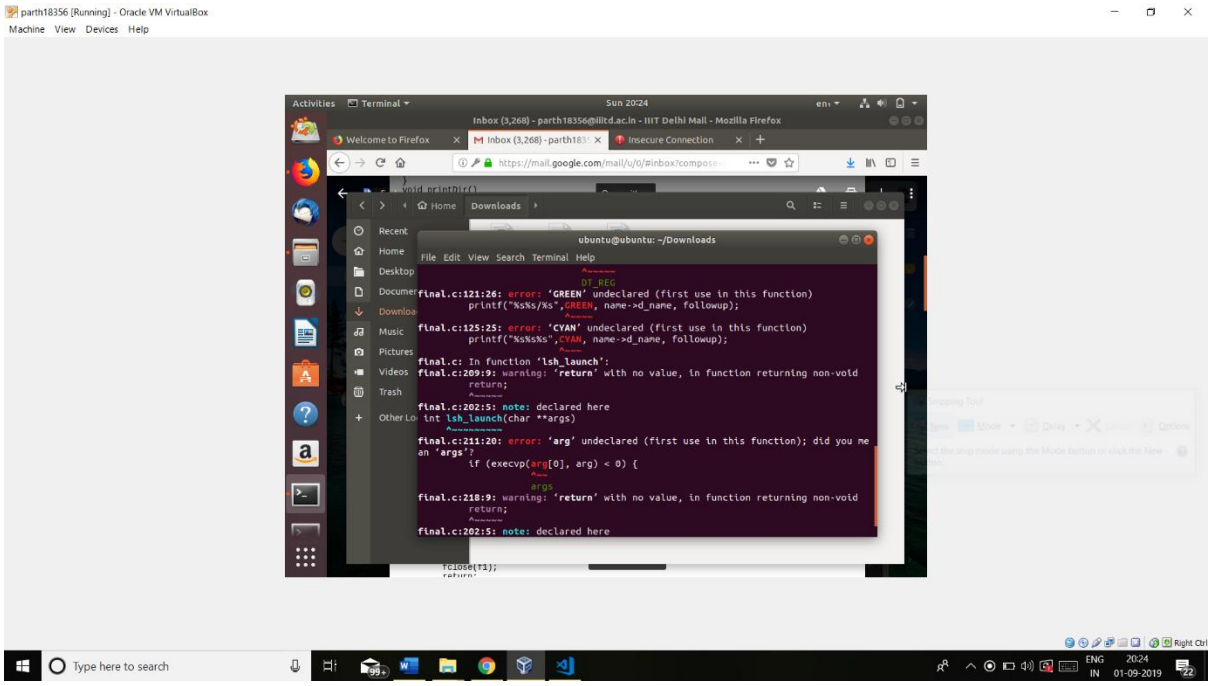
- The date was not correct in my shell.



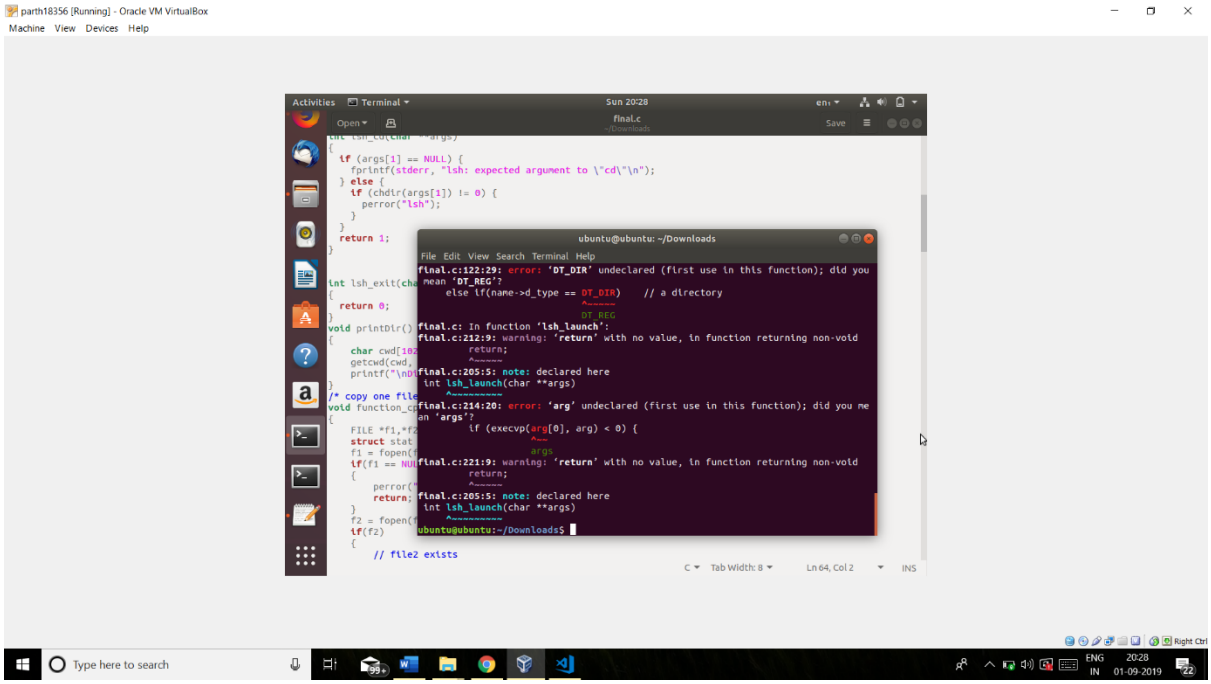
- Solved the date issue



- The colors were not defined in my code.



- The parameters were not being extracted in most of the places.



- Cat command was not running . Though coded for it.