

Project Report

GROUP : 16

SLOT : 11

STAGE 2 SUBMISSION

Project – Unix Shell in C++ with some mathematical functions

Project Name – MaSH

How did **MaSH** came into being :

We first had the idea to do some kernel level development / OS development. Such a topic was chosen because it would be a good place to start applying what we are learning in class. But our progress was hindered by "Assembly Code". As we are novice programmers so we thought, "What would be the next best thing to kernel ? ". Thus Shell development came into play, and being students of mathematics, we couldn't resist ourselves from mixing and interplaying shell and mathematics. Though it doesn't have any direct connection to mathematics, we tried our best to intermix these two things and hope that from this endeavour we will come-through as better coder.

How does **MaSH** works :

Shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands. Users typically interact with a Unix shell using a terminal emulator, however, direct operation via serial hardware connections, or networking session, are common for server systems.

In our Shell, when a user enters command then the shell searches for that command in the `/bin/` folder that is present on the root directory. If command is not found it then searches using path defined in the environment variable `PATH`. If it still doesn't find command then it prints that such command doesn't exist.

Here is our main function

```
int main (int argc, char **argv)
{
    int childPid, sig = 0 ;

    char * cmd_line, *argv[MAX], *cmd1[MAX], *cmd2[MAX] ;

    char *user_name, host_name[MAX], *const_msg, cwd[MAX] ;

    init_shell () ; // Initializing shell

    const_msg = NULL ;

    user_name = getenv ("USER") ;

    const_msg = user_name ;

    gethostname (host_name, sizeof(host_name)) ;
```

```

strcat (const_msg, "@") ;

strcat (const_msg, host_name) ;


system ("clear") ;


while (true)
{
    if (getcwd (cwd, sizeof(cwd)) == NULL)
    {
        perror ("\033[1;31mError: \033[0m") ;
        break ;
    }

    fprintf (stdout, "\033[1;32m%s\033[0m:~\033[36m%s\033[0m",
const_msg, cwd) ;

    cmd_line = readline (" # ") ;

    if (strlen (cmd_line) != 0)
    {
        add_history (cmd_line) ;

        sig = parse_cmd (cmd_line, argv, cmd1, cmd2) ;

        if (sig == PIPE_FOUND)
            pipe_command (cmd1, cmd2) ;
        else if (sig == REDIRECT_FOUND)
            redirect_command (cmd1, cmd2) ;
        else if (sig == -1)
            cerr << "Both piping and redirection simultaneously not
implemented" ;

```

```

        else if (strcmp (cmd_line, "exit") == 0)

            break ;

        else if (strcmp (argv[0], "cd") == 0)

            change_directory (argv) ;

        else

            run_cmd (argv) ;

    }

}

return 0 ;

}

```

In our shell we have used some system specific builtin functions like `fork()`, `execvp()`, `dup2()`, `pipe()`, `waitpid()`

Explanation for some important functions used in Shell :

- `dup2()` is a system call similar to `dup` in that it duplicates one file descriptor, making them aliases, and then deleting the old file descriptor. This becomes very useful when attempting to redirect output, as it automatically takes care of closing the old file descriptor, performing the redirection in one elegant command. For example, if you wanted to redirect standard output to a file, then you would simply call `dup2`, providing the open file descriptor for the file as the first command and 1 (standard output) as the second command.
- `run_cmd()` is our core function. It helps running basic *nix function. What it does is explained below :-
 - It gets the parsed commands, it helps with `execvp()` function call.
 - By calling `fork()` it creates a new process.
 - Now it calls `execvp` and runs the required command using the process id returned by `fork()`
- `change_directory()` helps in changing directory. It uses `chdir()` function for this purpose.
- `pipe_command()` is not built completely yet. It uses `pipe()` function and `dup2()` for piping, using a file descriptor.

- `pipe()` creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by `filedes`. `filedes[0]` is for reading, `filedes[1]` is for writing.
- `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.
- `execvp()` Each of the functions in the `exec` family replaces the current process image with a new process image. The new image is constructed from a regular, executable file called the new process image file. This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.
- `parse_cmd()` is another core function. It helps checking whether the input has I/O redirection or pipe. If it has, then it lets the main function know by sending appropriate signal. It also tokenize the input by splitting the command at spaces with the help of `strtok()` function present in `<cstring>` library.
- `run_math_cmd()` is a core function that helps us to run different mathematical commands based on the user input. It works in the same fashion as in `run_cmd()`. First it forks a child process, and then replaces that process with our math command process using the `execv()` function. This function requires us to pass `PATH` of the program.
- `password()` is a function that helps us secure our shell, though it may give to reverse engineering. We have used the `getpass()` function which is readily available in `<unistd.h>` header file, which is `Unix` specific.
- `init_shell()` When a shell program that normally performs job control is started, it has to be careful in case it has been invoked from another shell that is already doing its own job control. For implementing and handling `SIGNAL` control, this function is required.

Mathematical function :

Integration : For integration, we are using the trapezoidal rule. The rule is the following
The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that.

$$\int_a^b f(x) dx \approx (b - a) \left[\frac{f(a) + f(b)}{2} \right].$$

The trapezoidal rule is one of a family of formulas for numerical integration called Newton–

Cotes formulas, of which the midpoint rule is similar to the trapezoid rule. Simpson's rule is another member of the same family, and in general has faster convergence than the trapezoidal rule for functions which are twice continuously differentiable, though not in all specific cases. However for various classes of rougher functions (ones with weaker smoothness conditions), the trapezoidal rule has faster convergence in general than Simpson's rule.

Thus because of faster convergence we chose trapezoidal rule over the simpsons rule.

We approximate the integral by increasing the no.of partition.

Differentiation : Differentiation has been implemented using the first principal of derivative. First principal can be derived from the definition of derivatives or it a general consequence of Taylor's expansion of an analytic function.

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}.$$

We again approximate the differentiation by decreasing the value of h and making it a const variable in our program.

Finding Roots : For Finding roots of a given function we have used the Newton-Raphson method. In numerical analysis, Newton's method (also known as the Newton-Raphson method), named after Isaac Newton and Joseph Raphson, is a method for finding successively better approximations to the roots (or zeroes) of a real-valued function. The iterative formula is given as :-

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We have used our derivative program to the fullest in this program.

GCD and LCM : The GCD and LCM programs have been done for given any n number of integers. This was done by using the standard method of finding GCD and LCM of two integers and then recursively applying them.

Matrix : Simple Matrix manipulation has been done using class, constructor and operator overloading. Classes and operator overloading eased our task of manipulating matrices. Our base class in Matrix, which has a default constructor, parameterized constructor and a copy constructor. We have overloaded several operators like +, -, *, () etc. Thanks to our last few lectures on classes, our job was not that difficult to accomplish. Using these overloaded operators we are inputting and outputting matrices by `cin >> A ;` and `cout << A ;` making our life easy. Our class also has a user defined destructor, which helps us clean up the objects created.

Future plan for improvement :

- Implementing scripting in our shell
- Implementing concatenated piping and I/O redirecting
- Job control implementation
- Implementing piping and I/O redirecting together
- Providing online user technical support
- To be able to edit environmental variable from Shell
- Implement Recursive descent parser for implementing command line string evaluation.

Conclusion :

As expected, we have completed our project and it works perfectly with the mathematical functions that we have implemented. Nevertheless we have scope for improvement and optimizing our code and its working. We have created a shell which will work in most of the Unix based environment but it would be another level of challenge to make it have cross platform support. In future, we will try to improve this project or use the experience gained in this project in our future endeavour.

Acknowledgement :

This CS101 project has been an exhilarating journey and at the end of the day has made each one of us a better coder, thinker and problem solver. The lab sessions with our TA helped us in a great way. We are lucky to have such dedicated instructor and TA. We thank our Instructors and our TA for their continuous support, help and acknowledgement.

Team members :

Niran Meher - 145090029
Animesh Patel - 145090004
Abhishek Guha – 145090024

All are from Dept of Mathematics IIT Bombay M.Sc 1st year