VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Advanced Sciences

Programme : B.Sc Mathematics and Computing

Course Title : OPERATING SYSTEMS

Course Code : UCSE252P

Slot : M9+R9+U8

Title: Shell Command Line interface - Simulate an OS terminal

TEAM MEMBERS:

- **S.SANTHOSH 22BMC1009**
- **AKSHAYA J 22BMC1010**

Faculty: Mrs. SUVIDHA RUPESH KUMAR

Sign:

Date: 28.09.2023

ABSTRACT:

The main objective of this report is to define the code and the UI components that goes into building the Shell command line interface using Python.

INTRODUCTION:

The outermost layer of an operating system, a shell is a computer application software that offers the services of an operating system to an external user or another program. Considering the operation role of the computer, its shell may have either a CLI (Command Line Interface) or a GUI (Graphical User Interface).

Shells in most operating systems are note direct interfaces to the kernel that lies beneath, even if the shell communicates with the program or user through peripheral devices directly attached to the computer system. It manages basic user-system interactions by taking user input, evaluating it, and eventually dealing with the output.

A shell reads standard input from as well as passes the standard output or the standard error to the terminal. Terminal is an application program that enables a user to access and use a shell in the operating system. The shell is line oriented, which means that the commands are not executed until the enter key is pressed.

The shell partitions the line into what is referred to as tokens. A token is a variable, command, or some other symbol that the shell recognizes. A sequence of tokens is constructed repeatedly until a reserved function name, a keyword (a shell internal command that determines the control flow of a shell script), or an operator (any symbol that denotes a command separator, pipe, any logical condition, or any other operation the execution

of which cannot be started until the preceding command is interpreted) is formed.

The shell organizes tokens into 3 categories:

- I/O Redirection These are commands that determine where the input of a program is to be derived from or where the output of command execution is to be directed.

- Variable Assignment These are commands that assign some value to any variable.

- Miscellaneous Commands This category includes all the other commands. Other tokens are checked to determine whether they are aliases. This process is started with the first command. If it is found to be an alias, it is then replaced with the value (meaning) corresponding to the alias. If the command is not an alias or if it is followed by a whitespace character before the next command, the steps of alias checking are repeated so long as either there remain any commands, or until an alias has been recognized that is not followed by a space.

METHODOLOGY:

Simulation of an OS terminal:Breaking down the python code:

Step 1: Import the Python libraries.

```
import tkinter as tk
import subprocess
import os
```

To create the simulation, the following libraries are used:

- Tkinter = Used for developing the GUI of the terminal

- Subprocess = Allows us to run programs and other commands from the Python code.
- OS = Provides functions for interacting with the operating system.

Step 2: Create the frame of the terminal.

```
class Terminal(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.master = master

        self.text = tk.Text(self, height=20, width=80, bg="black", fg="white")
        self.text.pack(fill=tk.BOTH, expand=True)

        self.entry = tk.Entry(self, bg="black", fg="white", insertbackground="white")
        self.entry.pack(fill=tk.X)

        self.entry.bind("<Return>", self.run_command)

        self.history = []
        self.history_index = 0

        self.prompt = '>> '
        self.text.insert(tk.END, self.prompt)
```

- The line defines a class named ‘Terminal’, which inherits from the ‘tk.Frame’ class. This intends to create a GUI based terminal using Tkinter.
- ‘def__init__(self,master):’: This is the constructor method for the Terminal class. It takes a single argument, ‘master’, which represents the parent widget or window where this terminal will be placed.
- ‘super().__init__(master)’: This line calls the constructor of the parent class (‘tk.Frame’) to initialize the terminal as a frame within the ‘master’ widget or window.
- ‘self.text = tk.Text(self, height=20, width=80, bg="black", fg="white")’: Here, you create a Text widget (a multi-line text box) to display the terminal output. It has a black background and white text colour and is initially set to a size of 20 rows and 80 columns. This widget is a child of the ‘Terminal’ frame.
- ‘self.text.pack(fill=tk.BOTH, expand=True)’: This line uses the ‘pack’ geometry manager to make the text widget fill the available space within the

‘Terminal’ frame. It expands both horizontally and vertically to fill any available space.

- ‘self.entry = tk.Entry(self, bg="black", fg="white", insertbackground="white")’: This line creates an Entry widget, which is a single-line text input field. It has a black background, white text color, and a white cursor. This widget is also a child of the ‘Terminal’ frame.
- ‘self.entry.pack(fill=tk.X)’: This line uses the ‘pack’ geometry manager to place the entry widget at the top of the frame, allowing it to expand only horizontally (filling the available width).
- ‘self.entry.bind("<Return>", self.run_command)’: Here, you bind the ‘<Return>’ (Enter) key event to the ‘self.run_command’ method. This means that when the user presses Enter in the entry widget, the ‘run_command’ method will be called.
- ‘self.history = []’: This initializes an empty list called ‘history’, which is intended to store the command history.
- ‘self.history_index = 0’: This initializes a variable called ‘history_index’ to 0, which will be used to navigate through the command history.
- ‘self.prompt = '>> ''’: This sets the initial value of the ‘prompt’ attribute to '>> ', which is a common symbol used to indicate a command prompt.
- ‘self.text.insert(tk.END, self.prompt)’: This inserts the initial prompt '>>' into the text widget. It places the prompt at the end of the text widget, so the user can start typing their commands after it.

Step 3: Handling the execution of commands entered by the user in the terminal.

```
def run_command(self, event):
    command = self.entry.get()
    self.history.append(command)
    self.history_index = len(self.history)

    self.text.insert(tk.END, "\n" + self.prompt + command + "\n")

    try:
        if command.startswith("cd "):
            path = command[3:]
            os.chdir(path)
        elif command.endswith(".sh"):
            subprocess.run(command, shell=True)
        elif command.startswith("nano "):
            file_path = command[8:]
            subprocess.Popen(["notepad.exe", file_path])
        elif command.startswith("gcc "):
            file_path = command[4:]
            exe_path = os.path.splitext(file_path)[0] + ".exe"
            compile_command = f"gcc {file_path} -o {exe_path}"
            result = subprocess.run(compile_command, shell=True, stderr=subprocess.PIPE, text=True)
            if result.returncode == 0:
                run_command = f"{exe_path}"
                output = subprocess.check_output(run_command, shell=True, stderr=subprocess.STDOUT, text=True)
                self.text.insert(tk.END, output + "\n")
            else:
                self.text.insert(tk.END, "Compilation Error: " + result.stderr + "\n")
        else:
            output = subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT, text=True)
            self.text.insert(tk.END, output + "\n")
```

```
    except subprocess.CalledProcessError as e:
        self.text.insert(tk.END, "Error: " + str(e) + "\n")

    self.text.insert(tk.END, self.prompt)
    self.entry.delete( first: 0, tk.END)
```

The ‘run_command’ method is responsible for processing and executing user-entered commands in the Tkinter terminal.

- ‘command = self.entry.get()’: This line retrieves the text entered by the user in the Entry widget and stores it in the ‘command’ variable.
- ‘self.history.append(command)’: The entered command is added to the command history list.
- ‘self.history_index = len(self.history)’: The ‘history_index’ is updated to the current length of the history list, which helps in navigating through the command history.
- ‘self.text.insert(tk.END, "\n" + self.prompt + command + "\n")’: This line inserts the entered

command, along with the prompt and newline characters, into the Text widget. This displays the entered command in the terminal output.

- The code then checks the content of the command to determine how to handle it. Here are the different cases:

  ‘if command.startswith("cd ")’: If the command starts with "cd", it's treated as a change directory command (‘cd’). The code changes the current working directory accordingly using ‘os.chdir(path)’.

  ‘elif command.endswith(".sh")’: If the command ends with ".sh", it's treated as a shell script execution command. The code runs the command using ‘subprocess.run(command, shell=True)’.

  ‘elif command.startswith("nano ")’: If the command starts with "nano ", it's treated as a request to open a file in Notepad (a simple text editor). The code extracts the file path and uses ‘subprocess.Popen(["notepad.exe", file_path])’ to open the file in Notepad.

  ‘elif command.startswith("gcc ")’: If the command starts with "gcc ", it's treated as a C source code compilation command. The code extracts the file path, compiles it using GCC, and runs the resulting executable. If compilation is successful, the output is displayed.

  For any other command, it's executed using ‘subprocess.check_output(command, shell=True)’, and the output is displayed.

- If an exception of type ‘subprocess.CalledProcessError’ occurs, it's caught in the ‘except’ block, and an error message is displayed in the terminal.
- Finally, the code inserts the prompt back at the end of the Text widget, clears the input Entry widget by

deleting its content, and prepares the terminal for the next command.

Step 4: Adding methods to navigate ‘up’ the command line.

```
def handle_up_key(self, event):
    if self.history_index > 0:
        self.history_index -= 1
        self.entry.delete( first: 0, tk.END)
        self.entry.insert( index: 0, self.history[self.history_index])
```

The ‘handle_up_key’ method is meant to handle the “up” arrow key press event, allowing the user to navigate through their command history.

- ‘if self.history_index > 0’: This condition checks if there are previous commands in the command history (the index is greater than 0), it carries out the following commands:

  ‘self.history_index -= 1’: If there are previous commands, it decrements the ‘history_index’ by 1 to move to the previous command in the history.

  ‘self.entry.delete(0, tk.END)’: This line clears the content of the Entry widget to prepare it for the new command.

  ‘self.entry.insert(0, self.history[self.history_index])’: Here, the method inserts the command from the history (indexed by ‘self.history_index’) into the Entry widget. This displays the previous command in the Entry widget for editing or resubmission.

Step 5: Adding methods to navigate 'down' the command line.

```
def handle_down_key(self, event):
    if self.history_index < len(self.history) - 1:
        self.history_index += 1
        self.entry.delete( first: 0, tk.END)
        self.entry.insert( index: 0, self.history[self.history_index])
    elif self.history_index == len(self.history) - 1:
        self.history_index += 1
        self.entry.delete( first: 0, tk.END)
```

The 'handle_down_key' method is meant to handle the "down" arrow key press event, allowing the user to navigate through their command history in reverse order.

- 'if self.history_index < len(self.history) – 1': This condition checks if there are more recent (forward) commands in the command history. If 'history_index' is less than 'len(self.history) – 1', it means there are commands ahead in the history.
- If there are more recent commands, the function does the following:
  'self.history_index += 1': It increments the history_index by 1 to move forward in the history.

  'self.entry.delete(0, tk.END)': This line clears the content of the Entry widget to prepare it for the new command.

  'self.entry.insert(0, self.history[self.history_index])': Here, the method inserts the next command from the history (indexed by 'self.history_index') into the Entry widget. This displays the next command in the Entry widget for editing or resubmission.

- 'elif self.history_index == len(self.history) – 1': If the 'history_index' is already at the most recent command in the history, this condition will be true. In this case, you increment 'history_index' by 1 (to move it out of the bounds of the history) and clear

the Entry widget. This effectively clears the input field, allowing the user to start a new command.

Step 6: Running the Tkinter terminal application.

```
if __name__ == "__main__":
    root = tk.Tk()
    root.title("COMMAND LINE OF SMDS")
    root.geometry("1920x1080")
    root.configure(bg="black")
    terminal = Terminal(root)
    terminal.pack(fill=tk.BOTH, expand=True)

    root.bind("<Up>", terminal.handle_up_key)
    root.bind("<Down>", terminal.handle_down_key)

    root.mainloop()
```

The 'if __name__ == "__main__"': block is where the Tkinter terminal application is initialized and run.

- 'root = tk.Tk()': Creates the main application window using 'tk.Tk()'.This window serves as the container for your terminal interface.
- 'root.title("COMMAND LINE OF SMDS")': Setting the title of the application window to "COMMAND LINE OF SMDS".
- 'root.geometry("1920x1080")': This line sets the initial size of the application window to 1920 pixels in width and 1080 pixels in height.
- 'root.configure(bg="black")':Setting the background colour of the application window to black. This sets the overall background colour of the window.
- 'terminal = Terminal(root)': You create an instance of the 'Terminal' class, passing the 'root' window as the 'master'. This creates your terminal interface within the application window.
- 'terminal.pack(fill=tk.BOTH, expand=True)': You use the 'pack' geometry manager to make the 'Terminal' widget fill the available space within the 'root' window, expanding both horizontally and vertically.
- 'root.bind("<Up>", terminal.handle_up_key)': Binds the "Up" arrow key press event to the 'handle_up_key' method of Terminal widget. This

allows the user to navigate through the command history by pressing the "Up" arrow key.

- ‘root.bind("<Down>", terminal.handle_down_key)’: Similarly, we bind the "Down" arrow key press event to the ‘handle_down_key’ method of our Terminal widget. This allows the user to navigate through the command history in reverse order by pressing the "Down" arrow key.
- ‘root.mainloop()’: Finally, we start the Tkinter main event loop by calling mainloop(). This loop keeps your application running, waiting for user interactions and events.

KEY FEATURES:

Text-Based Interface: This CLI provides a text-based interface that allows users to interact with the program by typing commands and viewing textual output.

Text Widget: The use of Tkinter's Text widget provides a multi-line text area where the CLI displays output and user input. It's an essential part of the user interface.

Entry Widget: The Entry widget allows users to input commands easily. Users can type their commands in this widget.

Command Execution: This CLI is capable of executing various commands, including system commands, file editing, and code compilation.

Command History: It keeps a history of previously entered commands, allowing users to navigate through their command history using the Up and Down arrow keys.

Prompt: This CLI displays a prompt (">>") to indicate where users can input their commands, providing a familiar command-line experience.

Dynamic Output: The CLI displays the results of executed commands in real-time, enhancing user feedback and interaction.

File Editing: This CLI supports opening files using the "nano" command, which opens files in the default text editor (Notepad).

Code Compilation: It can compile C code using the "gcc" command and run the resulting executable, displaying compilation errors if they occur.

User-Friendly: The CLI's user interface is designed with a dark theme, making it visually appealing and easy on the eyes.

Keyboard Shortcuts: It includes keyboard shortcuts for navigating command history, making it more convenient for users.

Event Handling: Your CLI effectively handles events like pressing the Enter key, Up arrow, and Down arrow, improving the overall user experience.

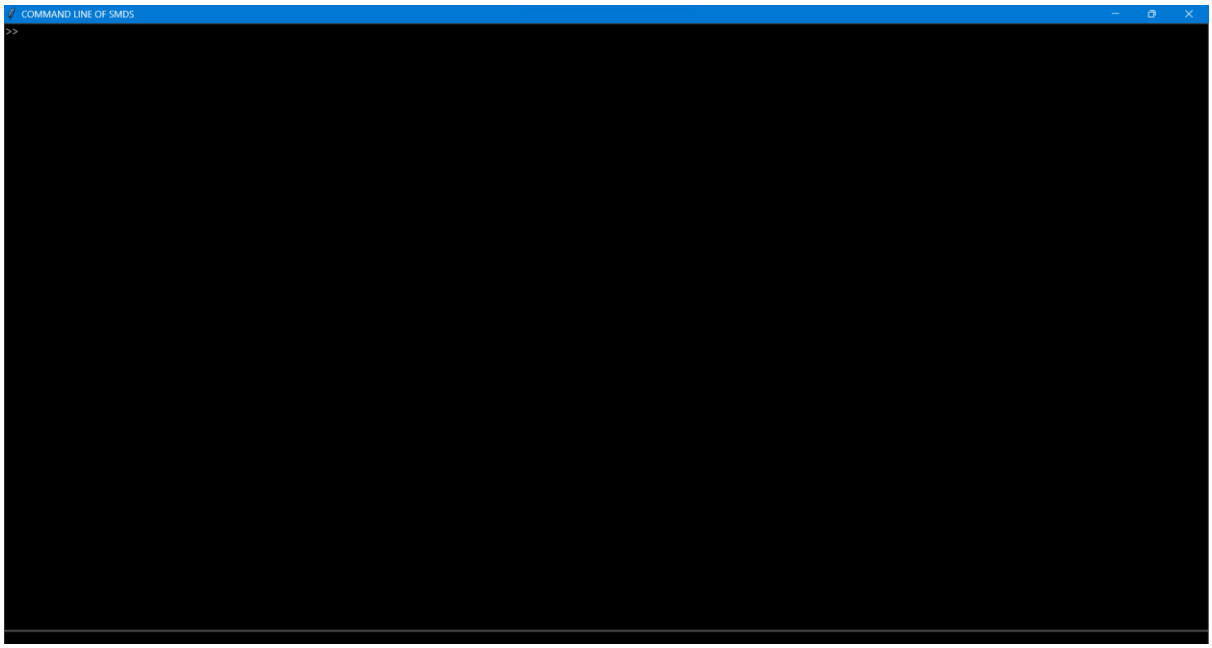
CODE:

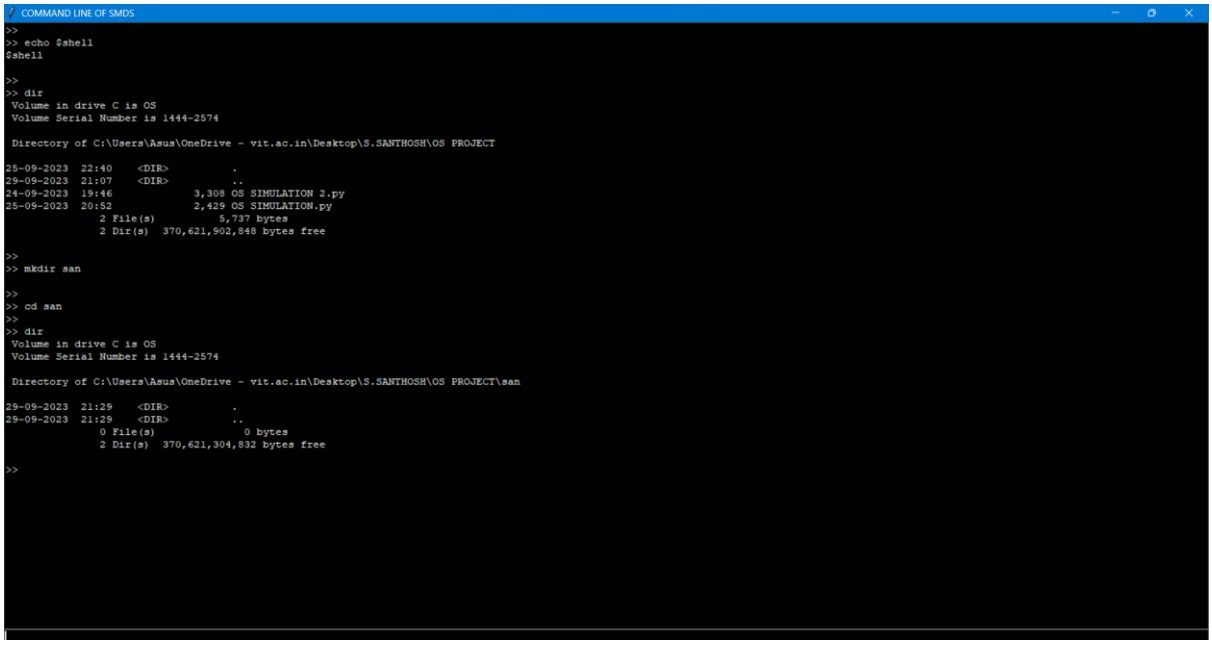
```
import tkinter as tk
import subprocess
import os

class Terminal(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.master = master

        self.text = tk.Text(self, height=20, width=80,
bg="black", fg="white")
        self.text.pack(fill=tk.BOTH, expand=True)

        self.entry = tk.Entry(self, bg="black",
fg="white", insertbackground="white")
        self.entry.pack(fill=tk.X)

        self.entry.bind("<Return>", self.run_command)

        self.history = []
        self.history_index = 0

        self.prompt = '>> '
        self.text.insert(tk.END, self.prompt)

    def run_command(self, event):
        command = self.entry.get()
        self.history.append(command)
        self.history_index = len(self.history)

        self.text.insert(tk.END, "\n" + self.prompt +
command + "\n")

        try:
            if command.startswith("cd "):
                path = command[3:]
                os.chdir(path)
```

```
            elif command.endswith(".sh"):
                subprocess.run(command, shell=True)
            elif command.startswith("nano "):
                file_path = command[8:]
                subprocess.Popen(["notepad.exe",
file_path])
            elif command.startswith("gcc "):
                file_path = command[4:]
                exe_path =
os.path.splitext(file_path)[0] + ".exe"
                compile_command = f"gcc {file_path} -o
{exe_path}"
                result =
subprocess.run(compile_command, shell=True,
stderr=subprocess.PIPE, text=True)
                if result.returncode == 0:
                    run_command = f"{exe_path}"
                    output =
subprocess.check_output(run_command, shell=True,
stderr=subprocess.STDOUT, text=True)
                    self.text.insert(tk.END, output +
"\n")
                else:
                    self.text.insert(tk.END,
"Compilation Error: " + result.stderr + "\n")
            else:
                output =
subprocess.check_output(command, shell=True,
stderr=subprocess.STDOUT, text=True)
                self.text.insert(tk.END, output + "\n")
        except subprocess.CalledProcessError as e:
            self.text.insert(tk.END, "Error: " + str(e)
+ "\n")

        self.text.insert(tk.END, self.prompt)
        self.entry.delete(0, tk.END)

    def handle_up_key(self, event):
        if self.history_index > 0:
            self.history_index -= 1
```

```
            self.entry.delete(0, tk.END)
            self.entry.insert(0,
self.history[self.history_index])

    def handle_down_key(self, event):
        if self.history_index < len(self.history) - 1:
            self.history_index += 1
            self.entry.delete(0, tk.END)
            self.entry.insert(0,
self.history[self.history_index])
        elif self.history_index == len(self.history) -
1:
            self.history_index += 1
            self.entry.delete(0, tk.END)

if __name__ == "__main__":
    root = tk.Tk()
    root.title("COMMAND LINE OF SMDS")
    root.geometry("1920x1080")
    root.configure(bg="black")
    terminal = Terminal(root)
    terminal.pack(fill=tk.BOTH, expand=True)

    root.bind("<Up>", terminal.handle_up_key)
    root.bind("<Down>", terminal.handle_down_key)

    root.mainloop()
```

OUTPUT:


COMMAND LINE OF SMDS
>>


COMMAND LINE OF SMDS
>>
>> echo $shell
$shell
>>
>> dir
Volume in drive C is OS
Volume Serial Number is 1444-2574
Directory of C:\Users\Asus\OneDrive - vit.ac.in\Desktop\S.SANTHOSH\OS PROJECT
25-09-2023 22:40 <DIR> .
29-09-2023 21:07 <DIR> ..
24-09-2023 19:46 3,308 OS SIMULATION 2.py
25-09-2023 20:52 2,429 OS SIMULATION.py
2 File(s) 5,737 bytes
2 Dir(s) 370,621,902,848 bytes free
>>
>> mkdir san
>>
>> cd san
>>
>> dir
Volume in drive C is OS
Volume Serial Number is 1444-2574
Directory of C:\Users\Asus\OneDrive - vit.ac.in\Desktop\S.SANTHOSH\OS PROJECT\san
29-09-2023 21:29 <DIR> .
29-09-2023 21:29 <DIR> ..
0 File(s) 0 bytes
2 Dir(s) 370,621,304,832 bytes free
>>

CONCLUSION:

The OS command line interface simulation project has successfully demonstrated the feasibility of using Python to create a realistic and interactive simulation of a command line interface. The Tkinter GUI provides a user-friendly interface for interacting with the simulation and the simulation itself can accurately replicate the behavior of a real command line interface.
Throughout this project, we have covered various aspects of (CLI) Command Line Interface development including user input, executing system commands, handling errors and providing a user-friendly interface.
This report serves as a comprehensive guide to creating a Python-based Shell command line interface.