Advanced Shell Simulation with Integrated OS Concepts

https://github.com/baralsamrat/MSCS630_Project

Deliverable 1

Basic Shell Implementation and Process Management

Samrat Baral

January 19, 2025

University of the Cumberlands

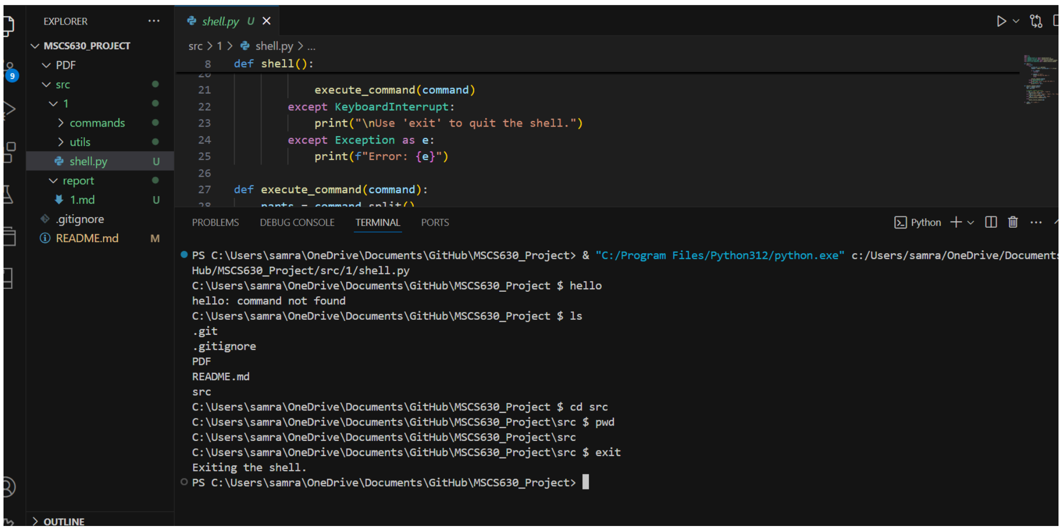
Advanced Operating Systems (MSCS-630-A01)

Primus Vekuh

Code Submission

https://github.com/baralsamrat/MSCS630_Project

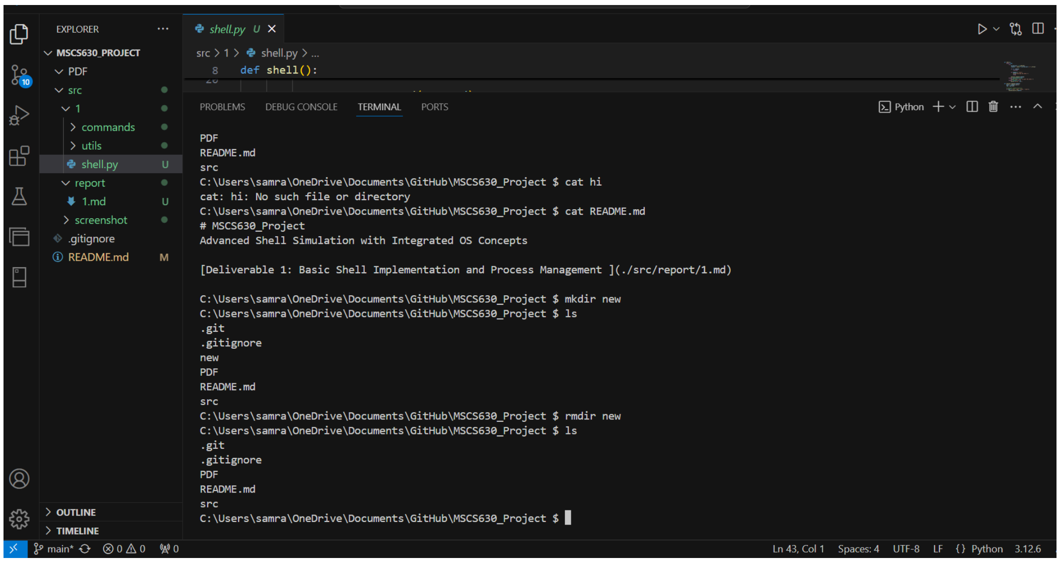
Screenshots:



```python
def shell():
                execute_command(command)
            except KeyboardInterrupt:
                print("\nUse 'exit' to quit the shell.")
            except Exception as e:
                print(f"Error: {e}")

def execute_command(command):
```

```
PS C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project> & "C:/Program Files/Python312/python.exe" c:/Users/samra/OneDrive/Document
Hub/MSCS630_Project/src/1/shell.py
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ hello
hello: command not found
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ cd src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project\src $ pwd
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project\src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project\src $ exit
Exiting the shell.
PS C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project>
```

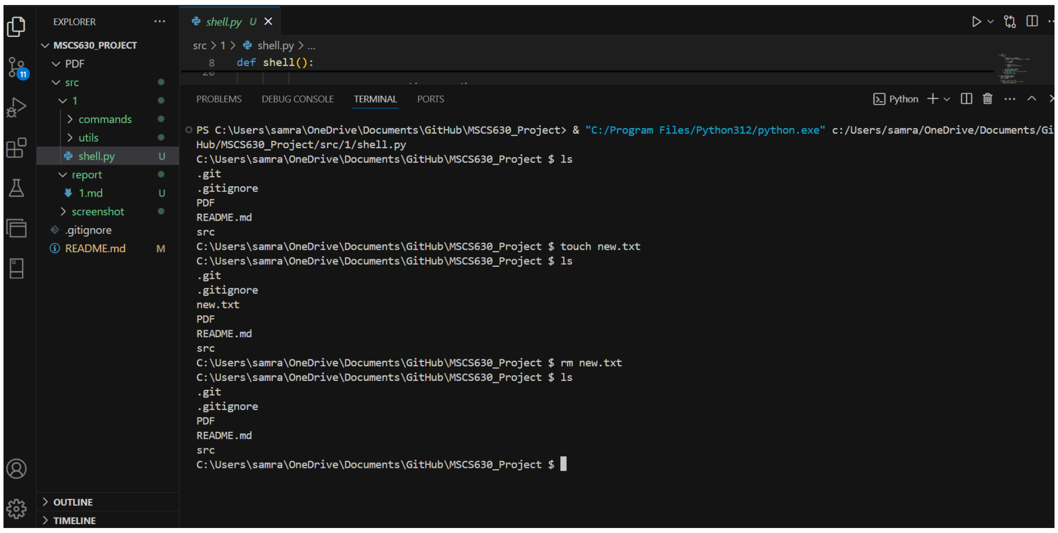


```python
def shell():
```

```
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ cat hi
cat: hi: No such file or directory
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ cat README.md
# MSCS630_Project
Advanced Shell Simulation with Integrated OS Concepts

[Deliverable 1: Basic Shell Implementation and Process Management ](./src/report/1.md)

C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ mkdir new
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
new
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ rmdir new
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $
```



```python
def shell():
```

```
PS C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project> & "C:/Program Files/Python312/python.exe" c:/Users/samra/OneDrive/Documents/Gi
Hub/MSCS630_Project/src/1/shell.py
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ touch new.txt
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
new.txt
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ rm new.txt
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $ ls
.git
.gitignore
PDF
README.md
src
C:\Users\samra\OneDrive\Documents\GitHub\MSCS630_Project $
```

Process Management:

In our custom shell, process management was implemented using system calls provided by Python's subprocess module. This allows the shell to execute user commands as separate processes, ensuring isolation and efficient management of resources.

Foreground Processes:

- *Creation*: Foreground processes are created using subprocess.run(). This method blocks the shell until the process completes, ensuring the user sees the output or errors before proceeding to the next command.
- *Execution:* When the user enters a command without the & symbol, it runs as a foreground process. The shell waits for the process to complete using a subprocess.run(), displaying output or errors in real-time.

Background Processes:

- *Creation*: Background processes are created using subprocess.Popen(). This allows the process to run asynchronously, freeing the shell to accept further commands.
- *Tracking:* Each background process is assigned a job ID and stored in a dictionary (background_jobs) along with its Process ID (PID) for tracking.

Commands:

- **jobs:** Lists all active background processes with their job IDs and PIDs.
- **fg [job_id]:** Brings a background job to the foreground using process.wait() to block the shell until the process completes.
- **bg [job_id]**: Resumes a stopped background job by interacting with the process's state.

Error Handling:

The shell includes robust error handling to manage invalid commands, arguments, and system failures gracefully.

Invalid Commands:

- If a user enters a command that is not recognized, the shell prints an error message:
- [command]: command not found
- Invalid Inputs for Built-in Commands:

Process Errors:

Commands like cd, cat, rm, and mkdir check for missing or incorrect arguments. For Example:

- cd without a directory argument displays: cd: missing argument

- cat for a non-existent file displays: cat: [filename]: No such file or directory
- For commands like kill [pid], invalid or non-existent PIDs are caught and reported: kill: [pid]: No such process

General Exceptions:

- Unanticipated errors are caught in the main loop and displayed with a generic error message: cError: [exception message]

Challenges and Improvements:

Process Tracking: Managing background processes requires a custom job control system. Assigning job IDs and ensuring they mapped correctly to PIDs was initially error-prone.

Testing Background Processes: Simulating and validating background process behavior required careful testing and debugging, as these processes often interacted with real system resources.

Error Feedback: Balancing user-friendly error messages with detailed debugging information was challenging, especially for invalid inputs or system call failures.

Cross-Platform Compatibility: Certain features like clear and file path handling required adjusting to support Windows and Unix-like systems.

Job Control Enhancements: Implemented a dictionary to track background jobs, enabling features like jobs, fg, and bg.

Error Handling Refinements: Added specific error messages for all commands, ensuring users receive clear feedback on invalid inputs.

Code Modularity: Refactored the shell into modular components (built_in.py, file_ops.py, etc.) for maintainability and scalability.

Testing Framework: Created a test script to systematically validate all shell features, identifying and addressing edge cases during development.

Conclusion

Implementing the basic shell successfully demonstrates process management, error handling, and modular design principles. While challenges arose during job control and cross-platform compatibility development, these were addressed through iterative testing and refactoring. The resulting shell is functional, user-friendly, and extensible for future enhancements.