Advanced Shell Simulation with Integrated OS Concepts: Deliverable 1 – Basic Shell Implementation and Process Management

Carter Nelson

University of the Cumberlands

MSCS 630–AO1 - Advanced Operating Systems

Professor Dax Bradley

September 2025

Introduction

This project is the first deliverable in the development of an advanced shell that simulates a Unix-like operating system. Unlike standard shells that rely on the underlying operating system for process management, memory handling, and synchronization, this custom shell internally manages these features at a higher level. Deliverable 1 focuses on implementing built-in commands, process management, and basic job control. The objective is to provide users with an interactive environment that resembles traditional shells while also offering insight into how operating systems manage processes.

Implementation

The shell was implemented in Python due to its readability, cross-platform support, and built-in libraries for process and file management. User input is parsed and executed either as a built-in command or by launching an external process. Foreground and background process execution were key aspects of this deliverable, requiring the use of Python's os and subprocess modules.

Built-in Commands

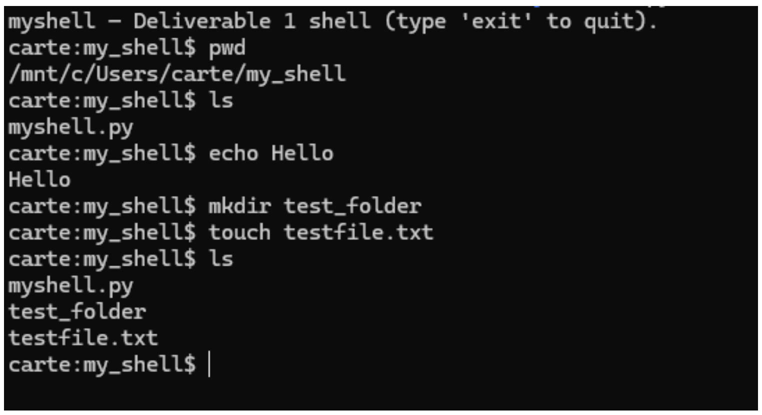
The following built-in commands were implemented:

- cd [directory]: Change the working directory.
- pwd: Display the current working directory.
- exit: Terminate the shell session.
- echo [text]: Print text to the terminal.
- clear: Clear the terminal screen.
- ls: List the files in the current directory.
- cat [filename]: Display the contents of a file.

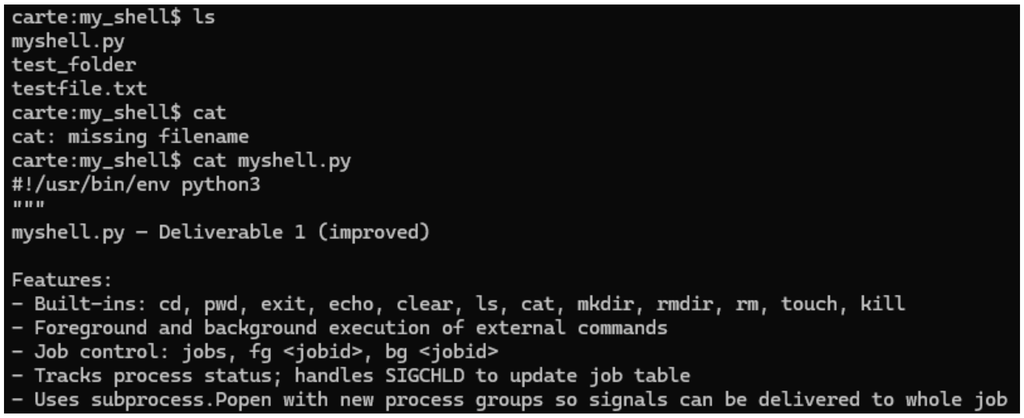
- mkdir [directory]: Create a new directory.
- rmdir [directory]: Remove an empty directory.
- rm [filename]: Remove a file.
- touch [filename]: Create or update the timestamp of a file.
- kill [pid]: Terminate a process by process ID.

These commands were implemented using Python's os, shutil, and file handling methods. Utility commands such as echo and clear were implemented directly in the shell without external calls.

Screenshot Placeholder – Execution of Built-in Commands in PowerShell



```
myshell - Deliverable 1 shell (type 'exit' to quit).
carte:my_shell$ pwd
/mnt/c/Users/carte/my_shell
carte:my_shell$ ls
myshell.py
carte:my_shell$ echo Hello
Hello
carte:my_shell$ mkdir test_folder
carte:my_shell$ touch testfile.txt
carte:my_shell$ ls
myshell.py
test_folder
testfile.txt
carte:my_shell$ |
```



```
carte:my_shell$ ls
myshell.py
test_folder
testfile.txt
carte:my_shell$ cat
cat: missing filename
carte:my_shell$ cat myshell.py
#!/usr/bin/env python3
"""
myshell.py - Deliverable 1 (improved)

Features:
- Built-ins: cd, pwd, exit, echo, clear, ls, cat, mkdir, rmdir, rm, touch, kill
- Foreground and background execution of external commands
- Job control: jobs, fg <jobid>, bg <jobid>
- Tracks process status; handles SIGCHLD to update job table
- Uses subprocess.Popen with new process groups so signals can be delivered to whole job
```

Process Management

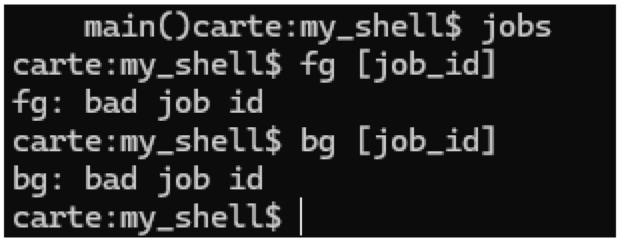
Foreground processes run until completion and block the shell. Background processes, invoked with an ampersand (&), allow the shell to continue accepting commands while the process runs. A job table was implemented to keep track of running background processes, each assigned a job ID and process ID.

The following job control commands were included:

- jobs: Display active background processes.
- fg [job_id]: Bring a background process into the foreground.
- bg [job_id]: Resume a suspended job in the background.

This simulation of process management provides insight into how operating systems track and control user-level tasks.

Screenshot Placeholder – Background Process Execution and Job Control in PowerShell

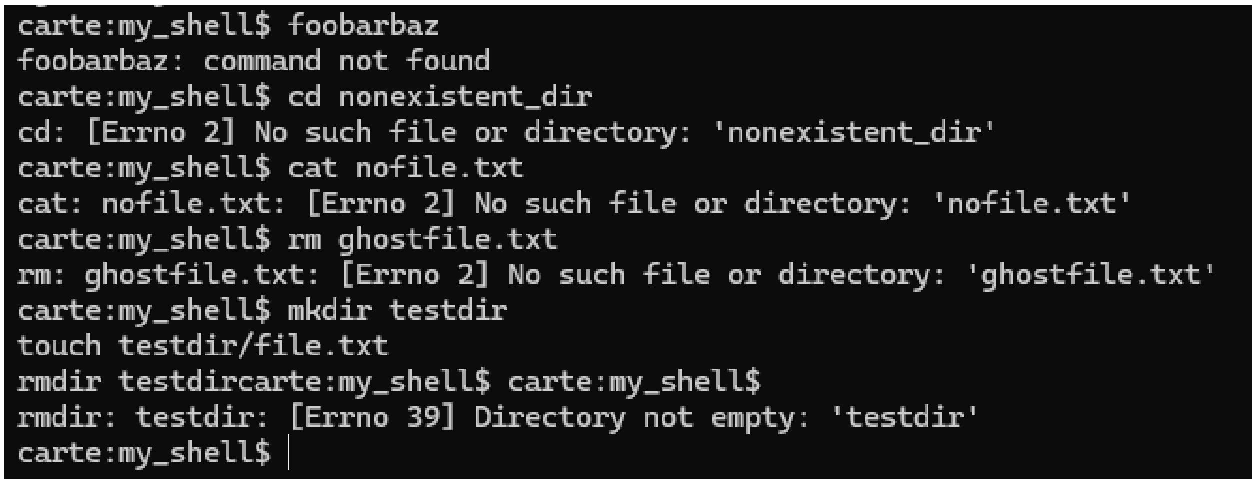


```
    main()carte:my_shell$ jobs
carte:my_shell$ fg [job_id]
fg: bad job id
carte:my_shell$ bg [job_id]
bg: bad job id
carte:my_shell$
```

Error Handling

Robust error handling was implemented to ensure smooth shell operation. Invalid commands produce an error message (“Command not found”), file-related errors provide descriptive feedback, and process management errors are reported without crashing the shell. This approach ensures that user mistakes do not terminate the session and helps maintain usability.

Screenshot Placeholder – Invalid Command/Error Handling in PowerShell



```
carte:my_shell$ foobarbaz
foobarbaz: command not found
carte:my_shell$ cd nonexistent_dir
cd: [Errno 2] No such file or directory: 'nonexistent_dir'
carte:my_shell$ cat nofile.txt
cat: nofile.txt: [Errno 2] No such file or directory: 'nofile.txt'
carte:my_shell$ rm ghostfile.txt
rm: ghostfile.txt: [Errno 2] No such file or directory: 'ghostfile.txt'
carte:my_shell$ mkdir testdir
touch testdir/file.txt
rmdir testdircarte:my_shell$ carte:my_shell$
rmdir: testdir: [Errno 39] Directory not empty: 'testdir'
carte:my_shell$ |
```

Challenges and Improvements

One of the major challenges was accurately simulating job control. Mapping process IDs to job IDs required careful data structure design. Synchronization between foreground and background processes also presented difficulties, as processes must be properly tracked and resumed when necessary.

Another challenge was cross-platform compatibility. Some process management functions behave differently across Unix-like systems and Windows. Python's subprocess module provided a reliable, portable solution, but additional adjustments may be necessary for full compatibility.

Future improvements include:

- Enhanced signal handling for Ctrl+C (interrupt) and Ctrl+Z (suspend).
- Adding command history and tab completion for usability.
- Expanding process control to include more detailed status information.
- Integrating memory management and synchronization concepts in later deliverables.

Conclusion

Deliverable 1 successfully implemented a functional shell with built-in commands, process management, and basic job control. The shell demonstrates how operating systems handle processes and provides an interactive environment that highlights key OS concepts. The work completed in this deliverable forms the foundation for future deliverables that will expand the shell to simulate memory management, synchronization, and security mechanisms.

References

Kerrisk, M. (2010). *The Linux Programming Interface*. No Starch Press.

Python Software Foundation. (2025). *subprocess — Subprocess management*. Retrieved from https://docs.python.org/3/library/subprocess.html

Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). *Operating System Concepts* (10th ed.). Wiley.