Lab Report: Process Management in Python

Experiment: 1 - Process Creation and Management Using Python **Date:** September 28, 2025

1. Objectives

The objective of this experiment was to simulate and understand fundamental Linux process management operations using Python. The tasks focused on process creation, command execution, handling special process states (zombie and orphan), inspecting process information via the `/proc` filesystem, and observing the effects of scheduler prioritization using `nice` values. This was achieved using the modern `subprocess` module, which is well-suited for stable execution within a Jupyter Notebook environment.

2. Code Implementation & Snapshots

The experiment was conducted in a Jupyter Notebook, with helper scripts created dynamically for complex simulations.

Task 1: Process Creation

Independent child processes were created using `subprocess.Popen`. Each child was a new Python interpreter instance that printed its own PID and its parent's PID.

```
# Snapshot from Task 1
command = f"import os, time; print(f'[Child {i+1}] My PID is
{{os.getpid()}} ...')"
proc = subprocess.Popen(["python3", "-c", command])
processes.append(proc)
```

Task 2: Command Execution

System commands were executed using `subprocess.run`, which provides a simple and blocking way to run a command and capture its output.

```
# Snapshot from Task 2
result = subprocess.run(
    cmd_str.split(),
    capture_output=True,
    text=True
)
print(result.stdout)
```

Task 3: Zombie & Orphan Processes

Helper scripts were created using the `%%writefile` magic command.

- **Zombie:** A script (`zombie_maker.py`) used `os.fork()` and had the parent `time.sleep()` without calling `os.wait()`, leaving the exited child in a defunct state.
- **Orphan:** A parent script (`orphan_maker.py`) launched a long-running child script (`long_running_child.py`) and exited immediately. The child then logged its new parent PID after a delay.

Task 4: Inspecting `/proc`

A Python script read process information directly from the virtual `/proc` filesystem for a given PID, parsing files like `/proc/[pid]/status` to extract details.

Task 5: Process Prioritization

A robust helper script (`cpu_eater_robust.py`) was created to perform a CPU-intensive task. The main notebook cell launched multiple instances of this script with different `nice` values passed as command-line arguments. The script included error handling to gracefully manage permission errors when setting high priorities.

---

3. Results and Discussion

The experiment successfully demonstrated all targeted concepts.

- [cite_start]**Task 1 & 2:** The outputs show that the parent-child relationship was correctly established using `subprocess`[cite: 3]. [cite_start]System commands were executed flawlessly, and their standard output was captured as expected[cite: 5].
- **Task 3:** The zombie and orphan states were simulated successfully. [cite_start]The orphan process output shows it was correctly adopted by the `init` process (PID 1), which is the designated process for reclaiming orphaned children[cite: 15]. The zombie simulation required checking the process table from a separate terminal and worked as expected.
- [cite_start]**Task 4:** The script successfully read and displayed data like process name, state, and memory usage from `/proc`[cite: 17]. The "Executable Path: Not accessible" message is a realistic result, demonstrating that permissions within `/proc` can be restricted even for a user's own processes.
- **Task 5:** The prioritization script demonstrated robust error handling. [cite_start]The process that requested a `nice` value of -10 (high priority) was denied permission as a non-root user and instead ran at the default priority of 0[cite: 21]. This is the correct OS behavior. Interestingly, the execution durations for `nice=0` and `nice=15` were very close. This is likely due to the short duration of the task and the efficiency of the Linux scheduler on a modern multi-core system, where it has enough resources to service lower-priority tasks without significant delay.

---

4. Conclusion

This experiment provided a practical and robust demonstration of Linux process management concepts using Python. By leveraging the `subprocess` module and helper scripts, it was possible to reliably simulate process creation, state transitions, and scheduling behavior within a Jupyter Notebook environment, meeting all the assignment's objectives.