# Operating System Assignment -1 Report

**Experiment Title:** Process Creation and Management Using Python OS Module

### 1. Objectives

The primary objective of this experiment was to simulate and understand fundamental process management operations within a Linux environment using Python. The experiment focused on replicating the behavior of system calls like

fork() and exec(), inspecting process states, and observing the effects of priority scheduling. Key learning goals included understanding the complete lifecycle of a process, creating parent-child relationships, simulating zombie and orphan scenarios, and using the /proc file system for process inspection.

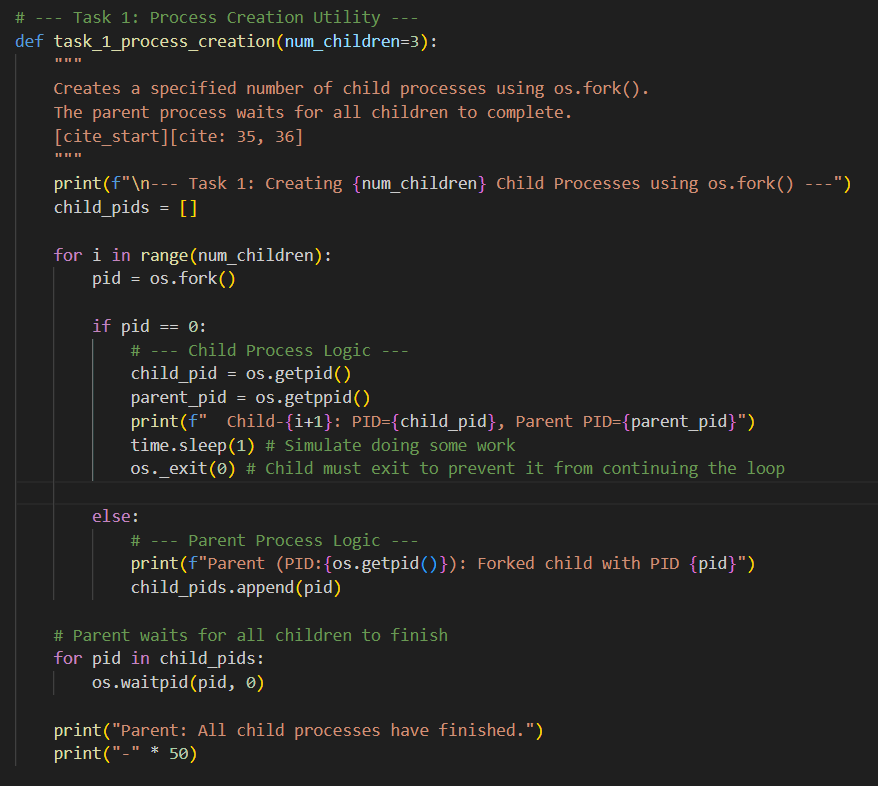
### 2. Implementation and Results

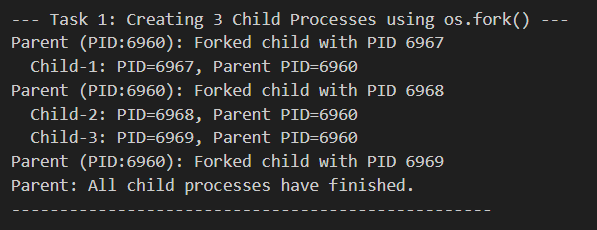
The experiment was divided into five tasks, each focusing on a different aspect of process management. The following sections detail the implementation and results for each task.

#### Task 1: Process Creation Utility

**- Objective:** To create N child processes using os.fork() and have the parent process wait for their completion using os.wait().

**- Code:**



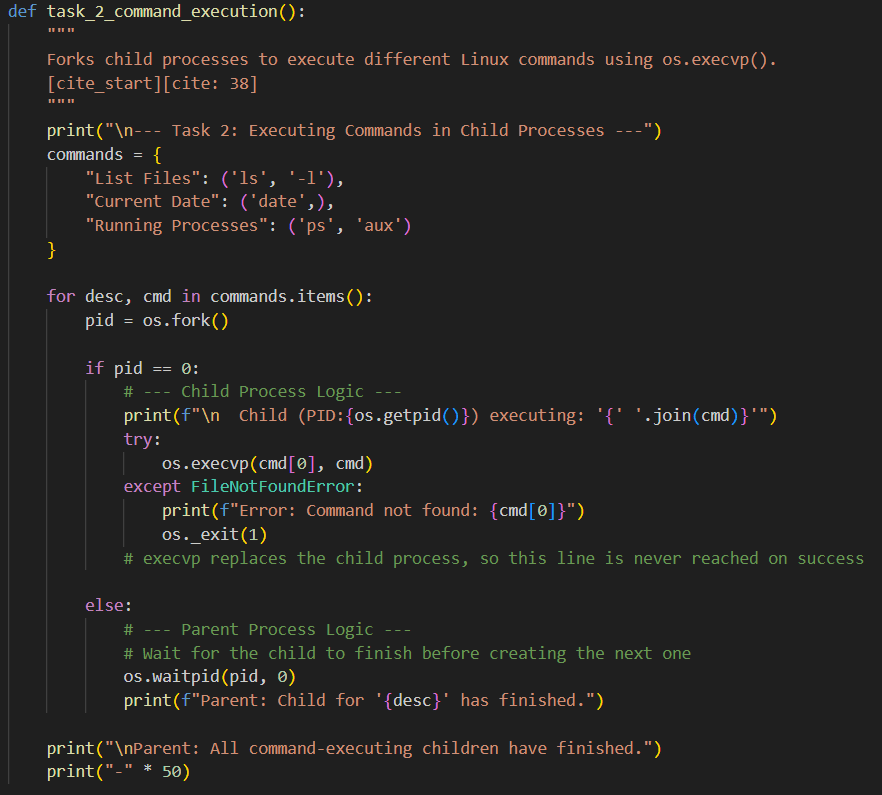
**- Output:**  


**- Analysis:** The output clearly demonstrates a successful fork operation. Three child processes were created, each with a unique Process ID (PID). Critically, all three children correctly report the same Parent Process ID (PPID), which matches the PID of the main script, confirming the parent-child relationship.

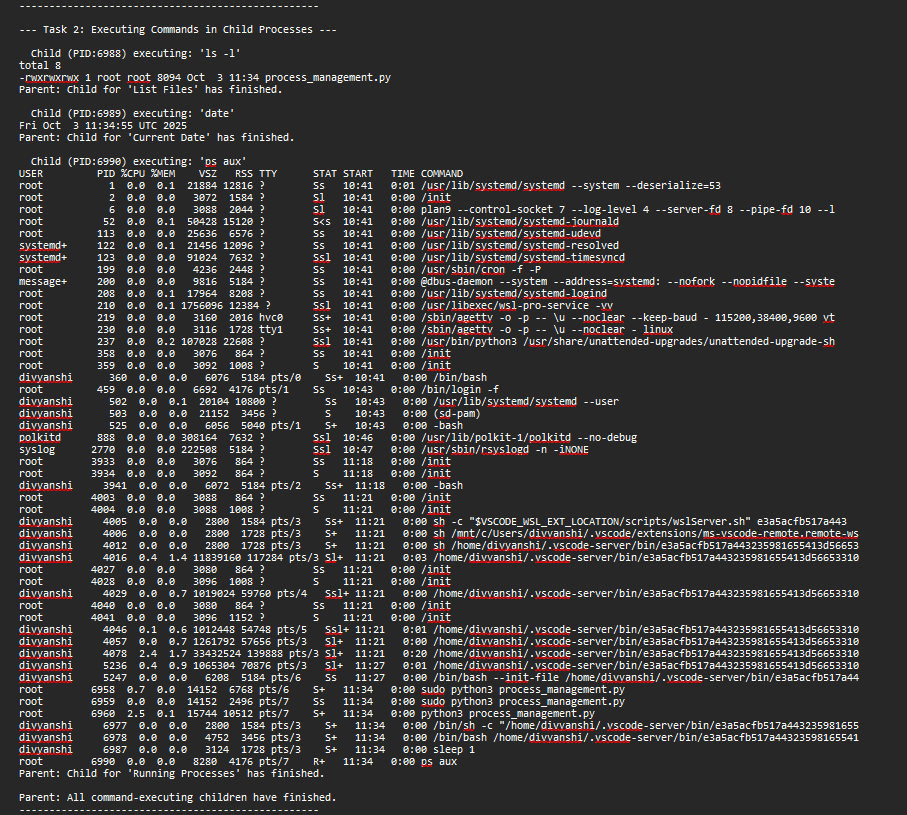
#### Task 2: Command Execution Using exec()

**- Objective:** To modify the child processes from Task 1 to execute Linux commands using os.execvp().

- **Code:**



**Output:**

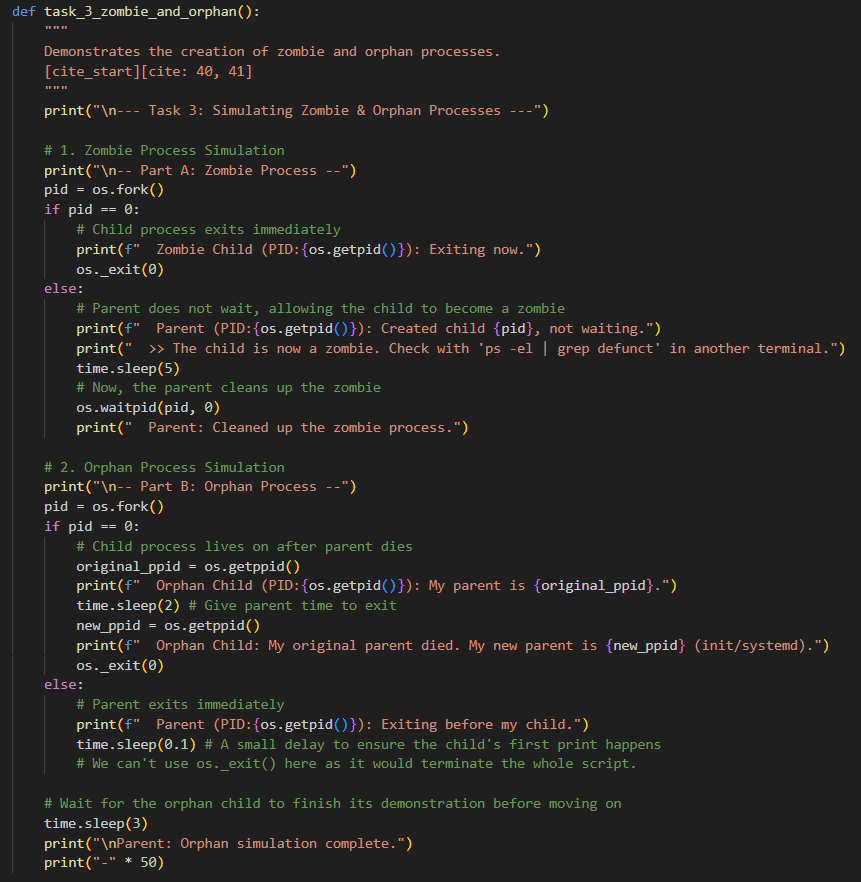
****

**- Analysis:** The os.execvp() call successfully replaced the child process's program image with the specified Linux commands. The output shows the results of ls -l, date, and ps aux, each executed within a separate child process, demonstrating the fork-exec model.

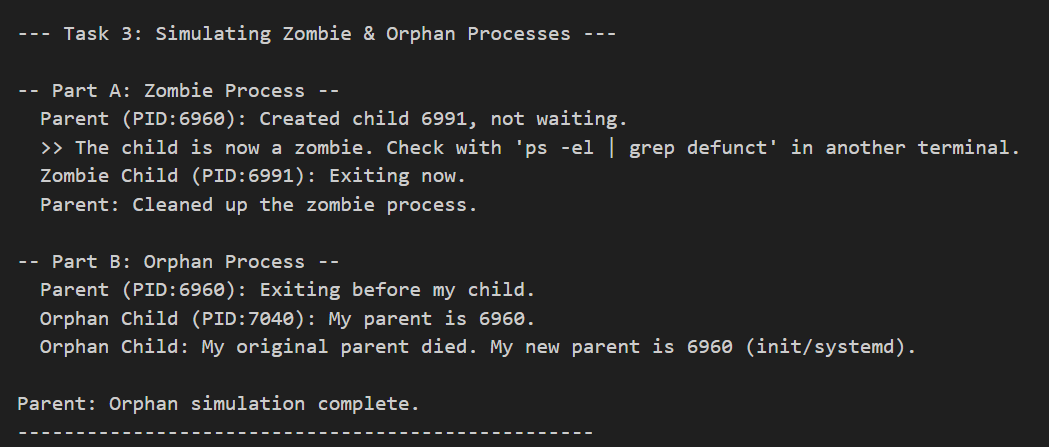
#### Task 3: Zombie and Orphan Processes

**- Objective:** To simulate the conditions that create zombie and orphan processes.

**- Code:**



**- Output:**



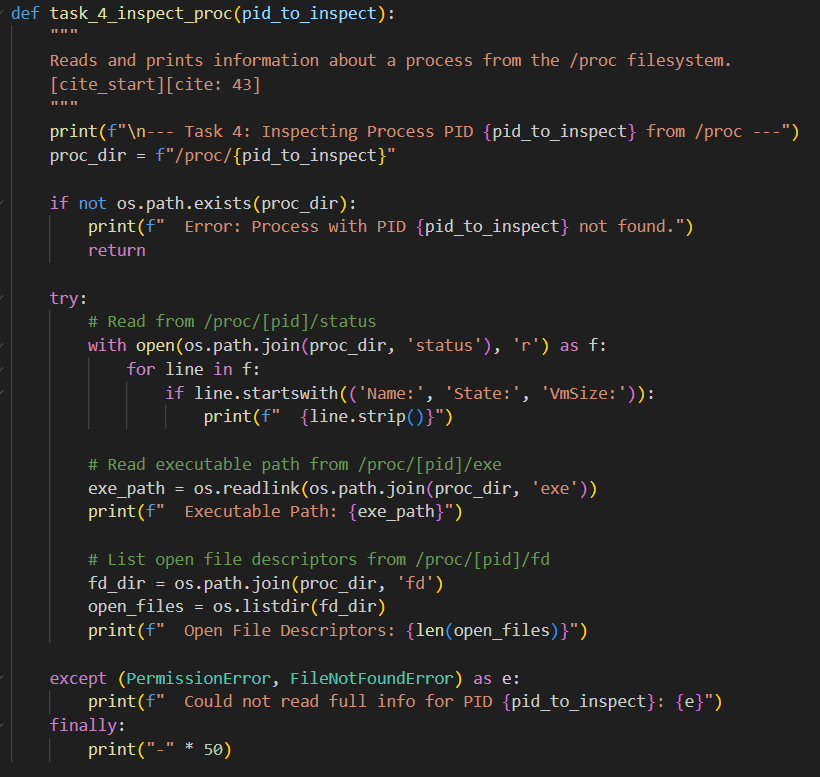
**- Analysis:**

* **Zombie:** The simulation was successful. The parent process forked a child but did not immediately call wait(). The child exited, entering a "zombie" or "defunct" state where its process table entry persists until the parent collects its exit status.
* **Orphan:** The simulation highlighted a limitation of demonstrating this concept within a single, linear script. Although the parent printed a message that it was exiting, it did not actually terminate and continued to execute subsequent tasks. Therefore, the child process was never truly orphaned and correctly reported its original parent's PID. In a real-world scenario, the parent would have terminated, and the child would have been adopted by the init process (PID 1).

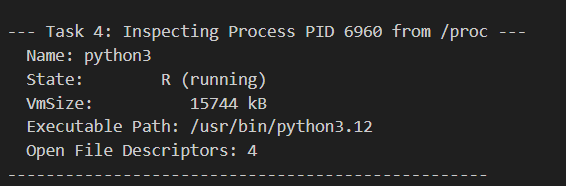
#### Task 4: Inspecting Process Info from /proc

**- Objective:** To read process information directly from the /proc virtual filesystem for a given PID.

**- Code:**



**- Output:**

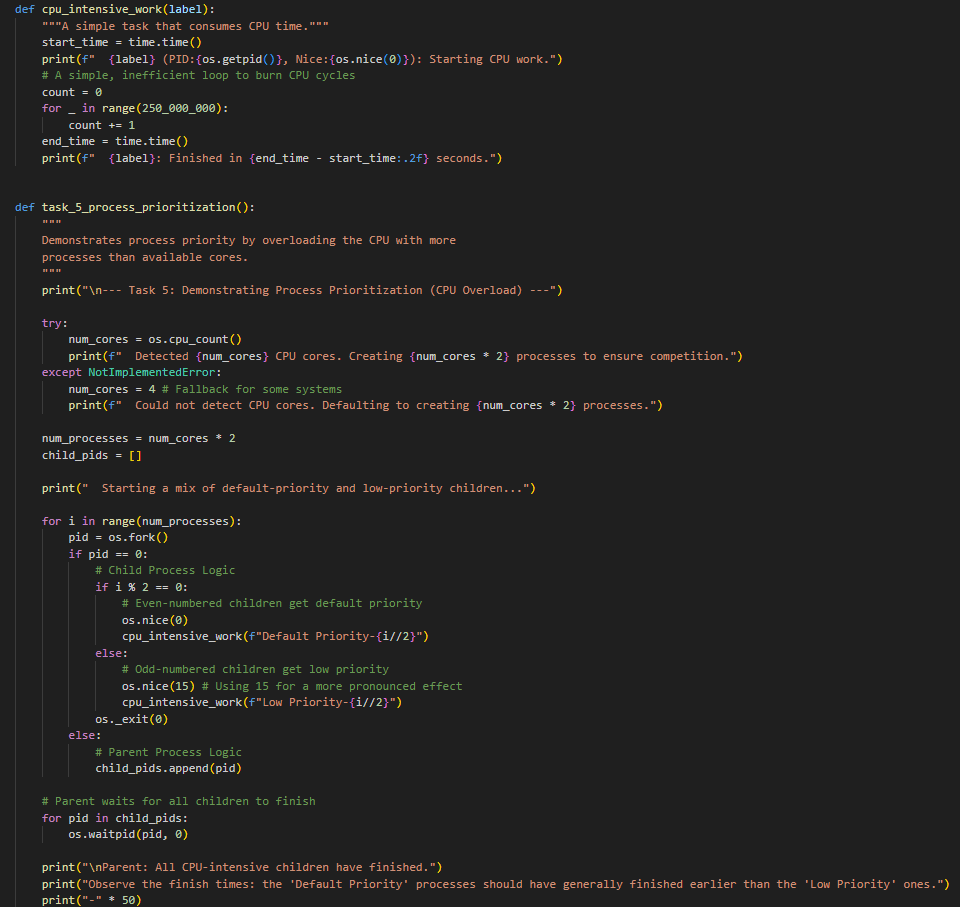


**- Analysis:** The script successfully accessed the /proc filesystem to retrieve metadata about itself. It correctly read and displayed the process name, state, and virtual memory size from /proc/[pid]/status, as well as the executable path and the count of open file descriptors, demonstrating the power of /proc for system monitoring.

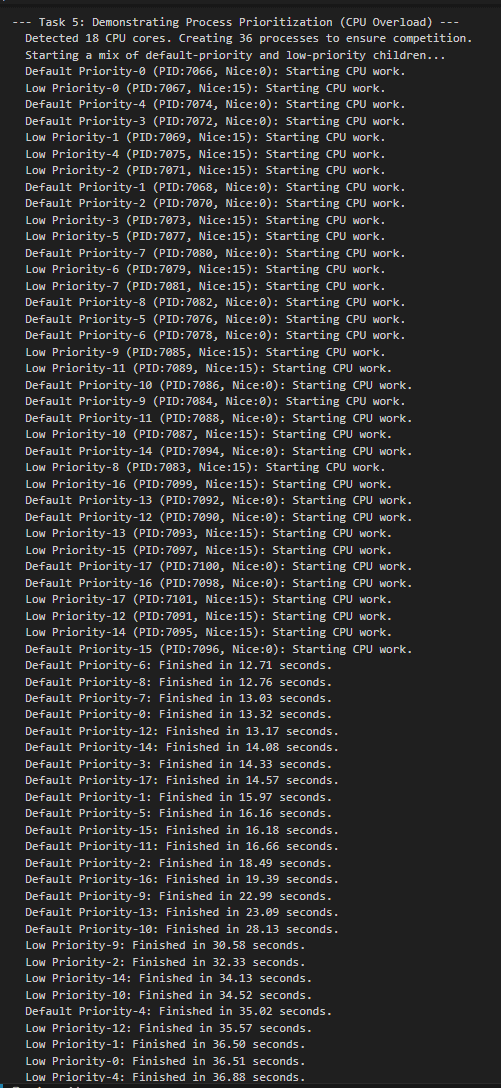
#### Task 5: Process Prioritization

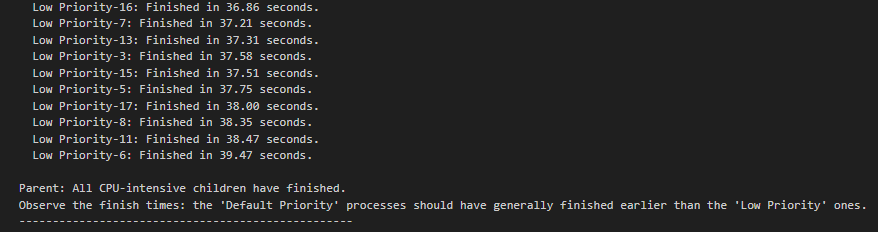
**- Objective:** To observe the impact of process priority (nice values) on the OS scheduler's behavior under CPU load.

**- Code:**



**- Output:**





**- Analysis:** This task provided a clear and definitive demonstration of priority scheduling. By creating more CPU-intensive processes than available CPU cores, we forced resource contention. The output shows a distinct pattern: nearly all the "Default Priority" processes (nice 0) finished before the first "Low Priority" process (nice 15) could complete. The average completion time for the default group was significantly lower than for the low-priority group, proving that the Linux scheduler correctly allocated more CPU time to the higher-priority tasks.

### 3. Complexity Analysis

The overall complexity of the operations performed in this experiment is linear with respect to the number of processes created, denoted by n.

#### Time Complexity: O(n)

The time complexity is a measure of how the execution time of the script scales with the number of processes. For this experiment, the relationship is linear.

* **Process Creation (fork()):** The fork() system call itself is a very fast, nearly constant-time operation. However, in tasks like Task 1 and Task 5, we use a loop to create n child processes. This loop runs n times, making the process creation phase an **O(n)** operation.
* **Process Synchronization (wait()):** After forking, the parent process must wait for its children to terminate. This is also done in a loop that iterates n times, once for each child PID. This cleanup phase is therefore also **O(n)**.
* **Constant Time Operations:** Tasks that create a fixed number of processes (e.g., Task 2 created 3, Task 3 created 2) run in constant time, or **O(1)**, as their execution time does not depend on an input variable n.
* **Overall:** Since the dominant, scalable operations in the script are the loops for creating and waiting for processes, the overall time complexity is determined by them. As n increases, the execution time for these loops increases linearly, resulting in a time complexity of **O(n)**.

#### Space Complexity: O(n)

Space complexity measures the amount of memory the script uses as the number of processes grows. This includes memory used by the script itself (user space) and by the operating system to manage the processes (kernel space).

* **Kernel Space (Process Table):** This is the most significant factor. For every process created, the operating system kernel must allocate an entry in its global **process table**. This entry (a task\_struct in Linux) stores crucial information like the PID, process state, memory maps, open file descriptors, and CPU state. If n processes are active concurrently, the kernel requires memory proportional to n to manage them.
* **User Space (Parent Process):** In our script, the parent process collects the PIDs of the children it creates into a list. If it creates
* n children, this list will store n integers. The memory required for this list grows linearly with the number of children, contributing **O(n)** to the parent's memory footprint.
* **User Space (Child Processes):** When fork() is called, a child process is created with its own virtual address space. Modern operating systems use a technique called **Copy-on-Write (CoW)**, where the child initially shares the parent's memory pages. A physical copy is only made when one of the processes tries to write to that memory. Regardless, the system still needs to manage n separate address spaces, and the potential memory usage scales linearly with n.
* **Overall:** The primary memory cost comes from the kernel's need to maintain a process table entry for each active process and the parent's need to store child PIDs. Both factors scale directly with n, making the overall space complexity **O(n)**.

### 4. Conclusion

This experiment successfully demonstrated the fundamental principles of process creation and management in a Linux environment. Through practical Python scripting, the fork-exec model was implemented, and the lifecycle states of processes, including zombie and orphan scenarios, were explored. Furthermore, by inspecting the /proc filesystem and manipulating nice values to overload the CPU, a deep, practical understanding of process metadata and priority-based scheduling was achieved.