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:

```python
# --- Task 1: Process Creation Utility ---
def task_1_process_creation(num_children=3):
    """
    Creates a specified number of child processes using os.fork().
    The parent process waits for all children to complete.
    [cite_start][cite: 35, 36]
    """
    print(f"\n--- Task 1: Creating {num_children} Child Processes using os.fork() ---")
    child_pids = []

    for i in range(num_children):
        pid = os.fork()

        if pid == 0:
            # --- Child Process Logic ---
            child_pid = os.getpid()
            parent_pid = os.getppid()
            print(f"  Child-{i+1}: PID={child_pid}, Parent PID={parent_pid}")
            time.sleep(1) # Simulate doing some work
            os._exit(0) # Child must exit to prevent it from continuing the loop

        else:
            # --- Parent Process Logic ---
            print(f"Parent (PID:{os.getpid()}): Forked child with PID {pid}")
            child_pids.append(pid)

    # Parent waits for all children to finish
    for pid in child_pids:
        os.waitpid(pid, 0)

    print("Parent: All child processes have finished.")
    print("-" * 50)
```

- Output:

```
--- Task 1: Creating 3 Child Processes using os.fork() ---
Parent (PID:6960): Forked child with PID 6967
  Child-1: PID=6967, Parent PID=6960
Parent (PID:6960): Forked child with PID 6968
  Child-2: PID=6968, Parent PID=6960
  Child-3: PID=6969, Parent PID=6960
Parent (PID:6960): Forked child with PID 6969
Parent: All child processes have finished.
--------------------------------------------------
```

- 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:**

```python
def task_2_command_execution():
    """
    Forks child processes to execute different Linux commands using os.execvp().
    [cite_start][cite: 38]
    """
    print("\n--- Task 2: Executing Commands in Child Processes ---")
    commands = {
        "List Files": ('ls', '-l'),
        "Current Date": ('date',),
        "Running Processes": ('ps', 'aux')
    }

    for desc, cmd in commands.items():
        pid = os.fork()

        if pid == 0:
            # --- Child Process Logic ---
            print(f"\n  Child (PID:{os.getpid()}) executing: '{' '.join(cmd)}'")
            try:
                os.execvp(cmd[0], cmd)
            except FileNotFoundError:
                print(f"Error: Command not found: {cmd[0]}")
                os._exit(1)
            # execvp replaces the child process, so this line is never reached on success

        else:
            # --- Parent Process Logic ---
            # Wait for the child to finish before creating the next one
            os.waitpid(pid, 0)
            print(f"Parent: Child for '{desc}' has finished.")

    print("\nParent: All command-executing children have finished.")
    print("-" * 50)
```

Output:

```
--- Task 2: Executing Commands in Child Processes ---

  Child (PID:6988) executing: 'ls -l'
total 8
-rwxrwxrwx 1 root root 8094 Oct  3 11:34 process_management.py
Parent: Child for 'List Files' has finished.

  Child (PID:6989) executing: 'date'
Fri Oct  3 11:34:55 UTC 2025
Parent: Child for 'Current Date' has finished.

  Child (PID:6990) executing: 'ps aux'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1  21884 12816 ?        Ss   10:41   0:01 /usr/lib/systemd/systemd --system --deserialize=53
root           2  0.0  0.0   3072  1584 ?        Sl   10:41   0:00 /init
root           6  0.0  0.0   3088  2044 ?        Sl   10:41   0:00 plan9 --control-socket 7 --log-level 4 --server-fd 8 --pipe-fd 10 --l
root          52  0.0  0.1  50428 15120 ?        S<s  10:41   0:00 /usr/lib/systemd/systemd-journald
root         113  0.0  0.0  25636  6576 ?        Ss   10:41   0:00 /usr/lib/systemd/systemd-udevd
systemd+     122  0.0  0.1  21456 12096 ?        Ss   10:41   0:00 /usr/lib/systemd/systemd-resolved
systemd+     123  0.0  0.0  91024  7632 ?        Ssl  10:41   0:00 /usr/lib/systemd/systemd-timesyncd
root         199  0.0  0.0   4236  2448 ?        Ss   10:41   0:00 /usr/sbin/cron -f -P
message+     200  0.0  0.0   9816  5184 ?        Ss   10:41   0:00 @dbus-daemon --system --address=systemd: --nofork --nopidfile --syste
root         208  0.0  0.1  17964  8208 ?        Ss   10:41   0:00 /usr/lib/systemd/systemd-logind
root         210  0.0  0.1 1756096 12384 ?       Ssl  10:41   0:00 /usr/libexec/wsl-pro-service -vv
root         219  0.0  0.0   3160  2016 hvc0     Ss+  10:41   0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud - 115200,38400,9600 vt
root         230  0.0  0.0   3116  1728 tty1     Ss+  10:41   0:00 /sbin/agetty -o -p -- \u --noclear - linux
root         237  0.0  0.2 107028 22608 ?        Ssl  10:41   0:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-sh
root         358  0.0  0.0   3076   864 ?        Ss   10:41   0:00 /init
root         359  0.0  0.0   3092  1008 ?        S    10:41   0:00 /init
abhinav      360  0.0  0.0   6076  5184 pts/0    Ss+  10:41   0:00 /bin/bash
root         459  0.0  0.0   6692  4176 pts/1    Ss   10:43   0:00 /bin/login -f
abhinav      502  0.0  0.1  20104 10800 ?        Ss   10:43   0:00 /usr/lib/systemd/systemd --user
abhinav      503  0.0  0.0  21152  3456 ?        S    10:43   0:00 (sd-pam)
abhinav      525  0.0  0.0   6056  5040 pts/1    S+   10:43   0:00 -bash
polkitd      888  0.0  0.0 308164  7632 ?        Ssl  10:46   0:00 /usr/lib/polkit-1/polkitd --no-debug
syslog      2770  0.0  0.0 222508  5184 ?        Ssl  10:47   0:00 /usr/sbin/rsyslogd -n -iNONE
root        3933  0.0  0.0   3076   864 ?        Ss   11:18   0:00 /init
root        3934  0.0  0.0   3092   864 ?        S    11:18   0:00 /init
abhinav     3941  0.0  0.0   6072  5184 pts/2    Ss+  11:18   0:00 -bash
root        4003  0.0  0.0   3088   864 ?        Ss   11:21   0:00 /init
root        4004  0.0  0.0   3088  1008 ?        S    11:21   0:00 /init
abhinav     4005  0.0  0.0   2800  1584 pts/3    Ss+  11:21   0:00 sh -c "$VSCODE_WSL_EXT_LOCATION/scripts/wslServer.sh" e3a5acfb517a443
abhinav     4006  0.0  0.0   2800  1728 pts/3    S+   11:21   0:00 sh /mnt/c/Users/Abhinav/.vscode/extensions/ms-vscode-remote.remote-ws
abhinav     4012  0.0  0.0   2800  1728 pts/3    S+   11:21   0:00 sh /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653
abhinav     4016  0.4  1.4 11839160 117284 pts/3 Sl+  11:21   0:03 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
root        4027  0.0  0.0   3080   864 ?        Ss   11:21   0:00 /init
root        4028  0.0  0.0   3096  1008 ?        S    11:21   0:00 /init
abhinav     4029  0.0  0.7 1019024 59760 pts/4   Ssl+ 11:21   0:00 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
root        4040  0.0  0.0   3080   864 ?        Ss   11:21   0:00 /init
root        4041  0.0  0.0   3096  1152 ?        S    11:21   0:00 /init
abhinav     4046  0.1  0.6 1012448 54748 pts/5   Ssl+ 11:21   0:01 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
abhinav     4057  0.0  0.7 1261792 57656 pts/3   Sl+  11:21   0:00 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
abhinav     4078  2.4  1.7 33432524 139888 pts/3 Sl+  11:21   0:20 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
abhinav     5236  0.4  0.9 1065304 70876 pts/3   Sl+  11:27   0:01 /home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655413d56653310
abhinav     5247  0.0  0.0   6208  5184 pts/6    Ss   11:27   0:00 /bin/bash --init-file /home/abhinav/.vscode-server/bin/e3a5acfb517a44
root        6958  0.7  0.0  14152  6768 pts/6    S+   11:34   0:00 sudo python3 process_management.py
root        6959  0.0  0.0  14152  2496 pts/7    Ss   11:34   0:00 sudo python3 process_management.py
root        6960  2.5  0.1  15744 10512 pts/7    S+   11:34   0:00 python3 process_management.py
abhinav     6977  0.0  0.0   2800  1584 pts/3    S+   11:34   0:00 /bin/sh -c "/home/abhinav/.vscode-server/bin/e3a5acfb517a443235981655
abhinav     6978  0.0  0.0   4752  3456 pts/3    S+   11:34   0:00 /bin/bash /home/abhinav/.vscode-server/bin/e3a5acfb517a44323598165541
abhinav     6987  0.0  0.0   3124  1728 pts/3    S+   11:34   0:00 sleep 1
root        6990  0.0  0.0   8280  4176 pts/7    R+   11:34   0:00 ps aux
Parent: Child for 'Running Processes' has finished.

Parent: All command-executing children have finished.
--------------------------------------------------
```

- 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:

```python
def task_3_zombie_and_orphan():
    """
    Demonstrates the creation of zombie and orphan processes.
    [cite_start][cite: 40, 41]
    """
    print("\n--- Task 3: Simulating Zombie & Orphan Processes ---")

    # 1. Zombie Process Simulation
    print("\n-- Part A: Zombie Process --")
    pid = os.fork()
    if pid == 0:
        # Child process exits immediately
        print(f"  Zombie Child (PID:{os.getpid()}): Exiting now.")
        os._exit(0)
    else:
        # Parent does not wait, allowing the child to become a zombie
        print(f"  Parent (PID:{os.getpid()}): Created child {pid}, not waiting.")
        print("  >> The child is now a zombie. Check with 'ps -el | grep defunct' in another terminal.")
        time.sleep(5)
        # Now, the parent cleans up the zombie
        os.waitpid(pid, 0)
        print("  Parent: Cleaned up the zombie process.")

    # 2. Orphan Process Simulation
    print("\n-- Part B: Orphan Process --")
    pid = os.fork()
    if pid == 0:
        # Child process lives on after parent dies
        original_ppid = os.getppid()
        print(f"  Orphan Child (PID:{os.getpid()}): My parent is {original_ppid}.")
        time.sleep(2) # Give parent time to exit
        new_ppid = os.getppid()
        print(f"  Orphan Child: My original parent died. My new parent is {new_ppid} (init/systemd).")
        os._exit(0)
    else:
        # Parent exits immediately
        print(f"  Parent (PID:{os.getpid()}): Exiting before my child.")
        time.sleep(0.1) # A small delay to ensure the child's first print happens
        # We can't use os._exit() here as it would terminate the whole script.

    # Wait for the orphan child to finish its demonstration before moving on
    time.sleep(3)
    print("\nParent: Orphan simulation complete.")
    print("-" * 50)
```

- Output:

```
--- Task 3: Simulating Zombie & Orphan Processes ---

-- Part A: Zombie Process --
  Parent (PID:6960): Created child 6991, not waiting.
  >> The child is now a zombie. Check with 'ps -el | grep defunct' in another terminal.
  Zombie Child (PID:6991): Exiting now.
  Parent: Cleaned up the zombie process.

-- Part B: Orphan Process --
  Parent (PID:6960): Exiting before my child.
  Orphan Child (PID:7040): My parent is 6960.
  Orphan Child: My original parent died. My new parent is 6960 (init/systemd).

Parent: Orphan simulation complete.
--------------------------------------------------
```

- 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:**

```python
def task_4_inspect_proc(pid_to_inspect):
    """
    Reads and prints information about a process from the /proc filesystem.
    [cite_start][cite: 43]
    """
    print(f"\n--- Task 4: Inspecting Process PID {pid_to_inspect} from /proc ---")
    proc_dir = f"/proc/{pid_to_inspect}"

    if not os.path.exists(proc_dir):
        print(f"  Error: Process with PID {pid_to_inspect} not found.")
        return

    try:
        # Read from /proc/[pid]/status
        with open(os.path.join(proc_dir, 'status'), 'r') as f:
            for line in f:
                if line.startswith(('Name:', 'State:', 'VmSize:')):
                    print(f"  {line.strip()}")

        # Read executable path from /proc/[pid]/exe
        exe_path = os.readlink(os.path.join(proc_dir, 'exe'))
        print(f"  Executable Path: {exe_path}")

        # List open file descriptors from /proc/[pid]/fd
        fd_dir = os.path.join(proc_dir, 'fd')
        open_files = os.listdir(fd_dir)
        print(f"  Open File Descriptors: {len(open_files)}")

    except (PermissionError, FileNotFoundError) as e:
        print(f"  Could not read full info for PID {pid_to_inspect}: {e}")
    finally:
        print("-" * 50)
```

- **Output:**

```
--- Task 4: Inspecting Process PID 6960 from /proc ---
  Name: python3
  State:        R (running)
  VmSize:          15744 kB
  Executable Path: /usr/bin/python3.12
  Open File Descriptors: 4
--------------------------------------------------
```

- **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:**

```python
def cpu_intensive_work(label):
    """A simple task that consumes CPU time."""
    start_time = time.time()
    print(f"  {label} (PID:{os.getpid()}, Nice:{os.nice(0)}): Starting CPU work.")
    # A simple, inefficient loop to burn CPU cycles
    count = 0
    for _ in range(250_000_000):
        count += 1
    end_time = time.time()
    print(f"  {label}: Finished in {end_time - start_time:.2f} seconds.")


def task_5_process_prioritization():
    """
    Demonstrates process priority by overloading the CPU with more
    processes than available cores.
    """
    print("\n--- Task 5: Demonstrating Process Prioritization (CPU Overload) ---")

    try:
        num_cores = os.cpu_count()
        print(f"  Detected {num_cores} CPU cores. Creating {num_cores * 2} processes to ensure competition.")
    except NotImplementedError:
        num_cores = 4 # Fallback for some systems
        print(f"  Could not detect CPU cores. Defaulting to creating {num_cores * 2} processes.")

    num_processes = num_cores * 2
    child_pids = []

    print("  Starting a mix of default-priority and low-priority children...")

    for i in range(num_processes):
        pid = os.fork()
        if pid == 0:
            # Child Process Logic
            if i % 2 == 0:
                # Even-numbered children get default priority
                os.nice(0)
                cpu_intensive_work(f"Default Priority-{i//2}")
            else:
                # Odd-numbered children get low priority
                os.nice(15) # Using 15 for a more pronounced effect
                cpu_intensive_work(f"Low Priority-{i//2}")
            os._exit(0)
        else:
            # Parent Process Logic
            child_pids.append(pid)

    # Parent waits for all children to finish
    for pid in child_pids:
        os.waitpid(pid, 0)

    print("\nParent: All CPU-intensive children have finished.")
    print("Observe the finish times: the 'Default Priority' processes should have generally finished earlier than the 'Low Priority' ones.")
    print("-" * 50)
```

- Output:

```
--- Task 5: Demonstrating Process Prioritization (CPU Overload) ---
  Detected 18 CPU cores. Creating 36 processes to ensure competition.
  Starting a mix of default-priority and low-priority children...
  Default Priority-0 (PID:7066, Nice:0): Starting CPU work.
  Low Priority-0 (PID:7067, Nice:15): Starting CPU work.
  Default Priority-4 (PID:7074, Nice:0): Starting CPU work.
  Default Priority-3 (PID:7072, Nice:0): Starting CPU work.
  Low Priority-1 (PID:7069, Nice:15): Starting CPU work.
  Low Priority-4 (PID:7075, Nice:15): Starting CPU work.
  Low Priority-2 (PID:7071, Nice:15): Starting CPU work.
  Default Priority-1 (PID:7068, Nice:0): Starting CPU work.
  Default Priority-2 (PID:7070, Nice:0): Starting CPU work.
  Low Priority-3 (PID:7073, Nice:15): Starting CPU work.
  Low Priority-5 (PID:7077, Nice:15): Starting CPU work.
  Default Priority-7 (PID:7080, Nice:0): Starting CPU work.
  Low Priority-6 (PID:7079, Nice:15): Starting CPU work.
  Low Priority-7 (PID:7081, Nice:15): Starting CPU work.
  Default Priority-8 (PID:7082, Nice:0): Starting CPU work.
  Default Priority-5 (PID:7076, Nice:0): Starting CPU work.
  Default Priority-6 (PID:7078, Nice:0): Starting CPU work.
  Low Priority-9 (PID:7085, Nice:15): Starting CPU work.
  Low Priority-11 (PID:7089, Nice:15): Starting CPU work.
  Default Priority-10 (PID:7086, Nice:0): Starting CPU work.
  Default Priority-9 (PID:7084, Nice:0): Starting CPU work.
  Default Priority-11 (PID:7088, Nice:0): Starting CPU work.
  Low Priority-10 (PID:7087, Nice:15): Starting CPU work.
  Default Priority-14 (PID:7094, Nice:0): Starting CPU work.
  Low Priority-8 (PID:7083, Nice:15): Starting CPU work.
  Low Priority-16 (PID:7099, Nice:15): Starting CPU work.
  Default Priority-13 (PID:7092, Nice:0): Starting CPU work.
  Default Priority-12 (PID:7090, Nice:0): Starting CPU work.
  Low Priority-13 (PID:7093, Nice:15): Starting CPU work.
  Low Priority-15 (PID:7097, Nice:15): Starting CPU work.
  Default Priority-17 (PID:7100, Nice:0): Starting CPU work.
  Default Priority-16 (PID:7098, Nice:0): Starting CPU work.
  Low Priority-17 (PID:7101, Nice:15): Starting CPU work.
  Low Priority-12 (PID:7091, Nice:15): Starting CPU work.
  Low Priority-14 (PID:7095, Nice:15): Starting CPU work.
  Default Priority-15 (PID:7096, Nice:0): Starting CPU work.
  Default Priority-6: Finished in 12.71 seconds.
  Default Priority-8: Finished in 12.76 seconds.
  Default Priority-7: Finished in 13.03 seconds.
  Default Priority-0: Finished in 13.32 seconds.
  Default Priority-12: Finished in 13.17 seconds.
  Default Priority-14: Finished in 14.08 seconds.
  Default Priority-3: Finished in 14.33 seconds.
  Default Priority-17: Finished in 14.57 seconds.
  Default Priority-1: Finished in 15.97 seconds.
  Default Priority-5: Finished in 16.16 seconds.
  Default Priority-15: Finished in 16.18 seconds.
  Default Priority-11: Finished in 16.66 seconds.
  Default Priority-2: Finished in 18.49 seconds.
  Default Priority-16: Finished in 19.39 seconds.
  Default Priority-9: Finished in 22.99 seconds.
  Default Priority-13: Finished in 23.09 seconds.
  Default Priority-10: Finished in 28.13 seconds.
  Low Priority-9: Finished in 30.58 seconds.
  Low Priority-2: Finished in 32.33 seconds.
  Low Priority-14: Finished in 34.13 seconds.
  Low Priority-10: Finished in 34.52 seconds.
  Default Priority-4: Finished in 35.02 seconds.
  Low Priority-12: Finished in 35.57 seconds.
  Low Priority-1: Finished in 36.50 seconds.
  Low Priority-0: Finished in 36.51 seconds.
  Low Priority-4: Finished in 36.88 seconds.
```

```
  Low Priority-16: Finished in 36.86 seconds.
  Low Priority-7: Finished in 37.21 seconds.
  Low Priority-13: Finished in 37.31 seconds.
  Low Priority-3: Finished in 37.58 seconds.
  Low Priority-15: Finished in 37.51 seconds.
  Low Priority-5: Finished in 37.75 seconds.
  Low Priority-17: Finished in 38.00 seconds.
  Low Priority-8: Finished in 38.35 seconds.
  Low Priority-11: Finished in 38.47 seconds.
  Low Priority-6: Finished in 39.47 seconds.

Parent: All CPU-intensive children have finished.
Observe the finish times: the 'Default Priority' processes should have generally finished earlier than the 'Low Priority' ones.
--------------------------------------------------
```

- 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.