Lab Report: Process Management in Python

Course: ENCS351 Operating System

Student: Gunjan Joshi

Course : B.Tech CSE

Roll no. 2301010062

1. Objectives

The primary objective of this experiment was to gain practical experience with fundamental process management concepts in a Linux environment. Using Python's os and subprocess modules, we simulated process creation, command execution, and state transitions. The key goals were:

To understand the parent-child relationship in processes using os.fork().

To execute system commands by replacing a process image with os.execvp().

To create and observe special process states, namely **zombie** and **orphan** processes.

To inspect live process data by reading from the /proc virtual filesystem.

To observe the effect of the Linux scheduler by manipulating process priorities with os.nice().

2. Code Implementation and Results

Task 1: Process Creation Utility

Code Snapshot:

```
def task1_create_processes(n):
    child_pids = []
    parent_pid = os.getpid()
    print(f"Parent Process PID: {parent_pid}")

    for i in range(n):
        pid = os.fork()
        if pid == 0: # Child process
            print(f"  -> Child-{i+1}: My PID is {os.getpid()}, My Parent's PID is 
            os._exit(0)
        else: # Parent process
            child_pids.append(pid)

    # Parent waits for all children
    for child_pid in child_pids:
        os.waitpid(child_pid, 0)
    print("Parent: All children have completed.")
```

Results & Analysis: The code successfully forks N child processes. os.fork() returns 0 in the child and the child's PID in the parent. Each child prints its own PID (os.getpid()) and its parent's PID (os.getppid()), confirming the parent-child relationship. The parent process stores the PIDs of its children and uses os.waitpid() to wait for each one to terminate, ensuring proper resource cleanup.

Task 2: Command Execution Using exec()

Code Snapshot:

```
def task2_execute_commands(commands):
    for cmd_str in commands:
        pid = os.fork()
        if pid == 0: # Child process
            print(f"\nChild {os.getpid()}: Executing '{cmd_str}'...")
            try:
                cmd_args = cmd_str.split()
                os.execvp(cmd_args[0], cmd_args)
            except FileNotFoundError:
                os._exit(1)
        else: # Parent process
            os.wait()
```

Results & Analysis: This task demonstrates how a child process can run a new program. After forking, the child process calls os.execvp(). This function replaces the current process's memory space and code with the new program (e.g., ls, date). The output shows the successful execution of each command within a dedicated child process. The parent waits for each child to complete before forking the next one.

Task 3: Zombie & Orphan Processes

Code Snapshot (Zombie):

```
pid = os.fork()
if pid == 0: # Child
    os._exit(0)
else: # Parent
    print(">>> NOW, open a new terminal and run: ps -el | grep 'Z'")
    time.sleep(30) # Parent does not wait
    os.wait()      # Parent finally reaps the child
```

Results & Analysis (Zombie): A zombie process is created when a child terminates, but its parent has not yet called wait() to read its exit status. The child's process table entry remains, marked as <defunct> or Z, to hold this information. Our simulation confirmed this behavior; by making the parent sleep without waiting, the child became a zombie, as verified by the ps -el command.

Code Snapshot (Orphan):

```
pid = os.fork()
if pid == 0: # Child
    print(f"Child: My parent is {os.getppid()}.")
    time.sleep(5)
    print(f"Child: My new parent is {os.getppid()}.")
    os._exit(0)
else: # Parent
    os._exit(0) # Parent exits immediately
```

Results & Analysis (Orphan): An orphan process is created when its parent terminates before the child does. The operating system prevents the child from being terminated. Instead, the init process (or systemd on modern systems), which has a PID of 1, "adopts" the orphan. Our output clearly shows the child's parent PID changing from the original parent's PID to 1.

Task 4: Inspecting Process Info from /proc
Code Snapshot:

```python
def task4_inspect_proc(pid):
    proc_dir = f"/proc/{pid}"
    # Read from /proc/[pid]/status
    with open(f"{proc_dir}/status", 'r') as f:
        # ... parsing logic ...

    # Read /proc/[pid]/exe symbolic link
    exe_path = os.readlink(f"{proc_dir}/exe")

    # List files in /proc/[pid]/fd
    fds = os.listdir(f"{proc_dir}/fd")
```

Results & Analysis: The /proc filesystem is a virtual filesystem that provides a window into the kernel's data structures. By reading files like /proc/[pid]/status, we can retrieve real-time information about a process, including its name, state, and memory usage. The experiment successfully demonstrated reading this file, resolving the /proc/[pid]/exe symlink to find the executable's path, and listing open file descriptors in /proc/[pid]/fd. This is the mechanism used by tools like ps and top.

Task 5: Process Prioritization
Code Snapshot:

```python
def task5_process_prioritization():
    nice_values = [19, 10, 0] # 19=low priority, 0=normal
    for nice_val in nice_values:
        pid = os.fork()
        if pid == 0: # Child
            os.nice(nice_val) # Set process priority
            # ... run CPU-bound task ...
            print(f"--> Child (PID {os.getpid()}, Nice {os.nice(0)}): FINISHED ...'
            os._exit(0)
    # Parent waits for all children
    for _ in nice_values:
        os.wait()
```

Results & Analysis: The nice() system call adjusts the priority of a process for the scheduler. A higher nice value (up to 19) means lower priority, while a lower value (down to -20) means higher priority. In our experiment, we created three CPU-intensive children with nice values of 0, 10, and 19. The output consistently showed that the process with the lowest nice value (0) finished its task first, followed by the one with the value 10, and finally the one with the lowest priority (nice value 19). This directly demonstrates the scheduler favoring higher-priority processes.