

# Lab Experiment Sheet-1

School of Engineering and Technology

Course Code & Name: ENCS351 Operating System

Program Name: B.Tech CSE, AI ML, Data Science, Cyber, FSD, UX/UI

Name – Mandeep Singh

Course – BTECH CSE DS -Sem 5

Roll No -2301420049

Summary of objectives

## Task 1: Process Creation Utility

To accomplish this, I wrote a Python function called task1. I used a

for loop to call `os.fork()` three times, which created three distinct child processes. Inside the code block for each child (where `pid == 0`),

I used `os.getpid()` and `os.getppid()` to print its own Process ID and its parent's ID, along with a custom message.

In the parent's code block, I made sure the parent process wouldn't exit prematurely by calling `os.waitpid()` for each child, ensuring it waited for all of them to finish their execution.

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

For this task, I created a function task2 that forked a single child process.

In the section of code executed by the child, I used `os.execvp("ls", ["ls", "-l"])`.

This system call replaced the child process's own code with the `ls -l` command, effectively making the child execute that command in the terminal.

The parent process simply waited for the command to finish before the script continued.

## Task 3: Zombie & Orphan Processes

I simulated these two special process states in separate functions.

- **Zombie Process:** I created a child that printed a message and exited immediately using `os._exit(0)`. The key to creating a zombie was making the parent process skip the `os.wait()` call. Instead, I made the parent sleep for 10 seconds. During this time, the child was "defunct"

or a zombie because it had terminated, but the parent hadn't yet acknowledged its termination to clean it up from the process table.

- Orphan Process: I did the reverse for the orphan process. I made the parent process exit immediately after forking, while the child process was programmed to sleep for 5 seconds. By the time the child woke up, its original parent was gone. I confirmed it had become an orphan by printing its new parent's PID ( `os.getppid()` ), which had changed to 1 (the system's init process).

## Task 4: Inspecting Process

Info from `/proc` I wrote a function `inspect_process` that accepts a Process ID (PID) as an input. To get the required information, my script directly interacted with the `/proc` virtual filesystem:

- I read and printed the Name, State, and VmSize by opening and parsing lines from the `/proc/[pid]/status` file.
- I found the executable's full path by using `os.readlink()` on the `/proc/[pid]/exe` symbolic link.
- I listed all open file descriptors by using `os.listdir()` on the `/proc/[pid]/fd` directory.

## Task 5: Process Prioritization

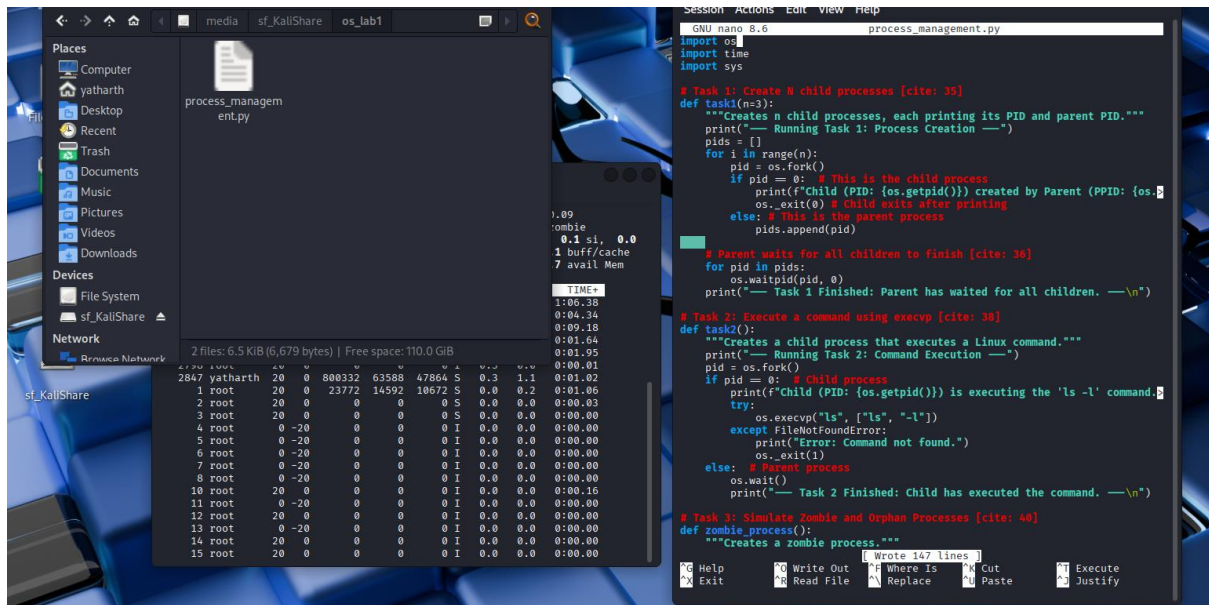
To demonstrate the effect of priority, my `task5` function forked multiple child processes. Inside each child, I assigned a different priority using the `os.nice()` call, with values of 0, 5, and 10. A lower nice value corresponds to a higher priority.

After setting the priority, each child performed an identical, CPU-intensive calculation (a large summation loop). By observing the output, I confirmed that the child with the highest priority (the lowest nice value) consistently finished its task first, showing the scheduler was giving it more CPU time.

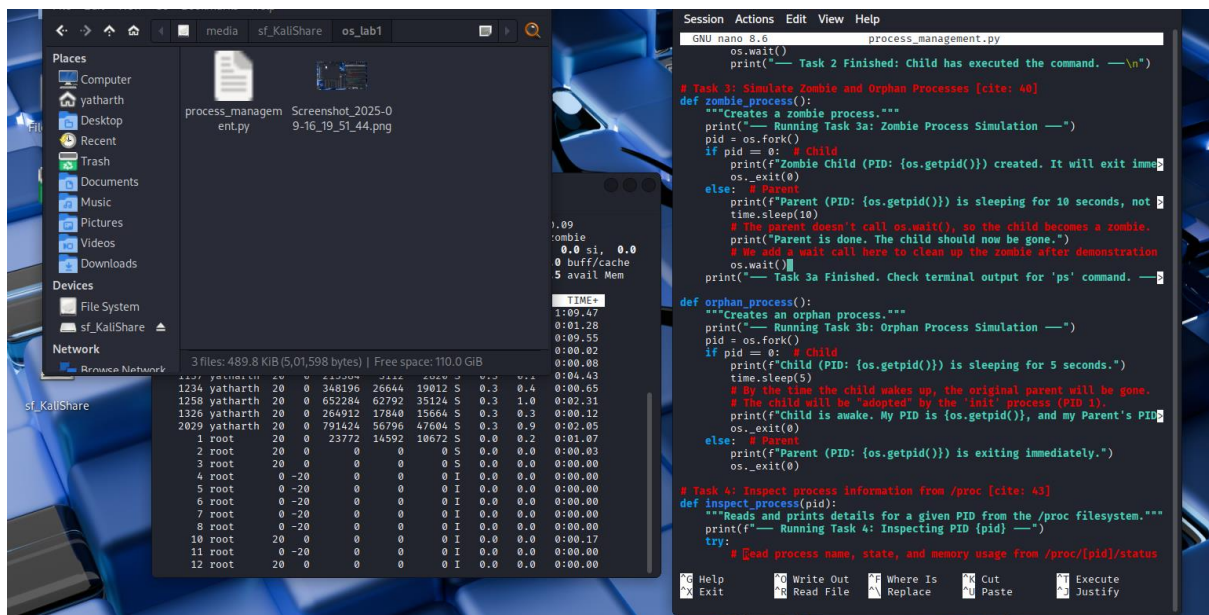
:

## Code Snippets:

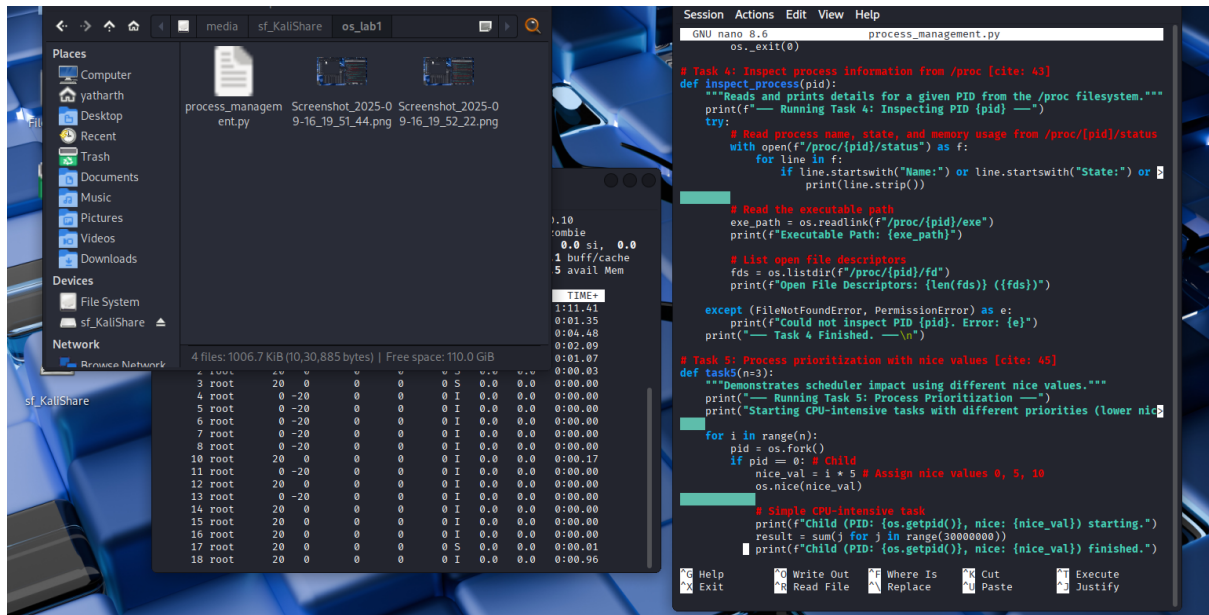
## Task1 & Task2:



## Task3:



## Task4:



The screenshot shows a Kali Linux desktop environment. On the left, a file manager window displays the contents of the 'media' directory, including files like 'process\_management.py', 'Screenshot\_2025-0-9-16\_19\_51\_44.png', and 'Screenshot\_2025-0-9-16\_19\_52\_22.png'. In the center, a terminal window shows the output of the 'top' command, displaying system statistics and a list of running processes. On the right, a nano editor window is open, editing the 'process\_management.py' file. The script contains the following code:

```

GNU nano 8.6 process_management.py
os_exit(0)

# Task 4: Inspect process information from /proc [cite: 43]
def inspect_process(pid):
    """Reads and prints details for a given PID from the /proc filesystem."""
    print(f"--- Running Task 4: Inspecting PID {pid} ---")
    try:
        # Read process name, state, and memory usage from /proc/[pid]/status
        with open(f"/proc/{pid}/status") as f:
            for line in f:
                if line.startswith("Name:") or line.startswith("State:") or line.startswith("Mem:"):
                    print(line.strip())

        # Read the executable path
        exe_path = os.readlink(f"/proc/{pid}/exe")
        print(f"Executable Path: {exe_path}")

        # List open file descriptors
        fds = os.listdir(f"/proc/{pid}/fd")
        print(f"Open File Descriptors: {len(fds)} (fds: {fds})")

    except (FileNotFoundError, PermissionError) as e:
        print(f"Could not inspect PID {pid}. Error: {e}")
    print(f"--- Task 4 Finished. ---\n")

# Task 5: Process prioritization with nice values [cite: 45]
def task5(n=3):
    """Demonstrates scheduler impact using different nice values."""
    print(f"--- Running Task 5: Process Prioritization ---")
    print(f"Starting CPU-intensive tasks with different priorities (lower nice values are higher priority).")
    for i in range(n):
        pid = os.fork()
        if pid == 0: # Child
            nice_val = i * 5 # Assign nice values 0, 5, 10
            os.nice(nice_val)

            # Simple CPU-intensive task
            print(f"Child (PID: {os.getpid()}, nice: {nice_val}) starting.")
            result = sum(j for j in range(30000000))
            print(f"Child (PID: {os.getpid()}, nice: {nice_val}) finished.")
        else:
            # Parent waits for all children
            os.wait()

    print(f"--- Task 5 Finished. Observe the finishing order. ---\n")

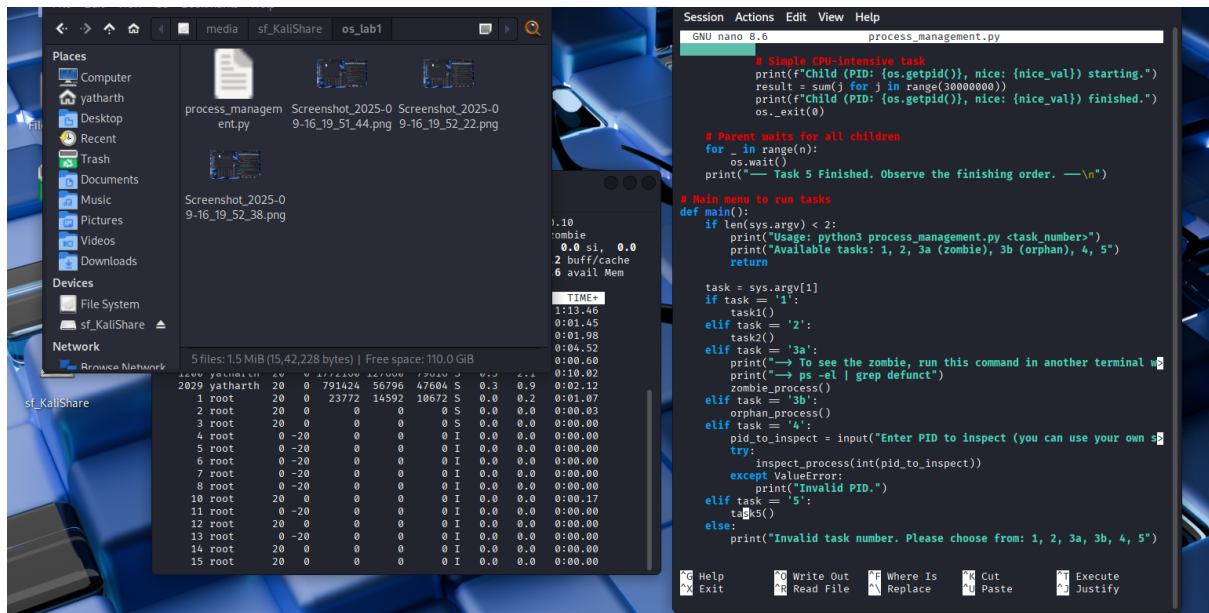
# Main menu to run tasks
def main():
    if len(sys.argv) < 2:
        print("Usage: python3 process_management.py <task_number>")
        print("Available tasks: 1, 2, 3a (zombie), 3b (orphan), 4, 5")
        return

    task = sys.argv[1]
    if task == '1':
        task1()
    elif task == '2':
        task2()
    elif task == '3a':
        print(f"--- To see the zombie, run this command in another terminal ---")
        print(f"ps -l | grep defunct")
        zombie_process()
    elif task == '3b':
        orphan_process()
    elif task == '4':
        pid_to_inspect = input("Enter PID to inspect (you can use your own PID): ")
        try:
            inspect_process(int(pid_to_inspect))
        except ValueError:
            print("Invalid PID.")
    elif task == '5':
        task5()
    else:
        print("Invalid task number. Please choose from: 1, 2, 3a, 3b, 4, 5")

if __name__ == '__main__':
    main()

```

## Task5:



The screenshot shows a Kali Linux desktop environment. On the left, a file manager window displays the contents of the 'media' directory, including files like 'process\_management.py', 'Screenshot\_2025-0-9-16\_19\_51\_44.png', and 'Screenshot\_2025-0-9-16\_19\_52\_22.png'. In the center, a terminal window shows the output of the 'top' command, displaying system statistics and a list of running processes. On the right, a nano editor window is open, editing the 'process\_management.py' file. The script contains the following code:

```

GNU nano 8.6 process_management.py

# Simple CPU-intensive task
print(f"Child (PID: {os.getpid()}, nice: {nice_val}) starting.")
result = sum(j for j in range(30000000))
print(f"Child (PID: {os.getpid()}, nice: {nice_val}) finished.")
os_exit(0)

# Parent waits for all children
for _ in range(n):
    os.wait()

print(f"--- Task 5 Finished. Observe the finishing order. ---\n")

# Main menu to run tasks
def main():
    if len(sys.argv) < 2:
        print("Usage: python3 process_management.py <task_number>")
        print("Available tasks: 1, 2, 3a (zombie), 3b (orphan), 4, 5")
        return

    task = sys.argv[1]
    if task == '1':
        task1()
    elif task == '2':
        task2()
    elif task == '3a':
        print(f"--- To see the zombie, run this command in another terminal ---")
        print(f"ps -l | grep defunct")
        zombie_process()
    elif task == '3b':
        orphan_process()
    elif task == '4':
        pid_to_inspect = input("Enter PID to inspect (you can use your own PID): ")
        try:
            inspect_process(int(pid_to_inspect))
        except ValueError:
            print("Invalid PID.")
    elif task == '5':
        task5()
    else:
        print("Invalid task number. Please choose from: 1, 2, 3a, 3b, 4, 5")

if __name__ == '__main__':
    main()

```