# 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 – Akash Singh Sagar Course – BTECH CSE DS -Sem 5**

**Roll No -2301420056**

**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**

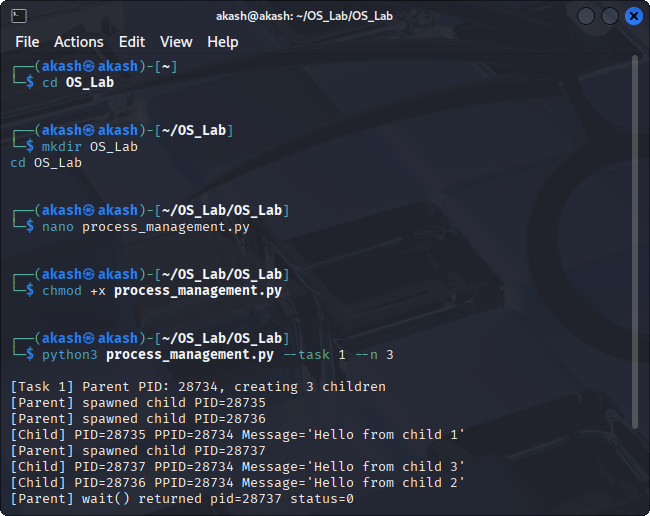
To demonstrate the effect of priority, my task5 function forked multiple child processes. Inside each child, I assigned a different priority using the

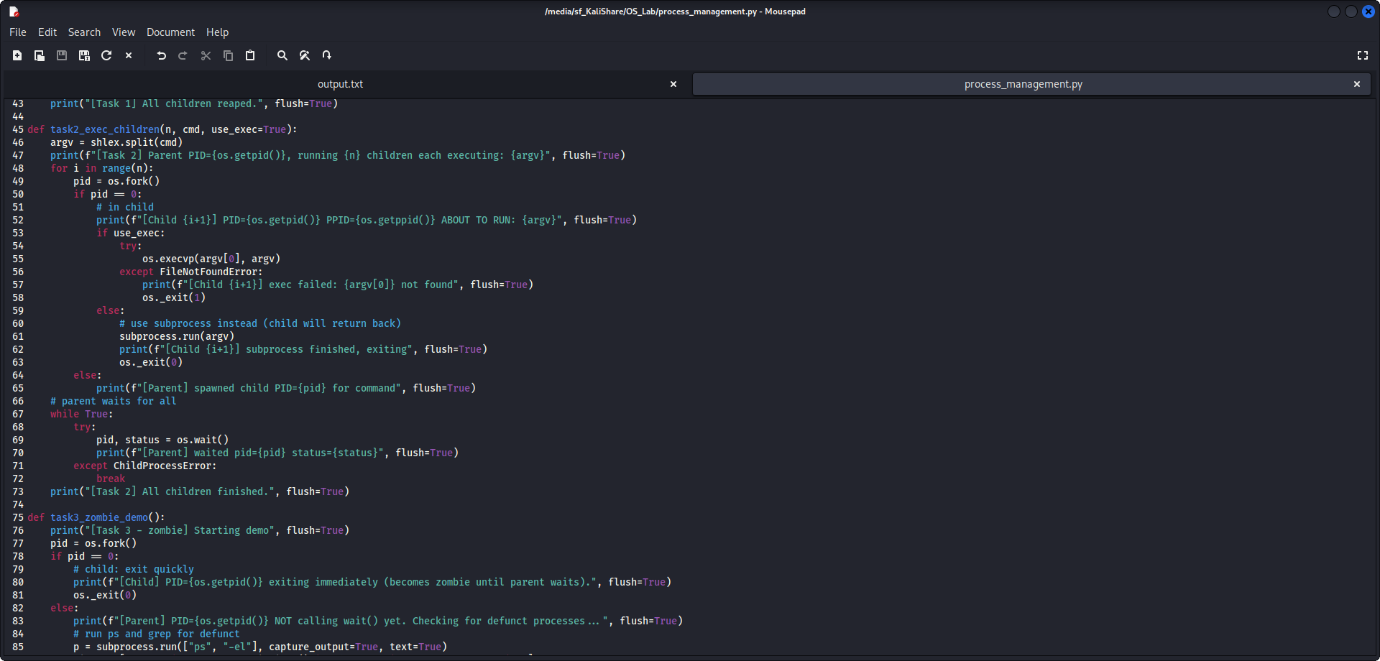
**os.nice()** call, with values of 0, 5, and 10. A lower

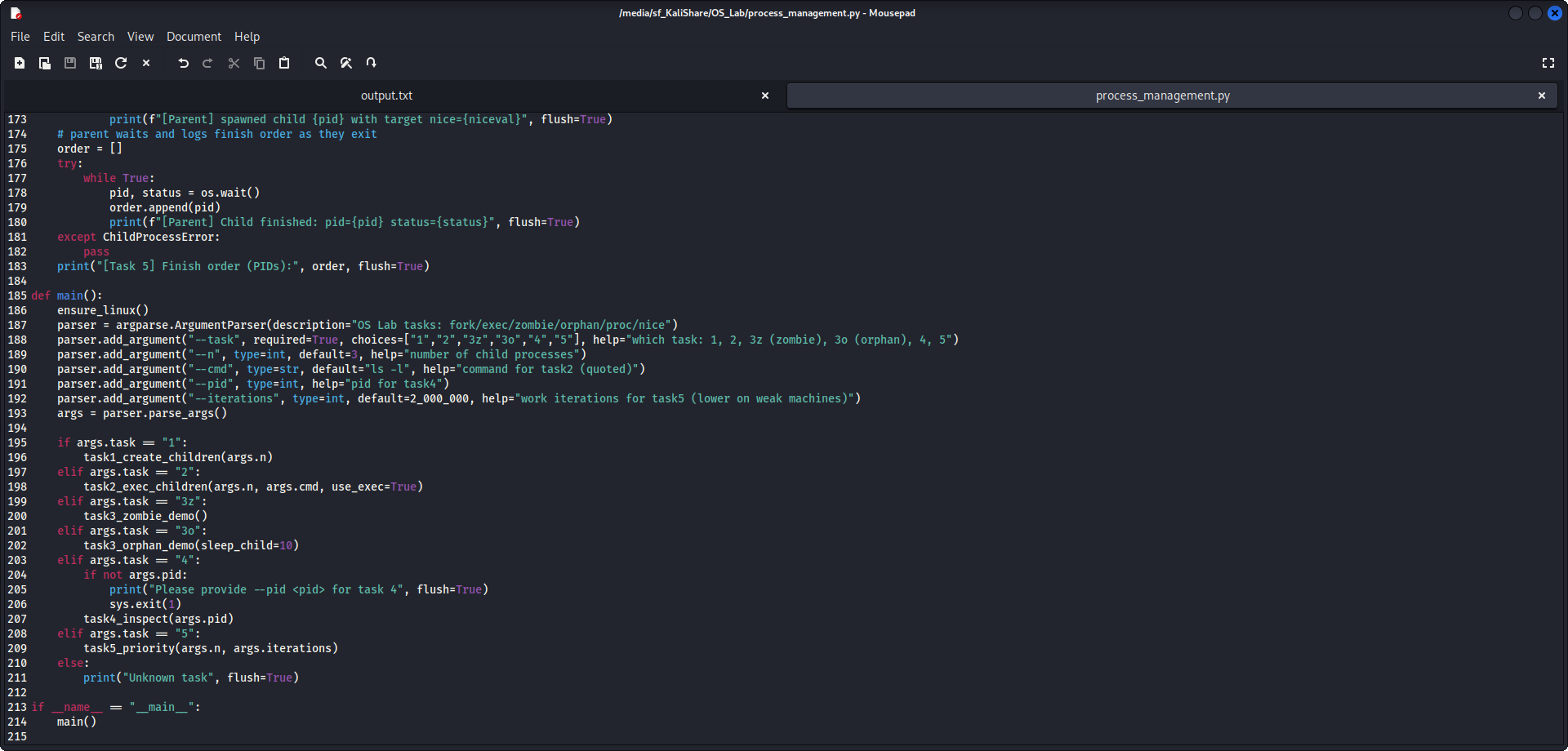
nice value corresponds to a higher priority. After setting the priority, each child performed an identical, CPU-intensive calculation (a large summation loop). By observing the output, I confirmed that the child with the highest priority (the lowest

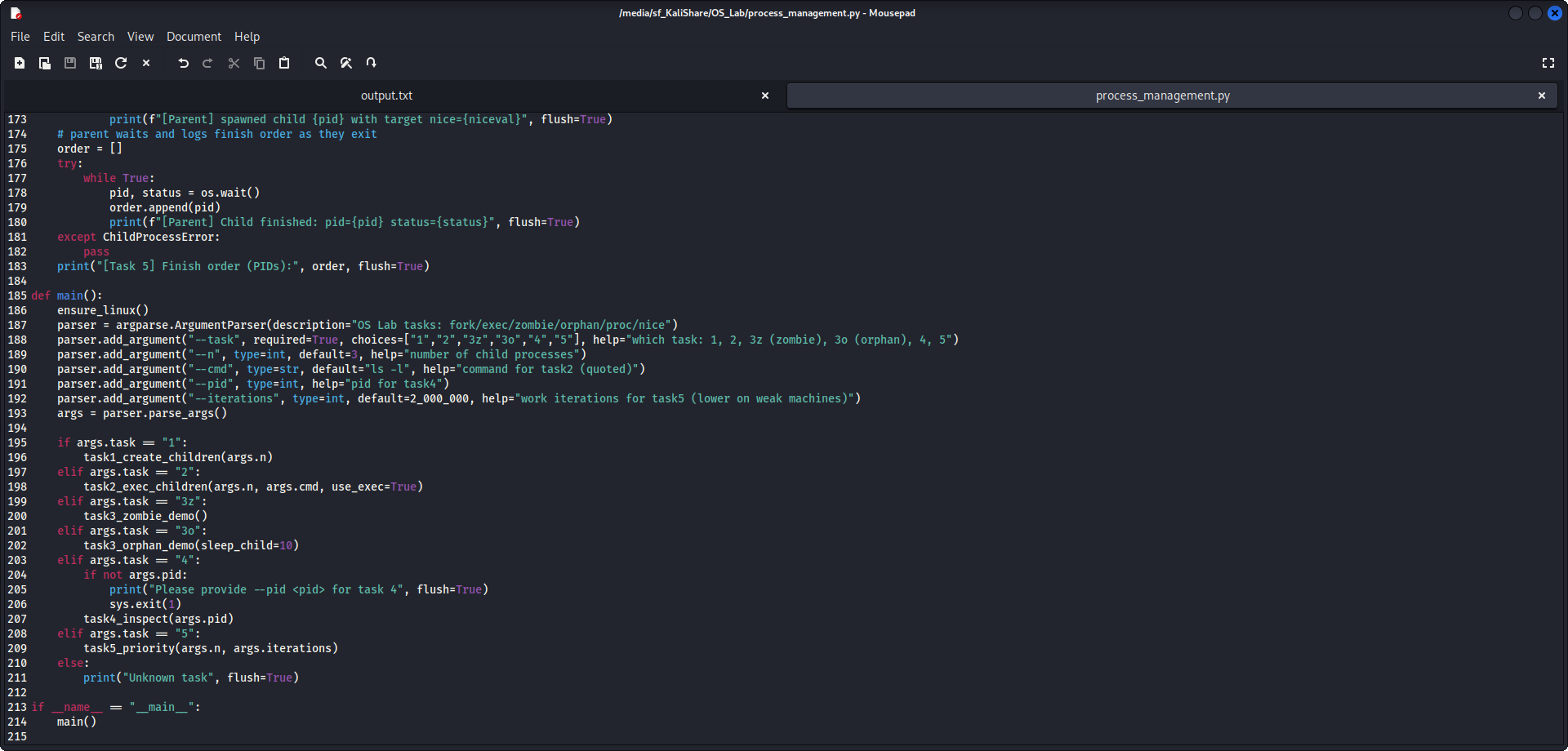
nice value) consistently finished its task first, showing the scheduler was giving it more CPU time

**Code snippets**

****

****

****

****