

Name – Nidhi Kumari

Roll No – 2401011482

Lab Experiment Sheet-1

Task 1: Process Creation Utility

Write a Python program that creates N child processes using os.fork(). Each child prints:

- Its PID
- Its Parent PID
- A custom message

The parent should wait for all children using os.wait().

CODE (PYTHON):

```
import os

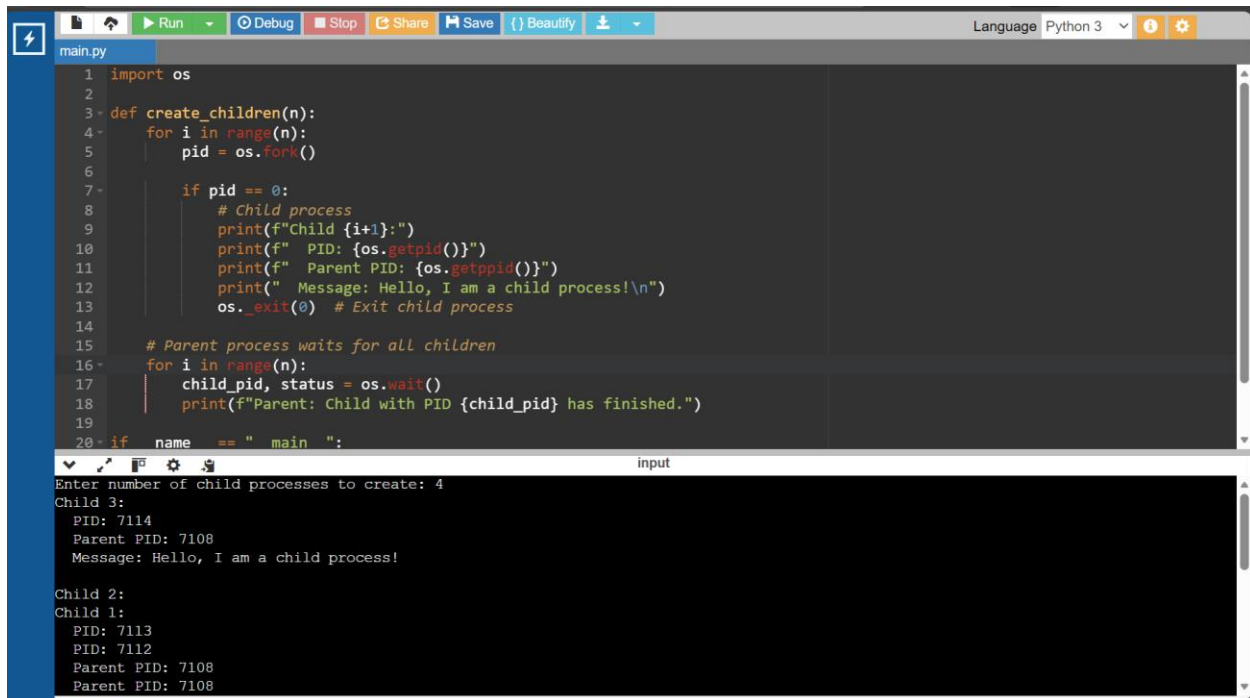
def create_children(n):
    for i in range(n):
        pid = os.fork()

        if pid == 0
            print(f"Child {i+1}:")
            print(f" PID: {os.getpid()}")
            print(f" Parent PID: {os.getppid()}")
            print(f" Message: Hello, I am a child process!\n")
            os._exit(0) # Exit child process
```

```
for i in range(n):  
  
    child_pid, status = os.wait()  
  
    print(f"Parent: Child with PID {child_pid} has finished.")
```

```
if __name__ == "__main__":  
  
    N = int(input("Enter number of child processes to create: "))  
  
    create_children(N)
```

OUTPUT:



The screenshot shows a code editor with a Python script named `main.py`. The script defines a function `create_children(n)` that forks `n` child processes. Each child process prints its own PID, the parent's PID, and a message. The parent process then waits for all children to finish and prints a confirmation message for each. The output window shows the execution results for `n=4`, displaying the PIDs and messages for four child processes and the corresponding parent confirmation messages.

```
1 import os  
2  
3 def create_children(n):  
4     for i in range(n):  
5         pid = os.fork()  
6  
7         if pid == 0:  
8             # Child process  
9             print(f"Child {i+1}:")  
10            print(f"   PID: {os.getpid()}")  
11            print(f"   Parent PID: {os.getppid()}")  
12            print("   Message: Hello, I am a child process!\n")  
13            os._exit(0) # Exit child process  
14  
15            # Parent process waits for all children  
16            for i in range(n):  
17                child_pid, status = os.wait()  
18                print(f"Parent: Child with PID {child_pid} has finished.")  
19  
20 if __name__ == "__main__":  
    N = int(input("Enter number of child processes to create: "))  
    create_children(N)
```

input
Enter number of child processes to create: 4
Child 3:
PID: 7114
Parent PID: 7108
Message: Hello, I am a child process!

Child 2:
Child 1:
PID: 7113
PID: 7112
Parent PID: 7108
Parent PID: 7108
Parent: Child with PID 7114 has finished.
Parent: Child with PID 7113 has finished.
Parent: Child with PID 7112 has finished.
Parent: Child with PID 7111 has finished.

Task 2: Command Execution Using exec()

Modify Task 1 so that each child process executes a Linux command (ls, date, ps, etc.)

using

`os.execvp()`

or

`subprocess.run()`.

CODE(PYTHON):

```
import os
```

```
def create_children_with_exec(n):
```

```
    commands = [
```

```
        ["ls"],
```

```
        ["date"],
```

```
        ["ps"]
```

```
    ]
```

```
    for i in range(n):
```

```
        pid = os.fork()
```

```
        if pid == 0:
```

```
            # Child process
```

```
            print(f"\nChild {i+1}:")
```

```
            print(f"  PID: {os.getpid()}")
```

```
            print(f"  Parent PID: {os.getppid()}")
```

```
            print("  Executing command...\n")
```

```
cmd = commands[i % len(commands)]
```

```
os.execvp(cmd[0], cmd)
```

```
print("exec failed!")
```

```
os._exit(1)
```

```
for i in range(n):
```

```
    child_pid, status = os.wait()
```

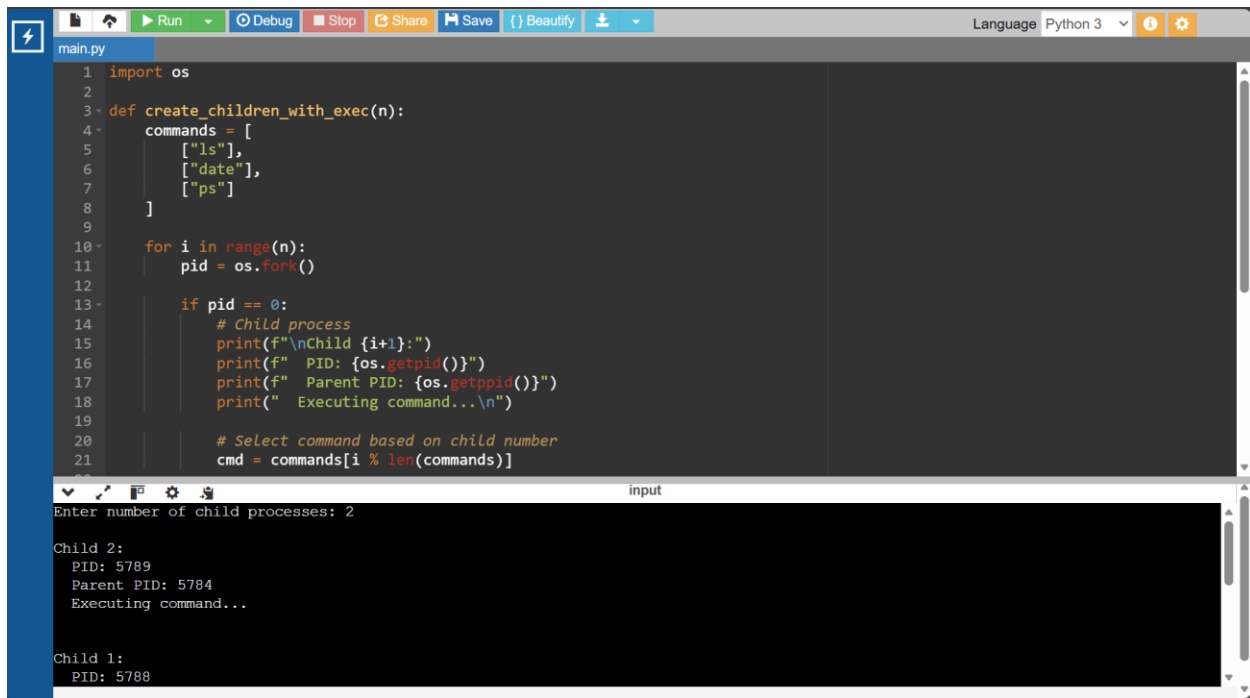
```
    print(f"\nParent: Child with PID {child_pid} finished.")
```

```
if __name__ == "__main__":
```

```
    N = int(input("Enter number of child processes: "))
```

```
    create_children_with_exec(N)
```

OUTPUT:



The screenshot shows a code editor with a Python script named `main.py`. The code defines a function `create_children_with_exec(n)` that forks `n` child processes. Each child process prints its own PID, its parent's PID, and the command it is executing. The parent process then waits for each child to finish and prints a confirmation message. The main block of the script prompts the user for the number of child processes and calls the function.

```
1 import os
2
3 def create_children_with_exec(n):
4     commands = [
5         ["ls"],
6         ["date"],
7         ["ps"]
8     ]
9
10    for i in range(n):
11        pid = os.fork()
12
13        if pid == 0:
14            # Child process
15            print(f"\nChild {i+1}:")
16            print(f"    PID: {os.getpid()}")
17            print(f"    Parent PID: {os.getppid()}")
18            print("    Executing command...\n")
19
20            # Select command based on child number
21            cmd = commands[i % len(commands)]
```

The output of the program is shown in the terminal window below the code editor. It shows the user entering '2' for the number of child processes. The output then shows the execution of two child processes, each printing their PID, parent PID, and the command they are executing. The parent process then prints a confirmation message for each child.

```
input
Enter number of child processes: 2

Child 2:
PID: 5789
Parent PID: 5784
Executing command...

Child 1:
PID: 5788
```

Task 3: Zombie & Orphan Processes

Zombie: Fork a child and skip wait() in the parent.

Orphan: Parent exits before the child finishes.

Use ps -el | grep defunct to identify zombies.

CODE(PYTHON):

```
import os
```

```
import time
```

```
def zombie_process():
```

```
    pid = os.fork()
```

```
    if pid == 0:
```

```
        print(f"Zombie Child PID {os.getpid()} exiting...")
```

```
        os._exit(0)
```

```
    else:
```

```
        print(f"Zombie Parent PID {os.getpid()} (not waiting).")
```

```
        print(f"Check zombie using: ps -el | grep {pid}")
```

```
        time.sleep(15)
```

```
def orphan_process():
```

```
    pid = os.fork()
```

```
    if pid == 0:
```

```
        print(f"Orphan Child PID {os.getpid()}, Old PPID: {os.getppid()}")
```

```
        time.sleep(5)
```

```
print(f"Orphan Child PID {os.getpid()}, New PPID: {os.getppid()}")
```

```
os._exit(0)
```

```
else:
```

```
print(f"Orphan Parent PID {os.getpid()} exiting.")
```

```
os._exit(0)
```

```
if __name__ == "__main__":
```

```
print("Creating Zombie Process...")
```

```
zombie_process()
```

```
time.sleep(2)
```

```
print("\nCreating Orphan Process...")
```

```
orphan_process()
```

OUTPUT:



The screenshot shows a Python IDE with a file named 'main.py'. The code defines two functions: 'zombie_process()' and 'orphan_process()'. The 'zombie_process()' function forks a child process, prints its PID, and then sleeps for 15 seconds. The 'orphan_process()' function forks a child process, prints its PID and PPID, sleeps for 5 seconds, and then prints its PID and PPID again. The output window shows the execution of the code, displaying the PIDs and PPIDs of the processes.

```
1 import os
2 import time
3
4 def zombie_process():
5     pid = os.fork()
6     if pid == 0:
7         print(f"Zombie Child PID {os.getpid()} exiting...")
8         os._exit(0)
9     else:
10        print(f"Zombie Parent PID {os.getpid()} (not waiting).")
11        print(f"Check zombie using: ps -el | grep {pid}")
12        time.sleep(15)
13
14 def orphan_process():
15     pid = os.fork()
16     if pid == 0:
17         print(f"Orphan Child PID {os.getpid()}, Old PPID: {os.getppid()}")
18         time.sleep(5)
19         print(f"Orphan Child PID {os.getpid()}, New PPID: {os.getppid()}")
20         os._exit(0)
21     else:
```

Creating Zombie Process...
Zombie Parent PID 727 (not waiting).
Check zombie using: ps -el | grep 731
Zombie Child PID 731 exiting...

Task 4: Inspecting Process Info from /proc

Take a PID as input. Read and print:

- Process name, state, memory usage from /proc/[pid]/status
- Executable path from /proc/[pid]/exe
- Open file descriptors from /proc/[pid]/fd

CODE(PYTHON):

```
import os

def read_process_info(pid):
    status_path = f"/proc/{pid}/status"
    exe_path = f"/proc/{pid}/exe"
    fd_path = f"/proc/{pid}/fd"

    with open(status_path, "r") as f:
        lines = f.readlines()

    name = state = memory = None
    for line in lines:
        if line.startswith("Name:"):
            name = line.split(":")[1].strip()
        elif line.startswith("State:"):
            state = line.split(":")[1].strip()
        elif line.startswith("VmSize:"):
```

```
memory = line.split(":")[1].strip()
```

```
print(f"Process Name: {name}")
```

```
print(f"State: {state}")
```

```
print(f"Memory Usage: {memory}")
```

```
try:
```

```
    exe = os.readlink(exe_path)
```

```
    print(f"Executable Path: {exe}")
```

```
except:
```

```
    print("Executable Path: Not accessible")
```

```
print("Open File Descriptors:")
```

```
try:
```

```
    for fd in os.listdir(fd_path):
```

```
        link = os.readlink(os.path.join(fd_path, fd))
```

```
        print(f" FD {fd} -> {link}")
```

```
except:
```

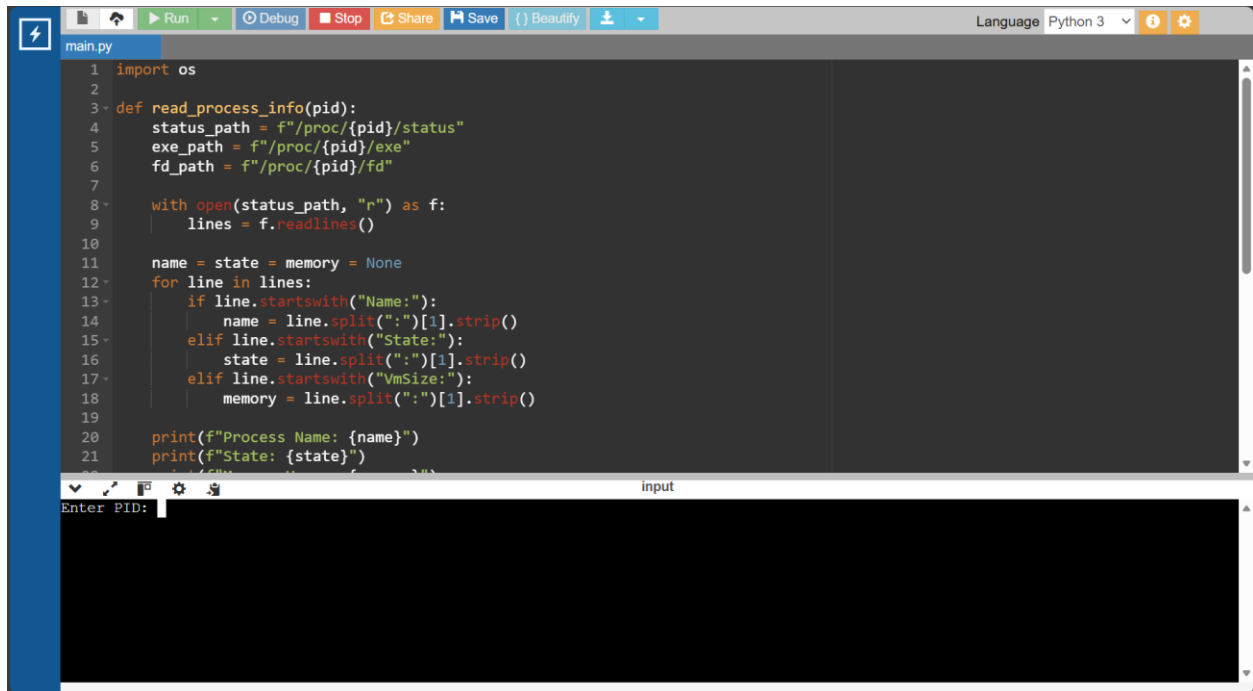
```
    print(" Cannot access file descriptors")
```

```
if __name__ == "__main__":
```

```
    pid = input("Enter PID: ")
```

```
    read_process_info(pid)
```


OUTPUT:



```
1 import os
2
3 def read_process_info(pid):
4     status_path = f"/proc/{pid}/status"
5     exe_path = f"/proc/{pid}/exe"
6     fd_path = f"/proc/{pid}/fd"
7
8     with open(status_path, "r") as f:
9         lines = f.readlines()
10
11     name = state = memory = None
12     for line in lines:
13         if line.startswith("Name:"):
14             name = line.split(":")[1].strip()
15         elif line.startswith("State:"):
16             state = line.split(":")[1].strip()
17         elif line.startswith("VmSize:"):
18             memory = line.split(":")[1].strip()
19
20     print(f"Process Name: {name}")
21     print(f"State: {state}")
22     print(f"Memory: {memory}")
23
24 if __name__ == "__main__":
25     pid = input("Enter PID: ")
26     read_process_info(int(pid))
```

Task 5: Process Prioritization

Create multiple CPU-intensive child processes. Assign different `nice()` values. Observe and log execution order to show scheduler impact.

CODE(PYTHON):

```
import os
```

```
import time
```

```
def cpu_task(label):
```

```
    s = 0
```

```
    for i in range(50_000_000):
```

```
        s += i
```

```
print(f"{label} finished. PID={os.getpid()}")
```

```
def create_process(priority, label):
```

```
    pid = os.fork()
```

```
    if pid == 0:
```

```
        os.nice(priority)
```

```
        start = time.time()
```

```
        cpu_task(label)
```

```
        end = time.time()
```

```
        print(f"{label} Time: {end - start:.2f}s Priority: {priority}")
```

```
        os._exit(0)
```

```
if __name__ == "__main__":
```

```
    print("Starting processes with different nice values...")
```

```
    create_process(0, "Normal Priority")
```

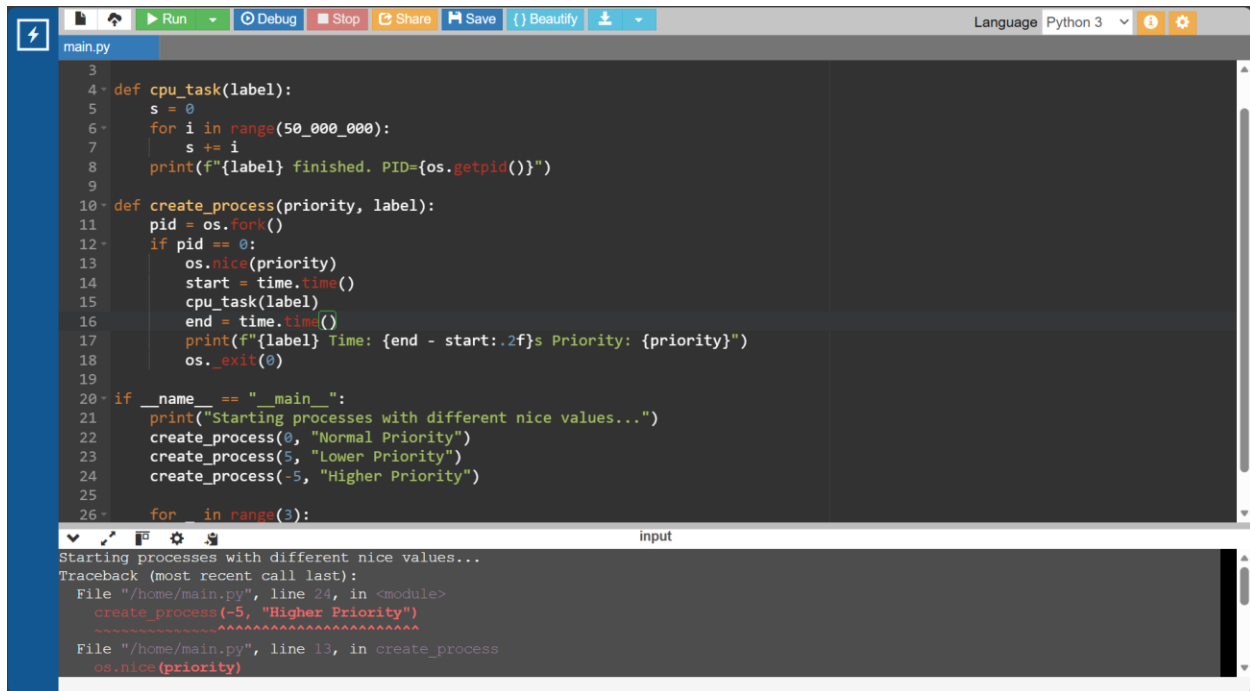
```
    create_process(5, "Lower Priority")
```

```
    create_process(-5, "Higher Priority")
```

```
for _ in range(3):
```

```
    os.wait()
```

OUTPUT:



The screenshot shows a code editor with a dark theme. The top toolbar includes buttons for Run, Debug, Stop, Share, Save, and Beautify. The language is set to Python 3. The code in the editor is as follows:

```
3
4 def cpu_task(label):
5     s = 0
6     for i in range(50_000_000):
7         s += i
8         print(f"{label} finished. PID={os.getpid()}")
9
10 def create_process(priority, label):
11     pid = os.fork()
12     if pid == 0:
13         os.nice(priority)
14         start = time.time()
15         cpu_task(label)
16         end = time.time()
17         print(f"{label} Time: {end - start:.2f}s Priority: {priority}")
18         os._exit(0)
19
20 if __name__ == "__main__":
21     print("Starting processes with different nice values...")
22     create_process(0, "Normal Priority")
23     create_process(5, "Lower Priority")
24     create_process(-5, "Higher Priority")
25
26 for _ in range(3):
```

The output window at the bottom shows the following text:

```
Starting processes with different nice values...
Traceback (most recent call last):
  File "/home/main.py", line 24, in <module>
    create_process(-5, "Higher Priority")
    ~~~~~^~~~~~
  File "/home/main.py", line 13, in create_process
    os.nice(priority)
```