

# **K. R. Mangalam University**

## **School of Engineering and Technology**



### **Operating System Lab File**

**Course Code: ENCS351**

**Submitted by**

**Name:** Aman Singh

**Roll No.:** 2301010352

**Program:** BTech CSE

**Section:** 6/F

**Session:** 2025/26

**Date:** 02/12/2025

**Submitted To**

**Dr. Satinder Pal Singh**

# Index

Serial No.	Assignments	Page No.
1.	Assignment 1	3-16
	1.1 Process Creation Utility.	3
	1.2 Command Execution Using exec ().	5
	1.3 Zombie & Orphan Processes.	11
	1.4 Inspecting Process Info from /proc.	13
	1.5 Task 5: Process Prioritization.	15
2.	Assignment 2	17-19
	2.1 Write a Python script to simulate a basic system startup sequence.	17
	2.2 Use the multiprocessing module to create at least two child processes that perform dummy tasks.	17
	2.3 Implement proper logging to track process start and end times.	17
	2.4 Generate a log file (process_log.txt) to reflect system-like behaviour.	17
	2.5 Submit the Python script and log file along with a short report explaining your implementation.	17
3.	Assignment 3	20-34
	3.1 CPU Scheduling with Gantt Chart.	20
	3.2 Sequential File Allocation.	25
	3.3 Indexed File Allocation.	27
	3.4 Contiguous Memory Allocation.	29
	3.5 MFT & MVT Memory Management.	32
4.	Assignment 4	35-51
	4.1 Batch Processing Simulation (Python).	35
	4.2 System Startup and Logging.	37
	4.3 System Calls and IPC (Python - fork, exec, pipe).	40
	4.4 VM Detection and Shell Interaction.	42
	4.5 CPU Scheduling Algorithms.	46

## 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("  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.")

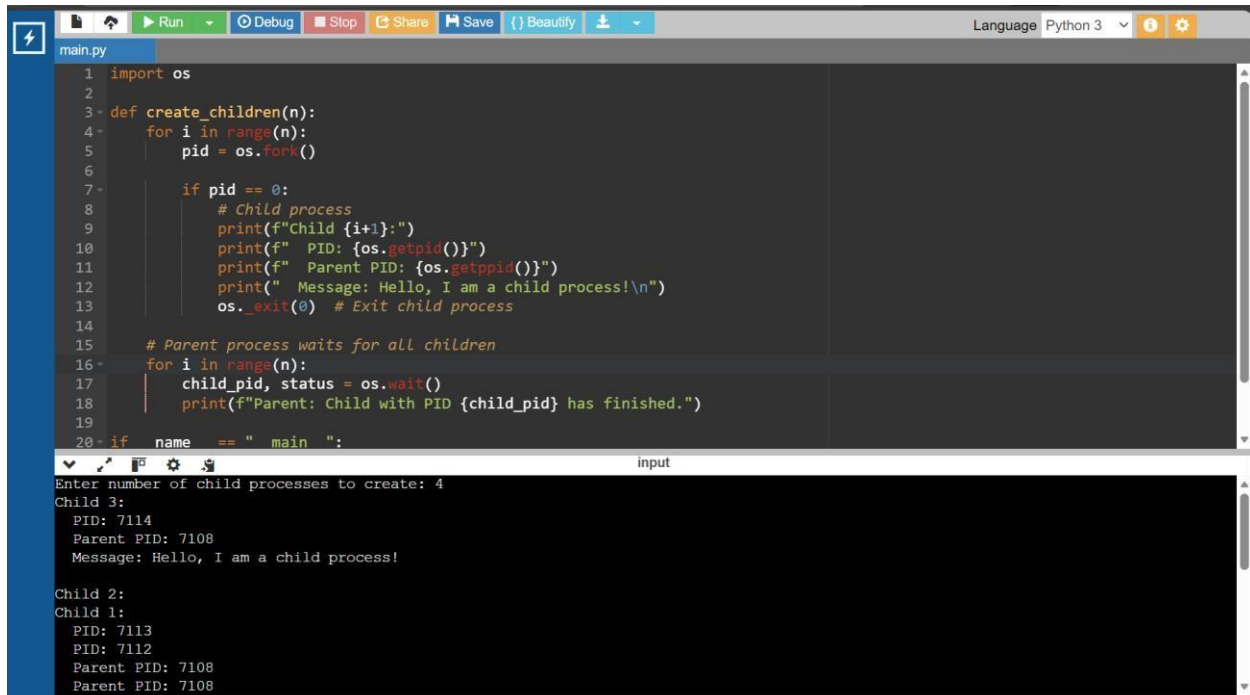
if __name__ == "__main__":

    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 imports the 'os' module and defines a function 'create\_children(n)'. This function uses a loop to create '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 main block of the script prompts the user for the number of child processes to create, which is 4 in this case. The output window shows the execution results, including the PIDs of the children and the parent, and the messages printed by each child process.

```

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__":

```

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

```

## 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()
```

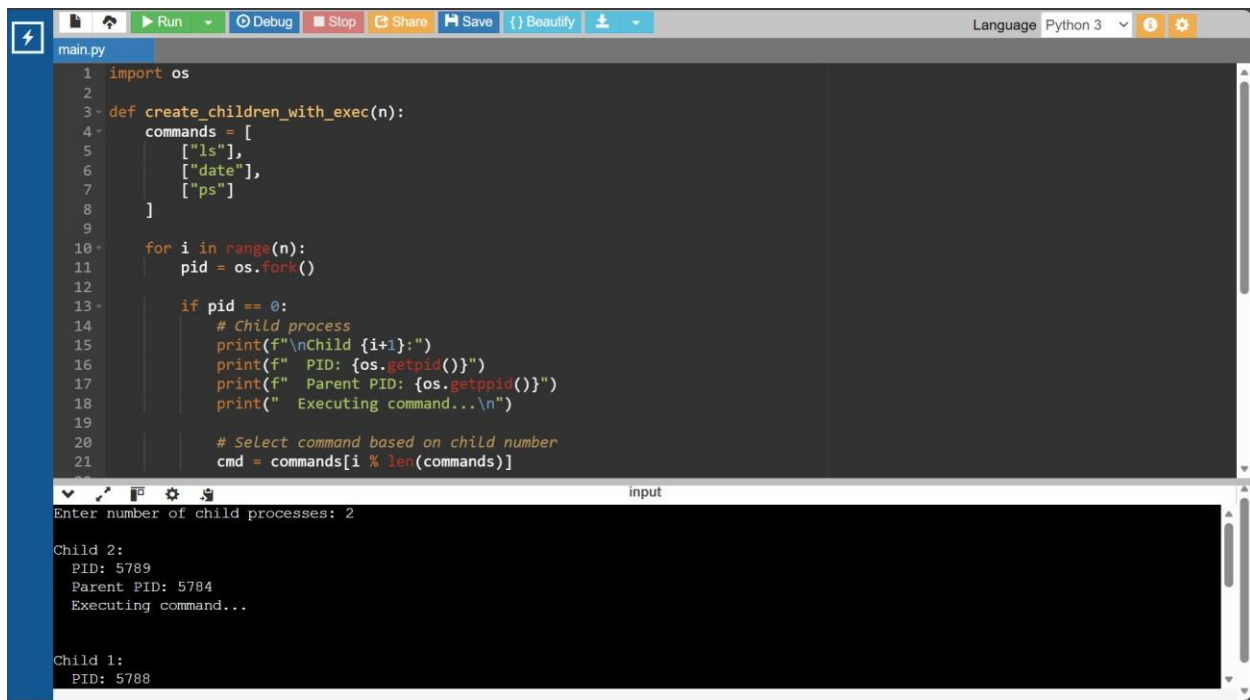
```
    print(f"\nParent: Child with PID {child_pid} finished.")
```

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

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

```
    create_children_with_exec(N)
```

## OUTPUT:



```
main.py
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)]
```

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...")
```

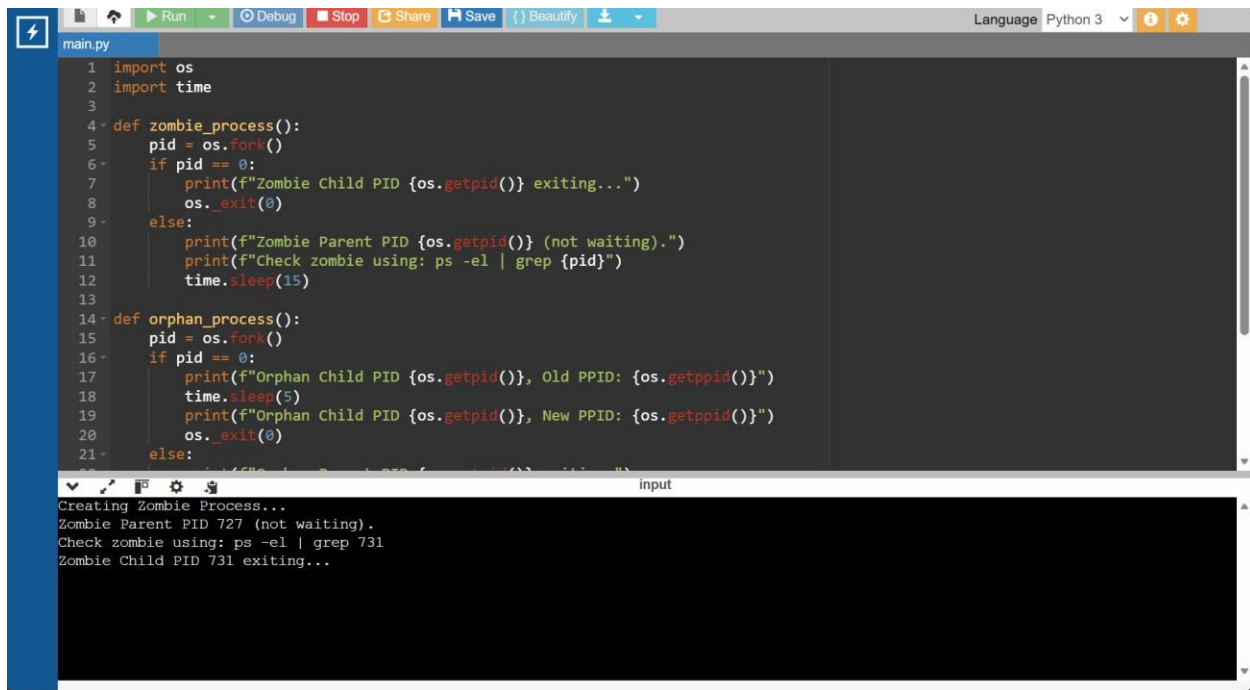
```
zombie_process()
```

```
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()'. 'zombie\_process()' forks a child process; if the child exits first, it prints 'Zombie Child PID {os.getpid()} exiting...' and exits. If the parent exits first, it prints 'Zombie Parent PID {os.getpid()} (not waiting).', checks for the zombie using 'ps -el | grep {pid}', and sleeps for 15 seconds. 'orphan\_process()' forks a child; if the child exits first, it prints 'Orphan Child PID {os.getpid()}, Old PPID: {os.getppid()}', sleeps for 5 seconds, and prints 'Orphan Child PID {os.getpid()}, New PPID: {os.getppid()}' before exiting. If the parent exits first, it prints 'Orphan Parent PID {os.getpid()} exiting.' and exits. The main block prints 'Creating Zombie Process...', calls 'zombie\_process()', sleeps for 2 seconds, prints '\nCreating Orphan Process...', and calls 'orphan\_process()'. The output window shows the execution results: 'Creating Zombie Process...', 'Zombie Parent PID 727 (not waiting).', 'Check zombie using: ps -el | grep 731', and 'Zombie Child PID 731 exiting...'.

```
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:
22        print(f"Orphan Parent PID {os.getpid()} exiting.")
23        os._exit(0)
24
25 if __name__ == "__main__":
26     print("Creating Zombie Process...")
27     zombie_process()
28     time.sleep(2)
29     print("\nCreating Orphan Process...")
30     orphan_process()
```

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:
```

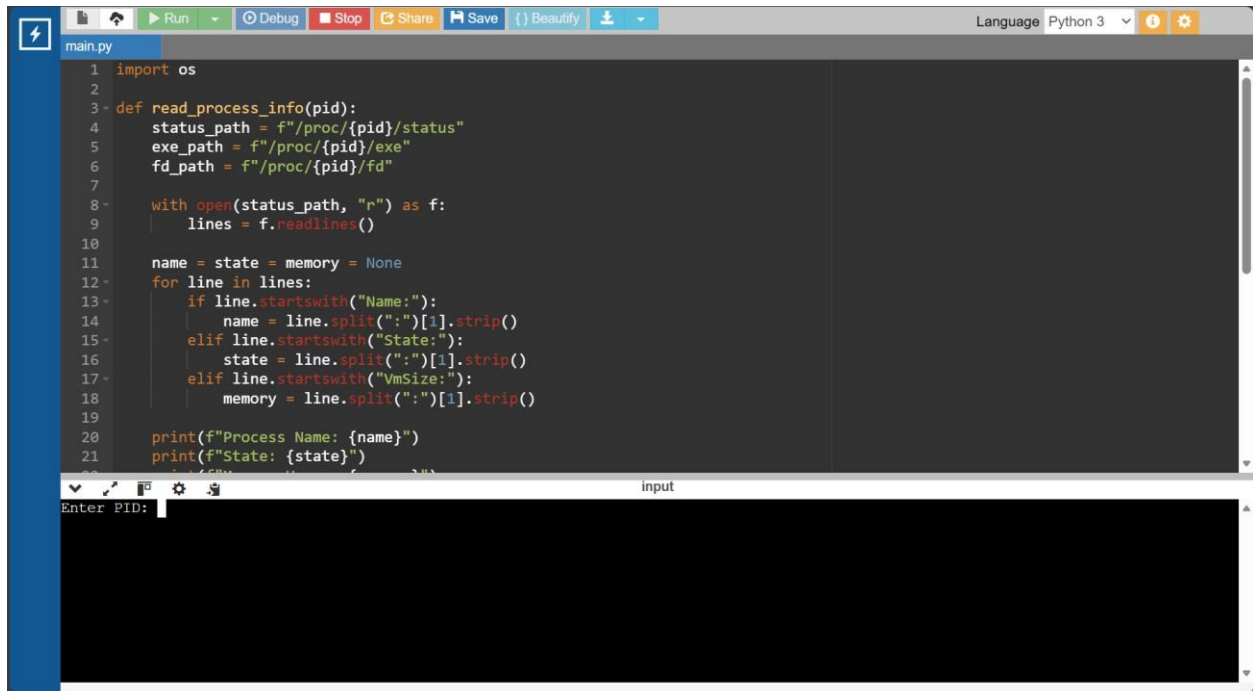
```
    print(" Cannot access file descriptors")
```

```
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")
```

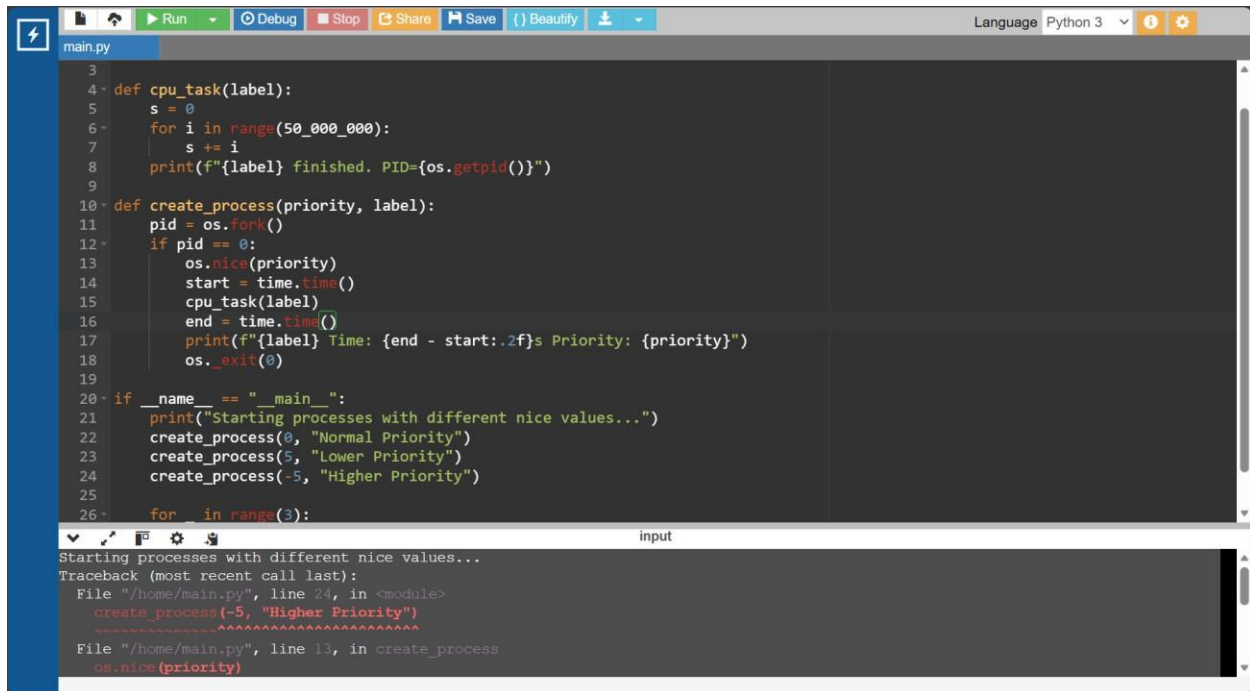
```
    create_process(5, "Lower Priority")
```

```
    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)
```

### **Implementation:**

```
import multiprocessing
```

```
import time
```

```
import logging
```

```
# Setup logger
```

```
logging.basicConfig(
```

```
    filename='process_log.txt',
```

```
    level=logging.INFO,
```

```
    format='%(asctime)s - %(processName)s - %(message)s'
```

```
)
```

```
# Dummy function to simulate a task
```

```
def system_process(task_name):
```

```
    logging.info(f'{task_name} started')
```

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

```
    logging.info(f'{task_name} ended')
```

```
if __name__ == '__main__':
```

```
print("System Starting...")

# Create processes

p1 = multiprocessing.Process(target=system_process, args=('Process-1',))
p2 = multiprocessing.Process(target=system_process, args=('Process-2',))


# Start processes

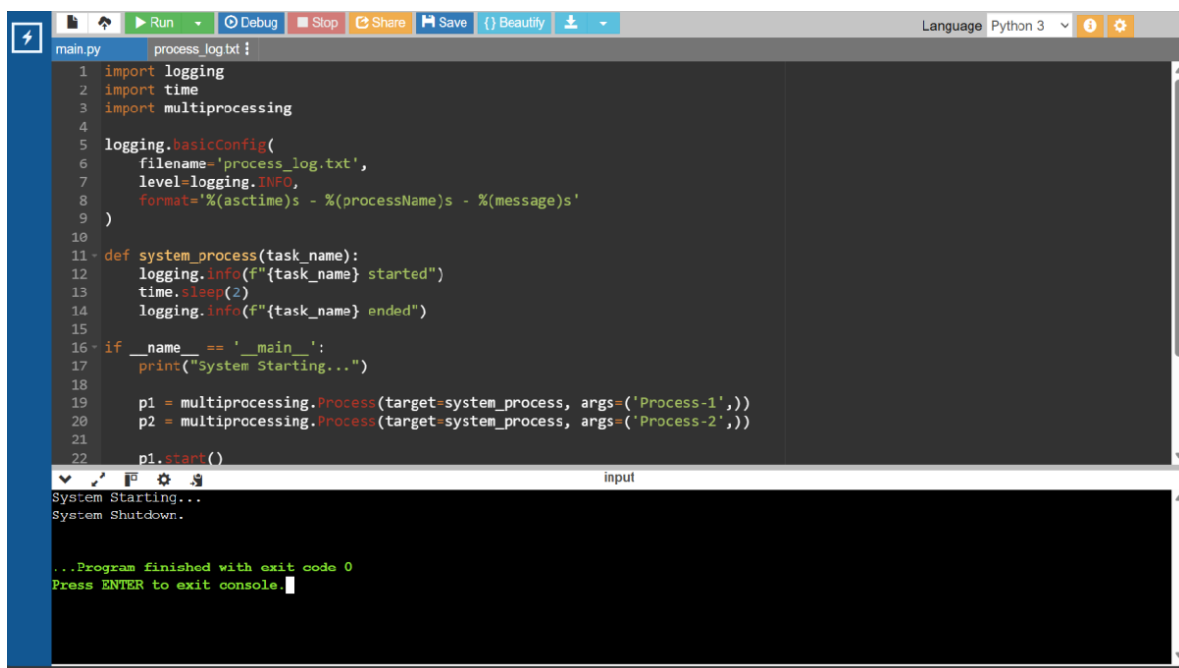
p1.start()
p2.start()


# Wait for processes to complete

p1.join()
p2.join()


print("System Shutdown.")
```

## Output:



The screenshot shows a Python IDE with a dark theme. The top toolbar includes icons for Run, Debug, Stop, Share, Save, and a dropdown menu. The language is set to Python 3. The editor displays a file named 'main.py' with the following code:

```
1 import logging
2 import time
3 import multiprocessing
4
5 logging.basicConfig(
6     filename='process_log.txt',
7     level=logging.INFO,
8     format='%(asctime)s - %(processName)s - %(message)s'
9 )
10
11 def system_process(task_name):
12     logging.info(f"{task_name} started")
13     time.sleep(2)
14     logging.info(f"{task_name} ended")
15
16 if __name__ == '__main__':
17     print("System Starting...")
18
19     p1 = multiprocessing.Process(target=system_process, args=('Process-1',))
20     p2 = multiprocessing.Process(target=system_process, args=('Process-2',))
21
22     p1.start()
```

The output console at the bottom shows the following text:

```
System Starting...
System Shutdown...
...Program finished with exit code 0
Press ENTER to exit console.
```

### Task 1: CPU Scheduling with Gantt Chart

Write a Python program to simulate Priority and Round Robin scheduling algorithms. Compute average waiting and turnaround times.

#### Implementation:

```
# Priority Scheduling Simulation
```

```
processes = []
```

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

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

```
    bt = int(input(f"Enter Burst Time for P{i+1}: "))
```

```
    pr = int(input(f"Enter Priority (lower number = higher priority) for P{i+1}: "))
```

```
    processes.append((i+1, bt, pr))
```

```
processes.sort(key=lambda x: x[2])
```

```
wt = 0
```

```
total_wt = 0
```

```
total_tt = 0
```

```
print("\nPriority Scheduling:")
```

```
print("PID\tBT\tPriority\tWT\tTAT")
```

```
for pid, bt, pr in processes:
```

```
    tat = wt + bt
```

```
    print(f"{pid}\t{bt}\t{pr}\t\t{wt}\t{tat}")
```

```
    total_wt += wt
```

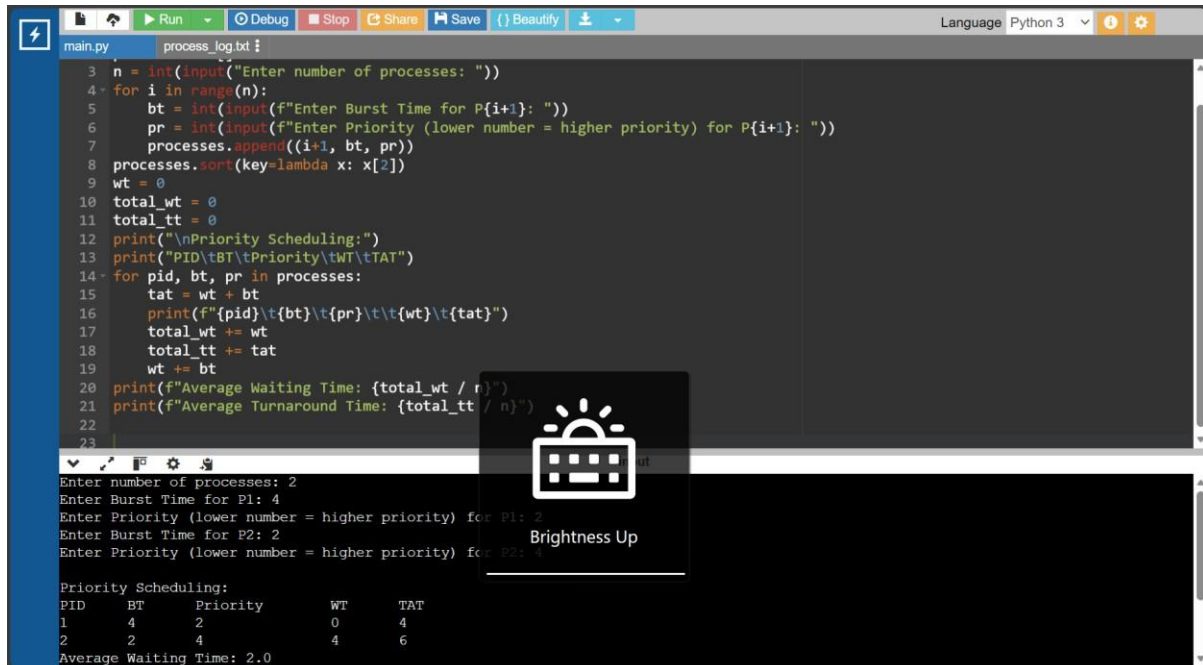
```
    total_tt += tat
```

```
wt += bt
```

```
print(f"Average Waiting Time: {total_wt / n}")
```

```
print(f"Average Turnaround Time: {total_tt / n}")
```

## Output:



```
main.py process_log.txt
3 n = int(input("Enter number of processes: "))
4 for i in range(n):
5     bt = int(input(f"Enter Burst Time for P{i+1}: "))
6     pr = int(input(f"Enter Priority (lower number = higher priority) for P{i+1}: "))
7     processes.append((i+1, bt, pr))
8     processes.sort(key=lambda x: x[2])
9     wt = 0
10    total_wt = 0
11    total_tt = 0
12    print("\nPriority Scheduling:")
13    print("PID\tBT\tPriority\tWT\tTAT")
14    for pid, bt, pr in processes:
15        tat = wt + bt
16        print(f"{pid}\t{bt}\t{pr}\t{wt}\t{tat}")
17        total_wt += wt
18        total_tt += tat
19        wt += bt
20    print(f"Average Waiting Time: {total_wt / n}")
21    print(f"Average Turnaround Time: {total_tt / n}")
22
23
Enter number of processes: 2
Enter Burst Time for P1: 4
Enter Priority (lower number = higher priority) for P1: 2
Enter Burst Time for P2: 2
Enter Priority (lower number = higher priority) for P2: 4

Priority Scheduling:
PID  BT   Priority   WT   TAT
1    4     2         0    4
2    2     4         4    6
Average Waiting Time: 2.0
```

## Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

### Implementation:

```
total_blocks = int(input("Enter total number of blocks: "))
```

```
block_status = [0] * total_blocks
```

```
n = int(input("Enter number of files: "))
```

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

```
    start = int(input(f"Enter starting block for file {i+1}: "))
```

```
    length = int(input(f"Enter length of file {i+1}: "))
```

```
    allocated = True
```

```
    for j in range(start, start+length):
```

```
        if j >= total_blocks or block_status[j] == 1:
```

```
            allocated = False
```

break

if allocated:

for j in range(start, start+length):

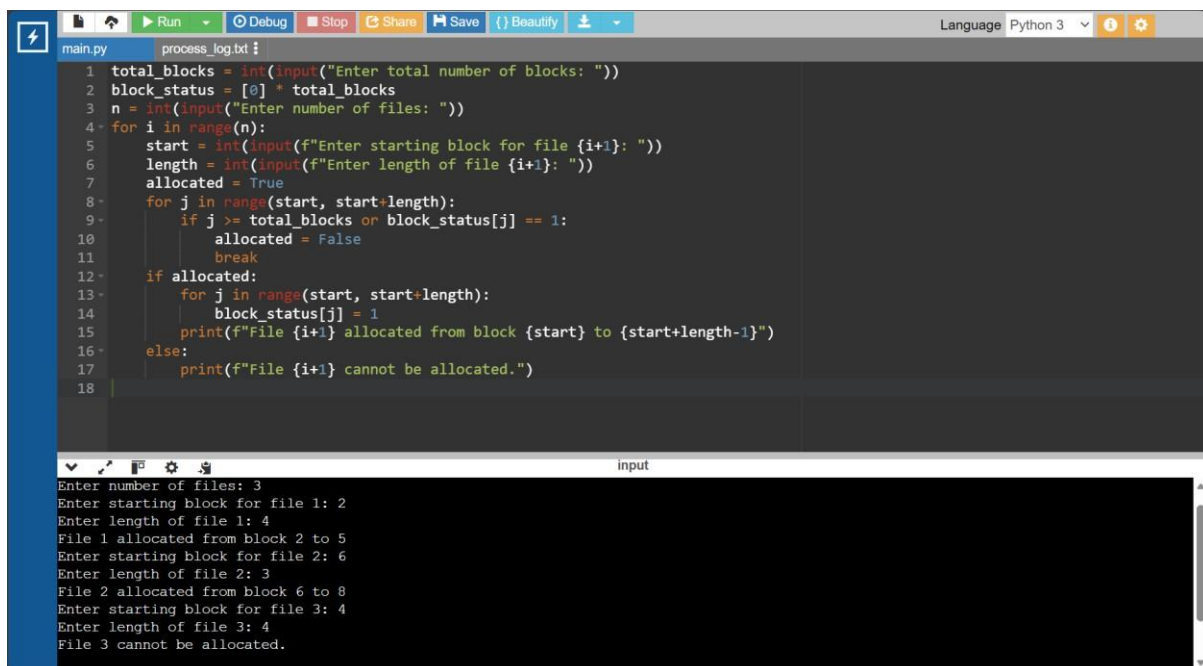
block\_status[j] = 1

print(f"File {i+1} allocated from block {start} to {start+length-1}")

else:

print(f"File {i+1} cannot be allocated.")

## Output:



The screenshot shows a Python IDE with a file named 'main.py' and a 'process\_log.txt' file. The code in 'main.py' implements a file allocation strategy. It prompts the user for the total number of blocks, the number of files, and then for each file, the starting block and length. It checks if the requested blocks are available and updates the 'block\_status' array accordingly. The output window shows the execution results for three files.

```
1 total_blocks = int(input("Enter total number of blocks: "))
2 block_status = [0] * total_blocks
3 n = int(input("Enter number of files: "))
4 for i in range(n):
5     start = int(input(f"Enter starting block for file {i+1}: "))
6     length = int(input(f"Enter length of file {i+1}: "))
7     allocated = True
8     for j in range(start, start+length):
9         if j >= total_blocks or block_status[j] == 1:
10             allocated = False
11             break
12     if allocated:
13         for j in range(start, start+length):
14             block_status[j] = 1
15         print(f"File {i+1} allocated from block {start} to {start+length-1}")
16     else:
17         print(f"File {i+1} cannot be allocated.")
18
```

input

```
Enter number of files: 3
Enter starting block for file 1: 2
Enter length of file 1: 4
File 1 allocated from block 2 to 5
Enter starting block for file 2: 6
Enter length of file 2: 3
File 2 allocated from block 6 to 8
Enter starting block for file 3: 4
Enter length of file 3: 4
File 3 cannot be allocated.
```

## Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

### Implementation:

total\_blocks = int(input("Enter total number of blocks: "))

block\_status = [0] \* total\_blocks

n = int(input("Enter number of files: "))

for i in range(n):

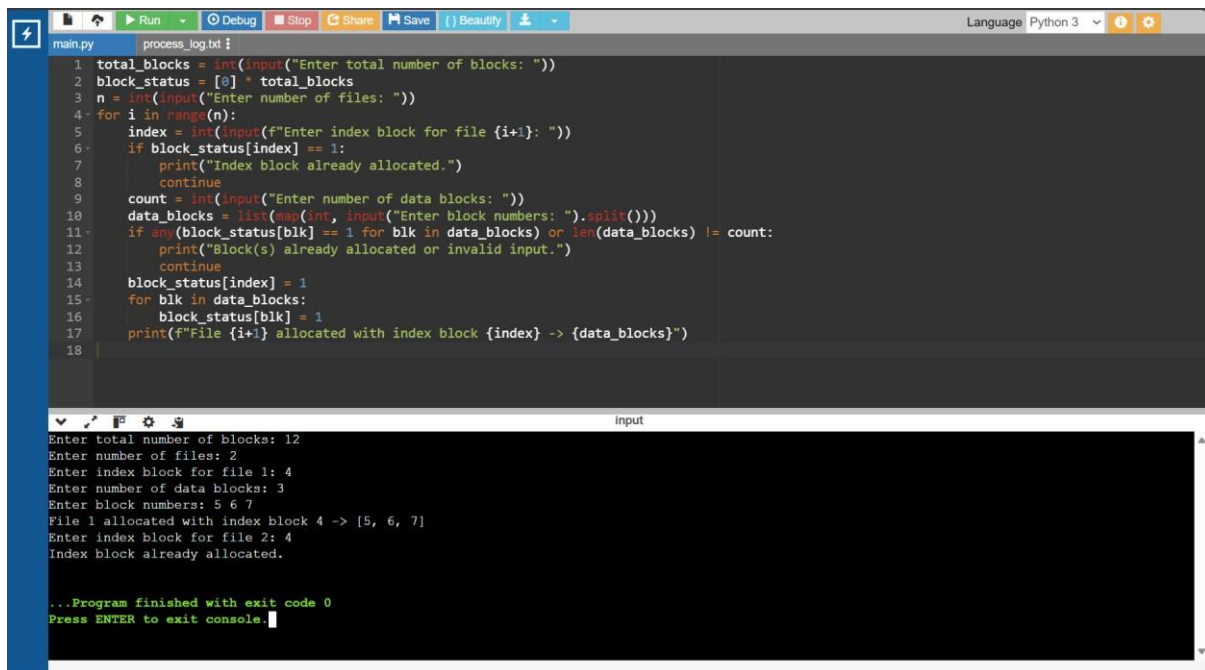
index = int(input(f"Enter index block for file {i+1}: "))

```

if block_status[index] == 1:
    print("Index block already allocated.")
    continue
count = int(input("Enter number of data blocks: "))
data_blocks = list(map(int, input("Enter block numbers: ").split()))
if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
    print("Block(s) already allocated or invalid input.")
    continue
block_status[index] = 1
for blk in data_blocks:
    block_status[blk] = 1
print(f"File {i+1} allocated with index block {index} -> {data_blocks}")

```

## Output:



The screenshot shows a Python IDE with a file named 'main.py' open. The code in the editor is as follows:

```

1 total_blocks = int(input("Enter total number of blocks: "))
2 block_status = [0] * total_blocks
3 n = int(input("Enter number of files: "))
4 for i in range(n):
5     index = int(input(f"Enter index block for file {i+1}: "))
6     if block_status[index] == 1:
7         print("Index block already allocated.")
8         continue
9     count = int(input("Enter number of data blocks: "))
10    data_blocks = list(map(int, input("Enter block numbers: ").split()))
11    if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
12        print("Block(s) already allocated or invalid input.")
13        continue
14    block_status[index] = 1
15    for blk in data_blocks:
16        block_status[blk] = 1
17    print(f"File {i+1} allocated with index block {index} -> {data_blocks}")
18

```

The output window shows the following text:

```

Enter total number of blocks: 12
Enter number of files: 2
Enter index block for file 1: 4
Enter number of data blocks: 3
Enter block numbers: 5 6 7
File 1 allocated with index block 4 -> [5, 6, 7]
Enter index block for file 2: 4
Index block already allocated.

...Program finished with exit code 0
Press ENTER to exit console.

```

## Task 4: Contiguous Memory Allocation

Simulate Worst-fit, Best-fit, and First-fit memory allocation strategies.

### Implementation:

```
def allocate_memory(strategy):
    partitions = list(map(int, input("Enter partition sizes: ").split()))
    processes = list(map(int, input("Enter process sizes: ").split()))
    allocation = [-1] * len(processes)
    for i, psize in enumerate(processes):
        idx = -1
        if strategy == "first":
            for j, part in enumerate(partitions):
                if part >= psize:
                    idx = j
                    break
        elif strategy == "best":
            best_fit = float("inf")
            for j, part in enumerate(partitions):
                if part >= psize and part < best_fit:
                    best_fit = part
                    idx = j
        elif strategy == "worst":
            worst_fit = -1
            for j, part in enumerate(partitions):
                if part >= psize and part > worst_fit:
                    worst_fit = part
                    idx = j
```

```

if idx != -1:
    allocation[i] = idx
    partitions[idx] -= psize
for i, a in enumerate(allocation):
    if a != -1:
        print(f"Process {i+1} allocated in Partition {a+1}")
    else:
        print(f"Process {i+1} cannot be allocated")

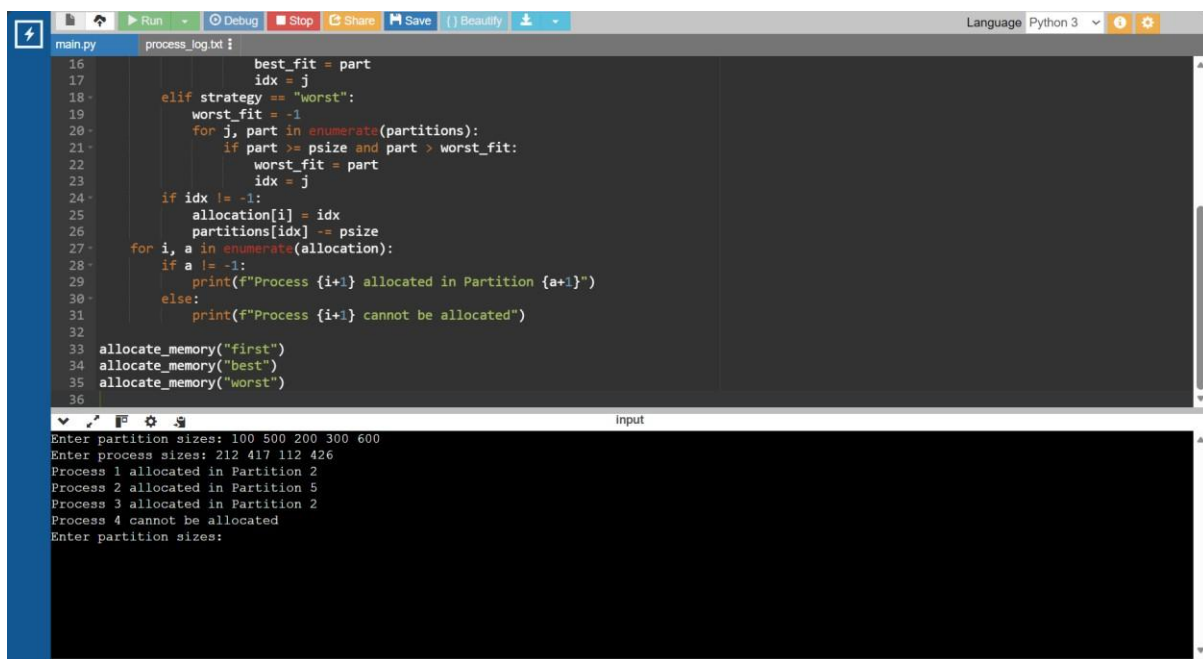
```

allocate\_memory("first")

allocate\_memory("best")

allocate\_memory("worst")

## Output:



The screenshot shows a Python IDE with a code editor and an output console. The code in the editor implements a memory allocation algorithm. It defines a function `allocate_memory(strategy)` that takes a strategy name as input. The strategy can be "first", "best", or "worst". The function iterates over processes and partitions, allocating memory based on the chosen strategy. The output console shows the results of running the code with specific input values.

```

16         best_fit = part
17         idx = j
18     elif strategy == "worst":
19         worst_fit = -1
20         for j, part in enumerate(partitions):
21             if part >= psize and part > worst_fit:
22                 worst_fit = part
23                 idx = j
24     if idx != -1:
25         allocation[i] = idx
26         partitions[idx] -= psize
27 for i, a in enumerate(allocation):
28     if a != -1:
29         print(f"Process {i+1} allocated in Partition {a+1}")
30     else:
31         print(f"Process {i+1} cannot be allocated")
32
33 allocate_memory("first")
34 allocate_memory("best")
35 allocate_memory("worst")
36

```

input

```

Enter partition sizes: 100 500 200 300 600
Enter process sizes: 212 417 112 426
Process 1 allocated in Partition 2
Process 2 allocated in Partition 5
Process 3 allocated in Partition 2
Process 4 cannot be allocated
Enter partition sizes:

```

### Task 5: MFT C MVT Memory Management

Implement MFT (fixed partitions) and MVT (variable partitions) strategies in Python.

#### Implementation:

```
def MFT():
```

```
    mem_size = int(input("Enter total memory size: "))
    part_size = int(input("Enter partition size: "))
    n = int(input("Enter number of processes: "))
    partitions = mem_size // part_size
    print(f'Memory divided into {partitions} partitions')
    for i in range(n):
        psize = int(input(f'Enter size of Process {i+1}: '))
        if psize <= part_size:
            print(f'Process {i+1} allocated.')
        else:
            print(f'Process {i+1} too large for fixed partition.')
```

```
def MVT():
```

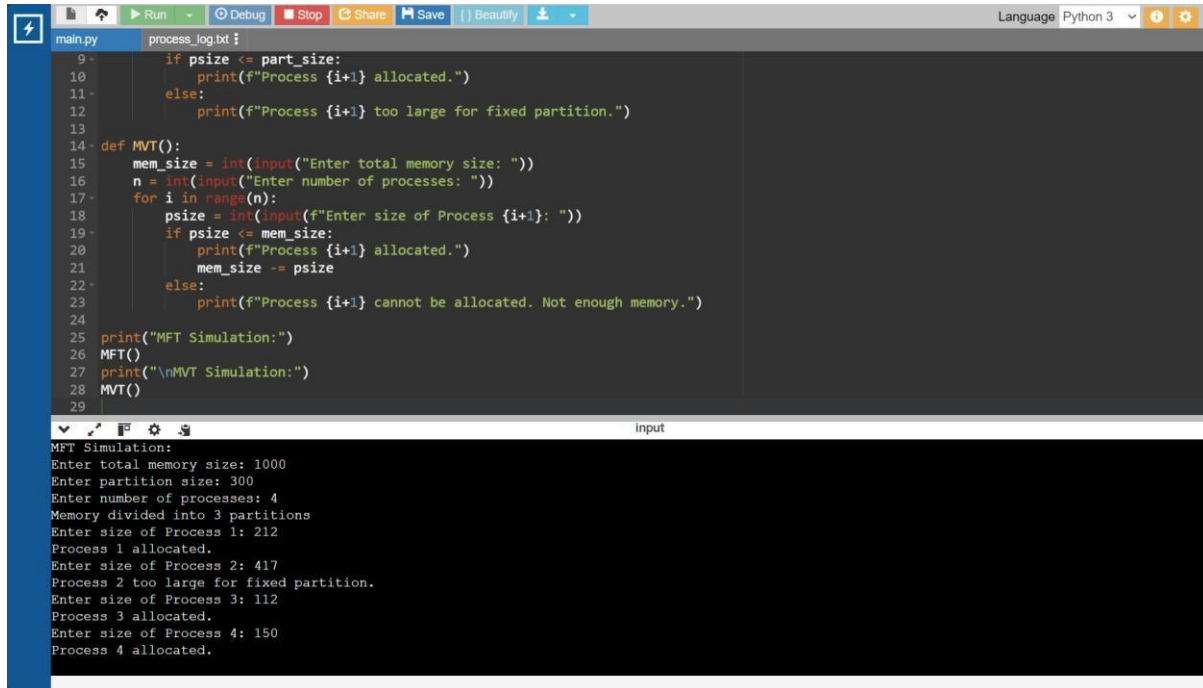
```
    mem_size = int(input("Enter total memory size: "))
    n = int(input("Enter number of processes: "))
    for i in range(n):
        psize = int(input(f'Enter size of Process {i+1}: '))
        if psize <= mem_size:
            print(f'Process {i+1} allocated.')
            mem_size -= psize
        else:
            print(f'Process {i+1} cannot be allocated. Not enough memory.')
```

```
print("MFT Simulation:")
```

```
MFT()
```

```
print("\nMVT Simulation:") MVT()
```

## Output:



The screenshot shows a Python IDE with a file named `main.py` and a terminal window. The code in `main.py` defines a function `MFT()` that simulates memory allocation. It prompts the user for total memory size, partition size, and number of processes. It then iterates through processes, checking if their size fits within the available memory partitions. The output in the terminal shows the results of the simulation for 4 processes.

```
main.py process_log.txt
9-      if psize <= part_size:
10-          print(f"Process {i+1} allocated.")
11-      else:
12-          print(f"Process {i+1} too large for fixed partition.")
13-
14- def MFT():
15-     mem_size = int(input("Enter total memory size: "))
16-     n = int(input("Enter number of processes: "))
17-     for i in range(n):
18-         psize = int(input(f"Enter size of Process {i+1}: "))
19-         if psize <= mem_size:
20-             print(f"Process {i+1} allocated.")
21-             mem_size -= psize
22-         else:
23-             print(f"Process {i+1} cannot be allocated. Not enough memory.")
24-
25- print("MFT Simulation:")
26- MFT()
27- print("\nMVT Simulation:")
28- MVT()
29-
input
MFT Simulation:
Enter total memory size: 1000
Enter partition size: 300
Enter number of processes: 4
Memory divided into 3 partitions
Enter size of Process 1: 212
Process 1 allocated.
Enter size of Process 2: 417
Process 2 too large for fixed partition.
Enter size of Process 3: 112
Process 3 allocated.
Enter size of Process 4: 150
Process 4 allocated.
```

## Task 1: Batch Processing Simulation (Python)

Write a Python script to execute multiple .py files sequentially, mimicking batch processing.

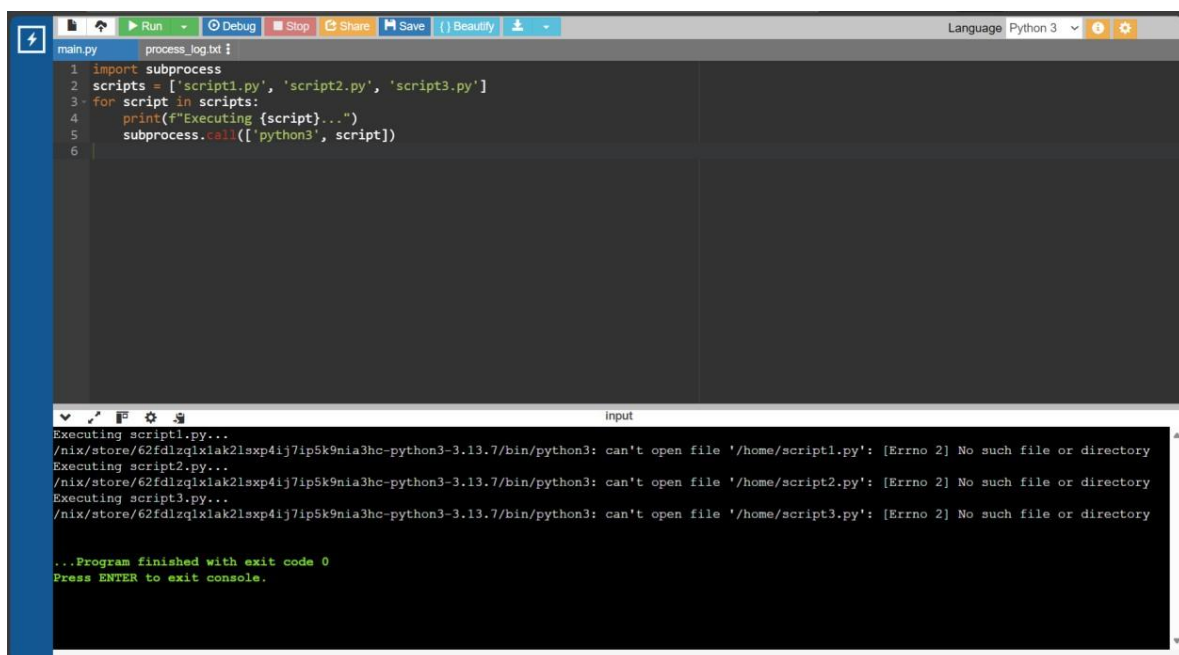
### Implementation:

```
import subprocess

scripts = ['script1.py', 'script2.py', 'script3.py']

for script in scripts:
    print(f"Executing {script}...")
    subprocess.call(['python3', script])
```

### Output:



```
main.py process_log.txt
1 import subprocess
2 scripts = ['script1.py', 'script2.py', 'script3.py']
3 for script in scripts:
4     print(f"Executing {script}...")
5     subprocess.call(['python3', script])
6

Executing script1.py...
/nix/store/62fdl2qlxak2l2sxp4ij7ip5k9nia3hc-python3-3.13.7/bin/python3: can't open file '/home/script1.py': [Errno 2] No such file or directory
Executing script2.py...
/nix/store/62fdl2qlxak2l2sxp4ij7ip5k9nia3hc-python3-3.13.7/bin/python3: can't open file '/home/script2.py': [Errno 2] No such file or directory
Executing script3.py...
/nix/store/62fdl2qlxak2l2sxp4ij7ip5k9nia3hc-python3-3.13.7/bin/python3: can't open file '/home/script3.py': [Errno 2] No such file or directory

...Program finished with exit code 0
Press ENTER to exit console.
```

## Task 2: System Startup and Logging

Simulate system startup using Python by creating multiple processes and logging their start and end into a log file.

### Implementation:

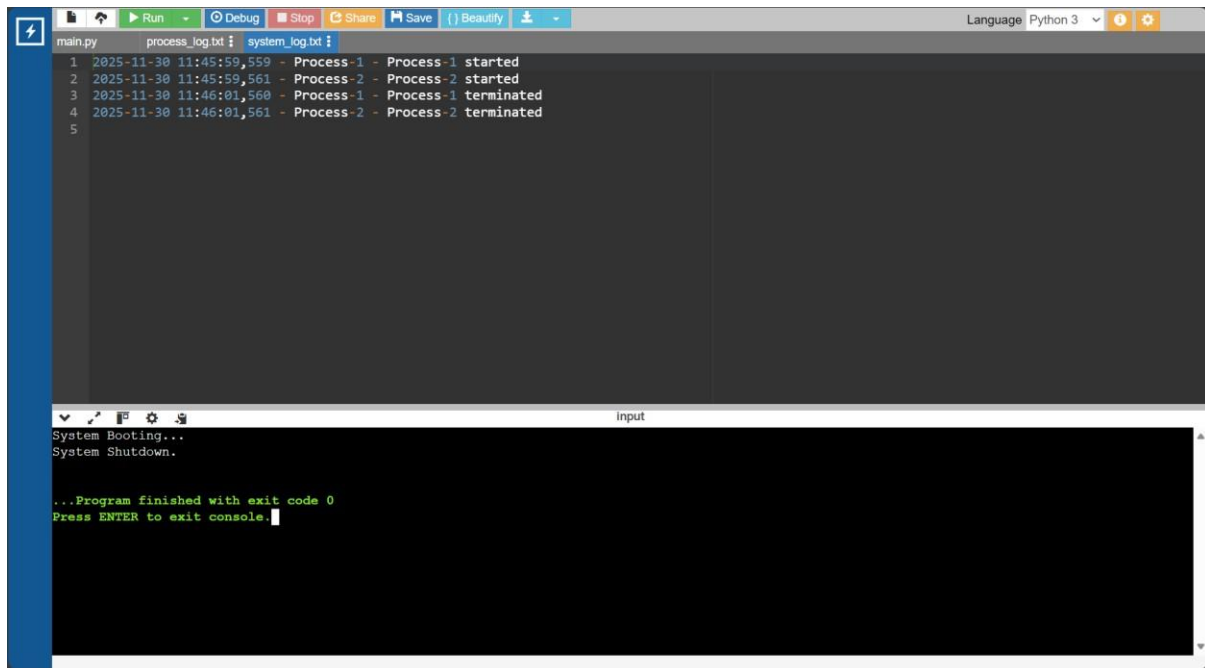
```
import multiprocessing
import logging
import time

logging.basicConfig(filename='system_log.txt', level=logging.INFO,
                    format='%(asctime)s - %(processName)s - %(message)s')

def process_task(name):
    logging.info(f"{name} started")
    time.sleep(2)
    logging.info(f"{name} terminated")

if __name__ == '__main__':
    print("System Booting...")
    p1 = multiprocessing.Process(target=process_task, args=("Process-1",))
    p2 = multiprocessing.Process(target=process_task, args=("Process-2",))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("System Shutdown.")
```

## Output:



The screenshot shows a code editor with a dark theme. The top toolbar includes icons for Run, Debug, Stop, Share, Save, and Beautify. The language is set to Python 3. The editor has three tabs: main.py, process\_log.txt, and system\_log.txt. The main.py tab is active, showing a script with five lines of code. The output window at the bottom shows the execution results, including system booting and shutdown messages, and a confirmation that the program finished with exit code 0.

```
main.py process_log.txt system_log.txt
1 2025-11-30 11:45:59,559 - Process-1 - Process-1 started
2 2025-11-30 11:45:59,561 - Process-2 - Process-2 started
3 2025-11-30 11:46:01,560 - Process-1 - Process-1 terminated
4 2025-11-30 11:46:01,561 - Process-2 - Process-2 terminated
5

Input
System Booting...
System Shutdown.

...Program finished with exit code 0
Press ENTER to exit console.
```

### Task 3: System Calls and IPC (Python - fork, exec, pipe)

Use system calls (fork(), exec(), wait()) and implement basic Inter-Process Communication using pipes in C or Python.

### Implementation:

```
import os

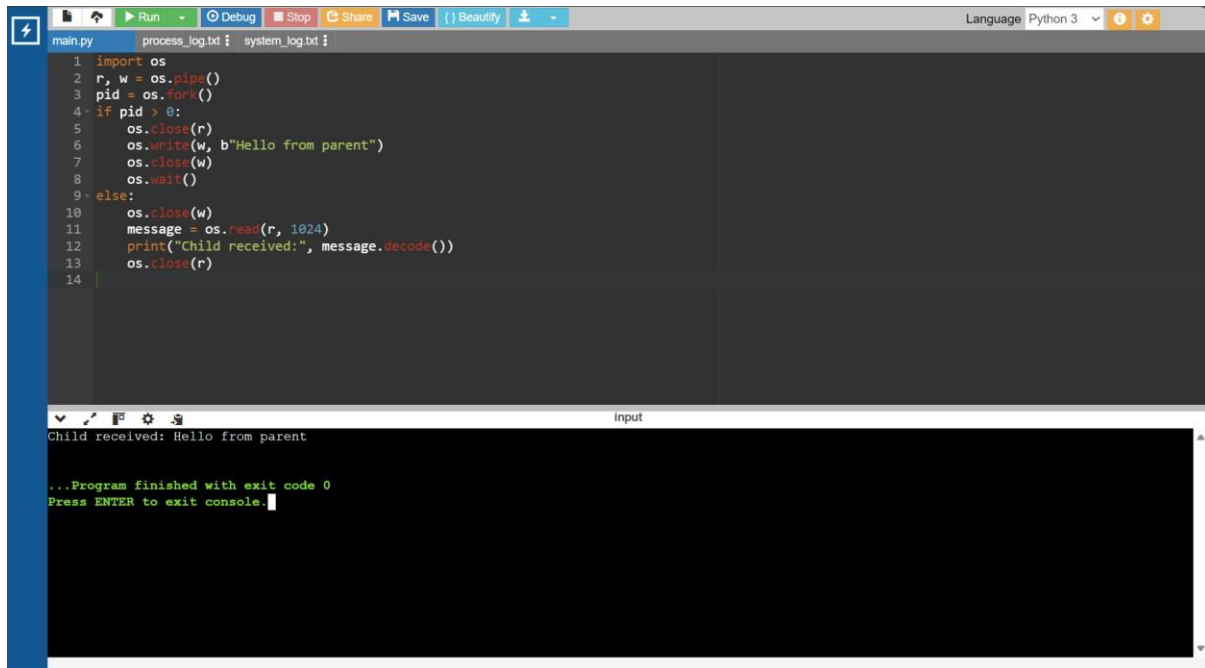
r, w = os.pipe()

pid = os.fork()

if pid > 0:
    os.close(r)
    os.write(w, b"Hello from parent")
    os.close(w)
    os.wait()
else:
    os.close(w)
    message = os.read(r, 1024)
    print("Child received:", message.decode())
```

```
os.close(r)
```

## Output:



The screenshot shows a Python IDE with a file named `main.py` containing the following code:

```
1 import os
2 r, w = os.pipe()
3 pid = os.fork()
4 if pid > 0:
5     os.close(r)
6     os.write(w, b"Hello from parent")
7     os.close(w)
8     os.wait()
9 else:
10    os.close(w)
11    message = os.read(r, 1024)
12    print("Child received:", message.decode())
13    os.close(r)
14
```

The output window at the bottom shows the execution results:

```
Child received: Hello from parent

...Program finished with exit code 0
Press ENTER to exit console.
```

## Task 4: VM Detection and Shell Interaction

Create a shell script to print system details and a Python script to detect if the system is running inside a virtual machine.

### Implementation:

```
#!/bin/bash
```

```
echo "Kernel Version:"
```

```
uname -r
```

```
echo "User:"
```

```
whoami
```

```
echo "Hardware Info:"
```

```
lscpu | grep 'Virtualization'
```

### Python Script:

```
import os
```

```
import subprocess
```

```
def check_dmi():
```

```
    """Check system DMI data for known VM identifiers."""
```

```
    vm_signatures = ["virtual", "vmware", "kvm", "qemu", "hyper-v", "xen"]
```

```
    try:
```

```
        output = subprocess.check_output(["sudo", "dmidecode"],
```

```
            stderr=subprocess.DEVNULL).decode().lower()
```

```
        return any(sig in output for sig in vm_signatures)
```

```
    except:
```

```
        return False
```

```
def check_cpu_flags():
```

```
    """Check CPU flags for hypervisor bit."""
```

```
    try:
```

```
        with open("/proc/cpuinfo") as f:
```

```
            data = f.read().lower()
```

```
        return "hypervisor" in data
```

```
    except:
```

```
        return False
```

```
def check_mac_address():
```

```
    """Check if the MAC address belongs to a VM vendor."""
```

```
    vm_mac_prefixes = [
```

```
        "00:05:69", "00:0C:29", "00:1C:14", # VMware
```

```
        "08:00:27",          # VirtualBox
```

```
        "52:54:00",          # QEMU / KVM
```

```
        "00:15:5D",          # Hyper-V
```

```
    ]
```

```
    try:
```

```
        output = subprocess.check_output(["ip", "link"]).decode().lower()
```

```

        for prefix in vm_mac_prefixes:
            if prefix.lower() in output:
                return True
    except:
        pass
    return False

def detect_vm():
    print("\n--- Virtual Machine Detection ---")
    dmi = check_dmi()
    hypervisor_flag = check_cpu_flags()
    mac_vm = check_mac_address()
    if dmi or hypervisor_flag or mac_vm:
        print("This system appears to be running inside a VIRTUAL MACHINE.")
    else:
        print("This system appears to be running on BARE METAL hardware.")
    print("\nDetails:")
    print(f"DMI-based detection: {dmi}")
    print(f"CPU hypervisor flag: {hypervisor_flag}")
    print(f"MAC address virtual: {mac_vm}")

if __name__ == "__main__":
    detect_vm()

```

Output:

## Task 5: CPU Scheduling Algorithms

Implement FCFS, SJF, Round Robin, and Priority Scheduling algorithms in Python to calculate WT and TAT.

### Implementation:

```
"""FCFS Scheduling"""
```

```
def fcfs(processes):
```

```
    processes.sort(key=lambda x: x['arrival'])
```

```
    time = 0
```

```
    for p in processes:
```

```
        if time < p['arrival']:
```

```
            time = p['arrival']
```

```
        p['wt'] = time - p['arrival']
```

```
        time += p['burst']
```

```
        p['tat'] = p['wt'] + p['burst']
```

```
    return processes
```

```
"""SJF Scheduling"""
```

```
def sjf(processes):
```

```
    processes = sorted(processes, key=lambda x: x['arrival'])
```

```
    completed, time = 0, 0
```

```
    n = len(processes)
```

```
    while completed < n:
```

```
        available = [p for p in processes if p['arrival'] <= time and 'done' not in p]
```

```
        if not available:
```

```
            time += 1
```

```
            continue
```

```
        p = min(available, key=lambda x: x['burst'])
```

```
        p['wt'] = time - p['arrival']
```

```
        time += p['burst']
```

```
        p['tat'] = p['wt'] + p['burst']
```

```
p['done'] = True
completed += 1
return processes
```

```
"""Round Robin"""
```

```
def round_robin(processes, quantum):
    from collections import deque
    q = deque()
    time = 0
    remaining = {p['pid']: p['burst'] for p in processes}
    processes.sort(key=lambda x: x['arrival'])
    i = 0
    completed = 0
    n = len(processes)
    while completed < n:
        while i < n and processes[i]['arrival'] <= time:
            q.append(processes[i])
            i += 1
        if not q:
            time = processes[i]['arrival']
            continue
        p = q.popleft()
        exec_time = min(quantum, remaining[p['pid']])
        remaining[p['pid']] -= exec_time
        time += exec_time
        while i < n and processes[i]['arrival'] <= time:
            q.append(processes[i])
            i += 1
        if remaining[p['pid']] == 0:
            p['tat'] = time - p['arrival']
            p['wt'] = p['tat'] - p['burst']
```

```

        completed += 1
    else:
        q.append(p)
    return processes

```

"""Priority Scheduling"""

```

def priority_scheduling(processes):
    time = 0
    completed = 0
    n = len(processes)
    processes.sort(key=lambda x: x['arrival'])
    while completed < n:
        available = [p for p in processes if p['arrival'] <= time and 'done' not in p]
        if not available:
            time += 1
            continue
        p = min(available, key=lambda x: x['priority'])
        p['wt'] = time - p['arrival']
        time += p['burst']
        p['tat'] = p['wt'] + p['burst']
        p['done'] = True
        completed += 1
    return processes

```

```

processes = [
    {'pid': 1, 'arrival': 0, 'burst': 5, 'priority': 2},
    {'pid': 2, 'arrival': 1, 'burst': 3, 'priority': 1},
    {'pid': 3, 'arrival': 2, 'burst': 8, 'priority': 4},
    {'pid': 4, 'arrival': 3, 'burst': 6, 'priority': 3},
]

```

```

import copy

print("\n--- FCFS ---")

for p in fcfs(copy.deepcopy(processes)):

    print(p)

print("\n--- SJF ---")

for p in sjf(copy.deepcopy(processes)):

    print(p)

print("\n--- Round Robin (Q=2) ---")

for p in round_robin(copy.deepcopy(processes), quantum=2):

    print(p)

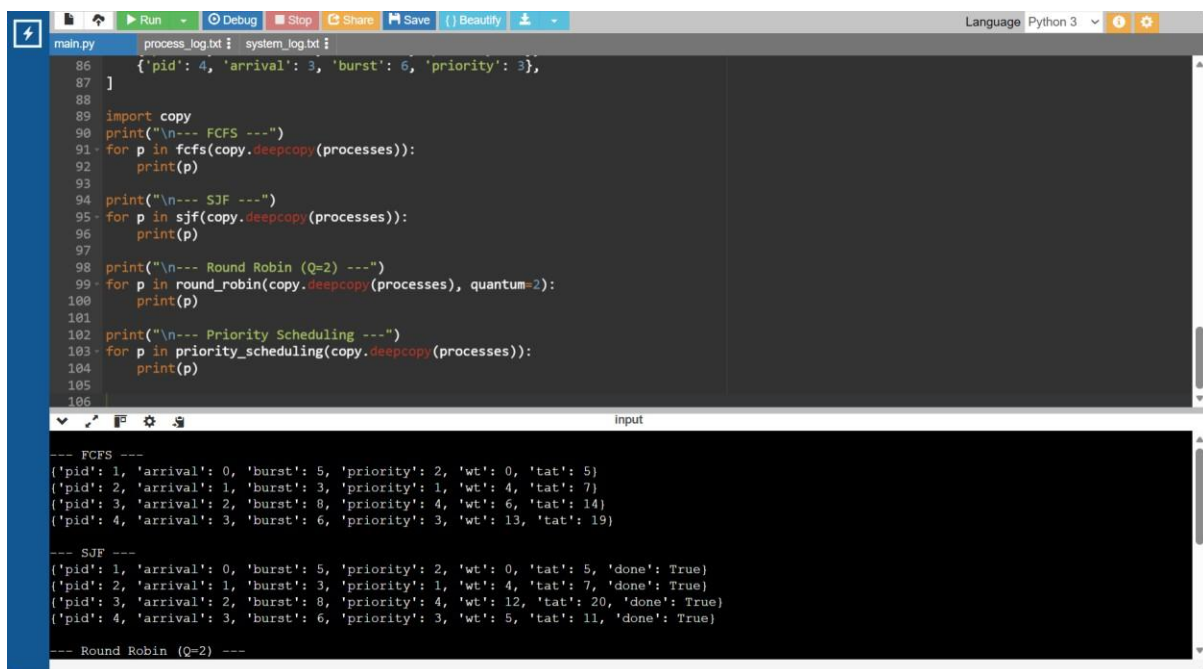
print("\n--- Priority Scheduling ---")

for p in priority_scheduling(copy.deepcopy(processes)):

    print(p)

```

## Output:



```

main.py  process_log.txt  system_log.txt
86  {'pid': 4, 'arrival': 3, 'burst': 6, 'priority': 3},
87  ]
88
89  import copy
90  print("\n--- FCFS ---")
91  for p in fcfs(copy.deepcopy(processes)):
92      print(p)
93
94  print("\n--- SJF ---")
95  for p in sjf(copy.deepcopy(processes)):
96      print(p)
97
98  print("\n--- Round Robin (Q=2) ---")
99  for p in round_robin(copy.deepcopy(processes), quantum=2):
100      print(p)
101
102  print("\n--- Priority Scheduling ---")
103  for p in priority_scheduling(copy.deepcopy(processes)):
104      print(p)
105
106
--- FCFS ---
({'pid': 1, 'arrival': 0, 'burst': 5, 'priority': 2, 'wt': 0, 'tat': 5})
({'pid': 2, 'arrival': 1, 'burst': 3, 'priority': 1, 'wt': 4, 'tat': 7})
({'pid': 3, 'arrival': 2, 'burst': 8, 'priority': 4, 'wt': 6, 'tat': 14})
({'pid': 4, 'arrival': 3, 'burst': 6, 'priority': 3, 'wt': 13, 'tat': 19})

--- SJF ---
({'pid': 1, 'arrival': 0, 'burst': 5, 'priority': 2, 'wt': 0, 'tat': 5, 'done': True})
({'pid': 2, 'arrival': 1, 'burst': 3, 'priority': 1, 'wt': 4, 'tat': 7, 'done': True})
({'pid': 3, 'arrival': 2, 'burst': 8, 'priority': 4, 'wt': 12, 'tat': 20, 'done': True})
({'pid': 4, 'arrival': 3, 'burst': 6, 'priority': 3, 'wt': 5, 'tat': 11, 'done': True})

--- Round Robin (Q=2) ---

```