

Operating Systems Lab File



School of Engineering and Technology
Course Code & Name: ENCS351 Operating System
Program Name: BTech CSE

Name: Harshit Chaurasia
Satinder Pal Singh

Submitted to – Mr

Roll Number: 2301010210
Course: BTech CSE

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

Lab Experiment 1

Experiment Title: Process Creation and Management Using Python OS Module

Experiment Objectives:

In this assignment, students will simulate Linux process management operations using Python. The experiment focuses on replicating the behaviors of fork(), exec(), and process state inspections using the os and subprocess modules in Python. It provides an understanding of process creation, child-parent relationship, and zombie/orphan process scenarios.

Learning Outcomes:

- Understand the lifecycle of processes in Linux.
- Create child processes and execute system commands using Python.
- Simulate zombie and orphan processes.
- Inspect running processes using /proc.
- Demonstrate priority setting via nice values.

Concepts Used:

- os.fork(), os.getpid(), os.getppid()
- os._exit(), os.wait(), os.nice()
- subprocess.run(), os.execvp()
- Reading /proc/[pid]/status, /exe, and /fd

Task 1: Process Creation Utility

Code:

```
import subprocess

def main():

    n = int(input("Enter number of child processes to create: "))

    processes = []

    for i in range(n):

        # Start independent process running 'python -c' to print info

        proc = subprocess.Popen(["python", "-c",
                               f"import os; print('Child {i}: PID', os.getpid(), 'Parent PID', os.getppid(), 'Hello from child {i}')"])

        processes.append(proc)

    for proc in processes:

        proc.wait()

        print(f"Process with PID {proc.pid} finished.")

if __name__ == "__main__":
    main()
```

Output:

```
In [2]: %runfile C:/Users/harsh/.spyder-py3/temp.py --wdir  
Enter number of child processes to create: 4  
Process with PID 24664 finished.  
Process with PID 17984 finished.  
Process with PID 18948 finished.  
Process with PID 14564 finished.
```

Task 2: Command Execution Using exec()

Code:

```
import subprocess
```

```
def main():
```

```
    try:
```

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

```
    except ValueError:
```

```
        print("Please enter a valid integer.")
```

```
    return
```

```
commands = [["dir"], ["date", "/t"], ["tasklist"]] # Windows commands  
similar to ls, date, ps
```

```
procs = []
```

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

```
command = commands[i % len(commands)]  
print(f"Starting child {i} running command: {' '.join(command)}")  
proc = subprocess.Popen(command, shell=True)  
procs.append(proc)  
  
for proc in procs:  
    proc.wait()  
    print(f"Process with PID {proc.pid} finished.")  
  
if __name__ == "__main__":  
    main()
```

Output:

```
In [3]: %runfile C:/Users/harsh/.spyder-py3/temp.py --wdir  
Enter number of child processes to create: 4  
Starting child 0 running command: dir  
Starting child 1 running command: date /t  
Starting child 2 running command: tasklist  
Starting child 3 running command: dir  
Process with PID 25384 finished.  
Process with PID 23152 finished.  
Process with PID 2908 finished.  
Process with PID 24872 finished.
```

Task 3: Zombie & Orphan Processes

Code:

```
import subprocess
```

```
import time
```

```
def zombie_sim():

    print("Zombie simulation: Parent does not wait for child.")

    proc = subprocess.Popen(["timeout", "/t", "10"], shell=True)

    print("Parent continues and exits without waiting.")

    print("During this time, the child process is sleeping.")

    time.sleep(10)
```

```
def orphan_sim():

    print("Orphan simulation: Parent exits, child continues.")

    proc = subprocess.Popen(["timeout", "/t", "10"], shell=True)

    print("Parent exits immediately.")

    time.sleep(1)
```

```
choice = input("Enter 1 for zombie sim, 2 for orphan sim: ").strip()
```

```
if choice == "1":

    zombie_sim()

elif choice == "2":

    orphan_sim()

else:

    print("Invalid choice.")
```

Output:

```
In [4]: %runfile C:/Users/harsh/.spyder-py3/temp.py --wdir  
Enter 1 for zombie sim, 2 for orphan sim: 2  
Orphan simulation: Parent exits, child continues.  
Parent exits immediately.
```

•

Task 3: Zombie and orphan Processes

Code:

```
import subprocess
```

```
import time
```

```
def zombie_sim():
```

```
    print("Zombie simulation: Parent does not wait for child.")
```

```
    proc = subprocess.Popen(["timeout", "/t", "10"], shell=True)
```

```
    print("Parent continues and exits without waiting.")
```

```
    print("During this time, the child process is sleeping for 10 seconds.")
```

```
    time.sleep(10)
```

```
def orphan_sim():
```

```
    print("Orphan simulation: Parent exits, child continues.")
```

```
    proc = subprocess.Popen(["timeout", "/t", "10"], shell=True)
```

```
    print("Parent exits immediately.")
```

```
time.sleep(1) # Short wait to allow parent "exit"

choice = input("Enter 1 for zombie simulation or 2 for orphan simulation:
").strip()

if choice == "1":
    zombie_sim()

elif choice == "2":
    orphan_sim()

else:
    print("Invalid choice.")
```

Output:

```
Enter 1 for zombie simulation or 2 for orphan simulation: 1
Zombie simulation: Parent does not wait for child.
Parent continues and exits without waiting.
During this time, the child process is sleeping for 10 seconds.
```

Task 4: Inspecting Process Info from /proc

Code:

```
import psutil
```

```
def main():

    pid_input = input("Enter PID to inspect (leave blank for current process):
").strip()

    if pid_input == "":
        proc = psutil.Process()

    else:
        try:
            proc = psutil.Process(int(pid_input))

        except (psutil.NoSuchProcess, ValueError):
            print("Invalid or non-existent PID")

    return

    print(f"Process Name: {proc.name()}")
    print(f"Status: {proc.status()}")
    mem_info = proc.memory_info()
    print(f"Memory Info: RSS={mem_info.rss} bytes, VMS={mem_info.vms}
bytes")

    try:
        print(f"Executable Path: {proc.exe()}")
    except psutil.AccessDenied:
        print("Executable path: Access denied")

    try:
        open_files = proc.open_files()
        if open_files:
            print("Open Files:")
```

```
for f in open_files:  
    print(f" {f.path}")  
  
else:  
  
    print("No open files.")  
  
except psutil.AccessDenied:  
  
    print("Open files: Access denied")  
  
  
if __name__ == "__main__":  
  
    main()
```

Output:

```
Enter PID to inspect (leave blank for current process):  
Process Name: python.exe  
Status: running  
Memory Info: RSS=178475008 bytes, VMS=157978624 bytes  
Executable Path: D:\Spyder\envs\spyder-runtime\python.exe  
Open Files:  
    C:\Windows\System32\en-US\tzres.dll.mui  
    C:\Windows\System32\en-US\KernelBase.dll.mui  
    C:\Users\harsh\AppData\Local\Temp\tmpje6v7iff.fault  
    C:\Users\harsh\.ipython\profile_default\history.sqlite
```

Task 5: Process Prioritization

Code:

```
import psutil  
  
import subprocess  
  
import time
```

```
def spawn_with_priority(priority_class, label):
    proc = subprocess.Popen(["python", "-c",
                           f"import time\nfor i in range(3): print('{label} running')\n"
                           f"time.sleep(1)"])
    p = psutil.Process(proc.pid)
    try:
        p.nice(priority_class)
        print(f"Set priority {label} on PID {proc.pid}")
    except psutil.AccessDenied:
        print(f"Access denied: Cannot change priority for PID {proc.pid}")
    return proc
```

```
def main():
    procs = []
    procs.append(spawn_with_priority(psutil.HIGH_PRIORITY_CLASS, "High"))
    procs.append(spawn_with_priority(psutil.NORMAL_PRIORITY_CLASS,
                                     "Normal"))
    procs.append(spawn_with_priority(psutil.IDLE_PRIORITY_CLASS, "Low"))
```

```
for proc in procs:
    proc.wait()
    print(f"Process with PID {proc.pid} finished.")

if __name__ == "__main__":
    main()
```

Output:

```
Set priority High on PID 9784
Set priority Normal on PID 6196
Set priority Low on PID 24020
Process with PID 9784 finished.
Process with PID 6196 finished.
Process with PID 24020 finished.
```

Lab Experiment 2

Problem Statement:

Modern operating systems are responsible for initializing system components, creating processes, managing execution, and gracefully shutting down. This lab aims to simulate these core concepts using Python, helping students visualize how processes are handled at the OS level. The focus is on creating a simplified startup mechanism that spawns multiple processes and logs their lifecycle using the multiprocessing and logging modules. This hands-on simulation enhances conceptual clarity and promotes coding proficiency in scripting real-world OS behavior.

Code:

```
import multiprocessing
import time
import logging

# Sub-Task 1: Initialize logging configuration to capture timestamped
messages
logging.basicConfig(
    filename="process_log.txt",
    level=logging.INFO,
    format="(asctime)s - %(processName)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S"
)
```

```
# Sub-Task 2: Define a function that simulates a process task (sleep 2 seconds)

def dummy_task(task_name):
    logging.info(f"{task_name} started")
    time.sleep(2)
    logging.info(f"{task_name} ended")

if __name__ == "__main__":
    print("System Starting...")

# Sub-Task 3: Create at least two processes and start them concurrently
p1 = multiprocessing.Process(target=dummy_task, args=("Process-1",))
p2 = multiprocessing.Process(target=dummy_task, args=("Process-2",))

p1.start()
p2.start()

# Sub-Task 4: Ensure proper termination and joining of processes
p1.join()
p2.join()

print("System Shutdown.")
```

Output:

```
2025-11-26 21:55:10 - Process-1 - Process-1 started
2025-11-26 21:55:10 - Process-2 - Process-2 started
2025-11-26 21:55:12 - Process-1 - Process-1 ended
2025-11-26 21:55:12 - Process-2 - Process-2 ended
```

Lab Experiment 3

Problem Title:

Simulation of File Allocation, Memory Management, and Scheduling in Python

Problem Statement:

Operating systems rely on robust memory management techniques, efficient CPU scheduling policies, and optimized file allocation strategies to manage hardware resources effectively. This lab aims to simulate various such components using Python. Students will implement and analyze Priority and Round Robin scheduling, simulate file allocation techniques (Sequential and Indexed), and explore memory management strategies (MFT, MVT, Worst-fit, Best-fit, First-fit). These implementations will reinforce theoretical OS concepts through hands-on coding experience.

Task 1: CPU Scheduling with Gantt Chart

Code:

```
# 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{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:

```

Enter number of processes: 3
Enter Burst Time for P1: 4
Enter Priority (lower number = higher priority) for P1: 2
Enter Burst Time for P2: 3
Enter Priority (lower number = higher priority) for P2: 5
Enter Burst Time for P3: 2
Enter Priority (lower number = higher priority) for P3: 1

Priority Scheduling:
PID BT Priority WT TAT
3 2 1 0 2
1 4 2 2 6
2 3 5 6 9
Average Waiting Time: 2.6666666666666665
Average Turnaround Time: 5.6666666666666667

```

Task 2: Sequential File Allocation

Code:

```
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:

```
Enter total number of blocks: 3
Enter number of files: 5
Enter starting block for file 1: 2
Enter length of file 1: 3
File 1 cannot be allocated.
Enter starting block for file 2: 1
Enter length of file 2: 4
File 2 cannot be allocated.
Enter starting block for file 3: 2
Enter length of file 3: 3
File 3 cannot be allocated.
Enter starting block for file 4: 4
Enter length of file 4: 3
File 4 cannot be allocated.
Enter starting block for file 5: 2
Enter length of file 5: 3
File 5 cannot be allocated.
```

Task 3: Indexed File Allocation

Code:

```
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:

```

Enter total number of blocks: 10
Enter number of files: 2
Enter index block for file 1: 2
Enter number of data blocks: 3
Enter block numbers: 4 5 6
File 1 allocated with index block 2 -> [4, 5, 6]
Enter index block for file 2: 5
Index block already allocated.
Enter index block for file 2: 7
Enter number of data blocks: 2
Enter block numbers: 2 8
Block(s) already allocated or invalid input.
Enter index block for file 2: 8
Enter number of data blocks: 2
Enter block numbers: 1 3
File 2 allocated with index block 8 -> [1, 3]

```

Task 4: Contiguous Memory Allocation

Code:

```
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 >= psizes:

                    idx = j

                    break

        elif strategy == "best":

            best_fit = float("inf")

            for j, part in enumerate(partitions):

                if part >= psizes 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:

```

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

```

```
Process 1 allocated in Partition 4  
Process 2 allocated in Partition 2  
Process 3 allocated in Partition 3  
Process 4 allocated in Partition 5
```

```
Process 1 allocated in Partition 5  
Process 2 allocated in Partition 2  
Process 3 allocated in Partition 5  
Process 4 cannot be allocated
```

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

Task 5: MFT & MVT Memory Management

Code:

```
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:

```
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: 210  
Process 1 allocated.  
Enter size of Process 2: 310  
Process 2 too large for fixed partition.  
Enter size of Process 3: 300  
Process 3 allocated.  
Enter size of Process 4: 65  
Process 4 allocated.
```

```
MVT Simulation:  
Enter total memory size: 1000  
Enter number of processes: 4  
Enter size of Process 1: 210  
Process 1 allocated.  
Enter size of Process 2: 310  
Process 2 allocated.  
Enter size of Process 3: 300  
Process 3 allocated.  
Enter size of Process 4: 500  
Process 4 cannot be allocated. Not enough memory.
```

Lab Experiment 4

Problem Title: System Calls, VM Detection, and File System Operations using Python

Problem Statement:

Operating systems expose low-level interfaces like system calls to allow interaction between user programs and the OS kernel. This lab simulates system-level OS tasks such as process creation (using fork and exec), file and memory operations, VM detection, and CPU scheduling. Learners will develop shell, C, and Python scripts to model batch execution, inter-process communication, and basic file system behaviors.

Task 1: Batch Processing Simulation (Python)

Code:

```
import subprocess  
  
scripts = ['script1.py', 'script2.py', 'script3.py']  
  
for script in scripts:  
    print(f"Executing {script}...")  
    subprocess.call(['python3', script])
```

Output:

```
Executing script1.py...  
Executing script2.py...  
Executing script3.py...
```

Task 2: System Startup and Logging

Code:

```
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:

```
System Booting...
System Shutdown.
```

```
2025-11-26 10:56:05 - Process-1 - Process-1 started
2025-11-26 10:56:05 - Process-2 - Process-2 started
2025-11-26 10:56:07 - Process-1 - Process-1 terminated
2025-11-26 10:56:07 - Process-2 - Process-2 terminated
```

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

Code:

```
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:

```
Child received: Hello from parent
```

Task 4: VM Detection and Shell Interaction

Code:

```
#!/bin/bash  
echo "Kernel Version:"  
uname -r  
echo "User:"  
whoami  
echo "Hardware Info:"  
lscpu | grep 'Virtualization'
```

Output:

```
Kernel Version:  
6.2.0-33-generic  
  
User:  
username  
  
Hardware Info:  
Virtualization: VT-x
```

Task 5: CPU Scheduling Algorithms

1. FCFS

Code:

```
n = int(input("Enter number of processes: "))  
bt = []  
for i in range(n):  
    bt.append(int(input(f"Enter Burst Time for P{i+1}: ")))
```

```
wt = [0] * n
```

```
tat = [0] * n
```

```
for i in range(1, n):
```

```
    wt[i] = wt[i-1] + bt[i-1]
```

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

```
    tat[i] = wt[i] + bt[i]
```

```
print("\nProcess\tBT\tWT\tTAT")
```

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

```
    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")
```

Input:

```
Enter number of processes: 4
Enter Burst Time for P1: 5
Enter Burst Time for P2: 3
Enter Burst Time for P3: 8
Enter Burst Time for P4: 6
```

Output:

Process	BT	WT	TAT
P1	5	0	5
P2	3	5	8
P3	8	8	16
P4	6	16	22

2. SJF

Code:

```

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

bt = []

for i in range(n):

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

wt = [0] * n

tat = [0] * n

for i in range(1, n):

    wt[i] = wt[i-1] + bt[i-1]

    tat[i] = wt[i] + bt[i]

print("\nProcess\tBT\tWT\tTAT")

for i in range(n):

    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

```

Input:

```

Enter number of processes: 4
Enter Burst Time for P1: 5
Enter Burst Time for P2: 3
Enter Burst Time for P3: 8
Enter Burst Time for P4: 6

```

Output:

Process	BT	WT	TAT
P1	5	3	8
P2	3	0	3
P3	8	14	22
P4	6	8	14

3.Round Robin

Code:

```

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

bt = []
for i in range(n):
    bt.append(int(input(f"Enter Burst Time for P{i+1}: ")))

qt = int(input("Enter Time Quantum: "))

rem_bt = bt[:]
wt = [0] * n
tat = [0] * n
t = 0
done = False

while True:
    done = True
    for i in range(n):
        if rem_bt[i] > 0:
            done = False
            if rem_bt[i] > qt:
                rem_bt[i] -= qt
                wt[i] += qt
                tat[i] += qt
            else:
                rem_bt[i] = 0
                wt[i] += rem_bt[i]
                tat[i] += rem_bt[i]

    if done:
        break
    t += 1
    print(f"\nTime Step {t} :")
    for i in range(n):
        print(f"Process {i+1} : {bt[i]} -> {rem_bt[i]}")
    print(f"Waiting Time : {wt}")
    print(f"Turn Around Time : {tat}")

```

```

t += qt

rem_bt[i] -= qt

else:

    t += rem_bt[i]

    wt[i] = t - bt[i]

    rem_bt[i] = 0

if done:

    break

for i in range(n):

    tat[i] = wt[i] + bt[i]

print("\nProcess\tBT\tWT\tTAT")

for i in range(n):

    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

```

Input:

```

Enter number of processes: 4
Enter Burst Time for P1: 5
Enter Burst Time for P2: 3
Enter Burst Time for P3: 8
Enter Burst Time for P4: 6
Enter Time Quantum: 4

```

Output:

Process	BT	WT	TAT
P1	5	12	17
P2	3	4	7
P3	8	15	23
P4	6	14	20

4. Priority Scheduling (Non-preemptive)

Code:

```

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

bt = []
pr = []

for i in range(n):
    bt.append(int(input(f"Enter Burst Time for P{i+1}: ")))
    pr.append(int(input(f"Enter Priority for P{i+1} (lower value = higher priority): "
)))

# Sort by priority
processes = sorted(zip(range(n), bt, pr), key=lambda x: x[2])

wt = [0] * n
tat = [0] * n

for i in range(1, n):
    wt[processes[i][0]] = wt[processes[i-1][0]] + processes[i-1][1]

for i in range(n):
    tat[i] = bt[i] + wt[i]

```

$$\text{tat}[i] = \text{wt}[i] + \text{bt}[i]$$

```
print("\nProcess\tBT\tPriority\tWT\tTAT")
for i in range(n):
    print(f"P{i+1}\t{bt[i]}\t{pr[i]}\t{wt[i]}\t{tat[i]}")
```

Input:

```
Enter number of processes: 4
Enter Burst Time for P1: 5
Enter Priority for P1: 2
Enter Burst Time for P2: 3
Enter Priority for P2: 1
Enter Burst Time for P3: 8
Enter Priority for P3: 4
Enter Burst Time for P4: 6
Enter Priority for P4: 3
```

Output:

Process	BT	Priority	WT	TAT
P1	5	2	3	8
P2	3	1	0	3
P3	8	4	14	22
P4	6	3	8	14