

DSCC Lab

Assignment-5

Objective: To implement and analyze scheduling strategies used in distributed systems and evaluate their performance using Completion Time, Waiting Time, and Turnaround Time metrics.

First Come First Serve

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

for i in range(n):
    at = int(input(f"P{i+1} Arrival Time: "))
    bt = int(input(f"P{i+1} Burst Time: "))
    processes.append([i+1, at, bt])
processes.sort(key=lambda x: x[1])

time = 0
total_wt = total_tat = 0
gantt = []

print("\nP AT BT CT TAT WT")

for p in processes:
    pid, at, bt = p
    if time < at:
        gantt.append(("Idle", at-time))

    gantt.append((f"P{pid}", bt))
    ct = time + bt
    tat = ct - at
    wt = tat - bt
    total_wt += wt
    total_tat += tat
    time = ct

    print(f"P{pid} {at} {bt} {ct} {tat} {wt}")
print("\nAverage WT:", round(total_wt/n,2))
print("Average TAT:", round(total_tat/n,2))

# Gantt Chart
print("\nGantt Chart:")
for p in gantt:
    print("|", p[0], end=" ")
print("|")
```

```

Enter number of processes: 3
P1 Arrival Time: 0
P1 Burst Time: 5
P2 Arrival Time: 1
P2 Burst Time: 3
P3 Arrival Time: 2
P3 Burst Time: 1

P  AT  BT  CT  TAT  WT
P1  0  5  5  5  0
P2  1  3  8  7  4
P3  2  1  9  7  6

Average WT: 3.3333333333333335
Average TAT: 6.333333333333333

```

Non Preemptive Priority Scheduling

```

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

for i in range(n):
    at = int(input(f"P{i+1} Arrival Time: "))
    bt = int(input(f"P{i+1} Burst Time: "))
    pr = int(input(f"P{i+1} Priority: "))
    processes.append([i+1, at, bt, pr, 0])

time = 0
completed = 0
total_wt = total_tat = 0
gantt = []
print("\nP  AT  BT  PR  CT  TAT  WT")

while completed < n:
    ready = [p for p in processes if p[1] <= time
              and p[4] == 0]

    if not ready:
        time += 1
        continue

    ready.sort(key=lambda x: x[3])
    p = ready[0]

    pid, at, bt, pr, _ = p
    ganttt.append((f"P{pid}", bt))

    ct = time + bt
    tat = ct - at
    wt = tat - bt
    total_wt += wt
    total_tat += tat
    time = ct

    p[4] = 1
    completed += 1

    print(f"P{pid} {at} {bt} {pr} {ct} {tat} {wt}")

print("\nAverage WT:", round(total_wt/n,2))
print("Average TAT:", round(total_tat/n,2))

```

```
print("\nGantt Chart:")
```

```
for p in gantt:
```

```
print("|", p[0], end=" ")
```

```
print("|")
```

```
Enter number of processes: 3
P1 Arrival Time: 0
P1 Burst Time: 4
P1 Priority: 2
P2 Arrival Time: 1
P2 Burst Time: 3
P2 Priority: 1
P3 Arrival Time: 2
P3 Burst Time: 2
P3 Priority: 3
```

P	AT	BT	PR	CT	TAT	WT
P1	0	4	2	4	4	0
P2	1	3	1	7	6	3
P3	2	2	3	9	7	5

```
Average WT: 2.67
Average TAT: 5.67
```

```
Gantt Chart:
```

```
| P1 | P2 | P3 |
```

Preemptive Priority Scheduling

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

```
processes = []
```

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

```
    at = int(input(f"P{i+1} Arrival Time: "))
```

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

```
    pr = int(input(f"P{i+1} Priority: "))
```

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

```
time = 0
```

```
completed = 0
```

```
ct = [0]*n
```

```
gantt = []
```

```
while completed < n:
```

```
    ready = [p for p in processes if p[1] <= time
and p[4] > 0]
```

```
    if not ready:
```

```
        gantt.append("Idle")
```

```
        time += 1
```

```
        continue
```

```
    ready.sort(key=lambda x: x[3])
```

```
    p = ready[0]
```

```
    gantt.append(f"P{p[0]}")
```

```
    p[4] -= 1
```

```
    time += 1
```

```
    if p[4] == 0:
```

```
        ct[p[0]-1] = time
```

```

        completed += 1
print("\nP AT BT PR CT TAT WT")

```

```

total_wt = total_tat = 0

```

```

for p in processes:

```

```

    pid, at, bt, pr, _ = p

```

```

    tat = ct[pid-1] - at

```

```

    wt = tat - bt

```

```

    total_wt += wt

```

```

    total_tat += tat

```

```

        print(f"P{pid} {at} {bt} {pr} {ct[pid-1]} {tat} {wt}")

```

```

print("\nAverage WT:", round(total_wt/n,2))

```

```

print("Average TAT:", round(total_tat/n,2))

```

```

print("\nGantt Chart:")

```

```

for g in gantt:

```

```

    print("|", g, end=" ")

```

```

print("|")

```

```

Enter number of processes: 3

```

```

P1 Arrival Time: 0

```

```

P1 Burst Time: 4

```

```

P1 Priority: 2

```

```

P2 Arrival Time: 1

```

```

P2 Burst Time: 3

```

```

P2 Priority: 1

```

```

P3 Arrival Time: 2

```

```

P3 Burst Time: 2

```

```

P3 Priority: 3

```

```

P  AT  BT  PR  CT  TAT  WT
P1  0   4   2   7   7   3
P2  1   3   1   4   3   0
P3  2   2   3   9   7   5

```

```

Average WT: 2.67

```

```

Average TAT: 5.67

```

```

Gantt Chart:

```

```

Average WT: 2.67

```

```

Average TAT: 5.67

```

```

Gantt Chart:

```

```

| P1 | P2 | P2 | P2 | P1 | P1 | P1 | P3 | P3 |

```

```

PS C:\Users\LENOVO\Desktop\DSCC Lab\Lab 4>

```

Analysis Questions

1. What is the impact of arrival time on FCFS scheduling?

In FCFS (First Come First Serve), processes are executed in the order they arrive.

Impact of Arrival Time:

- The process that arrives first gets CPU first.
- If a long process arrives early, all other processes must wait.
- It may increase Waiting Time (WT) and Turnaround Time (TAT).
- If no process has arrived yet, CPU becomes idle.

Problem:

FCFS suffers from the Convoy Effect:

- A long job delays many short jobs.
- Short processes experience large waiting times.

Example:

If P1 (BT=10) arrives at time 0
and P2 (BT=1) arrives at time 1
→ P2 must wait 9 units even though it needs only 1 unit.

So, arrival time directly affects fairness and performance in FCFS.

2. Why can Priority scheduling cause starvation?

In Priority Scheduling:

- Process with higher priority (lower number) executes first.

Starvation occurs when:

- Low priority processes keep waiting.
- New high-priority processes keep arriving.

As a result:

- Low priority processes may never get CPU time.
- They remain in ready queue indefinitely.

Example:

If low priority process P3 is waiting,
but high priority processes P1, P2, P4 keep arriving,
→ P3 may never execute.

Solution:

To prevent starvation:

- Use Aging Technique
→ Gradually increase priority of waiting processes.

3. Compare Preemptive and Non-Preemptive Priority scheduling.

Feature	Non-Preemptive	Preemptive
CPU interruption	No	Yes
Response time	Slower	Faster
Context switching	Less	More
Complexity	Simple	More complex
Suitable for	Batch systems	Real-time systems

Non-Preemptive:

- Once a process starts, it runs till completion.
- Easier to implement.
- May cause long waiting time for high-priority process.

Preemptive:

- If higher priority process arrives → CPU is interrupted.
- Better responsiveness.
- More context switching overhead.

4. Which scheduling policy is more suitable for cloud systems and why?

In cloud systems:

- Multiple users
- Dynamic workloads
- Real-time requests

- Fair resource allocation needed

Most Suitable: Preemptive Priority Scheduling (or Dynamic Scheduling)

Why?

Better response time

Handles real-time tasks

Can prioritize important jobs

Suitable for dynamic workloads

Supports SLA (Service Level Agreement)

However, in real cloud systems:

- Advanced schedulers like Weighted Fair Scheduling, Round Robin, and Multi-Level Queue Scheduling are commonly used.
- Cloud platforms use hybrid and adaptive scheduling algorithms.