K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: C2	**Roll No.: 16010122323**
Experiment No. 04	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of Basic Process management algorithms – Non Pre-emptive (FCFS , SJF, priority)

AIM: To implement basic Non –Pre-emptive Process management algorithms (FCFS , SJF , Priority)

Expected Outcome of Experiment:

CO 2. To understand the concept of process, thread and resource management.

Books/ Journals/ Websites referred:

1. Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.
2. Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.
3. William Stallings, "Operating System Internal & Design Principles", Pearson.
4. Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.

Pre Lab/ Prior Concepts:

Most systems handle numerous processes with short CPU bursts interspersed with I/O requests and a few processes with long CPU bursts. To ensure good time-sharing performance, a running process may be preempted to allow another to run. The ready list, or run queue, maintains all processes ready to run and not blocked by I/O or other system requests. Entries in this list point to the process control block, which stores all process information and state.

When an I/O request completes, the process moves from the waiting state to the ready state and is placed on the run queue. The process scheduler, a key component of the operating system, decides whether the current process should continue running or if another should take over. This decision is triggered by four events:

1. The current process issues an I/O request or system request, moving it from running to waiting.
2. The current process terminates.
3. A timer interrupt indicates the process has run for its allotted time, moving it from running to ready.
4. An I/O operation completes, moving the process from waiting to ready, potentially preempting the current process.

The scheduling algorithm, or policy, determines the sequence and duration of process execution, a complex task given the limited information about ready processes.

Description of the application to be implemented:

First-Come, First-Served Scheduling:

First-Come, First-Served (FCFS) scheduling is a process scheduling algorithm that executes tasks in the order they arrive. It operates on a simple principle: the first process to enter the queue is the first one to be executed. There is no preemption; once a process starts executing, it runs to completion before the next process in the queue begins. This method ensures that all processes are treated equally but can lead to inefficiencies such as the "convoy effect," where shorter processes wait behind longer ones, potentially increasing overall waiting time and turnaround time.

Shortest job first :

Shortest Job First (SJF) is a scheduling algorithm that selects the process with the smallest execution time (or burst time) for execution next. It operates under the principle of minimizing average waiting time by prioritizing shorter jobs over longer ones. There are two variations of SJF:

1. **Non-Preemptive SJF**: Once a process starts executing, it runs to completion before the next process is selected. In this version, the scheduler chooses the

process with the shortest burst time from the list of processes that have arrived and are waiting in the queue.

2. **Preemptive SJF** (also known as Shortest Remaining Time First, SRTF): The scheduler can interrupt a currently running process if a new process arrives with a shorter remaining burst time. This means that the currently executing process may be paused and moved back to the queue if a shorter job arrives.

SJF aims to reduce the average waiting time and turnaround time by minimizing the time processes spend waiting in the queue. However, it requires knowledge of the future burst times, which can be challenging to predict.

Priority scheduling

Priority Scheduling is a scheduling algorithm that selects processes based on their priority levels. Each process is assigned a priority, and the scheduler picks the process with the highest priority for execution next.

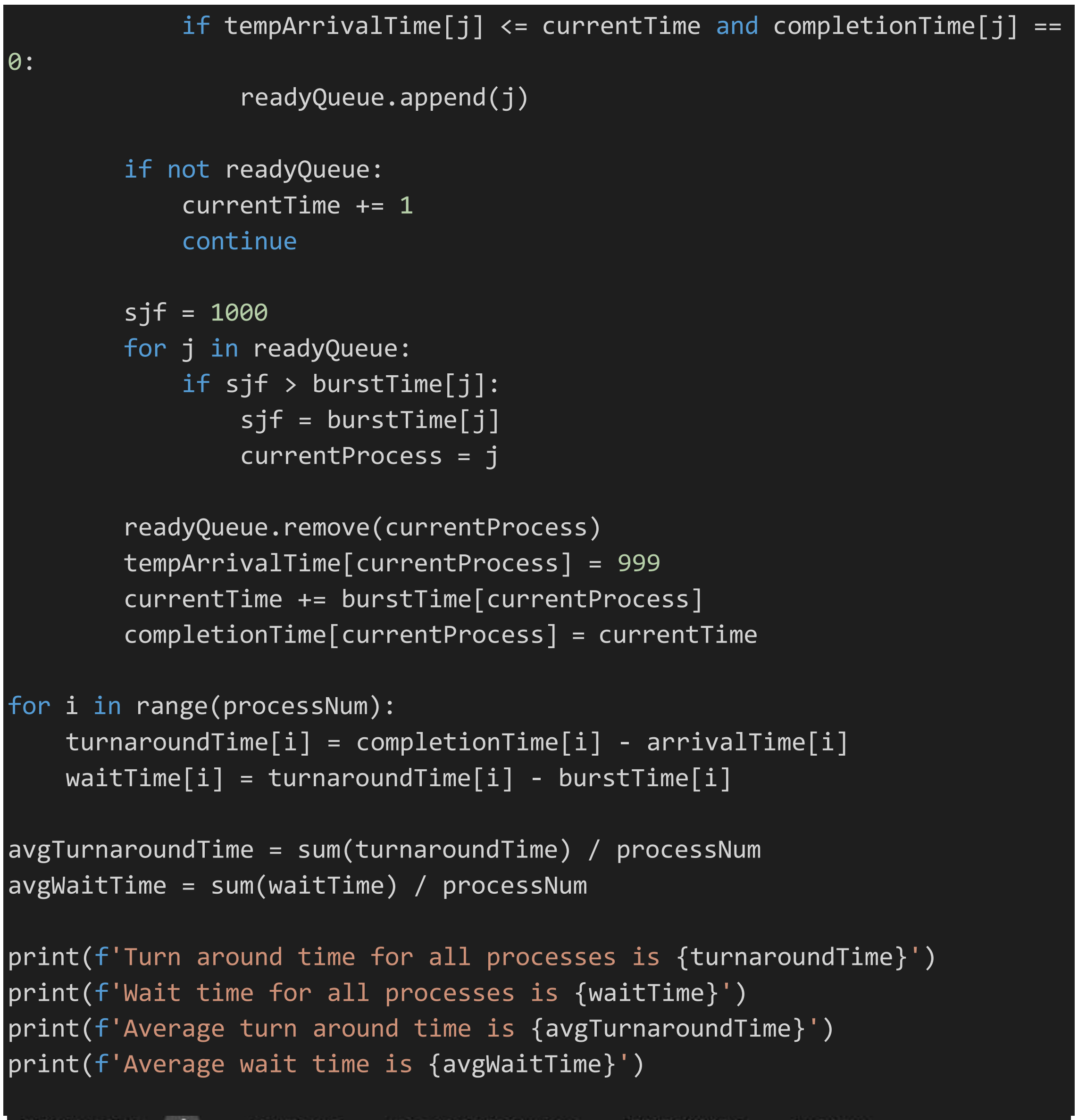
Here are key aspects of Priority Scheduling:

1. **Priority Levels**: Processes are assigned numerical priority values. The convention can vary: a higher number might indicate higher priority, or vice versa.
2. **Preemptive vs. Non-Preemptive**:
   - **Preemptive Priority Scheduling**: If a new process arrives with a higher priority than the currently running process, the current process is paused and the higher-priority process is executed immediately.
   - **Non-Preemptive Priority Scheduling**: Once a process starts executing, it runs to completion before any other process with a higher priority is considered for execution.
3. **Starvation**: Lower-priority processes may suffer from starvation, where they are perpetually preempted by higher-priority processes, leading to indefinite postponement.
4. **Dynamic Priorities**: Some implementations adjust process priorities dynamically based on certain factors, such as aging, to mitigate starvation.

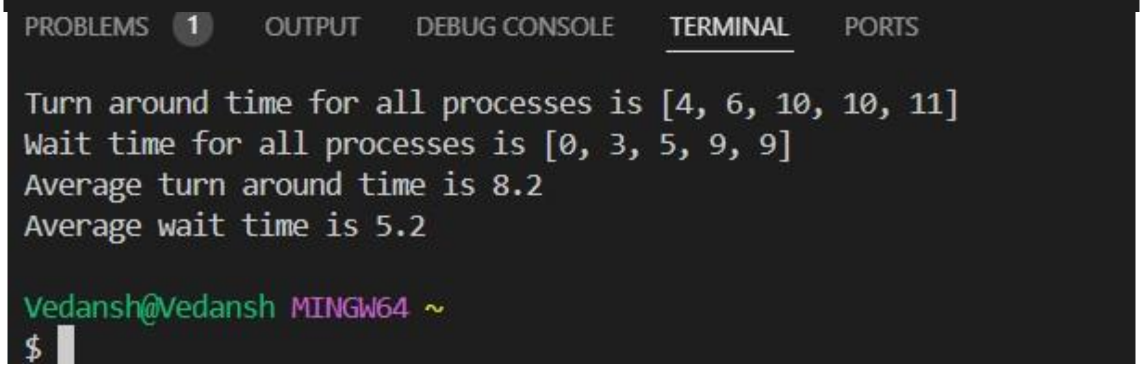
Priority Scheduling aims to ensure that important processes receive CPU time more quickly, but it can lead to unfairness if high-priority processes consistently dominate the queue.

Implementation details:

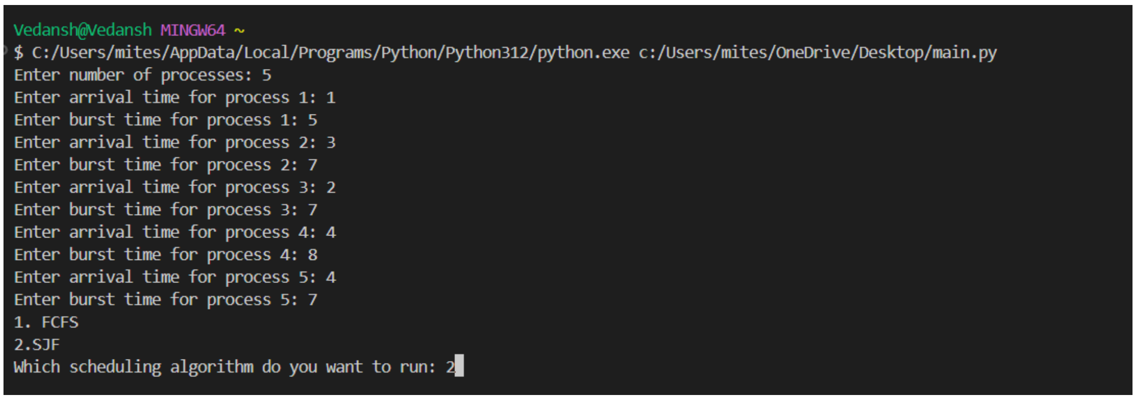
```
import os

processNum = int(input("Enter number of processes: "))
currentTime = 0
arrivalTime = []
burstTime = []
completionTime = [0] * processNum
turnaroundTime = [0] * processNum
waitTime = [0] * processNum

for i in range(processNum):
    arrivalTime.append(int(input(f'Enter arrival time for process {i+1}:
')))
    burstTime.append(int(input(f'Enter burst time for process {i+1}: ')))

print(f'1. FCFS\n2.SJF')
choice = int(input(f'Which scheduling algorithm do you want to run: '))
os.system('cls')

if choice == 1:
    tempArrivalTime = arrivalTime.copy()
    if min(tempArrivalTime) != 0:
        currentTime += min(tempArrivalTime)

    for i in range(processNum):
        currentProcess = tempArrivalTime.index(min(tempArrivalTime))
        tempArrivalTime[currentProcess] = 999
        currentTime += burstTime[currentProcess]
        completionTime[currentProcess] = currentTime

elif choice == 2:
    tempArrivalTime = arrivalTime.copy()
    readyQueue = []

    if min(tempArrivalTime) != 0:
        currentTime += min(tempArrivalTime)

    while 0 in completionTime:
        readyQueue = []

        for j in range(processNum):
```



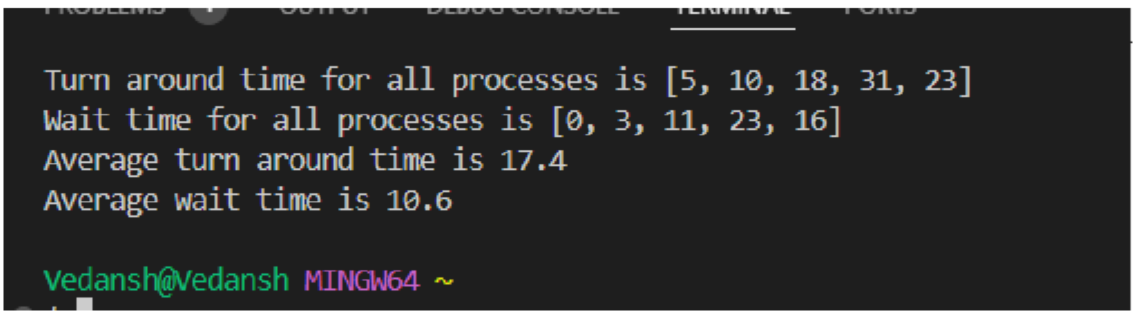
```
            if tempArrivalTime[j] <= currentTime and completionTime[j] ==
0:
                readyQueue.append(j)

        if not readyQueue:
            currentTime += 1
            continue

        sjf = 1000
        for j in readyQueue:
            if sjf > burstTime[j]:
                sjf = burstTime[j]
                currentProcess = j

        readyQueue.remove(currentProcess)
        tempArrivalTime[currentProcess] = 999
        currentTime += burstTime[currentProcess]
        completionTime[currentProcess] = currentTime

for i in range(processNum):
    turnaroundTime[i] = completionTime[i] - arrivalTime[i]
    waitTime[i] = turnaroundTime[i] - burstTime[i]

avgTurnaroundTime = sum(turnaroundTime) / processNum
avgWaitTime = sum(waitTime) / processNum

print(f'Turn around time for all processes is {turnaroundTime}')
print(f'Wait time for all processes is {waitTime}')
print(f'Average turn around time is {avgTurnaroundTime}')
print(f'Average wait time is {avgWaitTime}')
```



```
Turn around time for all processes is [4, 6, 10, 10, 11]
Wait time for all processes is [0, 3, 5, 9, 9]
Average turn around time is 8.2
Average wait time is 5.2

Vedansh@Vedansh MINGW64 ~
$
```



```
Vedansh@Vedansh MINGW64 ~
$ C:/Users/mites/AppData/Local/Programs/Python/Python312/python.exe c:/Users/mites/OneDrive/Desktop/main.py
Enter number of processes: 5
Enter arrival time for process 1: 1
Enter burst time for process 1: 5
Enter arrival time for process 2: 3
Enter burst time for process 2: 7
Enter arrival time for process 3: 2
Enter burst time for process 3: 7
Enter arrival time for process 4: 4
Enter burst time for process 4: 8
Enter arrival time for process 5: 4
Enter burst time for process 5: 7
1. FCFS
2.SJF
Which scheduling algorithm do you want to run: 2
```



```
Turn around time for all processes is [5, 10, 18, 31, 23]
Wait time for all processes is [0, 3, 11, 23, 16]
Average turn around time is 17.4
Average wait time is 10.6

Vedansh@Vedansh MINGW64 ~
```

Conclusion: Understood and implemented basic Non –Pre-emptive Process management algorithms (FCFS , SJF , Priority)

Post Lab Questions

1. What is a criterion to evaluate a scheduling algorithm?

When evaluating a scheduling algorithm, one common criterion is **turnaround time.** This is the total time taken from the arrival of a process to its completion, including the waiting time and execution time. Other important criteria include:

- **Waiting Time:** The total time a process spends waiting in the ready queue before it gets CPU time.
- **Response Time:** The time from the arrival of a request until the first response is produced.
- **Throughput:** The number of processes completed per unit time.
- **CPU Utilization:** The percentage of time the CPU is actively working.
- **Fairness:** How evenly the algorithm allocates CPU time among processes.
- **Starvation:** Whether any process is left waiting indefinitely.

2. Analyse the efficiency and suitability of FCFS, SJF, and Priority scheduling algorithms.

First-Come, First-Served (FCFS):

- **Efficiency:** Simple and easy to implement. However, it may not always be the most efficient in terms of turnaround time and waiting time.
- **Suitability:** Best for batch processing where simplicity is more critical than performance. It can lead to the "convoy effect," where short processes wait for a long process to complete.

Shortest Job First (SJF):

- **Efficiency:** Can be more efficient than FCFS because it minimizes the average waiting time. Short processes are executed before longer ones, which generally reduces turnaround time.
- **Suitability:** Ideal for systems where job lengths are known ahead of time. It is not feasible for interactive systems where job lengths are unknown.

Priority Scheduling:

- **Efficiency:** Can be very efficient if the priorities are assigned effectively, and it balances different types of processes based on their importance.
- **Suitability:** Suitable for real-time systems where certain tasks need to be prioritized. However, it may lead to **starvation** of lower-priority processes if higher-priority processes keep arriving.

3. A brief explanation of the concept of "starvation" in SJF scheduling and how to avoid it.

Starvation occurs in SJF (Shortest Job First) scheduling when a process with a longer burst time may continually be preempted by shorter jobs, causing it to wait indefinitely. This happens because SJF always selects the shortest available job, so longer jobs may never get the CPU if shorter ones keep arriving.

How to Avoid Starvation:

- **Aging:** Gradually increase the priority of a process the longer it waits. This makes sure that eventually, even long processes will get a chance to execute.
- **Fair Scheduling:** Combine SJF with other scheduling policies that ensure that all processes get a fair chance to execute, such as a round-robin approach in conjunction with SJF.
- **Dynamic Estimation:** Use dynamic estimates of job length that update as processes execute, which can help balance short and long jobs more effectively.

Each of these strategies aims to ensure that no process is indefinitely delayed and helps maintain a more balanced scheduling system.

Date: **Signature of faculty in-charge**