S.B. JAIN INSTITUTE OF TECHNOLOGY MANAGEMENT & RESEARCH, NAGPUR

Practical 05

Aim: Write a program to implement Shortest Job First (SJF) Preemptive Scheduling for three processes and calculate the total context switches and average waiting time. The processes have burst times 10ns, 20ns, and 30ns, arriving at 0ns, 2ns, and 6ns, respectively.

Name: DARSHAK K. BISANE

USN: CM25D004

Semester / Year: IV Semester / II Year

Academic Session: 2025 – 2026

Date of Performance: 10 / 02 / 26

Date of Submission: 17 / 02 / 26

❖ **Aim:** Write a program to implement Shortest Job First (SJF) Preemptive Scheduling for three processes and calculate the total context switches and average waiting time. The processes have burst times 10ns, 20ns, and 30ns, arriving at 0ns, 2ns, and 6ns, respectively.

❖ **Objectives:**

Understand SJF Preemptive Scheduling: Implement the **Shortest Job First (SJF) Preemptive Scheduling** algorithm to manage CPU execution efficiently.

Calculate Context Switches: Determine the total number of context switches required for the given set of processes.

Evaluate Waiting Time: Compute the **average waiting time** for all processes before getting CPU execution.

❖ **Requirements:**

✓ **Hardware Requirements:**

- Processor: Minimum 1 GHz
- RAM: 512 MB or higher
- Storage: 100 MB free space

✓ **Software Requirements:**

- Operating System: Linux/Unix-based
- Shell: Bash 4.0 or higher
- Text Editor: Nano, Vim, or any preferred editor

❖ **Theory:**

CPU Scheduling in Operating Systems

Introduction

Scheduling is the method by which processes are given access to the CPU. Efficient scheduling is essential for optimal system performance and user experience. There are two primary types of CPU scheduling: **Preemptive Scheduling** and **Non-Preemptive Scheduling**.

Understanding the differences between these scheduling types helps in designing and choosing the right scheduling algorithms for different operating systems.

1. Preemptive Scheduling

In **Preemptive Scheduling**, the operating system can interrupt or preempt a running process to allocate CPU time to another process, typically based on priority or time-sharing policies. A process can be switched from the **running state to the ready state** at any time.

Algorithms Based on Preemptive Scheduling:

- **Round Robin (RR)**
- **Shortest Remaining Time First (SRTF)**
- **Priority Scheduling (Preemptive version)**

Example:

In the following case, **P2 is preempted at time 1** due to the arrival of a higher-priority process.

Advantages of Preemptive Scheduling:

✔ Prevents a process from monopolizing the CPU, improving system reliability.

✔ Enhances **average response time**, making it beneficial for multi-programming environments.

✔ Used in modern operating systems like **Windows, Linux, and macOS**.

Disadvantages of Preemptive Scheduling:

✗ More complex to implement.

✗ Involves **overhead** for suspending a running process and switching contexts.

✗ **May cause starvation** if low-priority processes are frequently preempted.

✗ Can create **concurrency issues**, especially when accessing shared resources.

2. **Non-Preemptive Scheduling**

In **Non-Preemptive Scheduling**, a running process cannot be interrupted by the operating system. It continues executing until it **terminates** or **enters a waiting state** voluntarily.

Algorithms Based on Non-Preemptive Scheduling:

- **First Come First Serve (FCFS)**
- **Shortest Job First (SJF - Non-Preemptive)**
- **Priority Scheduling (Non-Preemptive version)**

Example:

Below is a **Gantt Chart** based on the **FCFS algorithm**, where each process executes fully before the next one starts.

Advantages of Non-Preemptive Scheduling:

✔ **Easy to implement** in an operating system (used in older versions like Windows 3.11 and early macOS).

✔ **Minimal scheduling overhead** due to fewer context switches.

✔ **Less computational resource usage**, making it more efficient for simpler systems.

Disadvantages of Non-Preemptive Scheduling:

✗ **Risk of Denial of Service (DoS) attacks**, as a process can monopolize the CPU.

✗ **Poor response time**, especially in multi-user systems.

3. **Differences Between Preemptive and Non-Preemptive Scheduling**

Parameter	Preemptive Scheduling	Non-Preemptive Scheduling
Basic Concept	CPU time is allocated for a **limited time**.	CPU is held until process **terminates** or enters waiting state.
Interrupts	Process **can be interrupted**.	Process **cannot be interrupted**.
Starvation	Frequent high-priority processes may starve low-priority ones.	A long-running process can starve later-arriving shorter processes.
Overhead	Higher overhead due to frequent **context switching**.	Minimal overhead.
Flexibility	More flexible (critical processes get priority).	Rigid scheduling approach.
Response Time	Faster response time.	Slower response time.
Process Control	**OS has more control** over scheduling.	OS has **less control** over scheduling.
Concurrency Issues	**Higher**, as processes may be preempted during shared resource access.	**Lower**, as processes run to completion.
Examples	Round Robin, SRTF.	FCFS, Non-Preemptive SJF.

4. Frequently Asked Questions (FAQs)

a. How is priority determined in Preemptive scheduling?

Ans: Preemptive scheduling systems assign priority based on **task importance, deadlines, or urgency**. Higher-priority tasks execute before lower-priority ones.

b. What happens in non-preemptive scheduling if a process does not yield the CPU?

Ans: If a process does not voluntarily yield the CPU, it can lead to **starvation or deadlock**, where other tasks are unable to execute.

c. Which scheduling method is better for real-time systems?

Ans: Preemptive scheduling is better for **real-time systems**, as it allows high-priority tasks to execute immediately.

❖ CODE

```
#include <stdio.h>
#define MAX_PROCESSES 4
// Process structure to hold the process details
typedef struct {
   int pid;           // Process ID
   int arrival_time;   // Arrival time
   int burst_time;     // Total burst time
   int remaining_time; // Remaining burst time (for preemption)
} Process;
// Function to calculate waiting time for each process
void calculate_waiting_time(Process processes[], int n, int completion_time[], int waiting_time[]) {
   for (int i = 0; i < n; i++) {
      waiting_time[i] = completion_time[i] - processes[i].arrival_time - processes[i].burst_time;
   }
}
// Function to calculate turnaround time for each process
void calculate_turnaround_time(Process processes[], int n, int completion_time[], int
     turnaround_time[]) {
   for (int i = 0; i < n; i++) {
      turnaround_time[i] = completion_time[i] - processes[i].arrival_time;
   }
}
// Function to find the process with the shortest remaining time
int find_shortest_process(Process processes[], int n, int current_time) {
   int min_time = 999999;  // Arbitrary large value to start with
   int min_index = -1;

   for (int i = 0; i < n; i++) {
      // Process should have arrived and not yet completed
      if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0) {
         if (processes[i].remaining_time < min_time) {
            min_time = processes[i].remaining_time;
            min_index = i;
         }
      }
   }
   return min_index; // Return the index of the process with the shortest remaining time
}
```

```
int main() {
    // Initializing processes with arrival time clashes and different burst times
    Process processes[MAX_PROCESSES] = {
        {1, 0, 10, 10},  // Process 1: Arrival 0ns, Burst 10ns
        {2, 2, 20, 20},  // Process 2: Arrival 2ns, Burst 20ns
        {3, 6, 30, 30},  // Process 3: Arrival 6ns, Burst 30ns
        {4, 2, 15, 15}   // Process 4: Arrival 2ns, Burst 15ns (Arrival clash with Process 2)
    };

    int current_time = 0; // Start from 0ns
    int completed = 0;    // To keep track of how many processes are completed
    int context_switches = 0; // To count the number of context switches

    // Arrays to store waiting time and turnaround time for each process
    int waiting_time[MAX_PROCESSES];
    int turnaround_time[MAX_PROCESSES];
    int completion_time[MAX_PROCESSES]; // To store when each process completes

    // Start the scheduling loop
    while (completed < MAX_PROCESSES) {
        printf("Time: %d ns\n", current_time);

        // Find the process with the shortest remaining time
        int process_index = find_shortest_process(processes, MAX_PROCESSES, current_time);

        // If there's a valid process to run
        if (process_index != -1) {
            Process *current_process = &processes[process_index];

            // Print the process currently selected for execution
            printf("Running Process %d (Remaining Time: %d ns, Burst Time: %d ns)\n",
                   current_process->pid, current_process->remaining_time, current_process->burst_time);

            // Execute the process for one unit of time
            current_process->remaining_time--;

            // If process is finished, record the completion time
            if (current_process->remaining_time == 0) {
                completed++;
                completion_time[process_index] = current_time + 1; // Process completes at current_time + 1

                // Count a context switch when a process is completed
                if (completed > 1) {
                    context_switches++;
```

```
            }

            // Print the completion of the process
            printf("Process %d completed at time %d ns\n", current_process->pid, current_time + 1);
        }
    }

    // Increment time unit
    current_time++;

    // Print the comparison of remaining times and burst times at each step
    printf("Remaining Times and Burst Times at time %d ns: ", current_time);
    for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("P%d: Rem=%d Burst=%d ", processes[i].pid, processes[i].remaining_time,
   processes[i].burst_time);
    }
    printf("\n\n");
}

// Calculate waiting time and turnaround time
calculate_waiting_time(processes, MAX_PROCESSES, completion_time, waiting_time);
calculate_turnaround_time(processes, MAX_PROCESSES, completion_time, turnaround_time);

// Display the results
printf("Total Context Switches: %d\n", context_switches);
printf("Process-wise Waiting and Turnaround Times:\n");
for (int i = 0; i < MAX_PROCESSES; i++) {
    printf("Process %d -> Waiting Time: %d ns, Turnaround Time: %d ns\n",
           processes[i].pid, waiting_time[i], turnaround_time[i]);
}

// Calculate and display the average waiting time
int total_waiting_time = 0;
for (int i = 0; i < MAX_PROCESSES; i++) {
    total_waiting_time += waiting_time[i];
}
printf("Average Waiting Time: %.2f ns\n", (float)total_waiting_time / MAX_PROCESSES);

return 0;
  }
```

❖ Output:

```
========================================
   SJF PREEMPTIVE SCHEDULING ALGORITHM
========================================

INPUT PROCESSES:
PID     Arrival Burst
------------------
P1      0       10
P2      2       20
P3      6       30

EXECUTION SEQUENCE (Gantt Chart):
Time    Process Remaining       Comparison Log
----    ------- ---------       --------------
0       P1      10
1       P1      9
2       P1      8               P2 (RT=20) : P1 (RT=8) : Continue P1
3       P1      7
4       P1      6
5       P1      5
6       P1      4               P3 (RT=30) : P1 (RT=4) : Continue P1
7       P1      3
8       P1      2
9       P1      1
10      P2      20
11      P2      19
12      P2      18
13      P2      17
14      P2      16
15      P2      15
16      P2      14
17      P2      13
18      P2      12
19      P2      11
20      P2      10
21      P2      9
22      P2      8
23      P2      7
24      P2      6
25      P2      5
26      P2      4
27      P2      3
28      P2      2
29      P2      1
30      P3      30
31      P3      29
32      P3      28
33      P3      27
34      P3      26
35      P3      25
36      P3      24
37      P3      23
38      P3      22
39      P3      21
40      P3      20
41      P3      19
42      P3      18
43      P3      17
44      P3      16

45      P3      15
46      P3      14
47      P3      13
48      P3      12
49      P3      11
50      P3      10
51      P3      9
52      P3      8
53      P3      7
54      P3      6
55      P3      5
56      P3      4
57      P3      3
58      P3      2
59      P3      1

========================================
          RESULTS & CALCULATIONS
========================================

PROCESS EXECUTION DETAILS:
PID     AT      BT      CT      TAT     WT
-----------------------------------------
P1      0       10      10      10      0
P2      2       20      30      28      8
P3      6       30      60      54      24

PERFORMANCE METRICS:
-----------------------------------------
Total Context Switches: 2
Total Waiting Time: 32 ns
Total Turnaround Time: 92 ns
Average Waiting Time: 10.67 ns
Average Turnaround Time: 30.67 ns

========================================
       VERIFICATION CALCULATIONS
========================================

Manual Calculations:
P1: CT = 10, TAT = 10-0 = 10, WT = 10-10 = 0
P2: CT = 30, TAT = 30-2 = 28, WT = 28-20 = 8
P3: CT = 60, TAT = 60-6 = 54, WT = 54-30 = 24
Avg WT = (0+8+24)/3 = 32/3 = 10.67 ns

--------------------------------
Process exited after 0.03574 seconds with return value 0
```

Conclusion: Preemptive scheduling offers better responsiveness but adds complexity, while non-preemptive scheduling is simpler but may cause inefficiencies. The choice depends on system needs, with preemptive suited for multitasking and non-preemptive for low-overhead scenarios.

❖ **Discussion Questions:**

1. **What is the key difference between preemptive and non-preemptive scheduling?**
2. **Why does preemptive scheduling require context switching?**
3. **Which CPU scheduling algorithm is most suitable for real-time systems and why?**
4. **What is starvation in CPU scheduling, and how can it be prevented?**
5. **Why is the Round Robin scheduling algorithm preferred in time-sharing systems?**

❖ **References:**

https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/

Date: 17 / 02 /2026

Signature
Course Coordinator
B.Tech CSE(AIML)
Sem: 4 / 2025-26