EXPERIMENT NO: 4 a **DATE –**

Pre-emptive CPU Scheduling Algorithm

(a) Shortest Remaining Time First Scheduling Algorithm

Aim-To implement Shortest remaining time first CPU scheduling algorithm

THEORY –

Preemptive CPU Scheduling Algorithms

Preemptive CPU scheduling is a type of process scheduling in which the **operating system can take back (preempt) the CPU from a running process** and assign it to another process that is ready to run. This ensures that higher priority or more urgent processes are not delayed by longer-running tasks.

How it works

1. A process is running on the CPU.
2. If another process arrives with **higher priority** or **shorter burst time** (depending on the algorithm), the CPU is **taken away** from the current process.
3. The preempted process is placed back into the **ready queue**, and it resumes later when it again gets the CPU.
4. This allows better **responsiveness** and is widely used in **real-time and interactive systems**.
5. The key advantage of preemptive scheduling is that it improves **turnaround time** for short processes and ensures fairness, but it also introduces **overhead** due to frequent context switching.

Shortest Remaining Time First (SRTF) – Theory

The **Shortest Remaining Time First (SRTF)** is the **preemptive version** of the Shortest Job First (SJF) scheduling algorithm.At any given time, the CPU is allocated to the

process that has the **shortest remaining burst time** among all the processes in the ready queue.If a new process arrives with a shorter remaining time than the currently running process, the CPU is **preempted** and given to the new process.This ensures that shorter processes finish quickly, reducing the **average waiting time** and **average turnaround time**.

Steps in SRTF

Keep track of the remaining burst times of all processes.

1.At each unit of time, check if a new process arrives.

2.Compare burst times of the current running process and the new process.

3.Preempt the running process if the new process has a shorter remaining burst time.

4.No preemption occurs—once a process starts, it will run until completion.

- **Performance Metrics**:
    - **Completion Time (CT):** The time at which a process finishes execution.
    - **Turnaround Time (TAT):** The total time spent in the system (from arrival to completion).

TAT=CT−ATAT

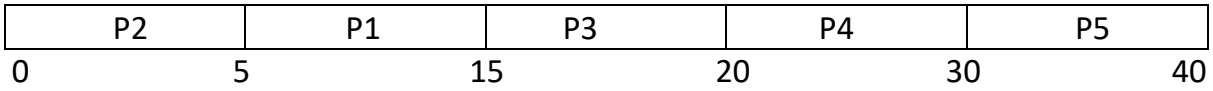
    - **Waiting Time (WT):** The time a process spends waiting in the ready queue.

WT=TAT−BT

EXAMPLE –

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	10
P2	0	5
P3	5	10
P4	15	5
P5	18	10

GANTT diagram



PROCESS	AT	BURST TIME	CT	TURN AROUND TIME (CT-ATAT)	WAIT TIME (TAT-BT)
P1	0	10	15	15	5
P2	0	5	5	5	0
P3	5	10	30	25	15
P4	15	5	20	5	0
P5	18	10	40	22	12

AVERAGE TURN AROUND TIME = 14.4

AVERAGE WAIT TIME = 6.4

PROGRAM –

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct Process {
    string name;
    int arrival, burst, remaining, start, completion, turnaround, waiting;
    bool started;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> p(n);
    for (int i = 0; i < n; i++) {
        cout << "\nEnter Process Name, Arrival Time, Burst Time for P" << i + 1 << ": ";
        cin >> p[i].name >> p[i].arrival >> p[i].burst;
        p[i].remaining = p[i].burst;
        p[i].started = false;
    }

    vector<bool> done(n, false);
    int time = 0, completed = 0;
    float avgTAT = 0, avgWT = 0;
    int totalTAT = 0;
    vector<string> ganttOrder;
    vector<int> ganttTime;
    string prevProcess = "";

    while (completed < n) {
        int idx = -1, minRemaining = 1e9;
        for (int i = 0; i < n; i++) {
            if (!done[i] && p[i].arrival <= time && p[i].remaining < minRemaining) {
                minRemaining = p[i].remaining;
                idx = i;
            }
        }

        if (idx == -1) {
            // No process available, add idle time
            int nextArrival = 1e9;
            for (int i = 0; i < n; i++) {
                if (!done[i] && p[i].arrival < nextArrival) {
                    nextArrival = p[i].arrival;
                }
            }
            if (prevProcess != "Idle") {
                ganttOrder.push_back("Idle");
                ganttTime.push_back(time);
                prevProcess = "Idle";
            }
            time = nextArrival;
        } else {
            if (!p[idx].started) {
                p[idx].start = time;
                p[idx].started = true;
            }

            // If current process is different from previous, update Gantt chart
            if (prevProcess != p[idx].name) {
                ganttOrder.push_back(p[idx].name);
```

```
            ganttTime.push_back(time);
            prevProcess = p[idx].name;
        }

        // Execute for 1 time unit
        time++;
        p[idx].remaining--;

        if (p[idx].remaining == 0) {
            p[idx].completion = time;
            p[idx].turnaround = p[idx].completion - p[idx].arrival;
            p[idx].waiting = p[idx].turnaround - p[idx].burst;
            avgTAT += p[idx].turnaround;
            avgWT += p[idx].waiting;
            totalTAT += p[idx].turnaround;
            done[idx] = true;
            completed++;
        }
    }

    // Add final time to Gantt chart
    ganttTime.push_back(time);

    cout << "\n----------------------------------------\n";
    cout << "Process\tAT\tBT\tCT\tTAT\tWT\n";
    cout << "----------------------------------------\n";
    for (int i = 0; i < n; i++) {
        cout << p[i].name << "\t" << p[i].arrival << "\t" << p[i].burst << "\t"
             << p[i].completion << "\t" << p[i].turnaround << "\t" << p[i].waiting << "\n";
    }
    cout << "----------------------------------------\n";
    cout << "Total Turnaround Time = " << totalTAT << endl;
    cout << "Average Turnaround Time = " << avgTAT / n << endl;
    cout << "Average Waiting Time = " << avgWT / n << endl;

    cout << "\nGantt Chart:\n";
    cout << "----------------------------------------\n";
    for (size_t i = 0; i < ganttOrder.size(); i++) {
        cout << "| " << setw(5) << ganttOrder[i] << " ";
    }
    cout << "|\n";
    cout << "----------------------------------------\n";

    // Print timeline for Gantt chart
    cout << setw(2) << ganttTime[0];
    for (size_t i = 1; i < ganttTime.size(); i++) {
        cout << setw(8) << ganttTime[i];
    }
    cout << "\n";

    return 0;
}
```

OUTPUT –

```
Enter number of processes: 5

Enter Process Name, Arrival Time, Burst Time for P1: P1
0
4

Enter Process Name, Arrival Time, Burst Time for P2: P2
3
7

Enter Process Name, Arrival Time, Burst Time for P3: P3
5
5

Enter Process Name, Arrival Time, Burst Time for P4: P4
7
6

Enter Process Name, Arrival Time, Burst Time for P5: P5
9
3

-----------------------------------------------------------------
Process AT      BT      CT      TAT     WT
-----------------------------------------------------------------
P1      0       4       4       4       0
P2      3       7       19      16      9
P3      5       5       10      5       0
P4      7       6       25      18      12
P5      9       3       13      4       1
-----------------------------------------------------------------
Total Turnaround Time = 47
Average Turnaround Time = 9.4
Average Waiting Time = 4.4

Gantt Chart:
-----------------------------------------------------------------
|    P1 |    P2 |    P3 |    P5 |    P2 |    P4 |
-----------------------------------------------------------------
 0       4       5       10      13      19      25
```

CONCLUSION -

The Shortest Remaining Time First (SRTF) algorithm optimizes the average waiting and turnaround times by always selecting the process with the least remaining time. Its preemptive nature makes it more efficient than non-preemptive SJF for dynamic arrivals. However, frequent **context switching** increases overhead, and starvation of longer processes may occur if many short processes keep arriving.