CPU Scheduling Algorithms: Implementation and Performance Analysis

Objectives

- Design and implement multiple CPU scheduling strategies (FCFS, SJF, Priority, and Round Robin with variable time quantum) and simulate their operation.
- Visualize process execution through Gantt charts and record performance results in tables for comparison.
- Evaluate scheduling performance by calculating Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT), both per process and on average.
- Compare algorithms in terms of efficiency and fairness, highlighting trade-offs and the effect of Round Robin's time quantum.
- Present the study through practical implementations, code, and a detailed lab report with analysis and conclusions.

Objectives

- Implement and simulate multiple CPU scheduling algorithms (FCFS, SJF, SRTF, Round Robin, Priority) on a given process workload.
- Visualize process scheduling (e.g., with Gantt charts) and collect performance data.
- Calculate each process's Turnaround Time, Waiting Time, and Response Time, and compute average values for each algorithm.
- Compare the scheduling algorithms in terms of efficiency (e.g., minimizing wait time) and fairness.
- Document methods, results, and conclusions in a clear lab report.
-

Introduction

CPU scheduling is a fundamental concept in operating systems where the CPU chooses which process to run next. Different scheduling algorithms (like FCFS, SJF, SRTF, Round Robin, and Priority) use different rules to decide the order of process execution. In this lab, we implement and analyze these algorithms by simulating the execution of a set of processes with specified arrival times, burst times, and priorities. We generate Gantt charts to visualize the execution order and measure performance metrics (waiting time, turnaround time, and response time) to compare how each algorithm performs in terms of efficiency and fairness.

Theoretical Background

CPU scheduling algorithms aim to optimize processor utilization and process performance. Key performance metrics include:

- **Waiting Time (WT):** The time a process spends waiting in the ready queue before execution.
- **Turnaround Time (TAT):** The total time from process arrival to completion (TAT = Completion Time – Arrival Time).
- **Response Time (RT):** The time from process arrival until it first starts execution.

Algorithms may be **non-preemptive** (a running process is not interrupted) or **preemptive** (the running process can be interrupted). For example, FCFS and non-preemptive SJF run processes to completion once they start. SRTF (preemptive SJF), Round Robin, and preemptive Priority scheduling can interrupt the current process to run another. Each approach has trade-offs: FCFS is simple and fair by arrival order but can suffer if a long job goes first. SJF/SRTF minimize average waiting time by running short tasks first, but they can starve longer processes. Round Robin ensures no process waits too long by giving fixed time slices to all processes, at the expense of higher context switching if the quantum is small. Priority scheduling lets important processes run first, but low-priority processes might wait indefinitely if higher-priority jobs keep coming. Understanding these differences helps evaluate algorithm performance.

Methodology

We test the scheduling algorithms using a fixed set of five processes. Each process has an assigned arrival time, burst time, and priority. The dataset is shown in Table 1.

Process	Arrival Time	Burst Time	Priority
P1	0	6	2
P2	1	8	1
P3	2	7	3
P4	3	3	4
P5	4	5	2

We implement each scheduling algorithm in a programming environment (for example, c coding). We simulate process arrivals and execution according to each algorithm’s rules. For each run, we record the start time and completion time of each process, then compute Turnaround Time, Waiting Time, and Response Time for each process and their averages. Gantt charts are drawn to illustrate the schedule of processes under each algorithm.

Algorithm Implementation

First Come First Serve (FCFS)

First-Come, First-Served (FCFS) schedules processes in the order they arrive. It is non-preemptive: once a process starts, it runs to completion. We sort processes by arrival time and execute each in turn. FCFS is simple but can cause later processes to wait longer if an earlier process has a long burst time.

```c
#include <stdio.h>
struct Process {
    char pid[10];
    int at, bt, st, ct, tat, wt, rt;
};

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i = 0; i < n; i++) {
        printf("\nEnter Process ID for P%d: ", i + 1);
        scanf("%s", p[i].pid);
        printf("Enter Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Enter Burst Time: ");
        scanf("%d", &p[i].bt);
    }
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at) {
                struct Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
    int current_time = 0;
    int total_tat = 0, total_wt = 0, total_rt = 0;

    for (int i = 0; i < n; i++) {

        p[i].st = (current_time > p[i].at) ? current_time : p[i].at;
        p[i].ct = p[i].st + p[i].bt;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        p[i].rt = p[i].st - p[i].at;

        current_time = p[i].ct;

        total_tat += p[i].tat;
        total_wt += p[i].wt;
        total_rt += p[i].rt;
    }

    printf("\n--- FCFS Scheduling Result ---\n");
    printf("PID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++) {
        printf("%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].st, p[i].ct,
               p[i].tat, p[i].wt, p[i].rt);
    }
    printf("\nAverage TAT = %.2f\n", (float)total_tat / n);
    printf("Average WT  = %.2f\n", (float)total_wt / n);
    printf("Average RT  = %.2f\n", (float)total_rt / n);

    return 0;
}
```

When implemented, the algorithm should output for each process its start time, completion time, and derived metrics (TAT, WT, RT). In FCFS, the first process has zero waiting time, while others may wait for all earlier processes to finish.

Shortest Job First (SJF)

Non-preemptive Shortest Job First (SJF) selects among the ready processes the one with the smallest burst time. Once a process starts under SJF, it runs to completion without interruption. SJF tends to reduce average waiting time by running short tasks first. However, it requires knowing burst times in advance and can delay long processes if short ones keep coming.

[Insert your SJF implementation here]

Shortest Remaining Time First (SRTF)

Shortest Remaining Time First (SRTF) is the preemptive version of SJF. At each time unit, SRTF runs the process with the smallest remaining burst time. If a new process arrives with a shorter remaining time than the current process, it preempts the running process. SRTF often achieves even lower waiting times than non-preemptive SJF because it adapts to new arrivals, but it may incur more context switches.

[Insert your SRTF implementation here]

Round Robin (RR)

Round Robin scheduling assigns each ready process a fixed time quantum in a cyclic order. If a process’s burst time exceeds the quantum, it uses its quantum and then goes to the back of the ready queue with its remaining burst time. Round Robin is preemptive and provides fair, time-shared CPU access. The choice of time quantum (for example, 4 time units) affects performance: a small quantum increases context switching and overhead, while a large quantum makes the system behave more like FCFS with longer response times.

[Insert your Round Robin implementation here]

Priority Scheduling (Non-preemptive)

Non-preemptive Priority Scheduling runs the process with the highest priority (lowest numerical value) among the ready processes. Once a process starts, it runs to completion. This approach ensures important tasks run earlier, but if a higher-priority process arrives later, it must wait until the current process finishes.

[Insert your Non-Preemptive Priority implementation here]

Priority Scheduling (Preemptive)

Preemptive Priority Scheduling also runs the highest-priority ready process, but it can preempt the current process whenever a higher-priority process arrives. This ensures urgent tasks start as soon as possible, at the cost of more context switches. Like non-preemptive priority, it risks starving low-priority processes if high-priority jobs keep arriving.

[Insert your Preemptive Priority implementation here]

Comparative Analysis

We compare the algorithms in terms of efficiency and fairness:

- **Efficiency (Minimizing Waiting Time):** SJF and SRTF tend to minimize average waiting time because they always run shorter jobs first. FCFS can have larger waiting times if a long job runs first, delaying shorter jobs. Round Robin generally results in higher average waiting time due to time slicing overhead, although it can improve average response time. Priority scheduling performance depends on priority assignment: if the highest-priority jobs are short or arrive early, it can reduce waiting time, but poorly chosen priorities may increase waiting for some processes.

- **Fairness:** Round Robin is the most fair to all processes because each process receives CPU time slices in rotation, preventing starvation. FCFS is fair by arrival order but can delay short processes if a long process is first. SJF/SRTF can be unfair to long processes (they might wait indefinitely if short tasks keep arriving). Priority scheduling can also be unfair to low-priority processes if higher-priority jobs keep arriving.

- **Effect of Time Quantum (Round Robin):** The time quantum controls the trade-off between responsiveness and overhead. A small quantum makes the system more responsive (shorter response times) but increases context switching overhead, while a large quantum reduces overhead but can make the behavior similar to FCFS (longer waiting times for some processes). A moderate time quantum is typically chosen to balance these effects.

Conclusion

Each scheduling algorithm has its own strengths and trade-offs. FCFS is simple and serves jobs in arrival order, but long jobs can delay others. SJF and SRTF algorithms minimize average waiting time by running short jobs first, making them efficient, but they can starve longer jobs. Round Robin provides good fairness and response time by time-slicing, though it may increase average waiting time if the quantum is not chosen well. Priority scheduling allows urgent tasks to run first, which can improve performance for high-priority jobs, but low-priority processes may suffer delays. The best choice of algorithm depends on the system goals (e.g., minimizing wait time vs. ensuring fairness) and workload characteristics. This lab’s implementation and analysis illustrate how these algorithms behave with the given set of processes.

```
Start
  Input n (number of processes)
  For each process i from 1 to n
      Input Process ID, Arrival Time, Burst Time

  Sort all processes by Arrival Time (earliest first)

  current_time = 0
  total_tat = total_wt = total_rt = 0

  For each process i from 1 to n
      st = max(current_time, arrival_time)
      ct = st + burst_time
      tat = ct - arrival_time
      wt  = tat - burst_time
      rt  = st - arrival_time
      current_time = ct

      total_tat += tat
      total_wt  += wt
      total_rt  += rt

  Print header: PID, AT, BT, ST, CT, TAT, WT, RT
  For each process i
      Print process details
  Print Average TAT = total_tat / n
  Print Average WT  = total_wt / n
  Print Average RT  = total_rt / n
End
```

```c
#include <stdio.h>
struct Process {
    char pid[10];
    int at, bt, st, ct, tat, wt, rt;
};
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i = 0; i < n; i++) {
        printf("\nEnter Process ID for P%d: ", i + 1);
        scanf("%s", p[i].pid);
        printf("Enter Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Enter Burst Time: ");
        scanf("%d", &p[i].bt);
    }
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at) {
                struct Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    int current_time = 0;
    int total_tat = 0, total_wt = 0, total_rt = 0;

    for (int i = 0; i < n; i++) {

        p[i].st = (current_time > p[i].at) ? current_time : p[i].at;
        p[i].ct = p[i].st + p[i].bt;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        p[i].rt = p[i].st - p[i].at;

        current_time = p[i].ct;

        total_tat += p[i].tat;
        total_wt += p[i].wt;
        total_rt += p[i].rt;
    }

    printf("\n--- FCFS Scheduling Result ---\n");
    printf("PID\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n");

    for (int i = 0; i < n; i++) {
        printf("%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].st, p[i].ct,
               p[i].tat, p[i].wt, p[i].rt);
    }
    printf("\nAverage TAT = %.2f\n", (float)total_tat / n);
    printf("Average WT  = %.2f\n", (float)total_wt / n);
    printf("Average RT  = %.2f\n", (float)total_rt / n);

    return 0;
}
```