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

Batch: D-2	**Roll No.:** 16010122151
Experiment No. 05	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority)

AIM: To implement basic Process management algorithms (Round Robin,SRTN, 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.

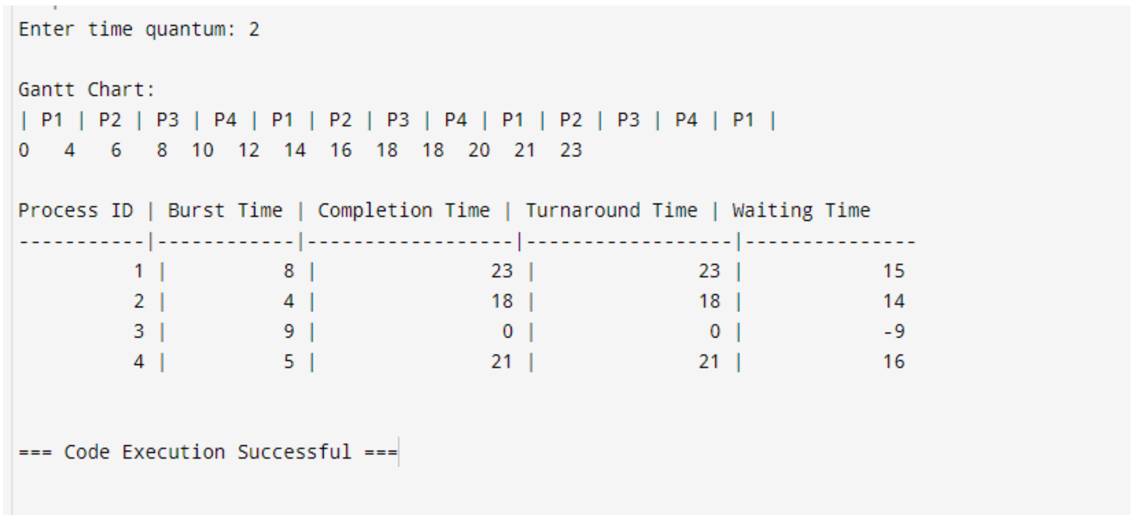
Description of the application to be implemented:

Round Robin Algorithm

```
#include <iostream>
#include <queue>
#include <vector>
#include <iomanip>

struct Process {
    int id;
    int burst_time;
    int remaining_time;
    int completion_time;
    int turnaround_time; // New field for Turnaround Time
    int waiting_time;    // New field for Waiting Time
};

void roundRobin(std::vector<Process>& processes, int time_quantum) {
    std::queue<Process*> readyQueue;
    int time = 0;
    int completed_processes = 0;

    // Initialize remaining times and completion times
    for (auto& p : processes) {
        p.remaining_time = p.burst_time;
        p.completion_time = 0; // Initialize completion time
    }

    std::vector<int> ganttChart; // To store Gantt chart process IDs
    std::vector<int> ganttTime;  // To store time at each step

    while (completed_processes < processes.size()) {
        for (auto& p : processes) {
            if (p.remaining_time > 0) {
                readyQueue.push(&p);
            }
        }

        if (readyQueue.empty()) break; // No more processes to schedule

        Process* current = readyQueue.front();
        readyQueue.pop();

        // Execute the current process
        if (current->remaining_time > time_quantum) {
            time += time_quantum;
            current->remaining_time -= time_quantum;
            ganttChart.push_back(current->id); // Record process in Gantt
chart
            ganttTime.push_back(time);         // Record time
        } else {
            time += current->remaining_time;
            current->completion_time = time; // Set completion time
            current->remaining_time = 0;      // Process completed
            ganttChart.push_back(current->id); // Record process in Gantt
chart
            ganttTime.push_back(time);         // Record time
            completed_processes++;             // Increment completed
processes
        }
    }

    // Calculate Turnaround Time and Waiting Time
    for (auto& p : processes) {
        p.turnaround_time = p.completion_time; // Turnaround time =
Completion time (arrival time is 0)
        p.waiting_time = p.turnaround_time - p.burst_time; // Waiting
time = Turnaround time - Burst time
```

```
    }

    // Print the Gantt Chart
    std::cout << "\nGantt Chart:\n|";
    for (auto id : ganttChart) {
        std::cout << " P" << id << " |";
    }
    std::cout << "\n";

    std::cout << "0";
    for (size_t i = 1; i < ganttTime.size(); ++i) {
        std::cout << std::setw(4) << ganttTime[i];
    }
    std::cout << std::endl;

    // Print process details in a table format
    std::cout << "\nProcess ID | Burst Time | Completion Time |
Turnaround Time | Waiting Time\n";
    std::cout << "-----------|------------|------------------|-----------
-------|---------------\n";
    for (const auto& p : processes) {
        std::cout << std::setw(11) << p.id << " | "
                  << std::setw(10) << p.burst_time << " | "
                  << std::setw(16) << p.completion_time << " | "
                  << std::setw(15) << p.turnaround_time << " | "
                  << std::setw(13) << p.waiting_time << "\n";
    }
}

int main() {
    std::vector<Process> processes = {
        {1, 8, 0, 0}, // Process 1: ID, Burst Time, Remaining Time,
Completion Time
        {2, 4, 0, 0}, // Process 2: ID, Burst Time, Remaining Time,
Completion Time
        {3, 9, 0, 0}, // Process 3: ID, Burst Time, Remaining Time,
Completion Time
        {4, 5, 0, 0}  // Process 4: ID, Burst Time, Remaining Time,
Completion Time
    };

    int time_quantum;
    std::cout << "Enter time quantum: ";
    std::cin >> time_quantum;

    roundRobin(processes, time_quantum);

    return 0;
}
```

OUTPUT:-

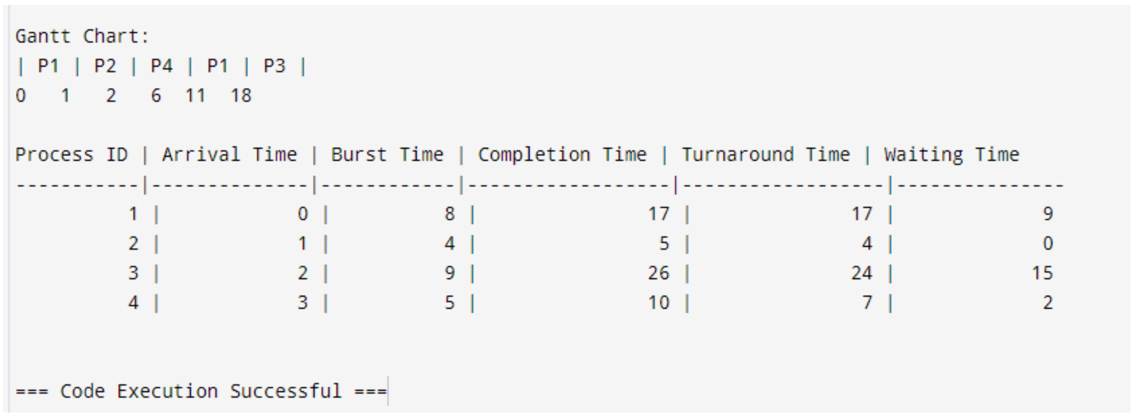


```
Enter time quantum: 2

Gantt Chart:
| P1 | P2 | P3 | P4 | P1 | P2 | P3 | P4 | P1 | P2 | P3 | P4 | P1 |
0    4    6    8   10   12   14   16   18   18   20   21   23

Process ID | Burst Time | Completion Time | Turnaround Time | Waiting Time
-----------|------------|-----------------|-----------------|---------------
         1 |          8 |              23 |              23 |             15
         2 |          4 |              18 |              18 |             14
         3 |          9 |               0 |               0 |             -9
         4 |          5 |              21 |              21 |             16


=== Code Execution Successful ===
```

Shortest Remaining Time First Algorithm :

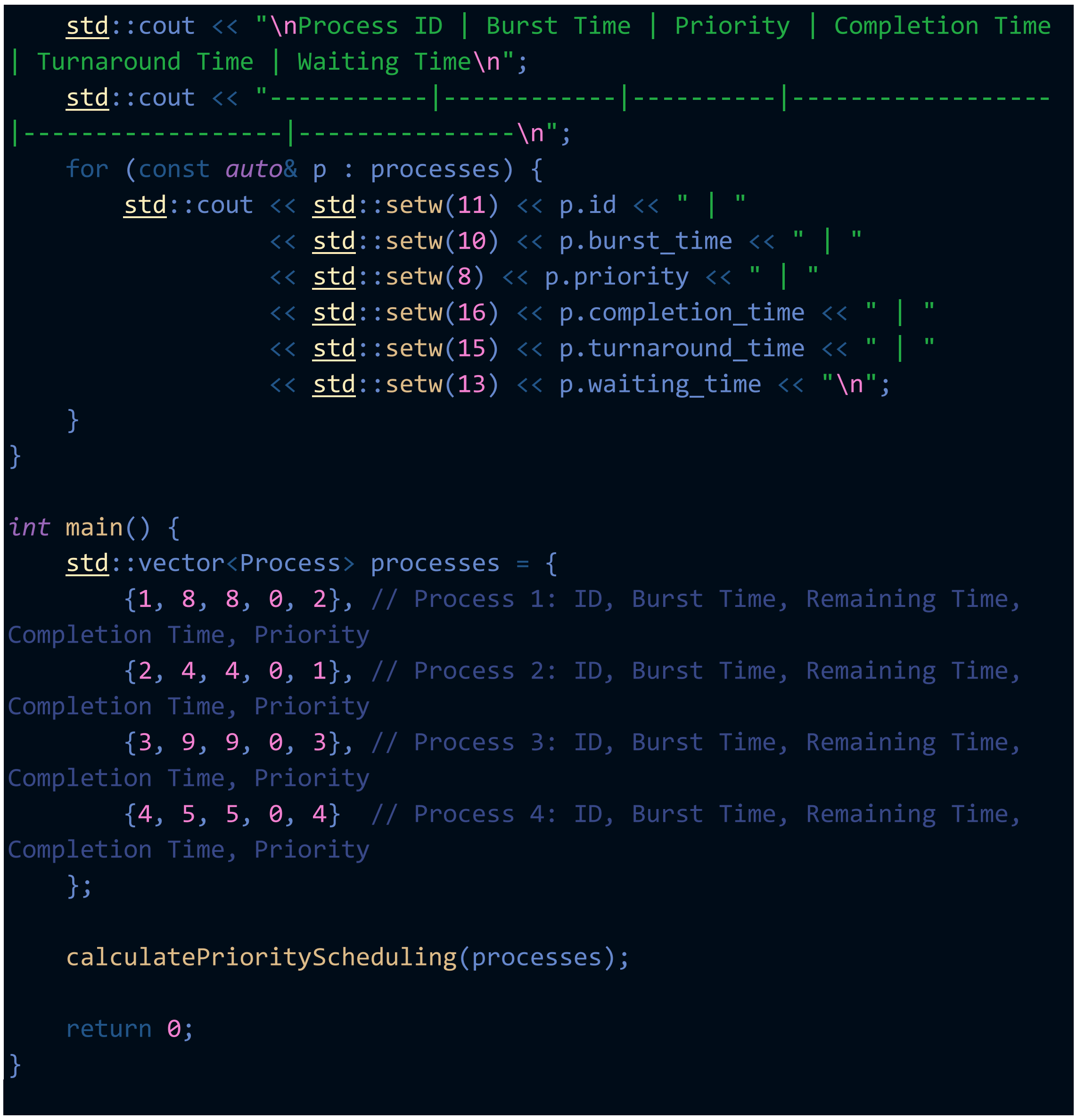
```cpp
#include <iostream>
#include <vector>
#include <iomanip>
#include <limits.h>

struct Process {
    int id;
    int burst_time;
    int remaining_time;
    int completion_time;
    int arrival_time;
    int turnaround_time; // New field for Turnaround Time
    int waiting_time;    // New field for Waiting Time
};

void calculateSRTF(std::vector<Process>& processes) {
    int time = 0;
    int completed_processes = 0;
    int n = processes.size();
    std::vector<int> ganttChart;
    std::vector<int> ganttTimes;

    while (completed_processes < n) {
        // Find the process with the smallest remaining time
        int idx = -1;
        int min_remaining_time = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0 &&
processes[i].arrival_time <= time) {
                if (processes[i].remaining_time < min_remaining_time) {
                    min_remaining_time = processes[i].remaining_time;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            // Execute the process for 1 time unit
            processes[idx].remaining_time--;
            time++;

            // Record the process in the Gantt chart
            if (ganttChart.empty() || ganttChart.back() !=
processes[idx].id) {
                ganttChart.push_back(processes[idx].id);
                ganttTimes.push_back(time); // Record the current time
            }

            // Check for completion
            if (processes[idx].remaining_time == 0) {
                processes[idx].completion_time = time;
                completed_processes++;
            }
        } else {
            time++; // Idle time
        }
    }

    // Calculate Turnaround Time and Waiting Time
    for (auto& p : processes) {
        p.turnaround_time = p.completion_time - p.arrival_time;
        p.waiting_time = p.turnaround_time - p.burst_time;
    }

    // Print the Gantt Chart
    std::cout << "\nGantt Chart:\n|";
    for (size_t i = 0; i < ganttChart.size(); ++i) {
        std::cout << " P" << ganttChart[i] << " |";
    }
    std::cout << "\n";

    // Print time increments
    std::cout << "0";
    for (size_t i = 0; i < ganttTimes.size(); ++i) {
        std::cout << std::setw(4) << ganttTimes[i];
    }
    std::cout << std::endl;

    // Print process details in a table format
    std::cout << "\nProcess ID | Arrival Time | Burst Time | Completion 
Time | Turnaround Time | Waiting Time\n";
    std::cout << "-----------|--------------|------------|--------------
---|------------------|---------------\n";
    for (const auto& p : processes) {
        std::cout << std::setw(11) << p.id << " | "
                  << std::setw(12) << p.arrival_time << " | "
                  << std::setw(10) << p.burst_time << " | "
                  << std::setw(16) << p.completion_time << " | "
                  << std::setw(15) << p.turnaround_time << " | "
                  << std::setw(13) << p.waiting_time << "\n";
    }
}

int main() {
    std::vector<Process> processes = {
        {1, 8, 8, 0, 0}, // Process 1: ID, Burst Time, Remaining Time, 
Completion Time, Arrival Time
        {2, 4, 4, 0, 1}, // Process 2: ID, Burst Time, Remaining Time, 
Completion Time, Arrival Time
        {3, 9, 9, 0, 2}, // Process 3: ID, Burst Time, Remaining Time, 
Completion Time, Arrival Time
        {4, 5, 5, 0, 3}  // Process 4: ID, Burst Time, Remaining Time, 
Completion Time, Arrival Time
    };

    calculateSRTF(processes);

    return 0;
}
```

OUTPUT:-


```
Gantt Chart:
| P1 | P2 | P4 | P1 | P3 |
0    1    2    6   11   18

Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time
-----------|--------------|------------|-----------------|-----------------|---------------
         1 |            0 |          8 |              17 |              17 |             9
         2 |            1 |          4 |               5 |               4 |             0
         3 |            2 |          9 |              26 |              24 |            15
         4 |            3 |          5 |              10 |               7 |             2


=== Code Execution Successful ===
```

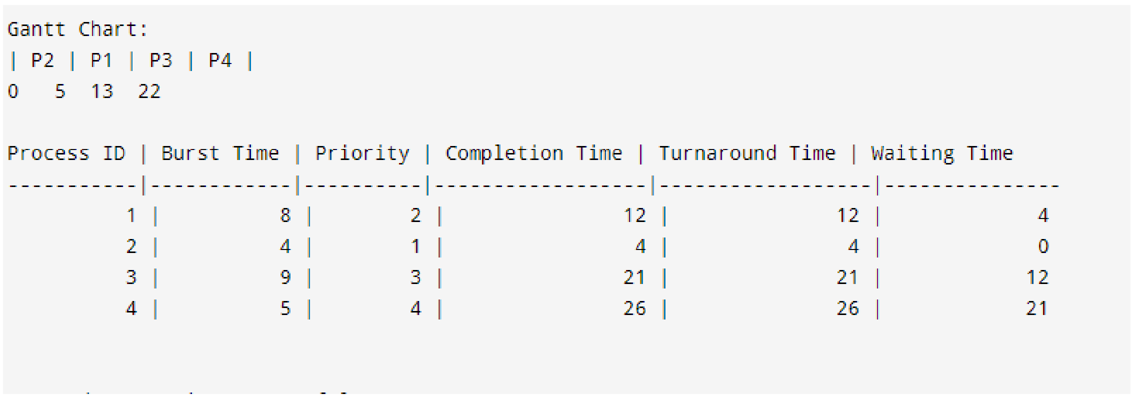
Priority scheduling:

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <limits.h>

struct Process {
    int id;
    int burst_time;
    int remaining_time;
    int completion_time;
    int priority;         // New field for priority
    int turnaround_time;  // New field for Turnaround Time
    int waiting_time;     // New field for Waiting Time
};

void calculatePriorityScheduling(std::vector<Process>& processes) {
    int time = 0;
    int completed_processes = 0;
    int n = processes.size();
    std::vector<int> ganttChart; // To store Gantt chart process IDs
    std::vector<int> ganttTime;  // To store time at each step

    int current_process = -1; // To track the currently running process

    while (completed_processes < n) {
        int idx = -1;
        int highest_priority = INT_MAX;

        // Find the process with the highest priority (lowest number)
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0 && processes[i].priority
< highest_priority) {
                highest_priority = processes[i].priority;
                idx = i;
            }
        }

        if (idx != -1) {
            // Execute the current process for 1 time unit
            processes[idx].remaining_time--;
            time++;

            // Check if we need to record the Gantt chart
            if (current_process != processes[idx].id) {
                ganttChart.push_back(processes[idx].id); // Record
process in Gantt chart
                ganttTime.push_back(time);                // Record time
                current_process = processes[idx].id;      // Update
current process
            }

            // Check for completion
            if (processes[idx].remaining_time == 0) {
                processes[idx].completion_time = time;
                completed_processes++;
            }
        } else {
            time++; // Idle time
        }
    }

    // Calculate Turnaround Time and Waiting Time
    for (auto& p : processes) {
        p.turnaround_time = p.completion_time; // Turnaround time =
Completion time
        p.waiting_time = p.turnaround_time - p.burst_time; // Waiting
time = Turnaround time - Burst time
    }

    // Print the Gantt Chart
    std::cout << "\nGantt Chart:\n|";
    for (size_t i = 0; i < ganttChart.size(); ++i) {
        std::cout << " P" << ganttChart[i] << " |";
    }
    std::cout << "\n";

    std::cout << "0";
    for (size_t i = 1; i < ganttTime.size(); ++i) {
        std::cout << std::setw(4) << ganttTime[i];
    }
    std::cout << std::endl;

    // Print process details in a table format
```



```
    std::cout << "\nProcess ID | Burst Time | Priority | Completion Time
| Turnaround Time | Waiting Time\n";
    std::cout << "-----------|------------|----------|------------------
|------------------|---------------\n";
    for (const auto& p : processes) {
        std::cout << std::setw(11) << p.id << " | "
                  << std::setw(10) << p.burst_time << " | "
                  << std::setw(8) << p.priority << " | "
                  << std::setw(16) << p.completion_time << " | "
                  << std::setw(15) << p.turnaround_time << " | "
                  << std::setw(13) << p.waiting_time << "\n";
    }
}

int main() {
    std::vector<Process> processes = {
        {1, 8, 8, 0, 2}, // Process 1: ID, Burst Time, Remaining Time,
Completion Time, Priority
        {2, 4, 4, 0, 1}, // Process 2: ID, Burst Time, Remaining Time,
Completion Time, Priority
        {3, 9, 9, 0, 3}, // Process 3: ID, Burst Time, Remaining Time,
Completion Time, Priority
        {4, 5, 5, 0, 4}  // Process 4: ID, Burst Time, Remaining Time,
Completion Time, Priority
    };

    calculatePriorityScheduling(processes);

    return 0;
}
```

Output:-



```
Gantt Chart:
| P2 | P1 | P3 | P4 |
0   5   13   22

Process ID | Burst Time | Priority | Completion Time | Turnaround Time | Waiting Time
-----------|------------|----------|-----------------|-----------------|---------------
         1 |          8 |        2 |              12 |              12 |              4
         2 |          4 |        1 |               4 |               4 |              0
         3 |          9 |        3 |              21 |              21 |             12
         4 |          5 |        4 |              26 |              26 |             21
```

Conclusion: Through this experiment we understood the preemptive algorithm for CPU scheduling and implemented Shortest Remaining Time Next (SRTN) and Priority Scheduling

Post Lab Descriptive Questions

1. Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?

A) Shortest remaining time (SRT) scheduling algorithm selects the process for execution which has the smallest amount of time remaining until completion. Let three processes be p0, p1 and p2. Their execution time is 10, 20 and 30 respectively. p0 spends first 2 time units in I/O, 7 units of CPU time and finally 1 unit in I/O. p1 spends first 4 units in I/O, 14 units of CPU time and finally 2 units in I/O. p2 spends first 6 units in I/O, 21 units of CPU time and finally 3 units in I/O.

PID	AT	IO	BT	IO
P0	0	2	7	1
P1	0	4	14	2
P2	0	6	21	3

AT- Arrival Time, IO-input/output, BT-Burst Time

First process p0 will spend 2 units in IO, next 7 units in BT, then process p1 will spend 14 units in BT (as its 4 units of IO has been spent already when previous process was running) and ten process p2 will spend 21 units in BT (as its 6 units of IO has been spent already when previous processes were running) and at last 3 units in IO (process p0,p1,p2's last IO included.) Total time spent = 47

- Idle time = 2 + 3 = 5
- • Percentage of idle time = (5/47)*100 = 10.6 %

2. What effect the time quantum has on its performance. What are the advantages and disadvantages of using a small versus a large time quantum?

A) Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Date: 10-10-2024 **Signature of faculty in-charge**