PROCESS MANAGEMENT REPORT

SUBMITTED BY:

- Bushra Khan
- Affan Ahmed
- Inshirah

SUBMITTED TO:

Miss Sumra
Sir Farzeen

TABLE OF CONTENT

. Introduction

Process and CPU management are critical components of operating systems, ensuring efficient and fair execution of multiple processes. This project focuses on implementing and simulating CPU scheduling algorithms in Java, including First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. The project also supports functionalities like process file loading, execution, and detailed statistics display for analyzing performance.

. Problem Statement

In operating systems, managing CPU time for multiple processes is challenging. It involves ensuring:

- Fair allocation of CPU resources among processes.
- Minimization of waiting and turnaround times.
- Proper handling of varying process burst times.

Without effective scheduling algorithms, processes might face starvation or inefficient CPU usage, degrading system performance. This project addresses these challenges by implementing and comparing two foundational scheduling techniques.

. Proposed Solution

The project implements two CPU scheduling algorithms:

First Come First Serve (FCFS):
Processes are scheduled in the order they arrive.
Simple and easy to implement but may cause high waiting time if a process with a large burst time arrives early.

Round Robin (RR):
Each process gets a fixed time slice (quantum).
Ensures fairness by cycling through processes in a round-robin fashion.

Shortest Job First (SJF):
Processes with the shortest burst time are executed first.
Minimizes average waiting time but may lead to starvation for longer processes.

Priority Scheduling:

Processes are scheduled based on their priority (lower values indicate higher priority).
Efficient for handling critical tasks but may lead to starvation for lower-priority processes.

2. Process Statistics:

- The project calculates and displays key performance metrics for each scheduling algorithm, including:
- Completion Time (CT): The time at which a process finishes execution.
- Turnaround Time (TAT): TAT=CT−ArrivalTimeTAT = CT - Arrival TimeTAT=CT−ArrivalTime.
- Waiting Time (WT): WT=TAT−BurstTimeWT = TAT - Burst TimeWT=TAT−BurstTime.

3. Process File Loading:

- Users can load a process file containing details such as process ID, arrival time, burst time, and priority.

4. Process Execution:

- Processes are executed based on the selected algorithm, and results are displayed in a tabular format for easy analysis.

. Code

Simulator class (main)

```java
import java.util.*;
import java.io.*;

public class Simulator {


    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Scheduler scheduler = new Scheduler();
        ProcessManager processManager = new ProcessManager();
        List<Process> processes = new ArrayList<>();
        List<Process> readyQueue = new ArrayList<>(); // Ready Queue for processes
        Statistics stats = null;

        while (true) {
            System.out.println("\nSelect an option:");
            System.out.println("1. Add Process");
            System.out.println("2. Load Processes from File");
            System.out.println("3. Save Processes to File");
            System.out.println("4. Generate Random Processes");
            System.out.println("5. Run Scheduler");
            System.out.println("6. View Statistics");
            System.out.println("7. Exit");

            int choice = getValidInput(scanner);
```

```java
        switch (choice) {
            case 1:
                // Manually add process
                System.out.print(s:"Enter PID:");
                int pid = getValidInput(scanner);
                System.out.print(s:"Enter Arrival Time:");
                int arrivalTime = getValidInput(scanner);
                System.out.print(s:"Enter Burst Time:");
                int burstTime = getValidInput(scanner);
                System.out.print(s:"Enter Priority:");
                int priority = getValidInput(scanner);
                Process newProcess = new Process(pid, arrivalTime, burstTime, priority);
                processes.add(newProcess);
                try {
                    // Save manually added process to the file
                    processManager.saveProcessesToFile(processes, filename:"processes.txt");
                    System.out.println(x:"Process added and saved to file.");
                } catch (IOException e) {
                    System.out.println("Error saving process to file: " + e.getMessage());
                }
                break;
```

```
                case 2:
                    System.out.print(s:"Enter filename to load processes from: ");
                    String loadFilename = scanner.next();
                    try {
                        processes = processManager.loadProcessesFromFile(loadFilename);
                        System.out.println(x:"Processes loaded successfully!");
                    } catch (IOException e) {
                        System.out.println("Error loading file: " + e.getMessage());
                    }
                    break;
                case 3:
                    System.out.print(s:"Enter filename to save processes: ");
                    String saveFilename = scanner.next();
                    try {
                        processManager.saveProcessesToFile(processes, saveFilename);
                        System.out.println(x:"Processes saved successfully!");
                    } catch (IOException e) {
                        System.out.println("Error saving file: " + e.getMessage());
                    }
                    break;
                case 4:
                    System.out.print(s:"Enter the number of processes to generate: ");
                    int numProcesses = getValidInput(scanner);
                    generateRandomProcesses(processes, numProcesses);
                    try {
                        // Save generated processes to file
                        processManager.saveProcessesToFile(processes, filename:"processes.txt");
                        System.out.println(numProcesses + " random processes generated and saved to file!");
```

```
                    } catch (IOException e) {
                        System.out.println("Error saving random processes to file: " + e.getMessage());
                    }
                    break;

                    case 5:
                    processes.clear();
                    try {
                        // Load processes from file
                        processes = processManager.loadProcessesFromFile(filename:"processes.txt");
                        System.out.println(x:"Processes loaded from file for scheduling.");
                    } catch (IOException e) {
                        System.out.println("Error loading processes from file: " + e.getMessage());
                    }

                    readyQueue.clear();
                    readyQueue.addAll(processes);

                    // Sort readyQueue by arrival time for initial scheduling
                    Collections.sort(readyQueue, Comparator.comparingInt(Process::getArrivalTime));

                    // Display scheduling algorithm menu
                    System.out.println(x:"Choose Scheduling Algorithm:");
                    System.out.println(x:"1. FCFS");
                    System.out.println(x:"2. Round Robin");
                    System.out.println(x:"3. Priority");
                    System.out.println(x:"4. Shortest Job First (SJF)");
                    System.out.println(x:"5. Shortest Remaining Time (SRT)");
```

```
                    int algorithmChoice = getValidInput(scanner);
                    String ganttChart = ""; // Initialize Gantt chart string

                    switch (algorithmChoice) {
                        case 1: // FCFS Scheduling
                            ganttChart = scheduler.fcfs(readyQueue);
                            break;

                        case 2: // Round Robin Scheduling
                            System.out.print(s:"Enter Time Quantum: ");
                            int timeQuantum = getValidInput(scanner);
                            ganttChart = scheduler.roundRobin(readyQueue, timeQuantum);
                            break;
```

```
                    case 3: // Priority Scheduling
                        System.out.print(s:"Choose Priority Type (1 for Preemptive, 2 for Non-Preemptive): ");
                        int priorityType = getValidInput(scanner);
                        if (priorityType == 1) {
                            ganttChart = scheduler.priorityScheduling(readyQueue);
                        } else if (priorityType == 2) {
                            ganttChart = scheduler.priorityScheduling(readyQueue);
                        } else {
                            System.out.println(x:"Invalid priority type. Returning to main menu.");
                            continue;
                        }
                        break;

                    case 4: // Shortest Job First (SJF) Scheduling
                        System.out.print(s:"Shortest Job First(SJF) Scheduling... ");
                        ganttChart = scheduler.shortestJobFirst(readyQueue); // true for Preemptive, false for Non-Preemptive
                        break;

                    case 5: // Shortest Remaining Time (SRT) Scheduling
                        ganttChart = scheduler.shortestRemainingTime(readyQueue);
                        break;

                    default:
                        System.out.println(x:"Invalid choice. Returning to main menu.");
                        continue;
                }
```

```
                    // Generate and display statistics for the selected scheduling algorithm
                    stats = new Statistics(readyQueue, ganttChart);
                    System.out.println(x:"Scheduling completed. View statistics now or return to main menu.");
                    break;




                case 6:
                    if (stats == null) {
                    System.out.println(x:"No statistics available. Please run a scheduling algorithm first.");
                } else {
                    stats.displayStatistics();
                }
                    break;

                case 7:
                    System.out.println(x:"Exiting the program...");
                    System.exit(status:0);
                    break;
                default:
                    System.out.println(x:"Invalid option. Try again.");
            }
        }
    }
```

```
    // Helper method for input validation
    private static int getValidInput(Scanner scanner) {
        while (!scanner.hasNextInt()) {
            System.out.println(x:"Invalid input. Please enter a valid number.");
            scanner.next(); // Consume the invalid input
        }
        return scanner.nextInt();
    }

    // Method to generate random processes
    private static void generateRandomProcesses(List<Process> processes, int numProcesses) {
        Random random = new Random();
        for (int i = 0; i < numProcesses; i++) {
            int pid = i + 1;  // Process ID starts from 1
            int arrivalTime = random.nextInt(bound:100);  // Random arrival time between 0 and 99
            int burstTime = random.nextInt(bound:20) + 1;  // Random burst time between 1 and 20
            int priority = random.nextInt(bound:5) + 1;  // Random priority between 1 and 5
            processes.add(new Process(pid, arrivalTime, burstTime, priority));
        }
    }
}
```

Schedule class

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class Scheduler {

    // FCFS Scheduling Algorithm with Real-time Demonstration
    public String fcfs(List<Process> readyQueue) {
        System.out.println(x:"\nStarting FCFS Scheduling...");

        // Sorting processes by arrival time
        Collections.sort(readyQueue, Comparator.comparingInt(Process::getArrivalTime));

        int currentTime = 0;
        StringBuilder ganttChart = new StringBuilder();

        for (Process process : readyQueue) {
            // Display the process that is being scheduled and the current time
            System.out.println("At time " + currentTime + "ms, scheduling process P" + process.getPid() +
                    " (Arrival: " + process.getArrivalTime() + ", Burst: " + process.getBurstTime() + ")");

            // If there's idle time (if current time is less than process arrival time)
            if (currentTime < process.getArrivalTime()) {
                currentTime = process.getArrivalTime(); // Jump forward to process arrival time
                System.out.println("System is idle until " + currentTime + "ms.");
            }

            // Calculate the process's completion time, turnaround time, and waiting time
            process.setCompletionTime(currentTime + process.getBurstTime());
            process.setTurnaroundTime(process.getCompletionTime() - process.getArrivalTime());
```

```java
public class Scheduler {
    public String fcfs(List<Process> readyQueue) {
            process.setWaitingTime(process.getTurnaroundTime() - process.getBurstTime());

            // Add process to the Gantt chart
            ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");

            // Update current time after process execution
            currentTime += process.getBurstTime();

            // Optional delay for real-time simulation (1-second delay per process)
            try {
                Thread.sleep(millis:1000); // 1-second delay for real-time scheduling
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        ganttChart.append(str:"|");

        // Print the final Gantt chart
        System.out.println("Gantt Chart: " + ganttChart.toString());

        // Save process information and Gantt chart to a file
        saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"processes_with_gantt.txt");

        // Return the Gantt chart
        return ganttChart.toString();
    }
```

```java
// Round Robin Scheduling Algorithm (Simplified)
public String roundRobin(List<Process> readyQueue, int timeQuantum) {
    System.out.println(x:"\nStarting Round Robin Scheduling...");

    // Queue to handle processes in Round Robin
    Queue<Process> queue = new LinkedList<>(readyQueue);
    int currentTime = 0;
    StringBuilder ganttChart = new StringBuilder();
    List<String> executionOrder = new ArrayList<>();

    // Track the remaining burst time for each process
    for (Process process : readyQueue) {
        process.setRemainingBurstTime(process.getBurstTime()); // Initialize remaining burst time
    }

    // Round Robin scheduling loop
    while (!queue.isEmpty()) {
        Process process = queue.poll();

        // Display the process that is currently running
        System.out.println("At time " + currentTime + "ms, running process P" + process.getPid() + " with remaining burst time: " + process.ge

        if (process.getRemainingBurstTime() > timeQuantum) {
            // Process has remaining burst time > timeQuantum
            currentTime += timeQuantum;
            process.setRemainingBurstTime(process.getRemainingBurstTime() - timeQuantum); // Decrease remaining burst time
            ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");
            executionOrder.add("P" + process.getPid());
```

```java
            queue.offer(process);  // Re-add the process to the queue if it's not finished
        } else {
            // Process finishes in this round
            currentTime += process.getRemainingBurstTime();
            ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");
            executionOrder.add("P" + process.getPid());

            // Set the completion time and calculate the waiting and turnaround times
            process.setCompletionTime(currentTime);
            process.setTurnaroundTime(process.getCompletionTime() - process.getArrivalTime());
            process.setWaitingTime(process.getTurnaroundTime() - process.getBurstTime());

            // Set the remaining burst time to 0 as the process has finished
            process.setRemainingBurstTime(remainingBurstTime:0);
        }

        // Optional delay for real-time simulation (optional, to make it visible in real-time)
        try {
            Thread.sleep(millis:1000); // 1-second delay for real-time scheduling
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Output the Gantt Chart
    System.out.println("Gantt Chart: " + ganttChart.toString());
```

```java
    // Output the Gantt Chart
    System.out.println("Gantt Chart: " + ganttChart.toString());

    // Save process details to a file
    saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"processes_with_gantt.txt");

    // Return the Gantt Chart string
    return ganttChart.toString(); // Return the Gantt Chart as a String
}


//priority Scheduling
public String priorityScheduling(List<Process> readyQueue) {
    System.out.println(x:"\nStarting Priority Scheduling...");

    AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Mutable time using AtomicInteger
    StringBuilder ganttChart = new StringBuilder();

    // Initialize the remaining burst time for all processes
    for (Process process : readyQueue) {
        process.setRemainingBurstTime(process.getBurstTime());
    }
```

```
        while (!readyQueue.isEmpty()) {
            // Find the process with the highest priority that has arrived
            Process currentProcess = readyQueue.stream()
                .filter(p -> p.getArrivalTime() <= currentTime.get())
                .min(Comparator.comparingInt(Process::getPriority)) // Smaller priority value means higher priority
                .orElse(other:null);

            if (currentProcess != null) {
                // Log the running process
                System.out.println("At time " + currentTime.get() + "ms, running process P" +
                    currentProcess.getPid() + " (Priority: " + currentProcess.getPriority() +
                    ", Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms)");

                // Execute the process for one time unit (preemptive)
                currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
                ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
                currentTime.incrementAndGet();

                // If the process finishes, calculate its metrics and remove it from the queue
                if (currentProcess.getRemainingBurstTime() == 0) {
                    currentProcess.setCompletionTime(currentTime.get());
                    currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
                    currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
                    readyQueue.remove(currentProcess);
                }
            } else {
                // Idle time if no process is ready
                System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
                ganttChart.append(str:"| Idle ");
```

```
                currentTime.incrementAndGet();
            }
        }
        ganttChart.append(str:"|");

        System.out.println("Gantt Chart: " + ganttChart.toString());
        saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"priority_gantt.txt");

        return ganttChart.toString();
    }


    //SJF

            public String shortestJobFirst(List<Process> readyQueue) {
    System.out.println(x:"\nStarting Shortest Job First (SJF) Scheduling...");

    AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Mutable time using AtomicInteger
    StringBuilder ganttChart = new StringBuilder();

    // Sort processes by arrival time initially
    readyQueue.sort(Comparator.comparingInt(Process::getArrivalTime));

    while (!readyQueue.isEmpty()) {
        // Find the process with the shortest burst time that has arrived
        Process currentProcess = readyQueue.stream()
            .filter(p -> p.getArrivalTime() <= currentTime.get())
            .min(Comparator.comparingInt(Process::getBurstTime))
```

```
            // Log the process running
            System.out.println("At time " + currentTime.get() + "ms, running process P" +
                currentProcess.getPid() + " (Burst Time: " + currentProcess.getBurstTime() + "ms)");

            // Simulate process execution
            ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
            currentTime.addAndGet(currentProcess.getBurstTime());

            // Calculate metrics and remove the process from the ready queue
            currentProcess.setCompletionTime(currentTime.get());
            currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
            currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
            readyQueue.remove(currentProcess);
        } else {
            // Idle time if no process is ready
            System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
            ganttChart.append(str:"| Idle ");
            currentTime.incrementAndGet();
        }
    }
    ganttChart.append(str:"|");

    System.out.println("Gantt Chart: " + ganttChart.toString());
```

```java
public String shortestRemainingTime(List<Process> readyQueue) {
    System.out.println(x:"\nStarting Shortest Remaining Time (SRT) Scheduling...");

    AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Use AtomicInteger for mutable time
    StringBuilder ganttChart = new StringBuilder();

    // Initialize remaining burst time for all processes
    for (Process process : readyQueue) {
        process.setRemainingBurstTime(process.getBurstTime());
    }

    while (!readyQueue.isEmpty()) {
        // Find the process with the shortest remaining time that has arrived
        Process currentProcess = readyQueue.stream()
            .filter(p -> p.getArrivalTime() <= currentTime.get())
            .min(Comparator.comparingInt(Process::getRemainingBurstTime))
            .orElse(other:null);

        if (currentProcess != null) {
            // Log running process
            System.out.println("At time " + currentTime.get() + "ms, running process P" + currentProcess.getPid() +
                " (Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms)");

            // Execute the process for one time unit
            currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
            ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
            currentTime.incrementAndGet();
```

```java
        if (currentProcess != null) {
            // Log running process
            System.out.println("At time " + currentTime.get() + "ms, running process P" + currentProcess.getPid() +
                " (Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms)");

            // Execute the process for one time unit
            currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
            ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
            currentTime.incrementAndGet();

            // If the process finishes, calculate its metrics
            if (currentProcess.getRemainingBurstTime() == 0) {
                currentProcess.setCompletionTime(currentTime.get());
                currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
                currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
                readyQueue.remove(currentProcess);
            }
        } else {
            // Idle time if no process is ready
            System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
            ganttChart.append(str:"| Idle ");
            currentTime.incrementAndGet();
        }
    }
    ganttChart.append(str:"|");
```

```java
    System.out.println("Gantt Chart: " + ganttChart.toString());
    saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"srt_gantt.txt");

    return ganttChart.toString();
}




public void saveProcessesToFile(List<Process> processes, String ganttChart, String filename) {
    try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {
        writer.println(x:"Process Scheduling Result with Gantt Chart:");
        writer.println("Gantt Chart: " + ganttChart);
        writer.println(x:"\nProcess Details:");
        for (Process process : processes) {
            writer.println("PID: " + process.getPid() + ", Arrival Time: " + process.getArrivalTime() +
                    ", Burst Time: " + process.getBurstTime() + ", Waiting Time: " + process.getWaitingTime() +
                    ", Turnaround Time: " + process.getTurnaroundTime() + ", Completion Time: " + process.getCompletionTime());
        }
        System.out.println("Process information and Gantt chart saved to file: " + filename);
    } catch (IOException e) {
        System.out.println("Error saving processes to file: " + e.getMessage());
    }
}
```

Statistic class

```java
import java.util.ArrayList;
import java.util.List;

public class Statistics {
    private List<Process> processes;
    private String ganttChart;

    public Statistics(List<Process> processes, String ganttChart) {
        this.processes = new ArrayList<>(processes); // Copy of processes for immutability
        this.ganttChart = ganttChart;
    }

    public void displayStatistics() {
        System.out.println(x:"\n================ Process Statistics ================");
        System.out.printf(format:"%-10s %-15s %-15s %-15s %-15s %-15s\n",
                ...args:"PID", "Arrival Time", "Burst Time", "Completion Time", "Turnaround Time", "Waiting Time");

        // Display each process' details
        for (Process process : processes) {
            System.out.printf(format:"%-10d %-15d %-15d %-15d %-15d %-15d\n",
                    process.getPid(),
                    process.getArrivalTime(),
                    process.getBurstTime(),
                    process.getCompletionTime(),
                    process.getTurnaroundTime(),
                    process.getWaitingTime());
        }

        System.out.println(x:"\n================ Gantt Chart ================");
        System.out.println(ganttChart);
```

```java
        System.out.println(x:"\n================ Average Times ================");
        System.out.printf(format:"Average Waiting Time: %.2f\n", calculateAverageWaitingTime());
        System.out.printf(format:"Average Turnaround Time: %.2f\n", calculateAverageTurnaroundTime());
    }

    public double calculateAverageWaitingTime() {
        return processes.stream().mapToDouble(Process::getWaitingTime).average().orElse(other:0);
    }

    public double calculateAverageTurnaroundTime() {
        return processes.stream().mapToDouble(Process::getTurnaroundTime).average().orElse(other:0);
    }
}
```

Process Management class

```java
import java.io.*;
import java.util.*;

public class ProcessManager {

    public List<Process> loadProcessesFromFile(String filename) throws IOException {
        List<Process> processes = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(regex:",");
                int pid = Integer.parseInt(parts[0]);
                int arrivalTime = Integer.parseInt(parts[1]);
                int burstTime = Integer.parseInt(parts[2]);
                int priority = Integer.parseInt(parts[3]);
                processes.add(new Process(pid, arrivalTime, burstTime, priority));
            }
```

```
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + filename);
        } catch (IOException e) {
            System.out.println("Error reading from file: " + e.getMessage());
        }
        return processes;
    }

    public void saveProcessesToFile(List<Process> processes, String filename) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (Process process : processes) {
                writer.write(process.getPid() + "," + process.getArrivalTime() + "," +
                             process.getBurstTime() + "," + process.getPriority() + "\n");
            }
        } catch (IOException e) {
            System.out.println("Error saving to file: " + e.getMessage());
        }
    }
}
```

Process class

```
public class Process {
    private int pid; // Process ID
    private int arrivalTime;
    private int burstTime;
    private int remainingBurstTime;
    private int remainingTime; // For algorithms like Round Robin
    private int priority;
    private int waitingTime;
    private int turnaroundTime;
    private int completionTime;

    public Process(int pid, int arrivalTime, int burstTime, int priority) {
        this.pid = pid;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.remainingTime = burstTime; // Initial remaining time is the same as burst time
        this.priority = priority;
    }

    // Getters and Setters

    public int getPid() {
        return pid;
    }

    public void setPid(int pid) {
        this.pid = pid;
    }
```

```
    public int getArrivalTime() {
    }

    public void setArrivalTime(int arrivalTime) {
        this.arrivalTime = arrivalTime;
    }

    public int getBurstTime() {
        return burstTime;
    }

    public void setBurstTime(int burstTime) {
        this.burstTime = burstTime;
    }
```

```
    public int getRemainingBurstTime() {
        return remainingBurstTime;
    }

    public void setRemainingBurstTime(int remainingBurstTime) {
        this.remainingBurstTime = remainingBurstTime;
    }

    public int getRemainingTime() {
        return remainingTime;
    }

    public void setRemainingTime(int remainingTime) {
        this.remainingTime = remainingTime;
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }

    public int getWaitingTime() {
        return waitingTime;
    }
```

```
    public int getWaitingTime() {
        return waitingTime;
    }

    public void setWaitingTime(int waitingTime) {
        this.waitingTime = waitingTime;
    }

    public int getTurnaroundTime() {
        return turnaroundTime;
    }

    public void setTurnaroundTime(int turnaroundTime) {
        this.turnaroundTime = turnaroundTime;
    }

    public int getCompletionTime() {
        return completionTime;
    }

    public void setCompletionTime(int completionTime) {
        this.completionTime = completionTime;
    }

}
```

. Methodology

The project follows a systematic approach, ensuring clarity and efficiency:

1. **Process Generation:**

- Random processes are generated with unique IDs, arrival times, and burst times.

2. Input Handling:

- The user selects the scheduling algorithm (FCFS or RR).
- In the case of RR, the user also provides the time quantum.

3. Algorithm Implementation:

- FCFS: Sorts processes based on arrival times and calculates metrics sequentially.
- RR: Iterates through processes in a cyclic manner using the given time quantum.

4. Calculation of Metrics:

- Completion Time (CT): The time at which a process finishes execution.
- Turnaround Time (TAT): TAT=CT−ArrivalTimeTAT = CT - Arrival TimeTAT=CT−ArrivalTime.
- Waiting Time (WT): WT=TAT−BurstTimeWT = TAT - Burst TimeWT=TAT−BurstTime.

5 . Output Display:

- Processes are displayed in a tabular format with IDs, arrival times, burst times, completion times, turnaround times, and waiting times.

6.Testing and Validation:

- The system is tested with varying numbers of processes and different time quantums to ensure accuracy and reliability.

. WORKING

The project is designed in Java, leveraging its object-oriented capabilities for modular and organized implementation. The main components include:

1.Process Class:

- Encapsulates details such as process ID, arrival time, burst time, and calculated metrics.

2.Scheduler Class:

- Implements both FCFS and RR algorithms.
- Handles process sorting and time calculations.

3.Display Module:

- Outputs process details in a well-organized tabular format.

4. Steps of Execution:

1. Generate a random number of processes with unique IDs, arrival times, and burst times.
2. User selects the scheduling algorithm (FCFS or RR).
3. For RR, the user provides the time quantum.
4. The chosen algorithm processes the list of processes and calculates all necessary metrics.
5. Outputs are displayed, including process ID, arrival time, burst time, completion time, turnaround time, and waiting time.

. Result & Output

Sample Output Format:

Process ID	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	0	5	10	10	5
2	2	3	12	10	7
3	4	7	19	15	8

```
---------- Process Statistics ----------
PID     Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
3       1               2               2               1               -1
5       8               15              176             168             153
8       16              3               8               -8              -11
6       22              8               109             87              79
7       31              8               111             80              72
9       35              6               71              36              30
10      42              15              179             137             122
1       43              2               22              -21             -23
4       53              7               115             62              55
3       62              12              151             89              77
9       62              3               31              -31             -34
10      68              20              205             137             117
2       72              2               36              -36             -38
7       75              10              155             80              70
1       83              14              184             101             87
6       85              14              186             101             87
5       86              13              187             101             88
8       87              18              200             113             95
3       90              13              191             101             88
4       98              2               56              -42             -44
2       99              20              207             108             88

---------- Gantt Chart ----------
| P3 | P5 | P8 | P6 | P7 | P9 | P10 | P1 | P4 | P3 | P9 | P10 | P2 | P7 | P1 | P6 | P5 | P8 | P3 | P4 | P2 | P5 | P6 | P7 | P9 | P10 | P4 | P3 | P10 | P7 | P1 | P6 | P5 | P8 | P3 | P2 | P5 |
10 | P4 | P3 | P10 | P7 | P1 | P6 | P5 | P8 | P3 | P2 | P5 | P10 | P3 | P10 | P7 | P1 | P6 | P5 | P8 | P3 | P2 | P5 | P10 | P10 | P1 | P6 | P5 | P8 | P3 | P2 | P10 | P8 | P2 | P10 | P2

---------- Average Times ----------
Average Waiting Time: 55.10
Average Turnaround Time: 64.95
```

Key Observations:

- FCFS: Simple but may result in long waiting times for processes arriving later.
- RR: Reduces waiting times for processes with smaller burst times, ensuring fairness.

. CONCLUSION

This project successfully demonstrates the implementation of two essential CPU scheduling algorithms: First Come First Serve (FCFS) and Round Robin (RR). By dynamically generating processes and allowing user selection of algorithms, it provides a practical simulation of process and CPU management in operating systems.

Key Takeaways:

- FCFS is straightforward but can lead to inefficiencies for certain workloads.
- Round Robin ensures fairness and is well-suited for time-sharing systems.
- The organized Java implementation ensures modularity, readability, and scalability.

This project serves as a valuable learning tool for understanding and comparing scheduling techniques, forming a foundation for further exploration of process management in operating systems.