

PROCESS MANAGEMENT REPORT

SUBMITTED BY:

- Bushra Khan
- Affan Ahmed
- Inshirah

SUBMITTED TO:

Miss Sumra
Sir Farzeen

TABLE OF CONTENT

- . INTRODUCTION
- . PROBLEM STATEMENT
- . PROPOSED SOLUTION
- . METHODOLOGY
- . WORKING
- . RESULT & OUTPUT
- . CONCLUSION

. 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): $TAT = CT - ARRIVAL\ TIME$
- WAITING TIME (WT): $WT = TAT - BURST\ TIME$

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)

```
1 import java.util.*;
2 import java.io.*;
3
4 public class Simulator {
5
6
7     public static void main(String[] args) {
8         Scanner scanner = new Scanner(System.in);
9         Scheduler scheduler = new Scheduler();
10        ProcessManager processManager = new ProcessManager();
11        List<Process> processes = new ArrayList<>();
12        List<Process> readyQueue = new ArrayList<>(); // Ready Queue for processes
13        Statistics stats = null;
14
15        while (true) {
16            System.out.println("\nSelect an option:");
17            System.out.println("1. Add Process");
18            System.out.println("2. Load Processes from File");
19            System.out.println("3. Save Processes to File");
20            System.out.println("4. Generate Random Processes");
21            System.out.println("5. Run Scheduler");
22            System.out.println("6. View Statistics");
23            System.out.println("7. Exit");
24
25            int choice = getValidInput(scanner);
26
27        }
28    }
29
30    int getValidInput(Scanner scanner) {
31        int choice;
32        while (true) {
33            try {
34                choice = scanner.nextInt();
35            } catch (InputMismatchException e) {
36                System.out.print("Please enter a valid integer value: ");
37                scanner.next();
38            }
39            if (choice < 1 || choice > 7) {
40                System.out.println("Invalid choice. Please enter a value between 1 and 7.");
41            } else {
42                break;
43            }
44        }
45        return choice;
46    }
47
48    void runScheduler() {
49        scheduler.run();
50    }
51
52    void saveProcessesToFile(List<Process> processes, String filename) {
53        try {
54            processManager.saveProcessesToFile(processes, filename: "processes.txt");
55            System.out.println("Process added and saved to file.");
56        } catch (IOException e) {
57            System.out.println("Error saving process to file: " + e.getMessage());
58        }
59    }
60
61    void generateRandomProcesses(int numProcesses) {
62        processManager.generateRandomProcesses(numProcesses);
63    }
64
65    void viewStatistics() {
66        stats.print();
67    }
68
69    void loadProcessesFromFile(String filename) {
70        processManager.loadProcessesFromFile(filename: "processes.txt");
71    }
72
73    void runScheduler() {
74        scheduler.run();
75    }
76
77    void saveProcessesToFile(List<Process> processes, String filename) {
78        try {
79            processManager.saveProcessesToFile(processes, filename: "processes.txt");
80            System.out.println("Process added and saved to file.");
81        } catch (IOException e) {
82            System.out.println("Error saving process to file: " + e.getMessage());
83        }
84    }
85
86    void generateRandomProcesses(int numProcesses) {
87        processManager.generateRandomProcesses(numProcesses);
88    }
89
90    void viewStatistics() {
91        stats.print();
92    }
93
94    void loadProcessesFromFile(String filename) {
95        processManager.loadProcessesFromFile(filename: "processes.txt");
96    }
97
98    void runScheduler() {
99        scheduler.run();
100    }
101}
```

```
switch (choice) {
    case 1:
        // Manually add process
        System.out.print("Enter PID:");
        int pid = getValidInput(scanner);
        System.out.print("Enter Arrival Time:");
        int arrivalTime = getValidInput(scanner);
        System.out.print("Enter Burst Time:");
        int burstTime = getValidInput(scanner);
        System.out.print("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("Process added and saved to file.");
        } catch (IOException e) {
            System.out.println("Error saving process to file: " + e.getMessage());
        }
        break;
}
```

```

47     case 2:
48         System.out.print(s:"Enter filename to load processes from: ");
49         String loadFilename = scanner.next();
50         try {
51             processes = processManager.loadProcessesFromFile(loadFilename);
52             System.out.println(x:"Processes loaded successfully!");
53         } catch (IOException e) {
54             System.out.println("Error loading file: " + e.getMessage());
55         }
56         break;
57     case 3:
58         System.out.print(s:"Enter filename to save processes: ");
59         String saveFilename = scanner.next();
60         try {
61             processManager.saveProcessesToFile(processes, saveFilename);
62             System.out.println(x:"Processes saved successfully!");
63         } catch (IOException e) {
64             System.out.println("Error saving file: " + e.getMessage());
65         }
66         break;
67     case 4:
68         System.out.print(s:"Enter the number of processes to generate: ");
69         int numProcesses = getValidInput(scanner);
70         generateRandomProcesses(processes, numProcesses);
71         try {
72             // Save generated processes to file
73             processManager.saveProcessesToFile(processes, filename:"processes.txt");
74             System.out.println(numProcesses + " random processes generated and saved to file!");

```

```

75         } catch (IOException e) {
76             System.out.println("Error saving random processes to file: " + e.getMessage());
77         }
78         break;
79
80     case 5:
81         processes.clear();
82         try {
83             // Load processes from file
84             processes = processManager.loadProcessesFromFile(filename:"processes.txt");
85             System.out.println(x:"Processes loaded from file for scheduling.");
86         } catch (IOException e) {
87             System.out.println("Error loading processes from file: " + e.getMessage());
88         }
89
90         readyQueue.clear();
91         readyQueue.addAll(processes);
92
93         // Sort readyQueue by arrival time for initial scheduling
94         Collections.sort(readyQueue, Comparator.comparingInt(Process::getArrivalTime));
95
96         // Display scheduling algorithm menu
97         System.out.println(x:"Choose Scheduling Algorithm:");
98         System.out.println(x:"1. FCFS");
99         System.out.println(x:"2. Round Robin");
100        System.out.println(x:"3. Priority");
101        System.out.println(x:"4. Shortest Job First (SJF)");
102        System.out.println(x:"5. Shortest Remaining Time (SRT)");

```

```

104
105         int algorithmChoice = getValidInput(scanner);
106         String ganttChart = ""; // Initialize Gantt chart string
107
108         switch (algorithmChoice) {
109             case 1: // FCFS Scheduling
110                 ganttChart = scheduler.fcfs(readyQueue);
111                 break;
112
113             case 2: // Round Robin Scheduling
114                 System.out.print(s:"Enter Time Quantum: ");
115                 int timeQuantum = getValidInput(scanner);
116                 ganttChart = scheduler.roundRobin(readyQueue, timeQuantum);
117                 break;

```

```

118     case 3: // Priority Scheduling
119         System.out.print(s:"Choose Priority Type (1 for Preemptive, 2 for Non-Preemptive): ");
120         int priorityType = getValidInput(scanner);
121         if (priorityType == 1) {
122             ganttChart = scheduler.priorityScheduling(readyQueue);
123         } else if (priorityType == 2) {
124             ganttChart = scheduler.priorityScheduling(readyQueue);
125         } else {
126             System.out.println(x:"Invalid priority type. Returning to main menu.");
127             continue;
128         }
129         break;
130
131     case 4: // Shortest Job First (SJF) Scheduling
132         System.out.print(s:"Shortest Job First(SJF) Scheduling... ");
133         ganttChart = scheduler.shortestJobFirst(readyQueue); // true for Preemptive, false for Non-Preemptive
134         break;
135
136     case 5: // shortest Remaining Time (SRT) Scheduling
137         ganttChart = scheduler.shortestRemainingTime(readyQueue);
138         break;
139
140     default:
141         System.out.println(x:"Invalid choice. Returning to main menu.");
142         continue;
143     }
144
145     // Generate and display statistics for the selected scheduling algorithm
146     stats = new Statistics(readyQueue, ganttChart);
147     System.out.println(x:"Scheduling completed. View statistics now or return to main menu.");
148     break;
149
150
151
152
153
154     case 6:
155         if (stats == null) {
156             System.out.println(x:"No statistics available. Please run a scheduling algorithm first.");
157         } else {
158             stats.displayStatistics();
159         }
160         break;
161
162     case 7:
163         System.out.println(x:"Exiting the program...");
164         System.exit(status:0);
165         break;
166     default:
167         System.out.println(x:"Invalid option. Try again.");
168     }
169 }
170 }
171

```

```

172     // Helper method for input validation
173     private static int getValidInput(Scanner scanner) {
174         while (!scanner.hasNextInt()) {
175             System.out.println(x:"Invalid input. Please enter a valid number.");
176             scanner.next(); // Consume the invalid input
177         }
178         return scanner.nextInt();
179     }
180
181     // Method to generate random processes
182     private static void generateRandomProcesses(List<Process> processes, int numProcesses) {
183         Random random = new Random();
184         for (int i = 0; i < numProcesses; i++) {
185             int pid = i + 1; // Process ID starts from 1
186             int arrivalTime = random.nextInt(bound:100); // Random arrival time between 0 and 99
187             int burstTime = random.nextInt(bound:20) + 1; // Random burst time between 1 and 20
188             int priority = random.nextInt(bound:5) + 1; // Random priority between 1 and 5
189             processes.add(new Process(pid, arrivalTime, burstTime, priority));
190         }
191     }
192 }
193

```

SCHEDULE CLASS

```
1  ✓ import java.io.FileWriter;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.util.*;
5  import java.util.concurrent.atomic.AtomicInteger;
6
7  ✓ public class Scheduler {
8
9      // FCFS Scheduling Algorithm with Real-time Demonstration
10     public String fcfs(List<Process> readyQueue) {
11         System.out.println("Starting FCFS Scheduling...");
12
13         // Sorting processes by arrival time
14         Collections.sort(readyQueue, Comparator.comparingInt(Process::getArrivalTime));
15
16         int currentTime = 0;
17         StringBuilder ganttChart = new StringBuilder();
18
19         for (Process process : readyQueue) {
20             // Display the process that is being scheduled and the current time
21             System.out.println("At time " + currentTime + "ms, scheduling process P" + process.getPid() +
22                 " (Arrival: " + process.getArrivalTime() + ", Burst: " + process.getBurstTime() + ")");
23
24             // If there's idle time (if current time is less than process arrival time)
25             if (currentTime < process.getArrivalTime()) {
26                 currentTime = process.getArrivalTime(); // Jump forward to process arrival time
27                 System.out.println("System is idle until " + currentTime + "ms.");
28             }
29
30             // Calculate the process's completion time, turnaround time, and waiting time
31             process.setCompletionTime(currentTime + process.getBurstTime());
32             process.setTurnaroundTime(process.getCompletionTime() - process.getArrivalTime());
33
34
35         public class Scheduler {
36             public String fcfs(List<Process> readyQueue) {
37                 process.setWaitingTime(process.getTurnaroundTime() - process.getBurstTime());
38
39                 // Add process to the Gantt chart
40                 ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");
41
42                 // Update current time after process execution
43                 currentTime += process.getBurstTime();
44
45                 // Optional delay for real-time simulation (1-second delay per process)
46                 try {
47                     Thread.sleep(1000); // 1-second delay for real-time scheduling
48                 } catch (InterruptedException e) {
49                     e.printStackTrace();
50                 }
51
52                 ganttChart.append(str:"|");
53
54                 // Print the final Gantt chart
55                 System.out.println("Gantt Chart: " + ganttChart.toString());
56
57                 // Save process information and Gantt chart to a file
58                 saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"processes_with_gantt.txt");
59
60                 // Return the Gantt chart
61                 return ganttChart.toString();
62             }
63
64
65
66
67
68
69
70
71 }
```

```

64 // Round Robin Scheduling Algorithm (Simplified)
65 public String roundRobin(List<Process> readyQueue, int timeQuantum) {
66     System.out.println(x:"\nStarting Round Robin Scheduling...");
67
68     // Queue to handle processes in Round Robin
69     Queue<Process> queue = new LinkedList<>(readyQueue);
70     int currentTime = 0;
71     StringBuilder ganttChart = new StringBuilder();
72     List<String> executionOrder = new ArrayList<>();
73
74     // Track the remaining burst time for each process
75     for (Process process : readyQueue) {
76         process.setRemainingBurstTime(process.getBurstTime()); // Initialize remaining burst time
77     }
78
79     // Round Robin scheduling Loop
80     while (!queue.isEmpty()) {
81         Process process = queue.poll();
82
83         // Display the process that is currently running
84         System.out.println("At time " + currentTime + "ms, running process P" + process.getPid() + " with remaining burst time: " + process.getRemainingBurstTime());
85
86         if (process.getRemainingBurstTime() > timeQuantum) {
87             // Process has remaining burst time > timeQuantum
88             currentTime += timeQuantum;
89             process.setRemainingBurstTime(process.getRemainingBurstTime() - timeQuantum); // Decrease remaining burst time
90             ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");
91             executionOrder.add("P" + process.getPid());
92
93             queue.offer(process); // Re-add the process to the queue if it's not finished
94         } else {
95             // Process finishes in this round
96             currentTime += process.getRemainingBurstTime();
97             ganttChart.append(str:"| P").append(process.getPid()).append(str:" ");
98             executionOrder.add("P" + process.getPid());
99
100            // Set the completion time and calculate the waiting and turnaround times
101            process.setCompletionTime(currentTime);
102            process.setTurnaroundTime(process.getCompletionTime() - process.getArrivalTime());
103            process.setWaitingTime(process.getTurnaroundTime() - process.getBurstTime());
104
105            // Set the remaining burst time to 0 as the process has finished
106            process.setRemainingBurstTime(remainingBurstTime:0);
107        }
108
109        // Optional delay for real-time simulation (optional, to make it visible in real-time)
110        try {
111            Thread.sleep(millis:1000); // 1-second delay for real-time scheduling
112        } catch (InterruptedException e) {
113            e.printStackTrace();
114        }
115    }
116
117    // Output the Gantt Chart
118    System.out.println("Gantt Chart: " + ganttChart.toString());
119
120    // Output the Gantt Chart
121    System.out.println("Gantt Chart: " + ganttChart.toString());
122
123    // Save process details to a file
124    saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"processes_with_gantt.txt");
125
126
127    // Return the Gantt Chart string
128    return ganttChart.toString(); // Return the Gantt Chart as a String
129
130
131    // priority Scheduling
132    public String priorityScheduling(List<Process> readyQueue) {
133        System.out.println(x:"\nStarting Priority Scheduling...");
134
135        AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Mutable time using AtomicInteger
136        StringBuilder ganttChart = new StringBuilder();
137
138        // Initialize the remaining burst time for all processes
139        for (Process process : readyQueue) {
140            process.setRemainingBurstTime(process.getBurstTime());
141        }

```

```

140     while (!readyQueue.isEmpty()) {
141         // Find the process with the highest priority that has arrived
142         Process currentProcess = readyQueue.stream()
143             .filter(p -> p.getArrivalTime() <= currentTime.get())
144             .min(Comparator.comparingInt(Process::getPriority)) // Smaller priority value means higher priority
145             .orElse(null);
146
147         if (currentProcess != null) {
148             // Log the running process
149             System.out.println("At time " + currentTime.get() + "ms, running process P" +
150                 " (" + currentProcess.getPid() + " (Priority: " + currentProcess.getPriority() +
151                 ", Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms)");
152
153             // Execute the process for one time unit (preemptive)
154             currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
155             ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
156             currentTime.incrementAndGet();
157
158             // If the process finishes, calculate its metrics and remove it from the queue
159             if (currentProcess.getRemainingBurstTime() == 0) {
160                 currentProcess.setCompletionTime(currentTime.get());
161                 currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
162                 currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
163                 readyQueue.remove(currentProcess);
164             }
165         } else {
166             // Idle time if no process is ready
167             System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
168             ganttChart.append(str:"| Idle ");
169
170         }
171     }
172     ganttChart.append(str:"|");
173
174     System.out.println("Gantt Chart: " + ganttChart.toString());
175     saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"priority_gantt.txt");
176
177     return ganttChart.toString();
178 }
179
180
181 //SJF
182
183     public String shortestJobFirst(List<Process> readyQueue) {
184     System.out.println(x:"\nStarting Shortest Job First (SJF) Scheduling...");
185
186     AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Mutable time using AtomicInteger
187     StringBuilder ganttChart = new StringBuilder();
188
189     // Sort processes by arrival time initially
190     readyQueue.sort(Comparator.comparingInt(Process::getArrivalTime));
191
192     while (!readyQueue.isEmpty()) {
193         // Find the process with the shortest burst time that has arrived
194         Process currentProcess = readyQueue.stream()
195             .filter(p -> p.getArrivalTime() <= currentTime.get())
196             .min(Comparator.comparingInt(Process::getBurstTime))
197
198             // Log the process running
199             System.out.println("At time " + currentTime.get() + "ms, running process P" +
200                 " (" + currentProcess.getPid() + " (Burst Time: " + currentProcess.getBurstTime() + "ms)");
201
202             // Simulate process execution
203             ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
204             currentTime.addAndGet(currentProcess.getBurstTime());
205
206             // Calculate metrics and remove the process from the ready queue
207             currentProcess.setCompletionTime(currentTime.get());
208             currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
209             currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
210             readyQueue.remove(currentProcess);
211
212     } else {
213         // Idle time if no process is ready
214         System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
215         ganttChart.append(str:"| Idle ");
216         currentTime.incrementAndGet();
217
218     }
219 }
220 ganttChart.append(str:"|");
221
222 System.out.println("Gantt Chart: " + ganttChart.toString());

```

```

232 public String shortestRemainingTime(List<Process> readyQueue) {
233     System.out.println(x:"\nStarting Shortest Remaining Time (SRT) Scheduling...");
234
235     AtomicInteger currentTime = new AtomicInteger(initialValue:0); // Use AtomicInteger for mutable time
236     StringBuilder ganttChart = new StringBuilder();
237
238     // Initialize remaining burst time for all processes
239     for (Process process : readyQueue) {
240         process.setRemainingBurstTime(process.getBurstTime());
241     }
242
243     while (!readyQueue.isEmpty()) {
244         // Find the process with the shortest remaining time that has arrived
245         Process currentProcess = readyQueue.stream()
246             .filter(p -> p.getArrivalTime() <= currentTime.get())
247             .min(Comparator.comparingInt(Process::getRemainingBurstTime))
248             .orElse(null);
249
250         if (currentProcess != null) {
251             // Log running process
252             System.out.println("At time " + currentTime.get() + "ms, running process P" + currentProcess.getPid() +
253                 " (Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms");
254
255             // Execute the process for one time unit
256             currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
257             ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
258             currentTime.incrementAndGet();
259
260             if (currentProcess != null) {
261                 // Log running process
262                 System.out.println("At time " + currentTime.get() + "ms, running process P" + currentProcess.getPid() +
263                     " (Remaining Burst Time: " + currentProcess.getRemainingBurstTime() + "ms");
264
265                 // Execute the process for one time unit
266                 currentProcess.setRemainingBurstTime(currentProcess.getRemainingBurstTime() - 1);
267                 ganttChart.append(str:"| P").append(currentProcess.getPid()).append(str:" ");
268                 currentTime.incrementAndGet();
269
270                 // If the process finishes, calculate its metrics
271                 if (currentProcess.getRemainingBurstTime() == 0) {
272                     currentProcess.setCompletionTime(currentTime.get());
273                     currentProcess.setTurnaroundTime(currentProcess.getCompletionTime() - currentProcess.getArrivalTime());
274                     currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentProcess.getBurstTime());
275                     readyQueue.remove(currentProcess);
276                 }
277             } else {
278                 // Idle time if no process is ready
279                 System.out.println("At time " + currentTime.get() + "ms, CPU is idle.");
280                 ganttChart.append(str:"| Idle ");
281                 currentTime.incrementAndGet();
282             }
283         }
284     }
285
286     System.out.println("Gantt Chart: " + ganttChart.toString());
287     saveProcessesToFile(readyQueue, ganttChart.toString(), filename:"srt_gantt.txt");
288
289     return ganttChart.toString();
290 }
291
292
293
294
295
296 public void saveProcessesToFile(List<Process> processes, String ganttChart, String filename) {
297     try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {
298         writer.println(x:"Process Scheduling Result with Gantt chart:");
299         writer.println("Gantt Chart: " + ganttChart);
300         writer.println(x:"\nProcess Details:");
301         for (Process process : processes) {
302             writer.println("PID: " + process.getPid() + ", Arrival Time: " + process.getArrivalTime() +
303                 ", Burst Time: " + process.getBurstTime() + ", Waiting Time: " + process.getWaitingTime() +
304                 ", Turnaround Time: " + process.getTurnaroundTime() + ", Completion Time: " + process.getCompletionTime());
305         }
306         System.out.println("Process information and Gantt chart saved to file: " + filename);
307     } catch (IOException e) {
308         System.out.println("Error saving processes to file: " + e.getMessage());
309     }
310 }

```

STATIC CLASS

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Statistics {
5     private List<Process> processes;
6     private String ganttChart;
7
8     public Statistics(List<Process> processes, String ganttChart) {
9         this.processes = new ArrayList<>(processes); // Copy of processes for immutability
10        this.ganttChart = ganttChart;
11    }
12
13    public void displayStatistics() {
14        System.out.println(x:"\n===== Process Statistics =====");
15        System.out.printf(format:"%-10s %-15s %-15s %-15s %-15s\n",
16                           ...args:"PID", "Arrival Time", "Burst Time", "Completion Time", "Turnaround Time", "Waiting Time");
17
18        // Display each process' details
19        for (Process process : processes) {
20            System.out.printf(format:"%-10d %-15d %-15d %-15d %-15d\n",
21                             process.getPid(),
22                             process.getArrivalTime(),
23                             process.getBurstTime(),
24                             process.getCompletionTime(),
25                             process.getTurnaroundTime(),
26                             process.getWaitingTime());
27        }
28
29        System.out.println(x:"\n===== Gantt Chart =====");
30        System.out.println(ganttChart);
31
32        System.out.println(x:"\n===== Average Times =====");
33        System.out.printf(format:"Average Waiting Time: %.2f\n", calculateAverageWaitingTime());
34        System.out.printf(format:"Average Turnaround Time: %.2f\n", calculateAverageTurnaroundTime());
35    }
36
37    public double calculateAverageWaitingTime() {
38        return processes.stream().mapToDouble(Process::getWaitingTime).average().orElse(0);
39    }
40
41    public double calculateAverageTurnaroundTime() {
42        return processes.stream().mapToDouble(Process::getTurnaroundTime).average().orElse(0);
43    }
44}
45|
```

PROCESS MANAGEMENT CLASS

```
1 import java.io.*;
2 import java.util.*;
3
4 public class ProcessManager {
5
6     public List<Process> loadProcessesFromFile(String filename) throws IOException {
7         List<Process> processes = new ArrayList<>();
8         try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
9             String line;
10            while ((line = reader.readLine()) != null) {
11                String[] parts = line.split(regex:",");
12                int pid = Integer.parseInt(parts[0]);
13                int arrivalTime = Integer.parseInt(parts[1]);
14                int burstTime = Integer.parseInt(parts[2]);
15                int priority = Integer.parseInt(parts[3]);
16                processes.add(new Process(pid, arrivalTime, burstTime, priority));
17            }
18        }
19    }
20}
```

```

18     } catch (FileNotFoundException e) {
19         System.out.println("File not found: " + filename);
20     } catch (IOException e) {
21         System.out.println("Error reading from file: " + e.getMessage());
22     }
23     return processes;
24 }
25
26 public void saveProcessesToFile(List<Process> processes, String filename) throws IOException {
27     try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
28         for (Process process : processes) {
29             writer.write(process.getPid() + "," + process.getArrivalTime() + "," +
30                         process.getBurstTime() + "," + process.getPriority() + "\n");
31         }
32     } catch (IOException e) {
33         System.out.println("Error saving to file: " + e.getMessage());
34     }
35 }
36 }
37

```

PROCESS CLASS

```

1  public class Process {
2      private int pid; // Process ID
3      private int arrivalTime;
4      private int burstTime;
5      private int remainingBurstTime;
6      private int remainingTime; // For algorithms like Round Robin
7      private int priority;
8      private int waitingTime;
9      private int turnaroundTime;
10     private int completionTime;
11
12     public Process(int pid, int arrivalTime, int burstTime, int priority) {
13         this.pid = pid;
14         this.arrivalTime = arrivalTime;
15         this.burstTime = burstTime;
16         this.remainingTime = burstTime; // Initial remaining time is the same as burst time
17         this.priority = priority;
18     }
19
20     // Getters and Setters
21
22     public int getPid() {
23         return pid;
24     }
25
26     public void setPid(int pid) {
27         this.pid = pid;
28     }
29
30     public int getArrivalTime() {
31     }
32
33     public void setArrivalTime(int arrivalTime) {
34         this.arrivalTime = arrivalTime;
35     }
36
37     public int getBurstTime() {
38         return burstTime;
39     }
40
41     public void setBurstTime(int burstTime) {
42         this.burstTime = burstTime;
43     }
44
45

```

```
46     public int getRemainingBurstTime() {
47         return remainingBurstTime;
48     }
49
50     public void setRemainingBurstTime(int remainingBurstTime) {
51         this.remainingBurstTime = remainingBurstTime;
52     }
53
54     public int getRemainingTime() {
55         return remainingTime;
56     }
57
58     public void setRemainingTime(int remainingTime) {
59         this.remainingTime = remainingTime;
60     }
61
62     public int getPriority() {
63         return priority;
64     }
65
66     public void setPriority(int priority) {
67         this.priority = priority;
68     }
69
70     public int getWaitingTime() {
71         return waitingTime;
72     }
73 }
```

```
70     public int getWaitingTime() {
71         return waitingTime;
72     }
73
74     public void setWaitingTime(int waitingTime) {
75         this.waitingTime = waitingTime;
76     }
77
78     public int getTurnaroundTime() {
79         return turnaroundTime;
80     }
81
82     public void setTurnaroundTime(int turnaroundTime) {
83         this.turnaroundTime = turnaroundTime;
84     }
85
86     public int getCompletionTime() {
87         return completionTime;
88     }
89
90     public void setCompletionTime(int completionTime) {
91         this.completionTime = completionTime;
92     }
93
94 }
```

. 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): $TAT = CT - ARRIVAL\ TIME$
- WAITING TIME (WT): $WT = TAT - BURST\ TIME$

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          89           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             28           205          137          117
2   72             2            36            -36          -38
7   75             10           155          89            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 | P10 | P4 | P3 | P10 | P7 | 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.