12/14/2024

OEL Submission

Data structure & algorithm

Group Members:
Danish Malhi (63888)
Eisha Anjum (63927)
Faizan Ahmed (64126)

CPU Scheduling Simulator

Open-Ended Lab Project Report

1. Introduction

This project implements a **CPU Scheduling Simulator**, designed to simulate various CPU scheduling algorithms used in operating systems. The simulator supports user input for processes and implements algorithms such as **First-Come-First-Serve (FCFS)**, **Shortest Job First (SJF)**, **Priority Scheduling**, and **Round Robin (RR)**. The aim is to understand the functioning of these algorithms and analyze their performance with respect to waiting time and turnaround time.

2. Problem Definition

The goal of this project is to design a program that allows users to simulate CPU scheduling for a set of processes, visualize the scheduling results, and calculate performance metrics such as average waiting time and average turnaround time.

- **Why this problem?**
  Efficient CPU scheduling is critical for optimizing resource utilization and ensuring fairness in multi-programming environments. This simulator helps explore how various algorithms perform under different conditions.
- **Chosen Data Structures:**
  - **List:** To store and manipulate process information.
  - **Priority Queue:** For efficient process selection in SJF and Priority Scheduling.
  - **Queue:** To implement Round Robin scheduling.

3. Key Features and Algorithms

1. **First-Come-First-Serve (FCFS):**
   - Processes are scheduled in the order of their arrival.
   - Simple but non-optimal, may cause the "convoy effect."
2. **Shortest Job First (SJF):**
   - Processes with the smallest burst time are executed first.
   - Optimal for minimizing average waiting time, but non-preemptive.
3. **Priority Scheduling:**
   - Processes with the highest priority are executed first.
   - Implements tie-breaking based on arrival time.
4. **Round Robin (RR):**
   - Uses a time quantum to ensure fair CPU allocation.
   - Suited for time-sharing systems, minimizing starvation.

4. Data Structure Design

- **Process Class:**
  Represents a process with attributes for process ID, arrival time, burst time, priority, and calculated metrics (completion time, waiting time, and turnaround time).
- **Algorithm Efficiency:**
  - Sorting is performed using the `Comparator` interface.
  - Priority Queue operations (O(log n)) are used for dynamic process selection.

5. Implementation Overview

Programming Language: Java
The project uses modular Java code to ensure readability and reusability. The main functionalities include:

1. **Process Input:** Users input process details interactively.
2. **Algorithm Selection:** Users choose the desired scheduling algorithm.
3. **Result Display:** Outputs include Gantt chart visualization and average time metrics.

Code Highlights:

- Dynamic handling of process arrival using queues.
- Reusability through process reset for each algorithm.
- Detailed output for results and performance analysis.

6. Challenges Faced

1. Implementing dynamic process handling in Round Robin while maintaining the Gantt Chart.
2. Ensuring fair priority assignment and avoiding starvation in Priority Scheduling.
3. Testing edge cases, such as simultaneous arrival of processes.

7. Results and Metrics

The simulator calculates and displays:

1. Completion Time
2. Turnaround Time
3. Waiting Time
4. Average Waiting Time
5. Average Turnaround Time

Sample Output:

PID	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
1	0	4	2	4	4	0

8. Learning Outcomes

1. Gained hands-on experience with designing and implementing data structures.
2. Learned optimization techniques for scheduling algorithms.
3. Enhanced debugging and modular programming skills.

9. Conclusion

The CPU Scheduling Simulator successfully demonstrates the implementation and analysis of scheduling algorithms. It highlights the trade-offs between different approaches and provides an educational tool for understanding CPU scheduling concepts.

10. Future Enhancements

- Incorporate **real-time process scheduling** for dynamic workloads.
- Add graphical visualization for Gantt charts.
- Support for additional scheduling algorithms, such as Multi-Level Feedback Queue.

11. Here is the code:

```
package oel2;

import java.util.*;

class Process {
    int pid; // Process ID
    int arrivalTime;
    int burstTime;
    int priority;
    int completionTime;
    int waitingTime;
    int turnaroundTime;
    int remainingTime; // For preemptive scheduling

    public Process(int pid, int arrivalTime, int burstTime, int priority) {
        this.pid = pid;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.priority = priority;
        this.remainingTime = burstTime;
    }
}

public class CPUScheduling {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Process> processes = new ArrayList<>();

        while (true) {
            System.out.println("================ CPU Scheduling Simulator ================");
            System.out.println("1. Enter Process Details");
            System.out.println("2. Select Scheduling Algorithm");
            System.out.println("3. Exit");
            System.out.print("Choose an option: ");
            int mainChoice = scanner.nextInt();

            if (mainChoice == 1) {
                processes.clear();
                System.out.print("Enter the number of processes: ");
                int n = scanner.nextInt();
                for (int i = 0; i < n; i++) {
```

```
                    System.out.println("Enter details for Process " + (i + 1) + " (Arrival Time, Burst
Time, Priority):");
                    System.out.print("Arrival Time: ");
                    int arrivalTime = scanner.nextInt();
                    System.out.print("Burst Time: ");
                    int burstTime = scanner.nextInt();
                    System.out.print("Priority: ");
                    int priority = scanner.nextInt();
                    processes.add(new Process(i + 1, arrivalTime, burstTime, priority));
                }
                System.out.println("\nProcess details saved successfully!\n");
            } else if (mainChoice == 2) {
                if (processes.isEmpty()) {
                    System.out.println("No processes found. Please enter process details first.\n");
                    continue;
                }

                while (true) {
                    System.out.println("\n================ Scheduling Algorithms ================");
                    System.out.println("1. First-Come-First-Serve (FCFS)");
                    System.out.println("2. Shortest Job First (SJF)");
                    System.out.println("3. Priority Scheduling");
                    System.out.println("4. Round Robin (RR)");
                    System.out.println("5. Back to Main Menu");
                    System.out.print("Choose an option: ");
                    int choice = scanner.nextInt();

                    List<Process> processesCopy = resetProcesses(processes);  // Clone the process list

                    switch (choice) {
                        case 1:
                            fcfs(processesCopy);
                            break;
                        case 2:
                            sjf(processesCopy);
                            break;
                        case 3:
                            priorityScheduling(processesCopy);
                            break;
                        case 4:
                            System.out.print("Enter Time Quantum: ");
                            int timeQuantum = scanner.nextInt();
                            roundRobin(processesCopy, timeQuantum);
                            break;
                        case 5:
                            System.out.println("Returning to main menu...\n");
                            break;
                        default:
                            System.out.println("Invalid choice. Please try again.");
                    }

                    if (choice == 5) break;
                }
            } else if (mainChoice == 3) {
                System.out.println("Exiting the simulator. Thank you!");
                break;
            } else {
                System.out.println("Invalid choice. Please try again.\n");
            }
        }

        scanner.close();
    }

    // Reset process list before each algorithm run
    public static List<Process> resetProcesses(List<Process> originalProcesses) {
        List<Process> newProcesses = new ArrayList<>();
        for (Process p : originalProcesses) {
            newProcesses.add(new Process(p.pid, p.arrivalTime, p.burstTime, p.priority));
```

```
        }
        return newProcesses;
    }

    // FCFS Scheduling
    public static void fcfs(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        int currentTime = 0;

        for (Process p : processes) {
            if (currentTime < p.arrivalTime) {
                currentTime = p.arrivalTime;
            }
            p.completionTime = currentTime + p.burstTime;
            p.turnaroundTime = p.completionTime - p.arrivalTime;
            p.waitingTime = p.turnaroundTime - p.burstTime;
            currentTime += p.burstTime;
        }

        printResults(processes, "FCFS");
    }

    // SJF Scheduling
    public static void sjf(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        PriorityQueue<Process> pq = new PriorityQueue<>(Comparator.comparingInt(p -> p.burstTime));
        int currentTime = 0;
        int completed = 0;

        while (completed < processes.size()) {
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0 && !pq.contains(p)) {
                    pq.add(p);
                }
            }

            if (!pq.isEmpty()) {
                Process current = pq.poll();
                currentTime += current.burstTime;
                current.completionTime = currentTime;
                current.turnaroundTime = current.completionTime - current.arrivalTime;
                current.waitingTime = current.turnaroundTime - current.burstTime;
                current.remainingTime = 0;
                completed++;
            } else {
                currentTime++;
            }
        }

        printResults(processes, "SJF");
    }

    // Priority Scheduling
    public static void priorityScheduling(List<Process> processes) {
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));
        PriorityQueue<Process> pq = new PriorityQueue<>((p1, p2) -> {
            if (p1.priority == p2.priority) {
                return Integer.compare(p1.arrivalTime, p2.arrivalTime);
            }
            return Integer.compare(p1.priority, p2.priority);
        });
        int currentTime = 0;
        int completed = 0;

        while (completed < processes.size()) {
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0 && !pq.contains(p)) {
                    pq.add(p);
                }
```

```
            }

            if (!pq.isEmpty()) {
                Process current = pq.poll();
                currentTime += current.burstTime;
                current.completionTime = currentTime;
                current.turnaroundTime = current.completionTime - current.arrivalTime;
                current.waitingTime = current.turnaroundTime - current.burstTime;
                current.remainingTime = 0;
                completed++;
            } else {
                currentTime++;
            }
        }

        printResults(processes, "Priority Scheduling");
    }

    // Round Robin Scheduling
    public static void roundRobin(List<Process> processes, int timeQuantum) {
        Queue<Process> queue = new LinkedList<>();
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime)); // Sort processes by arrival time

        int currentTime = 0;
        int completed = 0;
        List<String> ganttChart = new ArrayList<>();
        boolean[] isInQueue = new boolean[processes.size() + 1]; // Track if a process is already in queue

        // Initially add processes that have arrived at time 0
        for (Process p : processes) {
            if (p.arrivalTime <= currentTime) {
                queue.add(p);
                isInQueue[p.pid] = true;
            }
        }

        // Loop until all processes are completed
        while (completed < processes.size()) {
            if (queue.isEmpty()) {
                // If the queue is empty, move time forward to the next process arrival
                currentTime++;
                for (Process p : processes) {
                    if (p.arrivalTime <= currentTime && p.remainingTime > 0 && !isInQueue[p.pid]) {
                        queue.add(p);
                        isInQueue[p.pid] = true;
                    }
                }
                continue;
            }

            Process current = queue.poll(); // Pick the first process in the queue
            ganttChart.add("P" + current.pid);

            if (current.remainingTime > timeQuantum) {
                currentTime += timeQuantum;
                current.remainingTime -= timeQuantum;
            } else {
                currentTime += current.remainingTime;
                current.remainingTime = 0;
                current.completionTime = currentTime;
                current.turnaroundTime = current.completionTime - current.arrivalTime;
                current.waitingTime = current.turnaroundTime - current.burstTime;
                completed++;
            }

            // Check for newly arrived processes and add them to the queue
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0 && !isInQueue[p.pid]) {
                    queue.add(p);
```

```
                    isInQueue[p.pid] = true;
                }
            }

            // Re-add the current process to the queue if it is not completed
            if (current.remainingTime > 0) {
                queue.add(current);
            }
        }

        printResults(processes, "Round Robin");
        System.out.println("\nGantt Chart: " + String.join(" -> ", ganttChart));
    }

    // Print Results
    public static void printResults(List<Process> processes, String algorithmName) {
        System.out.println("\nResults for " + algorithmName + " Scheduling:");
        System.out.println("PID\tArrival\tBurst\tPriority\tCompletion\tTurnaround\tWaiting");

        for (Process p : processes) {
            System.out.printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p.pid, p.arrivalTime, p.burstTime,
p.priority, p.completionTime, p.turnaroundTime, p.waitingTime);
        }

        double avgWaitingTime = processes.stream().mapToInt(p -> p.waitingTime).average().orElse(0.0);
        double avgTurnaroundTime = processes.stream().mapToInt(p ->
p.turnaroundTime).average().orElse(0.0);

        System.out.printf("\nAverage Waiting Time: %.2f\n", avgWaitingTime);
        System.out.printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);
    }
}
```