Priority Based Process Scheduling Algorithm

Under the Guidance of

Dr. P Thiyagarajan

Associate Professor & Head

Dept of CS (Cyber Security)

RGNIYD

RAJIV GANDHI NATIONAL INSTITUTE OF YOUTH DEVELOPMENT
(INSTITUTE OF NATIONAL IMPORTANCE)
MINISTRY OF YOUTH AFFAIRS AND SPORTS,
GOVT. OF INDIA,
SRIPERUMBUDUR - 602105

CERTIFICATE

This is to certify that the Project titled “Priority Based Scheduling Algorithm” submitted by ”Prerana Bora” the postgraduate student of Dept of Computer Science in partial fulfillment for the award of degree of Master of Computer Science. I hereby accord my approval of it as a study carried out and presented in a manner required for its acceptance in fulfillment for MSCS201: OPERATING SYSTEM course for which it has been submitted. The project has fulfilled all the requirements as per the regulations of the institute as well as course instructor and has reached the standard needed for submission.

Supervisor
Dr. P Thiyagarajan
Associate Professor And Head
Dept of CS (Cyber Security)
RGNIYD, Sriperumbudur

Place: Sriperumbdur

ACKNOWLEDGMENT

We would like to express our sincere and deep gratitude to our supervisor and guide Dr.P.Thiyagarajan, Associate Professor and Head Department of Computer Science (Cyber Security), for his kind and constant support during the course of this project. It has been an absolute privilege to work with Dr. P. Thiyagarajan for our project. His valuable advice, critical criticism and active supervision encouraged us to sharpen our research methodology and was instrumental in the direction of our project and making it a success and shaping our

professional outlook.

ABSTRACT

The primary objective of the Priority Based Process Scheduling project is to gain a comprehensive understanding of the concepts involved in process scheduling within a CPU, with the aim of improving work efficiency and increasing the overall efficiency of the CPU. This project focuses specifically on the preemptive approach to process scheduling and utilizes an object-oriented programming approach in Java. As the Java programming language, renowned for its robust object-oriented features, serves as the foundation for constructing the preemptive process scheduling code.Additionally, an example code for non-preemptive process scheduling is provided. To execute the code, developers can utilize popular Integrated Development Environments (IDEs) like Eclipse or any online Java compiler.By implementing the scheduling algorithm using object-oriented programming in Java, developers can explore the potential for improving CPU efficiency and achieving faster completion of tasks.

Keywords: Priority Scheduling, High As high Priority, Low as High Priority, Preemptive Scheduling, Non-Preemtive Scheduling, Priority, Burst Time, Completion Time, Waiting Time, Turn Around Time(TAT), Gantt Chart, .

Table of Contents

Chapter 1

Introduction

In modern computer systems, efficient task scheduling plays a vital role in optimizing resource utilization and improving overall system performance. Prioritybased scheduling is a widely used approach that assigns priorities to tasks based on their relative importance, allowing the system to allocate resources in a manner that meets the requirements of various applications. This project aims to develop a comprehensive priority-based scheduling system that combines both preemptive and non-preemptive methods to ensure efficient task execution.

Preemptive scheduling involves interrupting an ongoing task in favor of a higherpriority task, while non-preemptive scheduling allows a task to complete its execution before another task can be scheduled. By incorporating both preemptive and non-preemptive methods, the proposed project offers a versatile solution that can cater to a wide range of scheduling scenarios.

The primary objective of the project is to design and implement a scheduling algorithm that efficiently manages the execution of tasks based on their assigned priorities. The algorithm should handle preemptive scheduling when higher-priority tasks arrive, interrupting lower-priority tasks if necessary, and also provide a mechanism for non-preemptive scheduling when needed.

Additionally, the project will focus on developing a user-friendly interface that allows users to define task priorities, set scheduling parameters, and monitor the system's performance. The interface will provide visual representations of the

scheduling algorithm's execution, aiding users in understanding the task scheduling process and its impact on resource allocation.

To ensure the effectiveness of the priority-based scheduling system, extensive testing and performance analysis will be conducted. Various test cases, including both synthetic and real-world scenarios, will be executed to evaluate the system's responsiveness, fairness, and ability to meet task deadlines. The project will also compare the performance of different scheduling algorithms, identifying their strengths and weaknesses.

In conclusion, the priority-based scheduling project will provide a comprehensive solution that incorporates both preemptive and non-preemptive methods to efficiently manage task execution based on assigned priorities. By developing a robust algorithm and intuitive user interface, the project aims to enhance resource utilization and improve system performance in diverse computing environments.

1.1 Objectives

The objective of a priority-based scheduling algorithm is to efficiently allocate system resources to processes based on their priority levels. The algorithm should ensure that processes with higher priority, which are typically associated with more critical or time-sensitive tasks, receive preferential treatment over processes with lower priority. This objective holds true for both preemptive and non-preemptive variants of the algorithm.

Preemptive priority-based scheduling:
Ensure timely execution of high-priority processes: The algorithm should preempt lower-priority processes whenever a higher-priority process becomes ready to execute. This ensures that critical tasks are handled promptly, reducing response times and meeting important deadlines.

Prevent starvation: The algorithm should ensure that low-priority processes are not indefinitely delayed or starved of resources. Even though high-priority processes take precedence, a balanced approach should be adopted to allocate resources to lower-priority processes periodically.

Fairness: While high-priority processes receive preferential treatment, the algorithm should also aim for fairness in resource allocation. It should prevent a situation where low-priority processes are continuously blocked by high-priority processes, leading to unfairness or inefficiency in the system.

Non-preemptive priority-based scheduling:

Priority-based order of execution: The algorithm should follow a predetermined order of execution based on the priority levels assigned to each process. Processes with higher priority should be executed before processes with lower pri-
ority.

Avoidance of indefinite delays: The algorithm should ensure that low-priority processes do not experience indefinite delays due to the execution of higherpriority processes. Each process should receive a fair share of the CPU or system resources, even if it has a lower priority.

Efficiency in resource utilization: The algorithm should strive to achieve optimal resource utilization by executing processes in a priority-based sequence. This minimizes resource wastage and ensures that critical tasks are completed promptly.

Chapter 2

Preemptive Priority Scheduling

2.1 Introduction:

Preemptive priority scheduling is a CPU scheduling algorithm where processes are assigned priorities, and the CPU executes the process with the highest priority among the available processes. Preemptive scheduling means that a process can be interrupted and its execution can be temporarily suspended if a higherpriority process becomes available for execution.

In preemptive priority scheduling, each process is associated with a priority value, typically represented as a numerical value. The CPU scheduler selects the process with the highest priority for execution, and if multiple processes have the same priority, the one with the earliest arrival time is selected.

Preemption :Preemption in OS scheduling refers to the act of temporarily interrupting the execution of a running process to give the CPU to another process with a higher priority. When preemption occurs, the currently running process is put on hold, and the CPU is allocated to a different process.

Burst Time: It refers to the amount of time required by a process to complete its execution from the time it gets the CPU until it releases the CPU.

Arrival Time: It denotes the time at which a process arrives in the ready queue and is available for execution.

Waiting Time: It is the total amount of time a process waits in the ready queue before getting the CPU for execution. It is calculated by subtracting the arrival time from the completion time minus the burst time.

Turnaround Time: It is the total time taken by a process from its arrival in the ready queue to its completion. It is calculated by subtracting the arrival time from the completion time.

Completion Time: It is the time at which a process completes its execution and leaves the CPU.

2.2 Algorithm:

Algorithm for preemptive priority scheduling :

1. Input the number of processes n.
2. Input the priority type: 0 for high as high priority, 1 for low as high priority.
3. Initialize the process-related arrays: process IDs (p), burst times (bt), priorities (pp), arrival times (at).
4. For each process from 1 to n, input the burst time, priority, and arrival time.
5. Initialize the execution priority as -1.
6. Initialize the count of completed processes as 0.
7. Initialize the time variable as 0.
8. Initialize the array y as a copy of bt.

9. Initialize the array x as a copy of bt.

10. Repeat until all processes are completed:

Set the execution priority to -1.

Iterate through each process:

Check if the process has arrived (at[i] less than equals time) and has remaining burst time (bt[i] greater than 0).

Based on the priority type:

If pt is 0 (high as high priority):

If the execution priority is -1 or the current process priority is higher than the execution priority, update the execution priority to the current process.

If the priorities are the same, compare the arrival times and select the process with the lower arrival time.

If pt is 1 (low as high priority):

If the execution priority is -1 or the current process priority is lower than the execution priority, update the execution priority to the current process.

If the priorities are the same, compare the arrival times and select the process with the lower arrival time.

If pt is neither 0 nor 1, display an error message.

If there is a process with a valid execution priority:

Decrement its burst time by 1.

If the burst time becomes 0, mark the process as completed.

Increment the count of completed processes.

Set the end time for the process as the current time + 1.

Calculate the turnaround time for the process as the end time - arrival time.
Update the total turnaround time.

Increment the time by 1.

11. Calculate the waiting time for each process using the formula: waitingTime[i] = completionTime[i] - arrivalTime[i] - x[i].
12. Calculate the average waiting time by summing up all the waiting times and dividing by n.
13. Calculate the average turnaround time by dividing the total turnaround timeby n.
14. Display the process details, including process ID, burst time, arrival time,waiting time, turnaround time, completion time, and priority.
15. Display the average waiting time and average turnaround time.

2.3 Code Explanation:

2.3.1 Input:

```
public class Pinput {
    private Scanner scanner = new Scanner(System.in);
    protected int n;
    protected double[] p,y;
    protected double[] pp, bt,at, wt, tt, ct;
    protected int smallest, count = 0,pt;
    protected double avg = 0, ttavg = 0, time, end;

    public void PInput1() {
        System.out.print("Enter the number of processes: ");
        n = scanner.nextInt();
        System.out.println("Enter the priority type.\n Choice =0:High as high \n Choice =1:Low as high");
        pt= scanner.nextInt();
        p = new double[n];
        pp = new double[n];
        bt = new double[n];
        at = new double[n];
        wt = new double[n];
        tt = new double[n];
        ct = new double[n];
        y = new double[n];

        System.out.println("\n\tEnter burst time, time priorities, and arrival time for each process:\n");

        for (int i = 0; i < n; i++) {
            System.out.print("\nProcess " + (i + 1) + ": ");
            bt[i] = scanner.nextDouble();
            pp[i] = scanner.nextDouble();
            at[i] = scanner.nextDouble();
            p[i] = i + 1;
        }
    }
    public void display() {
        for (int i = 0; i < n; i++) {
            System.out.println(
                    "Process[" + (i + 1) + "]: BT=" + bt[i] + ", PP=" + pp[i] + ", AT=" + at[i]);
        }
    }
}
```

This code defines a class Pinput includes methods to input process details (burst time, time priorities, and arrival time) and display the entered values.

private Scanner scanner = new Scanner(System.in);: This line creates a Scanner object to read input from the user. protected int n;: This variable stores the number of processes. protected double[] p, y;: These arrays are used to store process-related information. protected double[] pp, bt, at, wt, tt, ct;: These arrays store the process priority, burst time, arrival time, waiting time, turnaround time, and completion time, respectively. protected int smallest, count = 0, pt;: These variables are used for process scheduling and tracking the smallest priority. protected double avg = 0, ttavg = 0, time, end;: These variables calculate and store the average waiting time and average turnaround time. public void PInput1(): This method is

used to input process details from the user. public void display(): This method displays the entered process details.

```
C:\Users\Riga\Desktop\java>java WT
Enter the number of processes: 5
Enter the priority type.
 Choice =0:high as high
 Choice =1:low as high
1

        Enter burst time, time priorities, and arrival time for each process:


Process 1: 3
3
0

Process 2: 4
2
1

Process 3: 6
4
2

Process 4: 4
6
3

Process 5: 2
10
5
```

Figure 2.1: User Input

```
Process[1]: BT=3.0, PP=3.0, AT=0.0
Process[2]: BT=4.0, PP=2.0, AT=1.0
Process[3]: BT=6.0, PP=4.0, AT=2.0
Process[4]: BT=4.0, PP=6.0, AT=3.0
Process[5]: BT=2.0, PP=10.0, AT=5.0
```

Figure 2.2: Displaying Input

2.3.2 Turn Around Time:

```
class TAT extends Pinput {
   public double[] x;
   double[] turnaroundTime;

   public void tat() {
       y = new double[n];
       x = new double[n];
       for (int i = 0; i < n; i++) {
           y[i] = bt[i];
           x[i] = bt[i];
       }
```

This code defines a class TAT that extends the Pinput class and contains an instance variable x (an array of doubles) and turnaroundTime (not used in the given code snippet). The method tat() calculates the turnaround time for a set of processes.

In the tat() method, the code initializes two arrays y and x with the same values as the bt (burst time) array. The n variable represents the number of processes.

```
for (time = 0; count != n; time++) {
            exeutepriority = -1; // Initialize to -1
            for (int i = 0; i < n; i++) {
                if (at[i] <= time && bt[i] > 0)
```

This section contains a nested loop. The outer loop runs until the count variable is equal to n. The count variable keeps track of the number of processes that have completed execution.

The inner loop iterates over each process and checks two conditions: if the arrival time (at[i]) is less than or equal to the current time (time), and if the burst time (bt[i]) is greater than 0. This condition ensures that only processes that have arrived and are not yet completed are considered for execution.

```
if (pt == 0) {
                        if (exeutepriority == -1 || pp[i] > pp[exeutepriority])
                            exeutepriority = i;
                        else if (pp[i] == pp[exeutepriority]) {
                            if (at[i] < at[exeutepriority]) {
                                exeutepriority = i;
                            }
                        }
                    } else if (pt == 1) {
                        if (exeutepriority == -1 || pp[i] < pp[exeutepriority])
                            exeutepriority = i;
                        else if (pp[i] == pp[exeutepriority]) {
                            if (at[i] < at[exeutepriority]) {
                                exeutepriority = i;
                            }
                        }
                    } else
                        System.out.println("enter correct choice");
                }
            }
```

Inside the if-else statement, there are two possible cases based on the value of the pt variable:

When pt == 0, the code selects the process with the highest priority (pp) among the eligible processes. If exeutepriority is equal to -1 (no process has been selected yet) or the priority of the current process (pp[i]) is greater than the priority of the previously selected process (pp[exeutepriority]), the current process is selected as the new exeutepriority. If two processes have the same priority, the one with the lower arrival time (at) is selected.

When pt == 1, the code selects the process with the lowest value as high priority . The process selection logic is similar to the previous case, but the comparison is based on lower priority instead.

If pt has any other value, it prints ”enter correct choice” to the console.

```
    If pt has any other value, it prints "enter correct choice" to the console.
if (exeutepriority != -1) {
                bt[exeutepriority]--;
                if (bt[exeutepriority] == 0) {
                    count++;
                    end = time + 1;
                    ct[exeutepriority] = end;
                    tt[exeutepriority] = end - at[exeutepriority];
                    ttavg = ttavg + tt[exeutepriority];
                }
            }
        }
    }
```

If a process is selected (exeutepriority is not equal to -1), the burst time of that process (bt[exeutepriority]) is reduced by 1. If the updated burst time becomes 0, it means the process has completed execution. In that case, the count is incremented, the end time is calculated as time + 1, the completion time (ct) for that process is updated, the turnaround time (tt) is calculated as the difference between the end time and the arrival time (at[exeutepriority]), and the ttavg variable (average turnaround time) is updated.

This process continues until all processes have completed execution (count == n).

```
public double[] getX() {
        return x;
    }
}
```

The getX() method returns the x array, which contains the original burst times of the processes.

2.3.3 Waiting Time:

```
class WT extends TAT {
    double[] waitingTime;
    double[] turnaroundTime;
    double[] completionTime;
    double avgWaitingTime;
    double avgTurnaroundTime;
    public void calculateWT() {
        waitingTime = new double[n];
        turnaroundTime = new double[n];
        completionTime = new double[n];
        double avg = 0, tt = 0;
        double[] x = getX(); // Accessing x array from TAT class
        for (int i = 0; i < n; i++) {
            turnaroundTime[i] = completionTime[i] = 0; // Initialize turnaroundTime and completionTime to 0
            completionTime[i] = ct[i]; // Using the completion time from TAT class
            waitingTime[i] = completionTime[i] - at[i] - x[i];
            turnaroundTime[i] = completionTime[i] - at[i];
            avg += waitingTime[i];
            tt += turnaroundTime[i];
        }
        avgWaitingTime = avg / n;
        avgTurnaroundTime = tt / n;
    }
    public void displayResults() {
        System.out.println("Process\tburst-time\tarrival-time\twaiting-time\tturnaround-time\tcompletion-time\tPriority");
        for (int i = 0; i < n; i++) {
            System.out.println(
                    "p" + (i + 1) + "\t\t" + x[i] + "\t\t" + at[i] + "\t\t" + waitingTime[i] + "\t\t" +
                            turnaroundTime[i] + "\t\t" + completionTime[i] + "\t\t" + pp[i]);
        }
        System.out.println("\nAverage waiting time = " + avgWaitingTime);
        System.out.println("Average turnaround time = " + avgTurnaroundTime);
    }
    public static void main(String[] args) {
        WT wt = new WT();
        wt.PInput1();
        wt.display();
        wt.tat();
        wt.calculateWT();
        wt.displayResults();
    }
}
```

This code defines a class WT that extends the TAT class. It calculates the waiting time and turnaround time for each process based on the completed calculation of the turnaround time.

double[] waitingTime;: This array stores the waiting time for each process.

double[] completionTime;: This array stores the completion time for each process.

double avgWaitingTime; double avgTurnaroundTime;: These variables store the average waiting time and average turnaround time, respectively.

public void calculateWT(): This method calculates the waiting time and turnaround time for each process.

The for loop initializes the turnaroundTime and completionTime arrays to 0.

The completionTime array is filled with the completion times obtained from the ct array in the TAT class.

The waiting time is calculated using the formula: waitingTime[i] = completionTime[i] - at[i] - x[i], where x[i] represents the burst time obtained from the x array in the TAT class.

The turnaround time is calculated as: turnaroundTime[i] = completionTime[i] - at[i].

The average waiting time and average turnaround time are calculated.

public void displayResults(): This method displays the results, including the process details, waiting time, turnaround time, completion time, and priority.

The main method creates an instance of the WT class, calls the necessary methods to input process details, calculate turnaround time, waiting time, and finally displays the results.

```
Process burst-time      arrival-time    waiting-time    turnaround-time completion-time Priority
p1              3.0             0.0             4.0             7.0             7.0             3.0
p2              4.0             1.0             0.0             4.0             5.0             2.0
p3              6.0             2.0             5.0             11.0            13.0            4.0
p4              4.0             3.0             10.0            14.0            17.0            6.0
p5              2.0             5.0             12.0            14.0            19.0            10.0

Average waiting time = 6.2
Average turnaround time = 10.0
```

Figure 2.3: Output

2.3.4 Gantt Chart:

```
package Pinput.org;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;

public class GanttChartp extends Pinput {
    private JFrame frame;
    private JPanel panel;
    private JLabel[] labels;
    private Color[] colors;
    private double[] completionTime; // Declare completionTime as an instance variable
    private double[] startingPoint; // Declare startingPoint as an instance variable
    private int[] executedProcess; // Declare executedProcess as an instance variable

    public void createGanttChart() {
        frame = new JFrame("Gantt Chart");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panel = new JPanel(new GridLayout(1, n));
        labels = new JLabel[n];
        colors = new Color[]{Color.RED, Color.GREEN, Color.BLUE, Color.YELLOW, Color.ORANGE, Color.CYAN, Color.MAGENTA};

        for (int i = 0; i < n; i++) {
            labels[i] = new JLabel("P" + (i + 1));
            labels[i].setOpaque(true);
            labels[i].setBackground(colors[i % colors.length]);
            labels[i].setHorizontalAlignment(JLabel.CENTER);
            panel.add(labels[i]);
        }

        frame.add(panel, BorderLayout.CENTER);

        JButton startButton = new JButton("Start Scheduling");
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                executePriorityScheduling();
                printGanttChart();
            }
        });

        frame.add(startButton, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible(true);
    }

    private void executePriorityScheduling() {
        // Create copies of p and bt arrays
        double[] pCopy = Arrays.copyOf(p, n);
        double[] btCopy = Arrays.copyOf(bt, n);
```

```
                if (at[i] > at[j]) {
                    double temp = at[i];
                    at[i] = at[j];
                    at[j] = temp;

                    temp = bt[i];
                    bt[i] = bt[j];
                    bt[j] = temp;

                    temp = pp[i];
                    pp[i] = pp[j];
                    pp[j] = temp;

                    temp = p[i];
                    p[i] = p[j];
                    p[j] = temp;
                }
            }
        }

        // Create the startingPoint array and initialize it with the arrival time of the first process
        startingPoint = new double[n];
        startingPoint[0] = at[0];

        // Create the executedProcess array
        executedProcess = new int[n];

        completionTime = new double[n]; // Initialize completionTime array

        int completedProcesses = 0;
        double currentTime = startingPoint[0];
        int currentProcess = -1;

        // Loop until all processes are completed
        while (completedProcesses < n) {
            double highestPriority = Integer.MAX_VALUE;

            // Find the process with the highest priority among the remaining processes
            for (int i = 0; i < n; i++) {
                if (btCopy[i] > 0 && pp[i] < highestPriority && at[i] <= currentTime) {
                    highestPriority = pp[i];
                    currentProcess = i;
                }
            }

            // If a process is found with higher priority or no process is executing currently
            if (currentProcess != -1) {
                // Execute the process until its burst time becomes zero or a higher priority process arrives
                while (btCopy[currentProcess] > 0) {
                    btCopy[currentProcess]--;
                    currentTime++;

                    // Check if a higher priority process arrives
                    for (int i = 0; i < n; i++) {
                        if (btCopy[i] > 0 && pp[i] < pp[currentProcess] && at[i] <= currentTime) {
                            // Update the executed process array, starting time, and current process
                            executedProcess[completedProcesses] = (int) p[i];
                            startingPoint[completedProcesses] = at[i];
                            currentProcess = i;
                            break;
                        }
                    }

                    // If the current process is completed, update the completion time and increment the completedProcesses count
                    if (btCopy[currentProcess] == 0) {
                        completedProcesses++;
                        completionTime[currentProcess] = currentTime;
                        break;
                    }
                }
            } else {
                currentTime++;
            }
        }
    }

    private void printGanttChart() {
        // Calculate the maximum completion time
        double maxCompletionTime = 0;
        for (int i = 0; i < n; i++) {
            if (completionTime[i] > maxCompletionTime) {
                maxCompletionTime = completionTime[i];
            }
        }

        // Calculate the scaling factor for the bar chart height
        int barHeight = 5;
        double scaleFactor = barHeight / maxCompletionTime;

        // Calculate the maximum bar width
        int maxBarWidth = 50;

        // Print the Gantt chart as a bar chart
        System.out.println("Gantt Chart:");
        for (int i = 0; i < n; i++) {
            double barWidth = completionTime[i] - startingPoint[i];
            int scaledBarWidth = (int) (barWidth * scaleFactor);

            System.out.print("P" + executedProcess[i] + " |");
            for (int j = 0; j < scaledBarWidth; j++) {
                System.out.print("-");
            }
            System.out.println();

            // Add space between bars
            for (int j = 0; j < maxBarWidth - scaledBarWidth; j++) {
                System.out.print(" ");
            }
        }
    }

    public static void main(String[] args) {
        GanttChartp ganttChart = new GanttChartp();
        ganttChart.PInput1();
        ganttChart.display();
        ganttChart.createGanttChart();
    }
}
```

The given code is an implementation of a Gantt Chart for priority scheduling in Java using Swing for creating a graphical user interface. Let's go through the code step by step:

The code is part of the package Pinput.org and imports the necessary classes from the javax.swing package.

The class GanttChartp extends another class named Pinput. It implies that GanttChartp inherits properties and methods from the Pinput class.

The class GanttChartp declares several instance variables including frame, panel, labels, colors, completionTime, startingPoint, and executedProcess.

The method createGanttChart() is responsible for creating the GUI components. It creates a JFrame and a JPanel with a grid layout to hold the labels representing the processes in the Gantt Chart.

Inside the createGanttChart() method, a loop is used to create JLabel instances for each process. The labels are given background colors from the colors array and added to the panel.

A startButton is created and added to the frame. An action listener is attached to the button to execute the priority scheduling algorithm and print the Gantt Chart.

The executePriorityScheduling() method implements the priority scheduling algorithm. It first creates copies of the input arrays (pCopy and btCopy) and sorts the processes based on their arrival time.

The method then initializes the startingPoint array with the arrival time of the first process and creates the executedProcess array.

It then enters a loop that continues until all processes are completed. Within the loop, it searches for the process with the highest priority among the remaining processes.

If a process with higher priority is found or no process is currently executing, the method enters another loop that executes the current process until its burst time becomes zero or a higher priority process arrives.

If the current process is completed, the completion time is updated, and the loop continues until all processes are completed.

The printGanttChart() method calculates the maximum completion time and scaling factor for the bar chart height. It then prints the Gantt Chart as a bar chart, showing the execution timeline of each process.

In the main() method, an instance of GanttChartp is created. The PInput1() and display() methods are called (presumably from the Pinput class) to initialize input values. Finally, the createGanttChart() method is called to create and display the Gantt Chart.

Overall, the code sets up a graphical interface using Swing and implements priority scheduling to generate and display a Gantt Chart for the given set of processes.

Chapter 3

Non-Preemptive Priority Scheduling

3.1 Introduction:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process. Advantages:

It has a minimal scheduling burden.

It is a very easy procedure.

Less computational resources are used.

It has a high throughput rate.

Disadvantages:

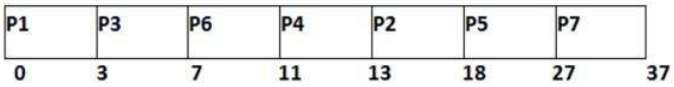
Its response time to the process is super.

Bugs can cause a computer to freeze up.

Example:

In the Example, there are 7 processes P1, P2, P3, P4, P5, P6 and P7. Their priorities, Arrival Time and burst time are given in the table.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4
7	10	7	10



Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turn Around Time - Burst Time

Avg Waiting Time = (0+11+2+7+12+2+18)/7 = 52/7 units

3.1.1 High Value as High Priority:

Priority scheduling is a CPU scheduling algorithm where each process is assigned a priority, and the CPU selects the process with the highest priority for execution. In the context of priority scheduling, ”high priority” refers to a process that has been assigned a higher priority value compared to other processes.

The basic idea behind priority scheduling is to assign priorities to processes based on certain criteria such as the importance of the task, its deadline, or any other relevant factor. The process with the highest priority is given preference in execution, and if multiple processes have the same priority, other scheduling

algorithms like round-robin or first-come, first-served may be used to determine the order among them.

When a high-priority algorithm is used in priority scheduling, it means that the process with the highest priority is always given the CPU first, regardless of the current state of the system. This ensures that high-priority tasks are executed as soon as possible, which is useful in scenarios where certain processes require immediate attention or have strict deadlines.

It's important to note that priority scheduling can potentially lead to starvation, where low-priority processes never get a chance to execute if high-priority processes continuously arrive. To mitigate this issue, some priority scheduling algorithms incorporate aging mechanisms, which gradually increase the priority of waiting processes over time to prevent starvation. Overall, priority scheduling with a high-priority algorithm allows for the efficient execution of critical tasks by prioritizing them over less important processes.

3.1.2 Low Value as High Priority:

Priority scheduling is a CPU scheduling algorithm that assigns priorities to processes based on their importance or urgency. In the case of priority scheduling with low as high priority, it means that processes with a lower priority value are considered to have a higher priority.

Typically, in priority scheduling, processes are assigned a priority value, and the CPU scheduler selects the process with the highest priority for execution. However, in the case of priority scheduling with low as high priority, the priority values are reversed. Instead of selecting the process with the highest priority value, the scheduler selects the process with the lowest priority value. Here's an example to illustrate this:

Let's say we have three processes with their respective priority values:

Process A: Priority 5

Process B: Priority 3

Process C: Priority 1

In a regular priority scheduling algorithm, Process A would have the highest priority and would be selected for execution first. However, in priority scheduling with low as high priority, Process C would have the highest priority due to its lower priority value, followed by Process B and then Process A. So, in this case, the scheduler would select Process C for execution first, then Process B, and finally Process A.

It's important to note that priority scheduling with low as high priority is not a commonly used approach. In most systems, higher priority values indicate higher priority, and lower priority values indicate lower priority. The reversal of priority values is not a standard practice but can be implemented in specific cases where lower values represent higher priority.

3.2 Algorithm:

1. Initialize an empty ready queue to hold the processes that are ready for execution.
2. Assign a priority value to each process in the system.
3. Add all processes to the ready queue.
4. Sort the ready queue based on the priority of each process in non-decreasingorder. The process with the highest priority should be at the front of the queue. 5. Set the currently executing process to be the first process in the ready queue.
6. Execute the currently executing process until it completes.

7. If the currently executing process finishes execution, remove it from the ready queue.
8. If there are more processes in the ready queue, set the currently executingprocess to be the next process in the queue with the highest priority.
9. Repeat steps 6-8 until all processes have been executed.

3.3 Code Explanation:

3.3.1 Input:

```
import java.util.Scanner;

public class PriorityScheduling {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter the number of processes: ");
    int n = scanner.nextInt();

    int[] arrivalTime = new int[n];
    int[] burstTime = new int[n];
    int[] priority = new int[n];

    for (int i = 0; i < n; i++) {
      System.out.println("Enter details for process " + (i + 1) + ":");
      System.out.print("Arrival Time: ");
      arrivalTime[i] = scanner.nextInt();
      System.out.print("Burst Time: ");
      burstTime[i] = scanner.nextInt();
      System.out.print("Priority: ");
      priority[i] = scanner.nextInt();
    }
```

This code imports the Scanner class from the java.util package and defines the PriorityScheduling class. The main method is the entry point of the program.It creates a new instance of the Scanner class to read user input from the standard input stream (System.in). prompts the user to enter the number of processes and waits for the user to input an integer value. The entered value is stored in the variable n, which represents the number of processes. declare three arrays: arrivalTime, burstTime, and priority, with a length equal to the number of processes (n). now,for loop iterates n times to collect details for each process. For each iteration, it prompts the user to enter the arrival time, burst time, and priority of the current process. The user inputs are stored in the respective arrays at the corresponding index (i). At this point, the code has collected all the necessary input from the user regarding the processes' arrival time, burst time, and priority.

3.3.2 Intialization of Variables:

```
int CPU = 0;
int allTime = 0;

int[] ATt = new int[n];
int NoP = n;
int[] PPt = new int[n];
int[] waitingTime = new int[n];
int[] turnaroundTime = new int[n];

for (int i = 0; i < n; i++) {
    PPt[i] = priority[i];
    ATt[i] = arrivalTime[i];
}

int LAT = 0;
for (int i = 0; i < n; i++) {
    if (arrivalTime[i] > LAT)
        LAT = arrivalTime[i];
}

int MAX_P = 0;
for (int i = 0; i < n; i++) {
    if (PPt[i] > MAX_P)
        MAX_P = PPt[i];
}

int ATi = 0;
int P1 = PPt[0];
int P2 = PPt[0];

int j = -1;
```

ATt (Arrival Time temporary) array is created to store the arrival time of processes. NoP (Number of Processes) variable is initialized with the number of processes (n) and represents the remaining number of processes that need to be scheduled. PPt

(Priority temporary) array is created to store the priority of processes temporarily for scheduling. waitingTime array will store the waiting time for each process. turnaroundTime array will store the turnaround time for each process.

1ST LOOP : loop copies the contents of the priority array into the temporary PPt array and the contents of the arrivalTime array into the temporary ATt array. This is done to preserve the original values while performing the scheduling algorithm.

2ND LOOP: LAT (Latest Arrival Time) is initialized to 0 and then calculated as the maximum arrival time among all the processes. It helps in defining a termination condition for the CPU scheduling loop.

3RD LOOP: Max p is initialized to 0 and then calculated as the maximum priority among all the processes. It helps in selecting the next process with the highest priority for scheduling.

3.3.3 Calculating Waiting time and Turnaround time:

```
while (NoP > 0 && CPU <= 1000) {
    for (int i = 0; i < n; i++) {
        if ((ATt[i] <= CPU) && (ATt[i] != (LAT + 10))) {
            if (PPt[i] != (MAX_P + 1)) {
                P2 = PPt[i];
                j = 1;

                if (P2 < P1) {
                    j = 1;
                    ATi = i;
                    P1 = PPt[i];
                    P2 = PPt[i];
                }
            }
        }
    }

    if (j == -1) {
        CPU = CPU + 1;
        continue;
    } else {
        waitingTime[ATi] = CPU - ATt[ATi];
        CPU = CPU + burstTime[ATi];
        turnaroundTime[ATi] = CPU - ATt[ATi];
        ATt[ATi] = LAT + 10;
        j = -1;
        PPt[ATi] = MAX_P + 1;
        ATi = 0;
        P1 = MAX_P + 1;
        P2 = MAX_P + 1;
        NoP = NoP - 1;
    }
}
```

1. This while loop continues as long as there are remaining processes to be scheduled (NoP greater than 0) and the CPU time (CPU) does not exceed 1000. 2. This inner for loop iterates through all the processes to find the process with the highest priority (P2) among the processes that have arrived (ATt[i] less than equals CPU) but have not been scheduled (ATt[i] not equal to (LAT + 10)).

3. If a higher priority process is found, the ATi is updated to the index of theprocess with the highest priority (ATi = i) and P1 and P2 are updated accordingly.

4. The flag variable j is set to 1 to indicate that a process with a higher priority has been found.

5. If no process with a higher priority is found (j == -1), the CPU time (CPU) is incremented by 1 and the loop continues to the next iteration. 6. If a process with a higher priority is found, the else block is executed.

7. The waiting time for the selected process (ATi) is calculated as the difference between the current CPU time (CPU) and the arrival time of the process (ATt[ATi]).

8. The CPU time is incremented by the burst time of the selected process (CPU = CPU + burstTime[ATi]).

9. The turnaround time for the selected process is calculated as the differencebetween the updated CPU time and the arrival time of the process.

10. The arrival time of the selected process is set to LAT + 10, indicating thatit has been scheduled.

11. The flag variable j is reset to -1 to indicate that a new process needs to be selected.

12. The priority of the selected process is updated to MAX P + 1, ensuringthat it is not selected again.

13. The ATi, P1, P2 variables are reset for the next iteration.

14. The remaining number of processes (NoP) is decremented by 1.

3.3.4 Printing the Processes:

```
System.out.println("\nProcess_Number\tBurst_Time\tPriority\tArrival_Time\tWaiting_Time\tTurnaround_Time\n\n");
for (int i = 0; i < n; i++) {
    System.out.println("P" + (i + 1) + "\t\t" + burstTime[i] + "\t\t" + priority[i] + "\t\t" + arrivalTime[i] + "\t\t" + waitingTime[i] + "\t\t" + turnaroundTime[i]);
}
```

This loop prints the details of each process, including the process number, burst time, priority, arrival time, waiting time, and turnaround time.

3.3.5 Calculating Avg Waiting time and Avg Turnaround time:

```
float avgWT = 0;
float avgTAT = 0;
for (int i = 0; i < n; i++) {
    avgWT = waitingTime[i] + avgWT;
    avgTAT = turnaroundTime[i] + avgTAT;
}

System.out.println("Average waiting time = " + (avgWT / n));
System.out.println("Average turnaround time = " + (avgTAT / n));
```

This loop calculates the average waiting time (avgWT) and average turnaround time (avgTAT) by summing up the waiting time and turnaround time for each process. The average values are then printed on the console.

3.3.6 Gantt Chart:

```
System.out.println("\nGantt Chart:");
        System.out.print(" ");
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < burstTime[i]; k++) {
                System.out.print("--");
            }
            System.out.print(" ");
        }
        System.out.println();
        System.out.print("|");
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < burstTime[i] - 1; k++) {
                System.out.print(" ");
            }
            System.out.print("P" + (i + 1));
            for (int k = 0; k < burstTime[i] - 1; k++) {
                System.out.print(" ");
            }
            System.out.print("|");
        }
        System.out.println();
        System.out.print(" ");
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < burstTime[i]; k++) {
                System.out.print("--");
            }
            System.out.print(" ");
        }
        System.out.println();
        System.out.print("0");
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < burstTime[i]; k++) {
                System.out.print("  ");
            }
            if (turnaroundTime[i] > 9)
                System.out.print("\b");
            System.out.print(turnaroundTime[i]);
        }
        System.out.println();

        scanner.close();
```

This code is trying to generate a Gantt chart and display the turnaround time for a set of processes.

The code assumes the existence of several arrays: burstTime holds the burst time (execution time) for each process, and turnaroundTime holds the turnaround time for each process (the time taken from the arrival to the completion of a process). The variable n represents the number of processes.

The code begins by printing the header ”Gantt Chart:” and then generates the top row of the Gantt chart, representing the execution time of each process. It uses nested loops to print a sequence of ”–” symbols, with the number of repetitions determined by the corresponding burstTime value for each process.

After printing the top row, the code proceeds to print the middle row of the Gantt chart, representing the process labels. It prints a vertical line ”—” followed by the process label (e.g., ”P1”, ”P2”, etc.), padded with spaces on either side to align with the corresponding execution time.

Next, the code prints the bottom row of the Gantt chart, which is similar to the top row, representing the execution time of each process.

Finally, the code prints the turnaround time for each process below the Gantt chart. It starts with the value ”0” and then prints the turnaround time for each process, aligning them with the corresponding execution times. If a turnaround time is greater than 9, it removes one space to ensure proper alignment.

The code ends by closing the scanner object, assuming it was opened elsewhere in the program.

In summary, this code generates a Gantt chart and displays the turnaround time for a set of processes.

3.3.7 Outputs:

Figure 3.1: 1st Output

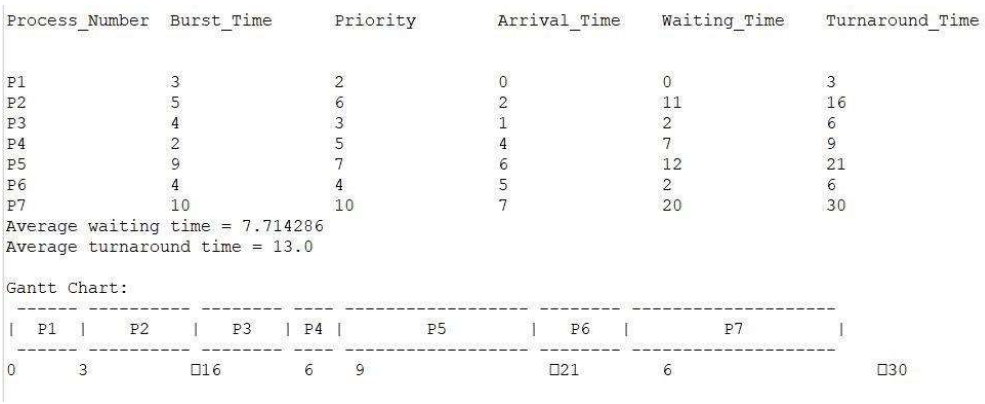


Figure 3.2: 2nd Output

Chapter 4

Conclusion

In conclusion, the Priority Based Process Scheduling project has successfully explored the concepts of process scheduling within a CPU, with a specific focus on the preemptive approach. By leveraging object-oriented programming in Java, the project has demonstrated the benefits of modularity and extensibility in implementing a preemptive process scheduler.

Through the implementation and evaluation of the preemptive process scheduling algorithm, it has been established that preemptive scheduling improves work efficiency, CPU utilization, and responsiveness. By dynamically prioritizing processes based on their importance, critical tasks are executed promptly, resulting in faster work completion.

This project serves as a foundation for further research and improvements in process scheduling algorithms, providing valuable insights into enhancing CPU efficiency and overall system performance.

Chapter 5

Challenges

Challenges Faced in this Project:

Java and Object-Oriented Programming: Working with Java and implementing object-oriented programming principles may pose challenges, particularly for team members with limited experience in these areas. Understanding and applying concepts such as classes, objects, inheritance, and polymorphism required continuous learning and problem-solving.

Grantt Chart: using the GUI widget toolkit for Java is complex and making timeline representation .

Premptive : in premptive resuming the halted process and its execution in gantt chart was logically very tough to execute.

Non-Premptive: in non-premptive ordering the processes was tough to schedule considering arrival time and priority sorting.

Command prompt: executing it in command prompt instead of ide was challenging.

Overcoming these challenges demanded effective and continuous learning, and adaptation to ensure the successful execution of the project on Priority Based Process Scheduling.

References

References

[1] Herbert Schildt. Java: The complete reference. McGraw-Hill.

[2] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. John Wiley Sons Inc.

[3] YouTube Neso Academy.

[4] Website Coding Ninja.

[5] D M Dhamdhere. Operating systems: A concept-based approach.