Synopsis :

This project involves developing a graphical user interface (GUI) to implement and simulate both preemptive and non-preemptive priority scheduling algorithms using Advanced Java. The user can input process details such as process ID, arrival time, burst time, and priority. Preemptive scheduling allows higher-priority processes to interrupt lower-priority ones, while non-preemptive scheduling processes tasks to completion based on priority.

A Gantt chart is dynamically generated to visualize the sequence of process execution. The implementation is done using Java Swing for the GUI, and the project provides detailed process statistics like turnaround and waiting times to compare the performance of both scheduling algorithms.

Microproject – Course Outcome matrix

Course Outcomes :

a. **Install Linux operating system and configure it.**
b. **Use operating system tools to perform various functions.**
c. **Execute process commands for performing process management operations.**
d. **Apply scheduling algorithm to calculate turnaround time and average waiting time.**
e. **Calculate efficiency of different memory management techniques.**
f. **Apply file management techniques.**

Sr. No.	Microproject	CO a	CO b	CO c	CO d	CO e	CO f
1	**Implementation of Priority Scheduling Algorithm**	✓		✓	✓		

Introduction:

Effective process scheduling is essential for managing CPU resources in modern operating systems. **Priority Scheduling** is one of the key algorithms used to manage this process, where tasks are executed based on their priority levels. This project focuses on implementing Priority Scheduling in two variations—**preemptive and non-preemptive**—using Advanced Java. The project's graphical user interface (GUI) provides an interactive environment for simulating and visualizing the impact of each scheduling approach on process execution.

In preemptive priority scheduling, higher-priority tasks can interrupt lower-priority ones, allowing critical processes to execute sooner. In contrast, non-preemptive priority scheduling completes each task in the order of arrival and priority without interruption. This project's GUI, designed with Java Swing, allows users to experience both scheduling methods in action.

The **GUI features** user-friendly input fields where users enter process details, including **Process ID, Arrival Time, Burst Time, and Priority**. These inputs form the basis of scheduling decisions in both algorithms. An algorithm selector **dropdown lets users choose between preemptive and non-preemptive modes**, determining whether interruptions for higher-priority tasks are allowed.

Once processes are added, the **Add Process and Calculate & Show Gantt Chart buttonsenable users to submit processes and view scheduling results**. Detailed process metrics, such as completion time, turnaround time, and waiting time, are displayed in a result area, alongside average turnaround and waiting times to allow for performance comparison. The Gantt Chart Panel visually represents the scheduling process over time, with **color-coded blocks showing each process's execution interval.** This **dynamic chart updates for each scheduling run**, giving a clear, visual representation of the CPU's process flow, including interruptions in preemptive mode

Source Code:

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.util.ArrayList;

import java.util.Comparator;



class Process {

    int pid, arrivalTime, burstTime, priority,
remainingTime, completionTime,
waitingTime, turnaroundTime;

    ArrayList<int[]> executionIntervals; //
Store intervals of execution (startTime,
endTime)



    public Process(int pid, int arrivalTime, int
burstTime, int priority) {

        this.pid = pid;

        this.arrivalTime = arrivalTime;

        this.burstTime = burstTime;

        this.remainingTime = burstTime; // For
preemptive scheduling

        this.priority = priority;

        this.executionIntervals = new
ArrayList<>(); // For preemption tracking

    }

}
```

```
public class PrioritySchedulingGUI extends
JFrame {



    private JTextField processField,
arrivalField, burstField, priorityField;

    private JButton addButton,
calculateButton;

    private JTextArea resultArea;

    private JPanel ganttPanel;

    private JComboBox<String>
algorithmSelector;

    private ArrayList<Process> processes =
new ArrayList<>();

    private ArrayList<int[]> ganttSequence =
new ArrayList<>(); // Track the sequence of
process executions with timeline



    public PrioritySchedulingGUI() {

        setTitle("Priority Scheduling -
Preemptive and Non-Preemptive Priority
Scheduling");

        setSize(800, 600);


setDefaultCloseOperation(JFrame.EXIT_O
N_CLOSE);

        setLayout(new BorderLayout());
```

```
        // Input Panel
        JPanel inputPanel = new JPanel(new GridLayout(6, 2));

inputPanel.setBorder(BorderFactory.createTitledBorder("Add Process Details"));

        inputPanel.add(new JLabel("Process ID:"));
        processField = new JTextField();
        inputPanel.add(processField);

        inputPanel.add(new JLabel("Arrival Time:"));
        arrivalField = new JTextField();
        inputPanel.add(arrivalField);

        inputPanel.add(new JLabel("Burst Time:"));
        burstField = new JTextField();
        inputPanel.add(burstField);

        inputPanel.add(new JLabel("Priority:"));
        priorityField = new JTextField();
        inputPanel.add(priorityField);

        // Algorithm Selector
        inputPanel.add(new JLabel("Algorithm:"));
        algorithmSelector = new JComboBox<>(new String[]{"Preemptive Priority", "Non-Preemptive Priority Scheduling"});
        inputPanel.add(algorithmSelector);

        addButton = new JButton("Add Process");
        inputPanel.add(addButton);

        calculateButton = new JButton("Calculate & Show Gantt Chart");
        inputPanel.add(calculateButton);

        add(inputPanel, BorderLayout.NORTH);

        // Gantt Panel
        ganttPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                drawGanttChart(g);
```

```
        }
    };

ganttPanel.setBorder(BorderFactory.createTitledBorder("Gantt Chart"));
    add(ganttPanel, BorderLayout.CENTER);

    // Result Panel
    resultArea = new JTextArea(10, 50);
    resultArea.setEditable(false);
    add(new JScrollPane(resultArea), BorderLayout.SOUTH);

    // Button Actions
    addButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            addProcess();
        }
    });

    calculateButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            String selectedAlgorithm = (String) algorithmSelector.getSelectedItem();
            if (selectedAlgorithm.equals("Preemptive Priority")) {
                calculatePreemptivePriorityScheduling();
            } else {
                calculateNonPreemptivePriorityScheduling(); // FCFS
            }
            ganttPanel.repaint();
        }
    });
}

private void addProcess() {
    try {
        int pid = Integer.parseInt(processField.getText());
        int arrival = Integer.parseInt(arrivalField.getText());
        int burst = Integer.parseInt(burstField.getText());
```

```
            int priority =
Integer.parseInt(priorityField.getText());



            processes.add(new Process(pid,
arrival, burst, priority));

            resultArea.append("Process P" + pid
+ " added.\n");



            // Clear the fields for the next process

            processField.setText("");

            arrivalField.setText("");

            burstField.setText("");

            priorityField.setText("");

        } catch (NumberFormatException ex) {


JOptionPane.showMessageDialog(this,
"Please enter valid numbers.");

        }

    }



    // Non-preemptive (FCFS)

    private void
calculateNonPreemptivePriorityScheduling(
) {


calculateNonPreemptivePriorityScheduling1
();

        }
        // Non-preemptive Priority Scheduling

private void
calculateNonPreemptivePriorityScheduling1
() {

    if (processes.isEmpty()) {

        JOptionPane.showMessageDialog(this,
"No processes added!");

        return;

    }



    // Clear previous Gantt chart sequence

    ganttSequence.clear();



    int currentTime = 0;

    float totalTurnaroundTime = 0,
totalWaitingTime = 0;

    int completed = 0;

    int n = processes.size();



    // Mark process as visited to track
completed processes

    boolean[] visited = new boolean[n];



    while (completed != n) {

        // Find the highest priority process that
has arrived and is not yet completed
```

```
        Process highestPriorityProcess = null;
        int highestPriorityIndex = -1;

        for (int i = 0; i < n; i++) {
            Process p = processes.get(i);
            if (!visited[i] && p.arrivalTime <= currentTime) {
                if (highestPriorityProcess == null || p.priority < highestPriorityProcess.priority) {
                    highestPriorityProcess = p;
                    highestPriorityIndex = i;
                }
            }
        }

        if (highestPriorityProcess != null) {
            // Execute the process
            if (currentTime < highestPriorityProcess.arrivalTime) {
                currentTime = highestPriorityProcess.arrivalTime;
            }

            int startTime = currentTime;
            int endTime = currentTime + highestPriorityProcess.burstTime;

            highestPriorityProcess.completionTime = endTime;

            highestPriorityProcess.turnaroundTime = highestPriorityProcess.completionTime - highestPriorityProcess.arrivalTime;
            highestPriorityProcess.waitingTime = highestPriorityProcess.turnaroundTime - highestPriorityProcess.burstTime;

            // Update total times
            totalTurnaroundTime += highestPriorityProcess.turnaroundTime;
            totalWaitingTime += highestPriorityProcess.waitingTime;

            // Mark process as completed
            visited[highestPriorityIndex] = true;
            completed++;

            // Update current time
            currentTime = endTime;

            // Add to the Gantt chart sequence
```

```
            ganttSequence.add(new
int[]{highestPriorityProcess.pid, startTime,
endTime});

        } else {

            // If no process is available,
increment the current time

            currentTime++;

        }

    }



    displayResults(totalTurnaroundTime,
totalWaitingTime);

}




    // Preemptive Priority Scheduling

    private void
calculatePreemptivePriorityScheduling() {

        if (processes.isEmpty()) {


JOptionPane.showMessageDialog(this, "No
processes added!");

            return;

        }
```

```
        // Sort processes by arrival time
initially


processes.sort(Comparator.comparingInt(p -
> p.arrivalTime));



        int currentTime = 0, completed = 0;

        int n = processes.size();

        float totalTurnaroundTime = 0,
totalWaitingTime = 0;



        ArrayList<Process>
remainingProcesses = new
ArrayList<>(processes);



        while (completed != n) {

            // Find the highest priority process
that has arrived by currentTime and has
remaining burst time

            Process currentProcess = null;

            for (Process p : remainingProcesses)
{

                if (p.arrivalTime <= currentTime
&& p.remainingTime > 0) {

                    if (currentProcess == null ||
p.priority < currentProcess.priority) {

                        currentProcess = p;

                    }

                }
```

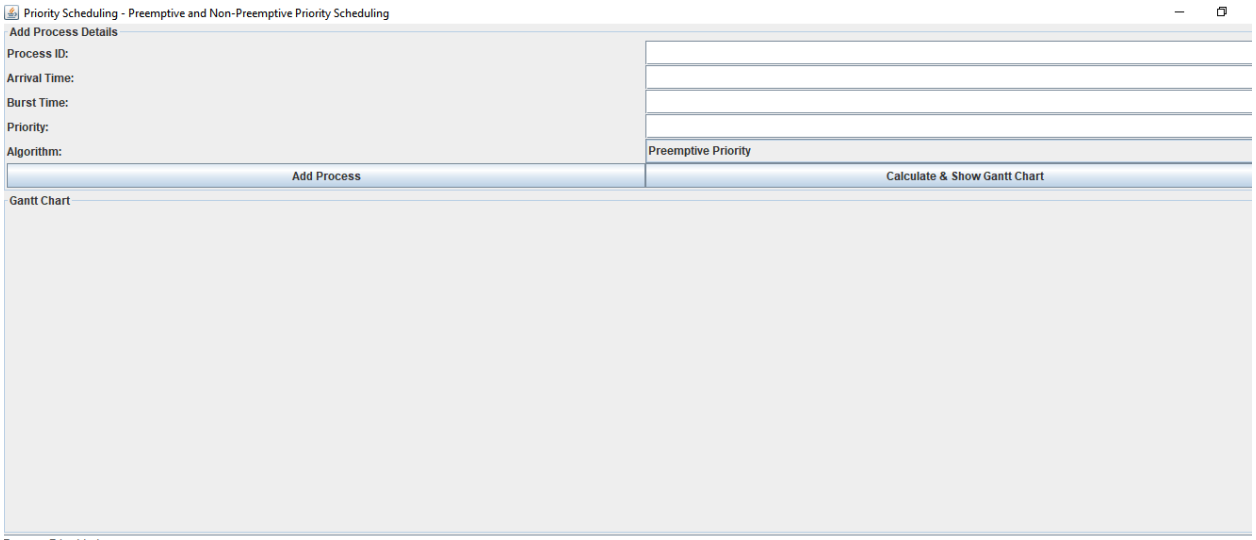
```
        }

        if (currentProcess != null) {

currentProcess.executionIntervals.add(new
int[]{currentTime, currentTime + 1});

            ganttSequence.add(new
int[]{currentProcess.pid, currentTime,
currentTime + 1});

            currentProcess.remainingTime--;

            currentTime++;


            if (currentProcess.remainingTime
== 0) {

                completed++;

                currentProcess.completionTime
= currentTime;

                currentProcess.turnaroundTime
= currentProcess.completionTime -
currentProcess.arrivalTime;

                currentProcess.waitingTime =
currentProcess.turnaroundTime -
currentProcess.burstTime;

                totalTurnaroundTime +=
currentProcess.turnaroundTime;

                totalWaitingTime +=
currentProcess.waitingTime;

            }

        } else {
                currentTime++;

            }

        }


        displayResults(totalTurnaroundTime,
totalWaitingTime);

    }


    private void displayResults(float
totalTurnaroundTime, float
totalWaitingTime) {

        resultArea.setText(""); // Clear
previous results

resultArea.append("Process\tAT\tBT\tPri\tC
T\tTAT\tWT\n");

        for (Process p : processes) {

            resultArea.append("P" + p.pid + "\t"
+ p.arrivalTime + "\t" + p.burstTime + "\t" +
p.priority + "\t" +

                    p.completionTime + "\t" +
p.turnaroundTime + "\t" + p.waitingTime +
"\n");

        }


        float avgTurnaroundTime =
totalTurnaroundTime / processes.size();

        float avgWaitingTime =
totalWaitingTime / processes.size();
```

```
        resultArea.append("\nAverage Turnaround Time: " + avgTurnaroundTime);

        resultArea.append("\nAverage Waiting Time: " + avgWaitingTime);

    }



    // Drawing Gantt Chart with Preemption Reflected

    private void drawGanttChart(Graphics g) {

        if (ganttSequence.isEmpty()) {

            return;

        }



        int currentX = 50;

        int y = 50;

        int height = 40;

        int timelineY = 100;



        for (int[] entry : ganttSequence) {

            int pid = entry[0];

            int startTime = entry[1];

            int endTime = entry[2];

            int width = (endTime - startTime) * 20;

            // Color assignment based on process ID for distinction


g.setColor(getColorByProcessID(pid));

            g.fillRect(currentX, y, width, height);

            g.setColor(Color.BLACK);

            g.drawRect(currentX, y, width, height);



            // Process ID label

            g.drawString("P" + pid, currentX + width / 2 - 10, y + height / 2);



            // Timeline markers


g.drawString(String.valueOf(startTime), currentX, timelineY);

            currentX += width;

        }

        g.drawString(String.valueOf(currentX / 20), currentX, timelineY); // Final timeline marker

    }



    // Color assignment based on process ID for distinction
```

```
    private Color getColorByProcessID(int pid) {

        Color[] colors = {Color.RED, Color.GREEN, Color.BLUE, Color.ORANGE, Color.CYAN, Color.MAGENTA, Color.YELLOW};

        return colors[pid % colors.length];

    }



    public static void main(String[] args) {

        SwingUtilities.invokeLater(new Runnable() {

            @Override

            public void run() {

                PrioritySchedulingGUI gui = new PrioritySchedulingGUI();

                gui.setVisible(true);

            }

        });

    }

}
```

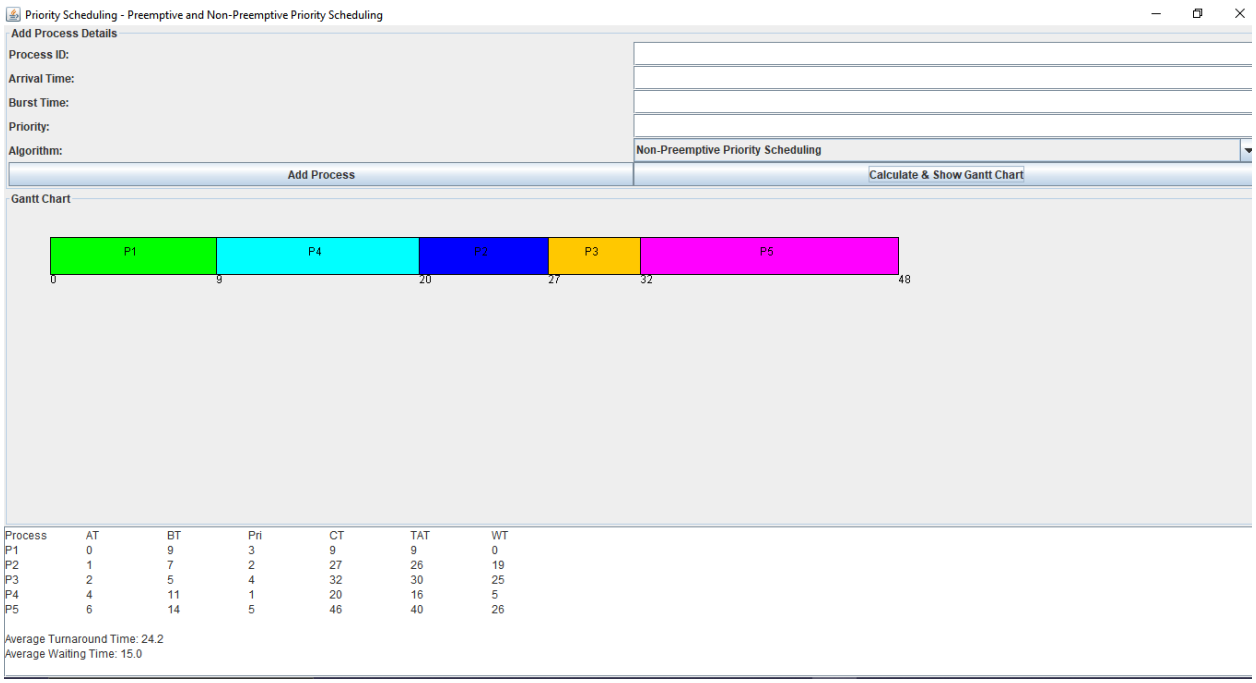
Output:

Adding Process:



Process P1 added.
Process P2 added.
Process P3 added.
Process P4 added.
Process P5 added.

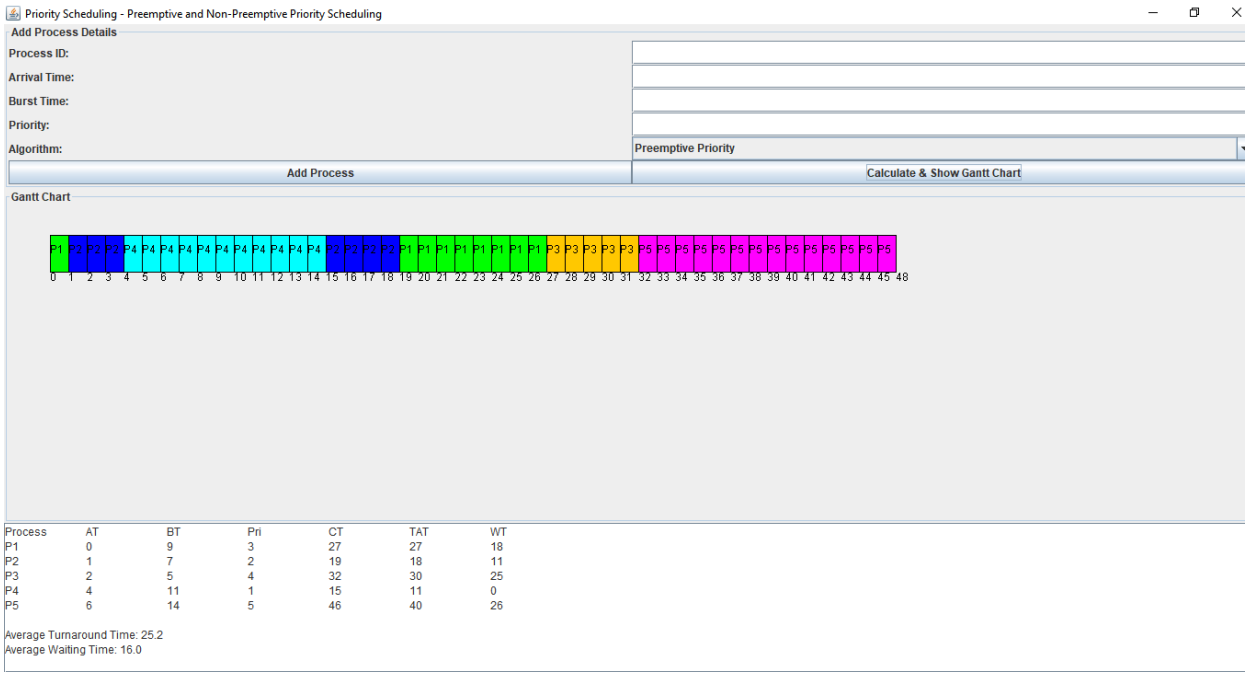
Non Preemptive :



Process	AT	BT	Pri	CT	TAT	WT
P1	0	9	3	9	9	0
P2	1	7	2	27	26	19
P3	2	5	4	32	30	25
P4	4	11	1	20	16	5
P5	6	14	5	46	40	26

Average Turnaround Time: 24.2
Average Waiting Time: 15.0

Preemptive:



Applications :

1. Real-Time Systems
2. Embedded Systems
3. Network Packet Scheduling
4. Telecommunication Systems

Reference :

1. OPERATING SYSTEM CONCEPTS
   Author: ABRAHAM SILBERSCHATZ ,PETER B.GALVIN GREG GAGNE

2. www.cs.wise.edu/~bart/537

3. www.en.wikipedia.org/wiki/Operating system