PROJECT 2

SHORTEST JOB FIRST SCHEDULING ALGORITHM

Under the guidance of

Dr. P. Thiyagarajan
Associate Professor
Head Dept. of Computer Science
(Cyber Security)

DEPARTMENT OF COMPUTER SCIENCE
RAJIV GANDHI NATIONAL INSTITUTE OF YOUTH DEVELOPMENT
SRIPERUMBUDUR, TAMIL NADU, 602105

MADE BY:			
NAME	ROLL NO.	TOPIC	PAGE NO.
ARUNDHATI SEN-SHARMA KAVYA P	MSAI22R005 MSAI22R010	MAIN CLASS	1-3
DIPANKAR BORA NUKULLA VAMSI LAKSHMANA PHANINDRA	MSAI22R008 MSCS22R011	GANTT CHART FOR PREEMPTIVE SJF	4-7
SRAVAN K GUDEPU ESHWARI	MSDS22R013 MSCS22R005	GANTT CHART FOR NON-PREEMPTIVE SJF	4-7
ATHAR OBAID KHAN	MSAI22R006	SJF PROCESS	
SHAHABAZ N VARADHA S AJITH	MSDS22R012 MSCS22R016	CALCULATION FOR PREEMPTIVE SJF	4-7
MOHAMED ATHFAN D G ROSHNI	MSCS22R007 MSDS22R010	CALCULATION FOR PREEMPTIVE SJF	4-7

Department of Computer Science Rajiv Gandhi National Institute of Youth Development Sriperumbudur, Tamil Nadu, India, 602105

CERTIFICATE

This is to certify that we have examined the project entitled " **Shortest job first scheduling algorithm** ", submitted by **Nukulla Vamsi Lakshmana Phanindra,Roshni G,Sravan K,Gudepu Eshwari,Mohamed Athfan D,Shahabaz N,Athar Obaid Khan,Arundhati Sensharma,Varadha S Ajith,Kavya p,Dipankar Bora** , the postgraduate students of **Department of Computer Science** in partial fulfillment for the award of degree of **Master of Computer Science** . We hereby accord our 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.

Place: Sriperumbudur
Date:. . . . . .

Supervisor
Dr.P.Thiyagarajan,
Dept. of Computer Science
(Cyber seurity),
RGNIYD , Sriperumbudur.

ACKNOWLEDGMENT

We would like to express our sincere and deep gratitude to our supervisor and guide, **Dr.P.Thiyagarajan**, Associate Professor,HOD of Department of Computer Science (Cyber Security), for his kind and constant support during our post-graduation study. 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 shaping our professional outlook.

CONTENTS

1 Introduction

Shortest job first (SJF) is a scheduling algorithm that selects for execution the waiting process with the smallest execution time from a queue of processes. It is used in operating systems to prioritize the execution of processes based on their burst time, which is the amount of time required by a process to complete its execution. It is also known as Shortest Job Next (SJN) or Shortest Process Next (SPN). SJF is a non-preemptive algorithm. The SJF algorithm selects the process with the shortest burst time as the next one to be executed, resulting in the completion of the shortest job first.

This is how the SJF algorithm works:

- **Process Arrival:** As processes arrive in the system, their burst time is determined. The burst time represents the time required by a process to complete its execution.
- **Ready Queue:** The processes that have arrived and are waiting to be executed are placed in a ready queue. The ready queue contains all the processes that are eligible for execution.
- **Shortest Job Selection:** The scheduler selects the process with the shortest burst time from the ready queue. This means that the process that requires the least amount of time to complete will be given priority.
- **Execution:** The selected process starts its execution and continues until it either completes or is interrupted by a higher-priority process.
- **Preemption:** SJF can be either preemptive or non-preemptive. In the preemptive version, if a new process arrives with a shorter burst time than the currently executing process, the scheduler preempts the running process and allows the new process to execute. The preemption involves saving the state of the running process and resuming it later. In the non-preemptive version, the currently running process continues until completion before the next shortest job is selected.
- Once a process completes its execution, its waiting time (the time spent in the ready queue) is recorded. This waiting time is later used to calculate the average waiting time for all processes.

The SJF algorithm repeats the steps 3 to 6 until all processes have been executed.

The various times related to a process are:

- **Arrival Time (AT):** It is the time of arrival of a process in its ready state (before its execution).
- **Burst Time (BT):** It is the total time that a process requires for its overall execution.
- **Completion Time (CT):** It is the exact time at which a process completes the execution.
- **Turnaround Time (TAT):** It refers to the total time interval present between the time of process submission and the time of its completion. The difference between the time of completion and the time of arrival is known as the Turn Around Time of the process.
- **Waiting Time (WT):** It refers to the total time that a process spends while waiting in a ready queue before reaching the CPU. The difference between (time) of the turnaround and burst time is known as the waiting time of a process.

1.1 Characteristics of SJF Scheduling:

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
- It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
- SJF can be used in specialized environments where accurate estimates of running time are available.

The SJF algorithm aims to minimize the average waiting time, as it prioritizes processes with shorter burst times.
However, it can suffer from a drawback known as "starvation," where long processes are constantly delayed by the arrival of shorter processes.
To mitigate this issue, a variant called **Shortest Remaining Time First (SRTF)** can be used, which is a preemptive version of SJF. SRTF allows the running process to be preempted if a new process with an even shorter burst time arrives.

1.2 Object Oriented Programming concepts used in this project are:

- **Encapsulation:** It is a fundamental principle in object-oriented programming that combines data and methods within a class, hiding the internal details of the implementation from external access. It provides control over the access to the internal state of an object. Encapsulation is achieved by declaring the class's fields (data) as private and providing public methods (getters and setters) to access and modify the data.
- **Inheritance:** Inheritance is a mechanism that allows a class (child or derived class) to inherit properties and behaviors from another class (parent or base class). The child class can access the fields and methods of the parent class and extend or override them. Inheritance facilitates code reuse and supports the concept of hierarchical classification.
- **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It refers to the ability of an object to take on many forms. Polymorphism is achieved through method overriding and method overloading.
- **Abstraction:** It is defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.
- **Modularity:** It refers to the practice of dividing a program into separate, independent, and reusable modules or components. Each module performs a specific task or encapsulates related functionalities, making the code easier to understand, maintain, and test. Modularity promotes code organization and reusability.

2 Graphic used : Swing

Swing is a powerful and widely-used graphical user interface (GUI) toolkit for Java applications, providing a rich set of components and functionalities for creating desktop applications.

Swing has been a popular choice for Java developers due to its cross-platform compatibility and ability to create visually appealing and interactive user interfaces.

2.1 Features and Benefits:

- **Platform independence:** Swing is built on Java's Write Once, Run Anywhere (WORA) principle, allowing applications developed using Swing to run seamlessly on multiple platforms, including Windows, macOS, and Linux, without any modifications.
- **Rich set of components:** Swing offers a wide range of components, such as buttons, labels, text fields, combo boxes, tables, and more, enabling developers to create intuitive and feature-rich user interfaces.
- **Customizability:** Swing components can be easily customized to match the application's design requirements. Properties like background color, font, and size can be modified, allowing developers to create a unique look and feel.

```
JButton button = new JButton("Click Me");
button.setBackground(Color.BLUE);
button.setFont(new Font("Arial", Font.BOLD, 16));
```

Figure 1: Component Customizability.

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Code to be executed when the button is clicked
    }
});
```

Figure 2: Rich set of components

- **Event-driven programming:** Swing follows an event-driven programming model, where user interactions trigger events that can be handled by

```
JFrame frame = new JFrame("My Application");
frame.setLayout(new FlowLayout());
frame.add(new JButton("Button 1"));
frame.add(new JButton("Button 2"));
```

Figure 3: Components

the application. This allows for responsive and interactive applications, where actions can be performed based on user input.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class EventDrivenExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event-Driven Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a label
        JLabel label = new JLabel("Click the button!");
        frame.getContentPane().add(label);

        // Create a button
        JButton button = new JButton("Click Me");
        button.addActionListener(e -> label.setText("Button Clicked!"));
        frame.getContentPane().add(button);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 4: Event-driven programming

- **Layout managers:** Swing provides layout managers that help in arranging and positioning components within containers. Layout managers handle component positioning and resizing, ensuring proper alignment across different screen resolutions and window sizes.
- **Support for internationalization:** Swing supports internationalization and localization, making it easier to develop applications that can be translated into different languages. It provides mechanisms to handle different character encodings, date and time formats, and resource

```java
public class LayoutManagerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Manager Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a JPanel with FlowLayout
        JPanel panel = new JPanel(new FlowLayout());

        // Create buttons
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");

        // Add buttons to the panel
        panel.add(button1);
        panel.add(button2);
        panel.add(button3);

        // Add the panel to the frame
        frame.getContentPane().add(panel);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 5: Layout managers

bundles for storing localized strings.

- Integration with other Java libraries: Swing seamlessly integrates with other Java libraries and frameworks, allowing developers to enhance their applications with additional features. For example, Swing applications can utilize the JDBC library for database connectivity or incorporate data visualization using the JavaFX library.

2.2 Basic Swing Components:

commonly used Swing components, such as JFrame (top-level window), JPanel (container for other components), JButton (button), JTextField (text input field), JCheckBox (checkbox), and JList (list of items).

- **JFrame:** A JFrame is a top-level window that serves as the main container for a Swing application. It provides the basic framework for the

```java
public class InternationalizationExample {
    public static void main(String[] args) {
        // Set the desired locale
        Locale locale = new Locale("fr", "FR");

        // Load the appropriate resource bundle based on the locale
        ResourceBundle bundle = ResourceBundle.getBundle("MessagesBundle", 

        // Create a JFrame with FlowLayout
        JFrame frame = new JFrame(bundle.getString("window.title"));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());

        // Create a JButton with the text from the resource bundle
        JButton button = new JButton(bundle.getString("button.text"));

        // Add the button to the frame
        frame.add(button);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Figure 6: internationalization

application's user interface and allows you to add other components to it. You can customize the JFrame's title, size, position, and behavior.

- **JPanel:** A JPanel is a container component used to group and organize other components within a JFrame or another container. It provides a lightweight and flexible way to manage the layout of components. You can add various components, such as buttons, labels, or text fields, to a JPanel.
- **JButton:** A JButton represents a standard button that triggers an action when clicked. It is widely used to initiate actions or perform operations within a Swing application. You can add event listeners to handle button click events and define the actions to be performed.
- **JTextField:** A JTextField is a component that allows the user to enter and edit text. It provides a single-line text input field. You can retrieve the entered text programmatically and handle events, such as pressing Enter or losing focus

- **JCheckBox:** A JCheckBox represents a checkbox that can be selected or deselected by the user. It is commonly used for options that can be toggled on or off. You can add event listeners to handle checkbox state changes and perform actions accordingly.
- **JList:** A JList represents a component that displays a list of items. It is commonly used for displaying selectable lists of data. You can populate a JList with items and add event listeners to handle selection changes.
- **Advanced Swing Concepts:** Model-View-Controller (MVC) architecture: The Model-View-Controller (MVC) architecture is a design pattern commonly used in software development, including Swing applications. It separates the application into three distinct components: the Model, the View, and the Controller. Each component has a specific role and responsibility, which leads to more modular and maintainable code.
- **Model:** The Model represents the application's data and business logic. It encapsulates the data structures, algorithms, and operations related to the application's functionality. The Model is independent of the user interface and does not directly interact with the GUI components. It provides methods for accessing and manipulating the data, as well as performing calculations or transformations. The Model can be updated based on user input or other external events.
- **View:** The View represents the graphical user interface (GUI) components that allow users to interact with the application. It is responsible for displaying the data and providing a visual representation of the Model. The View is passive and observes changes in the Model to update its display. In Swing applications, View components are typically subclasses of Swing classes like JFrame, JPanel, or JLabel. The View should not contain any application logic; its purpose is solely to present the data to the user.
- **Controller:** The Controller acts as an intermediary between the Model and the View. It receives user input from the View and translates it into actions that manipulate the Model. The Controller handles events triggered by user interactions with GUI components and initiates appropriate actions in the Model. It updates the Model based on user input and also updates the View to reflect any changes in the data. In Swing

applications, event listeners attached to GUI components are often used as Controllers.

- **SwingWorker and multithreading:** Background tasks and the Event Dispatch Thread (EDT): The Event Dispatch Thread (EDT) is the thread responsible for handling user interface events and updating the GUI components. All UI-related tasks, such as processing user input or updating component states, should be performed in the EDT. If a time-consuming task is executed in the EDT, it will block the thread and cause the GUI to freeze, resulting in an unresponsive application.
- **Keeping the UI responsive with SwingWorker :** SwingWorker is designed to execute time-consuming tasks in the background while allowing the EDT to remain free for handling user events and updating the UI. By moving lengthy operations to a background thread, the UI can remain responsive, and users can continue interacting with the application. SwingWorker provides methods for executing tasks asynchronously and retrieving results once they are completed.
- **Key features and benefits of SwingWorker:** Background execution: SwingWorker provides a convenient way to execute time-consuming tasks, such as data processing, network operations, or file I/O, in a separate thread.
- **Progress tracking:** SwingWorker supports reporting the progress of a task, allowing you to update progress bars or provide visual feedback to the user during lengthy operations.
- **Result retrieval and notification:** SwingWorker allows you to obtain the result of a background task once it completes. It also provides mechanisms to notify the UI about task completion or intermediate results.
- **Thread synchronization:** SwingWorker handles thread synchronization internally, ensuring that updates to the UI are performed safely and avoiding potential data race conditions. By using SwingWorker effectively, you can offload time-consuming tasks to background threads, ensuring a responsive user interface and enhancing the overall user experience of our Swing application.

3 Gantt Chart for Preemptive: Code Explanation

Below is the step-by-step explanation of the code for Preemptive process

The code snippet defines a class named FrameForPreemptiveSJF that extends the JFrame class in Java Swing. It represents a custom JFrame window for visualizing the execution of processes using the Preemptive Shortest Job First (SJF) scheduling algorithm.

- The class has three instance variables: CPUBurstTime[], obj of type Main, and leftStart of type int.
- The constructor FrameForPreemptiveSJF takes an instance of the Main class as a parameter. It initializes the JFrame by setting the title to "Preemptive SJF", making it visible, and setting its size based on the SCREEN WIDTH and SCREEN HEIGHT properties of the Main object. It also clones the CPUBurstTime array from the Main object.
- The paint method is overridden to customize the rendering of the JFrame. It starts by calling the super.paint(g) method to ensure the default painting is performed.
- The CPUBurstTime array is updated by cloning the array from the Main object using CPUBurstTime = obj.CPUBurstTime.clone().
- The background color of the JFrame is set to white using this.getContentPane().setBackground(Color.white).
- The current time is initialized to obj.minimumArrivalTime, which represents the minimum arrival time of the processes.
- Several variables are declared: min and mini to store the minimum CPU burst time and its index, prevmini to store the previous minimum index, and leftStart to keep track of the starting position for drawing rectangles.
- The Graphics object g is obtained from this.getContentPane().getGraphics().
- The minimum arrival time is drawn on the screen using g.drawString(). The value is converted to a string and drawn at the specified position (leftStart, obj.rectangleUpperPadding + obj.rectangleHeight + 20).
- A loop is executed obj.sumCPUBurstTime times to schedule the processes based on the Preemptive SJF algorithm.

- Within the loop, the minimum CPU burst time and its corresponding process index are found by iterating over the CPUBurstTime array and considering the arrival time. The minimum value is updated in the min variable, and its index is stored in the mini variable.
- On the first iteration of the loop (j == 0), the prevmini is set to mini to keep track of the previous process index.
- If the prevmini is different from the current mini or it is the last iteration of the loop (j == obj.sumCPUBurstTime-1), a rectangle is drawn using g.drawRect() to represent the execution of the previous process. The dimensions of the rectangle are determined based on the leftStart, obj.rectangleUpperPadding, obj.lengthOfEachBlock, and obj.rectangleHeight values.
- The label for the previous process (e.g., P1, P2) is drawn using g.drawString().
- The leftStart is updated based on the length of the drawn rectangle.
- If it is the last iteration of the loop, the current time is drawn using g.drawString().
- The current time is incremented, and the CPU burst time of the current process (CPUBurstTime[mini]) is reduced by one.
- The prevmini is updated to mini for the next iteration of the loop

4 Gantt Chart for Non-Preemptive: Code Explanation

Below is the step-by-step explanation of the code for non-Preemptive process:

- A custom class in Java Swing called FrameForNonPreemptiveSJF, which extends the JFrame class. This class is responsible for creating a JFrame window that visualizes the execution of processes using the Non-Preemptive Shortest Job First (SJF) scheduling algorithm.
- Inside the FrameForNonPreemptiveSJF class, there is an instance variable named CPUBurstTime, which appears to be an array that stores the CPU burst times of the processes.

- The class has a constructor FrameForNonPreemptiveSJF, which takes an instance of the Main class as a parameter. It initializes the JFrame by setting the title to "Non-preemptive SJF", making it visible, and setting its size based on the SCREEN WIDTH and SCREEN HEIGHT variables of the Main object. It also clones the CPUBurstTime array from the Main object.
- The paint method is overridden to customize the rendering of the JFrame. It starts by calling the super.paint(g) method to ensure the default painting is performed.
- The background color of the JFrame is set to white using .getContentPane().setBackground(Color.white).
- The current time is initialized to obj.minimumArrivalTime, which appears to be the minimum arrival time of the processes.
- The CPUBurstTime array is updated to match the array from the Main object using CPUBurstTime=obj.CPUBurstTime.clone().
- Variables i, j, min, and mini are declared to keep track of the minimum CPU burst time and its corresponding process index.
- The starting position for drawing rectangles is set to leftStart = 50.
- The Graphics object g is obtained from this.getContentPane().getGraphics().
- The minimum arrival time is drawn on the screen using g.drawString().
- A loop is executed for obj.numberOfProcesses times to schedule the processes based on the Non-Preemptive Shortest Job First algorithm.
- Within the loop, the minimum CPU burst time and its corresponding process index are found by iterating over the CPUBurstTime array.
- The rectangle representing the execution of the process is drawn using g.drawRect(), with the width of the rectangle determined by obj.CPUBurstTime[mini].
- The process label (e.g., P1, P2) is drawn using g.drawString().
- The leftStart is updated based on the length of the drawn rectangle.
- The current time is updated by adding the CPU burst time of the current process.

- The CPU burst time of the current process is set to Integer.MAX VALUE to indicate that it has been executed.
- he loop continues until all processes have been scheduled and drawn on the screen.

By following these steps, the FrameForNonPreemptiveSJF class creates a custom JFrame that visualizes the execution of processes using the Non-Preemptive Shortest Job First algorithm.

5 Process SJF

Process SJF The code here provided defines a class called ProcessSJF, which appears to implement the Shortest Job First (SJF) scheduling algorithm. Here's a breakdown of the code:

- The class has three protected arrays: processID (String array), arrivalTime (integer array), and burstTime (integer array). These arrays store the process ID, arrival time, and burst time of each process, respectively.
- The constructor ProcessSJF takes the numberOfProcesses as a parameter and initializes the arrays with the specified size.
- The inputProcessData method allows the user to input the process data (ID, arrival time, and burst time) for each process using the Scanner class. It iterates over the arrays and prompt the user for input.
- The displayProcessData method takes three arrays as parameters: completionTime, turnaroundTime, and waitingTime. It displays a table of the process data, including the process ID, arrival time, burst time, completion time, turnaround time, and waiting time. It uses a loop to iterate over the arrays and prints the values in a formatted manner.
- The calculateAverage method takes an array data as a parameter and calculates the average of its values. It iterates over the array, sums up all the elements, and then divides the sum by the length of the array. The method returns the calculated average as a float.

Overall, this code provides a basic structure for implementing the SJF scheduling algorithm.

6 NonPreemptiveSJF

NonPreemptiveSJF The code here provided extends the ProcessSJF class and defines a new class called NonPreemptiveSJF. This class represents the non-preemptive implementation of the Shortest Job First (SJF) scheduling algorithm. Here's an explanation of the code:

- The NonPreemptiveSJF constructor takes the numberOfProcesses parameter and calls the constructor of the parent class ProcessSJF using the super(numberOfProcesses) statement. This initializes the arrays for process data.
- The calculateSJF method calculates the scheduling based on the non-preemptive SJF algorithm. It creates copies of the original arrays (arrivalTime and burstTime) using Arrays.copyOf to avoid modifying the original data. It also creates new arrays for completionTime, turnaroundTime, and waitingTime.
- The method uses a while loop that continues until all processes are completed (completedProcesses ¡ processID.length). Inside the loop, it finds the process with the shortest burst time among the processes that have arrived (arrivalTimeCopy[i] ¡= currentTime), and whose burst time is greater than 0 (burstTimeCopy[i] ¿ 0).
- If a process is found (shortestJobIndex != -1), the current time is incremented by the burst time of the shortest job (currentTime += burstTimeCopy[shortestJobIndex]). The completion time, turnaround time, and waiting time for the process are calculated and stored in the respective arrays. The burst time of the completed process is set to 0, indicating completion.
- If no process is found (shortestJobIndex == -1), it means there are no arrived processes with burst time greater than 0. In this case, the current time is incremented by 1 (currentTime++) and the loop continues.
- After the loop, the displayProcessData method is called to show the process data in a tabular format, including the completion time, turnaround time, and waiting time for each process.
- Finally, the average turnaround time and average waiting time are calculated using the calculateAverage method and displayed on the console.

This implementation of the non-preemptive SJF scheduling algorithm finds the shortest job among the arrived processes and executes it to completion before moving on to the next job.

7 PreemptiveSJF

PreemptiveSJF The code here provided extends the ProcessSJF class and defines a new class called PreemptiveSJF. This class represents the preemptive implementation of the Shortest Job First (SJF) scheduling algorithm. Here's an explanation of the code:

- The PreemptiveSJF constructor takes the numberOfProcesses parameter and calls the constructor of the parent class ProcessSJF using the super(numberOfProcesses) statement. This initializes the arrays for process data.
- The calculateSJF method calculates the scheduling based on the preemptive SJF algorithm. It creates a copy of the original burstTime array using Arrays.copyOf to preserve the original data. It also creates new arrays for completionTime, turnaroundTime, and waitingTime.
- The method uses a while loop that continues until all processes are completed (completedProcesses ¡ processID.length). Inside the loop, it finds the process with the shortest burst time among the processes that have arrived (arrivalTime[i] ¡= currentTime), and whose burst time is greater than 0 (burstTimeCopy[i] ¿ 0).
- If a process is found (shortestJobIndex != -1), it decrements the burst time of the process (burstTimeCopy[shortestJobIndex]–) and increments the current time (currentTime++), simulating the execution of a unit of burst time.
- If the burst time of the process becomes 0 (burstTimeCopy[shortestJobIndex] == 0), it means the process has completed. The completion time, turnaround time, and waiting time for the process are calculated and stored in the respective arrays. The completedProcesses counter is incremented.
- If no process is found (shortestJobIndex == -1), it means there are no arrived processes with burst time greater than 0. In this case, the current

```java
class ProcessSJF {
    protected String[] processID;
    protected int[] arrivalTime;
    protected int[] burstTime;

    public ProcessSJF(int numberOfProcesses) {
        processID = new String[numberOfProcesses];
        arrivalTime = new int[numberOfProcesses];
        burstTime = new int[numberOfProcesses];
    }
    public void inputProcessData() {
        Scanner scanner = new Scanner(System.in);

        for (int i = 0; i < processID.length; i++) {
            System.out.print("Enter Process ID: ");
            processID[i] = scanner.next();
            System.out.print("Enter Arrival Time: ");
            arrivalTime[i] = scanner.nextInt();
            System.out.print("Enter Burst Time: ");
            burstTime[i] = scanner.nextInt();
        }
    }
    public void displayProcessData(int[] completionTime, int[] turnaroundTime, int[] waitingTime) {
        System.out.println("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time");
        for (int i = 0; i < processID.length; i++) {
            System.out.printf("%s\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
                    processID[i], arrivalTime[i], burstTime[i], completionTime[i], turnaroundTime[i], waitingTime[i]);
        }
    }
    public float calculateAverage(int[] data) {
        int sum = 0;
        for (int i = 0; i < data.length; i++) {
            sum += data[i];
        }
        return (float) sum / data.length;
    }
```

Figure 7: Process SJF

```java
class PreemptiveSJF extends ProcessSJF {
    public PreemptiveSJF(int numberOfProcesses) {
        super(numberOfProcesses);
    }
    public void calculateSJF() {
        // Create a copy of the original arrays to preserve the original data
        int[] burstTimeCopy = Arrays.copyOf(burstTime, burstTime.length);
        int[] completionTime = new int[processID.length];
        int[] turnaroundTime = new int[processID.length];
        int[] waitingTime = new int[processID.length];
        int currentTime = 0;
        int completedProcesses = 0;
        while (completedProcesses < processID.length) {
            int shortestJobIndex = -1;
            int shortestJobBurstTime = Integer.MAX_VALUE;
            for (int i = 0; i < processID.length; i++) {
                if (arrivalTime[i] <= currentTime && burstTimeCopy[i] < shortestJobBurstTime && burstTimeCopy[i] > 0) {
                    shortestJobIndex = i;
                    shortestJobBurstTime = burstTimeCopy[i];
                }
            }
            if (shortestJobIndex == -1) {
                currentTime++;
                continue;
            }
            burstTimeCopy[shortestJobIndex]--;
            currentTime++;
            if (burstTimeCopy[shortestJobIndex] == 0) {
                completionTime[shortestJobIndex] = currentTime;
                turnaroundTime[shortestJobIndex] = completionTime[shortestJobIndex] - arrivalTime[shortestJobIndex];
                waitingTime[shortestJobIndex] = turnaroundTime[shortestJobIndex] - burstTime[shortestJobIndex];
                completedProcesses++;
            }
        }
```

Figure 8: Preemptive SJF

time is incremented by 1 (currentTime++) and the loop continues.

- After the loop, the displayProcessData method is called to show the process data in a tabular format, including the completion time, turnaround time, and waiting time for each process.
- Finally, the average turnaround time and average waiting time are calculated using the calculateAverage method and displayed on the console.

] This implementation of the preemptive SJF scheduling algorithm repeatedly selects the process with the shortest remaining burst time among the arrived processes and executes them one unit at a time. It dynamically ad-

```
class NonPreemptiveSJF extends ProcessSJF {
    public NonPreemptiveSJF(int numberOfProcesses) {
        super(numberOfProcesses);
    }
    public void calculateSJF() {
        // Create a copy of the original arrays and sort them based on arrival time
        int[] arrivalTimeCopy = Arrays.copyOf(arrivalTime, arrivalTime.length);
        int[] burstTimeCopy = Arrays.copyOf(burstTime, burstTime.length);
        int[] completionTime = new int[processID.length];
        int[] turnaroundTime = new int[processID.length];
        int[] waitingTime = new int[processID.length];
        int currentTime = 0;
        int completedProcesses = 0;
        while (completedProcesses < processID.length) {
            int shortestJobIndex = -1;
            int shortestJobBurstTime = Integer.MAX_VALUE;
            for (int i = 0; i < processID.length; i++) {
                if (arrivalTimeCopy[i] <= currentTime && burstTimeCopy[i] < shortestJobBurstTime && burstTimeCopy[i] > 0) {
                    shortestJobIndex = i;
                    shortestJobBurstTime = burstTimeCopy[i];
                }
            }
            if (shortestJobIndex == -1) {
                currentTime++;
                continue;
            }
            currentTime += burstTimeCopy[shortestJobIndex];
            completionTime[shortestJobIndex] = currentTime;
            turnaroundTime[shortestJobIndex] = completionTime[shortestJobIndex] - arrivalTimeCopy[shortestJobIndex];
            waitingTime[shortestJobIndex] = turnaroundTime[shortestJobIndex] - burstTime[shortestJobIndex];
            burstTimeCopy[shortestJobIndex] = 0;
            completedProcesses++;
        }
        displayProcessData(completionTime, turnaroundTime, waitingTime);
        System.out.println("Average Turnaround Time (Non-Preemptive SJF): " + calculateAverage(turnaroundTime));
        System.out.println("Average Waiting Time (Non-Preemptive SJF): " + calculateAverage(waitingTime));
```

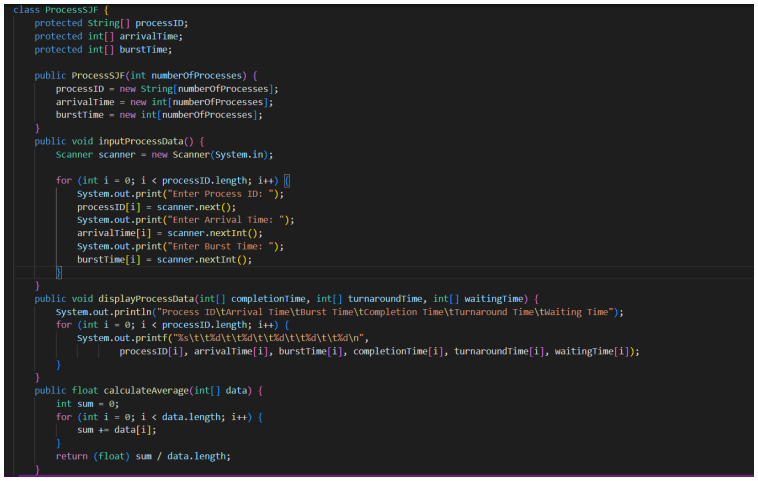
Figure 9: Non-Preemptive SJF

```
class FrameForPreemptiveSJF extends JFrame{
            int CPUBurstTime[];
            Main obj;
            FrameForPreemptiveSJF(Main obj){
                        super("Preemptive SJF");
                        this.obj=obj;
                        //this.setResizable(false);
                        this.setVisible(true);
                        this.setSize(obj.SCREEN_WIDTH+100, obj.SCREEN_HEIGHT);
                        CPUBurstTime=obj.CPUBurstTime.clone();}
            @Override
            public void paint(Graphics g){
                        super.paint(g);
                        CPUBurstTime=obj.CPUBurstTime.clone();
                        System.out.println("paint called");
                        this.getContentPane().setBackground(Color.white);
                        int currentTime=obj.minimumArrivalTime;
                        int min,mini=0,prevmini = 0;
                        int leftStart=50;
                        g=this.getContentPane().getGraphics();
                g.drawString(""+obj.minimumArrivalTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                            for(int j=0;j<obj.sumCPUBurstTime;j++){
                                        min=Integer.MAX_VALUE;
                    for(int i=0;i<obj.numberOfProcesses;i++){
                        if(min>CPUBurstTime[i] && obj.arrivalTime[i]<=currentTime && CPUBurstTime[i]!=0){
                            min=CPUBurstTime[i];
                            mini=i;}
                        if(j==0)
                prevmini=mini;
                        if(prevmini!=mini || j==obj.sumCPUBurstTime-1){
                                    g=this.getContentPane().getGraphics();
                                    if(j==obj.sumCPUBurstTime-1)
                                                g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*(currentTime+1),obj.rectangleHeight);
                                    else
                                                g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*(currentTime),obj.rectangleHeight);
```

Figure 10: Frame for Preemptive SJF

```
class FrameForNonPreemptiveSJF extends JFrame{
            int CPUBurstTime[];
            Main obj;
            FrameForNonPreemptiveSJF(Main obj){
                        super("Non preemptive SJF");
                        this.obj=obj;
                        //this.setResizable(false);
                        this.setVisible(true);
                        this.setSize(obj.SCREEN_WIDTH+100, obj.SCREEN_HEIGHT);
                        CPUBurstTime=obj.CPUBurstTime.clone();
            }
            @Override
            public void paint(Graphics g){
                        super.paint(g);
                        this.getContentPane().setBackground(Color.white);
                        int currentTime=obj.minimumArrivalTime;
                        CPUBurstTime=obj.CPUBurstTime.clone();
                        int i,j,min,mini = 0;
                int leftStart=50;
                g=this.getContentPane().getGraphics();
                g.drawString(""+obj.minimumArrivalTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                for(j=0;j<obj.numberOfProcesses;j++){
                    min=Integer.MAX_VALUE;
                    for(i=0;i<obj.numberOfProcesses;i++){
                        if(min>CPUBurstTime[i] && obj.arrivalTime[i]<=currentTime){
                            min=CPUBurstTime[i];
                            mini=i;
                        }}
                    g=this.getContentPane().getGraphics();
                    g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*obj.CPUBurstTime[mini],obj.rectangleHeight);
                    g.drawString("P"+(mini+1),leftStart+5,obj.rectangleUpperPadding+50);
                    leftStart+=obj.lengthOfEachBlock*obj.CPUBurstTime[mini];
                    currentTime+=obj.CPUBurstTime[mini];
                    g.drawString(""+currentTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                    CPUBurstTime[mini]=Integer.MAX_VALUE;
```

Figure 11: Frame for Non-Preemptive SJF

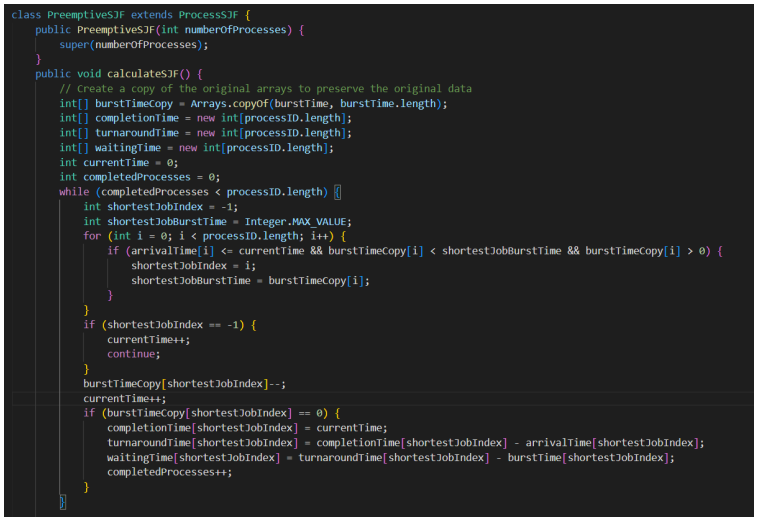


```
class ProcessSJF {
    protected String[] processID;
    protected int[] arrivalTime;
    protected int[] burstTime;

    public ProcessSJF(int numberOfProcesses) {
        processID = new String[numberOfProcesses];
        arrivalTime = new int[numberOfProcesses];
        burstTime = new int[numberOfProcesses];
    }
    public void inputProcessData() {
        Scanner scanner = new Scanner(System.in);

        for (int i = 0; i < processID.length; i++) {
            System.out.print("Enter Process ID: ");
            processID[i] = scanner.next();
            System.out.print("Enter Arrival Time: ");
            arrivalTime[i] = scanner.nextInt();
            System.out.print("Enter Burst Time: ");
            burstTime[i] = scanner.nextInt();
        }
    }
    public void displayProcessData(int[] completionTime, int[] turnaroundTime, int[] waitingTime) {
        System.out.println("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time");
        for (int i = 0; i < processID.length; i++) {
            System.out.printf("%s\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
                    processID[i], arrivalTime[i], burstTime[i], completionTime[i], turnaroundTime[i], waitingTime[i]);
        }
    }
    public float calculateAverage(int[] data) {
        int sum = 0;
        for (int i = 0; i < data.length; i++) {
            sum += data[i];
        }
        return (float) sum / data.length;
    }
```

Figure 12: Process SJF



```
class PreemptiveSJF extends ProcessSJF {
    public PreemptiveSJF(int numberOfProcesses) {
        super(numberOfProcesses);
    }
    public void calculateSJF() {
        // Create a copy of the original arrays to preserve the original data
        int[] burstTimeCopy = Arrays.copyOf(burstTime, burstTime.length);
        int[] completionTime = new int[processID.length];
        int[] turnaroundTime = new int[processID.length];
        int[] waitingTime = new int[processID.length];
        int currentTime = 0;
        int completedProcesses = 0;
        while (completedProcesses < processID.length) {
            int shortestJobIndex = -1;
            int shortestJobBurstTime = Integer.MAX_VALUE;
            for (int i = 0; i < processID.length; i++) {
                if (arrivalTime[i] <= currentTime && burstTimeCopy[i] < shortestJobBurstTime && burstTimeCopy[i] > 0) {
                    shortestJobIndex = i;
                    shortestJobBurstTime = burstTimeCopy[i];
                }
            }
            if (shortestJobIndex == -1) {
                currentTime++;
                continue;
            }
            burstTimeCopy[shortestJobIndex]--;
            currentTime++;
            if (burstTimeCopy[shortestJobIndex] == 0) {
                completionTime[shortestJobIndex] = currentTime;
                turnaroundTime[shortestJobIndex] = completionTime[shortestJobIndex] - arrivalTime[shortestJobIndex];
                waitingTime[shortestJobIndex] = turnaroundTime[shortestJobIndex] - burstTime[shortestJobIndex];
                completedProcesses++;
            }
        }
```

Figure 13: Preemptive SJF

justs the scheduling as burst times change during execution.

8 Overview of The Project Code

Here's a quick overview of the project code:

The program starts with the `Main` class, which contains the `main` method. It prompts the user to enter the number of processes.

It creates an instance of the `ProcessSJF` class and initializes the process details (process ID, arrival time, and burst time) by calling the `inputProcessData` method.

```java
class NonPreemptiveSJF extends ProcessSJF {
    public NonPreemptiveSJF(int numberOfProcesses) {
        super(numberOfProcesses);
    }
    public void calculateSJF() {
        // Create a copy of the original arrays and sort them based on arrival time
        int[] arrivalTimeCopy = Arrays.copyOf(arrivalTime, arrivalTime.length);
        int[] burstTimeCopy = Arrays.copyOf(burstTime, burstTime.length);
        int[] completionTime = new int[processID.length];
        int[] turnaroundTime = new int[processID.length];
        int[] waitingTime = new int[processID.length];
        int currentTime = 0;
        int completedProcesses = 0;
        while (completedProcesses < processID.length) {
            int shortestJobIndex = -1;
            int shortestJobBurstTime = Integer.MAX_VALUE;
            for (int i = 0; i < processID.length; i++) {
                if (arrivalTimeCopy[i] <= currentTime && burstTimeCopy[i] < shortestJobBurstTime && burstTimeCopy[i] > 0) {
                    shortestJobIndex = i;
                    shortestJobBurstTime = burstTimeCopy[i];
                }
            }
            if (shortestJobIndex == -1) {
                currentTime++;
                continue;
            }
            currentTime += burstTimeCopy[shortestJobIndex];
            completionTime[shortestJobIndex] = currentTime;
            turnaroundTime[shortestJobIndex] = completionTime[shortestJobIndex] - arrivalTimeCopy[shortestJobIndex];
            waitingTime[shortestJobIndex] = turnaroundTime[shortestJobIndex] - burstTime[shortestJobIndex];
            burstTimeCopy[shortestJobIndex] = 0;
            completedProcesses++;
        }
        displayProcessData(completionTime, turnaroundTime, waitingTime);
        System.out.println("Average Turnaround Time (Non-Preemptive SJF): " + calculateAverage(turnaroundTime));
        System.out.println("Average Waiting Time (Non-Preemptive SJF): " + calculateAverage(waitingTime));
```

Figure 14: Non-Preemptive SJF

```java
class FrameForPreemptiveSJF extends JFrame{
            int CPUBurstTime[];
            Main obj;
            FrameForPreemptiveSJF(Main obj){
                        super("Preemptive SJF");
                        this.obj=obj;
                        //this.setResizable(false);
                        this.setVisible(true);
                        this.setSize(obj.SCREEN_WIDTH+100, obj.SCREEN_HEIGHT);
                        CPUBurstTime=obj.CPUBurstTime.clone();}
            @Override
            public void paint(Graphics g){
                        super.paint(g);
                        CPUBurstTime=obj.CPUBurstTime.clone();
                        System.out.println("paint called");
                        this.getContentPane().setBackground(Color.white);
                        int currentTime=obj.minimumArrivalTime;
                        int min,mini=0,prevmini = 0;
                        int leftStart=50;
                        g=this.getContentPane().getGraphics();
                g.drawString(""+obj.minimumArrivalTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                                for(int j=0;j<obj.sumCPUBurstTime;j++){
                                            min=Integer.MAX_VALUE;
                        for(int i=0;i<obj.numberOfProcesses;i++){
                            if(min>CPUBurstTime[i] && obj.arrivalTime[i]<=currentTime && CPUBurstTime[i]!=0){
                                min=CPUBurstTime[i];
                                mini=i;}
                        if(j==0)
                            prevmini=mini;
                                if(prevmini!=mini || j==obj.sumCPUBurstTime-1){
                                            g=this.getContentPane().getGraphics();
                                            if(j==obj.sumCPUBurstTime-1)
                                                        g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*(currentTime+1),obj.rectangleHeight);
                                            else
                                                        g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*(currentTime),obj.rectangleHeight);
```

Figure 15: Frame for Preemptive SJF

```java
class FrameForNonPreemptiveSJF extends JFrame{
            int CPUBurstTime[];
            Main obj;
            FrameForNonPreemptiveSJF(Main obj){
                        super("Non preemptive SJF");
                        this.obj=obj;
                        //this.setResizable(false);
                        this.setVisible(true);
                        this.setSize(obj.SCREEN_WIDTH+100, obj.SCREEN_HEIGHT);
                        CPUBurstTime=obj.CPUBurstTime.clone();
            }
            @Override
            public void paint(Graphics g){
                        super.paint(g);
                        this.getContentPane().setBackground(Color.white);
                        int currentTime=obj.minimumArrivalTime;
                        CPUBurstTime=obj.CPUBurstTime.clone();
                        int i,j,min,mini = 0;
                int leftStart=50;
                g=this.getContentPane().getGraphics();
                g.drawString(""+obj.minimumArrivalTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                for(j=0;j<obj.numberOfProcesses;j++){
                    min=Integer.MAX_VALUE;
                    for(i=0;i<obj.numberOfProcesses;i++){
                        if(min>CPUBurstTime[i] && obj.arrivalTime[i]<=currentTime){
                            min=CPUBurstTime[i];
                            mini=i;
                        }}
                    g=this.getContentPane().getGraphics();
                    g.drawRect(leftStart,obj.rectangleUpperPadding,obj.lengthOfEachBlock*obj.CPUBurstTime[mini],obj.rectangleHeight);
                    g.drawString("P"+(mini+1),leftStart+5,obj.rectangleUpperPadding+50);
                    leftStart+=obj.lengthOfEachBlock*obj.CPUBurstTime[mini];
                    currentTime+=obj.CPUBurstTime[mini];
                    g.drawString(""+currentTime,leftStart,obj.rectangleUpperPadding+obj.rectangleHeight+20);
                    CPUBurstTime[mini]=Integer.MAX_VALUE;
```

Figure 16: Frame for Non-Preemptive SJF

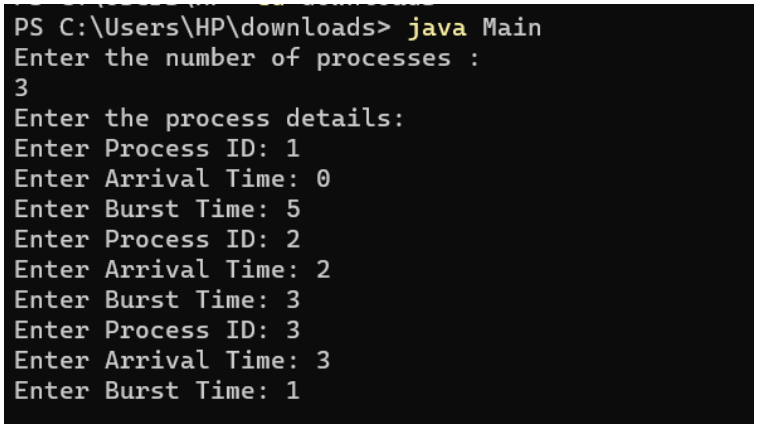


Figure 17: Taking User Input

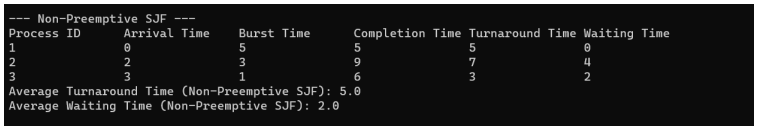


Figure 18: Output of Non-Preemptive SJF

The program then creates instances of `NonPreemptiveSJF` and `PreemptiveSJF` classes, passing the number of processes as a parameter. The `calculateSJF` method is called on both instances to calculate the scheduling for each algorithm.

After the scheduling is calculated, the `drawGanttChart` method is called to visualize the Gantt chart. The `drawGanttChart` method calculates the necessary parameters for drawing the chart and calls the methods `drawGanttChartForNonPree` and `drawGanttChartForPreemptiveSJF`. These methods create frames (`FrameForNonPreemptiveSJF` and `FrameForPreemptiveSJF`) and handle the drawing of the Gantt chart using the provided graphics objects.

The `ProcessSJF` class is a base class that holds the process data and provides methods to input/process data and display the process information.

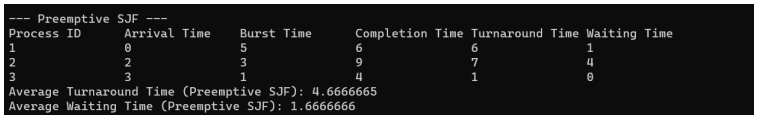


Figure 19: Output of Preemptive SJF

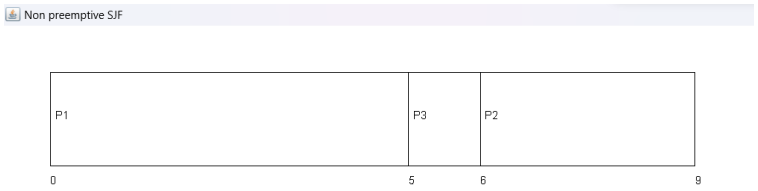


Figure 20: Gantt Chart of Non-Preemptive SJF

Figure 21: Gantt Chart of Preemptive SJF

The `NonPreemptiveSJF` and `PreemptiveSJF` classes extend `ProcessSJF` and implement the scheduling algorithms. They calculate the completion time, turnaround time, and waiting time for each process and display the process data and average times.

The program utilizes the Java Swing library to create graphical frames and draw the Gantt chart using the `Graphics` class.

Overall, the program allows the user to input process details, performs scheduling using non-preemptive and preemptive SJF algorithms, and visually represents the scheduling results through a Gantt chart.

9 Applications

The following are a few applications of this project:

- **Operating Systems**: The project simulates process scheduling in an operating system. The SJF algorithms ensure efficient CPU utilization by scheduling processes based on their burst time. This optimization improves the overall performance and responsiveness of the system.
- **Task Scheduling**: SJF algorithms can be used in task scheduling applications, such as job scheduling in data centers or task scheduling in cloud computing environments. By prioritizing tasks based on their expected execution time, these algorithms can optimize resource allocation and reduce overall execution time.

- **Job Scheduling in Manufacturing**: In manufacturing industries, SJF algorithms can be applied to schedule jobs on machines. By selecting the shortest job first, the algorithms can minimize the total time required to complete all the jobs, improving production efficiency.
- **Process Scheduling in Real-Time Systems**: Real-time systems, such as embedded systems or control systems, often require strict timing guarantees. SJF algorithms can be used to schedule real-time tasks based on their deadlines, ensuring that the most critical tasks are executed first and meet their timing requirements.
- **Multimedia Applications**: SJF algorithms are useful in multimedia applications that require real-time processing, such as video streaming or audio processing. By scheduling tasks based on their deadlines and processing requirements, these algorithms can ensure smooth playback and minimize delays.
- **Network Packet Scheduling**: In network routers or switches, SJF algorithms can be used for packet scheduling. By prioritizing packets based on their size or priority, the algorithms can reduce network latency and improve the overall quality of service.
- **Web Server Request Handling**: SJF algorithms can be employed in web servers to handle incoming client requests. By prioritizing requests based on their estimated execution time, the algorithms can ensure faster response times for shorter requests and prevent long-running requests from blocking the server.

These are just a few examples of how SJF scheduling algorithms can be applied in various real-world scenarios to optimize resource allocation, improve performance, and meet timing requirements.

10 Challenges

While working on this project, we faced several challenges. Here are some of those:

- **Input validation**: We needed to ensure that the user inputs are valid and within the expected range. For example, we needed to validate the

number of processes, arrival times, and burst times to prevent errors or unexpected behavior.

- **Graphics and GUI**: The project includes drawing a Gantt chart using Java's Graphics class and creating frames for displaying the chart. Working with graphics and GUI elements was complex, and we faced challenges in positioning the elements correctly or updating them dynamically. Firstly, we tried to go with Applet but then we switched to Swing as Swing is platform-independent and some of us already knew how to implement Swing. Also, the Gantt chart produced using this code needs some refinement in terms of visualization.
- **Error handling and exception management**: It's important to handle errors and exceptions properly to provide meaningful error messages and prevent the program from crashing. We encountered exceptions like *NumberFormatException* or *IOException*, and it was essential to handle them gracefully.
- **Testing and debugging**: As with any programming project, testing and debugging are crucial. We needed to thoroughly test the code to ensure that it produces the expected results in various scenarios. Identifying and fixing any bugs or logical errors were time-consuming and challenging. As an example, in earlier stages of testing and debugging, our code was providing the same output for both preemptive and non-preemptive algorithms, but later we rectified the semantic errors and got the correct outputs for the respective algorithms.

11 Conclusion

In conclusion, the Project implements the Shortest Job First (SJF) scheduling algorithm in both preemptive and non-preemptive versions.
The program allows the user to input the number of processes, their arrival time, and burst time.
It then calculates and displays the completion time, turnaround time, and waiting time for each process using the respective SJF algorithm.

The non-preemptive SJF algorithm selects the process with the shortest burst time among the available processes and executes it until completion,

while the preemptive SJF algorithm allows the processes to be interrupted and executed in smaller time slices based on their burst time.

The Project utilizes the Java Swing library to visualize the scheduling results by drawing a Gantt chart.
The Gantt chart represents the execution order and duration of each process, providing a visual representation of the scheduling algorithm's behavior.

Overall, this Project provides a practical implementation of the SJF scheduling algorithm and demonstrates its effectiveness in minimizing turnaround time and waiting time for processes with shorter burst times.
Additionally, the code does not handle scenarios such as process priorities or resource allocation, which could further enhance the scheduling algorithm's functionality.

12 References

The following are a few References of this project:

- **Processes and Threads**: Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson.

  Chapter 2: Processes and Threads

  This chapter provides an overview of processes and threads in operating systems.

- **Process Scheduling**: Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.

  Chapter 5: Process Scheduling

  The fifth chapter covers process scheduling in operating systems, including various scheduling algorithms.

- **Shortest Job First (SJF) Scheduling** Abraham, S. (2017). Operating System Concepts: Shortest Job First (SJF) Scheduling. [Blog post]. Retrieved from https://www.guru99.com/shortest-job-first-sjf-scheduling.html

  This blog post explains the Shortest Job First (SJF) scheduling algorithm, which aims to minimize average waiting time by selecting the process with the shortest burst time first.

- **Scheduling: Introduction** Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson.

Chapter 9: Scheduling: Introduction In this chapter, the author introduces the concept of scheduling in operating systems and discusses different scheduling algorithms.

- **CPU Scheduling Algorithms** Mandaviya, P. (2020). CPU Scheduling Algorithms with Examples: SJF, SRTF, Priority, RR. [Blog post]. Retrieved from https://www.guru99.com/cpu-scheduling-algorithms.html This blog post provides an overview of various CPU scheduling algorithms, including SJF, SRTF, Priority, and Round Robin.
- **Shortest Job First Scheduling** Garg, R. (2013). Operating System Concepts: Shortest Job First Scheduling.
  Retrieved from https://www.geeksforgeeks.org/shortest-job-first-scheduling.
  This blog post specifically focuses on the Shortest Job First (SJF) scheduling algorithm and provides explanations and examples.