**PROCESS SCHEDULER SIMULATION**

A PROJECT REPORT

*Submitted by*

**SALONI BHARDWAJ [Reg No: RA2211003010268]   
SHOVIK BANERJEE [Reg No: RA2211003010270]**

**ARCHISMAN HES [Reg No: RA2211003010273]**

*Under the Guidance of*

**DR. M. ELIAZER**

Assistant Professor(Sr. Grade), Department of Computing Technology

*In partial fulfilment of the requirements for the degree of* **BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE AND ENGINEERING**

****

**DEPARTMENT OF COMPUTING TECHNOLOGIES**

**COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR – 603 203**

**NOVEMBER 2023**



**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603 203**

**BONAFIDE CERTIFICATE**

Certified that this B.Tech project report titled “**PROCESS SCHEDULER SIMULATION**” is the bonafide work of Saloni Bhardwaj [Reg. No.: RA2211003010268], Shovik Banerjee [Reg. No.RA2211003010270], Archisman Hes[Reg. No.RA2211003010273] and who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

**DR. M.ELIAZER**

Assistant Professor

Department of Computing technology

SRM Institute of Science and Technology

Kattankulathur

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| ***S.No.*** | ***TITLE*** | ***PAGE NO.*** |
| ***1*** | ***Abstract*** | ***4*** |
| ***2*** | ***Problem Statement*** | ***6*** |
| ***3*** | ***Scheduling and its types*** | ***7*** |
| ***4*** | ***Code*** | ***11*** |
| ***5*** | ***Output*** | ***24*** |
| ***6*** | ***Conclusion*** | ***30*** |

**ABSTRACT**

A Process Scheduler Simulation serves as a vital instrument in comprehending the intricate mechanisms of how an operating system allocates CPU time to processes. Through its emulation, it unveils valuable insights into the behavior and performance of diverse scheduling algorithms employed in the management of processes. This comprehensive simulation involves several integral components. First and foremost, it requires the establishment of data structures that serve as the foundation for process management and scheduling. These data structures are instrumental in maintaining and organizing crucial information about processes, such as their priority, execution time, and current state. The heart of the simulation lies in the selection of a scheduling algorithm. Various algorithms, including First-Come-First-Serve (FCFS) and Shortest Job First (SJF), can be chosen based on the specific objectives and requirements of the system being studied. The selected algorithm guides the order in which processes are granted access to the CPU, significantly influencing system performance. Generating test cases is another essential aspect of the simulation. These test cases are designed to mimic real-world scenarios, allowing researchers to evaluate how the chosen scheduling algorithm performs under different conditions. They encompass a range of factors such as process arrival times, burst times, and priorities, enabling a comprehensive assessment. The implementation of the selected algorithm within the simulation framework is the next critical step. This involves coding the algorithm to accurately mimic its behavior in a real operating system. The simulation then runs these processes, simulating their execution on a virtual CPU. Finally, the simulation provides a platform for in-depth analysis of the results. Researchers can evaluate the system's performance by examining various metrics such as throughput, turnaround time, and waiting time. This analysis not only helps in assessing the effectiveness of the scheduling algorithm but also aids in understanding how resource utilization can be optimized. In summary, a Process Scheduler Simulation is an indispensable tool for studying the allocation of CPU time to processes. By defining data structures, selecting and implementing scheduling algorithms, generating test cases, and analyzing results, it contributes to a profound understanding of system performance and resource utilization optimization in various operating system environments..

**PROBLEM STATEMENT**

"Inefficient process scheduling within an operating system is a critical issue that hampers system performance and resource utilization. In today's dynamic computing environments, diverse workloads with varying priorities and execution times must be managed effectively. However, existing scheduling algorithms may not adequately address these challenges. The problem at hand is to design, implement, and evaluate a novel process scheduler that optimizes CPU time allocation for processes, considering factors like real-time constraints, multitasking, and resource contention. The scheduler should aim to minimize turnaround time, waiting time, and CPU idleness, ultimately enhancing system responsiveness and throughput. This problem requires a thorough investigation of existing scheduling algorithms and the development of innovative strategies to address the complex scheduling requirements of modern computing systems. The solution should balance the conflicting goals of fairness, efficiency, and responsiveness, while ensuring that high-priority and real-time tasks are prioritized. Additionally, the proposed scheduler should be evaluated through comprehensive simulations and real-world testing to ensure its effectiveness in various computing scenarios. The outcome of this research will lead to improved system performance, resource utilization, and user satisfaction in diverse computing environments."

**SCHEDULING AND IT’S TYPES**

Scheduling, in the context of computer science and operating systems, refers to the process of determining the order in which tasks, processes, or jobs are executed by a computer or other computing systems. It involves the allocation of system resources, primarily the central processing unit (CPU), to various tasks to ensure efficient utilization and effective management of these resources. Scheduling algorithms aim to prioritize and execute tasks in a manner that optimizes factors such as system responsiveness, throughput, fairness, and resource utilization. Different scheduling algorithms are employed based on the specific goals and constraints of the computing system, and they determine which task gets access to the CPU and for how long. Scheduling plays a vital role in ensuring the effective operation of multitasking, multi-user systems and is crucial in real-time computing applications, where meeting specific timing constraints is essential.

1. **First-Come-First-Serve (FCFS) scheduling** is one of the simplest and most straightforward scheduling algorithms used in operating systems. In FCFS scheduling, processes are executed in the order they arrive in the ready queue, with the first process to arrive being the first one to receive the CPU and execute.

Key characteristics of FCFS scheduling:

* Non-preemptive: FCFS is a non-preemptive scheduling algorithm, which means that once a process starts executing, it continues until it finishes or voluntarily releases the CPU.
* Simple to implement: FCFS is easy to understand and implement, making it suitable for simple systems.
* Lack of priority consideration: FCFS does not consider the priority of processes or their execution times. It simply follows the order in which processes arrive.
* Can lead to the "convoy effect": FCFS can result in the convoy effect, where a long process holding the CPU can cause shorter processes to wait, leading to inefficient resource utilization.
* Poor for interactive systems: FCFS is not well-suited for interactive or real-time systems where responsiveness is critical because it may result in long response times

1. **Shortest Job First (SJF) scheduling** is a CPU scheduling algorithm used in operating systems. It aims to minimize the average waiting time of processes by giving preference to the shortest job (process) in the ready queue. SJF is also known as Shortest Job Next (SJN) or Shortest Job First-Come-First-Served (SJF-FCFS).

Key characteristics of SJF scheduling:

* Preemptive and non-preemptive: SJF can be implemented in both preemptive and non-preemptive versions. In the preemptive version, a new job can interrupt the execution of the currently running job if a shorter job becomes available. In the non-preemptive version, the running job is not interrupted until it completes its execution.
* Dynamic priority: SJF essentially assigns priorities based on the expected burst time (time needed for a process to complete). The shorter the burst time, the higher the priority.
* Minimizes waiting time: SJF scheduling is designed to minimize the waiting time, which is the time a process spends in the ready queue before it gets CPU time.
* Potential for starvation: While SJF scheduling is efficient in terms of minimizing waiting times, it can lead to the starvation of longer jobs if a continuous stream of short jobs keeps arriving.

1. **Priority Scheduling - Non-Preemptive (PS-NP)** is a CPU scheduling algorithm used in operating systems. It assigns a priority value to each process, and the process with the highest priority is selected to run first. Unlike the preemptive version of priority scheduling, in non-preemptive priority scheduling, once a process is given the CPU, it continues to execute until it completes or voluntarily releases the CPU.

Key characteristics of Priority Scheduling - Non-Preemptive:

* Priority assignment: Each process is assigned a priority value based on its characteristics, such as its importance or resource requirements. Higher priority values indicate higher priority.
* Process selection: The process with the highest priority in the ready queue is selected to run. If multiple processes share the highest priority, the scheduling algorithm may use additional criteria, like First-Come-First-Serve (FCFS) within the same priority group.
* Lack of time-sharing: In non-preemptive priority scheduling, a process runs to completion without interruption. Only when it completes or enters the waiting state does the CPU become available for other processes with higher priorities.
* Potential for starvation: Lower-priority processes may experience starvation if higher-priority processes continually arrive in the system. To mitigate this, aging mechanisms can be implemented to increase the priority of waiting processes over time.
* Real-time systems: Priority scheduling is commonly used in real-time operating systems where certain tasks must meet strict timing constraints, and tasks are assigned priorities based on their deadlines and criticality.

1. **Priority Scheduling - Preemptive (PS-P)** is a CPU scheduling algorithm used in operating systems. It assigns a priority value to each process, and the process with the highest priority that is ready to run is given the CPU. However, unlike non-preemptive priority scheduling, in PS-P, a process currently executing can be preempted if a higher-priority process becomes available.

Key characteristics of Priority Scheduling - Preemptive:

* + Priority assignment: Each process is assigned a priority value based on various factors, such as the importance of the task, deadline constraints, or resource requirements. Higher priority values represent higher priority.
  + Process selection: The process with the highest priority in the ready queue is allowed to run on the CPU. If a higher-priority process arrives or becomes ready to run, the currently executing process can be preempted and moved back to the ready queue.
  + Time-sharing: Preemptive priority scheduling allows the CPU to be shared among processes more fairly and responsively. High-priority tasks can start promptly, and lower-priority tasks can still be executed.
* Avoiding starvation: Preemptive priority scheduling can help prevent starvation of lower-priority processes since they are given the opportunity to execute when the CPU becomes available.
* Real-time systems: PS-P is often used in real-time operating systems, particularly when tasks need to meet strict deadlines and the system must ensure that high-priority tasks are executed without delay.

1. **Round Robin Scheduling (RR)** is a widely used CPU scheduling algorithm in operating systems. It is designed to provide fair access to the CPU and ensure that no process monopolizes it for an extended period. In Round Robin scheduling, each process is assigned a fixed time quantum, and processes take turns executing for their allocated time, moving in a circular queue.

Key characteristics of Round Robin Scheduling:

* + Time quantum: The most defining feature of Round Robin scheduling is the time quantum or time slice, which is a fixed amount of time allocated to each process. When a process's turn arrives, it is allowed to run for the time quantum or until it voluntarily yields the CPU (e.g., due to I/O operations).
  + Preemptive: Round Robin is a preemptive scheduling algorithm, which means that if a process does not complete its execution within the time quantum, it is temporarily removed from the CPU, and the next process in the queue is given a chance to execute. The interrupted process is placed back at the end of the queue.
  + Fairness: Round Robin aims to provide fair access to the CPU for all processes. Because of the fixed time quantum, no process can monopolize the CPU for an extended period, ensuring that all processes get a chance to execute.
  + Response time: Round Robin provides good response times for interactive processes since they get a chance to execute frequently. However, it may not be as efficient as other scheduling algorithms for long-running CPU-bound tasks.

**PROGRAM CODE**

* 1. **GUI :**

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.Graphics;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.List;

import javax.swing.JButton;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTable;

import javax.swing.table.DefaultTableModel;

public class GUI

{

private JFrame frame;

private JPanel mainPanel;

private CustomPanel chartPanel;

private JScrollPane tablePane;

private JScrollPane chartPane;

private JTable table;

private JButton addBtn;

private JButton removeBtn;

private JButton computeBtn;

private JLabel wtLabel;

private JLabel wtResultLabel;

private JLabel tatLabel;

private JLabel tatResultLabel;

private JComboBox option;

private DefaultTableModel model;

public GUI()

{

model = new DefaultTableModel(new String[]{"Process", "AT", "BT", "Priority", "WT", "TAT"}, 0);

table = new JTable(model);

table.setFillsViewportHeight(true);

tablePane = new JScrollPane(table);

tablePane.setBounds(25, 25, 450, 250);

addBtn = new JButton("Add");

addBtn.setBounds(300, 280, 85, 25);

addBtn.setFont(new Font("Segoe UI", Font.PLAIN, 11));

addBtn.addActionListener(new ActionListener(){

@Override

public void actionPerformed(ActionEvent e) {

model.addRow(new String[]{"", "", "", "", "", ""});

}

});

removeBtn = new JButton("Remove");

removeBtn.setBounds(390, 280, 85, 25);

removeBtn.setFont(new Font("Segoe UI", Font.PLAIN, 11));

removeBtn.addActionListener(new ActionListener(){

@Override

public void actionPerformed(ActionEvent e) {

int row = table.getSelectedRow();

if (row > -1) {

model.removeRow(row);

}

}

});

chartPanel = new CustomPanel();

// chartPanel.setPreferredSize(new Dimension(700, 10));

chartPanel.setBackground(Color.WHITE);

chartPane = new JScrollPane(chartPanel);

chartPane.setBounds(25, 310, 450, 100);

wtLabel = new JLabel("Average Waiting Time:");

wtLabel.setBounds(25, 425, 180, 25);

tatLabel = new JLabel("Average Turn Around Time:");

tatLabel.setBounds(25, 450, 180, 25);

wtResultLabel = new JLabel();

wtResultLabel.setBounds(215, 425, 180, 25);

tatResultLabel = new JLabel();

tatResultLabel.setBounds(215, 450, 180, 25);

option = new JComboBox(new String[]{"FCFS", "SJF", "PS-NP", "PS-P", "RR"});

option.setBounds(390, 420, 85, 20);

computeBtn = new JButton("Compute");

computeBtn.setBounds(390, 450, 85, 25);

computeBtn.setFont(new Font("Segoe UI", Font.PLAIN, 11));

computeBtn.addActionListener(new ActionListener(){

@Override

public void actionPerformed(ActionEvent e) {

String selected = (String) option.getSelectedItem();

CPUScheduler scheduler;

switch (selected) {

case "FCFS":

scheduler = new FirstComeFirstServe();

break;

case "SJF":

scheduler = new ShortestJobFirst();

break;

case "PS-NP":

scheduler = new PriorityNonPreemptive();

break;

case "PS-P":

scheduler = new PriorityPreemptive();

break;

case "RR":

String tq = JOptionPane.showInputDialog("Time Quantum");

if (tq == null) {

return;

}

scheduler = new RoundRobin();

scheduler.setTimeQuantum(Integer.parseInt(tq));

break;

default:

return;

}

for (int i = 0; i < model.getRowCount(); i++)

{

String process = (String) model.getValueAt(i, 0);

int at = Integer.parseInt((String) model.getValueAt(i, 1));

int bt = Integer.parseInt((String) model.getValueAt(i, 2));

int pl;

if (selected.equals("PS-NP") || selected.equals("PS-P"))

{

if (!model.getValueAt(i, 3).equals(""))

{

pl = Integer.parseInt((String) model.getValueAt(i, 3));

}

else

{

pl = 1;

}

}

else

{

pl = 1;

}

scheduler.add(new Row(process, at, bt, pl));

}

scheduler.process();

for (int i = 0; i < model.getRowCount(); i++)

{

String process = (String) model.getValueAt(i, 0);

Row row = scheduler.getRow(process);

model.setValueAt(row.getWaitingTime(), i, 4);

model.setValueAt(row.getTurnaroundTime(), i, 5);

}

wtResultLabel.setText(Double.toString(scheduler.getAverageWaitingTime()));

tatResultLabel.setText(Double.toString(scheduler.getAverageTurnAroundTime()));

chartPanel.setTimeline(scheduler.getTimeline());

}

});

mainPanel = new JPanel(null);

mainPanel.setPreferredSize(new Dimension(500, 500));

mainPanel.add(tablePane);

mainPanel.add(addBtn);

mainPanel.add(removeBtn);

mainPanel.add(chartPane);

mainPanel.add(wtLabel);

mainPanel.add(tatLabel);

mainPanel.add(wtResultLabel);

mainPanel.add(tatResultLabel);

mainPanel.add(option);

mainPanel.add(computeBtn);

frame = new JFrame("CPU Scheduler Simulator");

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

frame.setResizable(false);

frame.add(mainPanel);

frame.pack();

}

public static void main(String[] args)

{

new GUI();

}

class CustomPanel extends JPanel

{

private List<Event> timeline;

@Override

protected void paintComponent(Graphics g)

{

super.paintComponent(g);

if (timeline != null)

{

// int width = 30;

for (int i = 0; i < timeline.size(); i++)

{

Event event = timeline.get(i);

int x = 30 \* (i + 1);

int y = 20;

g.drawRect(x, y, 30, 30);

g.setFont(new Font("Segoe UI", Font.BOLD, 13));

g.drawString(event.getProcessName(), x + 10, y + 20);

g.setFont(new Font("Segoe UI", Font.PLAIN, 11));

g.drawString(Integer.toString(event.getStartTime()), x - 5, y + 45);

if (i == timeline.size() - 1)

{

g.drawString(Integer.toString(event.getFinishTime()), x + 27, y + 45);

}

// width += 30;

}

// this.setPreferredSize(new Dimension(width, 75));

}

}

public void setTimeline(List<Event> timeline)

{

this.timeline = timeline;

repaint();

}

}

}

* 1. **FCFS Scheduling:**

import java.util.Collections;

import java.util.List;

public class FirstComeFirstServe extends CPUScheduler

{

@Override

public void process()

{

Collections.sort(this.getRows(), (Object o1, Object o2) -> {

if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime())

{

return 0;

}

else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime())

{

return -1;

}

else

{

return 1;

}

});

List<Event> timeline = this.getTimeline();

for (Row row : this.getRows())

{

if (timeline.isEmpty())

{

timeline.add(new Event(row.getProcessName(), row.getArrivalTime(), row.getArrivalTime() + row.getBurstTime()));

}

else

{

Event event = timeline.get(timeline.size() - 1);

timeline.add(new Event(row.getProcessName(), event.getFinishTime(), event.getFinishTime() + row.getBurstTime()));

}

}

for (Row row : this.getRows())

{

row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());

row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());

}

}

}

* 1. **Priority Non Pre-Emptive:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class PriorityNonPreemptive extends CPUScheduler

{

@Override

public void process()

{

Collections.sort(this.getRows(), (Object o1, Object o2) -> {

if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime())

{

return 0;

}

else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime())

{

return -1;

}

else

{

return 1;

}

});

List<Row> rows = Utility.deepCopy(this.getRows());

int time = rows.get(0).getArrivalTime();

while (!rows.isEmpty())

{

List<Row> availableRows = new ArrayList();

for (Row row : rows)

{

if (row.getArrivalTime() <= time)

{

availableRows.add(row);

}

}

Collections.sort(availableRows, (Object o1, Object o2) -> {

if (((Row) o1).getPriorityLevel()== ((Row) o2).getPriorityLevel())

{

return 0;

}

else if (((Row) o1).getPriorityLevel() < ((Row) o2).getPriorityLevel())

{

return -1;

}

else

{

return 1;

}

});

Row row = availableRows.get(0);

this.getTimeline().add(new Event(row.getProcessName(), time, time + row.getBurstTime()));

time += row.getBurstTime();

for (int i = 0; i < rows.size(); i++)

{

if (rows.get(i).getProcessName().equals(row.getProcessName()))

{

rows.remove(i);

break;

}

}

}

for (Row row : this.getRows())

{

row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());

row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());

}

}

}

* 1. **Priority Pre-Emptive:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class PriorityPreemptive extends CPUScheduler

{

@Override

public void process()

{

Collections.sort(this.getRows(), (Object o1, Object o2) -> {

if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime())

{

return 0;

}

else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime())

{

return -1;

}

else

{

return 1;

}

});

List<Row> rows = Utility.deepCopy(this.getRows());

int time = rows.get(0).getArrivalTime();

while (!rows.isEmpty())

{

List<Row> availableRows = new ArrayList();

for (Row row : rows)

{

if (row.getArrivalTime() <= time)

{

availableRows.add(row);

}

}

Collections.sort(availableRows, (Object o1, Object o2) -> {

if (((Row) o1).getPriorityLevel()== ((Row) o2).getPriorityLevel())

{

return 0;

}

else if (((Row) o1).getPriorityLevel() < ((Row) o2).getPriorityLevel())

{

return -1;

}

else

{

return 1;

}

});

Row row = availableRows.get(0);

this.getTimeline().add(new Event(row.getProcessName(), time, ++time));

row.setBurstTime(row.getBurstTime() - 1);

if (row.getBurstTime() == 0)

{

for (int i = 0; i < rows.size(); i++)

{

if (rows.get(i).getProcessName().equals(row.getProcessName()))

{

rows.remove(i);

break;

}

}

}

}

for (int i = this.getTimeline().size() - 1; i > 0; i--)

{

List<Event> timeline = this.getTimeline();

if (timeline.get(i - 1).getProcessName().equals(timeline.get(i).getProcessName()))

{

timeline.get(i - 1).setFinishTime(timeline.get(i).getFinishTime());

timeline.remove(i);

}

}

Map map = new HashMap();

for (Row row : this.getRows())

{

map.clear();

for (Event event : this.getTimeline())

{

if (event.getProcessName().equals(row.getProcessName()))

{

if (map.containsKey(event.getProcessName()))

{

int w = event.getStartTime() - (int) map.get(event.getProcessName());

row.setWaitingTime(row.getWaitingTime() + w);

}

else

{

row.setWaitingTime(event.getStartTime() - row.getArrivalTime());

}

map.put(event.getProcessName(), event.getFinishTime());

}

}

row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());

}

}

}

* 1. **Round Robin Scheduling:**

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class RoundRobin extends CPUScheduler

{

@Override

public void process()

{

Collections.sort(this.getRows(), (Object o1, Object o2) -> {

if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime())

{

return 0;

}

else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime())

{

return -1;

}

else

{

return 1;

}

});

List<Row> rows = Utility.deepCopy(this.getRows());

int time = rows.get(0).getArrivalTime();

int timeQuantum = this.getTimeQuantum();

while (!rows.isEmpty())

{

Row row = rows.get(0);

int bt = (row.getBurstTime() < timeQuantum ? row.getBurstTime() : timeQuantum);

this.getTimeline().add(new Event(row.getProcessName(), time, time + bt));

time += bt;

rows.remove(0);

if (row.getBurstTime() > timeQuantum)

{

row.setBurstTime(row.getBurstTime() - timeQuantum);

for (int i = 0; i < rows.size(); i++)

{

if (rows.get(i).getArrivalTime() > time)

{

rows.add(i, row);

break;

}

else if (i == rows.size() - 1)

{

rows.add(row);

break;

}

}

}

}

Map map = new HashMap();

for (Row row : this.getRows())

{

map.clear();

for (Event event : this.getTimeline())

{

if (event.getProcessName().equals(row.getProcessName()))

{

if (map.containsKey(event.getProcessName()))

{

int w = event.getStartTime() - (int) map.get(event.getProcessName());

row.setWaitingTime(row.getWaitingTime() + w);

}

else

{

row.setWaitingTime(event.getStartTime() - row.getArrivalTime());

}

map.put(event.getProcessName(), event.getFinishTime());

}

}

row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());

}

}

}

1. **Shortest Job First:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class ShortestJobFirst extends CPUScheduler

{

@Override

public void process()

{

Collections.sort(this.getRows(), (Object o1, Object o2) -> {

if (((Row) o1).getArrivalTime() == ((Row) o2).getArrivalTime())

{

return 0;

}

else if (((Row) o1).getArrivalTime() < ((Row) o2).getArrivalTime())

{

return -1;

}

else

{

return 1;

}

});

List<Row> rows = Utility.deepCopy(this.getRows());

int time = rows.get(0).getArrivalTime();

while (!rows.isEmpty())

{

List<Row> availableRows = new ArrayList();

for (Row row : rows)

{

if (row.getArrivalTime() <= time)

{

availableRows.add(row);

}

}

Collections.sort(availableRows, (Object o1, Object o2) -> {

if (((Row) o1).getBurstTime() == ((Row) o2).getBurstTime())

{

return 0;

}

else if (((Row) o1).getBurstTime() < ((Row) o2).getBurstTime())

{

return -1;

}

else

{

return 1;

}

});

Row row = availableRows.get(0);

this.getTimeline().add(new Event(row.getProcessName(), time, time + row.getBurstTime()));

time += row.getBurstTime();

for (int i = 0; i < rows.size(); i++)

{

if (rows.get(i).getProcessName().equals(row.getProcessName()))

{

rows.remove(i);

break;

}

}

}

for (Row row : this.getRows())

{

row.setWaitingTime(this.getEvent(row).getStartTime() - row.getArrivalTime());

row.setTurnaroundTime(row.getWaitingTime() + row.getBurstTime());

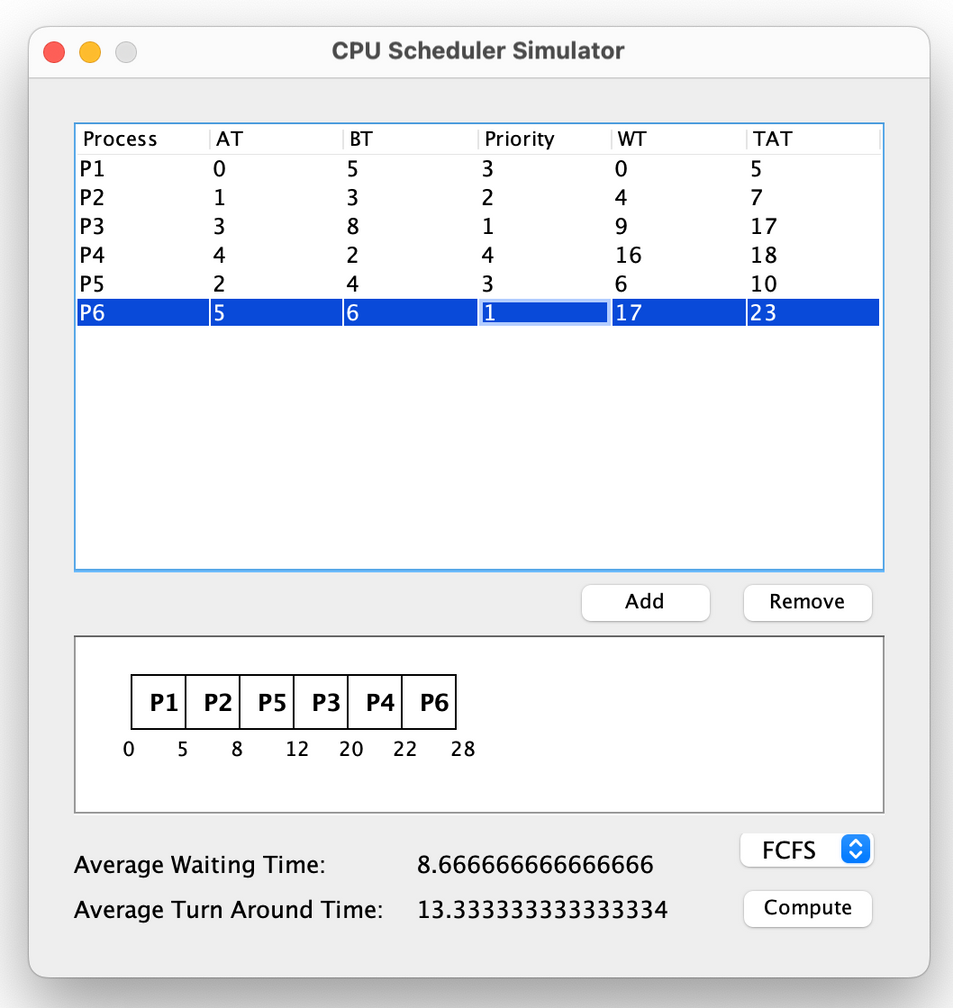
}

}

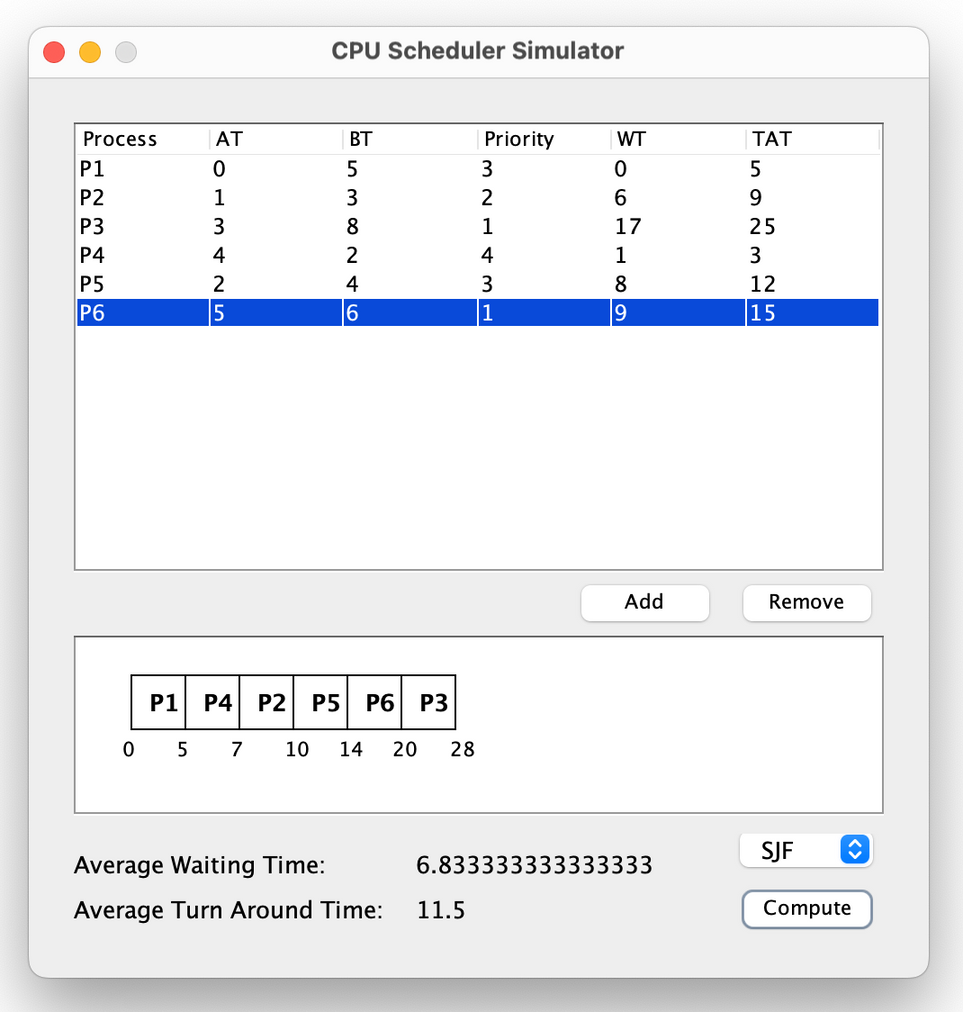
}

**OUTPUT**

1. **FCFS:**



1. **SJF:**

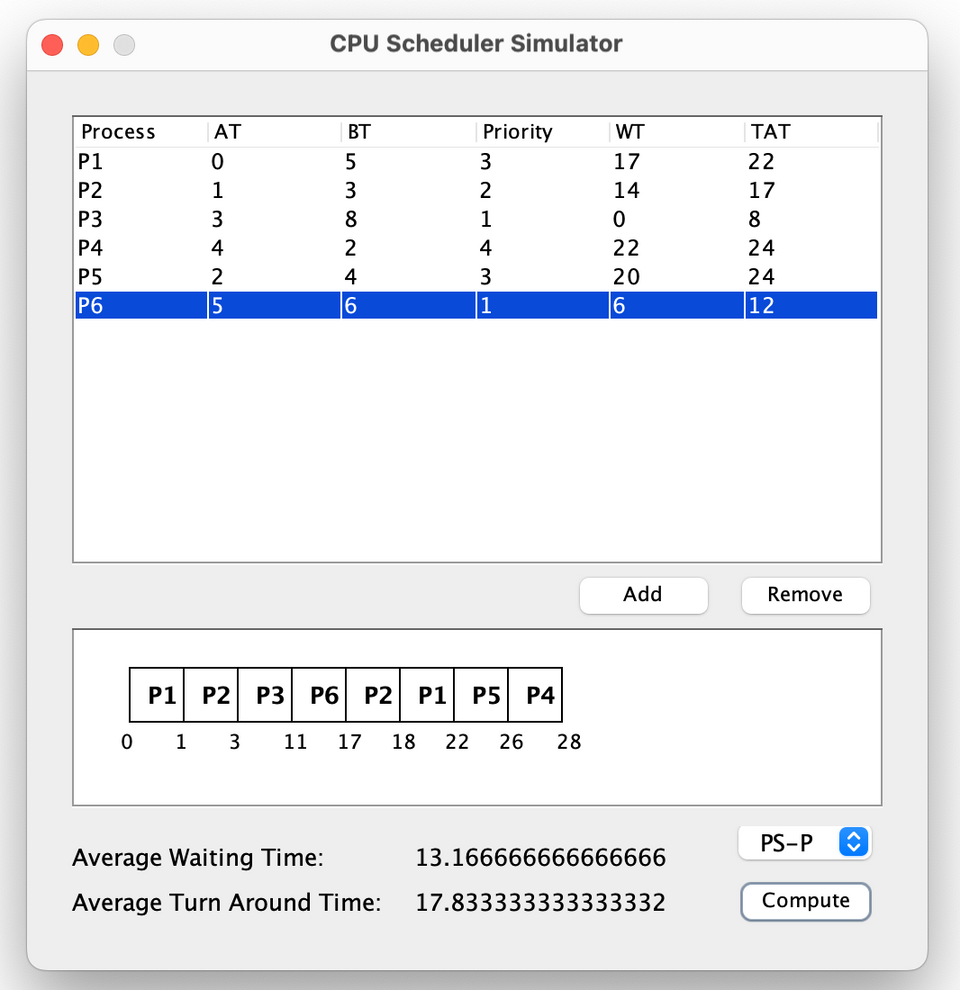
****

1. **Priority Scheduling - Non Preemptive:**

**A screenshot of a computer

Description automatically generated**

1. **Priority Scheduling – Preemptive**

****

1. **Round Robin:**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

**CONCLUSION**

In conclusion, the process scheduler project for operating systems represents a significant milestone in understanding and improving the management of system resources and the execution of processes. Through this project, we have delved into the intricate world of CPU scheduling algorithms, shedding light on their impact on system performance, efficiency, and responsiveness.

The project has equipped us with valuable insights into various scheduling algorithms, such as First-Come-First-Serve (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin. We've explored both non-preemptive and preemptive variants, each with its own unique characteristics and applications. These algorithms play a pivotal role in ensuring that the CPU is allocated efficiently to processes, ultimately influencing a system's ability to handle concurrent tasks and meet the demands of diverse workloads.

Our exploration of scheduling algorithms has highlighted the trade-offs that exist in the realm of process management. While some algorithms prioritize fairness and equitable access to system resources, others aim to optimize throughput and response times, making it crucial to choose the most suitable algorithm for a given context.

Furthermore, our project has emphasized the importance of accurate burst time estimation, particularly in preemptive scheduling, as well as the need for effective priority management in priority-based scheduling. We have also recognized the role of time quantum in the Round Robin algorithm, where the choice of this parameter can significantly impact the overall system performance.

As we conclude this project, we acknowledge that the world of process scheduling is not static. Advances in computing technology, evolving system requirements, and changing user expectations continue to shape the landscape of scheduling algorithms. Therefore, our project is not just a culmination but also a stepping stone to further research and innovation in the field.

In the ever-evolving domain of operating systems, the knowledge and experience gained from this project will be invaluable. It equips us to make informed decisions when selecting scheduling algorithms, optimizing their parameters, and addressing the dynamic needs of modern computing environments. The insights gathered during this project are essential not only for understanding the past and present but also for shaping the future of process scheduling in operating systems.