



GROUP PROJECT

GROUP MEMBERS:

- SALMAN KHAN - 53095
- MUSTAFA JAMIL - 53340
- TARIQ YASEEN - 52482
- SIKANDER MEHDI – 51976
- SYED HAMZA JAFFAR – 5231
- KASHIF ALI - 50420

COURSE : OPERATING SYSTEM LAB

FACULTY : ANEES AHMED

TIMING : TUESDAY (11:45 – 2:45)

CODE:

Class 1

```
import java.util.ArrayList;
import java.util.List;

public abstract class CPUScheduler{

    private final List<Row> rows;
    private final List<Event> timeline;
    private int timeQuantum;

    public CPUScheduler(){
        rows = new ArrayList();
        timeline = new ArrayList();
        timeQuantum = 1;}

    public boolean add(Row row)
    {return rows.add(row);}

    public void setTimeQuantum(int timeQuantum){
        this.timeQuantum = timeQuantum;}

    public int getTimeQuantum() {
        return timeQuantum;}

    public double getAverageWaitingTime() {
        double avg = 0.0;
        for (Row row : rows) {
            avg += row.getWaitingTime();
        }
```

```

        return avg / rows.size();    }
    public double getAverageTurnAroundTime()    {
        double avg = 0.0;
        for (Row row : rows)        {
            avg += row.getTurnaroundTime();        }
        return avg / rows.size();    }
    public Event getEvent(Row row)    {
        for (Event event : timeline)        {
            if (row.getProcessName().equals(event.getProcessName()))        {
                return event;        }        }
        return null;    }
    public Row getRow(String process)    {
        for (Row row : rows)        {
            if (row.getProcessName().equals(process))        {
                return row;        }        }
        return null;
    }

    public List<Row> getRows()
    {
        return rows;
    }
    public List<Event> getTimeline()
    {
        return timeline;
    }
    public abstract void process();
}

```

Class 2

```
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());
}
}
}

```

Class 3

```

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());
}
}
}

```

Class 4:

```

import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        System.out.println("-----FCFS-----");
        fcfs();
        System.out.println("-----SJF-----");
        sjf();
        System.out.println("-----SRT-----");
        srt();
        System.out.println("-----PSN-----");
        psn();
        System.out.println("-----PSP-----");
        psp();
        System.out.println("-----RR-----");
        rr();
    }

    public static void fcfs()
    {
        CPUScheduler fcfs = new FirstComeFirstServe();
        fcfs.add(new Row("P1", 0, 5));
        fcfs.add(new Row("P2", 2, 4));
        fcfs.add(new Row("P3", 4, 3));
        fcfs.add(new Row("P4", 6, 6));
        fcfs.process();
        display(fcfs);
    }

    public static void sjf()
    {
        CPUScheduler sjf = new ShortestJobFirst();
        sjf.add(new Row("P1", 0, 5));
        sjf.add(new Row("P2", 2, 3));
        sjf.add(new Row("P3", 4, 2));
        sjf.add(new Row("P4", 6, 4));
        sjf.add(new Row("P5", 7, 1));
        sjf.process();
        display(sjf);
    }

    public static void srt()
    {
        CPUScheduler srt = new ShortestRemainingTime();
        srt.add(new Row("P1", 8, 1));
        srt.add(new Row("P2", 5, 1));
        srt.add(new Row("P3", 2, 7));
        srt.add(new Row("P4", 4, 3));
        srt.add(new Row("P5", 2, 8));
        srt.add(new Row("P6", 4, 2));
        srt.add(new Row("P7", 3, 5));
        srt.process();
    }
}

```

```

        display(srt);
    }

    public static void psn()
    {
        CPUScheduler psn = new PriorityNonPreemptive();
        psn.add(new Row("P1", 8, 1));
        psn.add(new Row("P2", 5, 1));
        psn.add(new Row("P3", 2, 7));
        psn.add(new Row("P4", 4, 3));
        psn.add(new Row("P5", 2, 8));
        psn.add(new Row("P6", 4, 2));
        psn.add(new Row("P7", 3, 5));
        psn.process();
        display(psn);
    }

    public static void psp()
    {
        CPUScheduler psp = new PriorityPreemptive();
        psp.add(new Row("P1", 8, 1));
        psp.add(new Row("P2", 5, 1));
        psp.add(new Row("P3", 2, 7));
        psp.add(new Row("P4", 4, 3));
        psp.add(new Row("P5", 2, 8));
        psp.add(new Row("P6", 4, 2));
        psp.add(new Row("P7", 3, 5));
        psp.process();
        display(psp);
    }

    public static void rr()
    {
        CPUScheduler rr = new RoundRobin();
        rr.setTimeQuantum(2);
        rr.add(new Row("P1", 0, 4));
        rr.add(new Row("P2", 1, 5));
        rr.add(new Row("P3", 2, 6));
        rr.add(new Row("P4", 4, 1));
        rr.add(new Row("P5", 6, 3));
        rr.add(new Row("P6", 7, 2));
        rr.process();
        display(rr);
    }

    public static void display(CPUScheduler object)
    {
        System.out.println("Process\tAT\tBT\tWT\tTAT");

        for (Row row : object.getRows())
        {
            System.out.println(row.getProcessName() + "\t" + row.getArrivalTime() +
                "\t" + row.getBurstTime() + "\t" + row.getWaitingTime() + "\t" +
                row.getTurnaroundTime());
        }
    }

```

```

        System.out.println();

        for (int i = 0; i < object.getTimeline().size(); i++)
        {
            List<Event> timeline = object.getTimeline();
            System.out.print(timeline.get(i).getStartTime() + "(" +
            timeline.get(i).getProcessName() + ")");

            if (i == object.getTimeline().size() - 1)
            {
                System.out.print(timeline.get(i).getFinishTime());
            }

            System.out.println("\n\nAverage WT: " + object.getAverageWaitingTime() +
            "\n\nAverage TAT: " + object.getAverageTurnAroundTime());
        }
    }
}

```

Class 5:

```

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", "SRT", "PSN", "PSP", "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 "SRT":
```

```
                scheduler = new ShortestRemainingTime();
```

```
                break;
```

```
            case "PSN":
```

```
                scheduler = new PriorityNonPreemptive();
```

```
                break;
```

```
            case "PSP":
```

```
                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("PSN") || selected.equals("PSP"))
    {
        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();
    }
}

```

Class 6:

```

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());
}
}
}

```

Class 7:

```

public class Event
{
    private final String processName;
    private final int startTime;
    private int finishTime;

    public Event(String processName, int startTime, int finishTime)
    {
        this.processName = processName;
        this.startTime = startTime;
        this.finishTime = finishTime;
    }

    public String getProcessName()
    {
        return processName;
    }

    public int getStartTime()
    {
        return startTime;
    }

    public int getFinishTime()
    {
        return finishTime;
    }

    public void setFinishTime(int finishTime)
    {

```

```

        this.finishTime = finishTime;
    }
}

```

Class 8:

```

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());
}
}

```

Class 9:

```

public class Row
{
    private String processName;
    private int arrivalTime;

```



```

private int burstTime;
private int priorityLevel;
private int waitingTime;
private int turnaroundTime;

private Row(String processName, int arrivalTime, int burstTime, int
priorityLevel, int waitingTime, int turnaroundTime)
{
    this.processName = processName;
    this.arrivalTime = arrivalTime;
    this.burstTime = burstTime;
    this.priorityLevel = priorityLevel;
    this.waitingTime = waitingTime;
    this.turnaroundTime = turnaroundTime;
}

public Row(String processName, int arrivalTime, int burstTime, int priorityLevel)
{
    this(processName, arrivalTime, burstTime, priorityLevel, 0, 0);
}

public Row(String processName, int arrivalTime, int burstTime)
{
    this(processName, arrivalTime, burstTime, 0, 0, 0);
}

public void setBurstTime(int burstTime)
{
    this.burstTime = burstTime;
}

public void setWaitingTime(int waitingTime)
{
    this.waitingTime = waitingTime;
}

public void setTurnaroundTime(int turnaroundTime)
{
    this.turnaroundTime = turnaroundTime;
}

public String getProcessName()
{
    return this.processName;
}

public int getArrivalTime()
{
    return this.arrivalTime;
}

public int getBurstTime()
{
    return this.burstTime;
}

```

```

    public int getPriorityLevel()
    {
        return this.priorityLevel;
    }

    public int getWaitingTime()
    {
        return this.waitingTime;
    }

    public int getTurnaroundTime()
    {
        return this.turnaroundTime;
    }
}

```

Class 10:

```

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());
}
}
}

```

Class 11:

```

import java.util.ArrayList;

import java.util.List;

```

```

public class Utility

```

```

    {
        public static List<Row> deepCopy(List<Row> oldList)
        {
            List<Row> newList = new ArrayList();

            for (Row row : oldList)
            {
                newList.add(new Row(row.getProcessName(), row.getArrivalTime(), row.getBurstTime(),
row.getPriorityLevel()));
            }

            return newList;
        }
    }

```

Class 12:

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ShortestRemainingTime 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.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());

```


```

}

```

```
}  
}
```

OUTPUT:

 CPU Scheduler Simulator

Process	AT	BT	Priority	WT	TAT
A	0	4			
B	1	5			
C	2	6			
D	4	1			
E	6	3			
F	7	2			

Add


Remove

Average Waiting Time:

Average Turn Around Time:

FCFS

Compute

 CPU Scheduler Simulator

Process

AT

BT

Priority

WT

TAT

A

0

4

0

4

B

1

5

3

8

C

2

6

7

13

D

4

1

11

12

E

6

3

10

13

F

7

2

12

14

Add

Remove

A

B

C

D

E

F

0

4

9

15

16

19

21

Average Waiting Time:

7.166666666666667

FCFS

Average Turn Around Time:

10.666666666666666

Compute

CPU Scheduler Simulator

Process	AT	BT	Priority	WT	TAT
A	0	4	5		
B	1	5	3		
C	2	6	1		
D	4	1	4		
E	6	3	2		
F	7	2	1		

Input

?

Time Quantum

2

OK

Cancel

Add

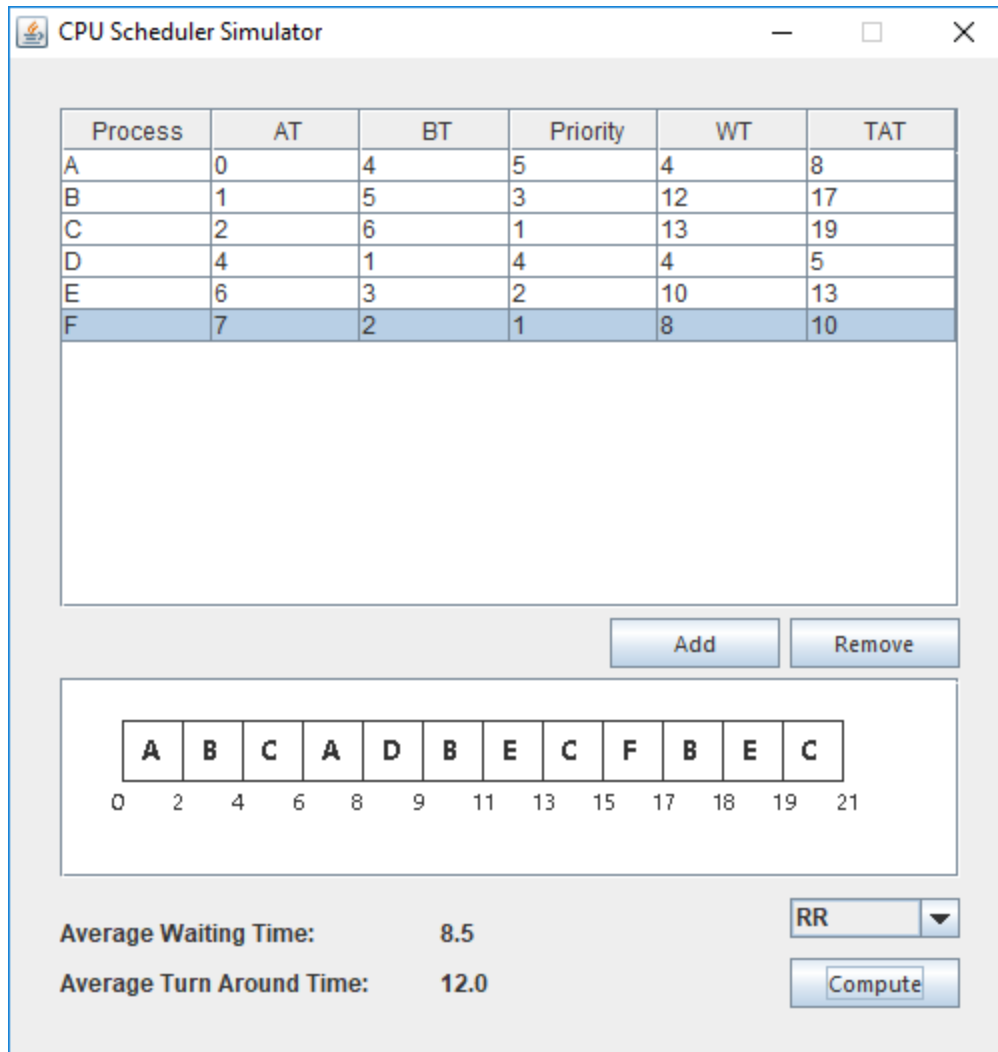
Remove

Average Waiting Time:

RR

Average Turn Around Time:

Compute



ALGORITHMS:

FCFS:

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.

Characteristics of FCFS method

- It supports non-preemptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis.
- It is easy to implement and use.
- This method is poor in performance, and the general wait time is quite high

Example of FCFS scheduling

A real-life example of the FCFS method is buying a movie ticket on the ticket counter. In this scheduling algorithm, a person is served according to the queue manner. The person who arrives first in the queue first buys the ticket and then the next one. This will continue until the last person in the queue purchases the ticket. Using this algorithm, the CPU process works in a similar manner.

Advantages of FCFS

Here, are pros/benefits of using FCFS scheduling algorithm:

- The simplest form of a CPU scheduling algorithm
- Easy to program
- First come first served

Disadvantages of FCFS

Here, are cons/ drawbacks of using FCFS scheduling algorithm:

- It is a Non-Preemptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.
- The Average Waiting Time is high.
- Short processes that are at the back of the queue have to wait for the long process at the front to finish.
- Not an ideal technique for time-sharing systems.
- Because of its simplicity, FCFS is not very efficient.

ROUND ROBIN:

The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

Characteristics of Round-Robin Scheduling

Here are the important characteristics of Round-Robin Scheduling:

- Round robin is a pre-emptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
- The process that is preempted is added to the end of the queue.
- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
- It is a real time algorithm which responds to the event within a specific time limit.
- Round robin is one of the oldest, fairest, and easiest algorithm.
- Widely used scheduling method in traditional OS.

Advantage of Round-robin Scheduling

Here, are pros/benefits of Round-robin scheduling method:

- It doesn't face the issues of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.
- It deals with all process without any priority
- If you know the total number of processes on the run queue, then you can also assume the worst-case response time for the same process.

- This scheduling method does not depend upon burst time. That's why it is easily implementable on the system.
- Once a process is executed for a specific set of the period, the process is preempted, and another process executes for that given time period.
- Allows OS to use the Context switching method to save states of preempted processes.
- It gives the best performance in terms of average response time.

Disadvantages of Round-robin Scheduling

Here, are drawbacks/cons of using Round-robin scheduling:

- If slicing time of OS is low, the processor output will be reduced.
- This method spends more time on context switching
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.
- Round-robin scheduling doesn't give special priority to more important tasks.
- Decreases comprehension
- Lower time quantum results in higher the context switching overhead in the system.
- Finding a correct time quantum is a quite difficult task in this system.

Priority Scheduling:

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

Characteristics of Priority Scheduling

- A CPU algorithm that schedules processes based on priority.
- It used in Operating systems for performing batch processes.
- If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.
- In priority scheduling, a number is assigned to each process that indicates its priority level.
- Lower the number, higher is the priority.
- In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

Advantages of priority scheduling

Here, are benefits/pros of using priority scheduling method:

- Easy to use scheduling method
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time
- This method provides a good mechanism where the relative important of each process may be precisely defined.

- Suitable for applications with fluctuating time and resource requirements.

Disadvantages of priority scheduling

Here, are cons/drawbacks of priority scheduling

- If the system eventually crashes, all low priority processes get lost.
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.
- This scheduling algorithm may leave some low priority processes waiting indefinitely.
- A process will be blocked when it is ready to run but has to wait for the CPU because some other process is running currently.
- If a new higher priority process keeps on coming in the ready queue, then the process which is in the waiting state may need to wait for a long duration of time.

SHORTEST JOB FIRST

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

There are basically two types of SJF methods:

- Non-Preemptive SJF
- Preemptive SJF

Characteristics of SJF Scheduling

- It is associated with each job as a unit of time to complete.
- This algorithm method is helpful for batch-type processing, where waiting for jobs to complete is not critical.
- It can improve process throughput by making sure that shorter jobs are executed first, hence possibly have a short turnaround time.

- It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time.

Non-Preemptive SJF

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

Preemptive SJF

In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Advantages of SJF

Here are the benefits/pros of using SJF method:

- SJF is frequently used for long term scheduling.
- It reduces the average waiting time over FIFO (First in First Out) algorithm.
- SJF method gives the lowest average waiting time for a specific set of processes.
- It is appropriate for the jobs running in batch, where run times are known in advance.
- For the batch system of long-term scheduling, a burst time estimate can be obtained from the job description.
- For Short-Term Scheduling, we need to predict the value of the next burst time.
- Probably optimal with regard to average turnaround time.

Disadvantages/Cons of SJF

Here are some drawbacks/cons of SJF algorithm:

- Job completion time must be known earlier, but it is hard to predict.
- It is often used in a batch system for long term scheduling.

- SJF can't be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst.
- This algorithm may cause very long turnaround times or starvation.
- Requires knowledge of how long a process or job will run.
- It leads to the starvation that does not reduce average turnaround time.
- It is hard to know the length of the upcoming CPU request.
- Elapsed time should be recorded, that results in more overhead on the processor.